GEOSX Documentation

Randolph Settgast

Nov 03, 2023

CONTENTS

GEOS is a code framework focused on enabling streamlined development of physics simulations on high performance computing platforms. Our documentation is organized into several separate guides, given that different users will have different needs.

We recommend all users begin with the Quick Start guide, which will walk you through downloading and compiling the code. Application focused users may then want to explore our Tutorials, which provide an introduction to the basic capabilities of the code. More detailed descriptions of these capabilities can then be found in the User Guide.

For those interested in developing new capabilities in GEOS, we provide a Developer Guide. The code itself is also documented inline using doxygen. The Build Guide contains more detailed information about third-party dependencies, the build system, and the continuous integration system. Finally, GEOS has a self-documenting data structure. The Datastructure Index is an automatically generated list of all available input parameters and data structures in the code. This is a comprehensive resource, but probably not the place to start.

High quality documentation is a critical component of a successful code. If you have suggestions for improving the guides below, please post an issue on our issue tracker.

Quick Start Guide

New to GEOS? We will walk you through downloading the source, compiling the code, and testing the installation.

To the Quick Start

Tutorials

Working tutorials that show how to run some common problems. After going through these examples, you should have a good understanding of how to set up and solve your own models.

To the Tutorials

Basic Examples

Example problems that are organized around physical processes (fluid flow, mechanics, etc.).

To the Basic Examples

Advanced Examples

Example problems that demonstrate additional physical models, constitutive models, advanced features, etc.

To the Advanced Examples

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Quick Start Guide

The goal of this page is to get you started as quickly as possible using GEOS. We will walk you through downloading the source, compiling the code, and testing the installation.

Before jumping to the installation process, we want to first address some frequently asked questions we get from new users. If you are itching to get started, feel free to jump ahead to the relevant sections.

1.1.1 Frequently Asked Questions

Does GEOS have a graphical user interface?:

Given the focus on rapid development and HPC environments, GEOS does not have a graphical user interface. This is consistent with many other high performance computing packages, but we recognize it can be a deal-breaker for certain users. For those who can get past this failing, we promise we still have a lot to offer. In a typical workflow, you will prepare an XML-based input file describing your problem. You may also prepare a mesh file containing geometric and property information describing, say, a reservoir you would like to simulate. There is no shortage of GUI tools that can help you in this model building stage. The resulting input deck is then consumed by GEOS to run the simulation and produce results. This may be done in a terminal of your local machine or by submitting a job to a remote server. The resulting output files can then be visualized by any number of graphical visualization programs (typically VisIt or paraview). Thus, while GEOS is GUI free, the typical workflow is not.

Do I need to be a code developer to use GEOS?:

For the moment, most users will need to download and compile the code from source, which we readily admit this requires a certain level of development expertise. We try to make this process as easy as possible, and we are working on additional deployment options to make this process easier. Once installed, however, our goal is to make GEOS accessible to developers and non-developers alike. Our target audience includes engineers and scientists who want to solve tough application problems, but could care less about the insides of the tool. For those of you who *are* interested in scientific computing, however, GEOS is an open source project and we welcome external contributions.

What are the system requirements?:

GEOS is primarily written in C++, with a focus on standards compliance and platform-to-platform portability. It is designed to run on everything from commodity laptops to the world's most powerful supercomputers. We regularly test the code across a variety of operating systems and compilers. Most of these operating systems are Linux/UNIX based (e.g. Ubuntu, CentOS, Mac OSX). We do have developers working in Windows environments, but they use a Virtual Machine or work within a docker image rather than directly in the Windows environment. In the instructions below, we assume you have access to fairly standard development tools. Using advanced features of GEOS, like GPU-acceleration, will of course introduce additional hardware and software requirements.

Help, I get errors while trying to download/compile/run!:

Unfortunately, no set of instructions is foolproof. It is simply impossible to anticipate every system configuration or user. If you run into problems during the installation, we recommend the following five-step process:

1. Take a moment to relax, and then re-read the instructions carefully. Perhaps you overlooked a key step? Re-read the error message(s) closely. Modern compilation tools are often quite helpful in reporting exactly why things fail.
2. Type a few keywords from your error into a search engine. It is possible someone else out there has encountered your problem before, and a well-chosen keyword can often produce an instant solution. Note that when a compilation fails, you may get pages and pages of errors. Try to identify the *first* one to occur and fix that. One error will often trigger subsequent errors, and looking at the *last* error on the screen may not be so helpful.
3. If you encounter problems building one of the third-party libraries we depend on, check out their support pages. They may be able to help you more directly than we can.
4. Still stuck? Check out our issues tracker, searching current or closed issues that may address your problem. Perhaps someone has had an identical issue, or something close. The issue tracker has a convenient search bar where you can search for relevant keywords. Remember to remove the default `is:open` keyword to search both open and closed issues.
5. If you have exhausted the options above, it is time to seek help from the developers. Post an issue on our issue tracker. Be specific, providing as much information as possible about your system setup and the error you are encountering. Please be patient in this process, as we may need to correspond a few times and ask you to run additional tests. Most of the time, users have a slightly unusual system configuration that we haven't encountered yet, such as an older version of a particular library. Other times there is a legitimate bug in GEOS to be addressed. Take pride in the fact that you may be saving the next user from wasted time and frustration.

1.1.2 Repository Organization

The source for GEOS and related tools are hosted on Github. We use Git workflows to version control our code and manage the entire development process. On Github, we have a GEOS Organization that hosts several related repositories.

You should sign up for a free Github account, particularly if you are interested in posting issues to our issue tracker and communicating with the developers. The main repository of interest is obviously GEOS itself: GEOS

We also rely on two types of dependencies: first-party and third-party. First-party dependencies are projects directly associated with the GEOS effort, but kept in separate repositories because they form stand-alone tools. For example, there is an equation-of-state package called PVTPackage or the streamlined CMake-based foundation **`BTL <https://github.com/LLNL/blt`_**. These packages are handled as Git Submodules, which provides a transparent way of coordinating multiple code development projects. Most users will never have to worry that these modules are in fact separate projects from GEOS.

We also rely on several open-source Third-Party Libraries (TPLs) (see thirdPartyLibs). These are well-respected projects developed externally to GEOS. We have found, however, that many compilation issues stem from version incompatibilities between different packages. To address this, we provide a mirror of these TPLs, with version combinations we know play nicely together. We also provide a build script that conveniently and consistently builds those dependencies.

Our build system will automatically use the mirror package versions by default. You are welcome to tune your configuration, however, to point to different versions installed on your system. If you work on an HPC platform, for example, common packages may already be available and optimized for platform hardware. For new users, however, it may be safer to begin with the TPL mirror.

Note: If you are working on an HPC platform with several other GEOS users, we often compile the TPLs in a shared location so individual users don't have to waste their storage quota. Inquire with your institution's point-of-contact whether this option already exists. For all LLNL systems, the answer is yes.

Finally, there are also several private repositories only accessible to the core development team, which we use for behind-the-scene testing and maintenance of the code.

1.1.3 Username and Authentication

New users should sign up for a free Github account.

If you intend to develop in the GEOS codebase, you may benefit from setting up your git credentials (see *Git Workflow*).

1.1.4 Download

It is possible to directly download the source code as a zip file. We strongly suggest, however, that users don't rely on this option. Instead, most users should use Git to either *clone* or *fork* the repository. This makes it much easier to stay up to date with the latest releases and bug fixes. If you are not familiar with the basics of Git, here is a helpful resource to get you started.

The tutorial here assumes you will use a https clone with no specific credentials. Using an ssh connection pattern requires a very slight modification. See the **Additional Notes** at the end of this section for details.

If you do not already have Git installed on your system, you will need to install it. We recommend using a relatively recent version of Git, as there have been some notable improvements over the past few years. You can check if Git is already available by opening a terminal and typing

```
git --version
```

You'll also need the git-lfs large file extension.

The first task is to clone the `GEOS` and `thirdPartyLibs` repositories. If you do not tell it otherwise, the build system will expect the GEOS and thirdPartyLibs to be parallel to each other in the directory structure. For example,

```
codes/
├── GEOS/
└── thirdPartyLibs/
```

where the toplevel `codes` directory can be re-named and located wherever you like. It is possible to customize the build system to expect a different structure, but for now let us assume you take the simplest approach.

First, using a terminal, create the `codes` directory wherever you like.

```
cd /insert/your/desired/path/
mkdir codes
cd codes
```

Inside this directory, we can clone the GEOS repository. We will also use some Git commands to initialize and download the submodules (e.g. `LvArray`). Note that most users will not have access to our integrated tests repository, and so we "deinit" (deactivate) this submodule. Developers who will be working with the integratedTests repository should skip this line.

```
git clone https://github.com/GEOS-DEV/GEOS.git
cd GEOS
git lfs install
git submodule init
git submodule deinit integratedTests
git submodule update
cd ..
```

If all goes well, you should have a complete copy of the GEOS source at this point. The most common errors people encounter here have to do with Github not recognizing their authentication settings and/or repository permissions. See the previous section for tips on ensuring your SSH is working properly.

Note: The integratedTests submodule is not publicly available, with access limited to the core development team. This may cause the `git submodule update` command to fail if you forget the `git submodule deinit integratedTests` step above. This submodule is not required for building GEOS. If you see an error message here, however, you may need to initialize and update the submodules manually:

```
cd GEOS
git submodule update --init src/cmake/blt
git submodule update --init src/coreComponents/LvArray
git submodule update --init src/coreComponents/fileIO/coupling/hdf5_interface
git submodule update --init src/coreComponents/constitutive/PVTPackage
cd ..
```

Once we have grabbed GEOS, we do the same for the thirdPartyLibs repository. From the `codes` directory, type

```
git clone https://github.com/GEOS-DEV/thirdPartyLibs.git
cd thirdPartyLibs
git lfs install
git pull
git submodule init
git submodule update
cd ..
```

Again, if all goes well you should now have a copy of all necessary TPL packages.

Additional Notes:

#. `git-lfs` may not function properly (or may be very slow) if your version of git and git-lfs are not current. If you are using an older version, you may need to add `git lfs pull` after `git pull` in the above procedures.

#. You can adapt the commands if you use an ssh connection instead. The clone `https://github.com/GEOS-DEV/GEOS.git` becomes `git clone git@github.com:GEOS-DEV/GEOS.git`. You may also be willing to insert your credentials in the command line (less secure) `git clone https://${USER}:${TOKEN}@github.com/GEOS-DEV/GEOS.git`.

1.1.5 Configuration

At a minimum, you will need a relatively recent compiler suite installed on your system (e.g. GCC, Clang) as well as CMake. If you want to run jobs using MPI-based parallelism, you will also need an MPI implementation (e.g. OpenMPI, MVAPICH). Note that GEOS supports a variety of parallel computing models, depending on the hardware and software environment. Advanced users are referred to the *Build Guide* for a discussion of the available configuration options.

Before beginning, it is a good idea to have a clear idea of the flavor and version of the build tools you are using. If something goes wrong, the first thing the support team will ask you for is this information.

```
cpp --version
mpic++ --version
cmake --version
```

Here, you may need to replace cpp with the full path to the C++ compiler you would like to use, depending on how your path and any aliases are configured.

GEOS compilations are driven by a cmake host-config file, which tells the build system about the compilers you are using, where various packages reside, and what options you want to enable. We have created a number of default hostconfig files for common systems. You should browse them to see if any are close to your needs:

```
cd GEOS/host-configs
```

We maintain host configs (ending in .cmake) for HPC systems at various institutions, as well as ones for common personal systems. If you cannot find one that matches your needs, we suggest beginning with one of the shorter ones and modifying as needed. A typical one may look like:

```
# file: your-platform.cmake

# detect host and name the configuration file
site_name(HOST_NAME)
set(CONFIG_NAME "your-platform" CACHE PATH "")
message("CONFIG_NAME = ${CONFIG_NAME}")

# set paths to C, C++, and Fortran compilers. Note that while GEOS does not contain any␣
↪Fortran code,
# some of the third-party libraries do contain Fortran code. Thus a Fortran compiler␣
↪must be specified.
set(CMAKE_C_COMPILER "/usr/bin/clang" CACHE PATH "")
set(CMAKE_CXX_COMPILER "/usr/bin/clang++" CACHE PATH "")
set(CMAKE_Fortran_COMPILER "/usr/local/bin/gfortran" CACHE PATH "")
set(ENABLE_FORTRAN OFF CACHE BOOL "" FORCE)

# enable MPI and set paths to compilers and executable.
# Note that the MPI compilers are wrappers around standard serial compilers.
# Therefore, the MPI compilers must wrap the appropriate serial compilers specified
# in CMAKE_C_COMPILER, CMAKE_CXX_COMPILER, and CMAKE_Fortran_COMPILER.
set(ENABLE_MPI ON CACHE BOOL "")
set(MPI_C_COMPILER "/usr/local/bin/mpicc" CACHE PATH "")
set(MPI_CXX_COMPILER "/usr/local/bin/mpicxx" CACHE PATH "")
set(MPI_Fortran_COMPILER "/usr/local/bin/mpifort" CACHE PATH "")
set(MPIEXEC "/usr/local/bin/mpirun" CACHE PATH "")

# disable CUDA and OpenMP
```

(continues on next page)

(continued from previous page)

```
set(ENABLE_CUDA OFF CACHE BOOL "" FORCE)
set(ENABLE_OPENMP OFF CACHE BOOL "" FORCE)

# enable PVTPackage
set(ENABLE_PVTPackage ON CACHE BOOL "" FORCE)

# enable tests
set(ENABLE_GTEST_DEATH_TESTS ON CACHE BOOL "" FORCE )

# define the path to your compiled installation directory
set(GEOSX_TPL_DIR "/path/to/your/TPL/installation/dir" CACHE PATH "")
# let GEOS define some third party libraries information for you
include(${CMAKE_CURRENT_LIST_DIR}/tpls.cmake)
```

The various `set()` commands are used to set environment variables that control the build. You will see in the above example that we set the C++ compiler to `/user/bin/clang++` and so forth. We also disable CUDA and OpenMP, but enable PVTPackage. The final line is related to our unit test suite. See the *Build Guide* for more details on available options.

Note: If you develop a new `host-config` for a particular platform that may be useful for other users, please consider sharing it with the developer team.

1.1.6 Compilation

We will begin by compiling the TPLs, followed by the main code. If you work on an HPC system with other GEOS developers, check with them to see if the TPLs have already been compiled in a shared directory. If this is the case, you can skip ahead to just compiling the main code. If you are working on your own machine, you will need to compile both.

We strongly suggest that GEOS and TPLs be built with the same hostconfig file. Below, we assume that you keep it in, say, `GEOS/host-configs/your-platform.cmake`, but this is up to you.

We begin with the third-party libraries, and use a python `config-build.py` script to configure and build all of the TPLs. Note that we will request a Release build type, which will enable various optimizations. The other option is a Debug build, which allows for debugging but will be much slower in production mode. The TPLS will then be built in a build directory named consistently with your hostconfig file.

```
cd thirdPartyLibs
python scripts/config-build.py -hc ../GEOS/host-configs/your-platform.cmake -bt Release
cd build-your-platform-release
make
```

Note that building all of the TPLs can take quite a while, so you may want to go get a cup of coffee at this point. Also note that you should *not* use a parallel `make -j N` command to try and speed up the build time.

The next step is to compile the main code. Again, the `config-build.py` sets up cmake for you, so the process is very similar.

```
cd ../../GEOS
python scripts/config-build.py -hc host-configs/your-platform.cmake -bt Release
cd build-your-platform-release
```

(continues on next page)

(continued from previous page)

```
make -j4
make install
```

The host-config file is the place to set all relevant configuration options. Note that the path to the previously installed third party libraries is typically specified within this file. An alternative is to set the path `GEOSX_TPL_DIR` via a cmake command line option, e.g.

```
python scripts/config-build.py -hc host-configs/your-platform.cmake -bt Release -D GEOSX_
↪TPL_DIR=/full/path/to/thirdPartyLibs
```

We highly recommend using full paths, rather than relative paths, whenever possible. The parallel `make -j 4` will use four processes for compilation, which can substantially speed up the build if you have a multi-processor machine. You can adjust this value to match the number of processors available on your machine. The `make install` command then installs GEOS to a default location unless otherwise specified.

If all goes well, a `geosx` executable should now be available:

```
GEOS/install-your-platform-release/bin/geosx
```

1.1.7 Running

We can do a quick check that the geosx executable is working properly by calling the executable with our help flag

```
./bin/geosx --help
```

This should print out a brief summary of the available command line arguments:

```
USAGE: geosx -i input.xml [options]

Options:
-?, --help
-i, --input,              Input xml filename (required)
-r, --restart,            Target restart filename
-x, --x-partitions,       Number of partitions in the x-direction
-y, --y-partitions,       Number of partitions in the y-direction
-z, --z-partitions,       Number of partitions in the z-direction
-s, --schema,             Name of the output schema
-b, --use-nonblocking,    Use non-blocking MPI communication
-n, --name,               Name of the problem, used for output
-s, --suppress-pinned,    Suppress usage of pinned memory for MPI communication buffers
-o, --output,             Directory to put the output files
-t, --timers,             String specifying the type of timer output
--trace-data-migration,   Trace host-device data migration
--pause-for,              Pause geosx for a given number of seconds before starting␣
↪execution
```

Obviously this doesn't do much interesting, but it will at least confirm that the executable runs. In typical usage, an input XML must be provided describing the problem to be run, e.g.

```
./bin/geosx -i your-problem.xml
```

In a parallel setting, the command might look something like

```
mpirun -np 8 ./bin/geosx -i your-problem.xml -x 2 -y 2 -z 2
```

Note that we provide a series of *Tutorials* to walk you through the actual usage of the code, with several input examples. Once you are comfortable the build is working properly, we suggest new users start working through these tutorials.

1.1.8 Testing

It is wise to run our unit test suite as an additional check that everything is working properly. You can run them in the build folder you just created.

```
cd GEOS/build-your-platform-release
ctest -V
```

This will run a large suite of simple tests that check various components of the code. If you have access, you may also consider running the integrated tests. Please refer to *Integrated Tests* for further information.

Note: If *all* of the unit tests fail, there is likely something wrong with your installation. Refer to the FAQs above for how best to proceed in this situation. If only a few tests fail, it is possible that your platform configuration has exposed some issue that our existing platform tests do not catch. If you suspect this is the case, please consider posting an issue to our issue tracker (after first checking whether other users have encountered a similar issue).

1.2 Tutorials

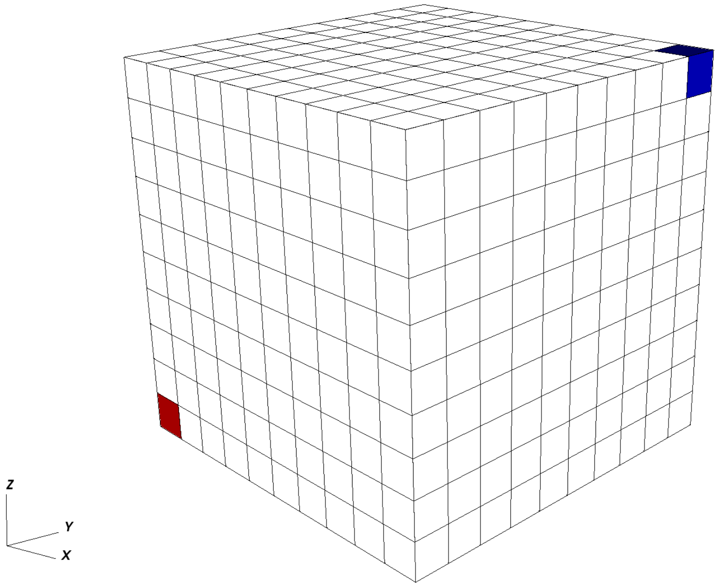
The easiest way to learn to use GEOS is through worked examples. Here, we have included tutorials showing how to run some common problems. After working through these examples, you should have a good understanding of how to set up and solve your own models.

Note that these tutorials are intended to be followed in sequence, as each step introduces a few new skills. Most of the tutorial models are also quite small, so that large computational resources are not required.

1.2.1 Tutorial 1: First Steps

Context

In this tutorial, we use a single-phase flow solver (see *Singlephase Flow Solver*) to solve for pressure propagation on a 10x10x10 cube mesh with anisotropic permeability values. The pressure source is the lowest-left corner element, and the pressure sink sits at the opposite top corner.



Objectives

At the end of this tutorial you will know:

- the basic structure of XML input files used by GEOS,
- how to run GEOS on a simple case requiring no external input files,
- the basic syntax of a solver block for single-phase problems,
- how to control output and visualize results.

Input file

GEOS runs by reading user input information from one or more XML files. For this tutorial, we only need a single GEOS input file located at:

```
inputFiles/singlePhaseFlow/3D_10x10x10_compressible_smoke.xml
```

Running GEOS

If our XML input file is called `my_input.xml`, GEOS runs this file by executing:

```
/path/to/geosx -i /path/to/my_input.xml
```

The `-i` flag indicates the path to the XML input file. To get help on what other command line input flags GEOS supports, run `geosx --help`.

Input file structure

XML files store information in a tree-like structure using nested blocks of information called *elements*. In GEOS, the root of this tree structure is the element called `Problem`. All elements in an XML file are defined by an opening *tag* (`<ElementName>`) and end by a corresponding closing tag (`</ElementName>`). Elements can have properties defined as *attributes* with `key="value"` pairs. A typical GEOS input file contains the following tags:

1. *Solver*
2. *Mesh*

3. *Geometry*
4. *Events*
5. *NumericalMethods*
6. *ElementRegions*
7. *Constitutive*
8. *FieldSpecifications*
9. *Outputs*

XML validation tools

If you have not already done so, please use or enable an XML validation tool (see **User Guide/Input Files/Input Validation**). Such tools will help you identify common issues that may occur when working with XML files.

Note: Common errors come from the fact that XML is case-sensitive, and all opened tags must be properly closed.

Single-phase solver

GEOS is a multiphysics simulator. To find the solution to different physical problems such as diffusion or mechanical deformation, GEOS uses one or more physics solvers. The `Solvers` tag is used to define and parameterize these solvers. Different combinations of solvers can be applied in different regions of the domain at different moments of the simulation.

In this first example, we use one type of solver in the entire domain and for the entire duration of the simulation. The solver we are specifying here is a single-phase flow solver. In GEOS, such a solver is created using a `SinglePhaseFVM` element. This type of solver is one among several cell-centered single-phase finite volume methods.

The XML block used to define this single-phase finite volume solver is shown here:

```
<Solvers>
  <SinglePhaseFVM
    name="SinglePhaseFlow"
    logLevel="1"
    discretization="singlePhaseTPFA"
    targetRegions="{ mainRegion }">
    <NonlinearSolverParameters
      newtonTol="1.0e-6"
      newtonMaxIter="8"/>
    <LinearSolverParameters
      solverType="gmres"
      preconditionerType="amg"
      krylovTol="1.0e-10"/>
  </SinglePhaseFVM>
</Solvers>
```

Each type of solver has a specific set of parameters that are required and some parameters that are optional. Optional values are usually set with sensible default values.

name

First, we register a solver of type `SinglePhaseFVM` with a user-chosen name, here `SinglePhaseFlow`. This unique user-defined name can be almost anything. However, some symbols are known to cause issues in names : avoid commas, slashes, curly braces. GEOS is case-sensitive: it makes a distinction between two `SinglePhaseFVM` solvers called mySolver and MySolver. Giving elements a name is a common practice in GEOS: users need to give unique identifiers to objects they define. That name is the handle to this instance of a solver class.

logLevel

Then, we set a solver-specific level of console logging (logLevel set to 1 here). Notice that the value (1) is between double-quotes. This is a general convention for all attributes: we write key="value" regardless of the value type (integers, strings, lists, etc.).

For logLevel, higher values lead to more console output or intermediate results saved to files. When debugging, higher logLevel values is often convenient. In production runs, you may want to suppress most console output.

discretization

For solvers of the SinglePhaseFVM family, one required attribute is a discretization scheme. Here, we use a Two-Point Flux Approximation (TPFA) finite volume discretization scheme called singlePhaseTPFA. To know the list of admissible values of an attribute, please see GEOS's XML schema. This discretization type must know how to find permeability values that it uses internally to compute transmissibilities. The permeabilityNames attribute tells the solver the user-defined name (the *handle*) of the permeability values that will be defined elsewhere in the input file. Note that the order of attributes inside an element is not important.

fluidNames, solidNames, targetRegions

Here, we specify a collection of fluids, rocks, and target regions of the mesh on which the solver will apply. Curly brackets are used in GEOS inputs to indicate collections of values (sets or lists). The curly brackets used here are necessary, even if the collection contains a single value. Commas are used to separate members of a set.

Nested elements

Finally, note that other XML elements can be nested inside the Solvers element. Here, we use specific XML elements to set values for numerical tolerances. The solver stops when numerical residuals are smaller than the specified tolerances (convergence is achieved) or when the maximum number of iterations allowed is exceeded (convergence not achieved).

Mesh

To solve this problem, we need to define a mesh for our numerical calculations. This is the role of the **Mesh** element.

There are two approaches to specifying meshes in GEOS: internal or external.

- The external approach allows to import mesh files created outside GEOS, such as a corner-point grid or an unstructured grid representing complex shapes and structures.
- The internal approach uses GEOS's built-in capability to create simple meshes from a small number of parameters. It does not require any external file information. The geometric complexity of internal meshes is limited, but many practical problems can be solved on such simple grids.

In this tutorial, to keep things self-contained, we use the internal mesh generator. We parameterize it with the **InternalMesh** element.

```xml
<Mesh>
  <InternalMesh
      name="mesh"
      elementTypes="{ C3D8 }"
      xCoords="{ 0, 10 }"
      yCoords="{ 0, 10 }"
      zCoords="{ 0, 10 }"
      nx="{ 10 }"
      ny="{ 10 }"
```

(continues on next page)

(continued from previous page)

```
        nz="{ 10 }"
        cellBlockNames="{ cellBlock }"/>
  </Mesh>
```

name

Just like for solvers, we register the `InternalMesh` element using a unique **name** attribute. Here the `InternalMesh` object is instantiated with the name `mesh`.

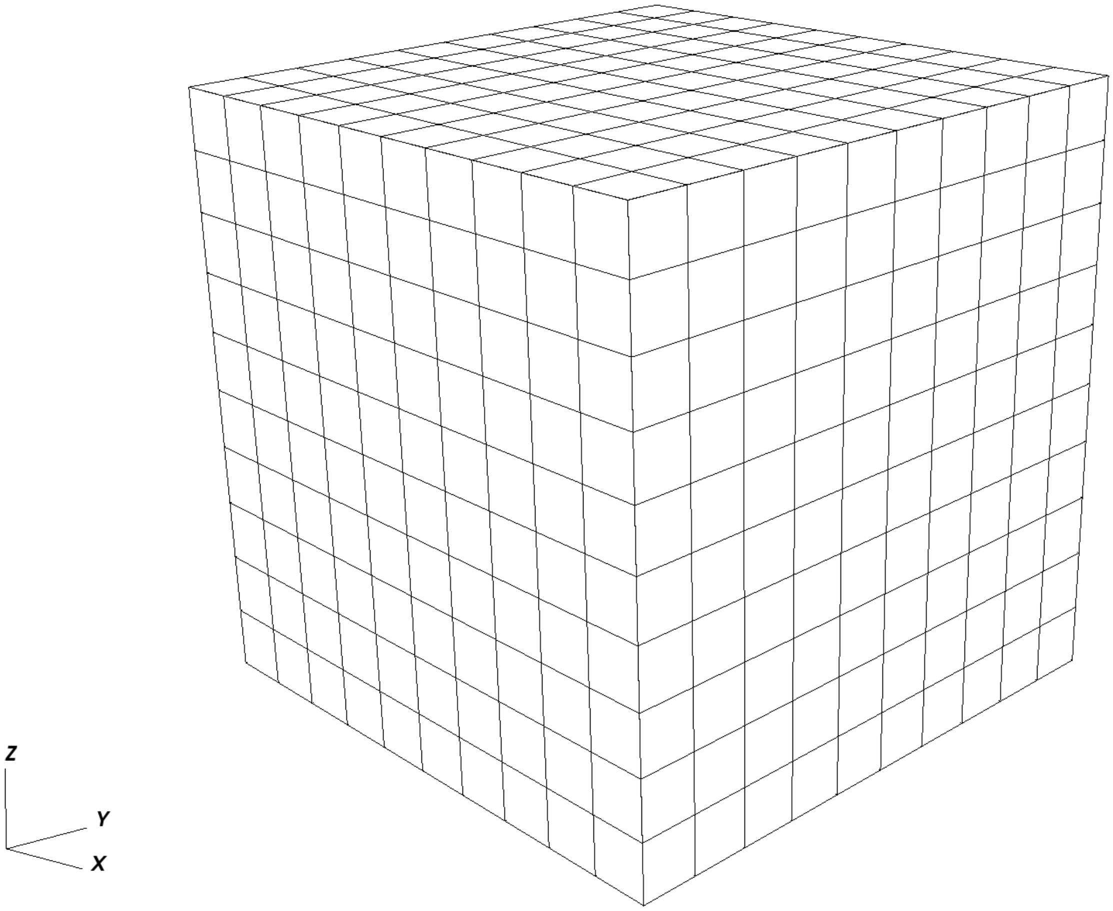
elementTypes

We specify the collection of elements types that this mesh contains. Tetrahedra, hexahedra, and wedges are examples of element types. If a mesh contains different types of elements (a hybrid mesh), we should indicate this here by listing all unique types of elements in curly brackets. Keeping things simple, our element collection has only one type of element: a `C3D8` type representing a hexahedral element (linear 8-node brick).

A mesh can contain several geometrical types of elements. For numerical convenience, elements are aggregated by types into `cellBlocks`. Here, we only linear 8-node brick elements, so the entire domain is one object called `cellBlock`.

xCoords, yCoords, zCoords, nx, ny, nz

This specifies the spatial arrangement of the mesh elements. The mesh defined here goes from coordinate x=0 to x=10 in the x-direction, with `nx=10` subdivisions along this segment. The same is true for the y-dimension and the z-dimension. Our mesh is a cube of 10x10x10=1,000 elements with a bounding box defined by corner coordinates (0,0,0) and (10,10,10).



Geometry

The Geometry tag allows users to capture subregions of a mesh and assign them a unique name. Here, we name two Box elements, one for the location of the source and one for the sink. Pressure values are assigned to these named regions elsewhere in the input file.

The pressure source is the element in the (0,0,0) corner of the domain, and the sink is the element in the (10,10,10) corner.

For an element to be inside a geometric region, it must have all its vertices strictly inside that region. Consequently, we need to extend the geometry limits a small amount beyond the actual coordinates of the elements to catch all vertices. Here, we use a safety padding of 0.01.

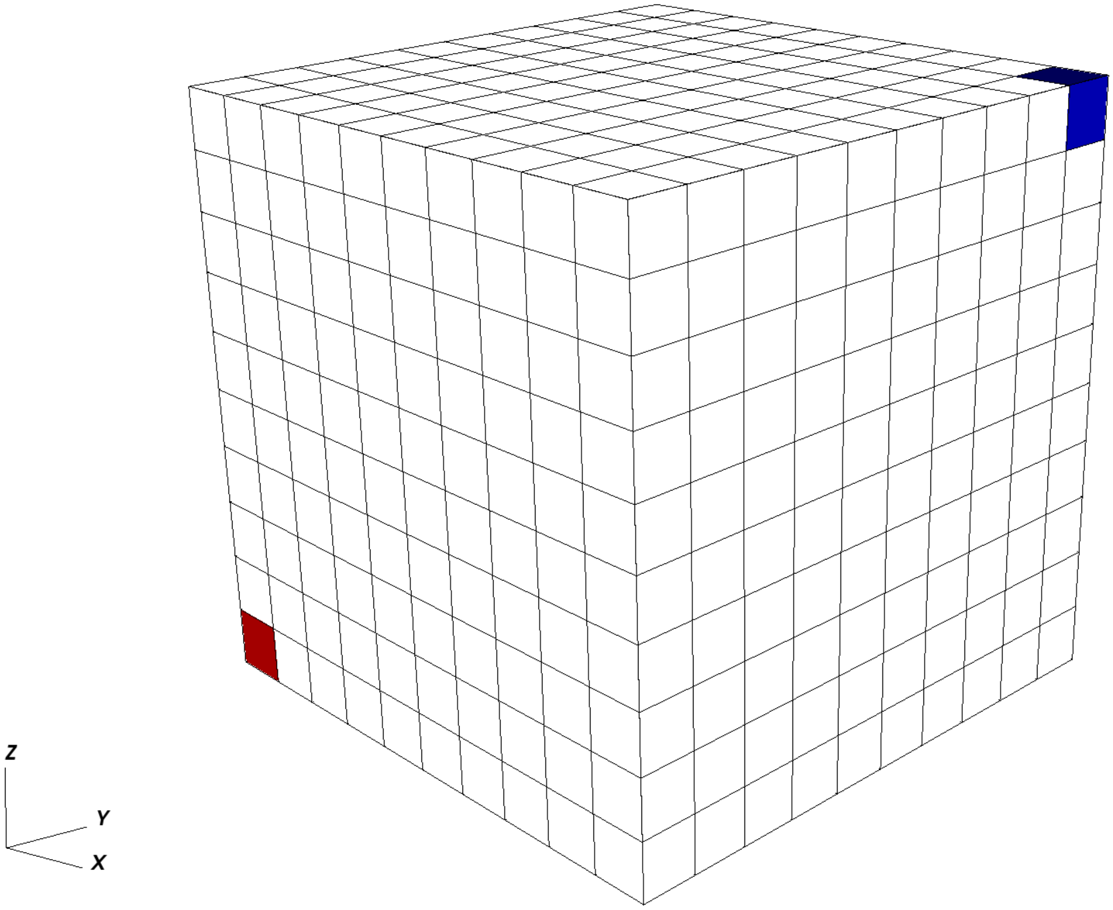
```
<Geometry>
  <Box
    name="source"
    xMin="{ -0.01, -0.01, -0.01 }"
    xMax="{ 1.01, 1.01, 1.01 }"/>

  <Box
```

(continues on next page)

(continued from previous page)

```
    name="sink"
    xMin="{ 8.99, 8.99, 8.99 }"
    xMax="{ 10.01, 10.01, 10.01 }"/>
</Geometry>
```

There are several methods to achieve similar conditions (Dirichlet boundary condition on faces, etc.). The Box defined here is one of the simplest approaches.



Events

In GEOS, we call Events anything that happens at a set time or frequency. Events are a central element for time-stepping in GEOS, and a dedicated section just for events is necessary to give them the treatment they deserve.

For now, we focus on three simple events: the time at which we wish the simulation to end (maxTime), the times at which we want the solver to perform updates, and the times we wish to have simulation output values reported.

In GEOS, all times are specified in **seconds**, so here maxTime=5000.0 means that the simulation will run from time 0 to time 5,000 seconds.

If we focus on the PeriodicEvent elements, we see :

1. A **periodic solver** application: this event is named solverApplications. With the attribute forceDt=20, it tells the solver to compute results at 20-second time intervals. We know what this event does by looking at its target attribute: here, from time 0 to maxTime and with a forced time step of 20 seconds, we instruct GEOS to call the solver registered as SinglePhaseFlow. Note the hierarchical structure of the target formulation, using '/' to indicate a specific named instance (SinglePhaseFlow) of an element (Solvers). If the solver needs to take smaller time steps, it is allowed to do so, but it will have to compute results for every 20-second increment between time zero and maxTime regardless of possible intermediate time steps.
2. An **output event**: this event is used for reporting purposes and instructs GEOS to write out results at specific frequencies. Here, we need to see results at every 100-second increment. This event triggers a full application of solvers, even if solvers were not summoned by the previous event. In other words, an output event will force an application of solvers, possibly in addition to the periodic events requested directly.

```
<Events maxTime="5000.0">
  <PeriodicEvent
      name="solverApplications"
      forceDt="20.0"
      target="/Solvers/SinglePhaseFlow"/>
  <PeriodicEvent
      name="outputs"
      timeFrequency="100.0"
      target="/Outputs/siloOutput"/>
</Events>
```

Numerical methods

GEOS comes with several useful numerical methods. In the Solvers elements, for instance, we had specified to use a two-point flux approximation as discretization scheme for the finite volume single-phase solver. Now to use this scheme, we need to supply more details in the NumericalMethods element.

```
<NumericalMethods>
  <FiniteVolume>
    <TwoPointFluxApproximation
      name="singlePhaseTPFA"
      />
  </FiniteVolume>
</NumericalMethods>
```

The fieldName attribute specifies which property will be used for flux computations, and also specifies that for Dirichlet boundary conditions, the pressure value at the element face is used. The coefficientName attribute is used for the stencil transmissibility computations.

Note that in GEOS, there is a difference between physics solvers and numerical methods. Their parameterizations are thus independent. We can have multiple solvers using the same numerical scheme but with different tolerances, for instance.

Regions

In GEOS, `ElementsRegions` are used to attach material properties to regions of elements. Here, we use only one **CellElementRegion** to represent the entire domain (user name: `mainRegion`). It contains all the blocks called `cellBlock` defined in the mesh section. We specify the materials contained in that region using a `materialList`. Several materials coexist in `cellBlock`, and we list them using their user-defined names: `water`, `rockPorosity`, and `rockPerm`, etc. What these names mean, and the physical properties that they are attached to are defined next.

```xml
<ElementRegions>
  <CellElementRegion
    name="mainRegion"
    cellBlocks="{ cellBlock }"
    materialList="{ water, rock }"/>
</ElementRegions>
```

Constitutive models

The `Constitutive` element attaches physical properties to all materials contained in the domain.

The physical properties of the materials defined as `water`, `rockPorosity`, and `rockPerm` are provided here, each material being derived from a different material type: `CompressibleSinglePhaseFluid` for the water, `PressurePorosity` for the rock porosity, and `ConstantPermeability` for rock permeability. The list of attributes differs between these constitutive materials.

```xml
<Constitutive>
  <CompressibleSinglePhaseFluid
    name="water"
    defaultDensity="1000"
    defaultViscosity="0.001"
    referencePressure="0.0"
    compressibility="5e-10"
    viscosibility="0.0"/>

  <CompressibleSolidConstantPermeability
    name="rock"
    solidModelName="nullSolid"
    porosityModelName="rockPorosity"
    permeabilityModelName="rockPerm"/>

  <NullModel
    name="nullSolid"/>

  <PressurePorosity
    name="rockPorosity"
    defaultReferencePorosity="0.05"
    referencePressure="0.0"
    compressibility="1.0e-9"/>

  <ConstantPermeability
    name="rockPerm"
    permeabilityComponents="{ 1.0e-12, 1.0e-12, 1.0e-15 }"/>
</Constitutive>
```

The names water, rockPorosity and rockPerm are defined by the user as handles to specific instances of physical materials. GEOS uses S.I. units throughout, not field units. Pressures, for instance, are in Pascal, not psia. The x- and y-permeability are set to 1.0e-12 m^2 corresponding to approximately to 1 Darcy.

We have used the handles water, rockPorosity and rockPerm in the input file in the ElementRegions section of the XML file, before the registration of these materials took place here, in Constitutive element.

Note: This highlights an important aspect of using XML in GEOS: the order in which objects are registered and used in the XML file is not important.

Defining properties

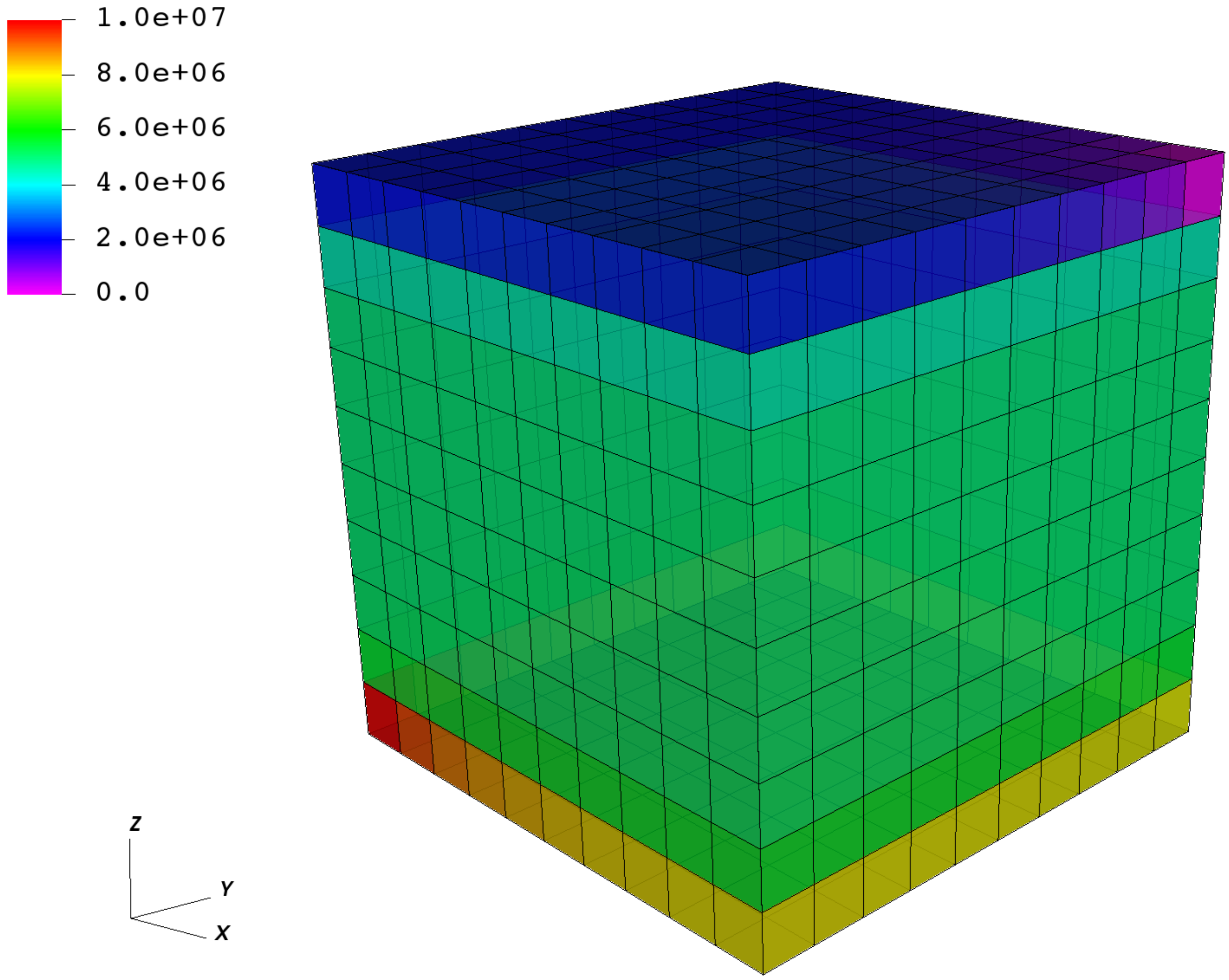
In the FieldSpecifications section, properties such as source and sink pressures are set. GEOS offers a lot of flexibility to specify field values through space and time.

Spatially, in GEOS, all field specifications are associated to a target object on which the field values are mounted. This allows for a lot of freedom in defining fields: for instance, one can have volume property values attached to a subset of volume elements of the mesh, or surface properties attached to faces of a subset of elements.

For each FieldSpecification, we specify a name, a fieldName (this name is used by solvers or numerical methods), an objectPath, setNames and a scale. The ObjectPath is important and it reflects the internal class hierarchy of the code. Here, for the fieldName pressure, we assign the value defined by scale (5e6 Pascal) to one of the ElementRegions (class) called mainRegions (instance). More specifically, we target the elementSubRegions called cellBlock (this contains all the C3D8 elements, effectively all the domain). The setNames allows to use the elements defined in Geometry, or use everything in the object path (using the all).

```xml
<FieldSpecifications>
  <FieldSpecification
    name="initialPressure"
    initialCondition="1"
    setNames="{ all }"
    objectPath="ElementRegions/mainRegion/cellBlock"
    fieldName="pressure"
    scale="5e6"/>

  <FieldSpecification
    name="sourceTerm"
    objectPath="ElementRegions/mainRegion/cellBlock"
    fieldName="pressure"
    scale="1e7"
    setNames="{ source }"/>

  <FieldSpecification
    name="sinkTerm"
    objectPath="ElementRegions/mainRegion/cellBlock"
    fieldName="pressure"
    scale="0.0"
    setNames="{ sink }"/>
</FieldSpecifications>
```

The image below shows the pressures after the very first time step, with the domain initialized at 5 MPa, the sink at 0 MPa on the top right, and the source in the lower left corner at 10 MPa.



Output

In order to retrieve results from a simulation, we need to instantiate one or multiple `Outputs`.

Here, we define a single object of type `Silo`. Silo is a library and a format for reading and writing a wide variety of scientific data. Data in Silo format can be read by VisIt.

This `Silo` output object is called `siloOutput`. We had referred to this object already in the `Events` section: it was the target of a periodic event named `outputs`. You can verify that the Events section is using this object as a target. It does so by pointing to `/Outputs/siloOutput`.

```
<Outputs>
  <Silo
    name="siloOutput"/>
</Outputs>
```

GEOS currently supports outputs that are readable by VisIt and Kitware's Paraview, as well as other visualization tools. In this example, we only request a Silo format compatible with VisIt.

All elements are now in place to run GEOS.

Running GEOS

The command to run GEOS is

`path/to/geosx -i path/to/this/xml_file.xml`

Note that all paths for files included in the XML file are relative to this XML file.

While running GEOS, it logs status information on the console output with a verbosity that is controlled at the object level, and that can be changed using the `logLevel` flag.

The first few lines appearing to the console are indicating that the XML elements are read and registered correctly:

```
Adding Solver of type SinglePhaseFVM, named SinglePhaseFlow
Adding Mesh: InternalMesh, mesh
Adding Geometric Object: Box, source
Adding Geometric Object: Box, sink
Adding Event: PeriodicEvent, solverApplications
Adding Event: PeriodicEvent, outputs
Adding Output: Silo, siloOutput
Adding Object CellElementRegion named mainRegion from ObjectManager::Catalog.
  mainRegion/cellBlock/water is allocated with 1 quadrature points.
  mainRegion/cellBlock/rock is allocated with 1 quadrature points.
  mainRegion/cellBlock/rockPerm is allocated with 1 quadrature points.
  mainRegion/cellBlock/rockPorosity is allocated with 1 quadrature points.
  mainRegion/cellBlock/nullSolid is allocated with 1 quadrature points.
```

Then, we go into the execution of the simulation itself:

```
Time: 0s, dt:20s, Cycle: 0
    Attempt:  0, NewtonIter:  0
    ( R ) = ( 5.65e+00 ) ;
    Attempt:  0, NewtonIter:  1
    ( R ) = ( 2.07e-04 ) ;
    Last LinSolve(iter,res) = (  63, 8.96e-11 ) ;
    Attempt:  0, NewtonIter:  2
    ( R ) = ( 9.86e-11 ) ;
    Last LinSolve(iter,res) = (  70, 4.07e-11 ) ;
```

Each time iteration at every 20s interval is logged to console, until the end of the simulation at `maxTime=5000`:

```
Time: 4980s, dt:20s, Cycle: 249
    Attempt:  0, NewtonIter:  0
    ( R ) = ( 4.74e-09 ) ;
    Attempt:  0, NewtonIter:  1
    ( R ) = ( 2.05e-14 ) ;
    Last LinSolve(iter,res) = (  67, 5.61e-11 ) ;
SinglePhaseFlow: Newton solver converged in less than 4 iterations, time-step required␣
↪will be doubled.
Cleaning up events
Umpire            HOST sum across ranks:   14.8 MB
Umpire            HOST         rank max:   14.8 MB
total time                          5.658s
initialization time                 0.147s
run time                            3.289s
```

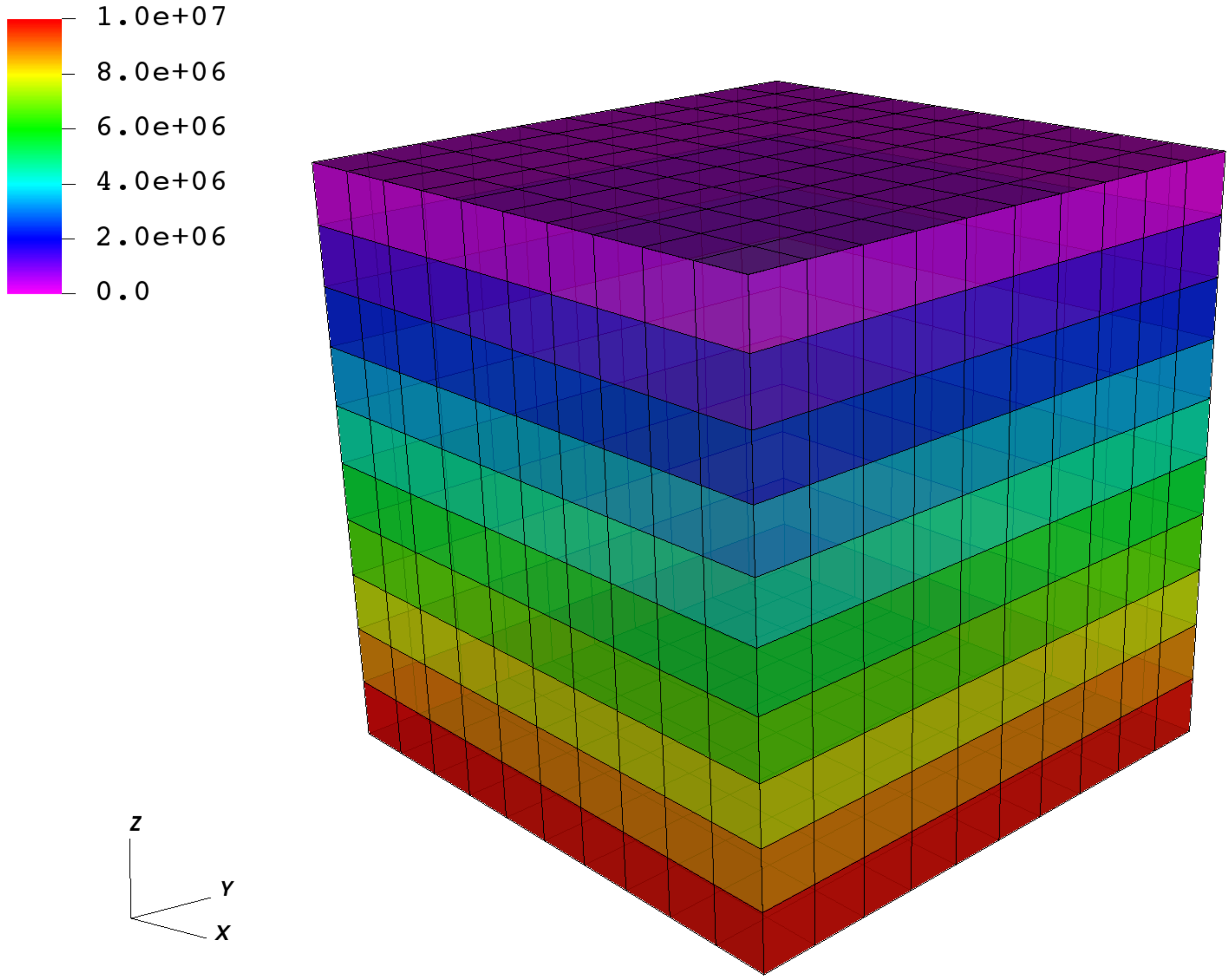
All newton iterations are logged along with corresponding nonlinear residuals for each time iteration. In turn, for each newton iteration, `LinSolve` provides the number of linear iterations and the final residual reached by the linear solver. Information on run times, initialization times, and maximum amounts of memory (high water mark) are given at the end of the simulation, if successful.

Congratulations on completing this first run!

Visualization

Here, we have requested results to be written in Silo, a format compatible with VisIt. To visualize results, open VisIt and directly load the database of simulation output files.

After a few time step, pressure between the source and sink are in equilibrium, as shown on the representation below.



To go further

Feedback on this tutorial

This concludes the single-phase internal mesh tutorial. For any feedback on this tutorial, please submit a GitHub issue on the project's GitHub page.

For more details

- More on single-phase flow solvers, please see *Singlephase Flow Solver*.
- More on meshes, please see *Meshes*.
- More on events, please see *Event Management*.

1.2.2 Tutorial 2: External Meshes

Context

In this tutorial, we use a simple single-phase flow solver (see *Singlephase Flow Solver*) to solve for pressure propagation on a mesh that is imported into GEOS. The main goal of this tutorial is to learn how to work with external meshes, and to learn how easy it is to swap meshes on the same physical problem in GEOS. This makes GEOS a powerful tool to solve real field applications with complex geometries and perform assessments of mesh geometry and resolution effects.

Objectives

At the end of this tutorial you will know:

- the syntax and format of input meshes,
- how to input external files into a GEOS input XML file,
- how to run the same physical problem with two different meshes,
- how to use and visualize hexahedral and tetrahedral meshes.

Input Files

This tutorial uses an XML file containing the main input for GEOS and a separate file with all the mesh information. As we will see later, the main XML file points to the external mesh file with an `include` statement. The XML input file for this test case is located at:

```
inputFiles/singlePhaseFlow/vtk/3D_10x10x10_compressible_hex_gravity_smoke.xml
```

The mesh file format used in this tutorial is vtk. This format is a standard scientific meshing format not specific to GEOS. `vtk` is a multi-purpose mesh format (structured, unstructured, serial, parallel, multi-block…) and contains a compact and complete representation of the mesh geometry and of its properties. The mesh file used here is human-readable ASCII, and there is a binary storage as well. It contains a list of nodes with their (x,y,z) coordinates, and a list of elements that are constructed from these nodes.

Hexahedral elements

In the first part of the tutorial, we will run flow simulations on a mesh made of hexahedral elements. These types of elements are used in classical cartesian grids (sugar cubes) or corner-point grids or pillar grids.

Brief discussion about hexahedral meshes in GEOS

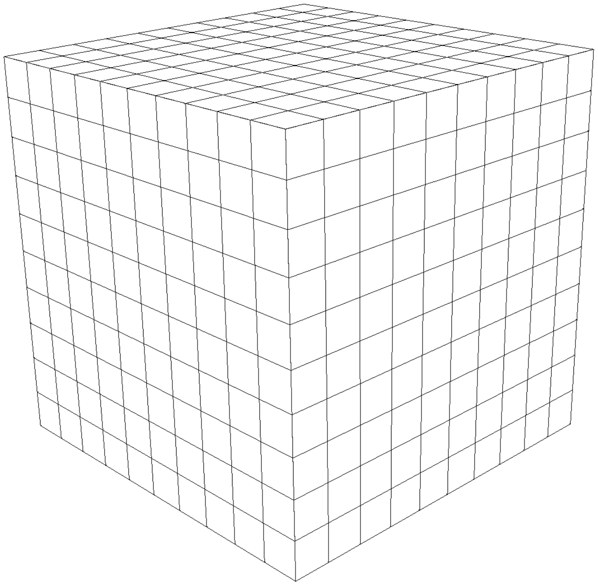
Although closely related, the hexahedral grids that GEOS can process are slightly different than either structured grid or corner-point grids. The differences are worth pointing out here. In GEOS:

- **Hexahedra can have irregular shapes**: no pillars are needed and vertices can be anywhere in space. This is useful for grids that turn, fold, or are heavily bent. Hexahedral blocks should nevertheless have 8 distinct vertices that are not coalesced. Some tolerance exists for degeneration to wedges in some solvers (finite element solvers), but it is best to avoid such situations and label elements according to their actual shape. Butterfly cells, flat cells, negative or zero volume cells will cause problems.
- **The mesh needs to be conformal:** in 3D, this means that neighboring grid blocks have to share exactly a complete face. Note that corner-point grids do not have this requirement and neighboring blocks can be offset. When importing grids from commonly-used geomodeling packages, this is an important consideration. This problem is solved by splitting shifted grid blocks to restore conformity. While it may seem convenient to be able to have offset grid blocks at first, the advantages of conformal grids used in GEOS are worth the extra meshing effort: by using conformal grids, GEOS can run finite element and finite volume simulations on the same mesh without problems, going seamlessly from one numerical method to the other. This is key to enabling multiphysics simulation.
- **There is no assumption of overall structure**: GEOS does not need to know a number of block in the X, Y, Z direction (no NX, NY, NZ) and does not assume that the mesh is a full cartesian domain that the interesting parts of the reservoir must be carved out from. Blocks are numbered by indices that assume nothing about spatial positioning and there is no concept of (i,j,k). This approach also implies that no “masks” are needed to remove inactive or dead cells, as often done in cartesian grids to get the actual reservoir contours from a bounding box, and here we only need to specify grid blocks that are active. For performance and flexibility, this lean approach to meshes is important.

Importing an external mesh with VTK

In this first part of the tutorial, we use an hexahedral mesh provided to GEOS. This hexahedral mesh is strictly identical to the grid used in the first tutorial (*Tutorial 1: First Steps*), but instead of using the internal grid generator GEOS, we specify it with spatial node coordinates in `vtk` format. To import external grid into GEOS, we did develop a component directly using the **vtk** library.

So here, our mesh consists of a simple sugar-cube stack of size 10x10x10. We inject fluid from one vertical face of a cube (the face corresponding to x=0), and we let the pressure equilibrate in the closed domain. The displacement is a single-phase, compressible fluid subject to gravity forces, so we expect the pressure to be constant on the injection face, and to be close to hydrostatic on the opposite plane (x=10). We use GEOS to compute the pressure inside each grid block over a period of time of 100 seconds.



To see how to import such a mesh, we inspect the following XML file:

```
inputFiles/singlePhaseFlow/vtk/3D_10x10x10_compressible_hex_gravity_smoke.xml
```

In the XML `Mesh` tag, instead of an `InternalMesh` tag, we have a `VTKMesh` tag. We see that a file called `cube_10x10x10_hex.vtk` is imported using `vtk`, and this object is instantiated with a user-defined `name` value. The file here contains geometric information in vtk format (it can also contain properties, as we will see in the next tutorial).

```
  <Mesh>
    <VTKMesh
      name="CubeHex"
      file="cube_10x10x10_hex.vtk"/>
  </Mesh>
```

Here is the `vtk` file :

Listing 1.1: cube_10x10x10_hex.vtk

```
# vtk DataFile Version 4.2
Cube structured points dataset
ASCII
DATASET STRUCTURED_POINTS
DIMENSIONS 11 11 11
ORIGIN 0.0 0.0 0.0
SPACING 1.0 1.0 1.0
```

GEOS can run different physical solvers on different regions of the mesh at different times. Here, to keep things simple, we run one solver (single-phase flow) on the entire domain throughout the simulation. Even this is trivial, we need to define and name a region encompassing the entire domain and assign it to the single-phase flow solver. We also need to provide material properties to the regions. This is done by specifying `ElementRegions`. Here, the entire field is one region called `Domain`, and contains multiple constitutive models, including `water`, `rockPorosity`, and `rockPerm`.

```
<ElementRegions>
  <CellElementRegion
    name="Domain"
    cellBlocks="{ hexahedra }"
    materialList="{ water, rock }"/>
</ElementRegions>
```

Running GEOS

The command to run GEOS is

```
path/to/geosx -i ../../../../../inputFiles/singlePhaseFlow/vtk/3D_10x10x10_compressible_
↪hex_gravity_smoke.xml
```

Note that all paths for files included in the XML file are relative to this XML file, not to the GEOS executable. When running GEOS, console messages will provide indications regarding the status of the simulation.

In our case, the first lines are:

```
Adding Mesh: VTKMesh, CubeHex
Adding Event: PeriodicEvent, solverApplications
Adding Event: PeriodicEvent, outputs
Adding Event: PeriodicEvent, restarts
Adding Solver of type SinglePhaseFVM, named SinglePhaseFlow
Adding Geometric Object: Box, left
Adding Output: Silo, siloOutput
Adding Output: Restart, restartOutput
Adding Object CellElementRegion named Domain from ObjectManager::Catalog.
```

This indicates initialization of GEOS. The mesh preprocessing tool `VTKMesh` is launched next, with console messages as follows.

```
VTKMesh 'CubeHex': reading mesh from /path/to/inputFiles/singlePhaseFlow/vtk/cube_
↪10x10x10_hex.vtk
Generating global Ids from VTK mesh
VTKMesh 'CubeHex': generating GEOS mesh data structure
Number of nodes: 1331
  Number of elems: 1000
             C3D8: 1000
Load balancing:  min  avg  max
(element/rank): 1000 1000 1000
```

Notice the specification of the number of nodes (1331), and hexahedra (1000). After the adjacency calculations, GEOS starts the simulation itself. with the time-step increments specified in the XML file.

At the end of your simulation, you should see something like:

```
Time: 96s, dt:2s, Cycle: 48
Time: 98s, dt:2s, Cycle: 49
Cleaning up events
SinglePhaseFlow, number of time steps: 50
SinglePhaseFlow, number of successful nonlinear iterations: 50
SinglePhaseFlow, number of successful linear iterations: 450
```

(continues on next page)

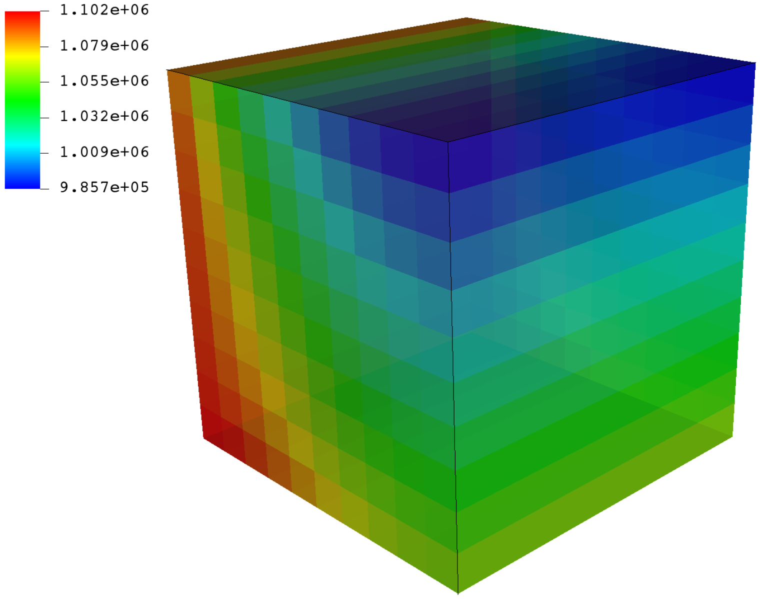
(continued from previous page)

```
SinglePhaseFlow, number of time step cuts: 0
SinglePhaseFlow, number of discarded nonlinear iterations: 0
SinglePhaseFlow, number of discarded linear iterations: 0
Umpire            HOST sum across ranks:    2.6 MB
Umpire            HOST         rank max:    2.6 MB
total time                         3.518s
initialization time                0.132s
run time                           3.076s

Process finished with exit code 0
```

Once this is done, GEOS is finished and we can inspect the outcome.

Visualization of results in VisIt

All results are written in a format compatible with VisIt. To load the results, point VisIt to the `database` file written in the Silo output folder.



We see that the face x=0 shown here in the back of the illustration applies a constant pressure boundary condition (colored in red), whereas the face across from it displays a pressure field under gravity effect, equilibrated and hydrostatic. These results are consistent with what we expect.

Let us now see if a tetrahedral mesh, under the same exact physical conditions, can reproduce these results.

Externally Generated Tetrahedral Elements

In the second part of the tutorial, we discretize the same cubic domain but with tetrahedral elements. Tetrahedral meshes are not yet common in geomodeling but offer tremendous flexibility in modeling fracture planes, faults, complex reservoir horizons and boundaries. Just like for hexahedral meshes, and for the same reasons (compatibility with finite volume and finite element methods), tetrahedral meshes in GEOS must be conformal.

As stated previously, the problem we wish to solve here is the exact same physical problem as with hexahedral grid blocks. We apply a constant pressure condition (injection) from the x=0 vertical face of the domain, and we let pressure equilibrate over time. We observe the opposite side of the cube and expect to see hydrostatic pressure profiles because of the gravitational effect. The displacement is a single phase, compressible flow subject to gravity forces. We use GEOS to compute the pressure inside each grid block.

The set-up for this problem is almost identical to the hexahedral mesh set-up. We simply point our `Mesh` tag to include a tetrahedral grid. The interest of not relying on I,J,K indices for any property specification or well trajectory makes it **easy to try different meshes for the same physical problems with GEOS**. Swapping out meshes without requiring other modifications to the input files makes mesh refinement studies easy to perform with GEOS.

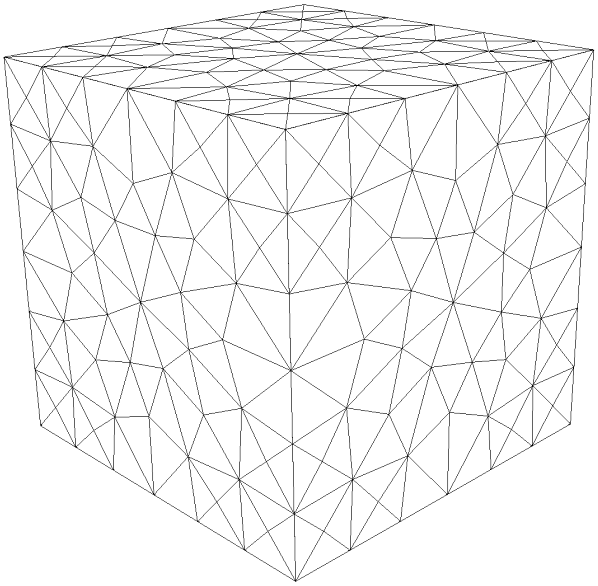
Like before, the XML file for this problem is the following:

```
inputFiles/singlePhaseFlow/vtk/3D_10x10x10_compressible_tetra_gravity_smoke.xml
```

The only difference, is that now, the `Mesh` tag points GEOS to a different mesh file called `cube_10x10x10_tet.vtk`. This file contains nodes and tetrahedral elements in vtk format, representing a different discretization of the exact same 10x10x10 cubic domain.

```
  <Mesh>
    <VTKMesh
        name="CubeTetra"
        file="cube_10x10x10_tet.vtk"/>
  </Mesh>
```

The mesh now looks like this:



And the `vtk` file starts as follows (notice the tetrahedral point coordinates as real numbers):

Listing 1.2: cube_10x10x10_tet.vtk

```
# vtk DataFile Version 2.0
cube
ASCII
DATASET UNSTRUCTURED_GRID
POINTS 366 float
0 0 10
0 0 0
0 10 10
0 10 0
10 0 10
10 0 0
10 10 10
10 10 0
0 0 1.666666666666662
0 0 3.333333333333323
0 0 4.999999999999986
0 0 6.666666666666647
0 0 8.333333333333321
0 1.666666666666662 10
0 3.333333333333323 10
```

Again, the entire field is one region called `Domain` and contains `water` and `rock` only.

```
  <ElementRegions>
    <CellElementRegion
      name="Domain"
      cellBlocks="{ tetrahedra }"
      materialList="{ water, rock }"/>
  </ElementRegions>
```

Running GEOS

The command to run GEOS is

```
path/to/geosx -i ../../../../../inputFiles/singlePhaseFlow/vtk/3D_10x10x10_compressible_
↪tetra_gravity_smoke.xml
```

Again, all paths for files included in the XML file are relative to this XML file, not to the GEOS executable. When running GEOS, console messages will provide indications regarding the status of the simulation. In our case, the first lines are:

```
Adding Mesh: VTKMesh, CubeTetra
Adding Event: PeriodicEvent, solverApplications
Adding Event: PeriodicEvent, outputs
Adding Event: PeriodicEvent, restarts
Adding Solver of type SinglePhaseFVM, named SinglePhaseFlow
Adding Geometric Object: Box, left
Adding Output: Silo, siloOutput
Adding Output: Restart, restartOutput
Adding Object CellElementRegion named Domain from ObjectManager::Catalog.
```

Followed by:

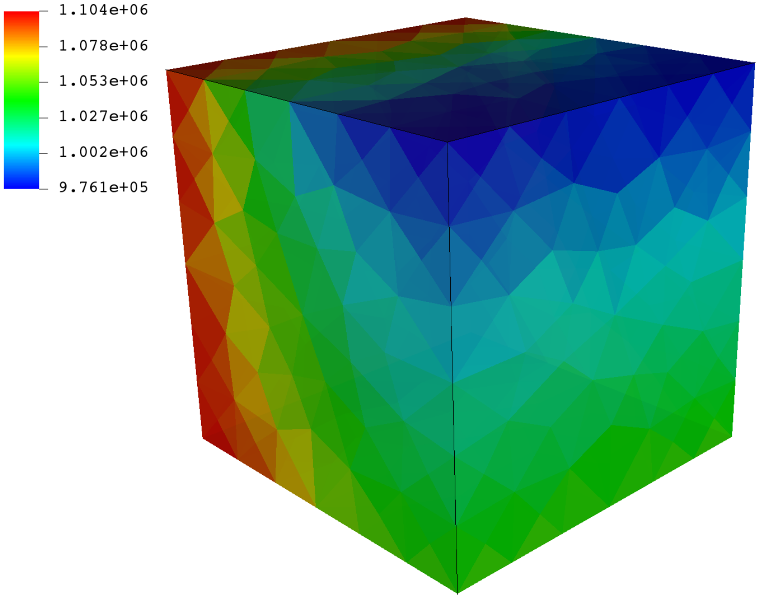
```
VTKMesh 'CubeTetra': reading mesh from /path/to/inputFiles/singlePhaseFlow/vtk/cube_10x10x10_tet.vtk
Generating global Ids from VTK mesh
VTKMesh 'CubeTetra': generating GEOS mesh data structure
Number of nodes:  366
  Number of elems: 1153
             C3D4: 1153
Load balancing:  min  avg  max
(element/rank): 1153 1153 1153
regionQuadrature: meshBodyName, meshLevelName, regionName, subRegionName = CubeTetra, Level0, Domain, tetrahedra
CubeTetra/Level0/Domain/tetrahedra/water allocated 1 quadrature points
CubeTetra/Level0/Domain/tetrahedra/rock allocated 1 quadrature points
```

We see that we have now 366 nodes and 1153 tetrahedral elements. And finally, when the simulation is successfully done we see:

```
Time: 0s, dt:1s, Cycle: 0
Time: 1s, dt:1s, Cycle: 1
Time: 2s, dt:1s, Cycle: 2
Time: 3s, dt:1s, Cycle: 3
Time: 4s, dt:1s, Cycle: 4
Time: 5s, dt:1s, Cycle: 5
...
Time: 95s, dt:1s, Cycle: 95
Time: 96s, dt:1s, Cycle: 96
Time: 97s, dt:1s, Cycle: 97
Time: 98s, dt:1s, Cycle: 98
Time: 99s, dt:1s, Cycle: 99
Cleaning up events
SinglePhaseFlow, number of time steps: 100
SinglePhaseFlow, number of successful nonlinear iterations: 100
SinglePhaseFlow, number of successful linear iterations: 1000
SinglePhaseFlow, number of time step cuts: 0
SinglePhaseFlow, number of discarded nonlinear iterations: 0
SinglePhaseFlow, number of discarded linear iterations: 0
Umpire            HOST sum across ranks:    1.9 MB
Umpire            HOST         rank max:    1.9 MB
total time                           5.837s
initialization time                  0.094s
run time                             5.432s

Process finished with exit code 0
```

Visualization of results in VisIt

All results are written in a format compatible with VisIt by default. If we load into VisIt the *.database* file found in the Silo folder, we observe the following results:



Here, we can see that despite the different mesh sizes and shapes, we are able to recover our pressure profile without any problems, or degradation in runtime performance.

To go further

Feedback on this tutorial

This concludes the single-phase external mesh tutorial. For any feedback on this tutorial, please submit a GitHub issue on the project's GitHub page.

For more details

- A complete description of the Internal Mesh generator is found here *Meshes*.
- `vtk` is extensively documented. You can start browsing here.
- GEOS can handle tetrahedra, hexahedra, pyramids, wedges, prisms, and any combination thereof in one mesh.

1.2.3 Tutorial 3: Regions and Property Specifications

Context

In this tutorial, we set up a simple field case for single-phase flow simulation (see *Singlephase Flow Solver*). We demonstrate how to run a basic flow simulation in the reservoir layer. We do not consider any coupling with wells. Injection and production will be specified by imposing a high pressure in the cells close to the injection area and a low pressure in the cells close to the production area.

Objectives

At the end of this tutorial you will know:

- how to import external mesh information and properties,
- how to run a specific solver (here, flow) in a specific region only,
- the basic method of using boxes to set up boundary conditions,
- how to use *TableFunction* to import fields varying in time and/or space,
- how to control output frequency and export results for visualization.

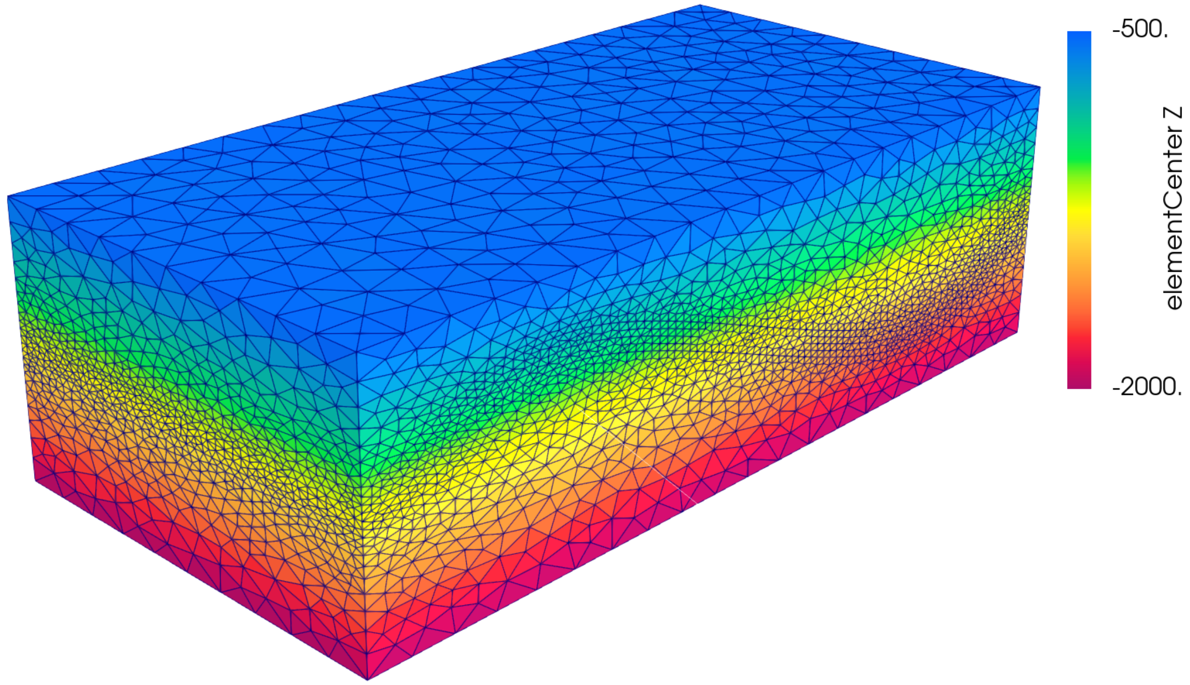
Input file

The XML input file for this test case is located at:

```
inputFiles/singlePhaseFlow/FieldCaseTutorial3_base.xml
```

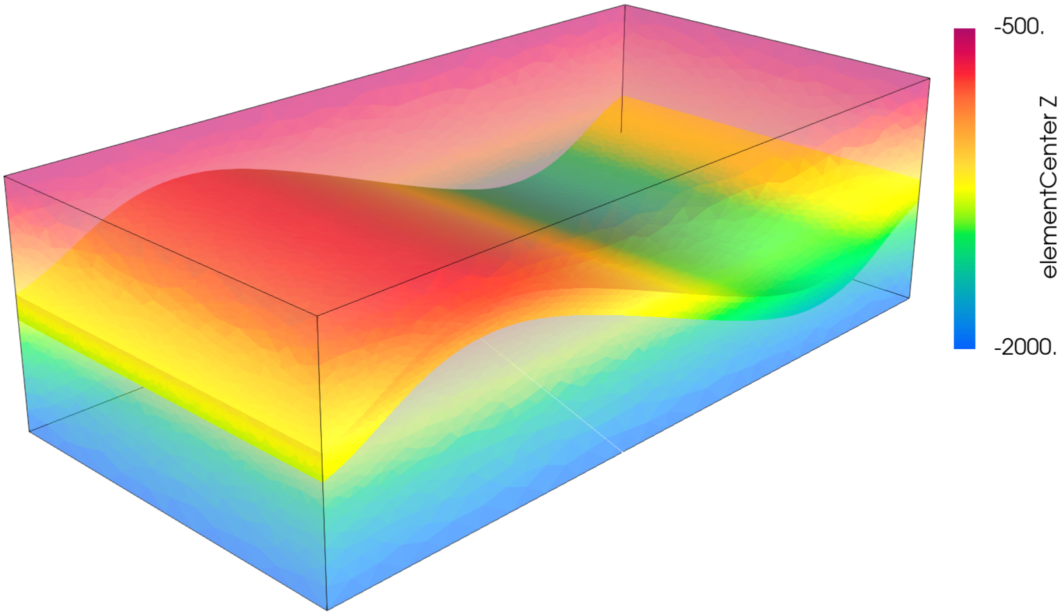
```
inputFiles/singlePhaseFlow/FieldCaseTutorial3_smoke.xml
```

We consider the following mesh as a numerical support to the simulations in this tutorial:



This mesh contains three continuous regions:

- a Top region (overburden, elementary tag = *Overburden*)
- a Middle region (reservoir layer, elementary tag = *Reservoir*)
- a Bottom region (underburden, elementary tag = *Underburden*)



The mesh is defined using the VTK file format (see *Meshes* for more information on the supported mesh file format). Each tetrahedron is associated to a unique tag.

The XML file considered here follows the typical structure of the GEOS input files:

1. *Solver*
2. *Mesh*
3. *Geometry*
4. *Events*
5. *NumericalMethods*
6. *ElementRegions*
7. *Constitutive*
8. *FieldSpecifications*
9. *Outputs*
10. *Functions*

Single-phase solver

Let us inspect the **Solver** XML tags.

```
<Solvers>
  <SinglePhaseFVM
    name="SinglePhaseFlow"
    discretization="singlePhaseTPFA"
    targetRegions="{ Reservoir }">
    <NonlinearSolverParameters
      newtonTol="1.0e-6"
      newtonMaxIter="8"/>
    <LinearSolverParameters
```

(continues on next page)

(continued from previous page)

```
      solverType="gmres"
      preconditionerType="amg"
      amgSmootherType="l1jacobi"
      krylovTol="1.0e-10"/>
  </SinglePhaseFVM>
</Solvers>
```

This node gathers all the information previously defined. We use a classical `SinglePhaseFVM` Finite Volume Method, with the two-point flux approximation as will be defined in the **NumericalMethods** tag. The `targetRegions` refers only to the Reservoir region because we only solve for flow in this region.

The `NonlinearSolverParameters` and `LinearSolverParameters` are used to set usual numerical solver parameters such as the linear and nonlinear tolerances, the preconditioner and solver types or the maximum number of nonlinear iterations.

Mesh

Here, we use the `VTKMesh` to load the mesh (see *Importing the Mesh*). The syntax to import external meshes is simple : in the XML file, the mesh `file` is included with its relative or absolute path to the location of the GEOS XML file and a user-specified `name` label for the mesh object.

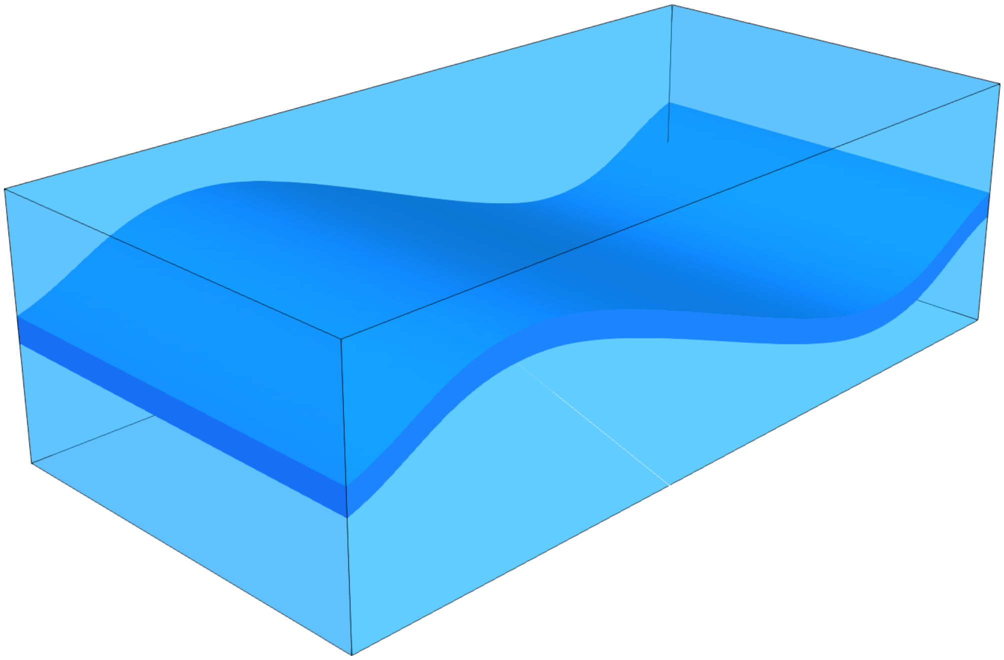
```
<Mesh>
  <VTKMesh name="SyntheticMesh"
           file="synthetic.vtu" />
</Mesh>
```

Geometry

> Here, we are using definition of `source` and `sink` boxes in addition to the `all` box in order to flag sets of nodes or cells which will act as injection or production.

```
<Geometry>
  <Box
    name="all"
    xMin="{ -1e9, -1e9, -1e9 }"
    xMax="{ 1e9, 1e9, 1e9 }"/>

  <Box
    name="source"
    xMin="{ 15500, 7000, -5000 }"
    xMax="{ 16000, 7500, 0 }"/>

  <Box
    name="sink"
    xMin="{ 6500, 1500, -5000 }"
    xMax="{ 7000, 2000, 0 }"/>
</Geometry>
```

In order to define a box, the user defines `xMax` and `xMin`, two diagonally opposite nodes of the box.



Events

The events are used here to guide the simulation through time, and specify when outputs must be triggered.

```
<Events maxTime="100.0e6">
  <PeriodicEvent name="solverApplications"
                 forceDt="10.0e6"
                 target="/Solvers/SinglePhaseFlow" />

  <PeriodicEvent name="outputs"
                 timeFrequency="10.0e6"
                 target="/Outputs/reservoir_with_properties" />
</Events>
```

The **Events** tag is associated with the `maxTime` keyword defining the maximum time. If this time is ever reached or exceeded, the simulation ends.

Two `PeriodicEvent` are defined. - The first one, `solverApplications`, is associated with the solver. The `forceDt` keyword means that there will always be time-steps of 10e6 seconds. - The second, `outputs`, is associated with the output. The `timeFrequency` keyword means that it will be executed every 10e6 seconds.

Numerical methods

Defining the numerical method used in the solver, we will provide information on how to discretize our equations. Here a classical two-point flux approximation (TPFA) scheme is used to discretize water fluxes over faces.

```
<NumericalMethods>
  <FiniteVolume>
    <TwoPointFluxApproximation
      name="singlePhaseTPFA"
    />
```

(continues on next page)

(continued from previous page)

```
  </FiniteVolume>
</NumericalMethods>
```

Regions

Assuming that the overburden and the underburden are impermeable, and flow only takes place in the reservoir, we need to define regions.

There are two methods to achieve this regional solve.

- The first solution is to define a unique `CellElementRegion` corresponding to the reservoir.

  ```
  <ElementRegions>
    <CellElementRegion
      name="Reservoir"
      cellBlocks="{2_tetrahedra}"
      materialList="{ water, rock, rockPerm, rockPorosity, nullSolid }"/>
  </ElementRegions>
  ```

- The second solution is to define all the `CellElementRegions` as they are in the VTK file, but defining the solvers only on the reservoir layer. In this case, the **ElementRegions** tag is :

  ```
    <ElementRegions>
      <CellElementRegion
        name="Reservoir"
        cellBlocks="{ 2_tetrahedra }"
        materialList="{ water, rock }"/>

      <CellElementRegion
        name="Burden"
        cellBlocks="{ 1_tetrahedra, 3_tetrahedra }"
        materialList="{ water, rock }"/>
    </ElementRegions>
  ```

We opt for the latest as it allows to visualize over- and underburdens and to change regions handling in their tag without needing to amend the **ElementRegion** tag.

Note: The material list here was set for a single-phase flow problem. This list is subject to change if the problem is not a single-phase flow problem.

Constitutive models

We simulate a single-phase flow in the reservoir layer, hence with multiple types of materials, a fluid (water) and solid (rock permeability and porosity).

```
<Constitutive>
  <CompressibleSinglePhaseFluid
    name="water"
    defaultDensity="1000"
    defaultViscosity="0.001"
```

(continues on next page)

(continued from previous page)

```
      referencePressure="0.0"
      compressibility="1e-9"
      viscosibility="0.0"/>

    <CompressibleSolidConstantPermeability
      name="rock"
      solidModelName="nullSolid"
      porosityModelName="rockPorosity"
      permeabilityModelName="rockPerm"/>

    <NullModel
      name="nullSolid"/>

    <PressurePorosity
      name="rockPorosity"
      defaultReferencePorosity="0.05"
      referencePressure="10e7"
      compressibility="1.0e-9"/>

    <ConstantPermeability
      name="rockPerm"
      permeabilityComponents="{ 1.0e-13, 1.0e-13, 1.0e-16 }"/>
  </Constitutive>
```

The constitutive parameters such as the density, the viscosity, and the compressibility are specified in the International System of Units.

Note: To consider an incompressible fluid, the user has to set the compressibility to 0.

Note: GEOS handles permeability as a diagonal matrix, so the three values of the permeability tensor are set individually using the `component` field.

Defining properties

The next step is to specify fields, including:

- The initial value (here, the pressure has to be initialized)
- The static properties (here, we have to define the permeability tensor and the porosity)
- The boundary conditions (here, the injection and production pressure have to be set)

```
  <FieldSpecifications>

    <FieldSpecification
       name="permx"
       initialCondition="1"
       component="0"
       setNames="{ all }"
       objectPath="ElementRegions/Reservoir"
```

(continues on next page)

(continued from previous page)

```
     fieldName="rockPerm_permeability"
     scale="1e-15"
     functionName="permxFunc"/>

   <FieldSpecification
     name="permy"
     initialCondition="1"
     component="1"
     setNames="{ all }"
     objectPath="ElementRegions/Reservoir"
     fieldName="rockPerm_permeability"
     scale="1e-15"
     functionName="permyFunc"/>

   <FieldSpecification
     name="permz"
     initialCondition="1"
     component="2"
     setNames="{ all }"
     objectPath="ElementRegions/Reservoir"
     fieldName="rockPerm_permeability"
     scale="3e-15"
     functionName="permzFunc"/>

  <FieldSpecification
    name="initialPressure"
    initialCondition="1"
    setNames="{ all }"
    objectPath="ElementRegions/Reservoir/2_tetrahedra"
    fieldName="pressure"
    scale="1e7"
    />

  <FieldSpecification
    name="sourceTerm"
    objectPath="ElementRegions/Reservoir/2_tetrahedra"
    fieldName="pressure"
    scale="15e7"
    setNames="{ source }"
    />

  <FieldSpecification
    name="sinkTerm"
    objectPath="ElementRegions/Reservoir/2_tetrahedra"
    fieldName="pressure"
    scale="5e7"
    setNames="{ sink }"/>
</FieldSpecifications>
```

You may note :

- All static parameters and initial value fields must have `initialCondition` field set to 1.
- The `objectPath` refers to the `ElementRegion` in which the field has its value,

- The `setName` field points to the box previously defined to apply the fields,
- `name` and `fieldName` have a different meaning: `name` is used to give a name to the XML block. This `name` must be unique. `fieldName` is the name of the field registered in GEOS. This value has to be set according to the expected input fields of each solver.

Output

The **Outputs** XML tag is used to trigger the writing of visualization files. Here, we write files in a format natively readable by Paraview under the tag *VTK*:

```
<Outputs>
  <VTK
    name="reservoir_with_properties"/>
</Outputs>
```

Note: The `name` keyword defines the name of the output directory.

Using functions to specify properties

Eventually, one can define varying properties using `TableFunction` (*Functions*) under the **Functions** tag:

```
<Functions>
  <TableFunction
    name="timeInj"
    inputVarNames="{ time }"
    coordinates="{ 1e6, 10e6, 50e6 }"
    values="{ 1, 0.01, 0.00001 }"/>

  <TableFunction
    name="initialPressureFunc"
    inputVarNames="{ elementCenter }"
    coordinateFiles="{ tables_FieldCaseTuto/xlin.geos, tables_FieldCaseTuto/ylin.geos,␣
↪tables_FieldCaseTuto/zlin.geos }"
    voxelFile="tables_FieldCaseTuto/pressure.geos"/>

  <TableFunction
    name="permxFunc"
    inputVarNames="{ elementCenter }"
    coordinateFiles="{ tables_FieldCaseTuto/xlin.geos, tables_FieldCaseTuto/ylin.geos,␣
↪tables_FieldCaseTuto/zlin.geos }"
    voxelFile="tables_FieldCaseTuto/permx.geos"
    interpolation="nearest"/>

  <TableFunction
    name="permyFunc"
    inputVarNames="{ elementCenter }"
    coordinateFiles="{ tables_FieldCaseTuto/xlin.geos, tables_FieldCaseTuto/ylin.geos,␣
↪tables_FieldCaseTuto/zlin.geos }"
    voxelFile="tables_FieldCaseTuto/permy.geos"
    interpolation="nearest"/>
```

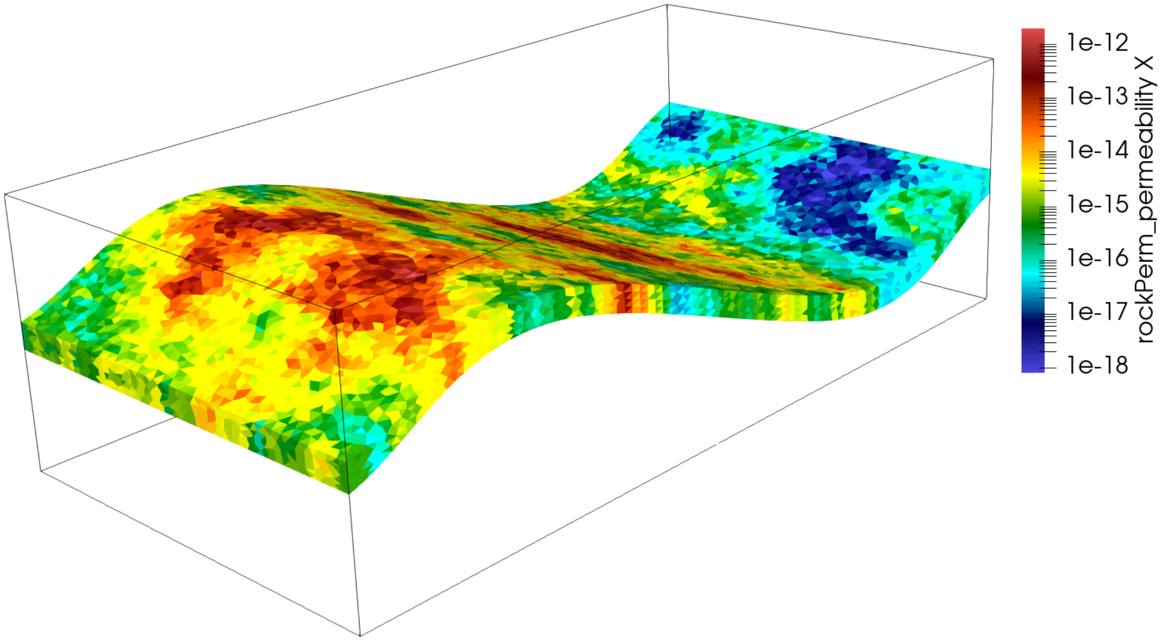
(continues on next page)

(continued from previous page)

```
    <TableFunction
      name="permzFunc"
      inputVarNames="{ elementCenter }"
      coordinateFiles="{ tables_FieldCaseTuto/xlin.geos, tables_FieldCaseTuto/ylin.geos,␣
↪tables_FieldCaseTuto/zlin.geos }"
      voxelFile="tables_FieldCaseTuto/permz.geos"
      interpolation="nearest"/>
  </Functions>
```

Here, the injection pressure is set to vary with time. Attentive reader might have noticed that `sourceTerm` was bound to a `TableFunction` named *timeInj* under **FieldSpecifications** tag definition. The initial pressure is set based on the values contained in the table formed by the files which are specified. In particular, the files *xlin.geos*, *ylin.geos* and *zlin.geos* define a regular meshing of the bounding box containing the reservoir. The *pressure.geos* file then defines the values of the pressure at those points.

We proceed in a similar manner as for *pressure.geos* to map a heterogeneous permeability field (here the 5th layer of the SPE 10 test case) onto our unstructured grid. This mapping will use a nearest point interpolation rule.



Note: The varying values imposed in *values* or passed through *voxelFile* are premultiplied by the *scale* attribute from **FieldSpecifications**.

Running GEOS

The simulation can be launched with:

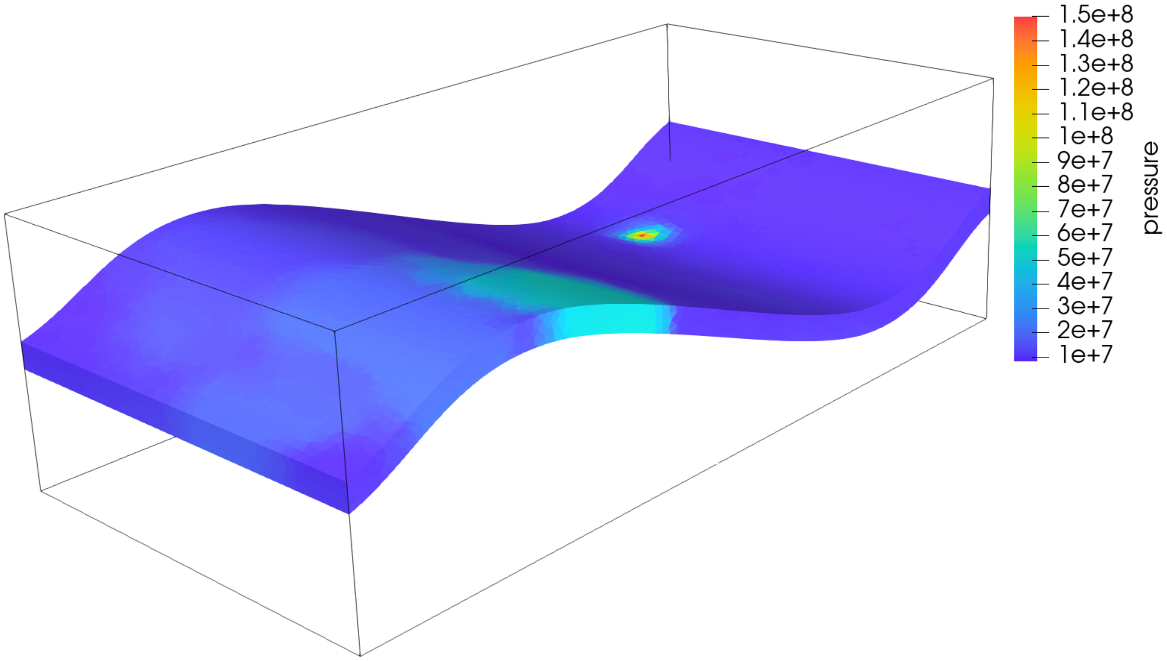
```
geosx -i FieldCaseTutorial3_smoke.xml
```

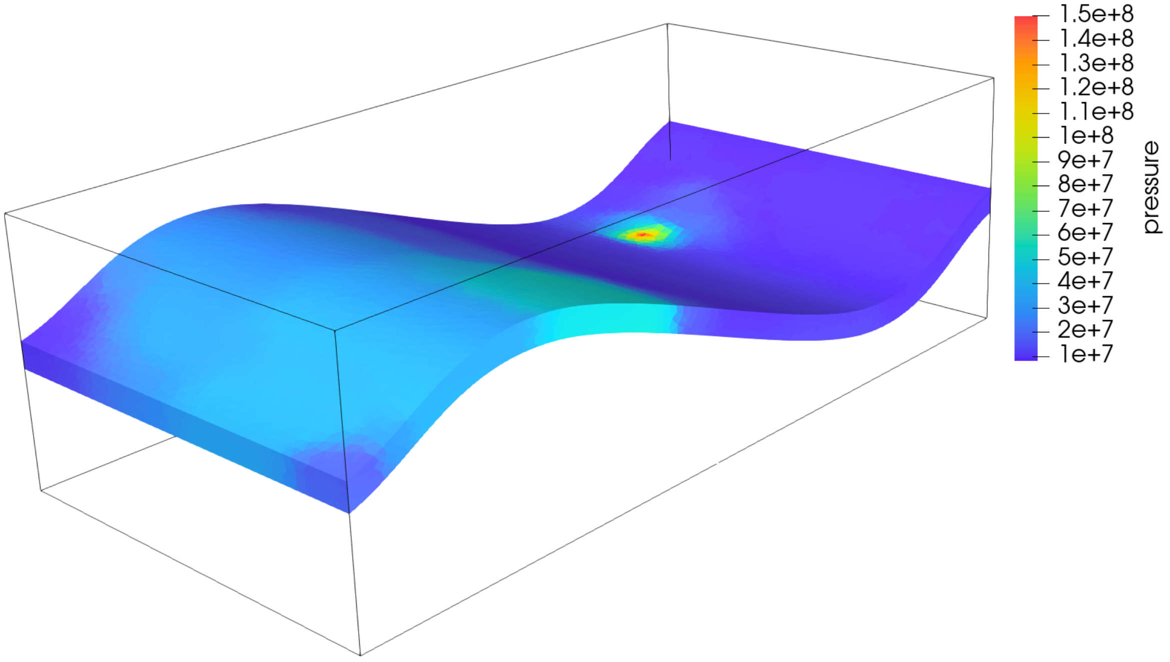
One can notice the correct load of the field function among the starting output messages

```
Adding Mesh: VTKMesh, SyntheticMesh
Adding Event: PeriodicEvent, solverApplications
Adding Event: PeriodicEvent, outputs
Adding Solver of type SinglePhaseFVM, named SinglePhaseFlow
Adding Geometric Object: Box, all
Adding Geometric Object: Box, source
Adding Geometric Object: Box, sink
Adding Output: VTK, reservoir_with_properties
   TableFunction: timeInj
   TableFunction: initialPressureFunc
   TableFunction: permxFunc
   TableFunction: permyFunc
   TableFunction: permzFunc
Adding Object CellElementRegion named Reservoir from ObjectManager::Catalog.
Adding Object CellElementRegion named Burden from ObjectManager::Catalog.
```

Visualization of results

We can open the file *syntheticReservoirVizFile.pvd* with Paraview to visualize the simulation results. In the event block, we have asked for the output to be generated at regular intervals throughout the simulation, we can thus visualize the pressure distribution at different simulation times, showing the variation in the injection control.





To go further

Feedback on this tutorial

This concludes this tutorial. For any feedback, please submit a GitHub issue on the project's GitHub page.

For more details

- More on meshes, please see *Meshes*.
- More on events, please see *Event Management*.

1.2.4 Tutorial 4: Boundary Conditions and Time-Dependent Functions

Context

In this tutorial, we use a small strain linear elastic based solid mechanics solver (see *Solid Mechanics Solver*) from GEOS to solve for the bending problem of a three-dimensional cantilever beam. The beam is fixed at one end, and subjects to a traction force pointing to the y-positive direction on the other end. The beam is deformed in the x-y plane.

Objectives

At the end of this tutorial, you will know:

- how to use the solid mechanics solver to solve a quasistatic problem,
- how to set up displacement boundary condition at element nodes,
- how to set up traction boundary condition on element surfaces,
- how to use a table function to control time-dependent loading.

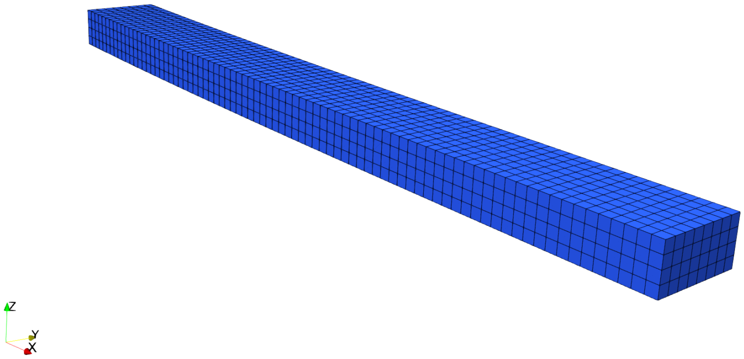
Input file

This tutorial uses no external input files and everything required is contained within a single GEOS input file. The xml input file for this test case is located at:

```
inputFiles/solidMechanics/beamBending_base.xml
inputFiles/solidMechanics/beamBending_benchmark.xml
```

Discretized computational domain

The following mesh is used in this tutorial:



This mesh contains 80 x 8 x 4 eight-node brick elements in the x, y and z directions, respectively. Here, the `InternalMesh` is used to generate a structured three-dimensional mesh with `C3D8` as the `elementTypes`. This mesh is defined as a cell block with the name `cb1`.

```
<Mesh>
  <InternalMesh
    name="mesh1"
    elementTypes="{ C3D8 }"
    xCoords="{ 0, 80 }"
    yCoords="{ 0, 8 }"
    zCoords="{ 0, 4 }"
    nx="{ 160 }"
    ny="{ 16 }"
    nz="{ 8 }"
    cellBlockNames="{ cb1 }"/>
</Mesh>
```

Gravity

The gravity is turned off explicitly at the beginning of the input file:

```
<Solvers
  gravityVector="{ 0.0, 0.0, 0.0 }">
```

Solid mechanics solver

The solid mechanics solver is based on the small strain Lagrangian finite element formulation. The problem is run as QuasiStatic without considering the beam inertial. The computational domain is discretized by FE1, which is defined in the NumericalMethods block. The material is designated as shale, whose properties are defined in the Constitutive block.

```
    <SolidMechanicsLagrangianSSLE
      name="lagsolve"
      timeIntegrationOption="QuasiStatic"
      discretization="FE1"
      targetRegions="{ Region2 }"
      logLevel="1">
```

Finite element discretization

The computational domain is discretized by C3D8 elements with the first order interpolation functions at each direction in the parent domain. The 2 x 2 x 2 Gauss quadrature rule is adopted to be compatible with the first order interpolation functions.

Constitutive model

Recall that in the SolidMechanicsLagrangianSSLE block, shale is designated as the material in the computational domain. Here, the material is defined as linear isotropic.

```
    <ElasticIsotropic
      name="shale"
      defaultDensity="2700"
      defaultBulkModulus="5.5556e9"
      defaultShearModulus="4.16667e9"/>
```

Boundary conditions

As aforementioned, the beam is fixed on one end, and subjects to surface traction on the other end. These boundary conditions are set up through the FieldSpecifications block. Here, nodeManager and faceManager in the objectPath indicate that the boundary conditions are applied to the element nodes and faces, respectively. Component 0, 1, and 2 refer to the x, y, and z direction, respectively. And the non-zero values given by Scale indicate the magnitude of the loading. Some shorthands, such as xneg and xpos, are used as the locations where the boundary conditions are applied in the computational domain. For instance, xneg means the portion of the computational domain located at the left-most in the x-axis, while xpos refers to the portion located at the right-most area in the x-axis. Similar shorthands include ypos, yneg, zpos, and zneg. Particularly, the time-dependent loading applied at the beam tip is defined through a function with the name timeFunction.

```
  <FieldSpecifications>
    <FieldSpecification
      name="xnegconstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="0"
      scale="0.0"
      setNames="{ xneg }"/>
```

(continues on next page)

(continued from previous page)

```
    <FieldSpecification
      name="yconstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="1"
      scale="0.0"
      setNames="{ xneg }"/>

    <FieldSpecification
      name="zconstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="2"
      scale="0.0"
      setNames="{ zneg, zpos }"/>

    <Traction
      name="xposconstraint"
      objectPath="faceManager"
      scale="1.0e6"
      direction="{ 0, 1, 0 }"
      functionName="timeFunction"
      setNames="{ xpos }"/>
  </FieldSpecifications>
```

Table function

A table function is used to define the time-dependent loading at the beam tip. The `coordinates` and `values` form a time-magnitude pair for the loading time history. In this case, the loading magnitude increases linearly as the time evolves.

```
  <Functions>
    <TableFunction
      name="timeFunction"
      inputVarNames="{ time }"
      coordinates="{ 0.0, 10.0 }"
      values="{ 0.0, 10.0 }"/>
  </Functions>
```

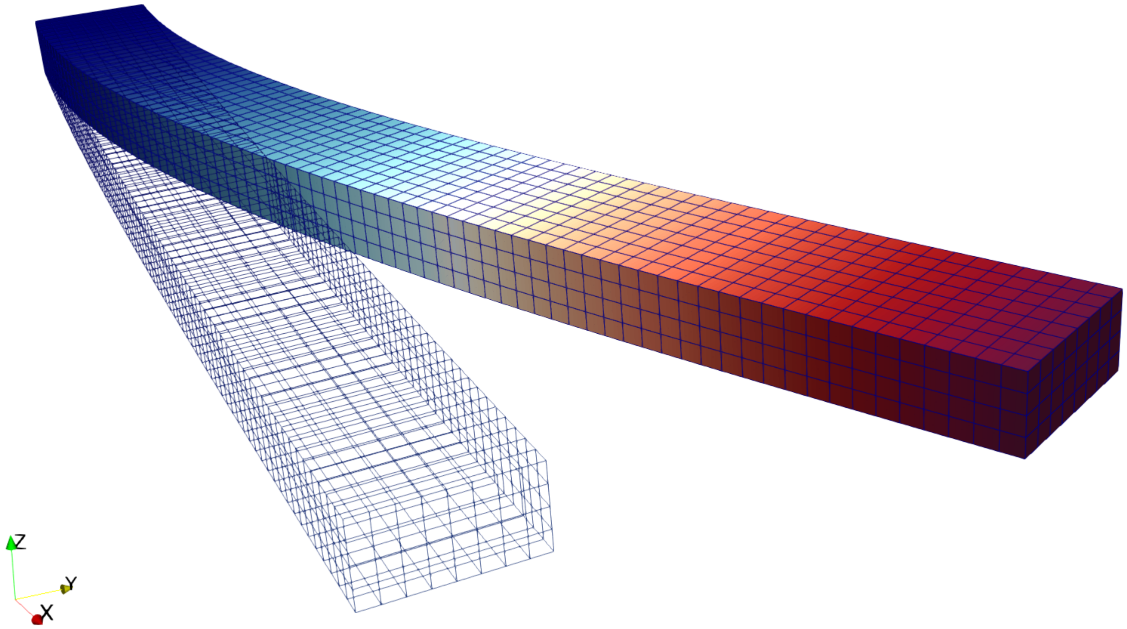
Execution

Finally, the execution of the simulation is set up in the `Events` block, where `target` points to the solid mechanics solver defined in the `Solvers` block, and the time increment `forceDt` is set as 1.0s.

```
    <PeriodicEvent
      name="solverApplications"
      forceDt="1.0"
      target="/Solvers/lagsolve"/>
```

Result

The deformed beam is shown as following (notice that the displacement is visually magnified):



To go further

Feedback on this tutorial

This concludes the solid mechanics for small-strain linear elasticity tutorial. For any feedback on this tutorial, please submit a GitHub issue on the project's GitHub page.

For more details

- More on meshes, please see *Meshes*.
- More on events, please see *Event Management*.

1.3 Basic Examples

1.3.1 Multiphase Flow

Context

In this example, we set up a multiphase, multicomponent test case (see *Compositional Multiphase Flow Solver*). The permeability field corresponds to the two bottom layers (layers 84 and 85) of the SPE10 test case. The thermodynamic behavior of the fluid mixture is specified using a simple immiscible two-phase (Dead-Oil) model. Injection and production are simulated using boundary conditions.

Objective

The main objective of this example is to review the main elements of a simple two-phase simulation in GEOS, including:

- the compositional multiphase flow solver,
- the multiphase constitutive models,

- the specifications of multiphase boundary conditions.

Input file

This example is based on the XML file located at

```
inputFiles/compositionalMultiphaseFlow/benchmarks/SPE10/deadOilSpe10Layers84_85_base_
↪iterative.xml
```

The XML file considered here follows the typical structure of the GEOS input files:

1. *Solver*
2. *Mesh*
3. *Geometry*
4. *Events*
5. *NumericalMethods*
6. *ElementRegions*
7. *Constitutive*
8. *FieldSpecifications*
9. *Outputs*

Multiphase flow solver

In GEOS, the setup of a multiphase simulation starts in the **Solvers** XML block of the input file. This example relies on a solver of type **CompositionalMultiphaseFVM** that implements a fully implicit finite-volume scheme based on the standard two-point approximation of the flux (TPFA). More information on this solver can be found at *Compositional Multiphase Flow Solver*.

Let us have a closer look at the **Solvers** XML block displayed below. The solver has a name (here, `compflow`) that can be chosen by the user and is not imposed by GEOS. Note that this name is used in the **Events** XML block to trigger the application of the solver. Using the `targetRegions` attribute, the solver defines the target regions on which it is applied. In this example, there is only one region, named `reservoir`.

The **CompositionalMultiphaseFVM** block contains two important sub-blocks, namely **NonlinearSolverParameters** and **LinearSolverParameters**. In **NonlinearSolverParameters**, one can finely tune the nonlinear tolerance, the application of the linear search algorithm, and the heuristics used to increase the time step size. In **LinearSolverParameters**, the user can specify the linear tolerance, the type of (direct or iterative) linear solver, and the type of preconditioner, if any. For large multiphase flow problems, we recommend using an iterative linear solver (`solverType="gmres"` or `solverType="fgmres"`) combined with the multigrid reduction (MGR) preconditioner (`preconditionerType="mgr"`). More information about the MGR preconditioner can be found in *Linear Solvers*.

Note: For non-trivial simulations, we recommend setting the `initialDt` attribute to a small value (relative to the time scale of the problem) in seconds. If the simulation appears to be slow, use `logLevel="1"` in **CompositionalMultiphaseFVM** to detect potential Newton convergence problems. If the Newton solver struggles, please set `lineSearchAction="Attempt"` in **NonlinearSolverParameters**. If the Newton convergence is good, please add `logLevel="1"` in the **LinearSolverParameters** block to detect linear solver problems, especially if an iterative linear solver is used.

Note: To use the linear solver options of this example, you need to ensure that GEOS is configured to use the Hypre linear solver package.

```
<Solvers>

  <CompositionalMultiphaseFVM
    name="compflow"
    logLevel="1"
    discretization="fluidTPFA"
    targetRegions="{ reservoir }"
    temperature="300"
    useMass="1"
    initialDt="1e3"
    maxCompFractionChange="0.1">
    <NonlinearSolverParameters
      newtonTol="1.0e-4"
      newtonMaxIter="40"
      maxTimeStepCuts="10"
      lineSearchAction="None"/>
    <LinearSolverParameters
      solverType="fgmres"
      preconditionerType="mgr"
      krylovTol="1.0e-5"/>
  </CompositionalMultiphaseFVM>

</Solvers>
```

Mesh

In this simulation, we define a simple mesh generated internally using the **InternalMesh** generator, as illustrated in the previous examples. The mesh dimensions and cell sizes are chosen to be those specified in the SPE10 test case, but are limited to the two bottom layers. The mesh description must be done in meters.

```
<Mesh>
  <InternalMesh
    name="mesh"
    elementTypes="{ C3D8 }"
    xCoords="{ 0, 365.76 }"
    yCoords="{ 0, 670.56 }"
    zCoords="{ 0, 1.22 }"
    nx="{ 60 }"
    ny="{ 220 }"
    nz="{ 2 }"
    cellBlockNames="{ block }"/>
</Mesh>
```

Geometry

As in the previous examples, the **Geometry** XML block is used to select the cells in which the boundary conditions are applied. To mimic the setup of the original SPE10 test case, we place a source term in the middle of the domain, and a sink term in each corner. The specification of the boundary conditions applied to the selected mesh cells is done in the **FieldSpecifications** block of the XML file using the names of the boxes defined here.

```
<Geometry>

  <Box
    name="source"
    xMin="{ 182.85, 335.25, -0.01 }"
    xMax="{ 189.00, 338.35, 2.00 }"/>
  <Box
    name="sink1"
    xMin="{ -0.01, -0.01, -0.01 }"
    xMax="{ 6.126, 3.078, 2.00 }"/>
  <Box
    name="sink2"
    xMin="{ -0.01, 667.482, -0.01 }"
    xMax="{ 6.126, 670.60, 2.00 }"/>
  <Box
    name="sink3"
    xMin="{ 359.634, -0.01, -0.01 }"
    xMax="{ 365.8, 3.048, 2.00 }"/>
  <Box
    name="sink4"
    xMin="{ 359.634, 667.482, -0.01 }"
    xMax="{ 365.8, 670.60, 2.00 }"/>

</Geometry>
```

Events

In the **Events** XML block of this example, we specify two types of **PeriodicEvents** serving different purposes, namely solver application and result output.

The periodic event named `solverApplications` triggers the application of the solver on its target region. This event must point to the solver by name. In this example, the name of the solver is `compflow` and was defined in the **Solvers** block. The time step is initialized using the `initialDt` attribute of the flow solver. Then, if the solver converges in less than a certain number of nonlinear iterations (by default, 40% of the maximum number of nonlinear iterations), the time step will be increased until it reaches the maximum time step size specified with `maxEventDt`. If the time step fails, the time step will be cut. The parameters defining the time stepping strategy can be finely tuned by the user in the flow solver block. Note that all times are in seconds.

The output event forces GEOS to write out the results at the frequency specified by the attribute `timeFrequency`. Here, we choose to output the results using the VTK format (see *Tutorial 2: External Meshes* for a tutorial that uses the Silo output file format). Using `targetExactTimestep=1` in this XML block forces GEOS to adapt the time stepping to ensure that an output is generated exactly at the time frequency requested by the user. In the `target` attribute, we must use the name defined in the **VTK** XML tag inside the **Output** XML section, as documented at the end of this example (here, `vtkOutput`).

More information about events can be found at *Event Management*.

```
<Events
  maxTime="2e6">

 <PeriodicEvent
    name="outputs"
    timeFrequency="5e5"
    targetExactTimestep="1"
    target="/Outputs/vtkOutput"/>

  <PeriodicEvent
    name="solverApplications"
    maxEventDt="5e5"
    target="/Solvers/compflow"/>

  <PeriodicEvent
    name="restarts"
    timeFrequency="1e6"
    targetExactTimestep="0"
    target="/Outputs/restartOutput"/>

</Events>
```

Numerical methods

In the **NumericalMethods** XML block, we select a two-point flux approximation (TPFA) finite-volume scheme to discretize the governing equations on the reservoir mesh. TPFA is currently the only numerical scheme that can be used with a flow solver of type **CompositionalMultiphaseFVM**.

```
<NumericalMethods>
  <FiniteVolume>
    <TwoPointFluxApproximation
      name="fluidTPFA"/>
  </FiniteVolume>
</NumericalMethods>
```

Reservoir region

In the **ElementRegions** XML block, we define a **CellElementRegion** named `reservoir` corresponding to the reservoir mesh. The attribute `cellBlocks` is set to `block` to point this element region to the hexahedral mesh defined internally.

The **CellElementRegion** must also point to the constitutive models that are used to update the dynamic rock and fluid properties in the cells of the reservoir mesh. The names `fluid`, `rock`, and `relperm` used for this in the `materialList` correspond to the attribute `name` of the **Constitutive** block.

```
<ElementRegions>
  <CellElementRegion
    name="reservoir"
    cellBlocks="{ block }"
    materialList="{ fluid, rock, relperm }"/>
</ElementRegions>
```

Constitutive models

For a simulation performed with the **CompositionalMultiphaseFVM** physics solver, at least four types of constitutive models must be specified in the **Constitutive** XML block:

- a fluid model describing the thermodynamics behavior of the fluid mixture,
- a relative permeability model,
- a rock permeability model,
- a rock porosity model.

All these models use SI units exclusively. A capillary pressure model can also be specified in this block but is omitted here for simplicity.

Here, we introduce a fluid model describing a simplified mixture thermodynamic behavior. Specifically, we use an immiscible two-phase (Dead Oil) model by placing the XML tag **DeadOilFluid**. Other fluid models can be used with the **CompositionalMultiphaseFVM** solver, as explained in *Fluid Models*.

With the tag **BrooksCoreyRelativePermeability**, we define a relative permeability model. A list of available relative permeability models can be found at *Relative Permeability Models*.

The properties are chosen to match those of the original SPE10 test case.

Note: The names and order of the phases listed for the attribute `phaseNames` must be identical in the fluid model (here, **DeadOilFluid**) and the relative permeability model (here, **BrooksCoreyRelativePermeability**). Otherwise, GEOS will throw an error and terminate.

We also introduce models to define rock compressibility and permeability. This step is similar to what is described in the previous examples (see for instance *Tutorial 1: First Steps*).

We remind the reader that the attribute `name` of the constitutive models defined here must be used in the **ElementRegions** and **Solvers** XML blocks to point the element regions and the physics solvers to their respective constitutive models.

```xml
<Constitutive>

  <DeadOilFluid
    name="fluid"
    phaseNames="{ oil, water }"
    surfaceDensities="{ 800.0, 1022.0 }"
    componentMolarWeight="{ 114e-3, 18e-3 }"
    hydrocarbonFormationVolFactorTableNames="{ B_o_table }"
    hydrocarbonViscosityTableNames="{ visc_o_table }"
    waterReferencePressure="30600000.1"
    waterFormationVolumeFactor="1.03"
    waterCompressibility="0.00000000041"
    waterViscosity="0.0003"/>

  <CompressibleSolidConstantPermeability
    name="rock"
    solidModelName="nullSolid"
    porosityModelName="rockPorosity"
    permeabilityModelName="rockPerm"/>

  <NullModel
```

(continues on next page)

(continued from previous page)

```
      name="nullSolid"/>

    <PressurePorosity
      name="rockPorosity"
      defaultReferencePorosity="0.1"
      referencePressure="1.0e7"
      compressibility="1e-10"/>

    <BrooksCoreyRelativePermeability
      name="relperm"
      phaseNames="{ oil, water }"
      phaseMinVolumeFraction="{ 0.0, 0.0 }"
      phaseRelPermExponent="{ 2.0, 2.0 }"
      phaseRelPermMaxValue="{ 1.0, 1.0 }"/>

    <ConstantPermeability
      name="rockPerm"
      permeabilityComponents="{ 1.0e-14, 1.0e-14, 1.0e-18 }"/>

  </Constitutive>
```

Initial and boundary conditions

In the **FieldSpecifications** section, we define the initial and boundary conditions as well as the geological properties (porosity, permeability). All this is done using SI units. Here, we focus on the specification of the initial and boundary conditions for a simulation performed with the **CompositionalMultiphaseFVM** solver. We refer to *Tutorial 1: First Steps* for a more general discussion on the **FieldSpecification** XML blocks.

For a simulation performed with the **CompositionalMultiphaseFVM** solver, we have to set the initial pressure as well as the initial global component fractions (in this case, the oil and water component fractions). The `component` attribute of the **FieldSpecification** XML block must use the order in which the `phaseNames` have been defined in the **DeadOilFluid** XML block. In other words, `component=0` is used to initialize the oil global component fraction and `component=1` is used to initialize the water global component fraction, because we previously set `phaseNames="{oil, water}"` in the **DeadOilFluid** XML block.

To specify the sink terms, we use the **FieldSpecification** mechanism in a similar fashion to impose the sink pressure and composition. This is done to mimic a pressure-controlled well (before breakthrough). To specify the source term, we use a **SourceFlux** block to impose a fixed mass injection rate of component 1 (water) to mimic a rate-controlled well.

```
  <FieldSpecifications>
    <FieldSpecification
      name="permx"
      component="0"
      initialCondition="1"
      setNames="{ all }"
      objectPath="ElementRegions/reservoir/block"
      fieldName="rockPerm_permeability"
      functionName="permxFunc"
      scale="9.869233e-16"/>
    <FieldSpecification
      name="permy"
```

(continues on next page)

(continued from previous page)

```
      component="1"
      initialCondition="1"
      setNames="{ all }"
      objectPath="ElementRegions/reservoir/block"
      fieldName="rockPerm_permeability"
      functionName="permyFunc"
      scale="9.869233e-16"/>
    <FieldSpecification
      name="permz"
      component="2"
      initialCondition="1"
      setNames="{ all }"
      objectPath="ElementRegions/reservoir/block"
      fieldName="rockPerm_permeability"
      functionName="permzFunc"
      scale="9.869233e-16"/>

    <FieldSpecification
      name="referencePorosity"
      initialCondition="1"
      setNames="{ all }"
      objectPath="ElementRegions/reservoir/block"
      fieldName="rockPorosity_referencePorosity"
      functionName="poroFunc"
      scale="1.0"/>

    <FieldSpecification
      name="initialPressure"
      initialCondition="1"
      setNames="{ all }"
      objectPath="ElementRegions/reservoir/block"
      fieldName="pressure"
      scale="4.1369e+7"/>
    <FieldSpecification
      name="initialComposition_oil"
      initialCondition="1"
      setNames="{ all }"
      objectPath="ElementRegions/reservoir/block"
      fieldName="globalCompFraction"
      component="0"
      scale="0.9995"/>
    <FieldSpecification
      name="initialComposition_water"
      initialCondition="1"
      setNames="{ all }"
      objectPath="ElementRegions/reservoir/block"
      fieldName="globalCompFraction"
      component="1"
      scale="0.0005"/>

    <SourceFlux
      name="sourceTerm"
```

(continues on next page)

(continued from previous page)

```
    objectPath="ElementRegions/reservoir/block"
    scale="-0.07279"
    component="1"
    setNames="{ source }"/>

  <FieldSpecification
    name="sinkPressure"
    setNames="{ sink1, sink2, sink3, sink4 }"
    objectPath="ElementRegions/reservoir/block"
    fieldName="pressure"
    scale="2.7579e+7"/>
  <FieldSpecification
    name="sinkComposition_oil"
    setNames="{ sink1, sink2, sink3, sink4 }"
    objectPath="ElementRegions/reservoir/block"
    fieldName="globalCompFraction"
    component="0"
    scale="0.9995"/>
  <FieldSpecification
    name="sinkComposition_water"
    setNames="{ sink1, sink2, sink3, sink4 }"
    objectPath="ElementRegions/reservoir/block"
    fieldName="globalCompFraction"
    component="1"
    scale="0.0005"/>

</FieldSpecifications>
```

Output

In this section, we request an output of the results in VTK format. Note that the name defined here must match the names used in the **Events** XML block to define the output frequency.

```
<Outputs>
  <VTK
    name="vtkOutput"/>

  <Restart
    name="restartOutput"/>

</Outputs>
```

All elements are now in place to run GEOS.

Running GEOS

The first few lines appearing to the console are indicating that the XML elements are read and registered correctly:

```
Adding Solver of type CompositionalMultiphaseFVM, named compflow
Adding Mesh: InternalMesh, mesh
Adding Geometric Object: Box, source
Adding Geometric Object: Box, sink1
Adding Geometric Object: Box, sink2
Adding Geometric Object: Box, sink3
Adding Geometric Object: Box, sink4
Adding Event: PeriodicEvent, outputs
Adding Event: PeriodicEvent, solverApplications
TableFunction: permxFunc
TableFunction: permyFunc
TableFunction: permzFunc
TableFunction: poroFunc
TableFunction: B_o_table
TableFunction: visc_o_table
Adding Output: VTK, vtkOutput
Adding Object CellElementRegion named region from ObjectManager::Catalog.
region/block/fluid is allocated with 1 quadrature points.
region/block/rock is allocated with 1 quadrature points.
aaregion/block/relperm is allocated with 1 quadrature points.
```

At this point, we are done with the case set-up and the code steps into the execution of the simulation itself:

```
Time: 0s, dt:1000s, Cycle: 0

  Attempt:  0, NewtonIter:  0
  ( Rfluid ) = (2.28e+00) ;     ( R ) = ( 2.28e+00 ) ;
  Attempt:  0, NewtonIter:  1
  ( Rfluid ) = (8.83e-03) ;     ( R ) = ( 8.83e-03 ) ;
  Last LinSolve(iter,res) = (   2, 2.74e-03 ) ;
  Attempt:  0, NewtonIter:  2
  ( Rfluid ) = (8.86e-05) ;     ( R ) = ( 8.86e-05 ) ;
  Last LinSolve(iter,res) = (   2, 8.92e-03 ) ;

compflow: Max phase CFL number: 0.00399585
compflow: Max component CFL number: 0.152466
compflow: Newton solver converged in less than 16 iterations, time-step required will be␣
↪doubled.
```

Visualization

A file compatible with Paraview is produced in this example. It is found in the output folder, and usually has the extension *.pvd*. More details about this file format can be found here. We can load this file into Paraview directly and visualize results:

To go further

Feedback on this example

This concludes the example on setting up an immiscible two-phase flow simulation in a channelized permeability field. For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

For more details

- A complete description of the reservoir flow solver is found here: *Compositional Multiphase Flow Solver*.
- The available constitutive models are listed at *Constitutive Models*.

1.3.2 Multiphase Flow with Wells

Context

In this example, we build on the concepts presented in *Multiphase Flow* to show how to set up a multiphase water injection problem with wells in the three-dimensional Egg model. The twelve wells (four producers and eight injectors) are placed according to the description of the original test case.

Objectives

In this example, we re-use many GEOS features already presented in *Multiphase Flow*, but we now focus on:

- how to import an external mesh with embedded geological properties (permeability) in the VTK format (`.vtu`),
- how to set up the wells.

Input file

This example is based on the XML file located at

```
../../../../../inputFiles/compositionalMultiphaseWell/benchmarks/Egg/deadOilEgg_
↪benchmark.xml
```

The mesh file corresponding to the Egg model is stored in the GEOSXDATA repository. Therefore, you must first download the GEOSXDATA repository in the same folder as the GEOS repository to run this test case.

Note: GEOSXDATA is a separate repository in which we store large mesh files in order to keep the main GEOS repository lightweight.

The XML file considered here follows the typical structure of the GEOS input files:

1. *Solver*
2. *Mesh*
3. *Events*
4. *NumericalMethods*
5. *ElementRegions*
6. *Constitutive*
7. *FieldSpecifications*
8. *Outputs*
9. *Tasks*

Coupling the flow solver with wells

In GEOS, the simulation of reservoir flow with wells is set up by combining three solvers listed and parameterized in the **Solvers** XML block of the input file. We introduce separately a flow solver and a well solver acting on different regions of the domain—respectively, the reservoir region and the well regions. To drive the simulation and bind these single-physics solvers, we also specify a *coupling solver* between the reservoir flow solver and the well solver. This coupling of single-physics solvers is the generic approach used in GEOS to define multiphysics problems. It is illustrated in *Poromechanics* for a poroelastic test case.

The three solvers employed in this example are:

- the single-physics reservoir flow solver, a solver of type **CompositionalMultiphaseFVM** named `compositionalMultiphaseFlow` (more information on this solver at *Compositional Multiphase Flow Solver*),
- the single-physics well solver, a solver of type **CompositionalMultiphaseWell** named `compositionalMultiphaseWell` (more information on this solver at *Compositional Multiphase Well Solver*),
- the coupling solver that binds the two single-physics solvers above, an object of type **CompositionalMultiphaseReservoir** named `coupledFlowAndWells`.

The **Solvers** XML block is shown below. The coupling solver points to the two single-physics solvers using the attributes `flowSolverName` and `wellSolverName`. These names can be chosen by the user and are not imposed by GEOS. The flow solver is applied to the reservoir and the well solver is applied to the wells, as specified by their respective `targetRegions` attributes.

The simulation is fully coupled and driven by the coupled solver. Therefore, the time stepping information (here, `initialDt`, but there may be other parameters used to fine-tune the time stepping strategy), the nonlinear solver parameters, and the linear solver parameters must be specified at the level of the coupling solver. There is no need to specify these parameters at the level of the single-physics solvers. Any solver information specified in the single-physics XML blocks will not be taken into account.

Note: It is worth repeating the `logLevel="1"` parameter at the level of the well solver to make sure that a notification is issued when the well control is switched (from rate control to BHP control, for instance).

Here, we instruct GEOS to perform at most `newtonMaxIter = "10"` Newton iterations. GEOS will adjust the time step size as follows:

- if the Newton solver converges in `timeStepIncreaseIterLimit x newtonMaxIter = 5` iterations or fewer, GEOS will double the time step size for the next time step,
- if the Newton solver converges in `timeStepDecreaseIterLimit x newtonMaxIter = 8` iterations or more, GEOS will reduce the time step size for the next time step by a factor `timestepCutFactor = 0.1`,
- if the Newton solver fails to converge in `newtonMaxIter = 10`, GEOS will cut the time step size by a factor `timestepCutFactor = 0.1` and restart from the previous converged time step.

The maximum number of time step cuts is specified by the attribute `maxTimeStepCuts`. Note that a backtracking line search can be activated by setting the attribute `lineSearchAction` to `Attempt` or `Require`. If `lineSearchAction = "Attempt"`, we accept the nonlinear iteration even if the line search does not reduce the residual norm. If `lineSearchAction = "Require"`, we cut the time step if the line search does not reduce the residual norm.

Note: To use the linear solver options of this example, you need to ensure that GEOS is configured to use the Hypre linear solver package.

```
  <Solvers>
    <CompositionalMultiphaseReservoir
      name="coupledFlowAndWells"
      flowSolverName="compositionalMultiphaseFlow"
      wellSolverName="compositionalMultiphaseWell"
      logLevel="1"
      initialDt="1e4"
      targetRegions="{ reservoir, wellRegion1, wellRegion2, wellRegion3, wellRegion4,␣
↪wellRegion5, wellRegion6, wellRegion7, wellRegion8, wellRegion9, wellRegion10,␣
↪wellRegion11, wellRegion12 }">
      <NonlinearSolverParameters
        newtonTol="1.0e-4"
        newtonMaxIter="25"
        timeStepDecreaseIterLimit="0.9"
        timeStepIncreaseIterLimit="0.6"
        timeStepCutFactor="0.1"
        maxTimeStepCuts="10"
        lineSearchAction="None"/>
      <LinearSolverParameters
        solverType="fgmres"
        preconditionerType="mgr"
        krylovTol="1e-4"
        krylovAdaptiveTol="1"
        krylovWeakestTol="1e-2"/>
    </CompositionalMultiphaseReservoir>

    <CompositionalMultiphaseFVM
      name="compositionalMultiphaseFlow"
      targetRegions="{ reservoir }"
      discretization="fluidTPFA"
      temperature="297.15"
      maxCompFractionChange="0.3"
      logLevel="1"
      useMass="1"/>

    <CompositionalMultiphaseWell
      name="compositionalMultiphaseWell"
      targetRegions="{ wellRegion1, wellRegion2, wellRegion3, wellRegion4, wellRegion5,␣
↪wellRegion6, wellRegion7, wellRegion8, wellRegion9, wellRegion10, wellRegion11,␣
↪wellRegion12 }"
      maxCompFractionChange="0.5"
      logLevel="1"
      useMass="1">
      <WellControls
        name="wellControls1"
        type="producer"
        control="BHP"
        referenceElevation="28"
        targetBHP="3.9e7"
        targetPhaseRate="1e6"
        targetPhaseName="oil"/>
      <WellControls
        name="wellControls2"
```

(continues on next page)

(continued from previous page)

```
          type="producer"
          control="BHP"
          referenceElevation="28"
          targetBHP="3.9e7"
          targetPhaseRate="1e6"
          targetPhaseName="oil"/>
```

Mesh definition and well geometry

In the presence of wells, the **Mesh** block of the XML input file includes two parts:

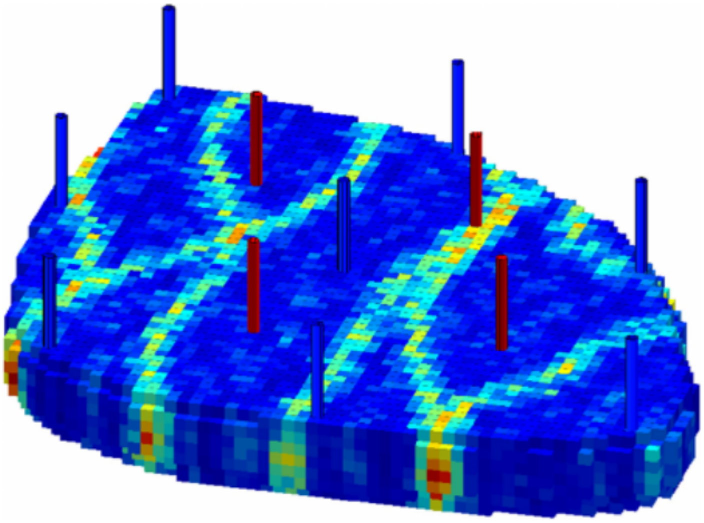
- a sub-block **VTKMesh** defining the reservoir mesh (see *Tutorial 2: External Meshes* for more on this),
- a collection of sub-blocks **InternalWell** defining the geometry of the wells.

The reservoir mesh is imported from a `.vtu` file that contains the mesh geometry and also includes the permeability values in the x, y, and z directions. These quantities must be specified using the metric unit system, i.e., in meters for the well geometry and square meters for the permeability field. We note that the mesh file only contains the active cells, so there is no keyword needed in the XML file to define them.

Each well is defined internally (i.e., not imported from a file) in a separate **InternalWell** XML sub-block. An **InternalWell** sub-block must point to the reservoir mesh that the well perforates using the attribute `meshName`, to the region corresponding to this well using the attribute `wellRegionName`, and to the control of this well using the attribute `wellControl`. Each block **InternalWell** must point to the reservoir mesh (using the attribute `meshName`), the corresponding well region (using the attribute `wellRegionName`), and the corresponding well control (using the attribute `wellControlName`).

Each well is defined using a vertical polyline going through the seven layers of the mesh, with a perforation in each layer. The well placement implemented here follows the pattern of the original test case. The well geometry must be specified in meters.

The location of the perforations is found internally using the linear distance along the wellbore from the top of the well, specified by the attribute `distanceFromHead`. It is the responsibility of the user to make sure that there is a perforation in the bottom cell of the well mesh otherwise an error will be thrown and the simulation will terminate. For each perforation, the well transmissibility factors employed to compute the perforation rates are calculated internally using the Peaceman formulation.



```
<Mesh>
  <VTKMesh
    name="mesh"
    file="../../../../../GEOSXDATA/DataSets/Egg/egg.vtu"
    fieldsToImport="{ PERM }"
    fieldNamesInGEOSX="{ rockPerm_permeability }">

    <InternalWell
      name="wellProducer1"
      wellRegionName="wellRegion1"
      wellControlsName="wellControls1"
      polylineNodeCoords="{ { 124, 340, 28 },
                            { 124, 340, 0 } }"
      polylineSegmentConn="{ { 0, 1 } }"
      radius="0.1"
      numElementsPerSegment="7">
      <Perforation
        name="producer1_perf1"
        distanceFromHead="2"/>
      <Perforation
        name="producer1_perf2"
        distanceFromHead="6"/>
      <Perforation
        name="producer1_perf3"
        distanceFromHead="10"/>
      <Perforation
        name="producer1_perf4"
        distanceFromHead="14"/>
      <Perforation
        name="producer1_perf5"
        distanceFromHead="18"/>
      <Perforation
        name="producer1_perf6"
        distanceFromHead="22"/>
      <Perforation
        name="producer1_perf7"
        distanceFromHead="26"/>
    </InternalWell>

    <InternalWell
      name="wellProducer2"
      wellRegionName="wellRegion2"
      wellControlsName="wellControls2"
      polylineNodeCoords="{ { 276, 316, 28 },
                            { 276, 316, 0 } }"
      polylineSegmentConn="{ { 0, 1 } }"
      radius="0.1"
      numElementsPerSegment="7">
      <Perforation
        name="producer2_perf1"
        distanceFromHead="2"/>
      <Perforation
        name="producer2_perf2"
```

(continues on next page)

(continued from previous page)

```
        distanceFromHead="6"/>
      <Perforation
        name="producer2_perf3"
        distanceFromHead="10"/>
      <Perforation
        name="producer2_perf4"
        distanceFromHead="14"/>
      <Perforation
        name="producer2_perf5"
        distanceFromHead="18"/>
      <Perforation
        name="producer2_perf6"
        distanceFromHead="22"/>
      <Perforation
        name="producer2_perf7"
        distanceFromHead="26"/>
    </InternalWell>
```

Events

In the **Events** XML block, we specify four types of **PeriodicEvents**.

The periodic event named `solverApplications` notifies GEOS that the coupled solver `coupledFlowAndWells` has to be applied to its target regions (here, reservoir and wells) at every time step. The time stepping strategy has been fully defined in the **CompositionalMultiphaseReservoir** coupling block using the `initialDt` attribute and the **NonlinearSolverParameters** nested block.

We also define an output event instructing GEOS to write out `.vtk` files at the time frequency specified by the attribute `timeFrequency`. Here, we choose to output the results using the VTK format (see *Tutorial 2: External Meshes* for a example that uses the Silo output file format). The `target` attribute must point to the **VTK** sub-block of the **Outputs** block (defined at the end of the XML file) by name (here, `vtkOutput`).

We define the events involved in the collection and output of the well production rates following the procedure defined in *Tasks Manager*. The time history collection events trigger the collection of the well rates at the desired frequency, while the time history output events trigger the output of the HDF5 files containing the time series. These events point by name to the corresponding blocks of the **Tasks** and **Outputs** XML blocks, respectively. Here, these names are `wellRateCollection1` and `timeHistoryOutput1`.

```
<Events
  maxTime="1.5e7">
  <PeriodicEvent
    name="vtk"
    timeFrequency="2e6"
    target="/Outputs/vtkOutput"/>

  <PeriodicEvent
    name="timeHistoryOutput1"
    timeFrequency="1.5e7"
    target="/Outputs/timeHistoryOutput1"/>

  <PeriodicEvent
    name="timeHistoryOutput2"
```

(continues on next page)

(continued from previous page)

```
      timeFrequency="1.5e7"
      target="/Outputs/timeHistoryOutput2"/>

    <PeriodicEvent
      name="timeHistoryOutput3"
      timeFrequency="1.5e7"
      target="/Outputs/timeHistoryOutput3"/>

    <PeriodicEvent
      name="timeHistoryOutput4"
      timeFrequency="1.5e7"
      target="/Outputs/timeHistoryOutput4"/>

    <PeriodicEvent
      name="solverApplications"
      maxEventDt="5e5"
      target="/Solvers/coupledFlowAndWells"/>

    <PeriodicEvent
      name="timeHistoryCollection1"
      timeFrequency="1e6"
      target="/Tasks/wellRateCollection1"/>

    <PeriodicEvent
      name="timeHistoryCollection2"
      timeFrequency="1e6"
      target="/Tasks/wellRateCollection2"/>

    <PeriodicEvent
      name="timeHistoryCollection3"
      timeFrequency="1e6"
      target="/Tasks/wellRateCollection3"/>

    <PeriodicEvent
      name="timeHistoryCollection4"
      timeFrequency="1e6"
      target="/Tasks/wellRateCollection4"/>

    <PeriodicEvent
      name="restarts"
      timeFrequency="7.5e6"
      targetExactTimestep="0"
      target="/Outputs/restartOutput"/>

  </Events>
```

Numerical methods

In the `NumericalMethods` XML block, we instruct GEOS to use a TPFA finite-volume numerical scheme. This part is similar to the corresponding section of *Multiphase Flow*, and has been adapted to match the specifications of the Egg model.

```
<NumericalMethods>
  <FiniteVolume>
    <TwoPointFluxApproximation
      name="fluidTPFA"/>
  </FiniteVolume>
</NumericalMethods>
```

Reservoir and well regions

In this section of the input file, we follow the procedure already described in *Multiphase Flow* for the definition of the reservoir region with multiphase constitutive models.

We associate a **CellElementRegion** named `reservoir` to the reservoir mesh. Since we have imported a mesh with one region consisting of hexahedral cells, we must set the attribute `cellBlocks` to `hexahedra`.

Note: If you use a name that is not `hexahedra` for this attribute, GEOS will throw an error at the beginning of the simulation.

We also associate a **WellElementRegion** to each well. As the **CellElementRegion**, it contains a `materialList` that must point (by name) to the constitutive models defined in the **Constitutive** XML block.

```
<ElementRegions>
  <CellElementRegion
    name="reservoir"
    cellBlocks="{ hexahedra }"
    materialList="{ fluid, rock, relperm }"/>

  <WellElementRegion
    name="wellRegion1"
    materialList="{ fluid, relperm }"/>

  <WellElementRegion
    name="wellRegion2"
    materialList="{ fluid, relperm }"/>
```

Constitutive models

The **CompositionalMultiphaseFVM** physics solver relies on at least four types of constitutive models listed in the **Constitutive** XML block:

- a fluid model describing the thermodynamics behavior of the fluid mixture,
- a relative permeability model,
- a rock permeability model,
- a rock porosity model.

All the parameters must be provided using the SI unit system.

This part is identical to that of *Multiphase Flow*.

```
<Constitutive>
  <DeadOilFluid
    name="fluid"
    phaseNames="{ oil, water }"
    surfaceDensities="{ 848.9, 1025.2 }"
    componentMolarWeight="{ 114e-3, 18e-3 }"
    tableFiles="{ pvdo.txt, pvtw.txt }"/>

  <BrooksCoreyRelativePermeability
    name="relperm"
    phaseNames="{ oil, water }"
    phaseMinVolumeFraction="{ 0.1, 0.2 }"
    phaseRelPermExponent="{ 4.0, 3.0 }"
    phaseRelPermMaxValue="{ 0.8, 0.75 }"/>

  <CompressibleSolidConstantPermeability
    name="rock"
    solidModelName="nullSolid"
    porosityModelName="rockPorosity"
    permeabilityModelName="rockPerm"/>

  <NullModel
    name="nullSolid"/>

  <PressurePorosity
    name="rockPorosity"
    defaultReferencePorosity="0.2"
    referencePressure="0.0"
    compressibility="1.0e-13"/>

  <ConstantPermeability
    name="rockPerm"
    permeabilityComponents="{ 1.0e-12, 1.0e-12, 1.0e-12 }"/>
</Constitutive>
```

Initial conditions

We are ready to specify the reservoir initial conditions of the problem in the **FieldSpecifications** XML block. The well variables do not have to be initialized here since they will be defined internally.

The formulation of the **CompositionalMultiphaseFVM** physics solver (documented at *Compositional Multiphase Flow Solver*) requires the definition of the initial pressure field and initial global component fractions. We define here a uniform pressure field that does not satisfy the hydrostatic equilibrium, but a hydrostatic initialization of the pressure field is possible using *Functions*:. For the initialization of the global component fractions, we remind the user that their `component` attribute (here, 0 or 1) is used to point to a specific entry of the `phaseNames` attribute in the **DeadOilFluid** block.

Note that we also define the uniform porosity field here since it is not included in the mesh file imported by the **VTKMesh**.

```
<FieldSpecifications>
  <FieldSpecification
    name="initialPressure"
    initialCondition="1"
    setNames="{ all }"
    objectPath="ElementRegions/reservoir/hexahedra"
    fieldName="pressure"
    scale="4e7"/>

  <FieldSpecification
    name="initialComposition_oil"
    initialCondition="1"
    setNames="{ all }"
    objectPath="ElementRegions/reservoir/hexahedra"
    fieldName="globalCompFraction"
    component="0"
    scale="0.9"/>

  <FieldSpecification
    name="initialComposition_water"
    initialCondition="1"
    setNames="{ all }"
    objectPath="ElementRegions/reservoir/hexahedra"
    fieldName="globalCompFraction"
    component="1"
    scale="0.1"/>
</FieldSpecifications>
```

Outputs

In this section, we request an output of the results in VTK format and an output of the rates for each producing well. Note that the name defined here must match the name used in the **Events** XML block to define the output frequency.

```
<Outputs>
  <VTK
    name="vtkOutput"/>

  <TimeHistory
    name="timeHistoryOutput1"
    sources="{ /Tasks/wellRateCollection1 }"
    filename="wellRateHistory1"/>

  <TimeHistory
    name="timeHistoryOutput2"
    sources="{ /Tasks/wellRateCollection2 }"
    filename="wellRateHistory2"/>

  <TimeHistory
    name="timeHistoryOutput3"
    sources="{ /Tasks/wellRateCollection3 }"
    filename="wellRateHistory3"/>
```

(continues on next page)

(continued from previous page)

```
    <TimeHistory
      name="timeHistoryOutput4"
      sources="{ /Tasks/wellRateCollection4 }"
      filename="wellRateHistory4"/>

    <Restart
      name="restartOutput"/>

  </Outputs>
```

Tasks

In the **Events** block, we have defined four events requesting that a task periodically collects the rate for each producing well. This task is defined here, in the **PackCollection** XML sub-block of the **Tasks** block. The task contains the path to the object on which the field to collect is registered (here, a `WellElementSubRegion`) and the name of the field (here, `wellElementMixtureConnectionRate`). The details of the history collection mechanism can be found in *Tasks Manager*.

```
  <Tasks>
    <PackCollection
      name="wellRateCollection1"
      objectPath="ElementRegions/wellRegion1/wellRegion1UniqueSubRegion"
      fieldName="wellElementMixtureConnectionRate"/>

    <PackCollection
      name="wellRateCollection2"
      objectPath="ElementRegions/wellRegion2/wellRegion2UniqueSubRegion"
      fieldName="wellElementMixtureConnectionRate"/>

    <PackCollection
      name="wellRateCollection3"
      objectPath="ElementRegions/wellRegion3/wellRegion3UniqueSubRegion"
      fieldName="wellElementMixtureConnectionRate"/>

    <PackCollection
      name="wellRateCollection4"
      objectPath="ElementRegions/wellRegion4/wellRegion4UniqueSubRegion"
      fieldName="wellElementMixtureConnectionRate"/>
  </Tasks>
```

All elements are now in place to run GEOS.

Running GEOS

The first few lines appearing to the console are indicating that the XML elements are read and registered correctly:

```
Adding Mesh: VTKMesh, mesh
Adding Mesh: InternalWell, wellProducer1
Adding Mesh: InternalWell, wellProducer2
Adding Mesh: InternalWell, wellProducer3
Adding Mesh: InternalWell, wellProducer4
Adding Mesh: InternalWell, wellInjector1
Adding Mesh: InternalWell, wellInjector2
Adding Mesh: InternalWell, wellInjector3
Adding Mesh: InternalWell, wellInjector4
Adding Mesh: InternalWell, wellInjector5
Adding Mesh: InternalWell, wellInjector6
Adding Mesh: InternalWell, wellInjector7
Adding Mesh: InternalWell, wellInjector8
Adding Solver of type CompositionalMultiphaseReservoir, named coupledFlowAndWells
Adding Solver of type CompositionalMultiphaseFVM, named compositionalMultiphaseFlow
Adding Solver of type CompositionalMultiphaseWell, named compositionalMultiphaseWell
Adding Event: PeriodicEvent, vtk
Adding Event: PeriodicEvent, timeHistoryOutput1
Adding Event: PeriodicEvent, timeHistoryOutput2
Adding Event: PeriodicEvent, timeHistoryOutput3
Adding Event: PeriodicEvent, timeHistoryOutput4
Adding Event: PeriodicEvent, solverApplications
Adding Event: PeriodicEvent, timeHistoryCollection1
Adding Event: PeriodicEvent, timeHistoryCollection2
Adding Event: PeriodicEvent, timeHistoryCollection3
Adding Event: PeriodicEvent, timeHistoryCollection4
Adding Event: PeriodicEvent, restarts
Adding Output: VTK, vtkOutput
Adding Output: TimeHistory, timeHistoryOutput1
Adding Output: TimeHistory, timeHistoryOutput2
Adding Output: TimeHistory, timeHistoryOutput3
Adding Output: TimeHistory, timeHistoryOutput4
Adding Output: Restart, restartOutput
Adding Object CellElementRegion named reservoir from ObjectManager::Catalog.
Adding Object WellElementRegion named wellRegion1 from ObjectManager::Catalog.
Adding Object WellElementRegion named wellRegion2 from ObjectManager::Catalog.
Adding Object WellElementRegion named wellRegion3 from ObjectManager::Catalog.
Adding Object WellElementRegion named wellRegion4 from ObjectManager::Catalog.
Adding Object WellElementRegion named wellRegion5 from ObjectManager::Catalog.
Adding Object WellElementRegion named wellRegion6 from ObjectManager::Catalog.
Adding Object WellElementRegion named wellRegion7 from ObjectManager::Catalog.
Adding Object WellElementRegion named wellRegion8 from ObjectManager::Catalog.
Adding Object WellElementRegion named wellRegion9 from ObjectManager::Catalog.
Adding Object WellElementRegion named wellRegion10 from ObjectManager::Catalog.
Adding Object WellElementRegion named wellRegion11 from ObjectManager::Catalog.
Adding Object WellElementRegion named wellRegion12 from ObjectManager::Catalog.
```

This is followed by the creation of the 18553 hexahedral cells of the imported mesh. At this point, we are done with the case set-up and the code steps into the execution of the simulation itself:

```
Time: 0s, dt:10000s, Cycle: 0
    Attempt:  0, ConfigurationIter:  0, NewtonIter:  0
    ( Rflow ) = ( 1.01e+01 ) ;     ( Rwell ) = ( 4.96e+00 ) ;     ( R ) = ( 1.13e+01 ) ;
    Attempt:  0, ConfigurationIter:  0, NewtonIter:  1
    ( Rflow ) = ( 1.96e+00 ) ;     ( Rwell ) = ( 8.07e-01 ) ;     ( R ) = ( 2.12e+00 ) ;
    Last LinSolve(iter,res) = (  44, 8.96e-03 ) ;
    Attempt:  0, ConfigurationIter:  0, NewtonIter:  2
    ( Rflow ) = ( 4.14e-01 ) ;     ( Rwell ) = ( 1.19e-01 ) ;     ( R ) = ( 4.31e-01 ) ;
    Last LinSolve(iter,res) = (  44, 9.50e-03 ) ;
    Attempt:  0, ConfigurationIter:  0, NewtonIter:  3
    ( Rflow ) = ( 1.77e-02 ) ;     ( Rwell ) = ( 9.38e-03 ) ;     ( R ) = ( 2.00e-02 ) ;
    Last LinSolve(iter,res) = (  47, 8.69e-03 ) ;
    Attempt:  0, ConfigurationIter:  0, NewtonIter:  4
    ( Rflow ) = ( 1.13e-04 ) ;     ( Rwell ) = ( 5.09e-05 ) ;     ( R ) = ( 1.24e-04 ) ;
    Last LinSolve(iter,res) = (  50, 9.54e-03 ) ;
    Attempt:  0, ConfigurationIter:  0, NewtonIter:  5
    ( Rflow ) = ( 2.17e-08 ) ;     ( Rwell ) = ( 1.15e-07 ) ;     ( R ) = ( 1.17e-07 ) ;
    Last LinSolve(iter,res) = (  55, 2.71e-04 ) ;
coupledFlowAndWells: Newton solver converged in less than 15 iterations, time-step␣
↪required will be doubled.
```

Visualization

A file compatible with Paraview is produced in this example. It is found in the output folder, and usually has the extension *.pvd*. More details about this file format can be found here. We can load this file into Paraview directly and visualize results:

We have instructed GEOS to output the time series of rates for each producer. The data contained in the corresponding hdf5 files can be extracted and plotted as shown below.

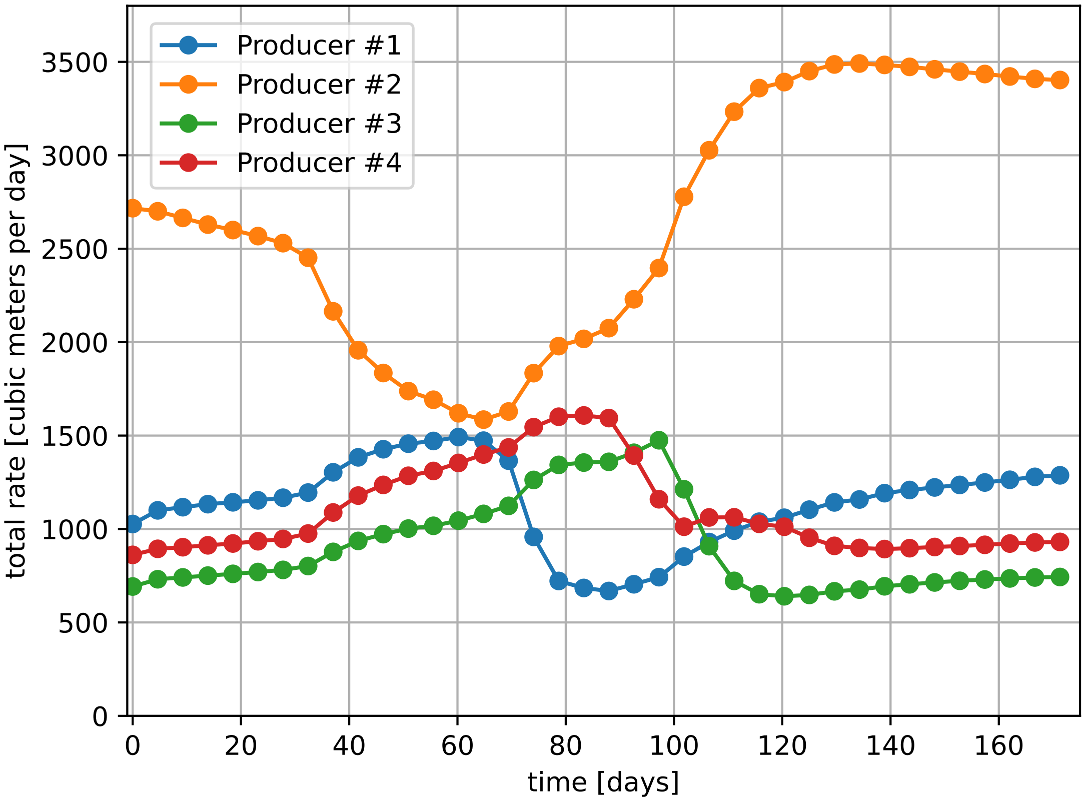
To go further

Feedback on this example

This concludes the example on setting up a Dead-Oil simulation in the Egg model. For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

For more details

- A complete description of the reservoir flow solver is found here: *Compositional Multiphase Flow Solver*.
- The well solver is description at *Compositional Multiphase Well Solver*.
- The available constitutive models are listed at *Constitutive Models*.


Producer #1
Producer #2
Producer #3
Producer #4
total rate [cubic meters per day]
0
500
1000
1500
2000
2500
3000
3500
0
20
40
60
80
100
120
140
160
time [days]

1.3.3 CO_2 Injection

Context

In this example, we show how to set up a multiphase simulation of CO_2 injection.

Objectives

At the end of this example you will know:

- how to set up a CO_2 injection scenario with a well,
- how to run a case using MPI-parallelism.

Input file

The XML file for this test case is located at :

```
inputFiles/compositionalMultiphaseWell/simpleCo2InjTutorial_base.xml
```

```
inputFiles/compositionalMultiphaseWell/simpleCo2InjTutorial_smoke.xml
```

This mesh is a simple internally generated regular grid (50 x 1 x 150). A single CO_2 injection well is at the center of the reservoir.

The XML file considered here follows the typical structure of the GEOS input files:

1. *Solver*
2. *Mesh*
3. *Events*
4. *NumericalMethods*
5. *ElementRegions*
6. *Constitutive*
7. *FieldSpecifications*
8. *Outputs*
9. *Tasks*

Multiphase flow and well solvers

Let us inspect the **Solver** XML tags. They consist of three blocks **CompositionalMultiphaseFVM**, **CompositionalMultiphaseWell** and **CompositionalMultiphaseReservoir**, which are respectively handling the solution from multiphase flow in the reservoir, multiphase flow in the wells, and coupling between those two parts.

```
  <Solvers>
    <CompositionalMultiphaseReservoir
      name="coupledFlowAndWells"
      flowSolverName="compositionalMultiphaseFlow"
      wellSolverName="compositionalMultiphaseWell"
      logLevel="1"
      initialDt="1e2"
      targetRegions="{ reservoir, wellRegion }">
      <NonlinearSolverParameters
        newtonTol="1.0e-4"
```

(continues on next page)

(continued from previous page)

```
      lineSearchAction="None"
      maxTimeStepCuts="10"
      newtonMaxIter="40"/>
    <LinearSolverParameters
      solverType="fgmres"
      preconditionerType="mgr"
      krylovTol="1e-5"/>
  </CompositionalMultiphaseReservoir>

  <CompositionalMultiphaseFVM
    name="compositionalMultiphaseFlow"
    targetRegions="{ reservoir }"
    discretization="fluidTPFA"
    temperature="368.15"
    maxCompFractionChange="0.2"
    logLevel="1"
    useMass="1"/>

  <CompositionalMultiphaseWell
    name="compositionalMultiphaseWell"
    targetRegions="{ wellRegion }"
    maxCompFractionChange="0.2"
    logLevel="1"
    useMass="1">
    <WellControls
      name="wellControls"
      type="injector"
      control="totalVolRate"
      enableCrossflow="0"
      referenceElevation="6650"
      useSurfaceConditions="1"
      surfacePressure="101325"
      surfaceTemperature="288.71"
      targetBHP="5e7"
      targetTotalRate="1.5"
      injectionTemperature="368.15"
      injectionStream="{ 1, 0 }"/>
  </CompositionalMultiphaseWell>
</Solvers>
```

In the **CompositionalMultiphaseFVM** (*Compositional Multiphase Flow Solver*), a classical multiphase compositional solver with a TPFA discretization is described.

The **CompositionalMultiphaseWell** (*Compositional Multiphase Well Solver*) consists of wellbore specifications (see *Multiphase Flow with Wells* for detailed example). As its reservoir counterpart, it includes references to fluid and relative permeability models, but also defines a **WellControls** sub-tag. This sub-tag specifies the CO_2 injector control mode: the well is initially rate-controlled, with a rate specified in `targetTotalRate` and a maximum pressure specified in `targetBHP`. The injector-specific attribute, `injectionStream`, describes the composition of the injected mixture (here, pure CO_2).

The **CompositionalMultiphaseReservoir** coupling section describes the binding between those two previous elements (see *Poromechanics* for detailed example on coupling physics in GEOS). In addition to being bound to the previously described blocks through `flowSolverName` and `wellSolverName` sub-tags, it contains the `initialDt` starting time-step size value and defines the **NonlinearSolverParameters** and **LinearSolverParameters** that are used to control

Newton-loop and linear solver behaviors (see *Linear Solvers* for a detailed description of linear solver attributes).

Note: To use the linear solver options of this example, you need to ensure that GEOS is configured to use the Hypre linear solver package.

Mesh and well geometry

In this example, the **Mesh** tag is used to generate the reservoir mesh internally (*Tutorial 1: First Steps*). The internal generation of well is defined with the **InternalWell** sub-tag. Apart from the `name` identifier attribute and their `wellRegionName` (*ElementRegions*) and `wellControlsName` (*Solver*) binding attributes, `polylineNodeCoords` and `polylineSegmentConn` attributes are used to define the path of the wellbore and connections between its nodes. The `numElementsPerSegment` discretizes the wellbore segments while the `radius` attribute specifies the wellbore radius (*Multiphase Flow with Wells* for details on wells). Once the wellbore is defined and discretized, the position of **Perforations** is defined using the linear distance from the head of the wellbore (`distanceFromHead`).

```
<Mesh>
  <InternalMesh
    name="cartesianMesh"
    elementTypes="{ C3D8 }"
    xCoords="{ 0, 1000 }"
    yCoords="{ 450, 550 }"
    zCoords="{ 6500, 7700 }"
    nx="{ 50 }"
    ny="{ 1 }"
    nz="{ 150 }"
    cellBlockNames="{ cellBlock }">

    <InternalWell
      name="wellInjector1"
      wellRegionName="wellRegion"
      wellControlsName="wellControls"
      polylineNodeCoords="{ { 525.0, 525.0, 6650.00 },
                            { 525.0, 525.0, 6600.00 } }"
      polylineSegmentConn="{ { 0, 1 } }"
      radius="0.1"
      numElementsPerSegment="2">
      <Perforation
        name="injector1_perf1"
        distanceFromHead="45"/>
    </InternalWell>
  </InternalMesh>
</Mesh>
```

Note: It is the responsibility of the user to make sure that there is a perforation in the bottom cell of the well mesh, otherwise an error will be thrown and the simulation will terminate.

Events

The solver is applied as a periodic event whose target is referred to as `coupledFlowAndWells` nametag. Using the `maxEventDt` attribute, we specify a max time step size of 5 x 10^6 seconds.

The output event triggers a VTK output every 10^7 seconds, constraining the solver schedule to match exactly these dates. The output path to data is specified as a `target` of this **PeriodicEvent**.

Another periodic event is defined under the name `restarts`. It consists of saved checkpoints every 5 x 10^7 seconds, whose physical output folder name is defined under the **Output** tag.

Finally, the time history collection and output events are used to trigger the mechanisms involved in the generation of a time series of well pressure (see the procedure outlined in *Tasks Manager*, and the example in *Multiphase Flow with Wells*).

```
<Events
  maxTime="5e8">

  <PeriodicEvent
    name="outputs"
    timeFrequency="1e7"
    targetExactTimestep="1"
    target="/Outputs/simpleReservoirViz"/>

  <PeriodicEvent
    name="restarts"
    timeFrequency="5e7"
    targetExactTimestep="1"
    target="/Outputs/restartOutput"/>

  <PeriodicEvent
    name="timeHistoryCollection"
    timeFrequency="1e7"
    targetExactTimestep="1"
    target="/Tasks/wellPressureCollection" />

  <PeriodicEvent
    name="timeHistoryOutput"
    timeFrequency="2e8"
    targetExactTimestep="1"
    target="/Outputs/timeHistoryOutput" />

  <PeriodicEvent
    name="solverApplications"
    maxEventDt="5e5"
    target="/Solvers/coupledFlowAndWells"/>

</Events>
```

Numerical methods

The **TwoPointFluxApproximation** is chosen for the fluid equation discretization. The tag specifies:

- A primary field to solve for as `fieldName`. For a flow problem, this field is pressure.
- A set of target regions in `targetRegions`.
- A `coefficientName` pointing to the field used for TPFA transmissibilities construction.
- A `coefficientModelNames` used to specify the permeability constitutive model(s).

```
<NumericalMethods>
  <FiniteVolume>
    <TwoPointFluxApproximation
      name="fluidTPFA"/>
  </FiniteVolume>
</NumericalMethods>
```

Element regions

We define a **CellElementRegion** pointing to the cell block defining the reservoir mesh, and a **WellElementRegion** for the well. The two regions contain a list of constitutive model names.

```
<ElementRegions>
  <CellElementRegion
    name="reservoir"
    cellBlocks="{ cellBlock }"
    materialList="{ fluid, rock, relperm }"/>

  <WellElementRegion
    name="wellRegion"
    materialList="{ fluid, relperm, rockPerm }"/>
</ElementRegions>
```

Constitutive laws

Under the **Constitutive** tag, four items can be found:

- **CO2BrinePhillipsFluid** : this tag defines phase names, component molar weights, and fluid behaviors such as CO_2 solubility in brine and viscosity/density dependencies on pressure and temperature.
- **PressurePorosity** : this tag contains all the data needed to model rock compressibility.
- **BrooksCoreyRelativePermeability** : this tag defines the relative permeability model for each phase, its end-point values, residual volume fractions (saturations), and the Corey exponents.
- **ConstantPermeability** : this tag defines the permeability model that is set to a simple constant diagonal tensor, whose values are defined in `permeabilityComponent`. Note that these values will be overwritten by the permeability field imported in **FieldSpecifications**.

```
<Constitutive>
  <CO2BrinePhillipsFluid
    name="fluid"
    phaseNames="{ gas, water }"
```

(continues on next page)

(continued from previous page)

```
      componentNames="{ co2, water }"
      componentMolarWeight="{ 44e-3, 18e-3 }"
      phasePVTParaFiles="{ pvtgas.txt, pvtliquid.txt }"
      flashModelParaFile="co2flash.txt"/>

    <CompressibleSolidConstantPermeability
      name="rock"
      solidModelName="nullSolid"
      porosityModelName="rockPorosity"
      permeabilityModelName="rockPerm"/>

    <NullModel
      name="nullSolid"/>

    <PressurePorosity
      name="rockPorosity"
      defaultReferencePorosity="0.1"
      referencePressure="1.0e7"
      compressibility="4.5e-10"/>

    <BrooksCoreyRelativePermeability
      name="relperm"
      phaseNames="{ gas, water }"
      phaseMinVolumeFraction="{ 0.05, 0.30 }"
      phaseRelPermExponent="{ 2.0, 2.0 }"
      phaseRelPermMaxValue="{ 1.0, 1.0 }"/>

    <ConstantPermeability
      name="rockPerm"
      permeabilityComponents="{ 1.0e-17, 1.0e-17, 3.0e-17 }"/>

  </Constitutive>
```

The PVT data specified by **CO2BrinePhillipsFluid** is set to model the behavior of the CO_2-brine system as a function of pressure, temperature, and salinity. We currently rely on a two-phase, two-component (CO_2 and H_2O) model in which salinity is a constant parameter in space and in time. The model is described in detail in *CO2-brine model*. The model definition requires three text files:

In *co2flash.txt*, we define the CO_2 solubility model used to compute the amount of CO_2 dissolved in the brine phase as a function of pressure (in Pascal), temperature (in Kelvin), and salinity (in units of molality):

```
FlashModel CO2Solubility  1e6 1.5e7 5e4 367.15 369.15 1 0
```

The first keyword is an identifier for the model type (here, a flash model). It is followed by the model name. Then, the lower, upper, and step increment values for pressure and temperature ranges are specified. The trailing 0 defines a zero-salinity in the model. Note that the water component is not allowed to evaporate into the CO_2 -rich phase.

The *pvtgas.txt* and *pvtliquid.txt* files define the models used to compute the density and viscosity of the two phases, as follows:

```
DensityFun SpanWagnerCO2Density 1e6 1.5e7 5e4 94 96 1
ViscosityFun FenghourCO2Viscosity 1e6 1.5e7 5e4 94 96
```

```
DensityFun PhillipsBrineDensity 1e6 1.5e7 5e4 94 96 1 0
ViscosityFun PhillipsBrineViscosity 0
```

In these files, the first keyword of each line is an identifier for the model type (either a density or a viscosity model). It is followed by the model name. Then, the lower, upper, and step increment values for pressure and temperature ranges are specified. The trailing 0 for `PhillipsBrineDensity` and `PhillipsBrineViscosity` entry is the salinity of the brine, set to zero.

Note: It is the responsibility of the user to make sure that the pressure and temperature values encountered in the simulation (in the reservoir and in the well) are within the bounds specified in the PVT files. GEOS will not throw an error if a value outside these bounds is encountered, but the (nonlinear) behavior of the simulation and the quality of the results will likely be negatively impacted.

Property specification

The **FieldSpecifications** tag is used to declare fields such as directional permeability, reference porosity, initial pressure, and compositions. Here, these fields are homogeneous, except for the permeability field that is taken as an heterogeneous log-normally distributed field and specified in **Functions** as in *Tutorial 3: Regions and Property Specifications*.

```
  <FieldSpecifications>
   <FieldSpecification
      name="permx"
      initialCondition="1"
      component="0"
      setNames="{ all }"
      objectPath="ElementRegions"
      fieldName="rockPerm_permeability"
      scale="1e-15"
      functionName="permxFunc"/>

    <FieldSpecification
      name="permy"
      initialCondition="1"
      component="1"
      setNames="{ all }"
      objectPath="ElementRegions"
      fieldName="rockPerm_permeability"
      scale="1e-15"
      functionName="permyFunc"/>

    <FieldSpecification
      name="permz"
      initialCondition="1"
      component="2"
      setNames="{ all }"
      objectPath="ElementRegions"
      fieldName="rockPerm_permeability"
      scale="3e-15"
      functionName="permzFunc"/>

    <FieldSpecification
```

(continues on next page)

(continued from previous page)

```
      name="initialPressure"
      initialCondition="1"
      setNames="{ all }"
      objectPath="ElementRegions/reservoir"
      fieldName="pressure"
      scale="1.25e7"/>

    <FieldSpecification
      name="initialComposition_co2"
      initialCondition="1"
      setNames="{ all }"
      objectPath="ElementRegions/reservoir"
      fieldName="globalCompFraction"
      component="0"
      scale="0.0"/>

    <FieldSpecification
      name="initialComposition_water"
      initialCondition="1"
      setNames="{ all }"
      objectPath="ElementRegions/reservoir"
      fieldName="globalCompFraction"
      component="1"
      scale="1.0"/>
  </FieldSpecifications>
```

Note: In this case, we are using the same permeability field (*perm.geos*) for all the directions. Note also that the `fieldName` values are set to *rockPerm_permeability* to access the permeability field handled as a **Constitutive** law. These permeability values will overwrite the values already set in the **Constitutive** block.

Output

The **Outputs** XML tag is used to write visualization, restart, and time history files.

Here, we write visualization files in a format natively readable by Paraview under the tag **VTK**. A **Restart** tag is also be specified. In conjunction with a **PeriodicEvent**, a restart file allows to resume computations from a set of checkpoints in time. Finally, we require an output of the well pressure history using the **TimeHistory** tag.

```
  <Outputs>
    <VTK
      name="simpleReservoirViz"/>

    <Restart
      name="restartOutput"/>

    <TimeHistory
      name="timeHistoryOutput"
      sources="{/Tasks/wellPressureCollection}"
      filename="wellPressureHistory" />

```

(continues on next page)

(continued from previous page)

```
</Outputs>
```

Tasks

In the **Events** block, we have defined an event requesting that a task periodically collects the pressure at the well. This task is defined here, in the **PackCollection** XML sub-block of the **Tasks** block. The task contains the path to the object on which the field to collect is registered (here, a `WellElementSubRegion`) and the name of the field (here, `pressure`). The details of the history collection mechanism can be found in *Tasks Manager*.

```
<Tasks>
  <PackCollection
    name="wellPressureCollection"
    objectPath="ElementRegions/wellRegion/wellRegionUniqueSubRegion"
    fieldName="pressure" />

</Tasks>
```

Running GEOS

The simulation can be launched with 4 cores using MPI-parallelism:

```
mpirun -np 4 geosx -i simpleCo2InjTutorial.xml -x 1 -y 1 -z 4
```

A restart from a checkpoint file *simpleCo2InjTutorial_restart_000000024.root* is always available thanks to the following command line :

```
mpirun -np 4 geosx -i simpleCo2InjTutorial.xml -r simpleCo2InjTutorial_restart_000000024␣
↪-x 1 -y 1 -z 4
```

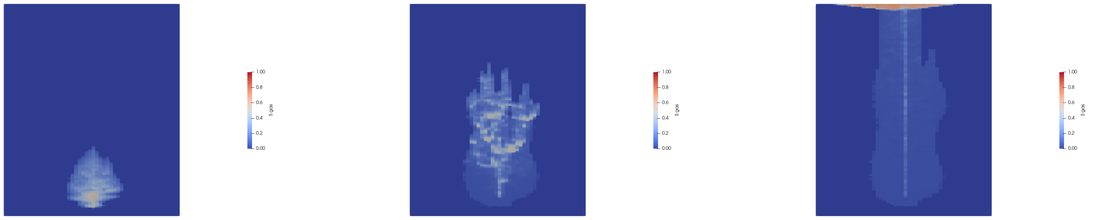
The output then shows the loading of HDF5 restart files by each core.

```
Loading restart file simpleCo2InjTutorial_restart_000000024
Rank 0: rankFilePattern = simpleCo2InjTutorial_restart_000000024/rank_%07d.hdf5
Rank 0: Reading in restart file at simpleCo2InjTutorial_restart_000000024/rank_0000000.
↪hdf5
Rank 1: Reading in restart file at simpleCo2InjTutorial_restart_000000024/rank_0000001.
↪hdf5
Rank 3: Reading in restart file at simpleCo2InjTutorial_restart_000000024/rank_0000003.
↪hdf5
Rank 2: Reading in restart file at simpleCo2InjTutorial_restart_000000024/rank_0000002.
↪hdf5
```

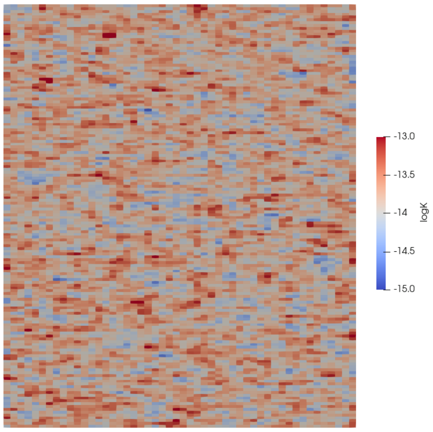
and the simulation restarts from this point in time.

Visualization

Using Paraview, we can observe the CO $_2$ plume moving upward under buoyancy effects and forming a gas cap at the top of the domain,



The heterogeneous values of the log permeability field can also be visualized in Paraview as shown below:



To go further

Feedback on this example

This concludes the CO $_2$ injection field case example. For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

For more details

- A complete description of the reservoir flow solver is found here: *Compositional Multiphase Flow Solver*.
- The well solver is described at *Compositional Multiphase Well Solver*.
- The available fluid constitutive models are listed at *Fluid Models*.

1.3.4 Poromechanics

Context

In this example, we use a coupled solver to solve a poroelastic Terzaghi-type problem, a classic benchmark in poroelasticity. We do so by coupling a single phase flow solver with a small-strain Lagrangian mechanics solver.

Objectives

At the end of this example you will know:

- how to use multiple solvers for poromechanical problems,

- how to define finite elements and finite volume numerical methods.

Input file

This example uses no external input files and everything required is contained within two GEOS input files located at:

```
inputFiles/poromechanics/PoroElastic_Terzaghi_base_direct.xml
```

```
inputFiles/poromechanics/PoroElastic_Terzaghi_smoke.xml
```

Description of the case

We simulate the consolidation of a poroelastic fluid-saturated column of height L having unit cross-section. The column is instantaneously loaded at time t = 0 s with a constant compressive traction w applied on the face highlighted in red in the figure below. Only the loaded face if permeable to fluid flow, with the remaining parts of the boundary subject to roller constraints and impervious.

Fig. 1.1: Sketch of the setup for Terzaghi's problem.

GEOS will calculate displacement and pressure fields along the column as a function of time. We will use the analytical solution for pressure to check the accuracy of the solution obtained with GEOS, namely

$$p(x,t) = \frac{4}{\pi} p_0 \sum_{m=0}^{\infty} \frac{1}{2m+1} \text{exp} \left[-\frac{(2m+1)^2 \pi^2 c_c t}{4L^2} \right] \text{sin} \left[\frac{(2m+1)\pi x}{2L} \right],$$

where $p_0 = \frac{b}{K_v S_\epsilon + b^2} |w|$ is the initial pressure, constant throughout the column, and $c_c = \frac{\kappa}{\mu} \frac{K_v}{K_v S_\epsilon + b^2}$ is the consolidation coefficient (or diffusion coefficient), with

- b Biot's coefficient
- $K_v = \frac{E(1-\nu)}{(1+\nu)(1-2\nu)}$ the uniaxial bulk modulus, E Young's modulus, and ν Poisson's ratio
- $S_\epsilon = \frac{(b-\phi)(1-b)}{K} + \phi c_f$ the constrained specific storage coefficient, ϕ porosity, $K = \frac{E}{3(1-2\nu)}$ the bulk modulus, and c_f the fluid compressibility
- κ the isotropic permeability
- μ the fluid viscosity

The characteristic consolidation time of the system is defined as $t_c = \frac{L^2}{c_c}$. Knowledge of t_c is useful for choosing appropriately the timestep sizes that are used in the discrete model.

Coupled solvers

GEOS is a multi-physics tool. Different combinations of physics solvers available in the code can be applied in different regions of the mesh at different moments of the simulation. The XML `Solvers` tag is used to list and parameterize these solvers.

We define and characterize each single-physics solver separately. Then, we define a *coupling solver* between these single-physics solvers as another, separate, solver. This approach allows for generality and flexibility in our multi-physics resolutions. The order in which these solver specifications is done is not important. It is important, though, to instantiate each single-physics solver with meaningful names. The names given to these single-physics solver instances will be used to recognize them and create the coupling.

To define a poromechanical coupling, we will effectively define three solvers:

- the single-physics flow solver, a solver of type `SinglePhaseFVM` called here `SinglePhaseFlowSolver` (more information on these solvers at *Singlephase Flow Solver*),
- the small-stress Lagrangian mechanics solver, a solver of type `SolidMechanicsLagrangianSSLE` called here `LinearElasticitySolver` (more information here: *Solid Mechanics Solver*),
- the coupling solver that will bind the two single-physics solvers above, an object of type `SinglePhasePoromechanics` called here `PoroelasticitySolver` (more information at *Poromechanics Solver*).

Note that the `name` attribute of these solvers is chosen by the user and is not imposed by GEOS.

The two single-physics solvers are parameterized as explained in their respective documentation.

Let us focus our attention on the coupling solver. This solver (`PoroelasticitySolver`) uses a set of attributes that specifically describe the coupling for a poromechanical framework. For instance, we must point this solver to the correct fluid solver (here: `SinglePhaseFlowSolver`), the correct solid solver (here: `LinearElasticitySolver`). Now that these two solvers are tied together inside the coupling solver, we have a coupled multiphysics problem defined. More parameters are required to characterize a coupling. Here, we specify the discretization method (`FE1`, defined further in the input file), and the target regions (here, we only have one, `Domain`).

```
    <SinglePhasePoromechanics
      name="PoroelasticitySolver"
      solidSolverName="LinearElasticitySolver"
      flowSolverName="SinglePhaseFlowSolver"
      logLevel="1"
      targetRegions="{ Domain }">
      <LinearSolverParameters
        directParallel="0"/>
    </SinglePhasePoromechanics>

    <SolidMechanicsLagrangianSSLE
      name="LinearElasticitySolver"
      timeIntegrationOption="QuasiStatic"
      logLevel="1"
      discretization="FE1"
      targetRegions="{ Domain }"/>

    <SinglePhaseFVM
      name="SinglePhaseFlowSolver"
      logLevel="1"
      discretization="singlePhaseTPFA"
      targetRegions="{ Domain }"/>
  </Solvers>
```

Multiphysics numerical methods

Numerical methods in multiphysics settings are similar to single physics numerical methods. All can be defined under the same `NumericalMethods` XML tag. In this problem, we use finite volume for flow and finite elements for solid mechanics. Both methods require additional parameterization attributes to be defined here.

As we have seen before, the coupling solver and the solid mechanics solver require the specification of a discretization method called `FE1`. This discretization method is defined here as a finite element method using linear basis functions and Gaussian quadrature rules. For more information on defining finite elements numerical schemes, please see the dedicated *FiniteElement* section.

The finite volume method requires the specification of a discretization scheme. Here, we use a two-point flux approximation as described in the dedicated documentation (found here: *FiniteVolume*).

```
<NumericalMethods>
  <FiniteElements>
    <FiniteElementSpace
      name="FE1"
      order="1"/>
  </FiniteElements>

  <FiniteVolume>
    <TwoPointFluxApproximation
      name="singlePhaseTPFA"/>
  </FiniteVolume>
</NumericalMethods>
```

Mesh, material properties, and boundary conditions

Last, let us take a closer look at the geometry of this simple problem. We use the internal mesh generator to create a beam-like mesh, with one single element along the Y and Z axes, and 21 elements along the X axis. All the elements are hexahedral elements (C3D8) of the same dimension (1x1x1 meters).

```
<Mesh>
  <InternalMesh
    name="mesh1"
    elementTypes="{ C3D8 }"
    xCoords="{ 0, 10 }"
    yCoords="{ 0, 1 }"
    zCoords="{ 0, 1 }"
    nx="{ 400 }"
    ny="{ 16 }"
    nz="{ 16 }"
    cellBlockNames="{ cb1 }"/>
</Mesh>
```

The parameters used in the simulation are summarized in the following table.

Symbol	Parameter	Units	Value
E	Young's modulus	[Pa]	10^4
ν	Poisson's ratio	[-]	0.2
b	Biot's coefficient	[-]	1.0
ϕ	Porosity	[-]	0.3
ρ_f	Fluid density	[kg/m^3]	1.0
c_f	Fluid compressibility	[Pa^{-1}]	0.0
κ	Permeability	[m^2]	10^{-4}
μ	Fluid viscosity	[Pa s]	1.0
$\lvert w\rvert$	Applied compression	[Pa]	1.0
L	Column length	[m]	10.0

Material properties and boundary conditions are specified in the `Constitutive` and `FieldSpecifications` sections. For such set of parameters we have p_0 = 1.0 Pa, c_c = 1.111 m^2 s^{-1}, and t_c = 90 s. Therefore, as shown in the `Events` section, we run this simulation for 90 seconds.

Running GEOS

To run the case, use the following command:

```
path/to/geosx -i inputFiles/poromechanics/PoroElastic_Terzaghi_smoke.xml
```

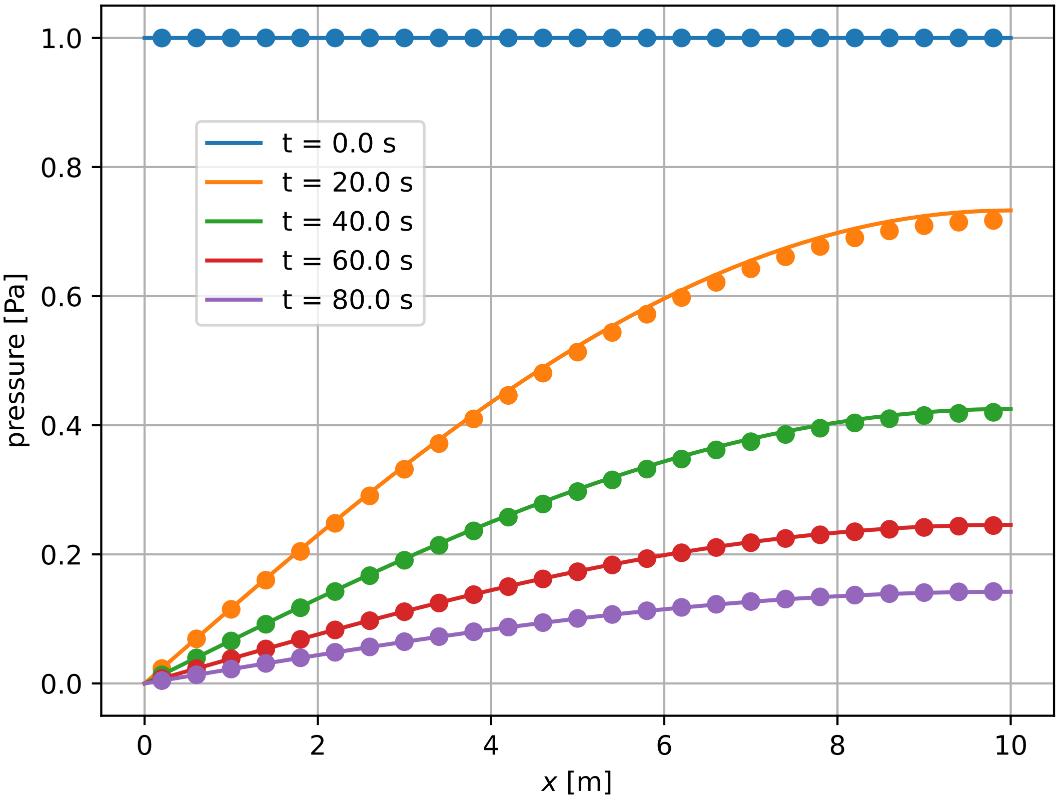
Here, we see for instance the `RSolid` and `RFluid` at a representative timestep (residual values for solid and fluid mechanics solvers, respectively)

```
Attempt:  0, NewtonIter:  0
( RSolid ) = (5.00e-01) ;     ( Rsolid, Rfluid ) = ( 5.00e-01, 0.00e+00 )
( R ) = ( 5.00e-01 ) ;
Attempt:  0, NewtonIter:  1
( RSolid ) = (4.26e-16) ;     ( Rsolid, Rfluid ) = ( 4.26e-16, 4.22e-17 )
( R ) = ( 4.28e-16 ) ;
```

As expected, since we are dealing with a linear problem, the fully implicit solver converges in a single iteration.

Inspecting results

This plot compares the analytical pressure solution (continuous lines) at selected times with the numerical solution (markers).



To go further

Feedback on this example

This concludes the poroelastic example. For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

For more details

- More on poroelastic multiphysics solvers, please see *Poromechanics Solver*.
- More on numerical methods, please see *Numerical Methods*.
- More on functions, please see *Functions*.

1.3.5 Hydraulic Fracturing

Context

In this example, we use a fully coupled hydrofracture solver from GEOS to solve for the propagation of a single fracture within a reservoir with heterogeneous in-situ properties. Advanced xml features will be used throughout the example.

Objectives

At the end of this example you will know:

- how to use multiple solvers for hydraulic fracturing problems,
- how to specify pre-existing fractures and where new fractures can develop,
- how to construct a mesh with bias,
- how to specify heterogeneous in-situ properties and initial conditions,
- how to use parameters, symbolic math, and units in xml files.

Input files

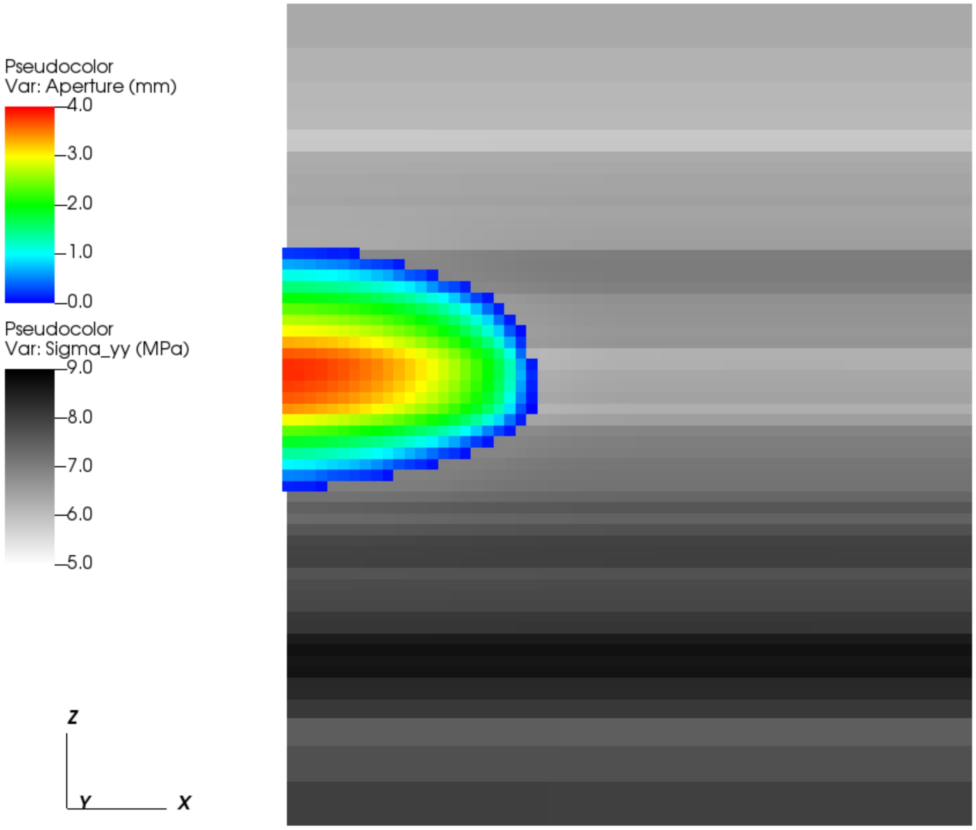
This example uses a set of input files and table files located at:

```
inputFiles/hydraulicFracturing
```

Because the input files use the advanced xml features, they must be preprocessed using the geosx_xml_tools package. If you have not already done so, setup these features by following the instructions here: *Advanced XML Features* .

Description of the case

Here, our goal is to demonstrate how hydraulic fractures are modeled in a typical environment. The in-situ properties and initial conditions are based upon a randomly generated, fractal, 1D layer-cake model.



The inputs for this case are contained inside a case-specific (`heterogeneousInSitu_benchmark.xml`) and base (`heterogeneousInSitu_base.xml`) XML files. The `tables` directory contains the pre-constructed geologic model. This example will first focus on the case-specific input file, which contains the key parameter definitions, then consider the base xml file.

Included: including external xml files

At the head of the case-specific xml file is a block that will instruct GEOS to include an external file. In our case, this points to the base hydraulic fracturing input file.

```
<Included>
  <File
    name="./heterogeneousInSitu_base.xml"/>
</Included>
```

Parameters: defining variables to be used throughout the file

The Parameters block defines a series of variables that can be used throughout the input file. These variables allow a given input file to be easily understood and/or modified for a specific environment, even by non-expert users. Parameters are specified in pairs of names and values. The names should only contain alphanumeric characters and underlines. The values can contain any type (strings, doubles, etc.).

Parameters can be used throughout the input file (or an included input file) by placing them in-between dollar signs. Barring any circular-definition errors, parameters can be used within other parameters. For example, see the parameter mu_upscaled. The value of this parameter is a symbolic expression, which is denoted by the surrounding back-ticks, and is dependent upon two other parameters. During pre-processing, geosx_xml_tools will substitute the parameter definitions, and evaluate the symbolic expression using a python-derived syntax.

A number of the input parameters include optional unit definitions, which are denoted by the square brackets following a value. For example, the parameter t_max is used to set the maximum time for the simulation to 20 minutes.

```
<Parameters>
  <!-- Use the swarm upscaling law -->
  <Parameter
    name="Nperf"
    value="5"/>

  <Parameter
    name="Nswarm"
    value="5"/>

  <Parameter
    name="mu_init"
    value="0.001"/>

  <Parameter
    name="K_init"
    value="1e6"/>

  <Parameter
    name="mu_upscaled"
    value="`$mu_init$*($Nswarm$**2)`"/>

  <Parameter
    name="K_upscaled"
    value="`$K_init$*($Nswarm$**0.5)`"/>

  <Parameter
    name="ContactStiffness"
    value="1e10"/>

  <!-- Event timing -->
  <Parameter
    name="pump_start"
    value="1 [min]"/>

  <Parameter
    name="pump_ramp"
    value="5 [s]"/>
```

(continues on next page)

(continued from previous page)

```
  <Parameter
    name="pump_ramp_dt_limit"
    value="0.2 [s]"/>

  <Parameter
    name="dt_max"
    value="30 [s]"/>

  <Parameter
    name="t_max"
    value="20 [min]"/>

  <!-- Etc. -->
  <Parameter
    name="table_root"
    value="./tables"/>

  <Parameter
    name="t_allocation"
    value="28 [min]"/>
</Parameters>
```

Mesh with biased boundaries

The mesh block for this example uses a biased mesh along the simulation boundaries to reduce the size of the problem, while maintaining the desired spatial extents. For a given element region with bias, the left-hand element will be x% smaller and the right-hand element will be x% larger than the average element size. Along the x-axis of the mesh, we select a value of zero for the first region to indicate that we want a uniform-sized mesh, and we select a bias of -0.6 for the second region to indicate that we want the element size to smoothly increase in size as it moves in the +x direction. The other dimensions of the mesh follow a similar pattern.

```
<Mesh>
  <InternalMesh
    name="mesh1"
    xCoords="{ 0, 200, 250 }"
    yCoords="{ -100, 0, 100 }"
    zCoords="{ -150, -100, 0, 100, 150 }"
    nx="{ 50, 5 }"
    ny="{ 10, 10 }"
    nz="{ 5, 25, 25, 5 }"
    xBias="{ 0, -0.6 }"
    yBias="{ 0.6, -0.6 }"
    zBias="{ 0.6, 0, 0, -0.6 }"
    cellBlockNames="{ cb1 }"
    elementTypes="{ C3D8 }"/>
</Mesh>
```

Defining a fracture nodeset

For this example, we want to propagate a single hydraulic fracture along the plane defined by y = 0. To achieve this, we need to define three nodesets:

- source_a: The location where we want to inject fluid. Typically, we want this to be a single face in the x-z plane.
- perf_a: This is the initial fracture for the simulation. This nodeset needs to be at least two-faces wide in the x-z plane (to represent the fracture at least one internal node needs to be open).
- fracturable_a: This is the set of faces where we will allow the fracture to grow. For a problem where we expect the fracture to curve out of the plane defined by y = 0 , this could be replaced.

```
<Geometry>
  <Box
    name="source_a"
    xMin="{ -0.1, -0.1, -0.1 }"
    xMax="{ 4.1, 0.1, 4.1 }"/>

  <Box
    name="perf_a"
    xMin="{ -4.1, -0.1, -4.1 }"
    xMax="{ 4.1, 0.1, 4.1 }"/>

  <ThickPlane
    name="fracturable_a"
    normal="{ 0, 1, 0 }"
    origin="{ 0, 0, 0 }"
    thickness="0.1"/>
</Geometry>
```

Boundary conditions

The boundary conditions for this problem are defined in the case-specific and the base xml files. The case specific block includes four instructions:

- frac: this marks the initial perforation.
- separableFace: this marks the set of faces that are allowed to break during the simulation.
- waterDensity: this initializes the fluid in the perforation.
- sourceTerm: this instructs the code to inject fluid into the source_a nodeset. Note the usage of the symbolic expression and parameters in the scale. This boundary condition is also driven by a function, which we will define later.

```
<FieldSpecifications>
  <!-- Fracture-related nodesets -->
  <FieldSpecification
    name="frac"
    fieldName="ruptureState"
    initialCondition="1"
    objectPath="faceManager"
    scale="1"
    setNames="{ perf_a }"/>

```

(continues on next page)

(continued from previous page)

```
    <FieldSpecification
      name="separableFace"
      fieldName="isFaceSeparable"
      initialCondition="1"
      objectPath="faceManager"
      scale="1"
      setNames="{ fracturable_a }"/>

    <!-- Fluid Initial Conditions -->
    <FieldSpecification
      name="waterDensity"
      fieldName="water_density"
      initialCondition="1"
      objectPath="ElementRegions"
      scale="1000"
      setNames="{ perf_a }"/>

    <!-- Fluid BC's -->
    <!-- Note: the units of the source flux BC are in kg/s not m3/s -->
    <SourceFlux
      name="sourceTerm"
      objectPath="ElementRegions/Fracture"
      scale="`-1000.0/$Nperf$`"
      functionName="flow_rate"
      setNames="{ source_a }"/>
  </FieldSpecifications>
```

The base block includes instructions to set the initial in-situ properties and stresses. It is also used to specify the external mechanical boundaries on the system. In this example, we are using roller-boundary conditions (zero normal-displacement). Depending upon how close they are to the fracture, they can significantly affect its growth. Therefore, it is important to test whether the size of the model is large enough to avoid this.

```
  <FieldSpecifications>
    <FieldSpecification
      name="bulk_modulus"
      initialCondition="1"
      setNames="{ all }"
      objectPath="ElementRegions"
      fieldName="rock_bulkModulus"
      functionName="bulk_modulus"
      scale="1.0"/>

    <FieldSpecification
      name="shear_modulus"
      initialCondition="1"
      setNames="{ all }"
      objectPath="ElementRegions"
      fieldName="rock_shearModulus"
      functionName="shear_modulus"
      scale="1.0"/>

    <FieldSpecification
```

(continues on next page)

(continued from previous page)

```
    name="sigma_xx"
    initialCondition="1"
    setNames="{ all }"
    objectPath="ElementRegions"
    fieldName="rock_stress"
    component="0"
    functionName="sigma_xx"
    scale="1.0"/>

  <FieldSpecification
    name="sigma_yy"
    initialCondition="1"
    setNames="{ all }"
    objectPath="ElementRegions"
    fieldName="rock_stress"
    component="1"
    functionName="sigma_yy"
    scale="1.0"/>

  <FieldSpecification
    name="sigma_zz"
    initialCondition="1"
    setNames="{ all }"
    objectPath="ElementRegions"
    fieldName="rock_stress"
    component="2"
    functionName="sigma_zz"
    scale="1.0"/>

  <!-- Mechanical BC's -->
  <FieldSpecification
    name="x_constraint"
    component="0"
    fieldName="totalDisplacement"
    objectPath="nodeManager"
    scale="0.0"
    setNames="{ xneg, xpos }"/>

  <FieldSpecification
    name="y_constraint"
    component="1"
    fieldName="totalDisplacement"
    objectPath="nodeManager"
    scale="0.0"
    setNames="{ yneg, ypos }"/>

  <FieldSpecification
    name="z_constraint"
    component="2"
    fieldName="totalDisplacement"
    objectPath="nodeManager"
    scale="0.0"
```

(continues on next page)

(continued from previous page)

```
    setNames="{ zneg, zpos }"/>
</FieldSpecifications>
```

Coupled hydraulic fracturing solver

The Solvers block is located in the base xml file. Note that the `gravityVector` attribute indicates that we are applying gravity in the z-direction in this problem.

Similar to other coupled physics solvers, the Hydrofracture solver is specified in three parts:

- Hydrofracture: this is the primary solver, which will be called by the event manager. Two of its key attributes are the names of the dependent solid and fluid solvers.
- SolidMechanicsLagrangianSSLE: this is the solid mechanics solver.
- SinglePhaseFVM: this is the fluid solver.

The final solver present in this example is the SurfaceGenerator, which manages how faces in the model break.

```
<Solvers
  gravityVector="{ 0.0, 0.0, -9.81 }">
  <Hydrofracture
    name="hydrofracture"
    solidSolverName="lagsolve"
    flowSolverName="SinglePhaseFlow"
    surfaceGeneratorName="SurfaceGen"
    logLevel="1"
    targetRegions="{ Fracture }"
    contactRelationName="fractureContact"
    maxNumResolves="2">
    <NonlinearSolverParameters
      newtonTol="1.0e-6"
      newtonMaxIter="40"
      lineSearchMaxCuts="3"/>
    <LinearSolverParameters
      solverType="gmres"
      preconditionerType="amg"/>
  </Hydrofracture>

  <SolidMechanicsLagrangianSSLE
    name="lagsolve"
    timeIntegrationOption="QuasiStatic"
    discretization="FE1"
    targetRegions="{ Domain, Fracture }"
    contactRelationName="fractureContact">
    <NonlinearSolverParameters
      newtonTol="1.0e-6"/>
    <LinearSolverParameters
      solverType="gmres"
      krylovTol="1.0e-10"/>
  </SolidMechanicsLagrangianSSLE>

  <SinglePhaseFVM
```

(continues on next page)

(continued from previous page)

```
      name="SinglePhaseFlow"
      discretization="singlePhaseTPFA"
      targetRegions="{ Fracture }">
      <NonlinearSolverParameters
        newtonTol="1.0e-5"
        newtonMaxIter="10"/>
      <LinearSolverParameters
        solverType="gmres"
        krylovTol="1.0e-12"/>
    </SinglePhaseFVM>

    <SurfaceGenerator
      name="SurfaceGen"
      targetRegions="{ Domain }"
      rockToughness="$K_upscaled$"
      nodeBasedSIF="1"
      mpiCommOrder="1"/>
  </Solvers>
```

Events

Rather than explicitly specify the desired timestep behavior, this example uses a flexible approach for timestepping. The hydrofracture solver is applied in the `solverApplications` event, which request a `maxEventDt = 30 s`. To maintain stability during the critical early phase of the model, we delay turning on the pump by `pump_start`. We then use the `pumpStart` event to limit the further limit the timestep to `pump_ramp_dt_limit` as the fracture experiences rapid development (`pump_start` to `pump_start + pump_ramp`). Note that while this event does not have a target, it can still influence the time step behavior. After this period, the hydraulic fracture solver will attempt to increase / decrease the requested timestep to maintain stability.

Other key events in this problem include:

- preFracture: this calls the surface generator at the beginning of the problem and helps to initialize the fracture.
- outputs_vtk and outputs_silo: these produces output vtk and silo files.
- restarts (inactive): this is a HaltEvent, which tracks the external clock. When the runtime exceeds the specified value (here $t_allocation$=28 minutes), the code will call the target (which writes a restart file) and instruct the code to exit.

```
  <Events
    maxTime="$t_max$"
    logLevel="1">
    <!-- Generate the initial fractures -->
    <SoloEvent
      name="preFracture"
      target="/Solvers/SurfaceGen"/>

    <!-- Primary outputs -->
    <PeriodicEvent
      name="outputs_vtk"
      timeFrequency="1 [min]"
      targetExactTimestep="0"
      target="/Outputs/vtkOutput"/>
```

(continues on next page)

(continued from previous page)

```
    <PeriodicEvent
      name="outputs_silo"
      timeFrequency="1 [min]"
      targetExactTimestep="0"
      target="/Outputs/siloOutput"/>

    <!-- Apply the hydrofracture solver -->
    <PeriodicEvent
      name="solverApplications"
      maxEventDt="$dt_max$"
      target="/Solvers/hydrofracture"/>

    <!-- Limit dt during the pump ramp-up -->
    <PeriodicEvent
      name="pumpStart"
      beginTime="$pump_start$"
      endTime="`$pump_start$+$pump_ramp$`"
      maxEventDt="$pump_ramp_dt_limit$"/>

    <!-- Watch the wall-clock, write a restart, and exit gracefully if necessary -->
    <!-- <HaltEvent
      name="restarts"
      maxRuntime="$t_allocation$"
      target="/Outputs/restartOutput"/> -->
  </Events>
```

Functions to set in-situ properties

The function definitions are in the base xml file, and rely upon the files in the tables directory. The functions in this example include the flow rate over time, the in-situ principal stress, and the bulk/shear moduli of the rock. Note the use of the table_root parameter, which contains the root path to the table files.

The flow_rate TableFunction is an example of a 1D function. It has a single input, which is time. The table is defined using a single coordinateFile:

```
0.00000e+00
6.00000e+01
1.20000e+02
3.72000e+03
3.78000e+03
1.00000e+09
```

And a single voxelFile:

```
0.00000e+00
0.00000e+00
5.00000e-02
5.00000e-02
0.00000e+00
0.00000e+00
```

Given the specified linear interpolation method, these values define a simple trapezoidal function. Note: since this is a 1D table, these values could alternately be given within the xml file using the `coordinates` and `values` attributes.

The sigma_xx TableFunction is an example of a 3D function. It uses `elementCenter` as its input, which is a vector. It is specified using a set of three coordinate files (one for each axis), and a single voxel file. The geologic model in this example is a layer-cake, which was randomly generated, so the size of the x and y axes are 1. The interpolation method used here is upper, so the values in the table indicate those at the top of each layer.

```xml
<Functions>
  <!-- Pumping Schedule -->
  <TableFunction
    name="flow_rate"
    inputVarNames="{ time }"
    coordinateFiles="{ $table_root$/flowRate_time.csv }"
    voxelFile="$table_root$/flowRate.csv"/>

  <!-- Geologic Model -->
  <TableFunction
    name="sigma_xx"
    inputVarNames="{ elementCenter }"
    coordinateFiles="{ $table_root$/x.csv, $table_root$/y.csv, $table_root$/z.csv }"
    voxelFile="$table_root$/sigma_xx.csv"
    interpolation="upper"/>

  <TableFunction
    name="sigma_yy"
    inputVarNames="{ elementCenter }"
    coordinateFiles="{ $table_root$/x.csv, $table_root$/y.csv, $table_root$/z.csv }"
    voxelFile="$table_root$/sigma_yy.csv"
    interpolation="upper"/>

  <TableFunction
    name="sigma_zz"
    inputVarNames="{ elementCenter }"
    coordinateFiles="{ $table_root$/x.csv, $table_root$/y.csv, $table_root$/z.csv }"
    voxelFile="$table_root$/sigma_zz.csv"
    interpolation="upper"/>

  <TableFunction
    name="init_pressure"
    inputVarNames="{ elementCenter }"
    coordinateFiles="{ $table_root$/x.csv, $table_root$/y.csv, $table_root$/z.csv }"
    voxelFile="$table_root$/porePressure.csv"
    interpolation="upper"/>

  <TableFunction
    name="bulk_modulus"
    inputVarNames="{ elementCenter }"
    coordinateFiles="{ $table_root$/x.csv, $table_root$/y.csv, $table_root$/z.csv }"
    voxelFile="$table_root$/bulkModulus.csv"
    interpolation="upper"/>

  <TableFunction
    name="shear_modulus"
```

(continues on next page)

(continued from previous page)

```
      inputVarNames="{ elementCenter }"
      coordinateFiles="{ $table_root$/x.csv, $table_root$/y.csv, $table_root$/z.csv }"
      voxelFile="$table_root$/shearModulus.csv"
      interpolation="upper"/>

    <TableFunction
      name="apertureTable"
      coordinates="{ -1.0e-3, 0.0 }"
      values="{ 1.0e-6, 1.0e-4 }"/>

  </Functions>
```

Running GEOS

Assuming that the preprocessing tools have been correctly installed (see *Advanced XML Features*), there will be a script in the GEOS build/bin directory called *geosx_preprocessed*. Replacing *geosx* with *geosx_preprocessed* in an input command will automatically apply the preprocessor and send the results to GEOS.

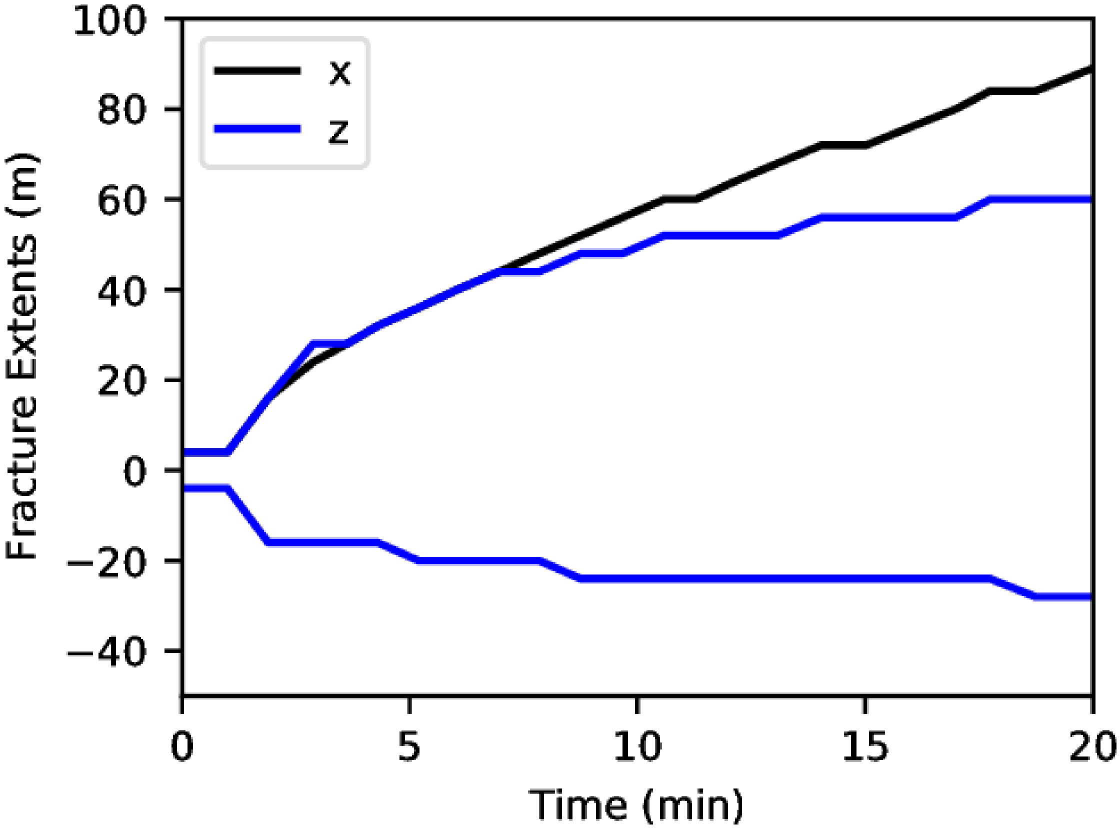
Before beginning, we reccomend that you make a local copy of the example and its tables. Because we are using advanced xml features in this example, the input file must be pre-processed before running. For example, this will run the code on a debug partition using a total of 36 cores.

```
cp -r examples/hydraulicFracturing ./hf_example
cd hf_example
srun -n 36 -ppdebug geosx_preprocessed -i heterogeneousInSitu_benchmark.xml -x 6 -y 2 -z␣
↪3 -o hf_results
```

Note that as part of the automatic preprocessing step a compiled xml file is written to the disk (by default '[input_name].preprocessed'). When developing an xml with advanced features, we reccomend that you check this file to ensure its accuracy.

Inspecting results

In the above example, we requested vtk- and silo-format output files every minute. We can therefore import these into VisIt, Paraview, or python and visualize the outcome. The following figure shows the extents of the generated fracture over time:

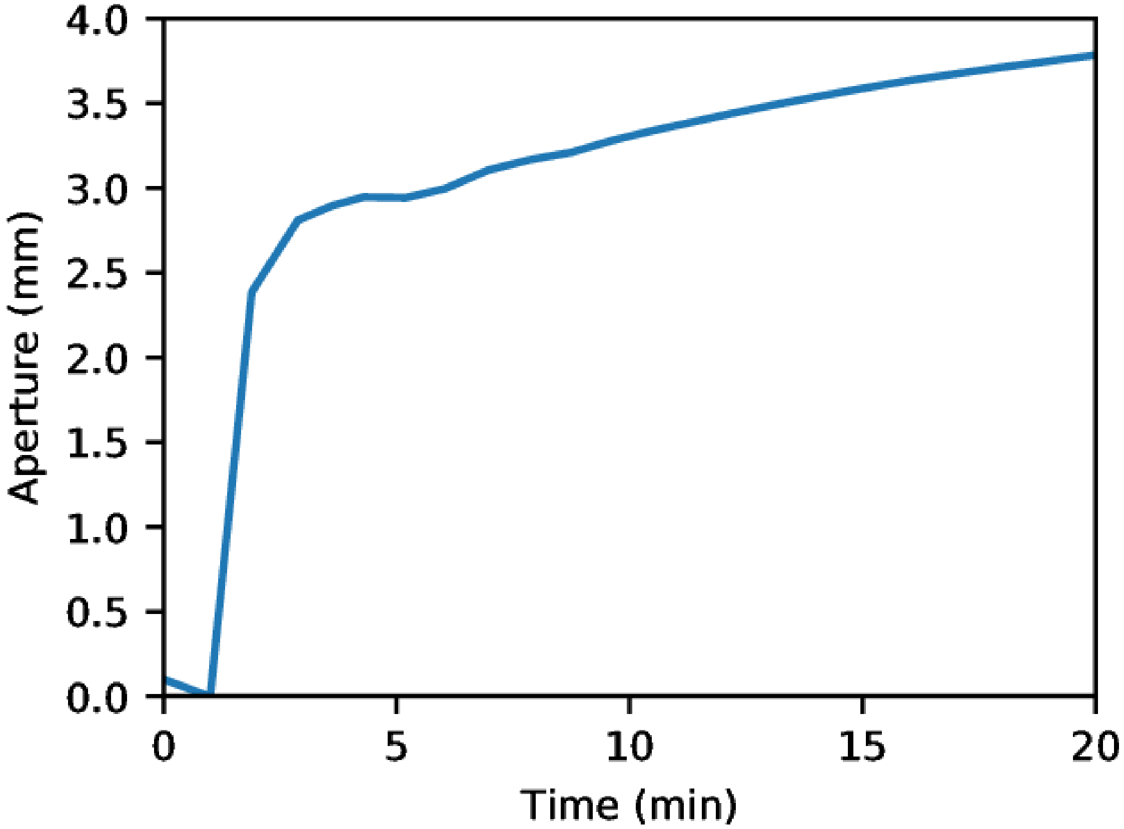


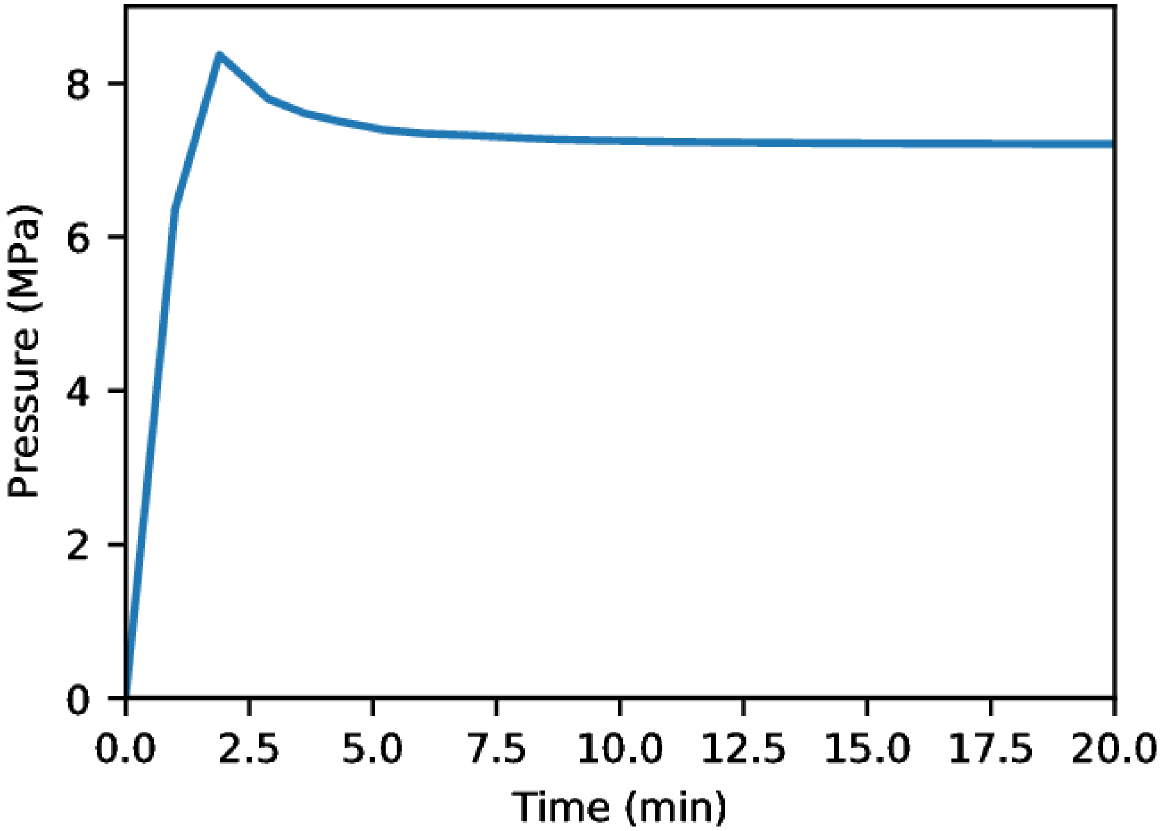
Notes for visualization tools:

1) In Visit, we currently recommend that you look at the silo-format files (due to a compatibility issue with vtk)
2) In Paraview, you may need to use the Multi-block Inspector (on the right-hand side of the screen by default) to limit the visualization to the fracture. In addition, the Properties inspector (on the left-hand side of the sceen by default) may not include some of the parameters present on the fracture. Instead, we recommend that you use the property dropdown box at the top of the screen.

Because we did not explicitly specify any fracture barriers in this example, the fracture dimensions are controlled by the in-situ stresses. During the first couple of minutes of growth, the fracture quickly reaches its maximum/minimum height, which corresponds to a region of low in-situ minimum stress.

The following figures show the aperture and pressure of the hydraulic fracture (near the source) over time:


4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Aperture (mm)
0
5
10
15
20
Time (min)



Modifying Parameters Via the Command-Line

The advanced xml feature preprocessor allows parameters to be set or overriden by specifying any number of *-p name value* arguments on the command-line. Note that if the parameter value has spaces, it needs to be enclosed by quotation marks.

To illustrate this feature, we can re-run the previous analysis with viscosity increased from 1 cP to 5 cP:

```
srun -n 36 -ppdebug geosx_preprocessed -i heterogeneousInSitu_benchmark.xml -p mu 0.005 -
↪x 6 -y 2 -z 3 -o hf_results_lower_mu
```

To go further

Feedback on this example

This concludes the hydraulic fracturing example. For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

For more details

- More on advanced xml features, please see *Advanced XML Features*.
- More on functions, please see *Functions*.
- More on biased meshes, please see *Mesh Bias*.

1.3.6 Triaxial Driver: Extended Drucker-Prager Elasto-Plastic Model

Context

In this example, we use the `TriaxialDriver` inside GEOS to simulate rock mechanics experiments, such as triaxial tests. The triaxial driver allows to calibrate properties and model parameters against experimental data before using them in field-scale simulations.

Objectives

At the end of this example, you will know:

- how to set up a triaxial test scenario with the `TriaxialDriver`,
- how to define a material model,
- how to specify loading conditions with `TableFunction`,
- how to save and postprocess simulation results.

Input file

The XML file for this test case is located at:

```
inputFiles/triaxialDriver/triaxialDriver_ExtendedDruckerPrager_basicExample.xml
```

This example also uses a set of table files located at:

```
inputFiles/triaxialDriver/tables/
```

Last, a Python script for post-processing the results is provided:

```
src/docs/sphinx/basicExamples/triaxialDriver/triaxialDriverFigure.py
```

Description of the case

Instead of launching a full finite-element simulation to mimic experimental loading conditions, GEOS provides a `TriaxialDriver` to investigate constitutive behaviors and simulate laboratory tests. The triaxial driver makes it easy to interpret the mechanical response and calibrate the constitutive models against experimental data.

In this example, the Extended Drucker-Prager model (see *Model: Extended Drucker-Prager*) is used to solve elasto-plastic deformations of rock samples when subject to controlled loading conditions. The strain-hardening Extended Drucker-Prager model with an associated plastic flow rule is tested in this example. To replicate a typical triaxial test, we use a table function to specify loading conditions in axial and radial directions. The resulting strains and stresses in both directions are numerically calculated and saved into a simple ASCII output file.

For this example, we focus on the `Task`, the `Constitutive`, and the `Functions` tags.

Task

In GEOS, the `TriaxialDriver` is defined with a dedicated XML structure. The `TriaxialDriver` is added to a standard XML input deck as a solo task to the `Tasks` queue and added as a `SoloEvent` to the event queue.

For this example, we simulate the elastoplastic deformation of a confined specimen caused by external load. A homogeneous domain with one solid material is assumed. The material is named `ExtendedDruckerPrager`, and its mechanical properties are specified in the `Constitutive` section.

Different testing modes are available in the `TriaxialDriver` to mimic different laboratory loading conditions:

mode	axial loading	radial loading	initial stress
strainContr	axial strain controlled with `axialControl`	radial strain controlled with `radialControl`	isotropic stress using `initialStress`
stressContr	axial stress controlled with `axialControl`	radial stress controlled with `radialControl`	isotropic stress using `initialStress`
mixedContrc	axial strain controlled with `axialControl`	radial stress controlled with `radialControl`	isotropic stress using `initialStress`

A triaxial test is usually conducted on a confined rock sample to determine material properties. As shown, a conventional triaxial test is described using the `mode="mixedControl"` testing mode in the `TriaxialDriver`.

In a triaxial test, the testing sample is under confining pressure (radial stresses) and subject to increased axial load. Therefore, a stress control `radialControl="stressFunction"` is defined in the radial direction to impose confining pressure. A strain control `axialControl="strainFunction"` is applied in the axial direction to represent axial compression.

The initial stress is specified by `initialStress="-10.e6"`. To ensure static equilibrium at the first timestep, its value should be consistent with the initial set of applied stresses defined in axial or radial loading functions. This stress has a negative value due to the negative sign convention for compressive stress in GEOS.

Then, `steps="200"` defines the number of load steps and `output="simulationResults.txt"` specifies an output file to which the simulation results will be written.

```
<Tasks>
  <TriaxialDriver
    name="triaxialDriver"
    material="ExtendedDruckerPrager"
    mode="mixedControl"
    axialControl="strainFunction"
    radialControl="stressFunction"
    initialStress="-10.e6"
    steps="200"
    output="simulationResults.txt" />
</Tasks>
```

In addition to triaxial tests, volumetric and oedometer tests can be simulated by changing the `strainControl` mode, and by defining loading controls in axial and radial direction accordingly. A volumetric test can be modelled by setting the axial and radial control functions to the same strain function, whereas an oedometer test runs by setting the radial strain to zero (see *Triaxial Driver*).

Constitutive laws

Any solid material model implemented in GEOS can be called by the `TriaxialDriver`.

For this problem, Extended Drucker Prager model `ExtendedDruckerPrager` is used to describe the mechanical behavior of an isotropic material, when subject to external loading. As for the material parameters, `defaultInitialFrictionAngle`, `defaultResidualFrictionAngle` and `defaultCohesion` denote the initial friction angle, the residual friction angle, and cohesion, respectively, as defined by the Mohr-Coulomb failure envelope. As the residual friction angle `defaultResidualFrictionAngle` is larger than the initial one `defaultInitialFrictionAngle`, a strain hardening model is adopted, whose hardening rate is given as `defaultHardening="0.0001"`. If the residual friction angle is set to be less than the initial one, strain weakening will take place. Setting `defaultDilationRatio="1.0"` corresponds to an associated flow rule.

```
<Constitutive>
  <ExtendedDruckerPrager
    name="ExtendedDruckerPrager"
    defaultDensity="2700"
    defaultBulkModulus="10.0e9"
    defaultShearModulus="6.0e9"
    defaultCohesion="6.0e6"
    defaultInitialFrictionAngle="16.0"
    defaultResidualFrictionAngle="20.0"
    defaultDilationRatio="1.0"
    defaultHardening="0.0001"
  />
</Constitutive>
```

All constitutive parameters such as density, viscosity, bulk modulus, and shear modulus are specified in the International System of Units.

Functions

The `TriaxialDriver` uses a simple form of time-stepping to advance through the loading steps, allowing for simulating both rate-dependent and rate-independent models.

In this case, users should define two different time history functions (`strainFunction` and `stressFunction`) to describe loading conditions in axial and radial direction respectively. More specifically, the table functions in this example include the temporal variations of radial stress and axial strain, which rely upon the external files in the table directory (see *Functions*). Note that for standard tests with simple loading history, functions can be embedded directly in the XML file without using external tables.

```
<Functions>
  <TableFunction
    name="strainFunction"
    inputVarNames="{ time }"
    coordinateFiles="{ tables/time.geos }"
    voxelFile="tables/axialStrain.geos"/>

  <TableFunction
    name="stressFunction"
    inputVarNames="{ time }"
    coordinateFiles="{ tables/time.geos }"
    voxelFile="tables/radialStress.geos"/>
</Functions>
```

The `strainFunction` TableFunction is an example of a 1D interpolated function, which describes the strain as a function of time `inputVarNames="{ time }"`. This table is defined using a single coordinate file:

```
0
1
2
3
4
5
```

And a single voxel file:

```
0.0
-0.004
-0.002
-0.005
-0.003
-0.006
```

Similarly, the correlation between the confining stress and time is given through the `stressFunction` defined using the same coordinate file:

```
0
1
2
3
4
5
```

And a different voxel file:

```
-10.0e6
-10.0e6
-10.0e6
-10.0e6
-10.0e6
-10.0e6
```

For this specific test, the confining stress is kept constant and equal to the `initialStress`. Instead of monotonic changing the axial load, two loading/unloading cycles are specified in the `strainFunction`. This way, both plastic loading and elastic unloading can be modeled.

Note that by convention in GEOS, `stressFunction` and `strainFunction` have negative values for a compressive test.

Mesh

Even if discretization is not required for the `TriaxialDriver`, a dummy mesh should be defined to pass all the necessary checks when initializing GEOS and running the module. A dummy mesh should be created in the `Mesh` section and assigned to the `cellBlocks` in the `ElementRegions` section.

```
  <Mesh>
    <InternalMesh
      name="mesh1"
      elementTypes="{ C3D8 }"
      xCoords="{ 0, 1 }"
      yCoords="{ 0, 1 }"
      zCoords="{ 0, 1 }"
      nx="{ 1 }"
      ny="{ 1 }"
      nz="{ 1 }"
      cellBlockNames="{ cellBlock01 }"/>
  </Mesh>

  <ElementRegions>
```

(continues on next page)

(continued from previous page)

```
    <CellElementRegion
      name="dummy"
      cellBlocks="{ cellBlock01 }"
      materialList="{ dummy }"/>
  </ElementRegions>
```

Once calibrated, the testing constitutive models can be easily used in full field-scale simulation by adding solver, discretization, and boundary condition blocks to the xml file. Also, it is possible to run a full GEOS model and generate identical results as those provided by the `TriaxialDriver`.

Running TriaxialDriver

The `TriaxialDriver` is launched like any other GEOS simulation by using the following command:

```
path/to/geosx -i triaxialDriver_ExtendedDruckerPrager_basicExample.xml
```

The running log appears to the console to indicate if the case can be successfully executed or not:

```
Max threads: 32
MKL max threads: 16
GEOS version 0.2.0 (HEAD, sha1: bb16d72)
Adding Event: SoloEvent, triaxialDriver
   TableFunction: strainFunction
   TableFunction: stressFunction
Adding Mesh: InternalMesh, mesh1
Adding Object CellElementRegion named dummy from ObjectManager::Catalog.
Total number of nodes:8
Total number of elems:1
Rank 0: Total number of nodes:8
        dummy/cellBlock01 does not have a discretization associated with it.
Time: 0s, dt:1s, Cycle: 0
Cleaning up events
Umpire            HOST sum across ranks:   23.2 KB
Umpire            HOST         rank max:   23.2 KB
total time                             0.435s
initialization time                    0.053s
run time                               0.004s
```

Inspecting results

The simulation results are saved in a text file, named `simulationResults.txt`. This output file has a brief header explaining the meaning of each column. Each row corresponds to one timestep of the driver, starting from initial conditions in the first row.

```
# column 1 = time
# column 2 = axial_strain
# column 3 = radial_strain_1
# column 4 = radial_strain_2
# column 5 = axial_stress
# column 6 = radial_stress_1
# column 7 = radial_stress_2
```

(continues on next page)

(continued from previous page)

```
# column 8 = newton_iter
# column 9 = residual_norm
0.0000e+00 0.0000e+00 0.0000e+00 0.0000e+00 -1.0000e+07 -1.0000e+07 -1.0000e+07 0.
↪0000e+00 0.0000e+00
2.5000e-02 -1.0000e-04 2.5000e-05 2.5000e-05 -1.1500e+07 -1.0000e+07 -1.0000e+07 1.
↪0000e+00 0.0000e+00
5.0000e-02 -2.0000e-04 5.0000e-05 5.0000e-05 -1.3000e+07 -1.0000e+07 -1.0000e+07 1.
↪0000e+00 0.0000e+00
7.5000e-02 -3.0000e-04 7.5000e-05 7.5000e-05 -1.4500e+07 -1.0000e+07 -1.0000e+07 1.
↪0000e+00 0.0000e+00
1.0000e-01 -4.0000e-04 1.0000e-04 1.0000e-04 -1.6000e+07 -1.0000e+07 -1.0000e+07 1.
↪0000e+00 0.0000e+00
...
```

Note that the file contains two columns for radial strains (`radial_strain_1` and `radial_strain_2`) and two columns for radial stresses (`radial_stress_1` and `radial_stress_2`). For isotropic materials, the stresses and strains along the two radial axes would be the same. However, the stresses and strains in the radial directions can differ for cases with anisotropic materials and true-triaxial loading conditions.

This output file can be processed and visualized using any tool. As an example here, with the provided python script, the simulated stress-strain curve, p-q diagram and relationship between volumetric strain and axial strain are plotted, and used to validate results against experimental observations:

To go further

Feedback on this example

For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

1.4 Advanced Examples

1.4.1 Validation and Verification Studies

Carbon Storage

Verification of CO2 Core Flood Experiment with Buckley-Leverett Solution

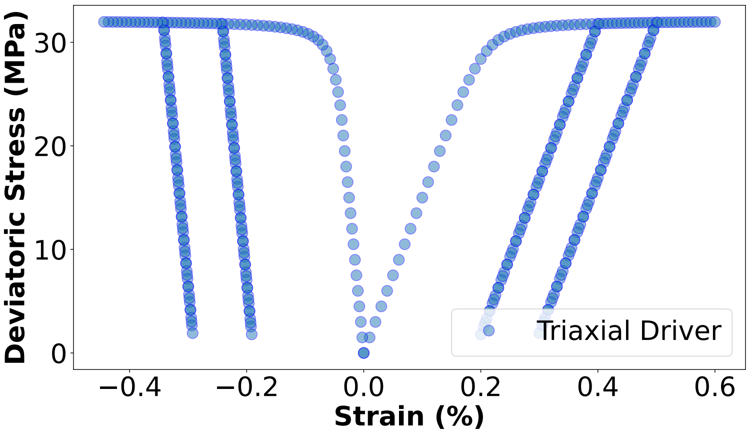
Context

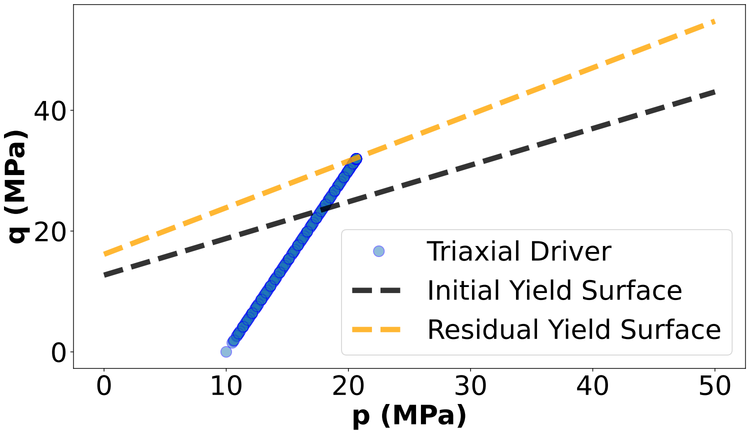
In this example, we simulate a CO2 core flood experiment representing immiscible transport of two-phase flow (CO2 and water) through porous media (Ekechukwu et al., 2022). This problem is solved using the multiphase flow solver in GEOS to obtain the temporal evolution of saturation along the flow direction, and verified against the Buckley-Leverett analytical solutions (Buckley and Leverett, 1942; Arabzai and Honma, 2013).

Input file

The xml input files for the test case are located at:

```
inputFiles/compositionalMultiphaseFlow/benchmarks/buckleyLeverettProblem/buckleyLeverett_
↪base.xml
inputFiles/compositionalMultiphaseFlow/benchmarks/buckleyLeverettProblem/buckleyLeverett_
↪benchmark.xml
```


Deviatoric Stress (MPa)
0
10
20
30
−0.4
−0.2
0.0
0.2
0.4
0.6
Strain (%)
Triaxial Driver


q (MPa)
0
20
40
0
10
20
30
40
50
p (MPa)
Triaxial Driver
Initial Yield Surface
Residual Yield Surface

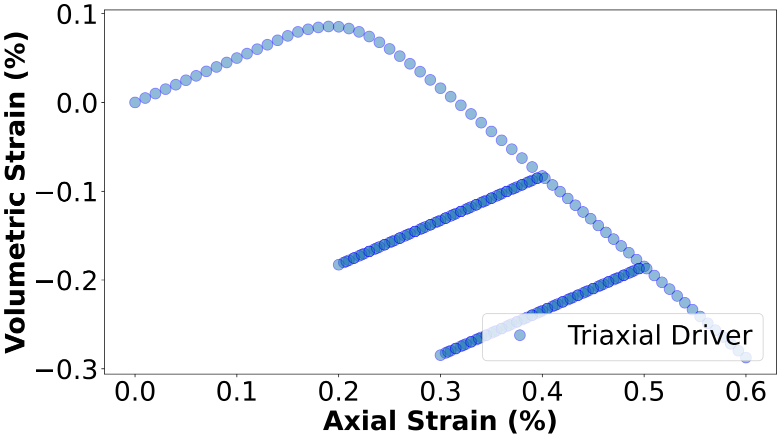

Volumetric Strain (%)
0.1
0.0
−0.1
−0.2
−0.3
0.0
0.1
0.2
0.3
0.4
0.5
0.6
Axial Strain (%)
Triaxial Driver

Table files and a Python script for post-processing the simulation results are provided:

```
inputFiles/compositionalMultiphaseFlow/benchmarks/buckleyLeverettProblem/buckleyLeverett_
↪table
```

```
src/docs/sphinx/advancedExamples/validationStudies/carbonStorage/buckleyLeverett/
↪buckleyLeverettFigure.py
```

Description of the case

We model the immiscible displacement of brine by CO2 in a quasi one-dimensional domain that mimics a CO2 core flood experiment, as shown below. The domain is horizontal, homogeneous, isotropic and isothermal. Prior to injection, the domain is fully saturated with brine. To match the analytical example configuration, supercritical CO2 is injected from the inlet and a constant flow rate is imposed. To meet the requirements of immiscible transport in one-dimensional domain, we assume linear and horizontal flow, incompressible and immiscible phases, negligible capillary pressure and gravitational forces, and no poromechanical effects. Upon injection, the saturation front of the injected phase (supercritical CO2) forms a sharp leading edge and advances with time.

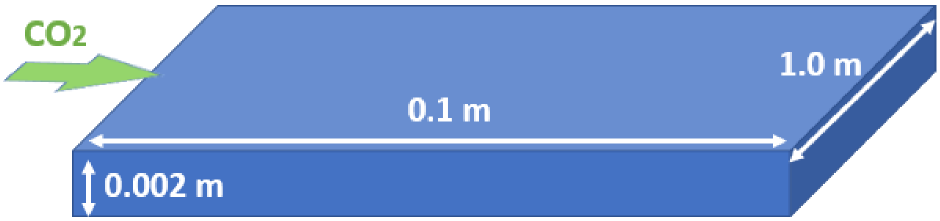


Fig. 1.2: Sketch of the problem

We set up and solve a multiphase flow model to obtain the spatial and temporal solutions of phase saturations and pore pressures across the domain upon injection. Saturation profiles along the flow direction are evaluated and compared with their corresponding analytical solutions (Arabzai and Honma, 2013).

A power-law Brooks-Corey relation is used here to describe gas k_{rg} and water k_{rw} relative permeabilities:

$$k_{rg} = {k_{rg}}^0 ({S_g}^\star)^{n_g}$$

$$k_{rw} = {k_{rw}}^0 ({S_w}^\star)^{n_w}$$

where ${k_{rg}}^0$ and ${k_{rw}}^0$ are the maximum relative permeability of gas and water phase respectively; n_g and n_w are the Corey exponents; dimensionless volume fraction (saturation) of gas phase ${S_g}^\star$ and water phase ${S_w}^\star$ are given as:

$${S_g}^\star = \frac{S_g - S_{gr}}{1 - S_{gr} - S_{wr}}$$

$${S_w}^\star = \frac{S_w - S_{wr}}{1 - S_{gr} - S_{wr}}$$

where S_{gr} and S_{wr} are the residual gas and water saturation;

According to the Buckley–Leverett theory with constant fluid viscosity, the fractional flow of gas phase f_g can be expressed as:

$$f_g = \frac{\frac{k_{rg}}{\mu_g}}{\frac{k_{rg}}{\mu_g} + \frac{k_{rw}}{\mu_w}}$$

where μ_g and μ_w represent the viscosity of gas and water phase respectively, assumed to be constant in this study.

The position of a particular saturation is given as a function of the injection time t and the value of the derivative $\frac{df_g}{dS_g}$ at that saturation:

$$x_{S_g} = \frac{Q_T t}{A\phi}\left(\frac{df_g}{dS_g}\right)$$

where Q_T is the total flow rate, A is the area of the cross-section in the core sample, ϕ is the rock porosity. In addition, the abrupt saturation front is determined based on the tangent point on the fractional flow curve.

For this example, we focus on the `Mesh`, the `Constitutive`, and the `FieldSpecifications` tags.

Mesh

The mesh was created with the internal mesh generator and parametrized in the `InternalMesh` XML tag. It contains 1000x1x1 eight-node brick elements in the x, y, and z directions respectively. Such eight-node hexahedral elements are defined as `C3D8` elementTypes, and their collection forms a mesh with one group of cell blocks named here `cellBlock`. The width of the domain should be large enough to ensure the formation of a one-dimension flow.

```
<Mesh>
  <InternalMesh
    name="mesh"
    elementTypes="{ C3D8 }"
    xCoords="{ 0, 0.1 }"
    yCoords="{ 0, 0.00202683 }"
    zCoords="{ 0, 1 }"
    nx="{ 1000 }"
    ny="{ 1 }"
    nz="{ 1 }"
    cellBlockNames="{ cellBlock }"/>
</Mesh>
```

Flow solver

The isothermal immiscible simulation is performed using the GEOS general-purpose multiphase flow solver. The multiphase flow solver, a solver of type `CompositionalMultiphaseFVM` called here `compflow` (more information on these solvers at *Compositional Multiphase Flow Solver*) is defined in the XML block **CompositionalMultiphaseFVM**:

```
<Solvers>
  <CompositionalMultiphaseFVM
    name="compflow"
    logLevel="1"
    discretization="fluidTPFA"
    temperature="300"
    initialDt="0.001"
```

(continues on next page)

(continued from previous page)

```
    useMass="1"
    targetRegions="{ region }">
    <NonlinearSolverParameters
      newtonTol="1.0e-6"
      newtonMaxIter="50"
      maxTimeStepCuts="2"
      lineSearchMaxCuts="2"/>
    <LinearSolverParameters
      solverType="direct"
      directParallel="0"
      logLevel="0"/>
  </CompositionalMultiphaseFVM>
</Solvers>
```

We use the `targetRegions` attribute to define the regions where the flow solver is applied. Here, we only simulate fluid flow in one region named as `region`. We specify the discretization method (`fluidTPFA`, defined in the `NumericalMethods` section), and the initial reservoir temperature (`temperature="300"`).

Constitutive laws

This benchmark example involves an immiscible, incompressible, two-phase model, whose fluid rheology and permeability are specified in the `Constitutive` section. The best approach to represent this fluid behavior in GEOS is to use the **DeadOilFluid** model in GEOS.

```
<Constitutive>
  <CompressibleSolidConstantPermeability
    name="rock"
    solidModelName="nullSolid"
    porosityModelName="rockPorosity"
    permeabilityModelName="rockPerm"/>

  <NullModel
    name="nullSolid"/>

  <PressurePorosity
    name="rockPorosity"
    defaultReferencePorosity="0.2"
    referencePressure="1e7"
    compressibility="1.0e-15"/>

  <ConstantPermeability
    name="rockPerm"
    permeabilityComponents="{ 9.0e-13, 9.0e-13, 9.0e-13}"/>

  <BrooksCoreyRelativePermeability
    name="relperm"
    phaseNames="{ gas, water }"
    phaseMinVolumeFraction="{ 0.0, 0.0 }"
    phaseRelPermExponent="{ 3.5, 3.5 }"
    phaseRelPermMaxValue="{ 1.0, 1.0 }"/>

```

(continues on next page)

(continued from previous page)

```
    <DeadOilFluid
      name="fluid"
      phaseNames="{ gas, water }"
      surfaceDensities="{ 280.0, 992.0 }"
      componentMolarWeight="{ 44e-3, 18e-3 }"
      tableFiles="{ buckleyLeverett_table/pvdg.txt, buckleyLeverett_table/pvtw.txt }"/>
  </Constitutive>
```

Constant fluid densities and viscosities are given in the external tables for both phases. The formation volume factors are set to 1 for incompressible fluids. The relative permeability for both phases are modeled with the power-law correlations BrooksCoreyRelativePermeability (more information at *Brooks-Corey relative permeability model*), as shown below. Capillary pressure is assumed to be negligible.

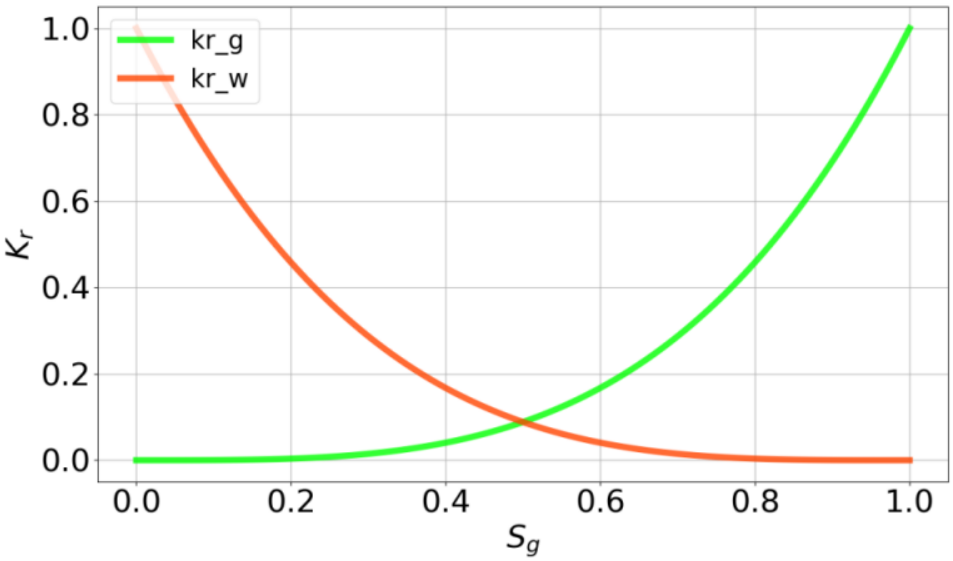


Fig. 1.3: Relative permeabilities of both phases

All constitutive parameters such as density, viscosity, and permeability are specified in the International System of Units.

Time history function

In the Tasks section, PackCollection tasks are defined to collect time history information from fields. Either the entire field or specified named sets of indices in the field can be collected. In this example, phaseVolumeFractionCollection is specified to output the time history of phase saturations fieldName="phaseVolumeFraction" across the computational domain.

```
  <Tasks>
    <PackCollection
      name="phaseVolumeFractionCollection"
      objectPath="ElementRegions/region/cellBlock"
      fieldName="phaseVolumeFraction"/>
  </Tasks>
```

This task is triggered using the Event manager with a PeriodicEvent defined for the recurring tasks. GEOS writes one file named after the string defined in the filename keyword, formatted as a HDF5 file (saturationHistory.hdf5). The TimeHistory file contains the collected time history information from the specified time history collector. This file includes datasets for the simulation time, element center, and the time history information for both phases. A Python script is prepared to read and plot any specified subset of the time history data for verification and visualization.

Initial and boundary conditions

The next step is to specify fields, including:

- The initial value (the pore pressure and phase saturations have to be initialized)
- The boundary conditions (fluid injection rates and controls on the fluid outlet have to be set)

In this example, the domain is initially saturated with brine in a uniform pressure field. The component attribute of the **FieldSpecification** XML block must use the order in which the phaseNames have been defined in the **DeadOilFluid** XML block. In other words, component=0 is used to initialize the gas global component fraction and component=1 is used to initialize the water global component fraction, because we previously set phaseNames="{gas, water}" in the **DeadOilFluid** XML block.

A mass injection rate SourceFlux (scale="-0.00007") of pure CO2 (component="0") is applied at the fluid inlet, named source. The value given for scale is $Q_T \rho_g$. Pressure and composition controls at the fluid outlet (sink) are also specified. The setNames="{ source } and setNames="{ sink }" are defined using the **Box** XML tags of the **Geometry** section.

These boundary conditions are set up through the FieldSpecifications section.

```
<FieldSpecifications>
  <FieldSpecification
    name="initialPressure"
    initialCondition="1"
    setNames="{ all }"
    objectPath="ElementRegions"
    fieldName="pressure"
    scale="1e7"/>

  <FieldSpecification
    name="initialComposition_gas"
    initialCondition="1"
    setNames="{ all }"
    objectPath="ElementRegions"
    fieldName="globalCompFraction"
    component="0"
    scale="0.001"/>

  <FieldSpecification
    name="initialComposition_water"
    initialCondition="1"
    setNames="{ all }"
    objectPath="ElementRegions"
    fieldName="globalCompFraction"
    component="1"
    scale="0.999"/>

  <SourceFlux
```

(continues on next page)

(continued from previous page)

```
      name="sourceTerm"
      objectPath="ElementRegions"
      scale="-0.00007"
      component="0"
      setNames="{ source }"/>

    <FieldSpecification
      name="sinkTermPressure"
      objectPath="faceManager"
      fieldName="pressure"
      scale="1e7"
      setNames="{ sink }"/>

    <FieldSpecification
      name="sinkTermTemperature"
      objectPath="faceManager"
      fieldName="temperature"
      scale="300"
      setNames="{ sink }"/>

    <FieldSpecification
      name="sinkTermComposition_gas"
      setNames="{ sink }"
      objectPath="faceManager"
      fieldName="globalCompFraction"
      component="0"
      scale="0.001"/>

    <FieldSpecification
      name="sinkTermComposition_water"
      setNames="{ sink }"
      objectPath="faceManager"
      fieldName="globalCompFraction"
      component="1"
      scale="0.999"/>
  </FieldSpecifications>
```

The parameters used in the simulation are summarized in the following table, and specified in the `Constitutive` and `FieldSpecifications` sections.

Symbol	Parameter	Unit	Value
${k_{rg}}^0$	Max Relative Permeability of Gas	[-]	1.0
${k_{rw}}^0$	Max Relative Permeability of Water	[-]	1.0
n_g	Corey Exponent of Gas	[-]	3.5
n_w	Corey Exponent of Water	[-]	3.5
S_{gr}	Residual Gas Saturation	[-]	0.0
S_{wr}	Residual Water Saturation	[-]	0.0
ϕ	Porosity	[-]	0.2
κ	Matrix Permeability	[m^2]	9.0*10^{-13}
μ_g	Gas Viscosity	[Pa s]	2.3*10^{-5}
μ_w	Water Viscosity	[Pa s]	5.5*10^{-4}
Q_T	Total Flow Rate	[m^3/s]	2.5*10^{-7}
D_L	Domain Length	[m]	0.1
D_W	Domain Width	[m]	1.0
D_T	Domain Thickness	[m]	0.002

Inspecting results

We request VTK-format output files and use Paraview to visualize the results. The following figure shows the distribution of phase saturations and pore pressures in the computational domain at $t = 70s$.

Two following dimensionless terms are defined when comparing the numerical solution with the analytical solutions:

$$t^{\star} = \frac{Q_T t}{A D_L \phi}$$

$$x_d = \frac{x_{S_g}}{D_L}$$

The figure below compares the results from GEOS (dashed curves) and the corresponding analytical solution (solid curves) for the change of gas saturation (S_g) and water saturation (S_w) along the flow direction.

GEOS reliably captures the immiscible transport of two phase flow (CO2 and water). GEOS matches the analytical solutions in the formation and the progress of abrupt fronts in the saturation profiles.

To go further

Feedback on this example

For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

CO2 Plume Evolution and Leakage Through an Abandoned Well

Context

We consider a benchmark problem used in (Class et al., 2009) to compare a number of numerical models applied to CO2 storage in geological formations. Using a simplified isothermal and immiscible two-phase setup, this test case addresses the simulation of the advective spreading of CO2 injected into an aquifer and the leakage of CO2 from the aquifer through an abandoned, leaky well.

Our goal is to review the different sections of the XML file reproducing the benchmark configuration and to demonstrate that the GEOS results (i.e., arrival time of the CO2 plume at the leaky well and leakage rate through the abandoned well) are in agreement with the reference results published in (Class et al., 2009).

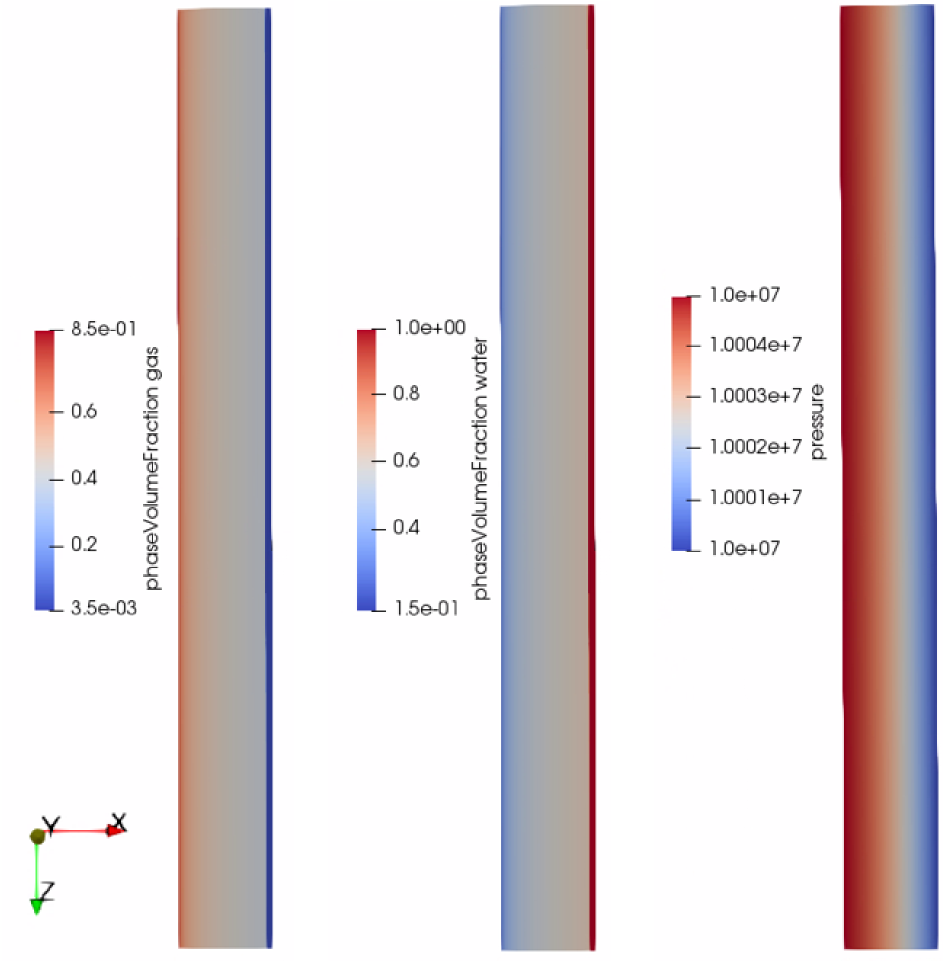
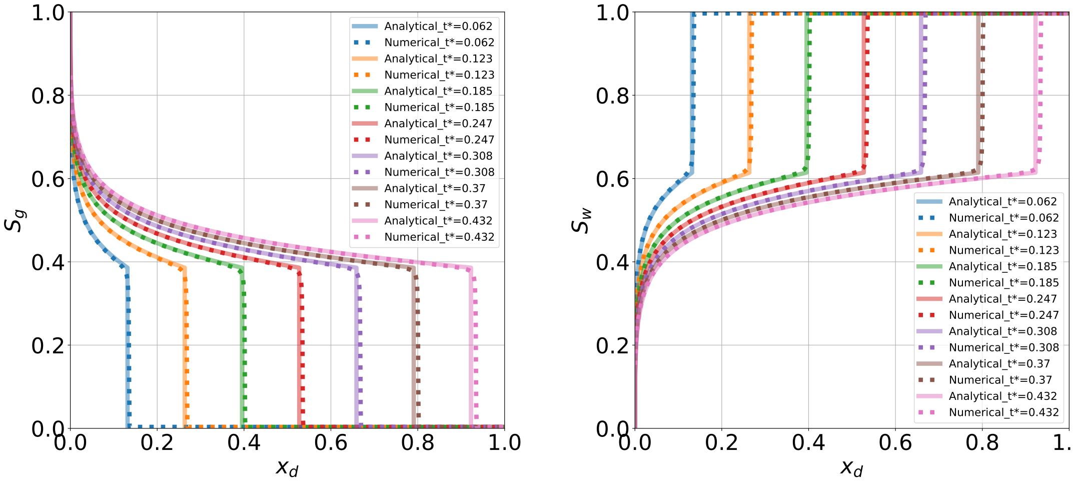


Fig. 1.4: Simulation results of phase saturations and pore pressure



The GEOS results obtained for the non-isothermal version of this test case (referred to as Problem 1.2 in (Class et al., 2009)) are presented in a separate documentation page.

Input file

This benchmark test is based on the XML file located below:

```
inputFiles/compositionalMultiphaseFlow/benchmarks/isothermalLeakyWell/
↪isothermalLeakyWell_benchmark.xml
```

Problem description

The following text is adapted from the detailed description of the benchmark test case presented in (Ebigbo, Class, Helmig, 2007) and (Class et al., 2009).

The leakage scenario considered here involves one CO2 injection well, one leaky well, two aquifers and an aquitard. The setup is illustrated in the figure below. The leaky well connects the two aquifers. CO2 is injected into in the lower aquifer, comes in contact with the leaky well and rises to the higher aquifer. The advective flow (including the buoyancy effects) of CO2 in the initially brine-saturated aquifers and through the leaky well is the most important process in this problem.

The model domain is located 2840 to 3000 m below the surface and has the following dimensions: 1000 x 1000 x 160 m. The distance between the injection and the leaky well is 100 m, and the injection rate of CO2 into the lower aquifer is constant (equal to 8.87 kg/s). The wells are described as cylinders with a 0.15 m radius and are treated as a porous medium with a higher permeability than the aquifers (i.e., this problem does not require a well model).

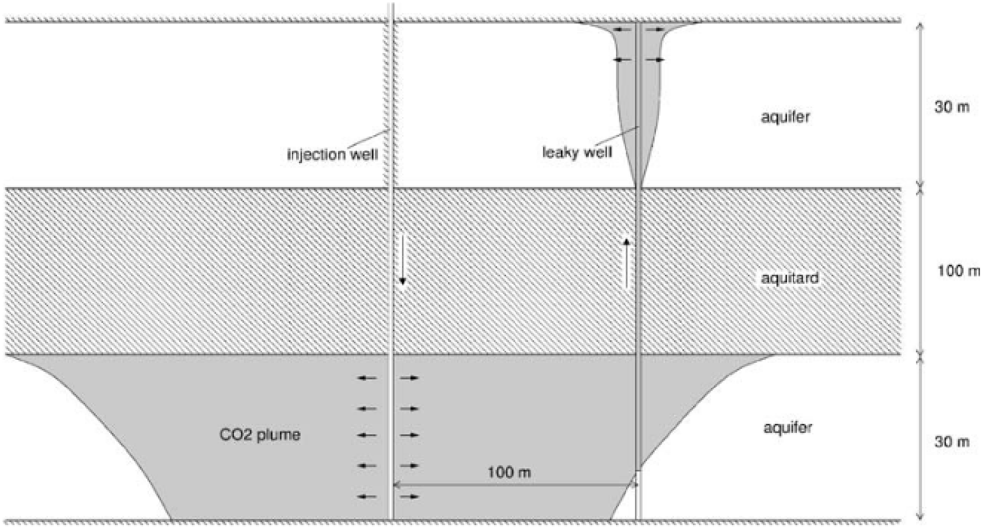


Fig. 1.5: Leakage scenario (image taken from (Ebigbo, Class, Helmig, 2007)).

The authors have used the following simplifying assumptions:

- The formation is isotropic.
- All processes are isothermal.
- CO2 and brine are two separate and immiscible phases. Mutual dissolution is neglected.
- Fluid properties such as density and viscosity are constant.

- The pressure conditions at the lateral boundaries are constant over time, and equal to the initial hydrostatic condition.
- Relative permeabilities are linear functions of saturation.
- Capillary pressure is negligible.

Mesh and element regions

The structured mesh is generated using the internal mesh generator as parameterized in the **InternalMesh** block of the XML file. The mesh contains 112 x 101 x 60 hexahedral elements (`C3D8` element type) in the x, y, and z directions respectively.

The attributes `nx`, `ny`, `nz`, and `cellBlockNames` are used to define 5 x 3 x 3 = 45 cell blocks. Note that the cell block names listed in `cellBlockNames` are mapped to their corresponding cell block using an k-j-i logic in which the k index is the fastest index, and the i index is the slowest index.

```
<Mesh>
  <InternalMesh
    name="mesh"
    elementTypes="{ C3D8 }"
    xCoords="{ -500, -0.1329, 0.1329, 99.8671, 100.1329, 500 }"
    yCoords="{ -500, -0.1329, 0.1329, 500 }"
    zCoords="{ -3000, -2970, -2870, -2840 }"
    nx="{ 50, 1, 20, 1, 40 }"
    ny="{ 50, 1, 50 }"
    nz="{ 20, 30, 10 }"
    cellBlockNames="{ aquiferBottom00, aquitard00, aquiferTop00,
                        aquiferBottom01, aquitard01, aquiferTop01,
                      aquiferBottom02, aquitard02, aquiferTop02,
                       aquiferBottom10, aquitard10, aquiferTop10,
                        aquiferBottom11, aquitard11, aquiferTop11,
                      aquiferBottom12, aquitard12, aquiferTop12,
                      aquiferBottom20, aquitard20, aquiferTop20,
                        aquiferBottom21, aquitard21, aquiferTop21,
                      aquiferBottom22, aquitard22, aquiferTop22,
                      aquiferBottom30, aquitard30, aquiferTop30,
                      aquiferBottom31, aquitard31, aquiferTop31,
                      aquiferBottom32, aquitard32, aquiferTop32,
                      aquiferBottom40, aquitard40, aquiferTop40,
                      aquiferBottom41, aquitard41, aquiferTop41,
                      aquiferBottom42, aquitard42, aquiferTop42 }"/>
</Mesh>
```

Note: In the GEOS input file, the two wells are defined as columns of hexahedral elements of individual size 0.2658 x 0.2658 x 1 m. The coefficient 0.2858 is chosen to ensure that the cross-section of the wells is equal to the one defined in the benchmark, in which wells are defined as cylinders with a radius of 0.15 m.

The cell block names are used in the **ElementRegions** block to define element regions in the aquifers and in the wells. In the benchmark definition, the wells are treated as a porous medium with a higher permeability and therefore the rock models are not the same in the aquifers (`rock`) and in the wells (`rockWell`). These names are defined in the **Constitutive** block.

```
<ElementRegions>
  <CellElementRegion
    name="aquiferBottom"
    cellBlocks="{ aquiferBottom00, aquiferBottom01, aquiferBottom02,
                  aquiferBottom10,                  aquiferBottom12,
                  aquiferBottom20, aquiferBottom21, aquiferBottom22,
                  aquiferBottom30,                  aquiferBottom32,
                  aquiferBottom40, aquiferBottom41, aquiferBottom42 }"
    materialList="{ fluid, rock, relperm }"/>
  <CellElementRegion
    name="aquiferTop"
    cellBlocks="{ aquiferTop00, aquiferTop01, aquiferTop02,
                  aquiferTop10,               aquiferTop12,
                  aquiferTop20, aquiferTop21, aquiferTop22,
                  aquiferTop30,               aquiferTop32,
                  aquiferTop40, aquiferTop41, aquiferTop42 }"
    materialList="{ fluid, rock, relperm }"/>
  <CellElementRegion
    name="injectionWell"
    cellBlocks="{ aquiferBottom31 }"
    materialList="{ fluid, rockWell, relperm }"/>
  <CellElementRegion
    name="leakyWell"
    cellBlocks="{ aquiferTop11, aquitard11, aquiferBottom11 }"
    materialList="{ fluid, rockWell, relperm }"/>
  <CellElementRegion
    name="barrier"
    cellBlocks="{ aquitard00, aquitard01, aquitard02,
                  aquitard10,             aquitard12,
                  aquitard20, aquitard21, aquitard22,
                  aquitard30, aquitard31, aquitard32,
                  aquitard40, aquitard41, aquitard42,
                  aquiferTop31 }"
    materialList="{ }"/>
</ElementRegions>
```

Defining these element regions allows the flow solver to be applied only to the top and bottom aquifers (and wells), since we follow the approach of (Class et al., 2009) and do not simulate flow in the aquitard, as we will see in the next section.

Note: Since the two aquifer regions share the same material properties, they could have been defined as a single `CellElementRegion` for a more concise XML file producing the same results. The same is true for the two well regions.

Flow solver

The isothermal immiscible simulation is performed by the GEOS general-purpose multiphase flow solver based on a TPFA discretization defined in the XML block **CompositionalMultiphaseFVM**:

```
<Solvers>
  <CompositionalMultiphaseFVM
    name="compflow"
    logLevel="1"
    discretization="fluidTPFA"
    temperature="313.15"
    initialDt="0.001"
    useMass="1"
    targetRegions="{ aquiferTop, aquiferBottom, injectionWell, leakyWell }">
    <NonlinearSolverParameters
      newtonTol="1.0e-3"
      newtonMaxIter="100"
      maxTimeStepCuts="5"
      lineSearchAction="Attempt"/>
    <LinearSolverParameters
      solverType="fgmres"
      preconditionerType="mgr"
      krylovTol="1e-4"/>
  </CompositionalMultiphaseFVM>
</Solvers>
```

We use the `targetRegions` attribute to define the regions where the flow solver is applied. Here, we follow the approach described in (Class et al., 2009) and do not simulate flow in the aquitard, considered as impermeable to flow. We only simulate flow in the two aquifers (`aquiferTop` and `aquiferBottom`), in the bottom part of the injection well (`injectionWell`), and in the leaky well connecting the two aquifers (`leakyWell`).

Constitutive laws

This benchmark test involves an immiscible, incompressible, two-phase model. The best approach to represent this fluid behavior in GEOS is to use the **DeadOilFluid** model, although this model was initially developed for simple isothermal oil-water or oil-gas systems (note that this is also the approach used for the Eclipse simulator, as documented in (Class et al., 2009)).

```
  <DeadOilFluid
    name="fluid"
    phaseNames="{ oil, water }"
    surfaceDensities="{ 479.0, 1045.0 }"
    componentMolarWeight="{ 114e-3, 18e-3 }"
    hydrocarbonFormationVolFactorTableNames="{ B_o_table }"
    hydrocarbonViscosityTableNames="{ visc_o_table }"
    waterReferencePressure="1e7"
    waterFormationVolumeFactor="1.0"
    waterCompressibility="1e-15"
    waterViscosity="2.535e-4"/>
```

The fluid properties (constant densities and constant viscosities) are those given in the article documenting the benchmark. To define an incompressible fluid, we set all the formation volume factors to 1.

The rock model defines an incompressible porous medium with a porosity equal to 0.15. The relative permeability model is linear. Capillary pressure is neglected.

Initial and boundary conditions

The domain is initially saturated with brine with a hydrostatic pressure field. This is specified using the **HydrostaticEquilibrium** XML tag in the **FieldSpecifications** block. The datum pressure and elevation used below are defined in (Class et al., 2009)).

```
<HydrostaticEquilibrium
  name="equil"
  objectPath="ElementRegions"
  datumElevation="-3000"
  datumPressure="3.086e7"
  initialPhaseName="water"
  componentNames="{ oil, water }"
  componentFractionVsElevationTableNames="{ initOilCompFracTable,
                                            initWaterCompFracTable }"
  temperatureVsElevationTableName="initTempTable"/>
```

In the **Functions** block, the **TableFunction** s named `initOilCompFracTable` and `initWaterCompFracTable` define the brine-saturated initial state, while the **TableFunction** named `initTempTable` defines the homogeneous temperature field (temperature is not used in this benchmark).

Since the fluid densities are constant for this simple incompressible fluid model, the same result could have been achieved using a simpler table in a **FieldSpecification** tag, as explained in *Hydrostatic Equilibrium Initial Condition*. To impose the Dirichlet boundary conditions on the four sides of the aquifers, we use this simple table-based approach as shown below:

```
<FieldSpecification
  name="bcPressureAquiferBottom"
  objectPath="ElementRegions/aquiferBottom"
  setNames="{ east, west, south, north }"
  fieldName="pressure"
  functionName="pressureFunction"
  scale="1"/>
<FieldSpecification
  name="bcPressureAquiferTop"
  objectPath="ElementRegions/aquiferTop"
  setNames="{ east, west, south, north }"
  fieldName="pressure"
  functionName="pressureFunction"
  scale="1"/>
<FieldSpecification
  name="bcCompositionOilAquiferBottom"
  setNames="{ east, west, south, north }"
  objectPath="ElementRegions/aquiferBottom"
  fieldName="globalCompFraction"
  component="0"
  scale="0.000001"/>
<FieldSpecification
  name="bcCompositionOilAquiferTop"
  setNames="{ east, west, south, north }"
```

(continues on next page)

(continued from previous page)

```
      objectPath="ElementRegions/aquiferTop"
      fieldName="globalCompFraction"
      component="0"
      scale="0.000001"/>
    <FieldSpecification
      name="bcCompositionWaterAquiferBottom"
      setNames="{ east, west, south, north }"
      objectPath="ElementRegions/aquiferBottom"
      fieldName="globalCompFraction"
      component="1"
      scale="0.999999"/>
    <FieldSpecification
      name="bcCompositionWaterAquiferTop"
      setNames="{ east, west, south, north }"
      objectPath="ElementRegions/aquiferTop"
      fieldName="globalCompFraction"
      component="1"
      scale="0.999999"/>
```

where the `setNames = "{ east, west, south, north }"` are defined using the **Box** XML tags of the **Geometry** section, and where the tables are defined as **TableFunction** in the **Functions** section.

To reproduce the behavior of a rate-controlled well, we use the **SourceFlux** tag on the `source` set (located in the `injectionWell` cell element region), with the injection rate specified in the benchmark description (8.87 kg/s):

```
    <SourceFlux
      name="sourceTerm"
      objectPath="ElementRegions/injectionWell"
      component="0"
      scale="-8.87"
      setNames="{ source }"/>
```

Note: If the `setNames` attribute of **SourceFlux** contains multiple cells (which is the case here), then the amount injected in each cell is equal to the total injection rate (specified with `scale`) divided by the number of cells.

Inspecting results

We request VTK-format output files and use Paraview to visualize the results. The following figure shows the distribution of CO2 saturation and pressure along the slice defined by x = 0 at t = 200 days.

To validate the GEOS results, we consider the metrics used in (Class et al., 2009).

First, we consider the arrival time of the CO2 plume at the leaky well. As in (Class et al., 2009), we use the leakage rate threshold of 0.005% to detect the arrival time. Although the arrival time is highly dependent on the degree of spatial refinement in the vicinity of the wells (not documented in (Class et al., 2009)), the next table shows that the GEOS arrival time at the leaky well (9.6 days) is in agreement with the values published in (Class et al., 2009).

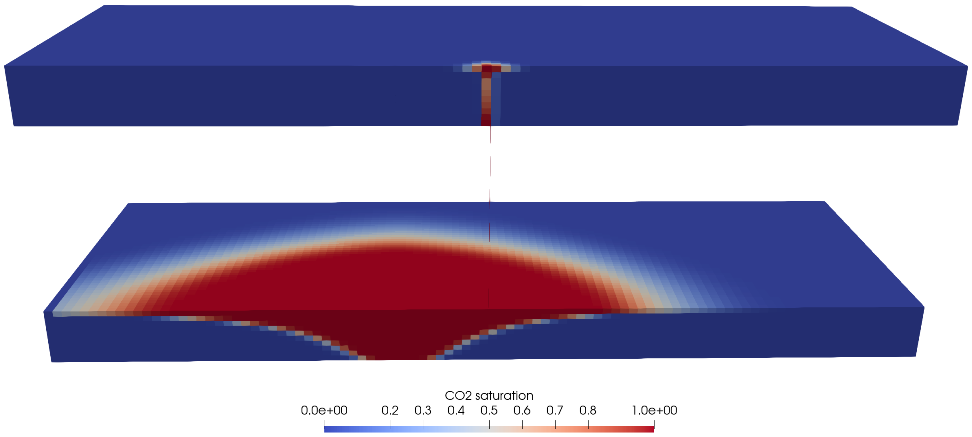


Fig. 1.6: CO2 saturation after 200 days

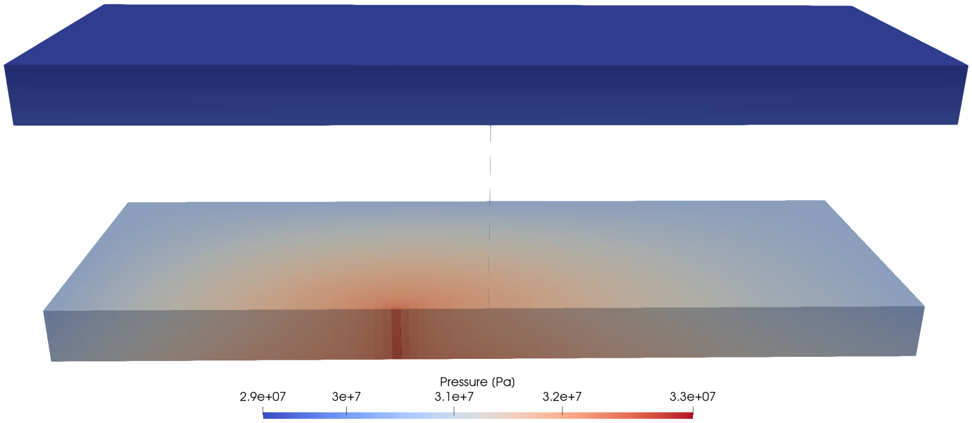


Fig. 1.7: Pressure after 200 days

Code	Arrival time [day]
GEOSX	9.6
COORES	8
DuMux	6
ECLIPSE	8
FEHM	4
IPARS-CO2	10
MUFTE	8
RockFlow	19
ELSA	14
TOUGH2/ECO2N	4
TOUGH2/ECO2N (2)	10
TOUGH2 (3)	9
VESA	7

Note: The time-stepping strategy used by the codes considered in (Class et al., 2009), as well as the meshes that have been used, are not documented in the benchmark description. Therefore, even on this simple test case, we cannot expect to obtain an exact match with the published results.

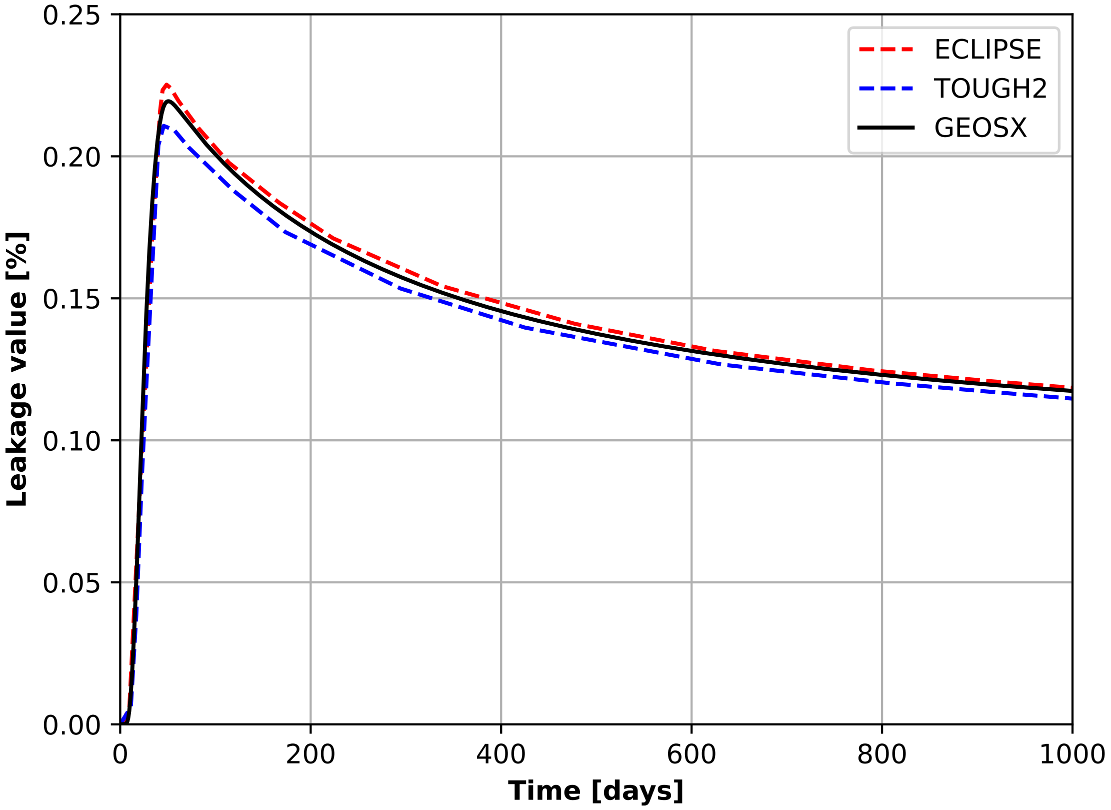
Next, we measure the CO2 leakage rate through the leaky well, defined by the authors as the CO2 mass flow at midway between top and bottom aquifers divided by the injection rate (8.87 kg/s), in percent. The GEOS leakage rate is shown in the figure below:

The leakage rates computed by the codes considered in (Class et al., 2009) are shown in the figure below.

The comparison between the previous two figures shows that GEOS can successfully reproduce the trend in leakage rate observed in the published benchmark results. To further validate the GEOS results, we reproduce below Table 8 of (Class et al., 2009) to compare the maximum leakage rate, the time at which this maximum leakage rate is attained, and the leakage rate at 1000 days.

Code	Max leakage [%]	Time at max leakage [day]	Leakage at 1000 days [%]
GEOSX	0.219	50.6	0.1172
COORES	0.219	50	0.146
DuMux	0.220	61	0.128
ECLIPSE	0.225	48	0.118
FEHM	0.216	53	0.119
IPARS-CO2	0.242	80	0.120
MUFTE	0.222	58	0.126
RockFlow	0.220	74	0.132
ELSA	0.231	63	0.109
TOUGH2/ECO2N	0.226	93	0.110
TOUGH2/ECO2N (2)	0.212	46	0.115
TOUGH2 (3)	0.227	89	0.112
VESA	0.227	41	0.120

This table confirms the agreement between GEOS and the results of (Class et al., 2009). A particularly good match is obtained with Eclipse, FEHM, and TOUGH2/ECO2N (2).


0.25
0.20
0.15
0.10
0.05
0.00
Leakage value [%]
0
200
400
600
800
1000
Time [days]
ECLIPSE
TOUGH2
GEOSX

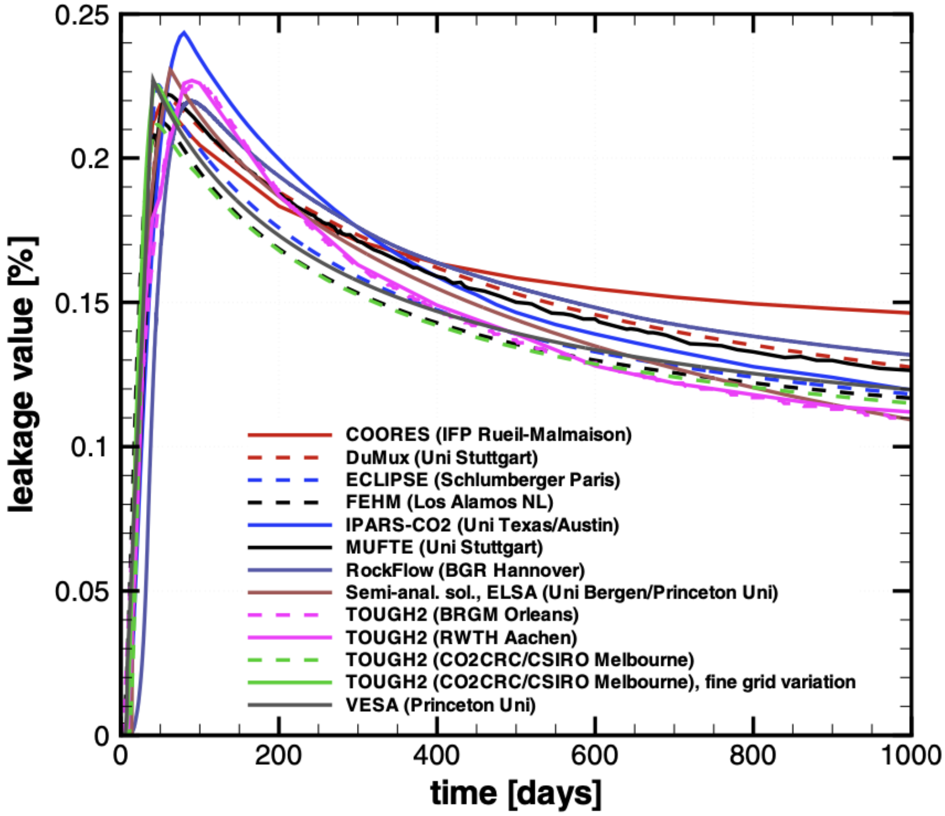


Fig. 1.8: Leakage rates [%] obtained with the simulators considered in (Class et al., 2009).

To go further

The more complex, non-isothermal version of this test is in *Non-isothermal CO2 Plume Evolution and Leakage Through an Abandoned Well*.

Feedback on this example

For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

Non-isothermal CO2 Plume Evolution and Leakage Through an Abandoned Well

Context

This validation case is a more complex version of the benchmark problem presented in *CO2 Plume Evolution and Leakage Through an Abandoned Well*. While the latter is based on simple isothermal and immiscible fluid properties, the present validation case relies on a more realistic fluid behavior accounting for thermal effects and mass exchange between phases. This non-isothermal benchmark test has been used in (Class et al., 2009) to compare different implementations of CO2-brine fluid properties in the context of CO2 injection and storage in saline aquifers.

Our goal is to review the sections of the XML file that are used to parameterize the CO2-brine fluid behavior, and to demonstrate that GEOS produces similar results as those presented in (Class et al., 2009).

Input file

This benchmark test is based on the XML file located below:

```
inputFiles/compositionalMultiphaseFlow/benchmarks/thermalLeakyWell/thermalLeakyWell_
↪benchmark.xml
```

Problem description

Some of the text below is adapted from (Ebigbo, Class, Helmig, 2007).

The benchmark scenario remains the same as in *CO2 Plume Evolution and Leakage Through an Abandoned Well*. CO2 is injected into an aquifer, spreads within the aquifer, and, upon reaching a leaky well, rises up to a shallower aquifer. The model domain still has the dimensions: 1000 x 1000 x 160 m, but it is now assumed to be shallower, between 640 m and 800 m of depth.

The figure below shows the pressure and temperature in the formation at the mentioned depths (assuming a geothermal gradient of 0.03 K/m). The conditions in the aquifer at the considered depths range from supercritical to liquid to gaseous. The figure also shows the CO2 density at the conditions of the formation. There is a large change in density at a certain depth. This depth corresponds to the point where the line depicting the formation conditions crosses the CO2 saturation vapor curve, that is, the boundary between liquid and gaseous CO2. Other fluid properties such as viscosity also change abruptly at that depth.

Therefore, as explained later, we use a more sophisticated fluid model in which the CO2 and brine fluid properties are now a function of the aquifer conditions, such as pressure, temperature, and salinity. Specifically:

- The CO2 component is present in the CO2-rich phase but can also dissolve in the brine phase. The amount of dissolved CO2 depends on pressure, temperature, and salinity. For now, in GEOS, the water component cannot be present in the CO2-rich phase.
- Densities and viscosities depend nonlinearly on pressure, temperature, and salinity.
- The hydrostatic initial condition accounts for the geothermal gradient of 0.03 K/m specified in the benchmark description.

We plan to use two types of physical models in this benchmark:

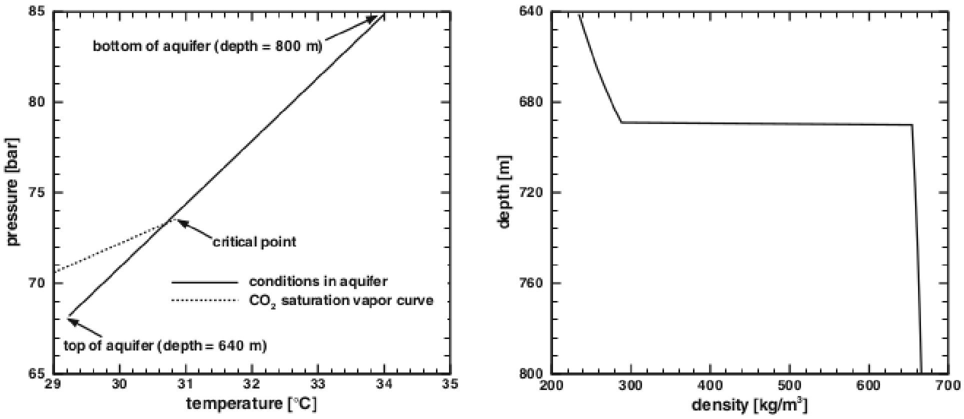


Fig. 1.9: Aquifer conditions (image taken from (Ebigbo, Class, Helmig, 2007)).

- A model simulating flow and mass transfer, but not heat transfer (i.e., no energy balance is used). The geothermal gradient is constant in time, and is taken into account in the calculation of temperature-dependent properties.
- A fully thermal model simulating flow as well as mass and heat transfer. The results obtained with this more complex model are not available yet and will be added to this page later.

Mesh and element regions

As illustrated by the ECLIPSE results in (Class et al., 2009), the leakage rate exhibits a high dependence on the degree of spatial refinement (particularly between the two wells in our observations). Therefore, we consider two meshes in this test case:

- A "coarse" mesh with 206070 cells, whose spatial resolution is similar to that used by most codes based on the information provided by Table 13 of (Class et al., 2009).
- A "fine" mesh with 339390 cells, whose spatial resolution is finer between the two wells.

These structured meshes are defined as in *CO2 Plume Evolution and Leakage Through an Abandoned Well*, as shown next for the "fine" mesh.

```
<Mesh>
  <InternalMesh
    name="mesh"
    elementTypes="{ C3D8 }"
    xCoords="{ -500, -0.1329, 0.1329, 99.8671, 100.1329, 500 }"
    yCoords="{ -500, -0.1329, 0.1329, 500 }"
    zCoords="{ -800, -770, -670, -640 }"
    nx="{ 50, 1, 20, 1, 40 }"
    ny="{ 50, 1, 50 }"
    nz="{ 20, 30, 10 }"
    cellBlockNames="{ aquiferBottom00, aquitard00, aquiferTop00,
                        aquiferBottom01, aquitard01, aquiferTop01,
                      aquiferBottom02, aquitard02, aquiferTop02,
                       aquiferBottom10, aquitard10, aquiferTop10,
                        aquiferBottom11, aquitard11, aquiferTop11,
                      aquiferBottom12, aquitard12, aquiferTop12,
                      aquiferBottom20, aquitard20, aquiferTop20,
```

(continues on next page)

(continued from previous page)

```
                        aquiferBottom21, aquitard21, aquiferTop21,
                      aquiferBottom22, aquitard22, aquiferTop22,
                      aquiferBottom30, aquitard30, aquiferTop30,
                      aquiferBottom31, aquitard31, aquiferTop31,
                      aquiferBottom32, aquitard32, aquiferTop32,
                      aquiferBottom40, aquitard40, aquiferTop40,
                      aquiferBottom41, aquitard41, aquiferTop41,
                      aquiferBottom42, aquitard42, aquiferTop42 }"/>
</Mesh>
```

As in the previous benchmark, we define four element regions whose material list now includes the name of the capillary pressure constitutive model (cappres). We refer the reader to *CO2 Plume Evolution and Leakage Through an Abandoned Well* for an example of this procedure.

Flow solver

Although the fluid behavior is significantly different from that of the previous benchmark, we still use the GEOS general-purpose multiphase flow solver defined in the XML block **CompositionalMultiphaseFVM**:

```
<Solvers>
  <CompositionalMultiphaseFVM
    name="compflow"
    logLevel="1"
    discretization="fluidTPFA"
    temperature="307.15"
    initialDt="1"
    useMass="1"
    targetRegions="{ aquiferTop, aquiferBottom, injectionWell, leakyWell }">
    <NonlinearSolverParameters
      newtonTol="1.0e-3"
      newtonMaxIter="20"
      timeStepIncreaseIterLimit="0.5"
      timeStepDecreaseIterLimit="0.9"
      maxTimeStepCuts="5"
      lineSearchAction="Attempt"/>
    <LinearSolverParameters
      solverType="fgmres"
      preconditionerType="mgr"
      krylovTol="1e-4"/>
  </CompositionalMultiphaseFVM>
</Solvers>
```

Note: The attribute temperature listed above is mandatory, but are overridden by GEOS to impose a non-uniform geothermal gradient along the z-axis, as we will see later.

Constitutive models

The Brooks-Corey relative permeabilities and capillary pressure are described using tables constructed from the parameters values provided in the benchmark description, with a wetting-phase saturation range between 0.2 and 0.95, an entry pressure of 10000 Pa, and a Brooks-Corey parameter of 2. We refer the reader to the files used in the **TableFunction** listed below for the exact values that we have used:

```
    <TableRelativePermeability
      name="relperm"
      phaseNames="{ gas, water }"
      wettingNonWettingRelPermTableNames="{ waterRelativePermeabilityTable,
                                            gasRelativePermeabilityTable }"/>
    <TableCapillaryPressure
      name="cappres"
      phaseNames="{ gas, water }"
      wettingNonWettingCapPressureTableName="waterCapillaryPressureTable"/>
```

The two-phase, two-component CO2-brine model implemented in GEOS is parameterized in the **CO2BrinePhillips** XML block:

```
    <CO2BrinePhillipsFluid
      name="fluid"
      phaseNames="{ gas, water }"
      componentNames="{ co2, water }"
      componentMolarWeight="{ 44e-3, 18e-3 }"
      phasePVTParaFiles="{ pvtgas.txt, pvtliquid.txt }"
      flashModelParaFile="co2flash.txt"/>
```

The components of this fluid model are described in detail in the *CO2-brine model* and are briefly summarized below. They are parameterized using three parameter files that must be written carefully to obtain the desired behavior, as explained next.

CO2 density and viscosity

These properties are obtained using the models proposed by Span and Wagner (1996) and Fenghour and Wakeham (1998) for density and viscosity, respectively. The density and viscosity values are internally tabulated by GEOS at the beginning of the simulation by solving the Helmholtz energy equation for each pair (p, T).

The tables size and spacing are specified in the file *pvtgas.txt*. Here, for both quantities, the values are tabulated between 6.6e6 Pa and 4e7 Pa, with a pressure spacing of 1e6 Pa, and between 302 K and 312 K, with a temperature increment of 5 K. These values have been chosen using the initial condition and an upper bound on the expected pressure increase during the simulation.

```
DensityFun SpanWagnerCO2Density 6.6e6 4e7 1e6 302.0 312.0 5
ViscosityFun FenghourCO2Viscosity 6.6e6 4e7 1e6 302.0 312.0 5
```

Note: If pressure or temperature go outside the values specified in this parameter file, constant extrapolation is used to obtain the density and viscosity values. Note that for now, no warning is issued by GEOS when this happens. We plan to add a warning message to document this behavior in the near future.

Brine density and viscosity

These properties depend on pressure, temperature, composition, and salinity via the models proposed by Phillips et al. (1981). The brine density is modified to account for the presence of dissolved CO2 using the method proposed by Garcia (2001). The values of (pure) brine density are also tabulated at a function of pressure and temperature, and we use the same range as for the CO2 properties to construct this table:

```
DensityFun PhillipsBrineDensity 6.6e6 4e7 1e6 302.0 312.0 5 1.901285269
ViscosityFun PhillipsBrineViscosity 1.901285269
```

Importantly, the last value on each line in the file *pvtliquid.txt* defines the salinity in the domain. In our model, salinity is constant in space and in time (i.e., unlike water and CO2, it is not tracked as a component in GEOS). In our model, salinity is specified as a molal concentration in mole of NaCl per kg of solvent (brine). The value used here (1000 x 10 / (58.44 x (100 - 10)) = 1.901285269 moles/kg) is chosen to match the value specified in the benchmark (weight% of 10%).

CO2 solubility in brine

As explained in *CO2-brine model*, we use the highly nonlinear model proposed by Duan and Sun (2004) to compute the CO2 solubility as a function of pressure, temperature, composition, and salinity. In *co2flash.txt*, we use the same parameters as above to construct the pressure-temperature tables of precomputed CO2 solubility in brine.

```
FlashModel CO2Solubility 6.6e6 4e7 1e6 302.0 312.0 5 1.901285269
```

Initial and boundary conditions

The domain is initially saturated with brine with a hydrostatic pressure field and a geothermal gradient of 0.03 K/m. This is specified using the **HydrostaticEquilibrium** XML tag in the **FieldSpecifications** block:

```
    <HydrostaticEquilibrium
      name="equil"
      objectPath="ElementRegions"
      datumElevation="-800"
      datumPressure="8.499e6"
      initialPhaseName="water"
      componentNames="{ co2, water }"
      componentFractionVsElevationTableNames="{ initCO2CompFracTable,
                                                initWaterCompFracTable }"
      temperatureVsElevationTableName="initTempTable"/>
```

Although this is the same block as in *CO2 Plume Evolution and Leakage Through an Abandoned Well*, GEOS is now enforcing the geothermal gradient specified in the **TableFunction** named `initTempTable`, and is also accounting for the nonlinear temperature dependence of brine density to equilibrate the pressure field.

We use the simple table-based approach shown below to impose the Dirichlet boundary conditions on the four sides of the domain.

```
    <FieldSpecification
      name="bcPressureAquiferBottom"
      objectPath="ElementRegions/aquiferBottom"
      setNames="{ east, west, south, north }"
      fieldName="pressure"
```

(continues on next page)

(continued from previous page)

```
      functionName="pressureFunction"
      scale="1"/>
    <FieldSpecification
      name="bcTemperatureAquiferBottom"
      objectPath="ElementRegions/aquiferBottom"
      setNames="{ east, west, south, north }"
      fieldName="temperature"
      functionName="initTempTable"
      scale="1"/>
    <FieldSpecification
      name="bcCompositionCO2AquiferBottom"
      setNames="{ east, west, south, north }"
      objectPath="ElementRegions/aquiferBottom"
      fieldName="globalCompFraction"
      component="0"
      scale="0.000001"/>
    <FieldSpecification
      name="bcCompositionWaterAquiferBottom"
      setNames="{ east, west, south, north }"
      objectPath="ElementRegions/aquiferBottom"
      fieldName="globalCompFraction"
      component="1"
      scale="0.999999"/>

    <FieldSpecification
      name="bcPressureAquiferTop"
      objectPath="ElementRegions/aquiferTop"
      setNames="{ east, west, south, north }"
      fieldName="pressure"
      functionName="pressureFunction"
      scale="1"/>
    <FieldSpecification
      name="bcTemperatureAquiferTop"
      objectPath="ElementRegions/aquiferTop"
      setNames="{ east, west, south, north }"
      fieldName="temperature"
      functionName="initTempTable"
      scale="1"/>
    <FieldSpecification
      name="bcCompositionCO2AquiferTop"
      setNames="{ east, west, south, north }"
      objectPath="ElementRegions/aquiferTop"
      fieldName="globalCompFraction"
      component="0"
      scale="0.000001"/>
    <FieldSpecification
      name="bcCompositionWaterAquiferTop"
      setNames="{ east, west, south, north }"
      objectPath="ElementRegions/aquiferTop"
      fieldName="globalCompFraction"
      component="1"
      scale="0.999999"/>
```

(continues on next page)

(continued from previous page)

```
    <FieldSpecification
      name="bcPressureLeakyWell"
      objectPath="ElementRegions/leakyWell"
      setNames="{ east, west, south, north }"
      fieldName="pressure"
      functionName="pressureFunction"
      scale="1"/>
    <FieldSpecification
      name="bcTemperatureLeakyWell"
      objectPath="ElementRegions/leakyWell"
      setNames="{ east, west, south, north }"
      fieldName="temperature"
      functionName="initTempTable"
      scale="1"/>
    <FieldSpecification
      name="bcCompositionCO2LeakyWell"
      setNames="{ east, west, south, north }"
      objectPath="ElementRegions/leakyWell"
      fieldName="globalCompFraction"
      component="0"
      scale="0.000001"/>
    <FieldSpecification
      name="bcCompositionWaterLeakyWell"
      setNames="{ east, west, south, north }"
      objectPath="ElementRegions/leakyWell"
      fieldName="globalCompFraction"
      component="1"
      scale="0.999999"/>

    <FieldSpecification
      name="bcPressureInjectionWell"
      objectPath="ElementRegions/injectionWell"
      setNames="{ east, west, south, north }"
      fieldName="pressure"
      functionName="pressureFunction"
      scale="1"/>
    <FieldSpecification
      name="bcTemperatureInjectionWell"
      objectPath="ElementRegions/injectionWell"
      setNames="{ east, west, south, north }"
      fieldName="temperature"
      functionName="initTempTable"
      scale="1"/>
    <FieldSpecification
      name="bcCompositionCO2InjectionWell"
      setNames="{ east, west, south, north }"
      objectPath="ElementRegions/injectionWell"
      fieldName="globalCompFraction"
      component="0"
      scale="0.000001"/>
    <FieldSpecification
```

(continues on next page)

(continued from previous page)

```
        name="bcCompositionWaterInjectionWell"
        setNames="{ east, west, south, north }"
        objectPath="ElementRegions/injectionWell"
        fieldName="globalCompFraction"
        component="1"
        scale="0.999999"/>
```

where the `setNames = "{ east, west, south, north }"` are defined using the **Box** XML tags of the **Geometry** section, and where the tables are defined as **TableFunction** in the **Functions** section.

Note: Due to the nonlinear dependence of brine density on temperature, this block does not exactly impose a Dirichlet pressure equal to the initial condition. Instead, here, we impose a linear pressure gradient along the z-axis, whose minimum and maximum values are the same as in the initial state. We could have imposed Dirichlet boundary conditions preserving the initial condition using as many points in *zlin.geos* as there are cells along the z-axis (instead of just two points).

The **SourceFlux** is the same as in the previous benchmark case (see *CO2 Plume Evolution and Leakage Through an Abandoned Well*).

Inspecting results

We request VTK-format output files and use Paraview to visualize the results. The following figures show the distribution of CO2 saturation and pressure along the slice defined by x = 0 at t = 1,000 days.

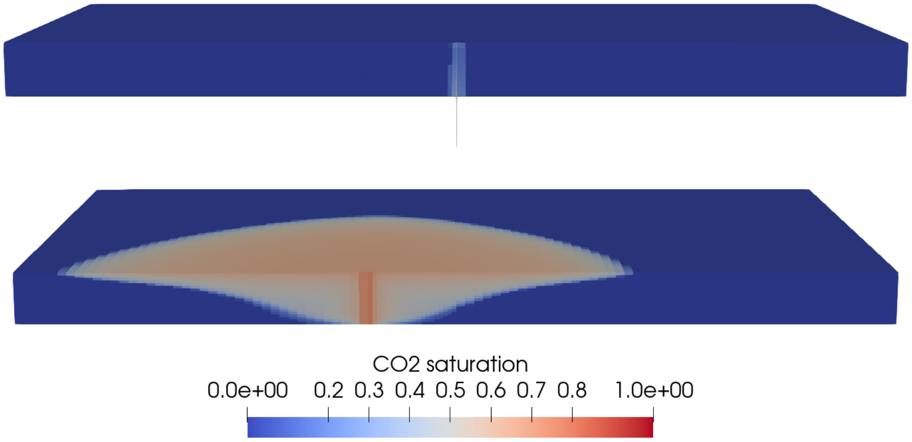


Fig. 1.10: CO2 saturation after 1,000 days

To validate the GEOS results, we consider the metrics used in (Class et al., 2009) as previously done in *CO2 Plume Evolution and Leakage Through an Abandoned Well*.

First, we consider the arrival time of the CO2 plume at the leaky well. As in (Class et al., 2009), we use the leakage rate threshold of 0.005% to detect the arrival time. In our numerical tests, the arrival time is highly dependent on the degree of spatial refinement in the vicinity of the wells and on the time step size, but these parameters are not documented in (Class et al., 2009). The next table reports the GEOS arrival time at the leaky well and compares it with the values published in (Class et al., 2009).

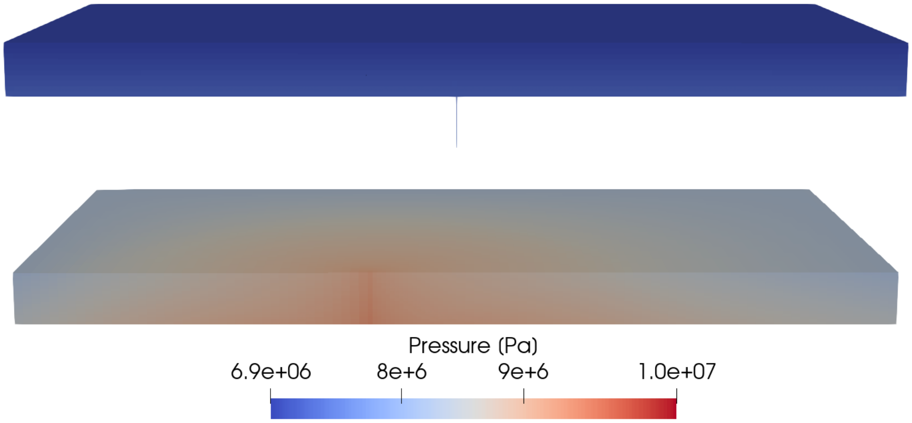


Fig. 1.11: Pressure after 1,000 days

Code	Arrival time [day]
GEOSX COARSE	36.8
GEOSX FINE	46.7
COORES	31
ECLIPSE HW	42
ECLIPSE SCHLUMBERGER COARSE	24
ECLIPSE SCHLUMBERGER FINE	34
RockFlow	30
TOUGH2	46

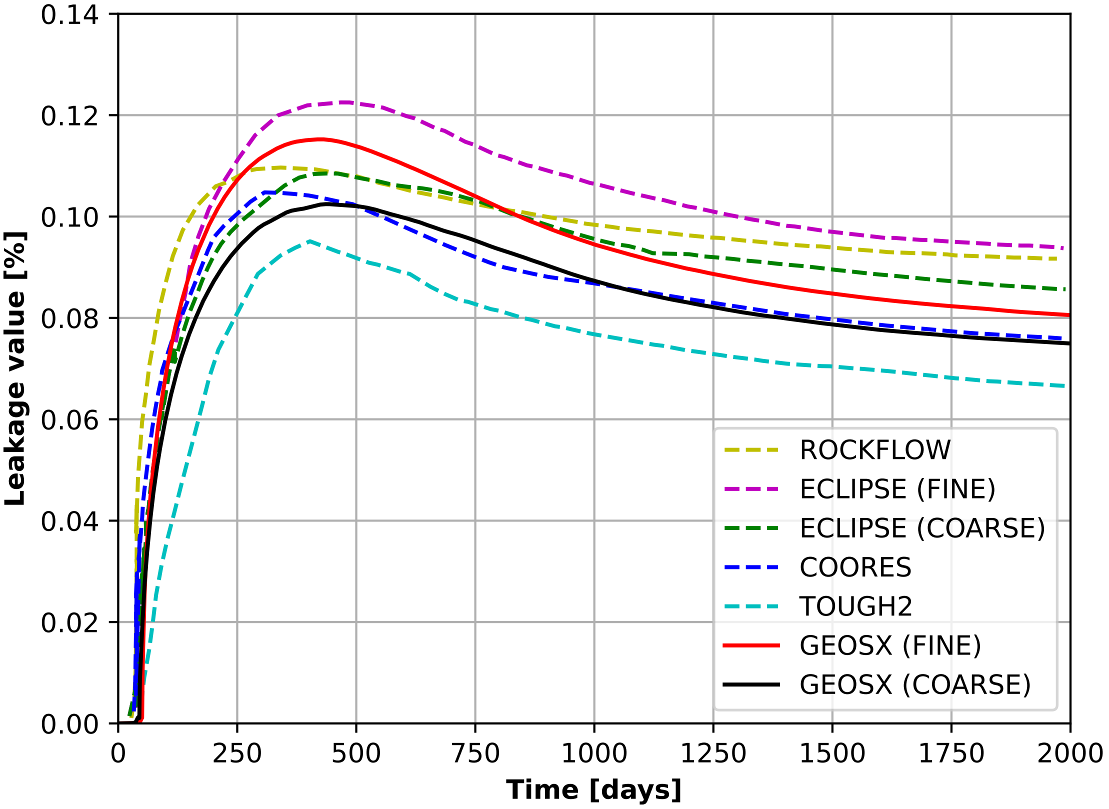
Note: In the table above, we only included the values obtained with the codes that do **not** solve an energy balance equation. The values obtained with the fully thermal codes (FEHM, MUFTE, and RTAFF2) are omitted for now.

Next, we measure the CO2 leakage rate through the leaky well, defined by the authors as the CO2 mass flow at midway between top and bottom aquifers divided by the injection rate (8.87 kg/s), in percent. The GEOS leakage rate is shown in the figure below:

We see that GEOS produces a reasonable match with the numerical codes considered in the study. Although it is not possible to exactly match the published results (due to the lack of information on the problem, such as mesh refinement and time step size), GEOS reproduces well the trend exhibited by the other codes.

For reference, we include below the original figure from (Class et al., 2009) containing all the results, including those obtained with the codes solving an energy equation.

To further validate the GEOS results, we reproduce below Table 9 of (Class et al., 2009) (only considering codes that do not solve an energy equation) to compare the maximum leakage rate, the time at which this maximum leakage rate is attained, and the leakage rate at 2000 days. We observe that the GEOS values are in the same range as those considered in the benchmark.


0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
Leakage value [%]
0
250
500
750
1000
1250
1500
1750
2000
Time [days]
ROCKFLOW
ECLIPSE (FINE)
ECLIPSE (COARSE)
COORES
TOUGH2
GEOSX (FINE)
GEOSX (COARSE)

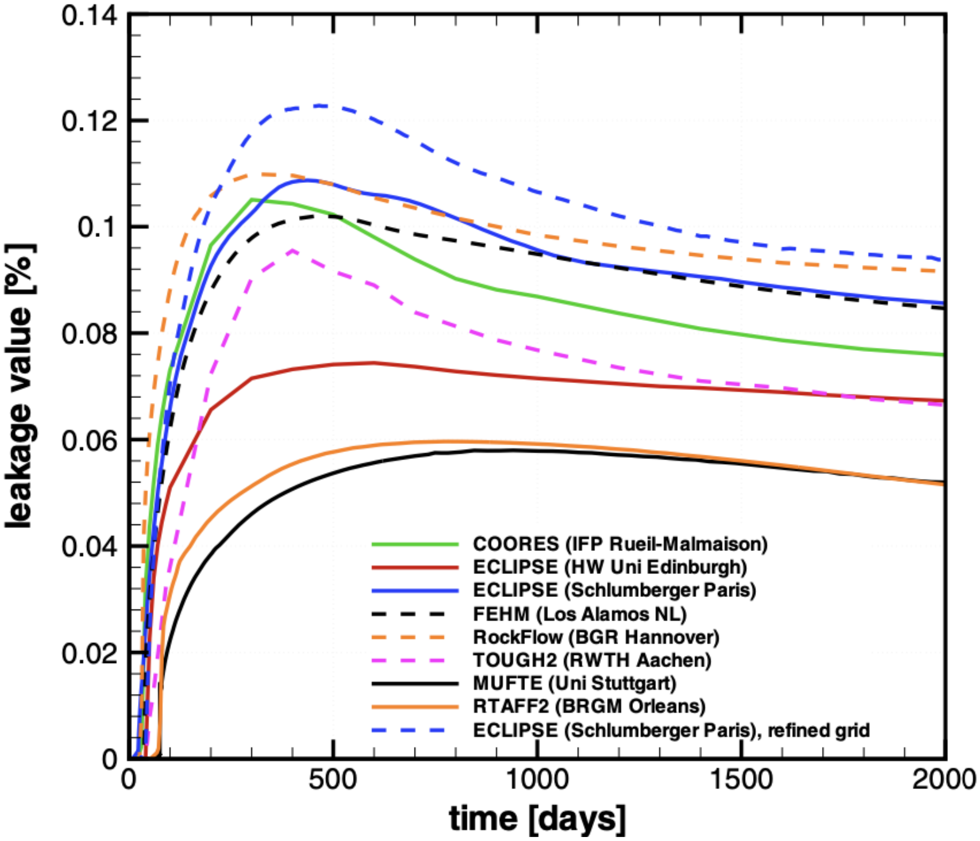


Fig. 1.12: Leakage rates [%] obtained with the simulators considered in (Class et al., 2009).

Code	Max leakage [%]	Time at max leakage [day]	Leakage at 2000 days [%]
GEOSX COARSE	0.102	438.5	0.075
GEOSX FINE	0.115	425.0	0.085
COORES	0.105	300	0.076
ECLIPSE HW	0.074	600	0.067
ECLIPSE SCHLUMBERGER COARSE	0.109	437	0.086
ECLIPSE SCHLUMBERGER FINE	0.123	465	0.094
RockFlow	0.11	279	0.09
TOUGH2	0.096	400	0.067

This table confirms the agreement between GEOS and the results of (Class et al., 2009).

To go further

Feedback on this example

For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

CO2 Plume Evolution With Hysteresis Effect on Relative Permeability

Context

We consider a benchmark problem used in (Class et al., 2009) to compare a number of numerical models applied to CO2 storage in geological formations. Using a simplified miscible two-phase setup, this test case illustrates the modeling of solubility trapping (with CO2 dissolution in brine) and residual trapping (with gas relative permeability hysteresis) in CO2-brine systems.

Our goal is to review the different sections of the XML file reproducing the benchmark configuration and to demonstrate that the GEOS results (i.e., mass of CO2 dissolved and mobile for both hysteretic and non-hysteretic configurations) are in agreement with the reference results published in (Class et al., 2009) Problems 3.1 and 3.2.

Input file

This benchmark test is based on the XML file located below:

```
../../../../../../../inputFiles/compositionalMultiphaseWell/benchmarks/Class09Pb3/class09_pb3_smoke_3d.xml
../../../../../../../inputFiles/compositionalMultiphaseWell/benchmarks/Class09Pb3/class09_pb3_drainageOnly_iterative_base.xml
```

Problem description

The following text is adapted from the detailed description of the benchmark test case presented in (Class et al., 2009).

The setup is illustrated in the figure below. The mesh can be found in GEOSXDATA and was provided for the benchmark. It discretizes the widely-used *Johansen* reservoir, which consists in a tilted reservoir with a main fault. The model domain has the following dimensions: 9600 x 8900 x [90-140] m. Both porosity and permeability are heterogeneous and given at vertices. A single CO2 injection well is located at (x,y) = (5440,3300) m with perforations only in the bottom 50 m of the reservoir. The injection takes place during the first 25 years of the 50-year simulation at the constant rate of 15 kg/s. A hydrostatic pressure gradient is imposed on the boundary faces as well as a constant geothermal gradient of 0.03 K/m.

The authors have used the following simplifying assumptions:

- The formation is isotropic.
- The temperature is constant in time.
- The pressure conditions at the lateral boundaries are constant over time, and equal to the initial hydrostatic condition.

Mesh and element regions

The proposed conforming discretization is fully hexahedral. A VTK filter `PointToCell` is used to map properties from vertices to cells, which by default builds a uniform average of values over the cell. The structured mesh is generated using some helpers python scripts from the formatted Point/Cells list provided. It is then imported using `meshImport`

```
<Mesh>
  <InternalMesh
    name="mesh1"
    elementTypes="{ C3D8 }"
    xCoords="{ 5240, 5640 }"
    yCoords="{ 3100, 3500 }"
    zCoords="{ -3000, -2950 }"
    nx="{ 5 }"
    ny="{ 5 }"
    nz="{ 5 }"
    cellBlockNames="{ 1_hexahedra }">
    <InternalWell
      name="wellInjector1"
      wellRegionName="wellRegion"
      wellControlsName="wellControls"
      logLevel="1"
      polylineNodeCoords="{ { 5440.0, 3300.0, -2950.0 },
                            { 5440.0, 3300.0, -3000.00 } }"
      polylineSegmentConn="{ { 0, 1 } }"
      radius="0.1"
      numElementsPerSegment="5">
      <Perforation
        name="injector1_perf1"
        distanceFromHead="45"/>
      <Perforation
        name="injector1_perf2"
        distanceFromHead="35"/>
      <Perforation
        name="injector1_perf3"
        distanceFromHead="25"/>
      <Perforation
        name="injector1_perf4"
        distanceFromHead="15"/>
      <Perforation
        name="injector1_perf5"
```

(continues on next page)

(continued from previous page)

```
        distanceFromHead="5"/>
    </InternalWell>
  </InternalMesh>
</Mesh>
```

The central wellbore is discretized internally by GEOS (see *CO 2 Injection*). It includes five segments with a perforation in each segment. It has its own region `wellRegion` and control labeled `wellControls` defined and detailed respectively in **ElementRegions** and **Solvers** (see below). In the **ElementRegions** block,

```
<ElementRegions>
  <CellElementRegion
    name="reservoir"
    cellBlocks="{ 1_hexahedra }"
    materialList="{ fluid, rock, relperm, cappres }"/>

  <WellElementRegion
    name="wellRegion"
    materialList="{ fluid, relperm, cappres }"/>
</ElementRegions>
```

one single reservoir region labeled `reservoir`. A second region `wellRegion` is associated with the well. All those regions define materials to be specified inside the **Constitutive** block.

Coupled solver

The simulation is performed by the GEOS coupled solver for multiphase flow and well defined in the XML block **CompositionalMultiphaseReservoir**:

```
  <CompositionalMultiphaseFVM
    name="compositionalMultiphaseFlow"
    targetRegions="{ reservoir }"
    discretization="fluidTPFA"
    temperature="363"
    maxCompFractionChange="0.2"
    logLevel="1"
    useMass="1"/>
```

It references the two coupled solvers under the tags `flowSolverName` and `wellSolverName`. These are defined inside the same **Solvers** block following this coupled solver. It also defined non-linear, **NonlinearSolverParameters** and and linear, **LinearSolverParameters**, strategies.

The next two blocks are used to define our two coupled physics solvers `compositionalMultiphaseFlow` (of type **CompositionalMultiphaseFVM**) and `compositionalMultiphaseWell` (of type **CompositionalMultiphaseWell**).

Flow solver

We use the `targetRegions` attribute to define the regions where the flow solver is applied.

```
<CompositionalMultiphaseReservoir
  name="coupledFlowAndWells"
  flowSolverName="compositionalMultiphaseFlow"
  wellSolverName="compositionalMultiphaseWell"
  logLevel="1"
  initialDt="1e2"
  targetRegions="{ reservoir, wellRegion }">
  <NonlinearSolverParameters
    newtonTol="1.0e-5"
    newtonMaxIter="40"/>
  <LinearSolverParameters
    solverType="fgmres"
    preconditionerType="mgr"
    krylovTol="1e-6"
    logLevel="1"/>
</CompositionalMultiphaseReservoir>
```

The FV scheme discretization used is TPFA (which definition can be found nested in **NumericalMethods/FiniteVolume**) and some parameter values.

Well solver

The well solver is applied on its own region `wellRegion` which consists of the five discretized segments. It is also the place where the **WellControls** are set thanks to `type`, `control`, `injectionStream` , `injectionTemperature`, `targetTotalRateTableName` and, `targetBHP` for instance if we consider an injection well.

For more details on the wellbore modeling please refer to *Compositional Multiphase Well Solver*.

```
<CompositionalMultiphaseWell
  name="compositionalMultiphaseWell"
  targetRegions="{ wellRegion }"
  logLevel="1"
  useMass="1">
  <WellControls
    name="wellControls"
    logLevel="1"
    type="injector"
    control="totalVolRate"
    referenceElevation="-3000"
    targetBHP="1e8"
    enableCrossflow="0"
    useSurfaceConditions="1"
    surfacePressure="101325"
    surfaceTemperature="288.71"
    targetTotalRateTableName="totalRateTable"
    injectionTemperature="353.15"
    injectionStream="{ 1.0, 0.0 }"/>
</CompositionalMultiphaseWell>
```

Constitutive laws

This benchmark test involves a compositional mixture that defines two phases (CO2-rich and aqueous) labeled as gas and water which contain two components co2 and water. The miscibility of CO2 results in the presence of CO2 in the aqueous phase. The vaporization of H2O in the CO2-rich phase is not considered here.

```
    <CO2BrineEzrokhiFluid
      name="fluid"
      phaseNames="{ gas, water }"
      componentNames="{ co2, water }"
      componentMolarWeight="{ 44e-3, 18e-3 }"
      phasePVTParaFiles="{ tables/pvtgas.txt, tables/pvtliquid_ez.txt }"
      flashModelParaFile="tables/co2flash.txt"/>
```

The brine properties are modeled using Ezrokhi correlation, hence the block name **CO2BrineEzrokhiFluid**. The external PVT files *tables/pvtgas.txt* and *tables/pvtliquid_ex.txt* give access to the models considered respectively for the computation of gas density and viscosity and the brine density and viscosity, along with pressure, temperature, salinity discretization of the parameter space. The external file *tables/co2flash.txt* gives the same type of information for the *CO2Solubility* model (see *CO2-brine model* for details).

The rock model defines a slightly compressible porous medium with a reference porosity equal to 0.1.

```
    <CompressibleSolidConstantPermeability
      name="rock"
      solidModelName="nullSolid"
      porosityModelName="rockPorosity"
      permeabilityModelName="rockPerm"/>
    <NullModel
      name="nullSolid"/>
    <PressurePorosity
      name="rockPorosity"
      defaultReferencePorosity="0.1"
      referencePressure="1.0e7"
      compressibility="4.5e-10"/>
    <ConstantPermeability
      name="rockPerm"
      permeabilityComponents="{ 1.0e-12, 1.0e-12, 1.0e-12 }"/>
```

The relative permeability model is input through tables thanks to **TableRelativePermeability** block.

```
    <TableRelativePermeability
      name="relperm"
      phaseNames="{ gas, water }"
      wettingNonWettingRelPermTableNames="{ waterRelativePermeabilityTable,
                                            gasRelativePermeabilityTable }"/>
```

As this benchmark is testing the sensitivity of the plume dynamics to the relative permeability hysteresis modeling, in commented block the **TableRelativePermeabilityHysteresis** block sets up bounding curves for imbibition and drainage under imbibitionNonWettingRelPermTableName, imbibitionWettingRelPermTableName and, drainageWettingNonWettingRelPermTableNames compared to the wettingNonWettingRelPermTableNames label of the drainage only **TableRelativePermeability** blocks. Those link to **TableFunction** blocks in **Functions**, which define sample points for piecewise linear interpolation. This feature is used and explained in more details in the following section dedicated to *Initial and Boundary conditions*.

See,

```
../../../../../../../inputFiles/compositionalMultiphaseWell/benchmarks/Class09Pb3/
↪class09_pb3_hystRelperm_iterative_base.xml
```

for the input base with relative permeability hysteresis setup.

```
    <TableRelativePermeabilityHysteresis
      name="relperm"
      phaseNames="{ gas, water }"
      drainageWettingNonWettingRelPermTableNames="{␣
↪drainageWaterRelativePermeabilityTable,
                                                   drainageGasRelativePermeabilityTable }"
      imbibitionNonWettingRelPermTableName="imbibitionGasRelativePermeabilityTable"
      imbibitionWettingRelPermTableName="imbibitionWaterRelativePermeabilityTable"/>
```

Note: `wettingNonWettingRelPermTableNames` in **TableRelativePermeability** and `drainageWettingNonWettingRelPermTableNames` in **TableRelativePermeabilityHysteresis** are identical.

Capillary pressure is also tabulated and defined in **TableCapillaryPressure**. No hysteresis is modeled yet on the capillary pressure.

```
    <TableCapillaryPressure
      name="cappres"
      phaseNames="{ gas, water }"
      wettingNonWettingCapPressureTableName="waterCapillaryPressureTable"/>
```

Initial and boundary conditions

The domain is initially saturated with brine with a hydrostatic pressure field. This is specified using the **HydrostaticEquilibrium** XML tag in the **FieldSpecifications** block. The datum pressure and elevation used below are defined in (Class et al., 2009)).

```
   <HydrostaticEquilibrium
      name="equil"
      objectPath="ElementRegions"
      datumElevation="-3000"
      datumPressure="3.0e7"
      initialPhaseName="water"
      componentNames="{ co2, water }"
      componentFractionVsElevationTableNames="{ initCO2CompFracTable,
                                                initWaterCompFracTable }"
      temperatureVsElevationTableName="initTempTable"/>
```

In the **Functions** block, the **TableFunction** s named `initCO2CompFracTable` and `initWaterCompFracTable` define the brine-saturated initial state, while the **TableFunction** named `initTempTable` defines the temperature field as a function of depth to impose the geothermal gradient.

The boundaries are set to have a constant 0.03 K/m temperature gradient as well as the hydrostatic pressure gradient. We supplement that with water dominant content. Each block is linking a `fieldName` to a **TableFunction** tagged as the value of `functionName`. In order to have those imposed on the boundary faces, we provide `faceManager` as `objectPath`.

```
<FieldSpecification
  name="bcPressure"
  objectPath="faceManager"
  setNames="{3}"
  fieldName="pressure"
  functionName="pressureFunction"
  scale="1"/>
<FieldSpecification
  name="bcTemperature"
  objectPath="faceManager"
  setNames="{3}"
  fieldName="temperature"
  functionName="temperatureFunction"
  scale="1"/>
<FieldSpecification
  name="bcCompositionCO2"
  objectPath="faceManager"
  setNames="{3}"
  fieldName="globalCompFraction"
  component="0"
  scale="0.000001"/>
<FieldSpecification
  name="bcCompositionWater"
  objectPath="faceManager"
  setNames="{3}"
  fieldName="globalCompFraction"
  component="1"
  scale="0.999999"/>
```

Outputing reservoir statistics

In order to output partitioning of CO2 mass, we use reservoir statistics implemented in GEOS. This is done by defining a **Task**, with `flowSolverName` pointing to the dedicated solver and `computeRegionStatistics` set to 1 to compute statistics by regions. The `setNames` field is set to 3 as it is its attribute tag in the input *vtu* mesh.

```
<CompositionalMultiphaseStatistics
  name="compflowStatistics"
  flowSolverName="compositionalMultiphaseFlow"
  logLevel="1"
  computeCFLNumbers="1"
  computeRegionStatistics="1"/>
```

and an **Event** for this to occur recursively with a *forceDt* argument for the period over which statistics are output and *target* pointing towards the aforementioned **Task**.

```
<PeriodicEvent
  name="statistics"
  timeFrequency="1e5"
  target="/Tasks/compflowStatistics"/>
```

This is a sample of the output we will have in the log file.

```
compflowStatistics, reservoir: Pressure (min, average, max): 2.5139e+07, 2.93668e+07, 3.
↪23145e+07 Pa
compflowStatistics, reservoir: Delta pressure (min, max): -12157, 158134 Pa
compflowStatistics, reservoir: Temperature (min, average, max): 293.15, 371.199, 379.828␣
↪K
compflowStatistics, reservoir: Total dynamic pore volume: 1.57994e+09 rm^3
compflowStatistics, reservoir: Phase dynamic pore volumes: { 1.00239e+06, 1.57894e+09 }␣
↪rm^3
compflowStatistics, reservoir: Phase mass (including both trapped and non-trapped): { 7.
↪24891e+08, 1.6245e+12 } kg
compflowStatistics, reservoir: Trapped phase mass: { 5.3075e+08, 3.2511e+11 } kg
compflowStatistics, reservoir: Dissolved component mass: { { 7.13794e+08, 4.53966e+06 },
↪{ 1.6338e+08, 1.62433e+12 } } kg
compflowStatistics: Max phase CFL number: 65.1284
compflowStatistics: Max component CFL number: 2.32854
```

Note: The log file mentioned above could be an explicit printout of the stdio from MPI launch or the autogenerated output from a SLURM job *slurm.out* or similar

Inspecting results

We request VTK-format output files and use Paraview to visualize the results under the **Outputs** block.

The following figure shows the distribution of CO2 saturation thresholded above a significant value (here 0.001). The displayed cells are colored with respect to the CO2 mass they contain. If the relative permeability for the gas phase drops below 10e-7, the cell is displayed in black.

Fig. 1.13: Plume of CO2 saturation for significant value where immobile CO2 is colored in black.

We observe the importance of hysteresis modeling in CO2 plume migration. Indeed, during the migration phase, the cells at the tail of the plume are switching from drainage to imbibition and the residual CO2 is trapped. This results in a slower migration and expansion of the plume.

To validate the GEOS results, we consider the metrics used in (Class et al., 2009). The reporting values are the dissolved and gaseous CO2 with respect to time using only the drainage relative permeability and using hysteretic relative permeabilities.

We can see that at the end of the injection period the mass of CO2 in the gaseous phase stops increasing and starts decreasing due to dissolution of CO2 in the brine phase. These curves confirm the agreement between GEOS and the results of (Class et al., 2009).

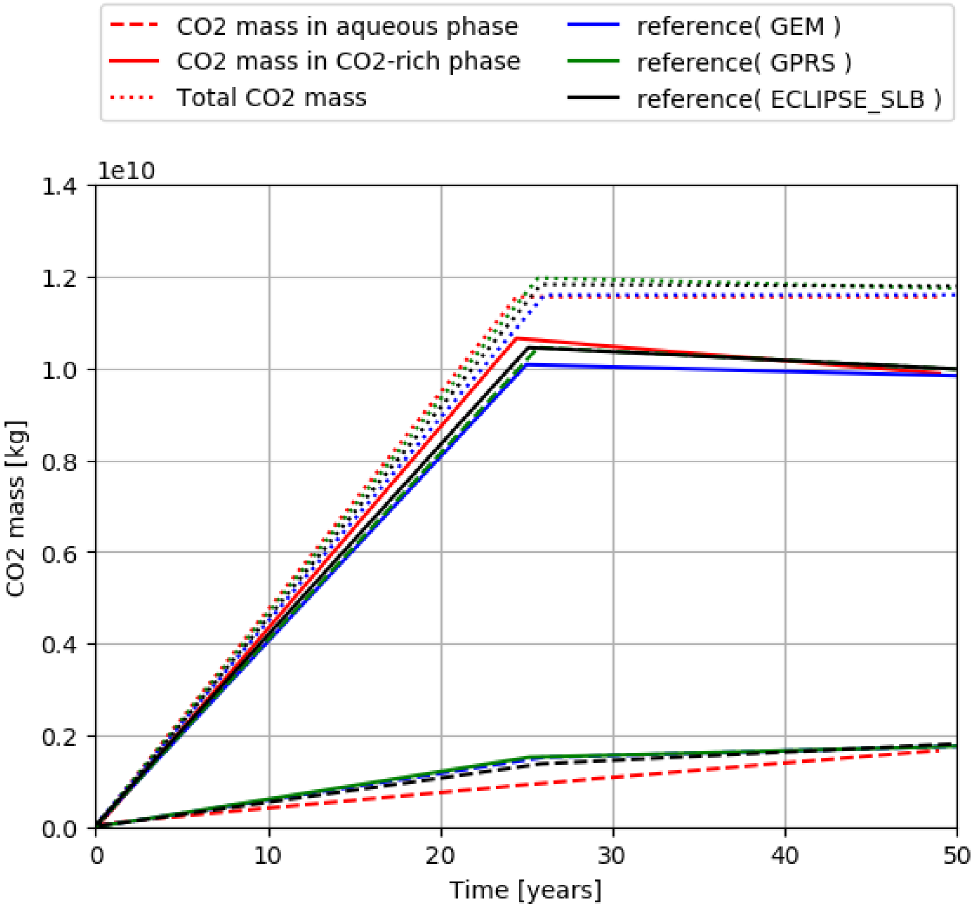


Fig. 1.14: CO2 mass in aqueous and CO2-rich phases as a function of without relative permeability hysteresis

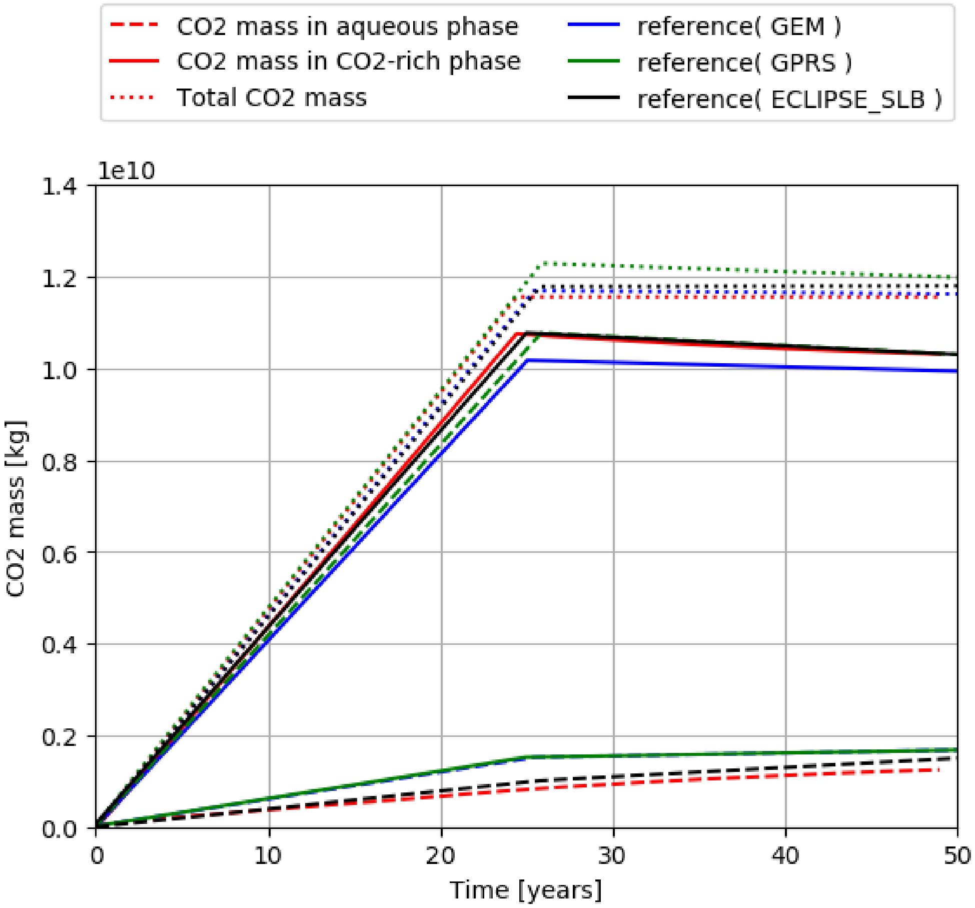


Fig. 1.15: CO2 mass in aqueous and CO2-rich phases as a function of time with relative permeability hysteresis

To go further

Feedback on this example

For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

Faults & fractures

Sneddon's Problem

Objectives

At the end of this example you will know:

- how to define fractures in a porous medium,
- how to use various solvers (EmbeddedFractures, LagrangianContact and HydroFracture) to solve the mechanics problems with fractures.

Input file

This example uses no external input files and everything required is contained within GEOS input files.

The xml input files for the case with EmbeddedFractures solver are located at:

```
inputFiles/efemFractureMechanics/Sneddon_embeddedFrac_base.xml
inputFiles/efemFractureMechanics/Sneddon_embeddedFrac_verification.xml
```

The xml input files for the case with LagrangianContact solver are located at:

```
inputFiles/lagrangianContactMechanics/Sneddon_contactMechanics_base.xml
inputFiles/lagrangianContactMechanics/Sneddon_contactMechanics_benchmark.xml
```

The xml input files for the case with HydroFracture solver are located at:

```
inputFiles/hydraulicFracturing/Sneddon_hydroFrac_base.xml
inputFiles/hydraulicFracturing/Sneddon_hydroFrac_benchmark.xml
```

Description of the case

We compute the displacement field induced by the presence of a pressurized fracture, of length L_f, in a porous medium.

GEOS will calculate the displacement field in the porous matrix and the displacement jump at the fracture surface. We will use the analytical solution for the fracture aperture, w_n (normal component of the jump), to verify the numerical results

$$w_n(s) = \frac{4(1-\nu^2)p_f}{E}\sqrt{\frac{L_f^2}{4} - s^2}$$

where - E is the Young's modulus - ν is the Poisson's ratio - p_f is the fracture pressure - s is the local fracture coordinate in $[-\frac{L_f}{2}, \frac{L_f}{2}]$

In this example, we focus our attention on the `Solvers`, the `ElementRegions`, and the `Geometry` tags.

Mechanics solver

To define a mechanics solver capable of including embedded fractures, we will define two solvers:

- a SolidMechanicsEmbeddedFractures solver, called mechSolve
- a small-strain Lagrangian mechanics solver, of type SolidMechanicsLagrangianSSLE called here matrixSolver (see: *Solid Mechanics Solver*)

Note that the name attribute of these solvers is chosen by the user and is not imposed by GEOS. It is important to make sure that the solidSolverName specified in the embedded fractures solver corresponds to the small-strain Lagrangian solver used in the matrix.

The two single-physics solvers are parameterized as explained in their respective documentation, each with their own tolerances, verbosity levels, target regions, and other solver-specific attributes.

Additionally, we need to specify another solver of type, EmbeddedSurfaceGenerator, which is used to discretize the fracture planes.

To setup a coupling between rock and fracture deformations in LagrangianContact solver, we define three different solvers:

- For solving the frictional contact, we define a Lagrangian contact solver, called here lagrangiancontact. In this solver, we specify targetRegions that include both the continuum region Region and the discontinuum region Fracture where the solver is applied to couple rock and fracture deformations. The contact constitutive law used for the fracture elements is named fractureMaterial, and is defined later in the Constitutive section.
- Rock deformations are handled by a solid mechanics solver SolidMechanicsLagrangianSSLE. The problem runs in QuasiStatic mode without inertial effects. The computational domain is discretized by FE1, which is defined in the NumericalMethods section. The solid material is named rock and its mechanical properties are specified later in the Constitutive section.
- The solver SurfaceGenerator defines the fracture region and rock toughness.

```
<LagrangianContact
  name="lagrangiancontact"
  solidSolverName="lagsolve"
  stabilizationName="TPFAstabilization"
  logLevel="1"
  discretization="FE1"
  targetRegions="{ Region, Fracture }"
  contactRelationName="fractureMaterial"
  fractureRegionName="Fracture">
  <NonlinearSolverParameters
    newtonTol="1.0e-8"
    logLevel="2"
    newtonMaxIter="10"
    maxNumConfigurationAttempts="10"
    lineSearchAction="Require"
    lineSearchMaxCuts="2"
    maxTimeStepCuts="2"/>
  <LinearSolverParameters
    solverType="direct"
    directParallel="0"
    logLevel="0"/>
</LagrangianContact>
```

(continues on next page)

(continued from previous page)

```
  <SolidMechanicsLagrangianSSLE
    name="lagsolve"
    timeIntegrationOption="QuasiStatic"
    logLevel="0"
    discretization="FE1"
    targetRegions="{ Region }"
    >
    <NonlinearSolverParameters
      newtonTol="1.0e-6"
      newtonMaxIter="5"/>
    <LinearSolverParameters
      krylovTol="1.0e-10"
      logLevel="0"/>
  </SolidMechanicsLagrangianSSLE>

  <SurfaceGenerator
    name="SurfaceGen"
    logLevel="0"
    fractureRegion="Fracture"
    targetRegions="{ Region }"
    rockToughness="1.0e6"
    mpiCommOrder="1"/>
</Solvers>
```

Three elementary solvers are combined in the solver `Hydrofracture` to model the coupling between fluid flow within the fracture, rock deformation, fracture opening/closure and propagation:

- Rock and fracture deformation are modeled by the solid mechanics solver `SolidMechanicsLagrangianSSLE`. In this solver, we define `targetRegions` that includes both the continuum region and the fracture region. The name of the contact constitutive behavior is also specified in this solver by the `contactRelationName`, besides the `solidMaterialNames`.
- The single phase fluid flow inside the fracture is solved by the finite volume method in the solver `SinglePhaseFVM`.
- The solver `SurfaceGenerator` defines the fracture region and rock toughness.

```
  <Hydrofracture
    name="hydrofracture"
    solidSolverName="lagsolve"
    flowSolverName="SinglePhaseFlow"
    surfaceGeneratorName="SurfaceGen"
    logLevel="1"
    targetRegions="{ Fracture }"
    contactRelationName="fractureContact"
    maxNumResolves="2">
    <NonlinearSolverParameters
      newtonTol="1.0e-5"
      newtonMaxIter="20"
      lineSearchMaxCuts="3"/>
    <LinearSolverParameters
      directParallel="0"/>
  </Hydrofracture>
```

(continues on next page)

(continued from previous page)

```
  <SolidMechanicsLagrangianSSLE
    name="lagsolve"
    timeIntegrationOption="QuasiStatic"
    discretization="FE1"
    targetRegions="{ Domain, Fracture }"
    contactRelationName="fractureContact"/>

  <SinglePhaseFVM
    name="SinglePhaseFlow"
    discretization="singlePhaseTPFA"
    targetRegions="{ Fracture }"/>

  <SurfaceGenerator
    name="SurfaceGen"
    targetRegions="{ Domain }"
    nodeBasedSIF="1"
    rockToughness="10.0e6"
    mpiCommOrder="1"/>
```

Events

For the case with EmbeddedFractures solver, we add multiple events defining solver applications:

- an event specifying the execution of the `EmbeddedSurfaceGenerator` to generate the fracture elements.
- a periodic event specifying the execution of the embedded fractures solver.
- three periodic events specifying the output of simulations results.

```
<Events
  maxTime="1.0">
  <SoloEvent
    name="preFracture"
    target="/Solvers/SurfaceGenerator"/>

  <PeriodicEvent
    name="solverApplications"
    beginTime="0.0"
    endTime="1.0"
    forceDt="1.0"
    target="/Solvers/mechSolve"/>

  <PeriodicEvent
    name="outputs"
    targetExactTimestep="0"
    target="/Outputs/vtkOutput"/>

  <PeriodicEvent
    name="timeHistoryCollection"
    timeFrequency="1.0"
    targetExactTimestep="0"
```

(continues on next page)

(continued from previous page)

```
    target="/Tasks/displacementJumpCollection" />

  <PeriodicEvent
    name="timeHistoryOutput"
    timeFrequency="1.0"
    targetExactTimestep="0"
    target="/Outputs/timeHistoryOutput"/>
</Events>
```

Similar settings are applied for the other two cases.

Mesh, material properties, and boundary conditions

Last, let us take a closer look at the geometry of this simple problem, if using EmbeddedFractures solver. We use the internal mesh generator to create a large domain ($40\,m \ \times 40\,m \ \times 1\,m$), with one single element along the Z axes, 121 elements along the X axis and 921 elements along the Y axis.

```
<Mesh>
  <InternalMesh
    name="mesh1"
    elementTypes="{ C3D8 }"
    xCoords="{ -20, -4, 4, 20 }"
    yCoords="{ -20, -4, 4, 20 }"
    zCoords="{ 0, 1 }"
    nx="{ 10, 101, 10 }"
    ny="{ 10, 901, 10 }"
    nz="{ 1 }"
    cellBlockNames="{ cb1 }"/>
</Mesh>
```

The mesh for the case with LagrangianContact solver was also created using the internal mesh generator, as parametrized in the `InternalMesh` XML tag. The mesh discretizes the same compational domain ($40\,m \times 40\,m \times 1\,m$) with 300 x 300 x 1 eight-node brick elements in the x, y, and z directions respectively.

```
<Mesh>
  <InternalMesh
    name="mesh1"
    elementTypes="{ C3D8 }"
    xCoords="{ -20, -2, 2, 20 }"
    yCoords="{ -20, -2, 2, 20 }"
    zCoords="{ 0, 1 }"
    nx="{ 40, 220, 40 }"
    ny="{ 40, 220, 40 }"
    nz="{ 1 }"
    cellBlockNames="{ cb1 }"/>
</Mesh>
```

Similarly, the internal mesh generator was used to discretize the same domain ($40\,m \ \times 40\,m \ \times 1\,m$) and generate the mesh for the case with Hydrofracture solver, which contains 280 x 280 x 1 eight-node brick elements in the x, y, and z directions.

```
<Mesh>
  <InternalMesh
    name="mesh1"
    elementTypes="{ C3D8 }"
    xCoords="{ -20, -2, 2, 20 }"
    yCoords="{ -20, -2, 2, 20 }"
    zCoords="{ 0, 1 }"
    nx="{ 40, 200, 40 }"
    ny="{ 40, 200, 40 }"
    nz="{ 1 }"
    cellBlockNames="{ cb1 }"/>
</Mesh>
```

In all the three cases, eight-node hexahedral elements are defined as `C3D8` elementTypes, and their collection forms a mesh with one group of cell blocks named here `cb1`. Refinement is necessary to conform with the fracture geometry specified in the `Geometry` section.

The parameters used in the simulation are summarized in the following table.

Symbol	Parameter	Units	Value
K	Bulk modulus	[GPa]	16.7
G	Shear modulus	[GPa]	10.0
L_f	Fracture length	[m]	2.0
p_f	Fracture pressure	[MPa]	-2.0

Note that the internal fracture pressure has a negative value, due to the negative sign convention for compressive stresses in GEOS.

Material properties and boundary conditions are specified in the `Constitutive` and `FieldSpecifications` sections.

Adding a fracture

The static fracture is defined by a nodeset occupying a small region within the computation domain, where the fracture tends to open upon internal pressurization:

- The test case with EmbeddedFractures solver:

```
<Geometry>
  <Rectangle
    name="FracturePlane"
    normal="{1.0, 0.0, 0.0}"
    origin="{0.0, 0.0, 0.0}"
    lengthVector="{0.0, 1.0, 0.0}"
    widthVector="{0.0, 0.0, 1.0}"
    dimensions="{ 2, 10 }"/>
</Geometry>
```

- The test case with LagrangianContact solver:

```
<Geometry>
  <Rectangle
    name="fracture"
```

(continues on next page)

(continued from previous page)

```
    normal="{1.0, 0.0, 0.0}"
    origin="{0.0, 0.0, 0.0}"
    lengthVector="{0.0, 1.0, 0.0}"
    widthVector="{0.0, 0.0, 1.0}"
    dimensions="{ 2, 10 }"/>

  <Rectangle
    name="core"
    normal="{1.0, 0.0, 0.0}"
    origin="{0.0, 0.0, 0.0}"
    lengthVector="{0.0, 1.0, 0.0}"
    widthVector="{0.0, 0.0, 1.0}"
    dimensions="{ 2, 10 }"/>
</Geometry>
```

- The test case with HydroFracture solver:

```
<Geometry>
  <Box
    name="fracture"
    xMin="{ -0.01,  -1.01, -0.01 }"
    xMax="{ 0.01,   1.01,  1.01 }"/>

  <Box
    name="source"
    xMin="{ -0.01, -0.11, -0.01 }"
    xMax="{ 0.01,  0.11, 1.01 }"/>

  <Box
    name="core"
    xMin="{ -0.01, -10.01, -0.01 }"
    xMax="{ 0.01,  10.01, 1.01 }"/>
</Geometry>
```

To make these cases identical to the analytical example, fracture propagation is not allowed in this example.

Time history function

In the `Tasks` section, `PackCollection` tasks are defined to collect time history information from fields. Either the entire field or specified named sets of indices in the field can be collected. In this example, a task is specified to output fracture aperture (normal opening); however, for different solvers, different `fieldName` and `objectPath` should be called:

- The test case with EmbeddedFractures solver:
- The test case with LagrangianContact solver:

```
<Tasks>
  <PackCollection
    name="displacementJumpCollection"
    objectPath="ElementRegions/Fracture/faceElementSubRegion"
    fieldName="displacementJump"/>
</Tasks>
```

- The test case with Hydrofracture solver:

```
<Tasks>
  <PackCollection
    name="apertureCollection"
    objectPath="ElementRegions/Fracture/faceElementSubRegion"
    fieldName="elementAperture"/>
</Tasks>
```

These tasks are triggered using the `Event` manager with a `PeriodicEvent` defined for these recurring tasks. GEOS writes output files named after the string defined in the `filename` keyword and formatted as HDF5 files. The `TimeHistory` file contains the collected time history information from each specified time history collector. This information includes datasets for the simulation time, element center defined in the local coordinate system, and the time history information. A Python script is used to read and plot any specified subset of the time history data for verification and visualization.

Running GEOS

To run these three cases, use the following commands:

```
path/to/geos -i inputFiles/efemFractureMechanics/Sneddon_embeddedFrac_verification.xml
```

```
path/to/geos -i inputFiles/lagrangianContactMechanics/Sneddon_contactMechanics_benchmark.xml
```

```
path/to/geos -i inputFiles/hydraulicFracturing/Sneddon_hydroFrac_benchmark.xml
```

Inspecting results

This plot compares the analytical solution (continuous lines) with the numerical solutions (markers) for the normal opening of the pressurized fracture. As shown below, consistently, numerical solutions with different solvers correlate very well with the analytical solution.

To go further

Feedback on this example

This concludes the Sneddon example. For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

Single Fracture Under Shear Compression

Context

In this example, a single fracture is simulated using a Lagrange contact model in a 2D infinite domain and subjected to a constant uniaxial compressive remote stress (Franceschini et al., 2020). An analytical solution (Phan et al., 2003) is available for verifying the accuracy of the numerical results, providing an analytical form for the normal traction and slip on the fracture surface due to frictional contact. In this example, the `TimeHistory` function and a Python script are used to output and postprocess multi-dimensional data (traction and displacement on the fracture surface).

Input file

Everything required is contained within two GEOS input files and one mesh file located at:

```
inputFiles/lagrangianContactMechanics/ContactMechanics_SingleFracCompression_base.xml
```

```
inputFiles/lagrangianContactMechanics/ContactMechanics_SingleFracCompression_benchmark.
↪xml
```

```
inputFiles/lagrangianContactMechanics/crackInPlane_benchmark.vtu
```

Description of the case

We simulate an inclined fracture under a compressive horizontal stress (σ), as shown below. This fracture is placed in an infinite, homogeneous, isotropic, and elastic medium. Uniaxial compression and frictional contact on the fracture surface cause mechanical deformation to the surrounding rock and sliding along the fracture plane. For verification purposes, plane strain deformation and Coulomb failure criterion are considered in this numerical model.

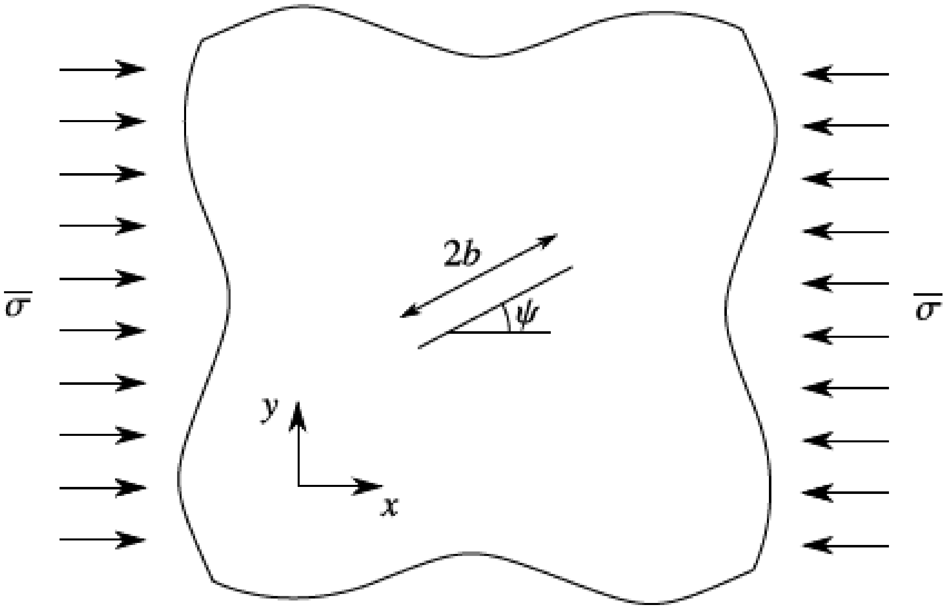


Fig. 1.16: Sketch of the problem

To simulate this phenomenon, we use a Lagrange contact model. Displacement and stress fields on the fracture plane are calculated numerically. Predictions of the normal traction (t_N) and slip (g_T) on the fracture surface are compared with the corresponding analytical solution (Phan et al., 2003).

$$t_N = -\sigma(\sin(\psi))^2$$

$$g_T = \frac{4(1-\nu^2)}{E}(\sigma \sin(\psi)(\cos(\psi) - \sin(\psi)\tan(\theta)))\sqrt{b^2 - (b-\xi)^2}$$

where ψ is the inclination angle, ν is Poisson's ratio, E is Young's modulus, θ is the friction angle, b is the fracture half-length, ξ is a local coordinate on the fracture varying in the range [0, $2b$].

In this example, we focus our attention on the `Mesh` tags, the `Constitutive` tags, and the `FieldSpecifications` tags.

Mesh

The following figure shows the mesh used in this problem.

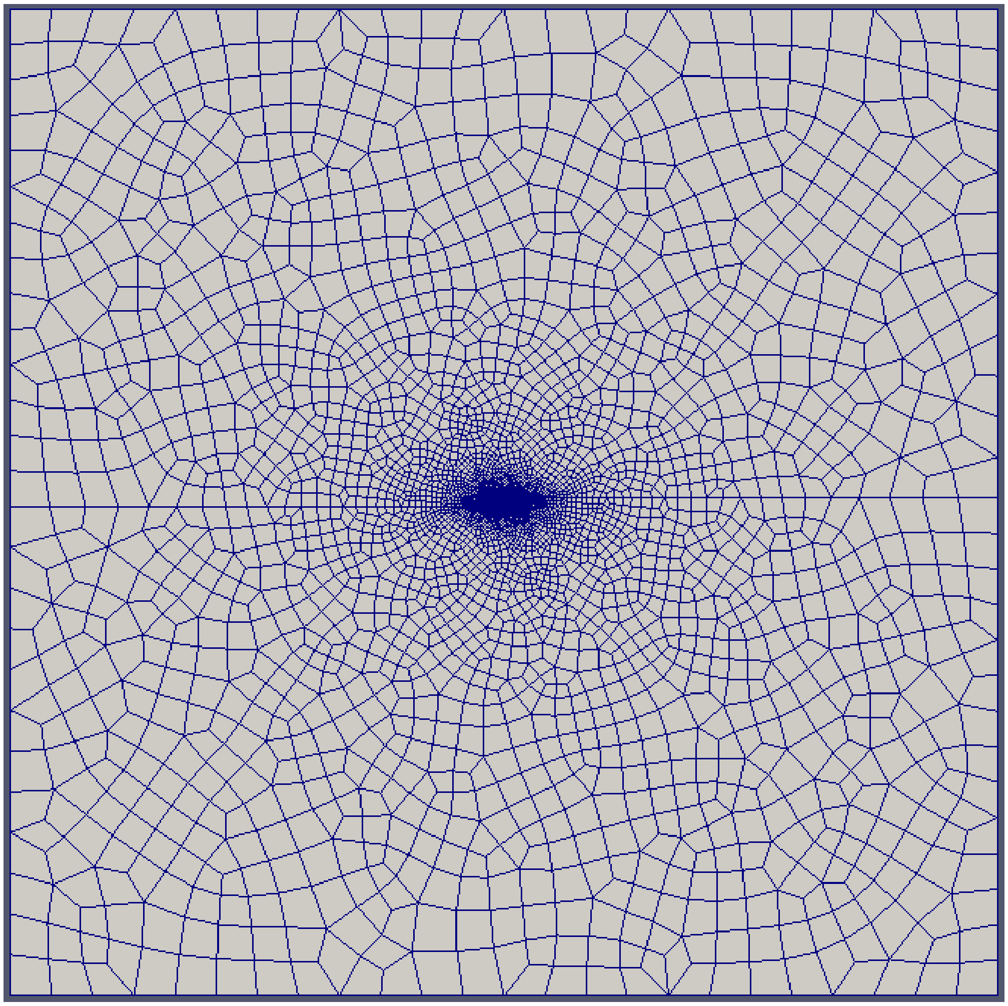


Fig. 1.17: Imported mesh

Here, we load the mesh with VTKMesh (see *Importing the Mesh*). The syntax to import external meshes is simple: in the XML file, the mesh file crackInPlane_benchmark.vtu is included with its relative or absolute path to the location of the GEOS XML file and a user-specified label (here CubeHex) is given to the mesh object. This unstructured mesh contains quadrilaterals elements and interface elements. Refinement is performed to conform with the fracture geometry specified in the Geometry section.

```
<Mesh>
  <VTKMesh
    name="CubeHex"
    file="crackInPlane_benchmark.vtu"/>
</Mesh>

<Geometry>
  <Rectangle
    name="fracture"
```

(continues on next page)

(continued from previous page)

```
      normal="{-0.342020143325669, 0.939692620785908, 0.0}"
      origin="{0.0, 0.0, 0.0}"
      lengthVector="{0.939692620785908, 0.342020143325669, 0.0}"
      widthVector="{0.0, 0.0, 1.0}"
      dimensions="{ 2, 10 }"/>

    <Rectangle
      name="core"
      normal="{-0.342020143325669, 0.939692620785908, 0.0}"
      origin="{0.0, 0.0, 0.0}"
      lengthVector="{0.939692620785908, 0.342020143325669, 0.0}"
      widthVector="{0.0, 0.0, 1.0}"
      dimensions="{ 2, 10 }"/>

    <Box
      name="rightPoint"
      xMin="{ 39.9, -40.1, -0.001}"
      xMax="{ 40.1,  40.1,  0.051}"/>

    <Box
      name="leftPoint"
      xMin="{-40.1, -40.1, -0.001}"
      xMax="{-39.9,  40.1,  0.051}"/>

    <Box
      name="topPoint"
      xMin="{-40.1, 39.9, -0.001}"
      xMax="{ 40.1, 40.1,  0.051}"/>

    <Box
      name="bottomPoint"
      xMin="{-40.1, -40.1, -0.001}"
      xMax="{ 40.1, -39.9,  0.051}"/>

    <Box
      name="front"
      xMin="{-40.1, -40.1, -0.001}"
      xMax="{ 40.1,  40.1,  0.001}"/>

    <Box
      name="rear"
      xMin="{-40.1, -40.1, 0.049}"
      xMax="{ 40.1,  40.1, 0.051}"/>

    <Box
      name="xmin"
      xMin="{-40.1, -40.1, -0.001}"
      xMax="{-39.9,  40.1,  0.051}"/>

    <Box
      name="xmax"
      xMin="{39.9, -40.1, -0.001}"
```

(continues on next page)

(continued from previous page)

```
    xMax="{40.1,  40.1, 0.051}"/>
</Geometry>
```

Solid mechanics solver

GEOS is a multi-physics platform. Different combinations of physics solvers available in the code can be applied in different regions of the domain and be functional at different stages of the simulation. The Solvers tag in the XML file is used to list and parameterize these solvers.

To specify a coupling between two different solvers, we define and characterize each single-physics solver separately. Then, we customize a *coupling solver* between these single-physics solvers as an additional solver. This approach allows for generality and flexibility in constructing multi-physics solvers. Each single-physics solver should be given a meaningful and distinct name because GEOS recognizes these single-physics solvers based on their given names to create the coupling.

To setup a coupling between rock and fracture deformations, we define three different solvers:

- For solving the frictional contact, we define a Lagrangian contact solver, called here lagrangiancontact. In this solver, we specify targetRegions that includes both the continuum region Region and the discontinuum region Fracture where the solver is applied to couple rock and fracture deformation. The contact constitutive law used for the fracture elements is named fractureMaterial, and defined later in the Constitutive section.
- Rock deformations are handled by a solid mechanics solver SolidMechanics_LagrangianFEM. This solid mechanics solver (see *Solid Mechanics Solver*) is based on the Lagrangian finite element formulation. The problem is run as QuasiStatic without considering inertial effects. The computational domain is discretized by FE1, which is defined in the NumericalMethods section. The solid material is named rock, and its mechanical properties are specified later in the Constitutive section.
- The solver SurfaceGenerator defines the fracture region and rock toughness.

```
<Solvers
  gravityVector="{0.0, 0.0, 0.0}">
  <LagrangianContact
    name="lagrangiancontact"
    solidSolverName="lagsolve"
    stabilizationName="TPFAstabilization"
    logLevel="1"
    discretization="FE1"
    targetRegions="{ Region, Fracture }"
    contactRelationName="fractureMaterial"
    fractureRegionName="Fracture">
    <NonlinearSolverParameters
      newtonTol="1.0e-8"
      logLevel="2"
      newtonMaxIter="10"
      maxNumConfigurationAttempts="10"
      lineSearchAction="Require"
      lineSearchMaxCuts="2"
      maxTimeStepCuts="2"/>
    <LinearSolverParameters
      solverType="direct"
      directParallel="0"
      logLevel="0"/>
```

(continues on next page)

(continued from previous page)

```
  </LagrangianContact>

  <SolidMechanics_LagrangianFEM
    name="lagsolve"
    timeIntegrationOption="QuasiStatic"
    logLevel="0"
    discretization="FE1"
    targetRegions="{ Region }">
    <NonlinearSolverParameters
      newtonTol="1.0e-6"
      newtonMaxIter="5"/>
    <LinearSolverParameters
      krylovTol="1.0e-10"
      logLevel="0"/>
  </SolidMechanics_LagrangianFEM>

  <SurfaceGenerator
    name="SurfaceGen"
    logLevel="0"
    fractureRegion="Fracture"
    targetRegions="{ Region }"
    rockToughness="1.0e6"
    mpiCommOrder="1"/>
</Solvers>
```

Constitutive laws

For this specific problem, we simulate the elastic deformation and fracture slippage caused by uniaxial compression. A homogeneous and isotropic domain with one solid material is assumed, with mechanical properties specified in the `Constitutive` section.

Fracture surface slippage is assumed to be governed by the Coulomb failure criterion. The contact constitutive behavior is named `fractureMaterial` in the `Coulomb` block, where cohesion `cohesion="0.0"` and friction coefficient `frictionCoefficient="0.577350269"` are specified.

```
<Constitutive>
  <ElasticIsotropic
    name="rock"
    defaultDensity="2700"
    defaultBulkModulus="16.66666666666666e9"
    defaultShearModulus="1.0e10"/>

  <Coulomb
    name="fractureMaterial"
    cohesion="0.0"
    frictionCoefficient="0.577350269"
    apertureTableName="apertureTable"/>
</Constitutive>
```

Recall that in the `SolidMechanics_LagrangianFEM` section, `rock` is the material of the computational domain. Here, the isotropic elastic model `ElasticIsotropic` is used to simulate the mechanical behavior of `rock`.

All constitutive parameters such as density, bulk modulus, and shear modulus are specified in the International System of Units.

Time history function

In the Tasks section, PackCollection tasks are defined to collect time history information from fields. Either the entire field or specified named sets of indices in the field can be collected. In this example, tractionCollection and displacementJumpCollection tasks are specified to output the local traction fieldName="traction" and relative displacement fieldName="displacementJump" on the fracture surface.

```
<Tasks>
  <PackCollection
    name="tractionCollection"
    objectPath="ElementRegions/Fracture/faceElementSubRegion"
    fieldName="traction"/>

  <PackCollection
    name="displacementJumpCollection"
    objectPath="ElementRegions/Fracture/faceElementSubRegion"
    fieldName="displacementJump"/>
  </Tasks>
```

These two tasks are triggered using the Event management, with PeriodicEvent defined for these recurring tasks. GEOS writes two files named after the string defined in the filename keyword and formatted as HDF5 files (displacementJump_history.hdf5 and traction_history.hdf5). The TimeHistory file contains the collected time history information from each specified time history collector. This information includes datasets for the simulation time, element center defined in the local coordinate system, and the time history information. Then, a Python script is used to access and plot any specified subset of the time history data for verification and visualization.

Initial and boundary conditions

The next step is to specify fields, including:

- The initial value (the remote compressive stress needs to be initialized),
- The boundary conditions (the constraints of the outer boundaries have to be set).

In this tutorial, we specify an uniaxial horizontal stress (σ_x = -1.0e8 Pa). The remaining parts of the outer boundaries are subjected to roller constraints. These boundary conditions are set up through the FieldSpecifications section.

```
<FieldSpecifications>
  <FieldSpecification
    name="frac"
    initialCondition="1"
    setNames="{ fracture }"
    objectPath="faceManager"
    fieldName="ruptureState"
    scale="1"/>

  <FieldSpecification
    name="separableFace"
    initialCondition="1"
    setNames="{ core }"
```

(continues on next page)

(continued from previous page)

```
      objectPath="faceManager"
      fieldName="isFaceSeparable"
      scale="1"/>

    <FieldSpecification
      name="xconstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="0"
      scale="0.0"
      setNames="{ leftPoint, rightPoint }"/>

    <FieldSpecification
      name="yconstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="1"
      scale="0.0"
      setNames="{ bottomPoint, topPoint }"/>

    <FieldSpecification
      name="zconstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="2"
      scale="0.0"
      setNames="{ front, rear }"/>

    <FieldSpecification
      name="Sigmax"
      initialCondition="1"
      setNames="{ all }"
      objectPath="ElementRegions/Region"
      fieldName="rock_stress"
      component="0"
      scale="-1.0e8"/>
  </FieldSpecifications>
```

Note that the remote stress has a negative value, due to the negative sign convention for compressive stresses in GEOS.

The parameters used in the simulation are summarized in the following table.

Symbol	Parameter	Unit	Value
K	Bulk Modulus	[GPa]	16.67
G	Shear Modulus	[GPa]	10.0
σ	Compressive Stress	[MPa]	-100.0
θ	Friction Angle	[Degree]	30.0
ψ	Inclination Angle	[Degree]	20.0
b	Fracture Half Length	[m]	1.0

Inspecting results

We request VTK-format output files and use Paraview to visualize the results. The following figure shows the distribution of u_y in the computational domain.

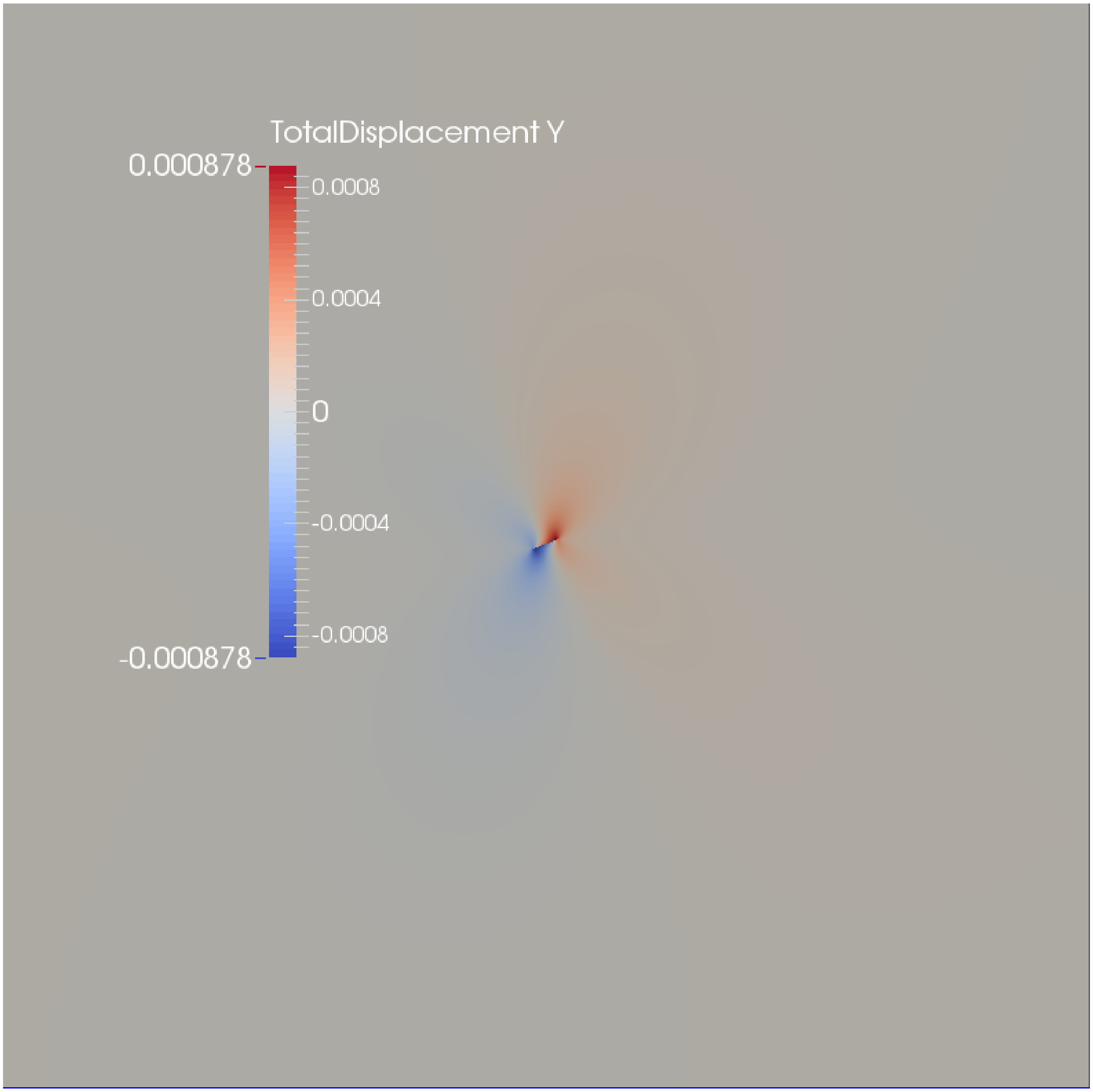


Fig. 1.18: Simulation result of u_y

The next figure shows the distribution of relative shear displacement values on the fracture surface.

The figure below shows a comparison between the numerical predictions (marks) and the corresponding analytical solutions (solid curves) for the normal traction (t_N) and slip (g_T) distributions on the fracture surface. One can observe that the numerical results obtained by GEOS and the analytical solutions are nearly identical.

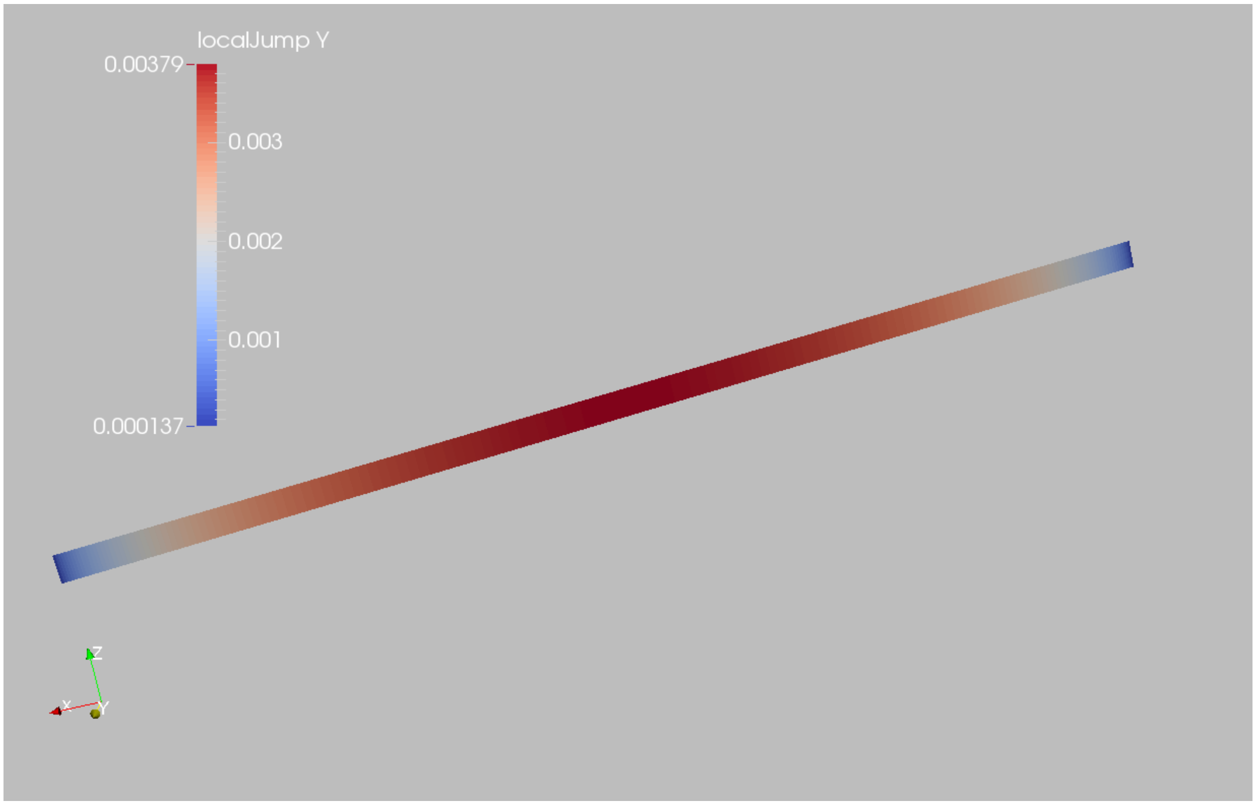
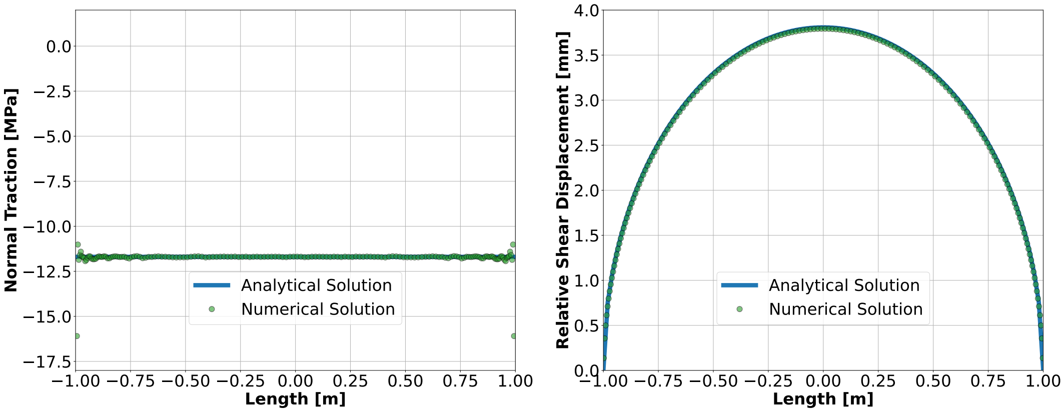


Fig. 1.19: Simulation result of fracture slip



To go further

Feedback on this example

For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

Fracture Intersection Problem

Context

In this example, two fractures intersecting at a right angle are simulated using a Lagrange contact model in a 2D infinite domain and subjected to a constant uniaxial compressive remote stress. Numerical solutions based on the symmetric-Galerkin boundary element method (Phan et al., 2003) is used to verify the accuracy of the GEOS results for the normal traction, normal opening, and shear slippage on the fracture surfaces, considering frictional contact and fracture-fracture interaction. In this example, the `TimeHistory` function and a Python script are used to output and post-process multi-dimensional data (traction and displacement on the fracture surfaces).

Input file

Everything required is contained within two xml files located at:

```
inputFiles/lagrangianContactMechanics/ContactMechanics_TFrac_base.xml
```

```
inputFiles/lagrangianContactMechanics/ContactMechanics_TFrac_benchmark.xml
```

Description of the case

We simulate two intersecting fractures under a remote compressive stress constraint, as shown below. The two fractures sit in an infinite, homogeneous, isotropic, and elastic medium. The vertical fracture is internally pressurized and perpendicularly intersects the middle of the horizontal fracture. A combination of uniaxial compression, frictional contact, and opening of the vertical fracture causes mechanical deformations of the surrounding rock, thus leads to sliding of the horizontal fracture. For verification purposes, a plane strain deformation and Coulomb failure criterion are considered in this numerical model.

To simulate this problem, we use a Lagrange contact model. Displacement and stress fields on the fracture plane are calculated numerically. Predictions of the normal traction and slip along the sliding fracture and mechanical aperture of the pressurized fracture are compared with the corresponding literature work (Phan et al., 2003).

For this example, we focus on the `Mesh`, the `Constitutive`, and the `FieldSpecifications` tags.

Mesh

The following figure shows the mesh used in this problem.

This mesh was created using the internal mesh generator as parametrized in the `InternalMesh` XML tag. The mesh contains 300 x 300 x 1 eight-node brick elements in the x, y, and z directions respectively. Such eight-node hexahedral elements are defined as `C3D8` elementTypes, and their collection forms a mesh with one group of cell blocks named here `cb1`.

```
  <Mesh>
    <InternalMesh
      name="mesh1"
      elementTypes="{ C3D8 }"
```

(continues on next page)

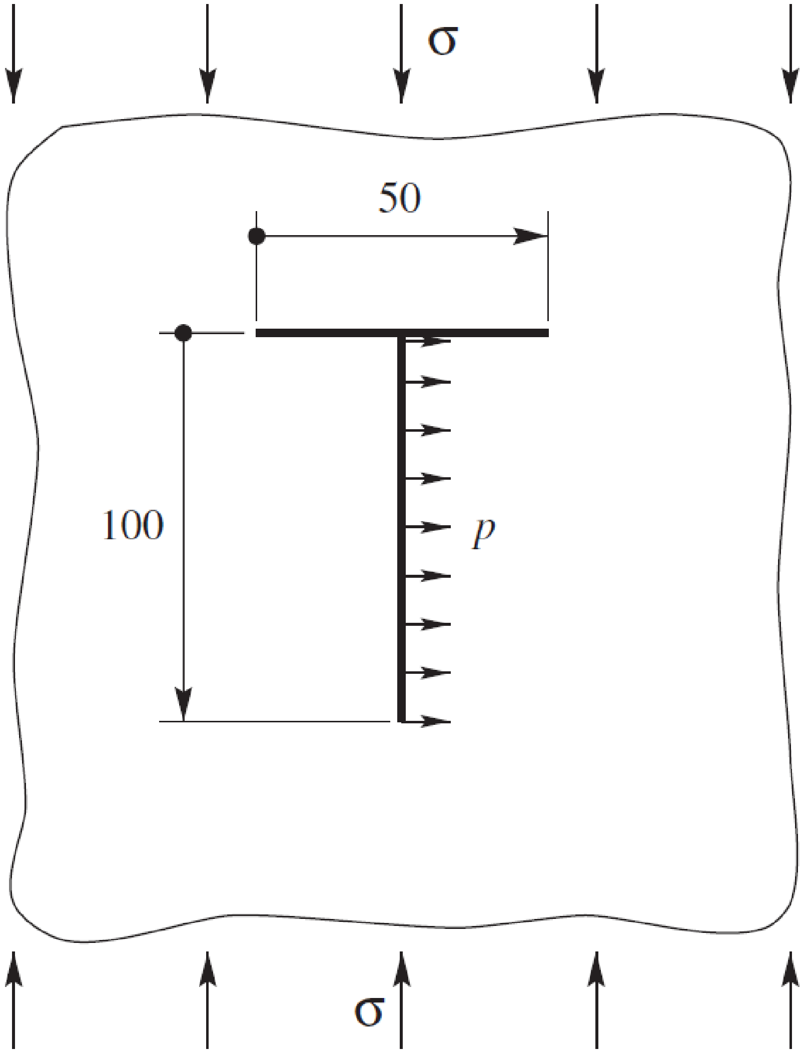


Fig. 1.20: Sketch of the problem (Phan et al., 2003)

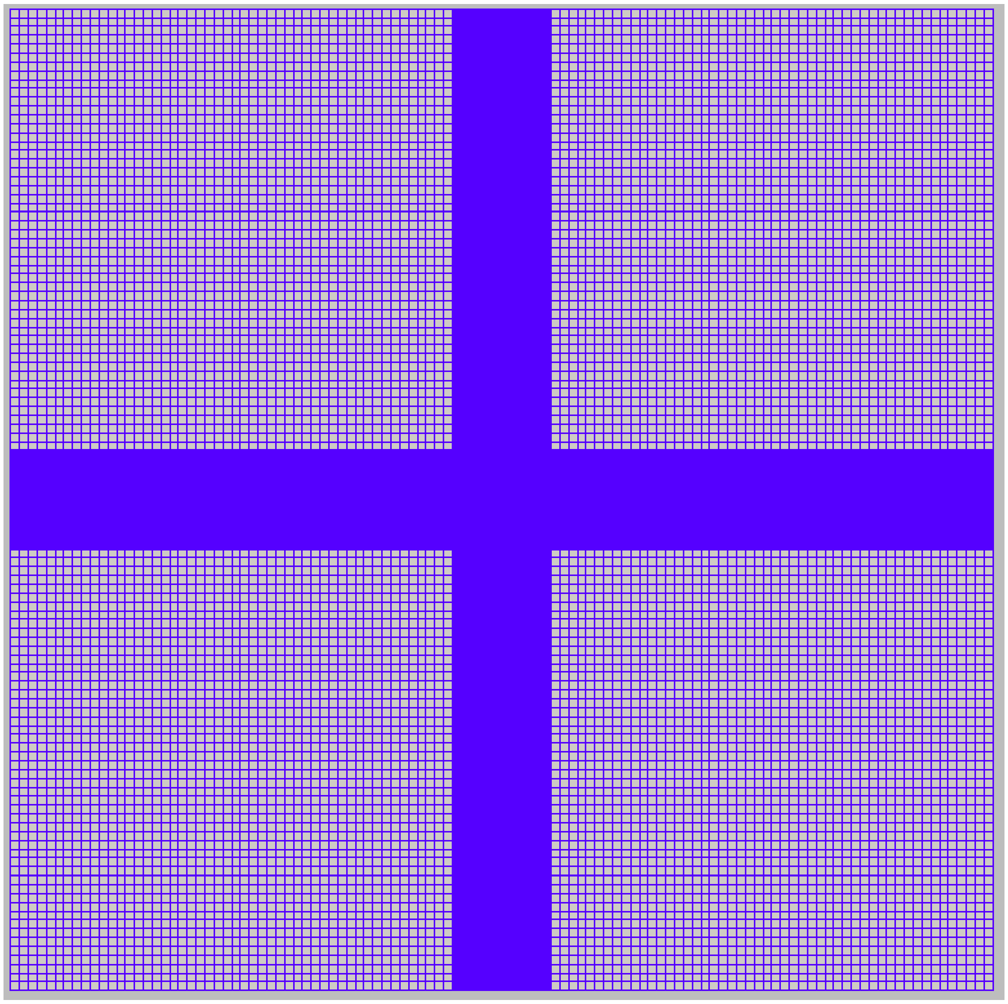


Fig. 1.21: Generated mesh

(continued from previous page)

```
    xCoords="{ -1000, -100, 100, 1000 }"
    yCoords="{ -1000, -100, 100, 1000 }"
    zCoords="{ 0, 1 }"
    nx="{ 50, 200, 50 }"
    ny="{ 50, 200, 50 }"
    nz="{ 1 }"
    cellBlockNames="{ cb1 }"/>
</Mesh>
```

Refinement is necessary to conform with the fracture geometry specified in the `Geometry` section.

```
<Geometry>
  <Rectangle
    name="fracture1"
    normal="{1.0, 0.0, 0.0}"
    origin="{0.0, 0.0, 0.0}"
    lengthVector="{0.0, 1.0, 0.0}"
    widthVector="{0.0, 0.0, 1.0}"
    dimensions="{ 100, 10 }"/>

  <Rectangle
    name="core1"
    normal="{1.0, 0.0, 0.0}"
    origin="{0.0, 0.0, 0.0}"
    lengthVector="{0.0, 1.0, 0.0}"
    widthVector="{0.0, 0.0, 1.0}"
    dimensions="{ 100, 10 }"/>

  <Rectangle
    name="fracture2"
    normal="{0.0, 1.0, 0.0}"
    origin="{0.0, 50.0, 0.0}"
    lengthVector="{1.0, 0.0, 0.0}"
    widthVector="{0.0, 0.0, 1.0}"
    dimensions="{ 50, 10 }"/>

  <Rectangle
    name="core2"
    normal="{0.0, 1.0, 0.0}"
    origin="{0.0, 50.0, 0.0}"
    lengthVector="{1.0, 0.0, 0.0}"
    widthVector="{0.0, 0.0, 1.0}"
    dimensions="{ 50, 10 }"/>
</Geometry>
```

Solid mechanics solver

GEOS is a multiphysics simulation platform. Different combinations of physics solvers can be applied in different regions of the domain at different stages of the simulation. The Solvers tag in the XML file is used to list and parameterize these solvers.

To specify a coupling between two different solvers, we define and characterize each single-physics solver separately. Then, we customize a *coupling solver* between these single-physics solvers as an additional solver. This approach allows for generality and flexibility in constructing multiphysics solvers. Each single-physics solver should be given a meaningful and distinct name, because GEOS recognizes these single-physics solvers by their given names to create the coupling.

To setup a coupling between rock and fracture deformations, we define three different solvers:

- For solving the frictional contact, we define a Lagrangian contact solver, called here lagrangiancontact. In this solver, we specify targetRegions that include both the continuum region Region and the discontinuum region Fracture where the solver is applied to couple rock and fracture deformations. The contact constitutive law used for the fracture elements is named fractureMaterial, and is defined later in the Constitutive section.
- Rock deformations are handled by a solid mechanics solver SolidMechanics_LagrangianFEM. This solid mechanics solver (see *SolidMechanicsLagrangianFEM*) is based on the Lagrangian finite element formulation. The problem runs in QuasiStatic mode without inertial effects. The computational domain is discretized by FE1, which is defined in the NumericalMethods section. The solid material is named rock and its mechanical properties are specified later in the Constitutive section.
- The solver SurfaceGenerator defines the fracture region and rock toughness.

```
    <LagrangianContact
      name="lagrangiancontact"
      solidSolverName="lagsolve"
      stabilizationName="TPFAstabilization"
      logLevel="1"
      discretization="FE1"
      targetRegions="{ Region, Fracture }"
      contactRelationName="fractureMaterial"
      fractureRegionName="Fracture">
      <NonlinearSolverParameters
        newtonTol="1.0e-8"
        logLevel="2"
        maxNumConfigurationAttempts="10"
        newtonMaxIter="10"
        lineSearchAction="Require"
        lineSearchMaxCuts="2"
        maxTimeStepCuts="2"/>
      <LinearSolverParameters
        solverType="direct"
        directParallel="0"
        logLevel="0"/>
    </LagrangianContact>

    <SolidMechanics_LagrangianFEM
      name="lagsolve"
      timeIntegrationOption="QuasiStatic"
      logLevel="0"
      discretization="FE1"
```

(continues on next page)

(continued from previous page)

```
    targetRegions="{ Region }">
    <NonlinearSolverParameters
      newtonTol="1.0e-6"
      newtonMaxIter="5"/>
    <LinearSolverParameters
      krylovTol="1.0e-10"
      logLevel="0"/>
  </SolidMechanics_LagrangianFEM>

  <SurfaceGenerator
    name="SurfaceGen"
    logLevel="0"
    fractureRegion="Fracture"
    targetRegions="{ Region }"
    rockToughness="1.0e6"
    mpiCommOrder="1"/>
```

Constitutive laws

For this problem, we simulate the elastic deformation and fracture slippage caused by the uniaxial compression. A homogeneous and isotropic domain with one solid material is assumed, and its mechanical properties are specified in the `Constitutive` section.

Fracture surface slippage is assumed to be governed by the Coulomb failure criterion. The contact constitutive behavior is named `fractureMaterial` in the `Coulomb` block, where cohesion `cohesion="0.0"` and friction coefficient `frictionCoefficient="0.577350269"` are specified.

```
<Constitutive>
  <ElasticIsotropic
    name="rock"
    defaultDensity="2700"
    defaultBulkModulus="38.89e9"
    defaultShearModulus="29.17e9"/>

  <Coulomb
    name="fractureMaterial"
    cohesion="0.0"
    frictionCoefficient="0.577350269"
    apertureTableName="apertureTable"/>
</Constitutive>
```

Recall that in the `SolidMechanics_LagrangianFEM` section, `rock` is the material of the computational domain. Here, the isotropic elastic model `ElasticIsotropic` is used to simulate the mechanical behavior of `rock`.

All constitutive parameters such as density, bulk modulus, and shear modulus are specified in the International System of Units.

Time history function

In the Tasks section, PackCollection tasks are defined to collect time history information from fields. Either the entire field or specified named sets of indices in the field can be collected. In this example, tractionCollection and displacementJumpCollection tasks are specified to output the local traction fieldName="traction" and relative displacement fieldName="displacementJump" on the fracture surface.

```
<Tasks>
  <PackCollection
    name="tractionCollection"
    objectPath="ElementRegions/Fracture/faceElementSubRegion"
    fieldName="traction"/>

  <PackCollection
    name="displacementJumpCollection"
    objectPath="ElementRegions/Fracture/faceElementSubRegion"
    fieldName="displacementJump"/>
</Tasks>
```

These two tasks are triggered using the Event manager with a PeriodicEvent defined for these recurring tasks. GEOS writes two files named after the string defined in the filename keyword and formatted as HDF5 files (displacementJump_history.hdf5 and traction_history.hdf5). The TimeHistory file contains the collected time history information from each specified time history collector. This information includes datasets for the simulation time, element center defined in the local coordinate system, and the time history information. A Python script is used to read and plot any specified subset of the time history data for verification and visualization.

Initial and boundary conditions

The next step is to specify fields, including:

- The initial value (the remote compressive stress needs to be initialized),
- The boundary conditions (traction loaded on the vertical fracture and the constraints of the outer boundaries have to be set).

In this tutorial, we specify an uniaxial vertical stress SigmaY (σ_y = -1.0e8 Pa). A compressive traction NormalTraction (P_in = -1.0e8 Pa) is applied at the surface of vertical fracture. The remaining parts of the outer boundaries are subjected to roller constraints. These boundary conditions are set up through the FieldSpecifications section.

```
<FieldSpecifications>
  <FieldSpecification
    name="frac"
    initialCondition="1"
    setNames="{ fracture1, fracture2 }"
    objectPath="faceManager"
    fieldName="ruptureState"
    scale="1"/>

  <FieldSpecification
    name="separableFace"
    initialCondition="1"
    setNames="{ core1, core2 }"
    objectPath="faceManager"
```

(continues on next page)

(continued from previous page)

```
      fieldName="isFaceSeparable"
      scale="1"/>

    <FieldSpecification
      name="xconstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="0"
      scale="0.0"
      setNames="{ xpos, xneg }"/>

    <FieldSpecification
      name="yconstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="1"
      scale="0.0"
      setNames="{ ypos, yneg }"/>

    <FieldSpecification
      name="zconstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="2"
      scale="0.0"
      setNames="{ zpos, zneg }"/>

    <Traction
      name="NormalTraction"
      objectPath="faceManager"
      tractionType="normal"
      scale="-1.0e8"
      functionName="ForceTimeFunction"
      setNames="{ core1 }"/>

    <FieldSpecification
      name="SigmaY"
      initialCondition="1"
      setNames="{ all }"
      objectPath="ElementRegions/Region"
      fieldName="rock_stress"
      component="1"
      scale="-1.0e8"/>
  </FieldSpecifications>
```

Note that the remote stress and internal fracture pressure has a negative value, due to the negative sign convention for compressive stresses in GEOS.

The parameters used in the simulation are summarized in the following table.

Symbol	Parameter	Unit	Value
K	Bulk Modulus	[GPa]	38.89
G	Shear Modulus	[GPa]	29.17
σ_y	Remote Stress	[MPa]	-100.0
P_{in}	Internal Pressure	[MPa]	-100.0
θ	Friction Angle	[Degree]	30.0
L_h	Horizontal Frac Length	[m]	50.0
L_v	Vertical Frac Length	[m]	100.0

Inspecting results

We request VTK-format output files and use Paraview to visualize the results. The following figure shows the distribution of σ_{xx} in the computational domain.

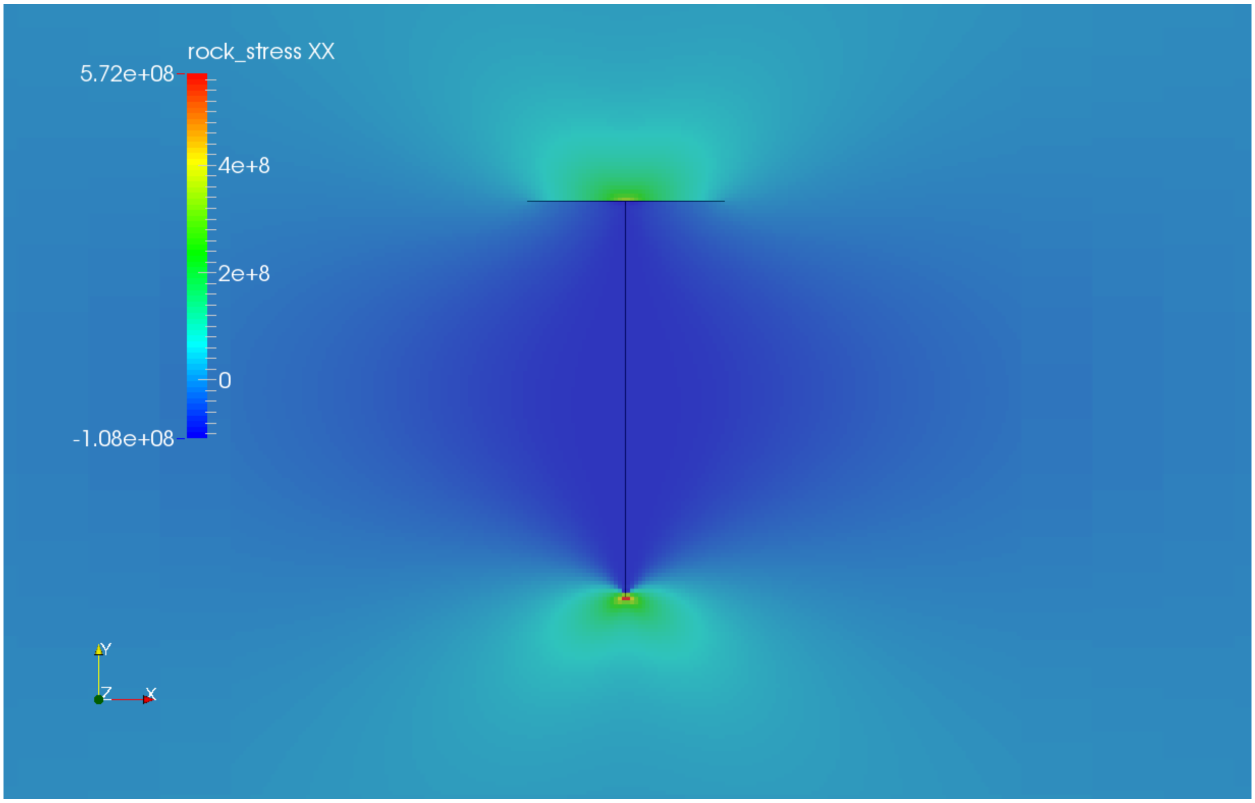


Fig. 1.22: Simulation result of σ_{xx}

The next figure shows the distribution of relative shear displacement values along the surface of two intersected fractures.

The figure below compares the results from GEOS (marks) and the corresponding literature reference solution (solid curves) for the normal traction and slip distributions along the horizontal fracture and opening of the vertical fracture. GEOS reliably captures the mechanical interactions between two intersected fractures and shows excellent agreement with the reference solution. Due to sliding of the horizontal fracture, GEOS prediction as well as the reference solution on the normal opening of pressurized vertical fracture deviates away from Sneddon's analytical solution, especially near the intersection point.

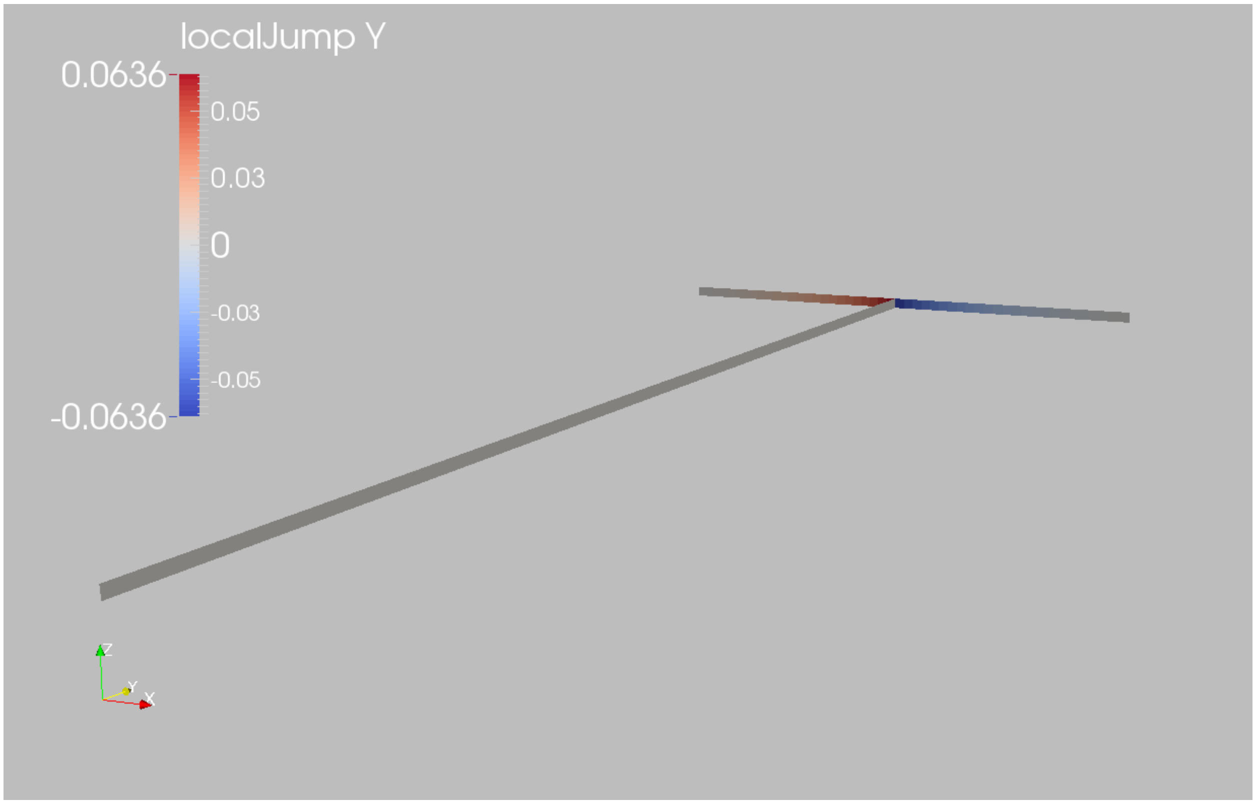
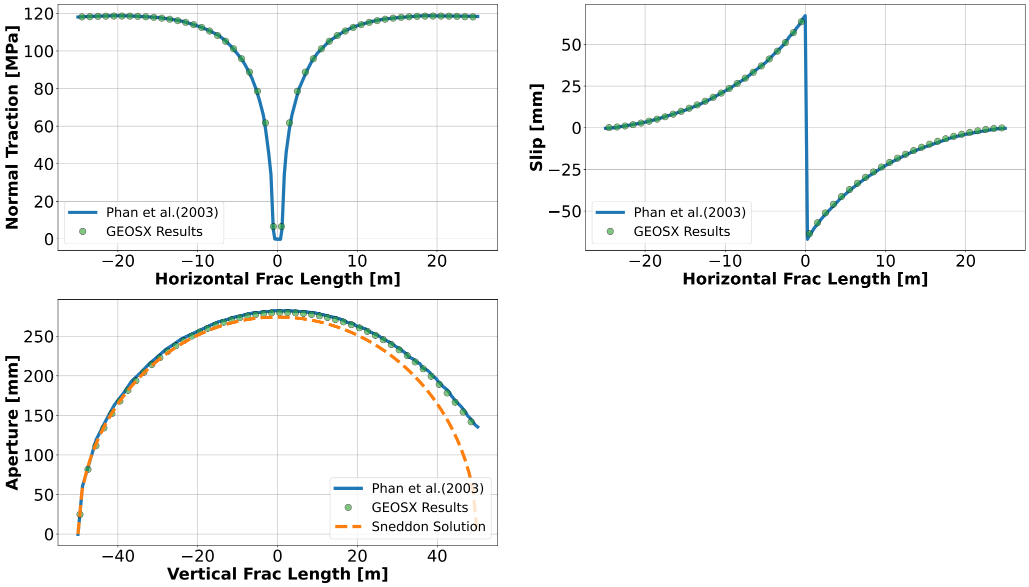


Fig. 1.23: Simulation result of fracture slip



To go further

Feedback on this example

For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

Verification of Induced Stresses Along a Fault

Context

In this example, we evaluate the induced stresses in a pressurized reservoir displaced by a normal fault (permeable or impermeable). This problem is solved using the poroelastic solver in GEOS to obtain the stress perturbations along the fault plane, which are verified against the corresponding analytical solution (Wu et al., 2020).

Input file

The xml input files for the test case with impermeable fault are located at:

```
inputFiles/poromechanics/faultPoroelastic_base.xml
inputFiles/poromechanics/impermeableFault_benchmark.xml
```

The xml input files for the test case with permeable fault are located at:

```
inputFiles/poromechanics/faultPoroelastic_base.xml
inputFiles/poromechanics/permeableFault_benchmark.xml
```

A mesh file and a python script for post-processing the simulation results are also provided:

```
inputFiles/poromechanics/faultMesh.vtu
```

```
src/docs/sphinx/advancedExamples/validationStudies/faultMechanics/faultVerification/
↪faultVerificationFigure.py
```

Description of the case

We simulate induced stresses along a normal fault in a pressurized reservoir and compare our results against an analytical solution. In conformity to the analytical set-up, the reservoir is divided into two parts by an inclined fault. The fault crosses the entire domain, extending into the overburden and the underburden. The domain is horizontal, infinite, homogeneous, isotropic, and elastic. The reservoir is pressurized uniformely upon injection, and we neglect the transient effect of fluid flow. A pressure buildup is applied to: (i) the whole reservoir in the case of a permeable fault; (ii) the left compartment in the case of an impermeable fault. The overburden and underburden are impermeable (no pressure changes). Due to poromechanical effects, pore pressure changes in the reservoir cause a mechanical deformation of the entire domain. This deformation leads to a stress perturbation on the fault plane that could potentially trigger sliding of the fault. Here, the fault serves only as a flow boundary, and the mechanical separation of the fault plane (either by shear slippage or normal opening) is prohibited, like in the analytical example. For verification purposes, a plane strain deformation is considered in the numerical model.

In this example, we set up and solve a poroelastic model to obtain the spatial solutions of displacement and stress fields across the domain upon pressurization. Changes of total stresses along the fault plane are evaluated and compared with the corresponding published work (Wu et al., 2020).

For this example, we focus on the `Mesh`, the `Constitutive`, and the `FieldSpecifications` tags.

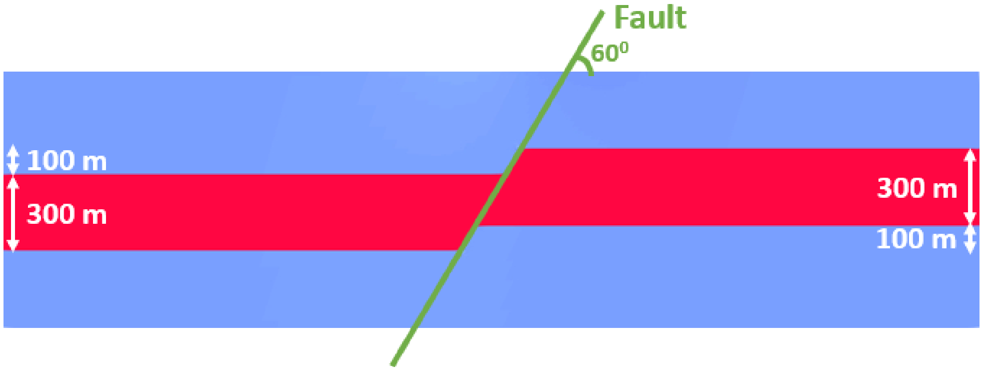


Fig. 1.24: Sketch of the problem

Mesh

The following figure shows the mesh used in this problem.

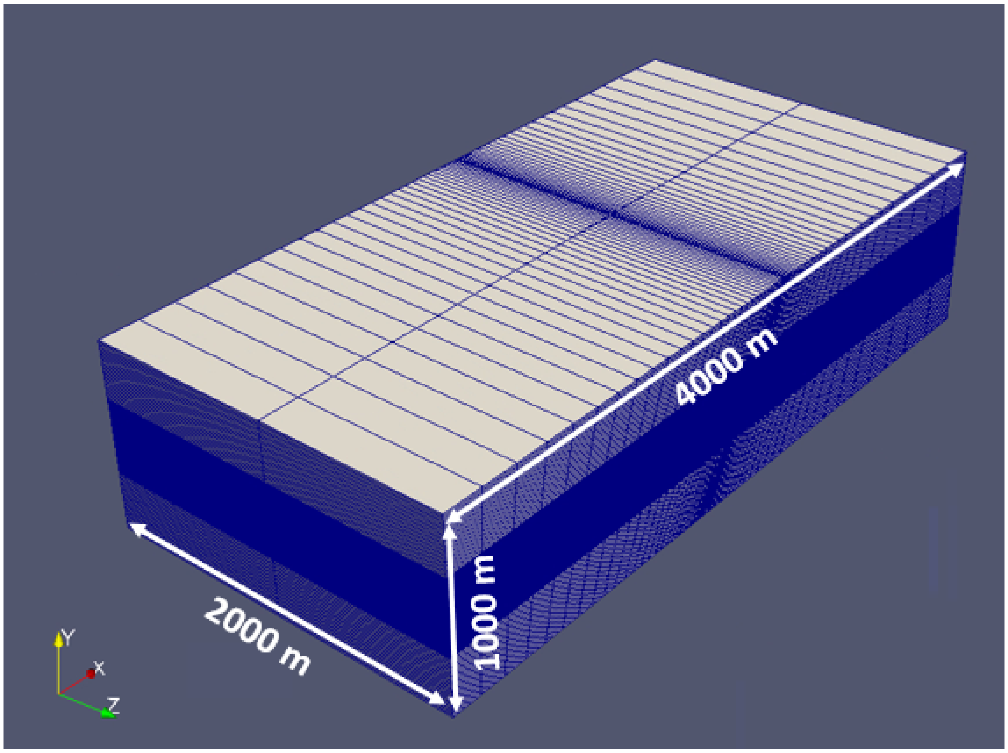


Fig. 1.25: Imported mesh

Here, we load the mesh with VTKMesh. The syntax to import external meshes is simple: in the XML file, the mesh file faultMesh.vtu is included with its relative or absolute path to the location of the GEOS XML file and a user-specified label (here FaultModel) is given to the mesh object. This mesh contains quadrilateral elements and local refinement to conform with the fault geometry, and two reservoir compartments displaced by the fault. The size of the reservoir should be large enough to avoid boundary effects.

```
  <Mesh>
    <VTKMesh
```

(continues on next page)

(continued from previous page)

```
      name="FaultModel"
      file="faultMesh.vtu"
      regionAttribute="CellEntityIds"/>
  </Mesh>
```

Solid mechanics solver

GEOS is a multi-physics platform. Different combinations of physics solvers available in the code can be applied in different regions of the domain and be functional at different stages of the simulation. The `Solvers` tag in the XML file is used to list and parameterize these solvers.

To specify a coupling between two different solvers, we define and characterize each single-physics solver separately. Then, we customize a *coupling solver* between these single-physics solvers as an additional solver. This approach allows for generality and flexibility in constructing multi-physics solvers. The order in which solvers are specified is not important in GEOS. Note that end-users should give each single-physics solver a meaningful and distinct name, as GEOS will recognize these single-physics solvers based on their customized names to create the expected couplings.

As demonstrated in this example, to setup a poromechanical coupling, we need to define three different solvers in the XML file:

- the mechanics solver, a solver of type `SolidMechanics_LagrangianFEM` called here `mechanicsSolver` (more information here: *Solid Mechanics Solver*),

```
    <SolidMechanics_LagrangianFEM
      name="mechanicsSolver"
      timeIntegrationOption="QuasiStatic"
      logLevel="1"
      discretization="FE1"
      targetRegions="{ Domain }">
      <NonlinearSolverParameters
        newtonTol = "1.0e-5"
        newtonMaxIter = "15"/>
      <LinearSolverParameters
        solverType="gmres"
        krylovTol="1.0e-10"/>
    </SolidMechanics_LagrangianFEM>
```

- the single-phase flow solver, a solver of type `SinglePhaseFVM` called here `singlePhaseFlowSolver` (more information on these solvers at *Singlephase Flow Solver*),

```
    <SinglePhaseFVM
      name="singlePhaseFlowSolver"
      logLevel="1"
      discretization="singlePhaseTPFA"
      targetRegions="{ Domain }">
      <NonlinearSolverParameters
        newtonTol = "1.0e-6"
        newtonMaxIter = "8"
      />
      <LinearSolverParameters
        solverType="gmres"
        krylovTol="1.0e-12"/>
```

(continues on next page)

(continued from previous page)

```
  </SinglePhaseFVM>
</Solvers>
```

- the coupling solver (`SinglePhasePoromechanics`) that will bind the two single-physics solvers above, named `poromechanicsSolver` (more information at *Poromechanics Solver*).

```
<Solvers gravityVector="{0.0, 0.0, 0.0}">
  <SinglePhasePoromechanics
    name="poromechanicsSolver"
    solidSolverName="mechanicsSolver"
    flowSolverName="singlePhaseFlowSolver"
    logLevel="1"
    targetRegions="{ Domain }">
    <LinearSolverParameters
      solverType="gmres"
      preconditionerType="mgr"
      logLevel="1"
      krylovAdaptiveTol="1"
    />
    <NonlinearSolverParameters
      newtonMaxIter = "40"
    />
  </SinglePhasePoromechanics>
```

The two single-physics solvers are parameterized as explained in their corresponding documents.

In this example, let us focus on the coupling solver. This solver (`poromechanicsSolver`) uses a set of attributes that specifically describe the coupling process within a poromechanical framework. For instance, we must point this solver to the designated fluid solver (here: `singlePhaseFlowSolver`) and solid solver (here: `mechanicsSolver`). These solvers are forced to interact with all the constitutive models in the target regions (here, we only have one, `Domain`). More parameters are required to characterize a coupling procedure (more information at *Poromechanics Solver*). This way, the two single-physics solvers will be simultaneously called and executed for solving the problem.

Discretization methods for multiphysics solvers

Numerical methods in multiphysics settings are similar to single physics numerical methods. In this problem, we use finite volume for flow and finite elements for solid mechanics. All necessary parameters for these methods are defined in the `NumericalMethods` section.

As mentioned before, the coupling solver and the solid mechanics solver require the specification of a discretization method called `FE1`. In GEOS, this discretization method represents a finite element method using linear basis functions and Gaussian quadrature rules. For more information on defining finite elements numerical schemes, please see the dedicated *Finite Element Discretization* section.

The finite volume method requires the specification of a discretization scheme. Here, we use a two-point flux approximation scheme (`singlePhaseTPFA`), as described in the dedicated documentation (found here: *Finite Volume Discretization*).

```
<NumericalMethods>
  <FiniteElements>
    <FiniteElementSpace
      name="FE1"
      order="1"/>
```

(continues on next page)

(continued from previous page)

```
  </FiniteElements>

  <FiniteVolume>
    <TwoPointFluxApproximation
      name="singlePhaseTPFA"
      />
  </FiniteVolume>
</NumericalMethods>
```

Constitutive laws

For this problem, a homogeneous and isotropic domain with one solid material is assumed for both the reservoir and its surroundings. The solid and fluid materials are named as `rock` and `water` respectively, and their mechanical properties are specified in the `Constitutive` section. `PorousElasticIsotropic` model is used to describe the linear elastic isotropic response of `rock` when subjected to fluid injection. And the single-phase fluid model `CompressibleSinglePhaseFluid` is selected to simulate the flow of `water`.

```
<Constitutive>
  <PorousElasticIsotropic
    name="porousRock"
    solidModelName="rock"
    porosityModelName="rockPorosity"
    permeabilityModelName="rockPerm"
  />

  <ElasticIsotropic
    name="rock"
    defaultDensity="2700"
    defaultYoungModulus="14.95e9"
    defaultPoissonRatio="0.15"
  />

  <CompressibleSinglePhaseFluid
    name="water"
    defaultDensity="1000"
    defaultViscosity="0.001"
    referencePressure="0e6"
    referenceDensity="1000"
    compressibility="2.09028227021e-10"
    referenceViscosity="0.001"
    viscosibility="0.0"
  />

  <BiotPorosity
    name="rockPorosity"
    grainBulkModulus="7.12e10"
    defaultReferencePorosity="0.3"
  />

  <ConstantPermeability
    name="rockPerm"
```

(continues on next page)

(continued from previous page)

```
    permeabilityComponents="{1.0e-18, 1.0e-18, 1.0e-18}"
  />
</Constitutive>
```

All constitutive parameters such as density, viscosity, and Young's modulus are specified in the International System of Units.

Initial and boundary conditions

The next step is to specify fields, including:

- The initial value (the in-situ stresses and pore pressure have to be initialized),
- The boundary conditions (pressure buildup within the reservoir and constraints of the outer boundaries have to be set).

In this example, we need to specify isotropic horizontal total stress (σ_h = -60.0 MPa and σ_H = -60.0 MPa), vertical total stress (σ_v = -70.0 MPa), and initial reservoir pressure (P_0 = 35.0 MPa). When initializing the model, a normal traction (`name="NormalTraction"`) of -70.0 MPa is imposed on the upper boundary (`setNames="{ 91 }"`) to reach mechanical equilibrium. The lateral and lower boundaries are subjected to roller constraints. These boundary conditions are set up through the `FieldSpecifications` section.

```
<FieldSpecifications>
  <FieldSpecification
    name="initialPressure"
    initialCondition="1"
    setNames="{all}"
    objectPath="ElementRegions/Domain"
    fieldName="pressure"
    scale="35.0e6"
  />

  <FieldSpecification
    name="stressXX"
    initialCondition="1"
    setNames="{all}"
    objectPath="ElementRegions/Domain"
    fieldName="rock_stress"
    component="0"
    scale="-28.499545e6"
  />

  <FieldSpecification
    name="stressYY"
    initialCondition="1"
    setNames="{all}"
    objectPath="ElementRegions/Domain"
    fieldName="rock_stress"
    component="1"
    scale="-38.499545e6"
  />

  <FieldSpecification
```

(continues on next page)

(continued from previous page)

```
      name="stressZZ"
      initialCondition="1"
      setNames="{all}"
      objectPath="ElementRegions/Domain"
      fieldName="rock_stress"
      component="2"
      scale="-28.499545e6"
    />

    <FieldSpecification
      name="xconstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="0"
      scale="0.0"
      setNames="{ 89, 88 }"/>

    <FieldSpecification
      name="yconstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="1"
      scale="0.0"
      setNames="{ 90 }"/>

    <FieldSpecification
      name="zconstraintFront"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="2"
      scale="0.0"
      setNames="{ 92, 93 }"/>

    <Traction
      name="NormalTraction"
      objectPath="faceManager"
      tractionType="normal"
      scale="-70.0e6"
      setNames="{ 91 }"/>
  </FieldSpecifications>
```

In this example, the only difference between the impermeable fault and permeable fault cases is how to apply pressure buildup. For the impermeable fault case, a constant pressure buildup is imposed to the left compartment of the reservoir (`objectPath="ElementRegions/Domain/97_hexahedra"`):

```
  <FieldSpecifications>
    <FieldSpecification
      name="injection"
      initialCondition="0"
      setNames="{all}"
      objectPath="ElementRegions/Domain/97_hexahedra"
      fieldName="pressure"
```

(continues on next page)

(continued from previous page)

```
    scale="55.0e6"/>
</FieldSpecifications>
```

For the permeable fault case, a constant pressure buildup is imposed to both compartments of the reservoir: (`objectPath="ElementRegions/Domain/97_hexahedra"` and `objectPath="ElementRegions/Domain/96_hexahedra"`):

```
<FieldSpecifications>
 <FieldSpecification
    name="injection"
    initialCondition="0"
    setNames="{all}"
    objectPath="ElementRegions/Domain/97_hexahedra"
    fieldName="pressure"
    scale="55.0e6"/>

  <FieldSpecification
    name="injection2"
    initialCondition="0"
    setNames="{all}"
    objectPath="ElementRegions/Domain/96_hexahedra"
    fieldName="pressure"
    scale="55.0e6"/>
</FieldSpecifications>
```

The parameters used in the simulation are summarized in the following table, which are specified in the `Constitutive` and `FieldSpecifications` sections. Note that stresses and traction have negative values, due to the negative sign convention for compressive stresses in GEOS.

Symbol	Parameter	Unit	Value
E	Young's Modulus	[GPa]	14.95
ν	Poisson's Ratio	[-]	0.15
σ_h	Min Horizontal Stress	[MPa]	-60.0
σ_H	Max Horizontal Stress	[MPa]	-60.0
σ_v	Vertical Stress	[MPa]	-70.0
p_0	Initial Reservoir Pressure	[MPa]	35.0
Δp	Pressure Buildup	[MPa]	20.0
K_s	Grain Bulk Modulus	[GPa]	71.2
θ	Fault Dip	[Degree]	60.0
κ	Matrix Permeability	[m^2]	1.0*10^{-18}
ϕ	Porosity	[-]	0.3
D_L	Domain Length	[m]	4000.0
D_W	Domain Width	[m]	2000.0
D_T	Domain Thickness	[m]	1000.0
Res_T	Reservoir Thickness	[m]	300.0
F_{off}	Fault Vertical Offset	[m]	100.0

Inspecting results

We request VTK-format output files and use Paraview to visualize the results. The following figure shows the distribution of resulting shear stress (σ_{xy}) in the computational domain for two different cases (a permeable vs. an impermeable fault). Numerical solutions for both cases are also compared with the corresponding analytical solutions.

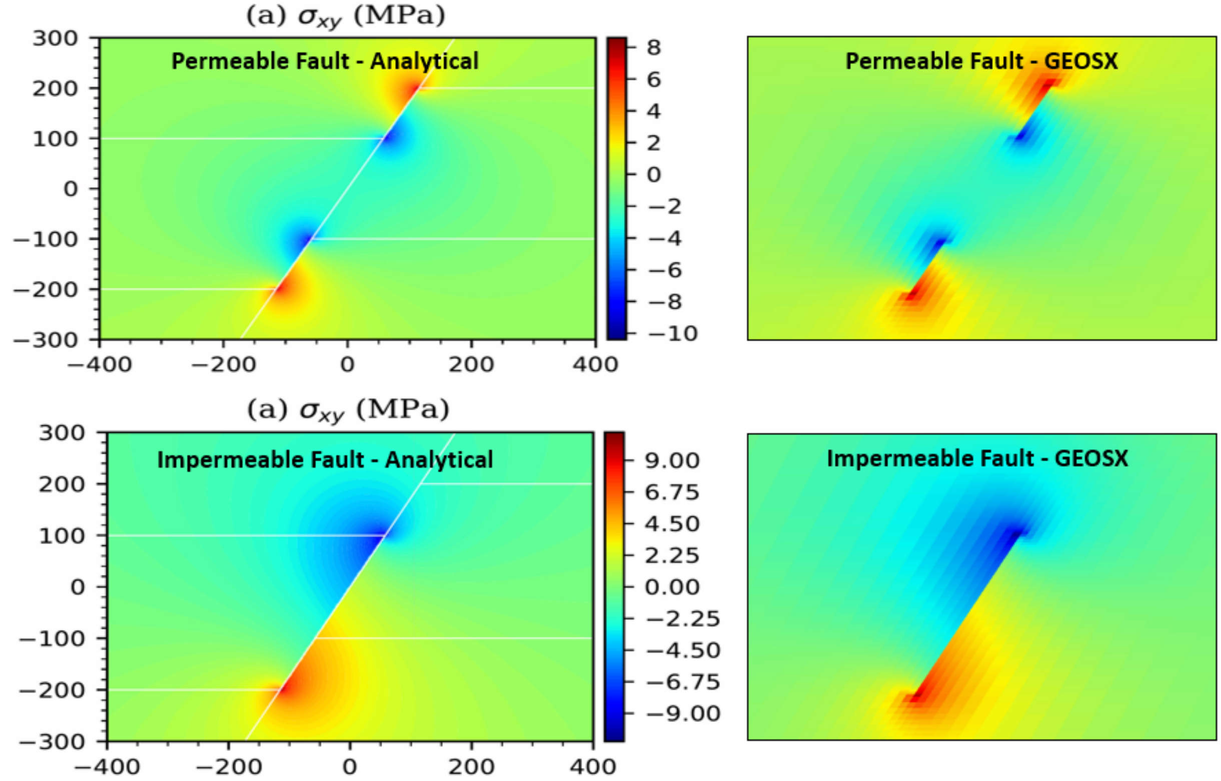
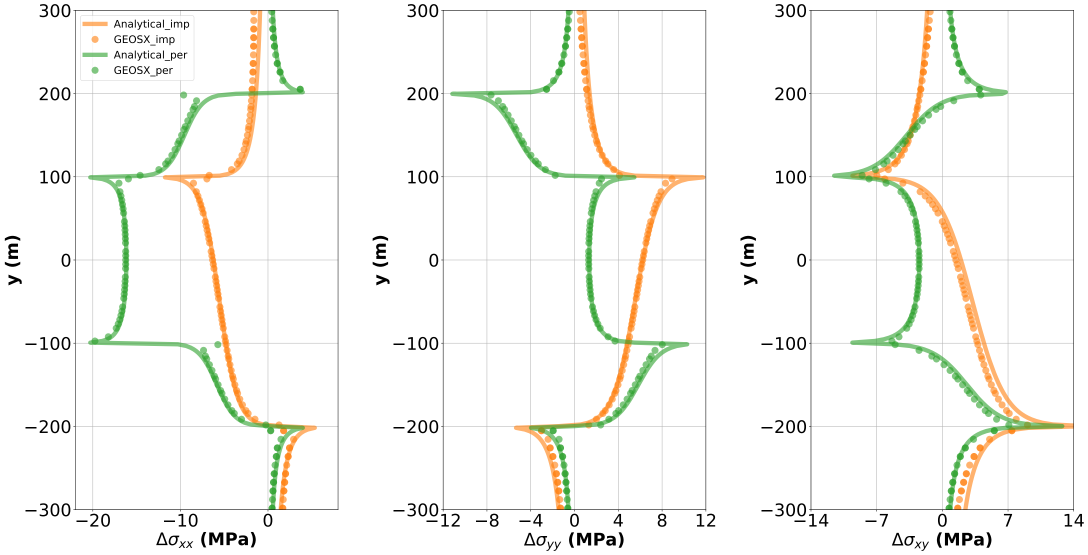


Fig. 1.26: Simulation results of σ_{xy}

The figure below compares the results from GEOS (marks) and the corresponding analytical solution (solid curves) for the change of total stresses (σ_{xx}, σ_{yy} and σ_{xy}) along the fault plane. As shown, GEOS reliably captures the mechanical deformation of the faulted reservoir and shows excellent agreement with the analytical solutions for two different scenarios. Differences in the stress perturbations between the cases with permeable and impermeable fault are also noticeable, which suggests that fault permeability plays a crucial role in governing reservoir deformation for the problems with reservoir pressurization or depletion.



To go further

Feedback on this example

For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

Hydraulic Fracture

Toughness dominated KGD hydraulic fracture

Description of the case

In this example, we consider a plane-strain hydraulic fracture propagating in an infinite, homogeneous and elastic medium, due to fluid injection at a rate Q_0 during a period from 0 to t_{max}. Two dimensional KGD fracture is characterized as a vertical fracture with a rectangle-shaped cross section. For verification purpose, the presented numerical model is restricted to the assumptions used to analytically solve this problem (Bunger et al., 2005). Vertical and impermeable fracture surface is assumed, which eliminate the effect of fracture plane inclination and fluid leakoff. The injected fluid flows within the fracture, which is assumed to be governed by the lubrication equation resulting from the mass conservation and the Poiseuille law. Fracture profile is related to fluid pressure distribution, which is mainly dictated by fluid viscosity μ. In addition, fluid pressure contributes to the fracture development through the mechanical deformation of the solid matrix, which is characterized by rock elastic properties, including the Young modulus E, and the Poisson ratio ν.

For toughness-dominated fractures, more work is spent to split the intact rock than that applied to move the fracturing fluid. To make the case identical to the toughness dominated asymptotic solution, incompressible fluid with an ultra-low viscosity of 0.001 cp and medium rock toughness should be defined. Fracture is propagating with the creation of new surface if the stress intensity factor exceeds rock toughness K_{Ic}.

In toughness-storage dominated regime, asymptotic solutions of the fracture length ℓ, the net pressure p_0 and the fracture aperture w_0 at the injection point for the KGD fracture are provided by (Bunger et al., 2005):

$$\ell = 0.9324 X^{-1/6} (\frac{E_p Q_0^3}{12\mu})^{1/6} t^{2/3}$$

$$w_0^2 = 0.5 X^{1/2} (\frac{12\mu Q_0}{E_p})^{1/2} \ell$$

$$w_0 p_0 = 0.125 X^{1/2} (12\mu Q_0 E_p)^{1/2}$$

where the plane modulus E_p is defined by

$$E_p = \frac{E}{1-\nu^2}$$

```
inputFiles/hydraulicFracturing/kgdToughnessDominated_benchmark.xml
```

The corresponding integrated test with coarser mesh and smaller injection duration is also prepared:

```
inputFiles/hydraulicFracturing/kgdToughnessDominated_Smoke.xml
```

Python scripts for post-processing and visualizing the simulation results are also prepared:

```
inputFiles/hydraulicFracturing/scripts/hydrofractureQueries.py
```

```
inputFiles/hydraulicFracturing/scripts/hydrofractureFigure.py
```

Mechanics solvers

The solver `SurfaceGenerator` defines rock toughness K_{Ic} as:

```
    <SurfaceGenerator
      name="SurfaceGen"
      targetRegions="{ Domain }"
      nodeBasedSIF="1"
      rockToughness="1e6"
      mpiCommOrder="1"/>
```

Rock and fracture deformation are modeled by the solid mechanics solver `SolidMechanicsLagrangianSSLE`. In this solver, we define `targetRegions` that includes both the continuum region and the fracture region. The name of the contact constitutive behavior is also specified in this solver by the `contactRelationName`, besides the `solidMaterialNames`.

```
    <SolidMechanicsLagrangianSSLE
      name="lagsolve"
      timeIntegrationOption="QuasiStatic"
      discretization="FE1"
      targetRegions="{ Domain, Fracture }"
      contactRelationName="fractureContact"/>
```

The single phase fluid flow inside the fracture is solved by the finite volume method in the solver `SinglePhaseFVM` as:

```
    <SinglePhaseFVM
      name="SinglePhaseFlow"
      discretization="singlePhaseTPFA"
      targetRegions="{ Fracture }"/>
```

All these elementary solvers are combined in the solver `Hydrofracture` to model the coupling between fluid flow within the fracture, rock deformation, fracture opening/closure and propagation. A fully coupled scheme is defined by setting a flag `FIM` for `couplingTypeOption`.

```
    <Hydrofracture
      name="hydrofracture"
      solidSolverName="lagsolve"
      flowSolverName="SinglePhaseFlow"
      surfaceGeneratorName="SurfaceGen"
            logLevel="1"
```

(continues on next page)

(continued from previous page)

```
      targetRegions="{ Fracture }"
      contactRelationName="fractureContact"
      maxNumResolves="2">
```

The constitutive laws

The constitutive law `CompressibleSinglePhaseFluid` defines the default and reference fluid viscosity, compressibility and density. For this toughness dominated example, ultra low fluid viscosity is used:

```
    <CompressibleSinglePhaseFluid
      name="water"
      defaultDensity="1000"
      defaultViscosity="1.0e-6"
      referencePressure="0.0"
      compressibility="5e-10"
      referenceViscosity="1.0e-6"
      viscosibility="0.0"/>
```

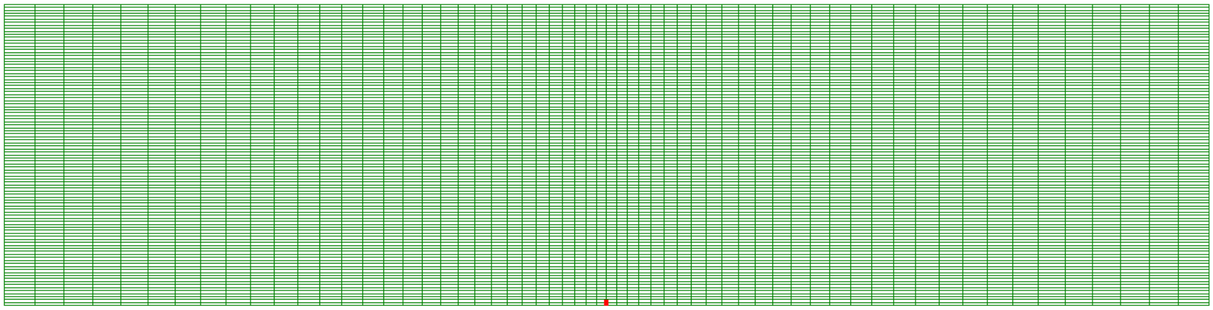
The isotropic elastic Young modulus and Poisson ratio are defined in the `ElasticIsotropic` block. The density of rock defined in this block is useless, as gravity effect is ignored in this example.

```
    <ElasticIsotropic
      name="rock"
      defaultDensity="2700"
      defaultYoungModulus="30.0e9"
      defaultPoissonRatio="0.25"/>
```

Mesh

Internal mesh generator is used to generate the geometry of this example. The domain size is large enough comparing to the final size of the fracture. A sensitivity analysis has shown that the domain size in the direction perpendicular to the fracture plane, i.e. x-axis, must be at least ten times of the final fracture half-length to minimize the boundary effect. However, smaller size along the fracture plane, i.e. y-axis, of only two times the fracture half-length is good enough. It is also important to note that at least two layers are required in z-axis to ensure a good match between the numerical results and analytical solutions, due to the node based fracture propagation criterion. Also in x-axis, bias parameter `xBias` is added for optimizing the mesh by refining the elements near the fracture plane.

```
    <InternalMesh
      name="mesh1"
      elementTypes="{C3D8}"
      xCoords="{ -100, 0, 100 }"
      yCoords="{ 0, 50 }"
      zCoords="{ 0, 1 }"
      nx="{ 30, 30 }"
      ny="{ 100 }"
      nz="{ 2 }"
      xBias="{ 0.5, -0.5 }"
      cellBlockNames="{cb1}"/>
```



Defining the initial fracture

The initial fracture is defined by a nodeset occupying a small area where the KGD fracture starts to propagate:

```
    <Box
      name="fracture"
      xMin="{ -0.01, -0.01, -0.01 }"
      xMax="{  0.01,  1.01,  1.01 }"/>
```

This initial `ruptureState` condition must be specified for this area in the following `FieldSpecification` block:

```
    <FieldSpecification
      name="frac"
      initialCondition="1"
      setNames="{ fracture }"
      objectPath="faceManager"
      fieldName="ruptureState"
      scale="1"/>
```

Defining the fracture plane

The plane within which the KGD fracture propagates is predefined to reduce the computational cost. The fracture plane is outlined by a separable nodeset by the following initial `FieldSpecification` condition:

```
    <Box
      name="core"
      xMin="{ -0.01, -0.01, -0.01 }"
      xMax="{  0.01, 50.01,  1.01 }"/>
```

```
    <FieldSpecification
      name="separableFace"
      initialCondition="1"
      setNames="{ core }"
      objectPath="faceManager"
      fieldName="isFaceSeparable"
      scale="1"/>
```

Defining the injection rate

Fluid is injected into a sub-area of the initial fracture. Only half of the injection rate is defined in this boundary condition because only half-wing of the KGD fracture is modeled regarding its symmetry. Hereby, the mass injection rate is actually defined, instead of the volume injection rate. More precisely, the value given for `scale` is $Q_0 \rho_f / 2$ (not $Q_0/2$).

```
    <SourceFlux
      name="sourceTerm"
      objectPath="ElementRegions/Fracture"
      scale="-5e-2"
      setNames="{ source }"/>
```

Time history function

In the `Tasks` section, `PackCollection` tasks are defined to collect time history information from fields. Either the entire field or specified named sets of indices in the field can be collected. In this example, `pressureCollection`, `apertureCollection`, `hydraulicApertureCollection` and `areaCollection` are specified to output the time history of fracture characterisctics (pressure, width and area). `objectPath="ElementRegions/Fracture/FractureSubRegion"` indicates that these `PackCollection` tasks are applied to the fracure element subregion.

```
  <Tasks>
    <PackCollection
      name="pressureCollection"
      objectPath="ElementRegions/Fracture/FractureSubRegion"
      fieldName="pressure"/>

    <PackCollection
      name="apertureCollection"
      objectPath="ElementRegions/Fracture/FractureSubRegion"
      fieldName="elementAperture"/>

    <PackCollection
      name="hydraulicApertureCollection"
      objectPath="ElementRegions/Fracture/FractureSubRegion"
      fieldName="hydraulicAperture"/>

    <PackCollection
      name="areaCollection"
      objectPath="ElementRegions/Fracture/FractureSubRegion"
      fieldName="elementArea"/>

    <!-- Collect aperture, pressure at the source for curve checks -->
    <PackCollection
      name="sourcePressureCollection"
      objectPath="ElementRegions/Fracture/FractureSubRegion"
      fieldName="pressure"
      setNames="{ source }"/>

    <PackCollection
      name="sourceHydraulicApertureCollection"
      objectPath="ElementRegions/Fracture/FractureSubRegion"
```

(continues on next page)

(continued from previous page)

```
      fieldName="hydraulicAperture"
      setNames="{ source }"/>
  </Tasks>
```

These tasks are triggered using the `Event` manager with a `PeriodicEvent` defined for the recurring tasks. GEOS writes one file named after the string defined in the `filename` keyword and formatted as a HDF5 file (`kgdToughnessDominated_output.hdf5`). This TimeHistory file contains the collected time history information from specified time history collector. This file includes datasets for the simulation time, fluid pressure, element aperture, hydraulic aperture and element area for the propagating hydraulic fracture. A Python script is prepared to read and query any specified subset of the time history data for verification and visualization.

The parameters used in the simulation are summarized in the following table.

Symbol	Parameter	Units	Value
Q_0	Injection rate	[m^3/s]	10^{-4}
E	Young's modulus	[GPa]	30
ν	Poisson's ratio	[-]	0.25
μ	Fluid viscosity	[Pa.s]	10^{-6}
K_{Ic}	Rock toughness	[MPa.m$^{1/2}$]	1

Inspecting results

Fracture propagation during the fluid injection period is shown in the figure below.

First, by running the query script

```
python ./hydrofractureQueries.py kgdToughnessDominated
```

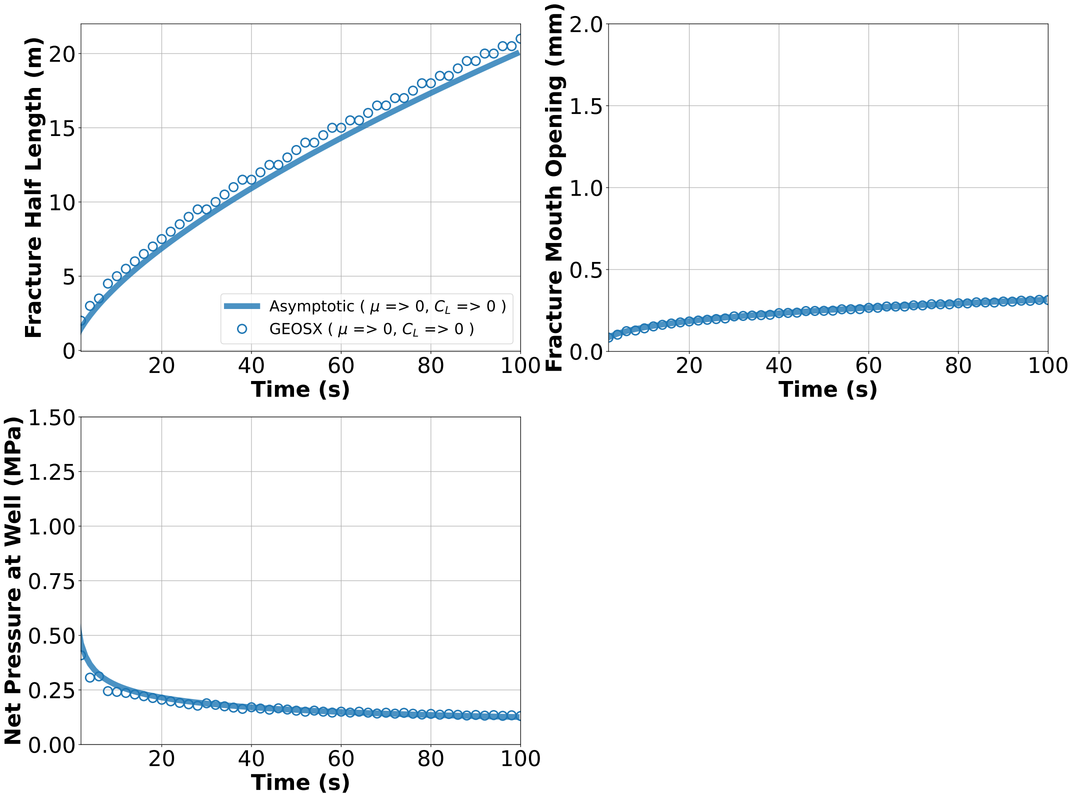
the HDF5 output is postprocessed and temporal evolution of fracture characterisctics (fluid pressure and fracture width at fluid inlet and fracure half length) are saved into a txt file `model-results.txt`, which can be used for verification and visualization:

```
[['      time', '  pressure', '  aperture', '    length']]
         2 4.086e+05 8.425e-05         2
         4 3.063e+05 0.0001021         3
         6 3.121e+05 0.0001238       3.5
         8 2.446e+05 0.0001277       4.5
        10 2.411e+05 0.0001409         5
```

Note: GEOS python tools `geosx_xml_tools` should be installed to run the query script (See *Python Tools Setup* for details).

A good agreement between GEOS results and analytical solutions is shown in the comparison below, which is generated using the visualization script:

```
python ./kgdToughnessDominatedFigure.py
```


Fracture Half Length (m)
0
5
10
15
20
20
40
60
80
100
Time (s)
Asymptotic (μ => 0, C_L => 0)
GEOSX (μ => 0, C_L => 0)
Fracture Mouth Opening (mm)
0.0
0.5
1.0
1.5
2.0
20
40
60
80
100
Time (s)
Net Pressure at Well (MPa)
0.00
0.25
0.50
0.75
1.00
1.25
1.50
20
40
60
80
100
Time (s)

To go further

Feedback on this example

This concludes the toughness dominated KGD example. For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

Viscosity dominated KGD hydraulic fracture

Description of the case

The KGD problem addresses a single plane strain fracture growing in an infinite elastic domain. Basic assumptions and characteristic shape for this example is similar to those of another case (*Viscosity dominated KGD hydraulic fracture*) except that the viscosity dominated regime is now considered. In this regime, more work is spent to move the fracturing fluid than to split the intact rock. In this test, slickwater with a constant viscosity of 1 cp is chosen as fracturing fluid, whose compressibility is neglected. To make the case identical to the viscosity dominated asymptotic solution, an ultra-low rock toughness K_{Ic} is defined and fracture is assumed to be always propagating following fluid front. Asymptotic solutions of the fracture length ℓ, the net pressure p_0 and the fracture aperture w_0 at the injection point for the KGD fracture with a viscosity dominated regime are provided by (Adachi and Detournay, 2002):

$$\begin{aligned} \ell &= 0.6152(\frac{E_p Q_0^3}{12\mu})^{1/6} t^{2/3} \\ w_0^2 &= 2.1(\frac{12\mu Q_0}{E_p})^{1/2} \ell \\ w_0 p_0 &= 0.62(12\mu Q_0 E_p)^{1/2} \end{aligned}$$

where the plane modulus E_p is defined by

$$E_p = \frac{E}{1 - \nu^2}$$

and the term X is given as:

$$X = \frac{256}{3\pi^2} \frac{K_{Ic}^4}{\mu Q_0 {E_p}^3}$$

Input file

The input xml files for this test case are located at:

```
inputFiles/hydraulicFracturing/kgdViscosityDominated_base.xml
```

and

```
inputFiles/hydraulicFracturing/kgdViscosityDominated_benchmark.xml
```

The corresponding integrated test with coarser mesh and smaller injection duration is also prepared:

```
inputFiles/hydraulicFracturing/kgdViscosityDominated_smoke.xml
```

Python scripts for post-processing and visualizing the simulation results are also prepared:

```
inputFiles/hydraulicFracturing/scripts/hydrofractureQueries.py
```

```
inputFiles/hydraulicFracturing/scripts/hydrofractureFigure.py
```

Fluid rheology and rock toughness are defined in the xml blocks below. Please note that setting an absolute zero value for the rock toughness could lead to instability issue. Therefore, a low value of K_{Ic} is used in this example.

```
    <SurfaceGenerator
      name="SurfaceGen"
      targetRegions="{ Domain }"
      nodeBasedSIF="1"
      rockToughness="1e4"
      mpiCommOrder="1"/>
```

```
    <CompressibleSinglePhaseFluid
      name="water"
      defaultDensity="1000"
      defaultViscosity="1.0e-3"
      referencePressure="0.0"
      compressibility="5e-10"
      referenceViscosity="1.0e-3"
      viscosibility="0.0"/>
```

First, by running the query script

```
python ./hydrofractureQueries.py kgdViscosityDominated
```

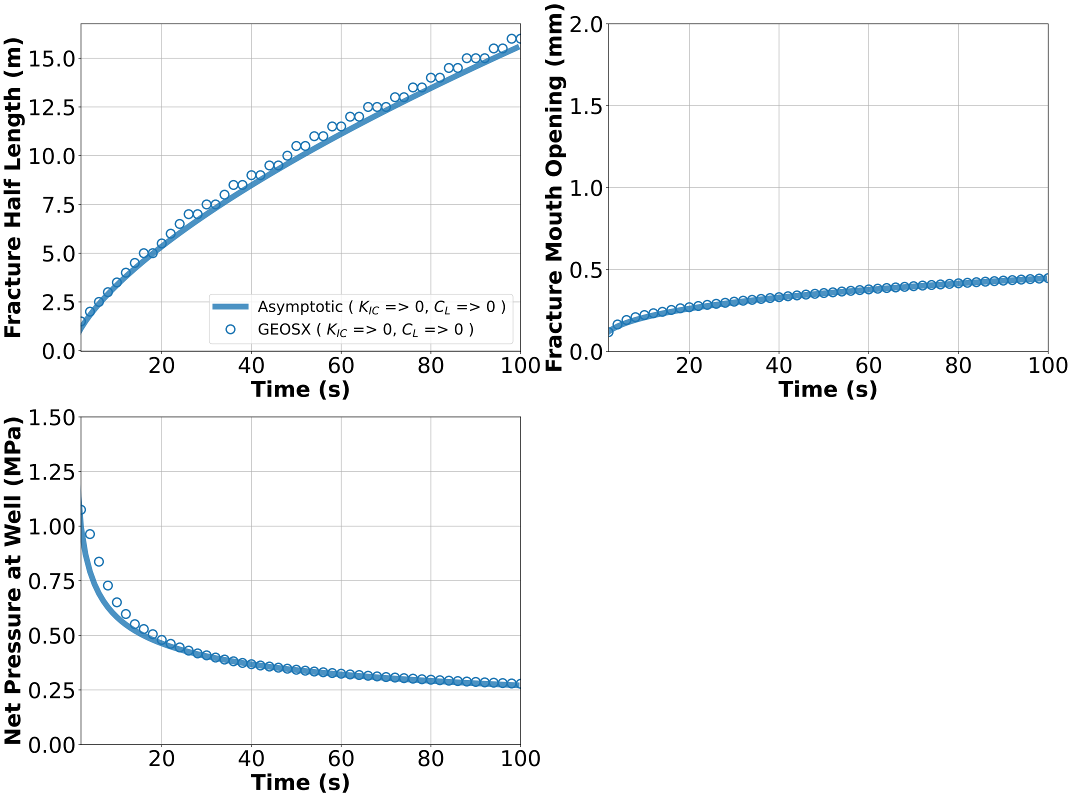
the HDF5 output is postprocessed and temporal evolution of fracture characterisctics (fluid pressure and fracture width at fluid inlet and fracure half length) are saved into a txt file `model-results.txt`, which can be used for verification and visualization:

```
[['      time', '  pressure', '  aperture', '    length']]
         2 1.075e+06 0.0001176       1.5
         4 9.636e+05 0.0001645         2
         6 8.372e+05 0.0001917       2.5
         8  7.28e+05  0.000209         3
        10 6.512e+05  0.000222       3.5
```

Note: GEOS python tools `geosx_xml_tools` should be installed to run the query script (See *Python Tools Setup* for details).

A good agreement between GEOS results and analytical solutions is shown in the comparison below, which is generated using the visualization script:

```
python ./kgdViscosityDominatedFigure.py
```


Fracture Half Length (m)
0.0
2.5
5.0
7.5
10.0
12.5
15.0
20
40
60
80
100
Time (s)
Asymptotic (K_{IC} => 0, C_L => 0)
GEOSX (K_{IC} => 0, C_L => 0)
Fracture Mouth Opening (mm)
0.0
0.5
1.0
1.5
2.0
20
40
60
80
100
Time (s)
Net Pressure at Well (MPa)
0.00
0.25
0.50
0.75
1.00
1.25
1.50
20
40
60
80
100
Time (s)

To go further

Feedback on this example

This concludes the viscosity dominated KGD example. For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

Validating KGD Hydraulic Fracture with Experiment

Context

In this example, we use GEOS to model a planar hydraulic fracture propagating in a finite domain subject to traction-free external boundaries. Contrary to the classic KGD problems, we do not assume an infinite rock domain. Existing analytical solutions cannot model fracture behavior in this scenario, so this problem is solved using the hydrofracture solver in GEOS. We validate the simulation results against a benchmark experiment (Rubin, 1983).

Input file

This example uses no external input files. Everything we need is contained within two GEOS input files:

```
inputFiles/hydraulicFracturing/kgdValidation_base.xml
```

```
inputFiles/hydraulicFracturing/kgdValidation_benchmark.xml
```

Python scripts for post-processing and visualizing the simulation results are also prepared:

```
src/docs/sphinx/advancedExamples/validationStudies/hydraulicFracture/kgdValidation/
↪kgdValidationQueries.py
```

```
src/docs/sphinx/advancedExamples/validationStudies/hydraulicFracture/kgdValidation/
↪kgdValidationFigure.py
```

Description of the case

We simulate a hydraulic fracturing experiment within a finite domain made of three layers of polymethylmethacrylate (PMMA). As shown below, we inject viscous fluid to create a single planar fracture in the middle layer. The target layer is bonded weakly to the adjacent layers, so a vertical fracture develops inside the middle layer. Four pressure gages are placed to monitor wellbore pressure (gage 56) and fluid pressure along the fracture length (gage 57, 58, and 59). A linear variable differential transducer (LVDT) measures the fracture aperture at 28.5 mm away from the wellbore. Images are taken at regular time intervals to show the temporal evolution of the fracture extent. All experimental measurements for the time history of pressure, aperture, and length are reported in Rubin (1983). We use GEOS to reproduce the conditions of this test, including material properties and pumping parameters. In the experiment, the upper and lower layers are used only to restrict the fracture height growth, they are therefore not simulated in GEOS but are present as boundary conditions. Given the vertical plane of symmetry, only half of the middle layer is modeled. For verification purposes, a plane strain deformation and zero fluid leak-off are considered in the numerical model.

In this example, we solve the hydraulic fracturing problem with the `hydrofrac` solver to obtain the temporal solution of the fracture characteristics (length, aperture and pressure). These modeling predictions are compared with the corresponding experimental results (Rubin, 1983).

For this example, we focus on the `Mesh`, the `Constitutive`, and the `FieldSpecifications` tags.

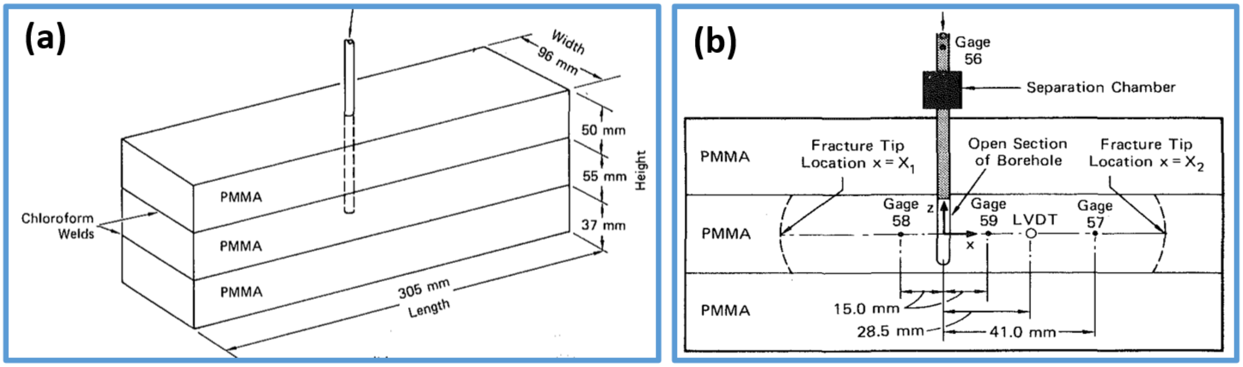


Fig. 1.27: Sketch of the problem

Mesh

The following figure shows the mesh used in this problem.

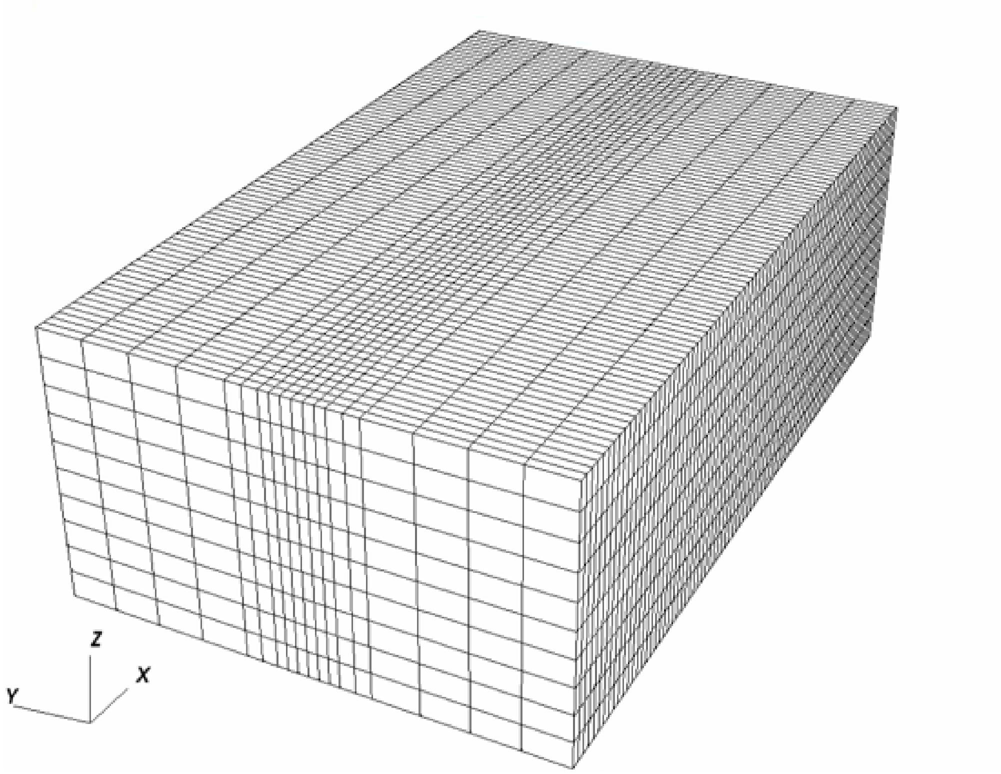


Fig. 1.28: Generated mesh

We use the internal mesh generator to create a computational domain ($0.1525\,m \times 0.096\,m \times 0.055\,m$), as parametrized in the `InternalMesh` XML tag. The structured mesh contains 80 x 18 x 10 eight-node brick elements in the x, y, and z directions respectively. Such eight-node hexahedral elements are defined as `C3D8` elementTypes, and their collection forms a mesh with one group of cell blocks named here `cb1`. Along the y-axis, refinement is performed for the elements in the vicinity of the fracture plane.

```
    <InternalMesh
      name="mesh1"
      elementTypes="{C3D8}"
      xCoords="{ 0, 0.1525 }"
      yCoords="{ -0.048, -0.012, -0.006, 0.006, 0.012, 0.048 }"
      zCoords="{ 0.037, 0.092 }"
      nx="{ 80 }"
      ny="{ 4, 2, 6, 2, 4 }"
      nz="{ 10 }"
      cellBlockNames="{cb1}"/>
```

The fracture plane is defined by a nodeset occupying a small region within the computation domain, where the fracture tends to open and propagate upon fluid injection:

```
    <Box
      name="core"
      xMin="{ -0.1, -0.001, 0.036 }"
      xMax="{  0.2, 0.001, 0.093 }"/>
```

Solid mechanics solver

GEOS is a multi-physics platform. Different combinations of physics solvers available in the code can be applied in different regions of the domain and be functional at different stages of the simulation. The `Solvers` tag in the XML file is used to list and parameterize these solvers.

Three elementary solvers are combined in the solver `Hydrofracture` to model the coupling between fluid flow within the fracture, rock deformation, fracture deformation and propagation:

```
    <Hydrofracture
      name="hydrofracture"
      solidSolverName="lagsolve"
      flowSolverName="SinglePhaseFlow"
      surfaceGeneratorName="SurfaceGen"
      logLevel="1"
      targetRegions="{ Fracture }"
      contactRelationName="fractureContact"
      maxNumResolves="2">
      <NonlinearSolverParameters
        newtonTol="1.0e-5"
        newtonMaxIter="20"
        lineSearchMaxCuts="3"/>
      <LinearSolverParameters
        directParallel="0"/>
    </Hydrofracture>
```

- Rock and fracture deformation are modeled by the solid mechanics solver `SolidMechanicsLagrangianSSLE`. In this solver, we define `targetRegions` that includes both the continuum region and the fracture region. The name of the contact constitutive behavior is specified in this solver by the `contactRelationName`.

```
    <SolidMechanicsLagrangianSSLE
      name="lagsolve"
      timeIntegrationOption="QuasiStatic"
```

(continues on next page)

(continued from previous page)

```
    discretization="FE1"
    targetRegions="{ Domain, Fracture }"
    contactRelationName="fractureContact"/>
```

- The single-phase fluid flow inside the fracture is solved by the finite volume method in the solver `SinglePhaseFVM`.

```
  <SinglePhaseFVM
    name="SinglePhaseFlow"
    discretization="singlePhaseTPFA"
    targetRegions="{ Fracture }"/>
```

- The solver `SurfaceGenerator` defines the fracture region and rock toughness. With `nodeBasedSIF="0"`, edge-based Stress Intensity Factor (SIF) calculation is chosen for the fracture propagation criterion.

```
  <SurfaceGenerator
    name="SurfaceGen"
    logLevel="1"
    targetRegions="{ Domain }"
    nodeBasedSIF="0"
    rockToughness="1.2e6"
    mpiCommOrder="1"/>
```

Constitutive laws

For this problem, a homogeneous and isotropic domain with one solid material is assumed, and its mechanical properties and associated fluid rheology are specified in the `Constitutive` section. `ElasticIsotropic` model is used to describe the mechanical behavior of `rock`, when subjected to fluid injection. The single-phase fluid model `CompressibleSinglePhaseFluid` is selected to simulate the response of `water` upon fracture propagation.

```
<Constitutive>
  <CompressibleSinglePhaseFluid
    name="water"
    defaultDensity="1000"
    defaultViscosity="97.7"
    referencePressure="0.0"
    compressibility="5e-12"
    referenceViscosity="97.7"
    viscosibility="0.0"/>

  <ElasticIsotropic
    name="rock"
    defaultDensity="2700"
    defaultBulkModulus="4.110276e9"
    defaultShearModulus="1.19971e9"/>

  <CompressibleSolidParallelPlatesPermeability
    name="fractureFilling"
    solidModelName="nullSolid"
    porosityModelName="fracturePorosity"
    permeabilityModelName="fracturePerm"/>
```

(continues on next page)

(continued from previous page)

```

  <NullModel
    name="nullSolid"/>

  <PressurePorosity
    name="fracturePorosity"
    defaultReferencePorosity="1.00"
    referencePressure="0.0"
    compressibility="0.0"/>

  <ParallelPlatesPermeability
    name="fracturePerm"/>

  <FrictionlessContact
    name="fractureContact"
    penaltyStiffness="1.0"
    apertureTableName="apertureTable" />
</Constitutive>
```

All constitutive parameters such as density, viscosity, bulk modulus, and shear modulus are specified in the International System of Units.

Time history function

In the Tasks section, PackCollection tasks are defined to collect time history information from fields. Either the entire field or specified named sets of indices in the field can be collected. In this example, pressureCollection, apertureCollection, hydraulicApertureCollection and areaCollection are specified to output the time history of fracture characterisctics (pressure, width and area). objectPath="ElementRegions/Fracture/FractureSubRegion" indicates that these PackCollection tasks are applied to the fracure element subregion.

```
<Tasks>
  <PackCollection
    name="pressureCollection"
    objectPath="ElementRegions/Fracture/FractureSubRegion"
    fieldName="pressure"/>

  <PackCollection
    name="apertureCollection"
    objectPath="ElementRegions/Fracture/FractureSubRegion"
    fieldName="elementAperture"/>

  <PackCollection
    name="hydraulicApertureCollection"
    objectPath="ElementRegions/Fracture/FractureSubRegion"
    fieldName="hydraulicAperture"/>

  <PackCollection
    name="areaCollection"
    objectPath="ElementRegions/Fracture/FractureSubRegion"
    fieldName="elementArea"/>
</Tasks>
```

These tasks are triggered using the `Event` manager with a `PeriodicEvent` defined for the recurring tasks. GEOS writes one file named after the string defined in the `filename` keyword and formatted as a HDF5 file (`KGD_validation_output.hdf5`). This TimeHistory file contains the collected time history information from specified time history collector. This file includes datasets for the simulation time, fluid pressure, element aperture, hydraulic aperture and element area for the propagating hydraulic fracture. A Python script is prepared to read and query any specified subset of the time history data for verification and visualization.

Initial and boundary conditions

The next step is to specify fields, including:

- The initial values: the `waterDensity`, `separableFace` and the `ruptureState` of the propagating fracture have to be initialized,
- The boundary conditions: fluid injection rates and the constraints of the outer boundaries have to be set.

In this example, a mass injection rate `SourceFlux` (`scale="-0.0000366"`) is applied at the surfaces of the initial fracture. Only half of the injection rate is defined in this boundary condition because only a half-wing of the fracture is modeled (the problem is symmetric). The value given for `scale` is $Q_0 \rho_f / 2$ (not $Q_0/2$). The lateral surfaces (`xpos`, `ypos` and `yneg`) are traction free. The remaining parts of the outer boundaries are subjected to roller constraints. These boundary conditions are set up through the `FieldSpecifications` section.

```
<FieldSpecifications>
  <FieldSpecification
    name="waterDensity"
    initialCondition="1"
    setNames="{ fracture }"
    objectPath="ElementRegions"
    fieldName="water_density"
    scale="1000"/>

  <FieldSpecification
    name="separableFace"
    initialCondition="1"
    setNames="{ core }"
    objectPath="faceManager"
    fieldName="isFaceSeparable"
    scale="1"/>

  <FieldSpecification
    name="frac"
    initialCondition="1"
    setNames="{ fracture }"
    objectPath="faceManager"
    fieldName="ruptureState"
    scale="1"/>

  <FieldSpecification
    name="zconstraint"
    objectPath="nodeManager"
    fieldName="totalDisplacement"
    component="2"
    scale="0.0"
    setNames="{ zneg, zpos }"/>
```

(continues on next page)

(continued from previous page)

```
    <FieldSpecification
      name="xConstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="0"
      scale="0.0"
      setNames="{ xneg }"/>

    <SourceFlux
      name="sourceTerm"
      objectPath="ElementRegions/Fracture"
      scale="-0.0000366"
      setNames="{ source }"/>
  </FieldSpecifications>
```

Note that the applied traction has a negative value, due to the negative sign convention for compressive stresses in GEOS.

The parameters used in the simulation are summarized in the following table.

Symbol	Parameter	Unit	Value
K	Bulk Modulus	[GPa]	4.11
G	Shear Modulus	[GPa]	1.2
K_{Ic}	Rock Toughness	[MPa.m$^{1/2}$]	1.2
μ	Fluid Viscosity	[Pa.s]	97.7
Q_0	Injection Rate	[m^3/s]	73.2x10^{-9}
t_{inj}	Injection Time	[s]	100
h_f	Fracture Height	[mm]	55

Inspecting results

The following figure shows the distribution of σ_{yy} at $t = 100s$ within the computational domain..

By running the query script `kgdValidationQueries.py`, the HDF5 output is postprocessed and temporal evolution of fracture characterisctics (fluid pressure and fracture width at fluid inlet and fracure half length) are saved into a txt file `model-results.txt`, which can be used for verification and visualization:

```
[['      time', ' wpressure', '58pressure', '57pressure', ' Laperture', '      area']]
          0          0          0          0          0  0.0001048
        0.1  1.515e+07          0          0          0  0.0003145
        0.2  1.451e+07          0          0          0  0.0003774
        0.3  1.349e+07          0          0          0  0.0004194
        0.4  1.183e+07          0          0          0  0.0005662
        0.5  1.125e+07          0          0          0  0.0005662
```

Note: GEOS python tools `geosx_xml_tools` should be installed to run the query script (See *Python Tools Setup* for details).

The figure below shows simulation results of the fracture extent at the end of the injection, which is generated using the visualization script `kgdValidationFigure.py`. The temporal evolution of the fracture characteristics (length, aperture and pressure) from the GEOS simulation are extracted and compared with the experimental data gathered at

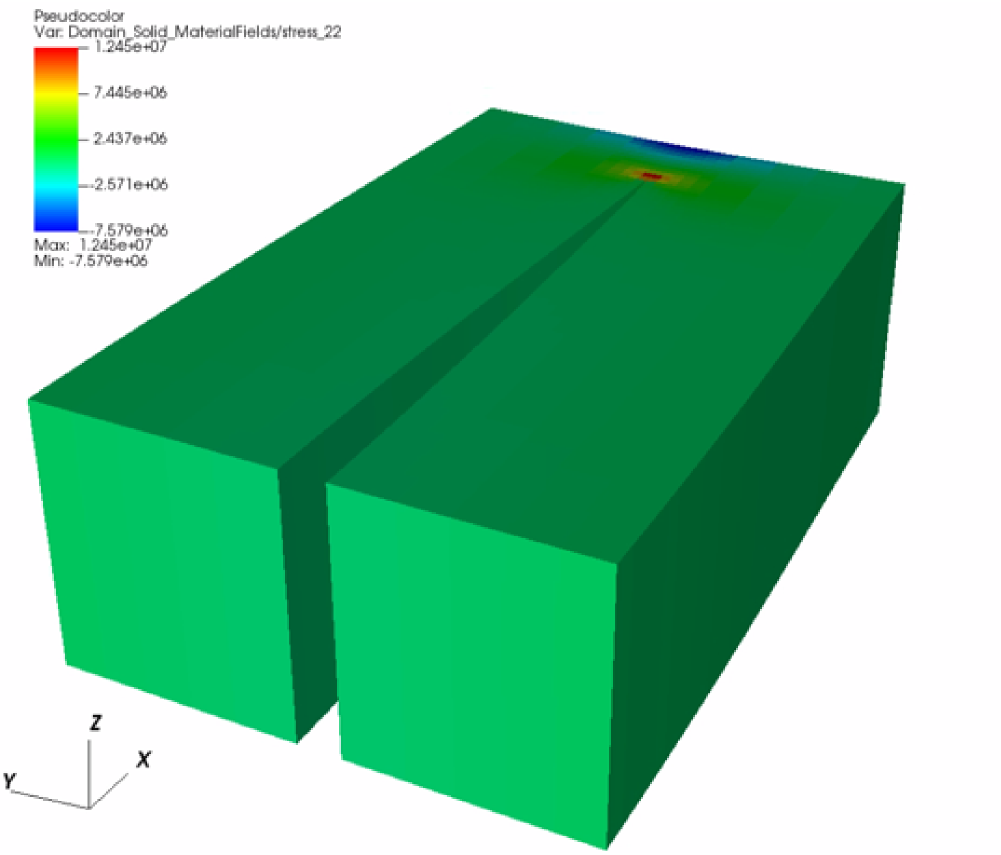
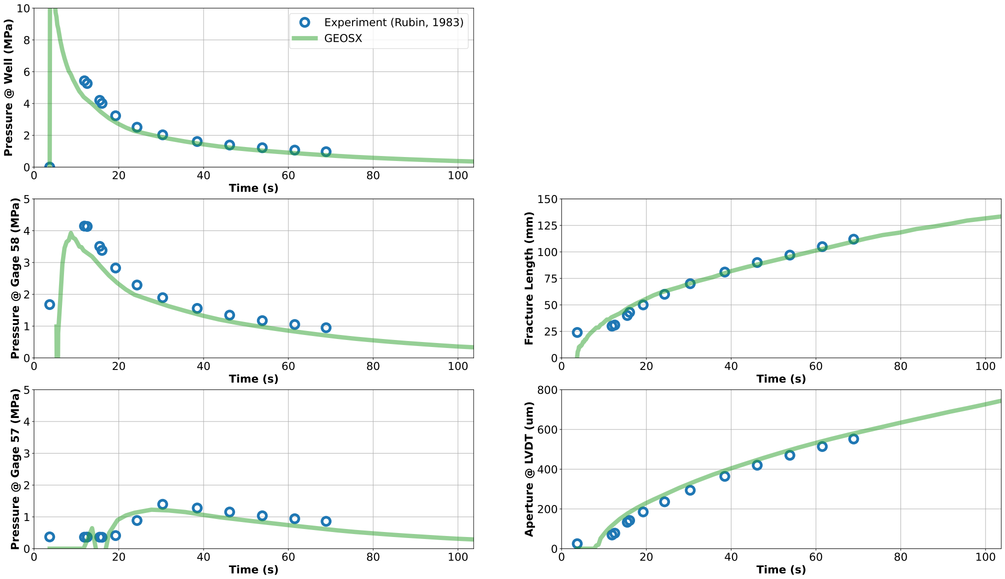


Fig. 1.29: Simulation result of σ_{xx} at $t = 100s$

specific locations. As observed, the time history plots of the modelling predictions (green curves) for the pressure at three gage locations, the fracture length, and the fracture aperture at LVDT location correlate well with the experimental data (blue circles).



To go further

Feedback on this example

For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

Toughness-Storage-Dominated Penny Shaped Hydraulic Fracture

Context

In this example, we simulate the growth of a radial hydraulic fracture in toughness-storage-dominated regime, a classic benchmark in hydraulic fracturing (Settgast et al., 2016). The developed fracture is characterized as a planar fracture with an elliptical cross-section perpendicular to the fracture plane and a circular fracture tip. This problem is solved using the hydrofracture solver in GEOS. The modeling predictions on the temporal evolutions of the fracture characteristics (length, aperture, and pressure) are verified against the analytical solutions (Savitski and Detournay, 2002).

Input file

This example uses no external input files. Everything we need is contained within two GEOS input files:

```
inputFiles/hydraulicFracturing/pennyShapedToughnessDominated_base.xml
```

```
inputFiles/hydraulicFracturing/pennyShapedToughnessDominated_benchmark.xml
```

Python scripts for post-processing and visualizing the simulation results are also prepared:

```
inputFiles/hydraulicFracturing/scripts/hydrofractureQueries.py
```

```
inputFiles/hydraulicFracturing/scripts/hydrofractureFigure.py
```

Description of the case

We model a radial fracture emerging from a point source and forming a perfect circular shape in an infinite, isotropic, and homogenous elastic domain. As with the KGD problem, we simplify the model to a radial fracture in a toughness-storage-dominated propagation regime. For toughness-dominated fractures, more work is spent on splitting the intact rock than on moving the fracturing fluid. Storage-dominated propagation occurs if most of the fracturing fluid is contained within the propagating fracture. In this analysis, incompressible fluid with ultra-low viscosity ($0.001cp$) and medium rock toughness ($3.0MPa\sqrt{m}$) are specified. In addition, an impermeable fracture surface is assumed to eliminate the effect of fluid leak-off. This way, the GEOS simulations represent cases within the valid range of the toughness-storage-dominated assumptions.

In this model, the injected fluid within the fracture follows the lubrication equation resulting from mass conservation and Poiseuille's law. The fracture propagates by creating new surfaces if the stress intensity factor exceeds the local rock toughness K_{IC}. By symmetry, the simulation is reduced to a quarter-scale to save computational cost. For verification purposes, a plane strain deformation is considered in the numerical model.

In this example, we set up and solve a hydraulic fracture model to obtain the temporal solutions of the fracture radius R, the net pressure p_0 and the fracture aperture w_0 at the injection point for the penny-shaped fracture developed in this toughness-storage-dominated regime. The numerical predictions from GEOS are then compared with the corresponding asymptotic solutions (Savitski and Detournay, 2002):

$$R(t) = 0.8546(\frac{E_p^2 Q_0^2 t^2}{K_p^2})^{1/5}$$

$$w_0(t) = 0.6537(\frac{K_p^4 Q_0 t}{E_p^4})^{1/5}$$

$$p_0(t) = 0.3004(\frac{K_p^6}{E_p Q_0 t})^{1/5}$$

where the plane modulus E_p is related to Young's modulus E and Poisson's ratio ν:

$$E_p = \frac{E}{1-\nu^2}$$

The term K_p is proportional to the rock toughness K_{IC}:

$$K_p = \frac{8}{\sqrt{2\pi}} K_{IC}$$

For this example, we focus on the `Mesh`, the `Constitutive`, and the `FieldSpecifications` tags.

Mesh

The following figure shows the mesh used in this problem.

We use the internal mesh generator to create a computational domain ($400\,m\ \times 400\,m\ \times 800\,m$), as parametrized in the `InternalMesh` XML tag. The structured mesh contains 80 x 80 x 60 eight-node brick elements in the x, y, and z directions respectively. Such eight-node hexahedral elements are defined as `C3D8` elementTypes, and their collection forms a mesh with one group of cell blocks named here `cb1`. Local refinement is performed for the elements in the vicinity of the fracture plane.

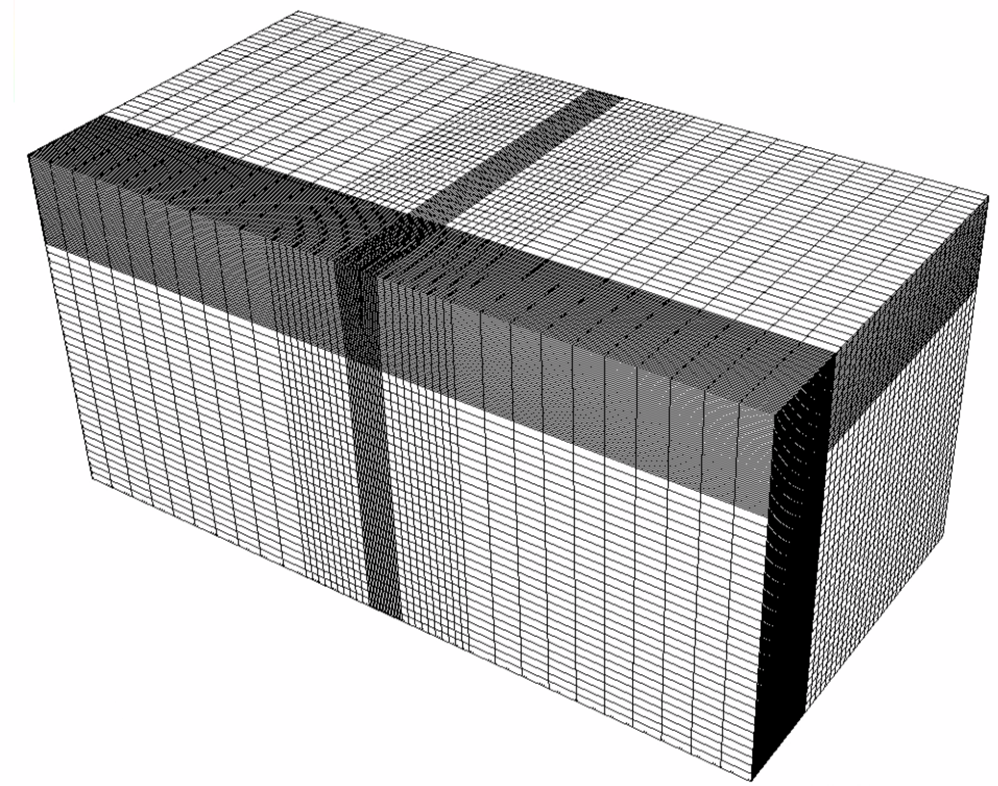


Fig. 1.30: Generated mesh

Note that the domain size in the direction perpendicular to the fracture plane, i.e. z-axis, must be at least ten times of the final fracture radius to minimize possible boundary effects.

```
<Mesh>
  <InternalMesh
    name="mesh1"
    elementTypes="{ C3D8 }"
    xCoords="{ 0, 100, 200, 400 }"
    yCoords="{ 0, 100, 200, 400 }"
    zCoords="{ -400, -100, -20, 20, 100, 400 }"
    nx="{ 50, 10, 20 }"
    ny="{ 50, 10, 20 }"
    nz="{ 10, 10, 20, 10, 10 }"
    cellBlockNames="{ cb1 }"/>
</Mesh>
```

The fracture plane is defined by a nodeset occupying a small region within the computation domain, where the fracture tends to open and propagate upon fluid injection:

```
  <Box
    name="core"
    xMin="{ -500.1, -500.1, -0.1 }"
    xMax="{ 500.1, 500.1, 0.1 }"/>
```

Solid mechanics solver

GEOS is a multi-physics platform. Different combinations of physics solvers available in the code can be applied in different regions of the domain and be functional at different stages of the simulation. The `Solvers` tag in the XML file is used to list and parameterize these solvers.

Three elementary solvers are combined in the solver `Hydrofracture` to model the coupling between fluid flow within the fracture, rock deformation, fracture deformation and propagation:

```
<Hydrofracture
  name="hydrofracture"
  solidSolverName="lagsolve"
  flowSolverName="SinglePhaseFlow"
  surfaceGeneratorName="SurfaceGen"
  logLevel="1"
  targetRegions="{ Fracture }"
  contactRelationName="fractureContact"
  maxNumResolves="5"
  initialDt="0.1">
  <NonlinearSolverParameters
    newtonTol="1.0e-4"
    newtonMaxIter="50"
    logLevel="1"/>
  <LinearSolverParameters
    solverType="gmres"
    preconditionerType="mgr"
    logLevel="1"
    krylovAdaptiveTol="1"/>
</Hydrofracture>
```

- Rock and fracture deformation are modeled by the solid mechanics solver `SolidMechanicsLagrangianSSLE`. In this solver, we define `targetRegions` that includes both the continuum region and the fracture region. The name of the contact constitutive behavior is specified in this solver by the `contactRelationName`.

```
<SolidMechanicsLagrangianSSLE
  name="lagsolve"
  timeIntegrationOption="QuasiStatic"
  logLevel="1"
  discretization="FE1"
  targetRegions="{ Domain, Fracture }"
  contactRelationName="fractureContact">
  <NonlinearSolverParameters
    newtonTol="1.0e-6"/>
  <LinearSolverParameters
    solverType="gmres"
    krylovTol="1.0e-10"/>
</SolidMechanicsLagrangianSSLE>
```

- The single-phase fluid flow inside the fracture is solved by the finite volume method in the solver `SinglePhaseFVM`.

```
<SinglePhaseFVM
  name="SinglePhaseFlow"
  logLevel="1"
```

(continues on next page)

(continued from previous page)

```
      discretization="singlePhaseTPFA"
      targetRegions="{ Fracture }">
      <NonlinearSolverParameters
        newtonTol="1.0e-5"
        newtonMaxIter="10"/>
      <LinearSolverParameters
        solverType="gmres"
        krylovTol="1.0e-12"/>
    </SinglePhaseFVM>
```

- The solver `SurfaceGenerator` defines the fracture region and rock toughness `rockToughness="3.0e6"`. With `nodeBasedSIF="1"`, a node-based Stress Intensity Factor (SIF) calculation is chosen for the fracture propagation criterion.

```
    <SurfaceGenerator
      name="SurfaceGen"
      targetRegions="{ Domain }"
      nodeBasedSIF="1"
      rockToughness="3.0e6"
      mpiCommOrder="1"/>
```

Constitutive laws

For this problem, a homogeneous and isotropic domain with one solid material is assumed. Its mechanical properties and associated fluid rheology are specified in the `Constitutive` section. `ElasticIsotropic` model is used to describe the mechanical behavior of `rock` when subjected to fluid injection. The single-phase fluid model `CompressibleSinglePhaseFluid` is selected to simulate the response of `water` upon fracture propagation.

```
  <Constitutive>
    <CompressibleSinglePhaseFluid
      name="water"
      defaultDensity="1000"
      defaultViscosity="1.0e-6"
      referencePressure="0.0"
      compressibility="5e-13"
      referenceViscosity="1.0e-6"
      viscosibility="0.0"/>

    <ElasticIsotropic
      name="rock"
      defaultDensity="2700"
      defaultBulkModulus="20.0e9"
      defaultShearModulus="12.0e9"/>

    <CompressibleSolidParallelPlatesPermeability
      name="fractureFilling"
      solidModelName="nullSolid"
      porosityModelName="fracturePorosity"
      permeabilityModelName="fracturePerm"/>

    <NullModel
```

(continues on next page)

(continued from previous page)

```
      name="nullSolid"/>

    <PressurePorosity
      name="fracturePorosity"
      defaultReferencePorosity="1.00"
      referencePressure="0.0"
      compressibility="0.0"/>

    <ParallelPlatesPermeability
      name="fracturePerm"/>

    <FrictionlessContact
      name="fractureContact"
      penaltyStiffness="1.0e0"
      apertureTableName="apertureTable"/>
  </Constitutive>
```

All constitutive parameters such as density, viscosity, bulk modulus, and shear modulus are specified in the International System of Units.

Time history function

In the Tasks section, PackCollection tasks are defined to collect time history information from fields. Either the entire field or specified named sets of indices in the field can be collected. In this example, pressureCollection, apertureCollection, hydraulicApertureCollection and areaCollection are specified to output the time history of fracture characterisctics (pressure, width and area). objectPath="ElementRegions/Fracture/FractureSubRegion" indicates that these PackCollection tasks are applied to the fracure element subregion.

```
  <Tasks>
    <PackCollection
      name="pressureCollection"
      objectPath="ElementRegions/Fracture/FractureSubRegion"
      fieldName="pressure"/>

    <PackCollection
      name="apertureCollection"
      objectPath="ElementRegions/Fracture/FractureSubRegion"
      fieldName="elementAperture"/>

    <PackCollection
      name="hydraulicApertureCollection"
      objectPath="ElementRegions/Fracture/FractureSubRegion"
      fieldName="hydraulicAperture"/>

    <PackCollection
      name="areaCollection"
      objectPath="ElementRegions/Fracture/FractureSubRegion"
      fieldName="elementArea"/>

    <!-- Collect aperture, pressure at the source for curve checks -->
    <PackCollection
```

(continues on next page)

(continued from previous page)

```
      name="sourcePressureCollection"
      objectPath="ElementRegions/Fracture/FractureSubRegion"
      fieldName="pressure"
      setNames="{ source }"/>

    <PackCollection
      name="sourceHydraulicApertureCollection"
      objectPath="ElementRegions/Fracture/FractureSubRegion"
      fieldName="hydraulicAperture"
      setNames="{ source }"/>
  </Tasks>
```

These tasks are triggered using the `Event` manager with a `PeriodicEvent` defined for the recurring tasks. GEOS writes one file named after the string defined in the `filename` keyword and formatted as a HDF5 file (`pennyShapedToughnessDominated_output.hdf5`). This TimeHistory file contains the collected time history information from specified time history collector. This file includes datasets for the simulation time, fluid pressure, element aperture, hydraulic aperture and element area for the propagating hydraulic fracture. A Python script is prepared to read and query any specified subset of the time history data for verification and visualization.

Initial and boundary conditions

The next step is to specify fields, including:

- The initial values: the `waterDensity`, `separableFace` and the `ruptureState` of the propagating fracture have to be initialized,
- The boundary conditions: fluid injection rates and the constraints of the outer boundaries have to be set.

In this example, a mass injection rate `SourceFlux` (`scale="-6.625"`) is applied at the surfaces of the initial fracture. Only one fourth of the total injection rate is defined in this boundary condition because only a quarter of the fracture is modeled (the problem is symmetric). The value given for `scale` is $Q_0 \rho_f / 4$ (not $Q_0/4$). All the outer boundaries are subject to roller constraints. These boundary conditions are set through the `FieldSpecifications` section.

```
  <FieldSpecifications>
    <FieldSpecification
      name="waterDensity"
      initialCondition="1"
      setNames="{ fracture }"
      objectPath="ElementRegions"
      fieldName="water_density"
      scale="1000"/>

    <FieldSpecification
      name="separableFace"
      initialCondition="1"
      setNames="{ core }"
      objectPath="faceManager"
      fieldName="isFaceSeparable"
      scale="1"/>

    <FieldSpecification
      name="frac"
      initialCondition="1"
```

(continues on next page)

(continued from previous page)

```
      setNames="{ fracture }"
      objectPath="faceManager"
      fieldName="ruptureState"
      scale="1"/>

    <FieldSpecification
      name="yconstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="1"
      scale="0.0"
      setNames="{ yneg, ypos }"/>

    <FieldSpecification
      name="zconstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="2"
      scale="0.0"
      setNames="{ zneg, zpos }"/>

    <FieldSpecification
      name="xconstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="0"
      scale="0.0"
      setNames="{ xneg, xpos }"/>

    <SourceFlux
      name="sourceTerm"
      objectPath="ElementRegions/Fracture"
      scale="-6.625"
      setNames="{ source }"/>
  </FieldSpecifications>
```

The parameters used in the simulation are summarized in the following table.

Symbol	Parameter	Unit	Value
K	Bulk Modulus	[GPa]	20.0
G	Shear Modulus	[GPa]	12.0
K_{IC}	Rock Toughness	[MPa.m$^{1/2}$]	3.0
μ	Fluid Viscosity	[Pa.s]	1.0x10^{-6}
Q_0	Injection Rate	[m^3/s]	0.0265
t_{inj}	Injection Time	[s]	400

Inspecting results

The following figure shows the distribution of σ_{zz} at $t = 400s$ within the computational domain..

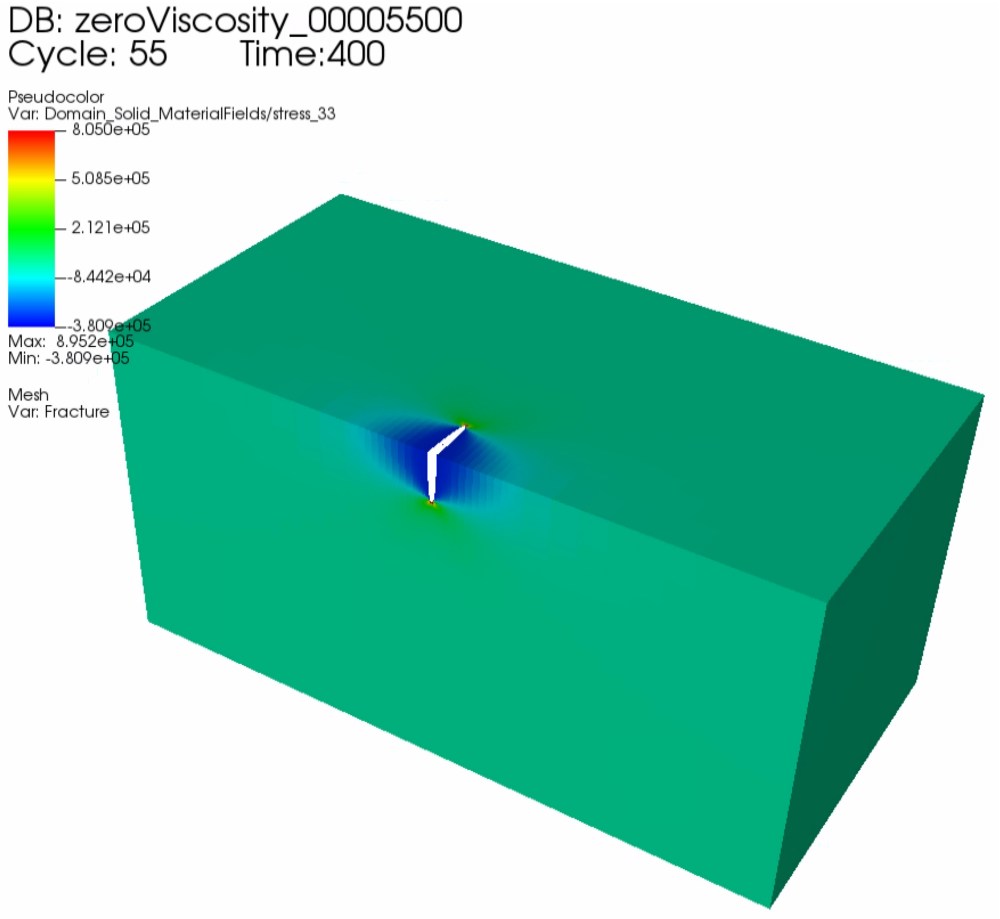


Fig. 1.31: Simulation result of σ_{zz} at $t = 400s$

First, by running the query script

```
python ./hydrofractureQueries.py pennyShapedToughnessDominated
```

the HDF5 output is postprocessed and temporal evolution of fracture characterisctics (fluid pressure and fracture width at fluid inlet and fracure radius) are saved into a txt file `model-results.txt`, which can be used for verification and visualization:

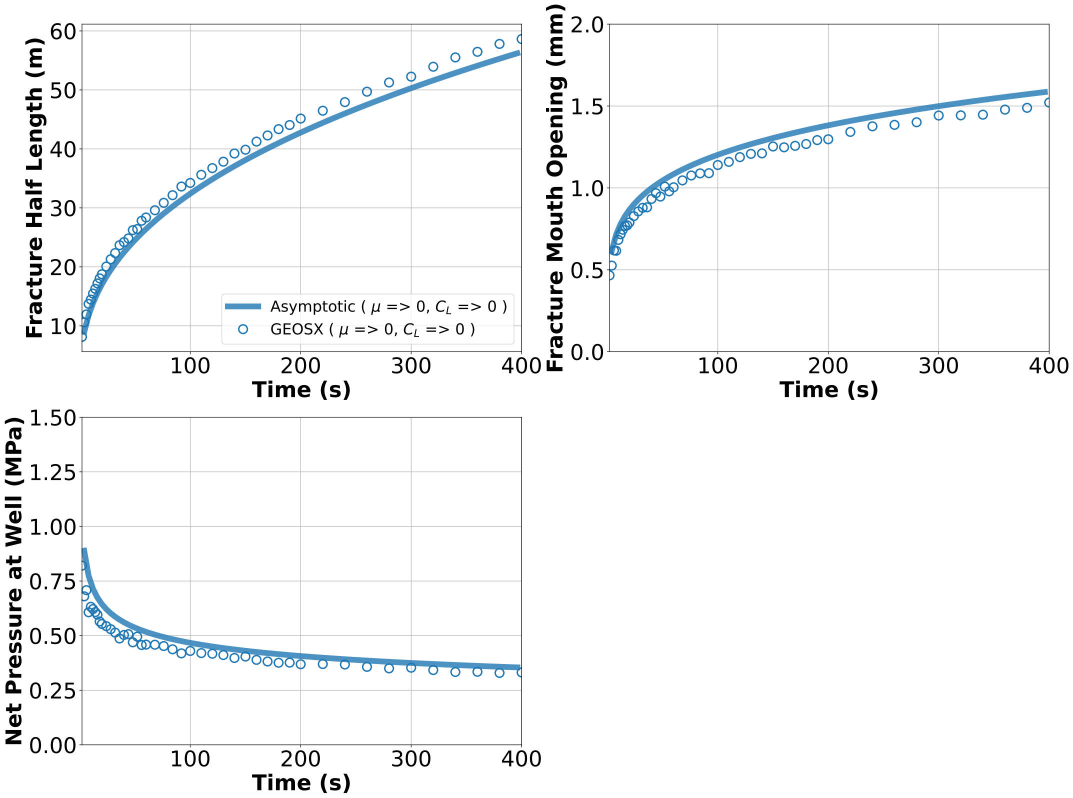
```
[['      time', '  pressure', '  aperture', '    length']]
         2 8.207e+05 0.0004661     8.137
         4 6.799e+05 0.0005258     10.59
         6 7.082e+05 0.0006183     11.94
         8  6.07e+05 0.0006163     13.73
        10  6.32e+05 0.0006827     14.45
```

Note: GEOS python tools `geosx_xml_tools` should be installed to run the query script (See *Python Tools Setup* for details).

Next, the figure below compares the asymptotic solutions (curves) and the GEOS simulation results (markers) for this analysis, which is generated using the visualization script:

```
python ./pennyShapedToughnessDominatedFigure.py
```

The time history plots of fracture radius, fracture aperture and fluid pressure at the point source match the asymptotic solutions, confirming the accuracy of GEOS simulations.



To go further

Feedback on this example

For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

Viscosity-Storage-Dominated Penny Shaped Hydraulic Fracture

Context

In this example, we simulate the propagation of a radial hydraulic fracture in viscosity-storage-dominated regime, another classic benchmark in hydraulic fracturing (Settgast et al., 2016). The fracture develops as a planar fracture with an elliptical cross-section perpendicular to the fracture plane and a circular fracture tip. Unlike the toughness-storage-dominated fractures, fluid frictional loss during the transport of viscous fracturing fluids governs the growth of viscosity-storage-dominated fractures. We solve this problem using the hydrofracture solver in GEOS. We simulate the change in length, aperture, and pressure of the fracture, and compare them against the corresponding analytical solutions (Savitski and Detournay, 2002).

Input file

This example uses no external input files. Everything we need is contained within two GEOS input files:

```
inputFiles/hydraulicFracturing/pennyShapedViscosityDominated_base.xml
```

```
inputFiles/hydraulicFracturing/pennyShapedViscosityDominated_benchmark.xml
```

Python scripts for post-processing and visualizing the simulation results are also prepared:

```
inputFiles/hydraulicFracturing/scripts/hydrofractureQueries.py
```

```
inputFiles/hydraulicFracturing/scripts/hydrofractureFigure.py
```

Description of the case

We model a radial fracture emerging from a point source and forming a perfect circular shape in an infinite, isotropic, and homogenous elastic domain. As with the viscosity-dominated KGD problem, we restrict the model to a radial fracture developed in a viscosity-storage-dominated propagation regime. For viscosity-dominated fractures, more energy is applied to move the fracturing fluid than to split the intact rock. If we neglect fluid leak-off, the storage-dominated propagation occurs from most of the injected fluid confined within the opened surfaces. We use a low rock toughness ($0.3MPa\sqrt{m}$), and the slickwater we inject has a constant viscosity value ($1.0cp$) and zero compressibility. In addition, we assume that the fracture surfaces are impermeable, thus eliminating fluid leak-off. With this configuration, our GEOS simulations meet the requirements of the viscosity-storage-dominated assumptions.

The fluid injected in the fracture follows the lubrication equation resulting from mass conservation and Poiseuille's law. The fracture propagates by creating new surfaces if the stress intensity factor exceeds the local rock toughness K_{IC}. By symmetry, the simulation is reduced to a quarter-scale to save computational cost. For verification purposes, a plane strain deformation is considered in the numerical model.

We set up and solve a hydraulic fracture model to obtain the evolution with time of the fracture radius R, the net pressure p_0 and the fracture aperture w_0 at the injection point for the penny-shaped fracture developed in viscosity-storage-dominated regime. Savitski and Detournay (2002) presented the corresponding asymptotic solutions, used here to validate the results of our GEOS simulations:

$$R(t) = 0.6955(\frac{E_p Q_0^3 t^4}{M_p})^{1/9}$$

$$w_0(t) = 1.1977(\frac{M_p^2 Q_0^3 t}{E_p^2})^{1/9}$$

$$p_0(\Pi, t) = \Pi_{mo}(\xi)(\frac{E_p^2 M_p}{t})^{1/3}$$

where the plane modulus E_p is related to Young's modulus E and Poisson's ratio ν:

$$E_p = \frac{E}{1 - \nu^2}$$

The term M_p is proportional to the fluid viscosity μ:

$$M_p = 12\mu$$

The viscosity scaling function Π_{mo} is given as:

$$\Pi_{mo}(\xi) = A_1[2.479 - \frac{2}{3(1-\xi)^{1/3}}] - B[\ln(\frac{\xi}{2}) + 1]$$

with $A_1 = 0.3581$, $B = 0.09269$, $c_1 = 0.6846$, $c_2 = 0.07098$, and $\xi = r/R(t)$ denoting a dimensionless radial coordinate along the fracture.

For this example, we focus on the `Mesh`, the `Constitutive`, and the `FieldSpecifications` tags.

Mesh

The following figure shows the mesh used in this problem.

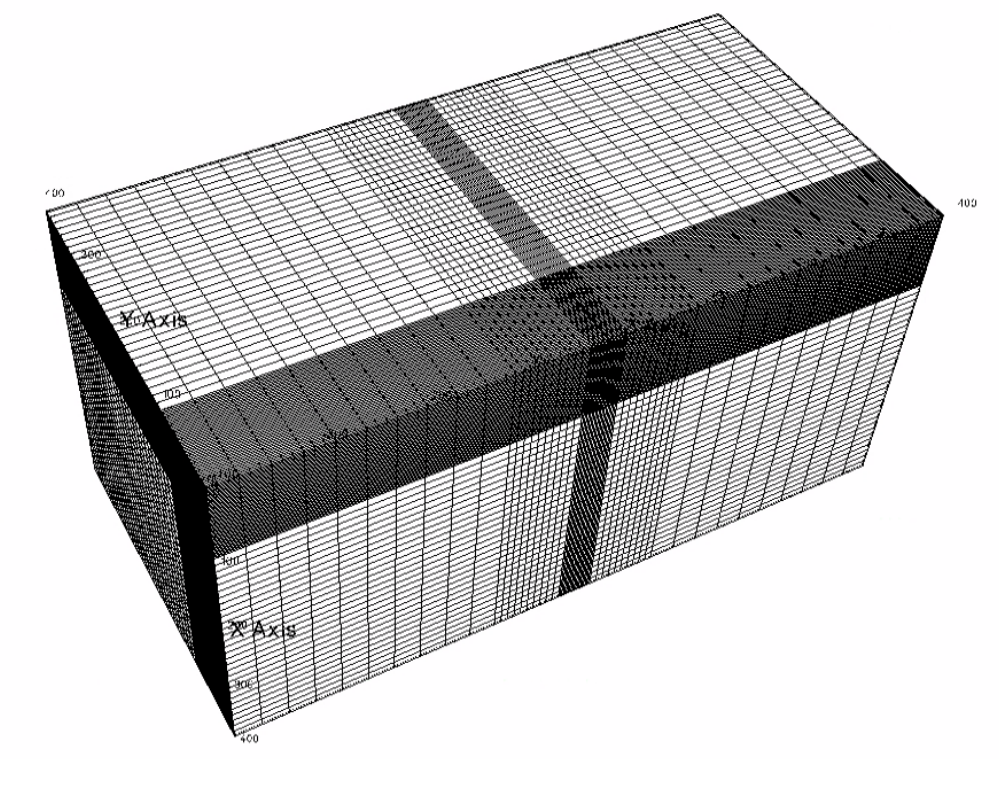


Fig. 1.32: Generated mesh

We use the internal mesh generator to create a computational domain ($400\,m \times 400\,m \times 800\,m$), as parametrized in the `InternalMesh` XML tag. The structured mesh contains 80 x 80 x 60 eight-node brick elements in the x, y, and z directions respectively. Such eight-node hexahedral elements are defined as `C3D8` elementTypes, and their collection forms a mesh with one group of cell blocks named here `cb1`. Local refinement is performed for the elements in the vicinity of the fracture plane.

Note that the domain size in the direction perpendicular to the fracture plane, i.e. z-axis, must be at least ten times of the final fracture radius to minimize possible boundary effects.

```
<Mesh>
  <InternalMesh
    name="mesh1"
    elementTypes="{ C3D8 }"
    xCoords="{ 0, 100, 200, 400 }"
    yCoords="{ 0, 100, 200, 400 }"
    zCoords="{ -400, -100, -20, 20, 100, 400 }"
    nx="{ 50, 10, 20 }"
    ny="{ 50, 10, 20 }"
    nz="{ 10, 10, 20, 10, 10 }"
    cellBlockNames="{ cb1 }"/>
</Mesh>
```

The fracture plane is defined by a nodeset occupying a small region within the computation domain, where the fracture tends to open and propagate upon fluid injection:

```
  <Box
    name="core"
    xMin="{ -500.1, -500.1, -0.1 }"
    xMax="{ 500.1, 500.1, 0.1 }"/>
```

Solid mechanics solver

GEOS is a multi-physics platform. Different combinations of physics solvers available in the code can be applied in different regions of the domain and be functional at different stages of the simulation. The `Solvers` tag in the XML file is used to list and parameterize these solvers.

Three elementary solvers are combined in the solver `Hydrofracture` to model the coupling between fluid flow within the fracture, rock deformation, fracture deformation and propagation:

```
  <Hydrofracture
    name="hydrofracture"
    solidSolverName="lagsolve"
    flowSolverName="SinglePhaseFlow"
    surfaceGeneratorName="SurfaceGen"
    logLevel="1"
    targetRegions="{ Fracture }"
    contactRelationName="fractureContact"
    maxNumResolves="1"
    initialDt="0.1">
    <NonlinearSolverParameters
      newtonTol="1.0e-4"
      newtonMaxIter="10"
      maxTimeStepCuts="5"
      logLevel="1"/>
    <LinearSolverParameters
      solverType="gmres"
      preconditionerType="mgr"
      logLevel="1"
```

(continues on next page)

(continued from previous page)

```
    krylovAdaptiveTol="1"/>
</Hydrofracture>
```

- Rock and fracture deformation are modeled by the solid mechanics solver `SolidMechanicsLagrangianSSLE`. In this solver, we define `targetRegions` that includes both the continuum region and the fracture region. The name of the contact constitutive behavior is specified in this solver by the `contactRelationName`.

```
<SolidMechanicsLagrangianSSLE
  name="lagsolve"
  timeIntegrationOption="QuasiStatic"
  logLevel="1"
  discretization="FE1"
  targetRegions="{ Domain, Fracture }"
  contactRelationName="fractureContact">
  <NonlinearSolverParameters
    newtonTol="1.0e-6"/>
  <LinearSolverParameters
    solverType="gmres"
    krylovTol="1.0e-10"/>
</SolidMechanicsLagrangianSSLE>
```

- The single-phase fluid flow inside the fracture is solved by the finite volume method in the solver `SinglePhaseFVM`.

```
<SinglePhaseFVM
  name="SinglePhaseFlow"
  logLevel="1"
  discretization="singlePhaseTPFA"
  targetRegions="{ Fracture }">
  <NonlinearSolverParameters
    newtonTol="1.0e-5"
    newtonMaxIter="10"/>
  <LinearSolverParameters
    solverType="gmres"
    krylovTol="1.0e-12"/>
</SinglePhaseFVM>
```

- The solver `SurfaceGenerator` defines the fracture region and rock toughness `rockToughness="0.3e6"`. With `nodeBasedSIF="1"`, a node-based Stress Intensity Factor (SIF) calculation is chosen for the fracture propagation criterion.

```
<SurfaceGenerator
  name="SurfaceGen"
  targetRegions="{ Domain }"
  nodeBasedSIF="1"
  rockToughness="0.3e6"
  mpiCommOrder="1"/>
```

Constitutive laws

For this problem, a homogeneous and isotropic domain with one solid material is assumed. Its mechanical properties and associated fluid rheology are specified in the Constitutive section. The ElasticIsotropic model is used to describe the mechanical behavior of rock when subjected to fluid injection. The single-phase fluid model CompressibleSinglePhaseFluid is selected to simulate the response of water upon fracture propagation.

```xml
<Constitutive>
  <CompressibleSinglePhaseFluid
    name="water"
    defaultDensity="1000"
    defaultViscosity="0.001"
    referencePressure="0.0"
    compressibility="5e-12"
    referenceViscosity="1.0e-3"
    viscosibility="0.0"/>

  <ElasticIsotropic
    name="rock"
    defaultDensity="2700"
    defaultBulkModulus="20.0e9"
    defaultShearModulus="12.0e9"/>

  <CompressibleSolidParallelPlatesPermeability
    name="fractureFilling"
    solidModelName="nullSolid"
    porosityModelName="fracturePorosity"
    permeabilityModelName="fracturePerm"/>

  <NullModel
    name="nullSolid"/>

  <PressurePorosity
    name="fracturePorosity"
    defaultReferencePorosity="1.00"
    referencePressure="0.0"
    compressibility="0.0"/>

  <ParallelPlatesPermeability
    name="fracturePerm"/>

  <FrictionlessContact
    name="fractureContact"
    penaltyStiffness="1.0e0"
    apertureTableName="apertureTable"/>
</Constitutive>
```

All constitutive parameters such as density, viscosity, bulk modulus, and shear modulus are specified in the International System of Units.

Time history function

In the Tasks section, PackCollection tasks are defined to collect time history information from fields. Either the entire field or specified named sets of indices in the field can be collected. In this example, pressureCollection, apertureCollection, hydraulicApertureCollection and areaCollection are specified to output the time history of fracture characterisctics (pressure, width and area). objectPath="ElementRegions/Fracture/FractureSubRegion" indicates that these PackCollection tasks are applied to the fracure element subregion.

```xml
<Tasks>
  <PackCollection
    name="pressureCollection"
    objectPath="ElementRegions/Fracture/FractureSubRegion"
    fieldName="pressure"/>

  <PackCollection
    name="apertureCollection"
    objectPath="ElementRegions/Fracture/FractureSubRegion"
    fieldName="elementAperture"/>

  <PackCollection
    name="hydraulicApertureCollection"
    objectPath="ElementRegions/Fracture/FractureSubRegion"
    fieldName="hydraulicAperture"/>

  <PackCollection
    name="areaCollection"
    objectPath="ElementRegions/Fracture/FractureSubRegion"
    fieldName="elementArea"/>

  <!-- Collect aperture, pressure at the source for curve checks -->
  <PackCollection
    name="sourcePressureCollection"
    objectPath="ElementRegions/Fracture/FractureSubRegion"
    fieldName="pressure"
    setNames="{ source }"/>

  <PackCollection
    name="sourceHydraulicApertureCollection"
    objectPath="ElementRegions/Fracture/FractureSubRegion"
    fieldName="hydraulicAperture"
    setNames="{ source }"/>
</Tasks>
```

These tasks are triggered using the Event manager with a PeriodicEvent defined for the recurring tasks. GEOS writes one file named after the string defined in the filename keyword and formatted as a HDF5 file (pennyShapedViscosityDominated_output.hdf5). This TimeHistory file contains the collected time history information from specified time history collector. This file includes datasets for the simulation time, fluid pressure, element aperture, hydraulic aperture and element area for the propagating hydraulic fracture. A Python script is prepared to read and query any specified subset of the time history data for verification and visualization.

Initial and boundary conditions

Next, we specify initial and boundary conditions:

- Initial values: the `waterDensity`, `separableFace` and the `ruptureState` of the propagating fracture have to be initialized,
- Boundary conditions: fluid injection rates and the constraints of the outer boundaries have to be set.

In this example, a mass injection rate `SourceFlux` (`scale="-6.625"`) is applied at the surfaces of the initial fracture. Only one fourth of the total injection rate is defined in this boundary condition because only a quarter of the fracture is modeled (the problem is symmetric). The value given for `scale` is $Q_0 \rho_f / 4$ (not $Q_0/4$). All the outer boundaries are subject to roller constraints. These boundary conditions are set through the `FieldSpecifications` section.

```
<FieldSpecifications>
  <FieldSpecification
    name="waterDensity"
    initialCondition="1"
    setNames="{ fracture }"
    objectPath="ElementRegions"
    fieldName="water_density"
    scale="1000"/>

  <FieldSpecification
    name="separableFace"
    initialCondition="1"
    setNames="{ core }"
    objectPath="faceManager"
    fieldName="isFaceSeparable"
    scale="1"/>

  <FieldSpecification
    name="frac"
    initialCondition="1"
    setNames="{ fracture }"
    objectPath="faceManager"
    fieldName="ruptureState"
    scale="1"/>

  <FieldSpecification
    name="yconstraint"
    objectPath="nodeManager"
    fieldName="totalDisplacement"
    component="1"
    scale="0.0"
    setNames="{ yneg, ypos }"/>

  <FieldSpecification
    name="zconstraint"
    objectPath="nodeManager"
    fieldName="totalDisplacement"
    component="2"
    scale="0.0"
    setNames="{ zneg, zpos }"/>
```

(continues on next page)

(continued from previous page)

```
    <FieldSpecification
      name="xconstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="0"
      scale="0.0"
      setNames="{ xneg, xpos }"/>

    <SourceFlux
      name="sourceTerm"
      objectPath="ElementRegions/Fracture"
      scale="-6.625"
      setNames="{ source }"/>
  </FieldSpecifications>
```

The parameters used in the simulation are summarized in the following table.

Symbol	Parameter	Unit	Value
K	Bulk Modulus	[GPa]	20.0
G	Shear Modulus	[GPa]	12.0
K_{IC}	Rock Toughness	[MPa.m$^{1/2}$]	0.3
μ	Fluid Viscosity	[Pa.s]	1.0x10^{-3}
Q_0	Injection Rate	[m^3/s]	0.0265
t_{inj}	Injection Time	[s]	400

Inspecting results

The following figure shows the distribution of σ_{zz} at $t = 400s$ within the computational domain..

First, by running the query script

```
python ./hydrofractureQueries.py pennyShapedViscosityDominated
```

the HDF5 output is postprocessed and temporal evolution of fracture characterisctics (fluid pressure and fracture width at fluid inlet and fracure radius) are saved into a txt file `model-results.txt`, which can be used for verification and visualization:

```
[['      time', '  pressure', '  aperture', '    length']]
         2 1.654e+06 0.0006768     8.137
         4 1.297e+06  0.000743     10.59
         6 1.115e+06 0.0007734     12.36
         8 1.005e+06 0.0007918     13.73
        10 9.482e+05 0.0008189     15.14
```

Note: GEOS python tools `geosx_xml_tools` should be installed to run the query script (See *Python Tools Setup* for details).

Next, GEOS simulation results (markers) and asymptotic solutions (curves) for the case with viscosity-storage dominated assumptions are plotted together in the following figure, which is generated using the visualization script:

```
python ./pennyShapedViscosityDominatedFigure.py
```

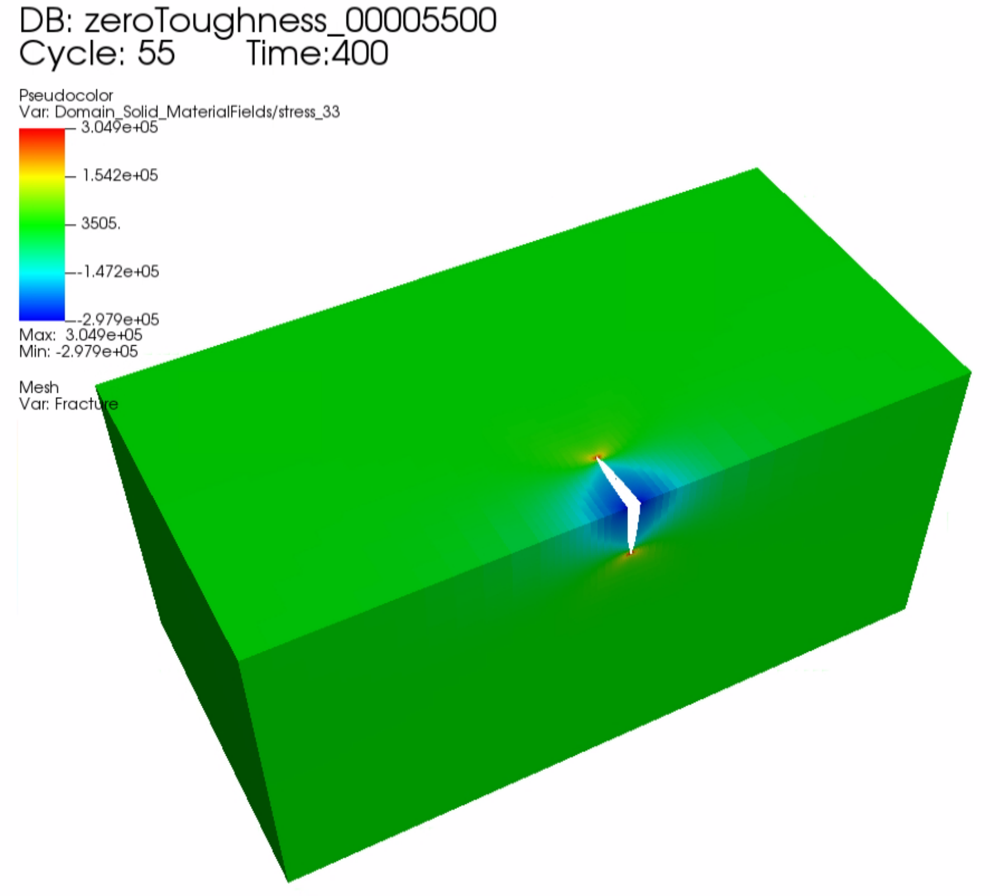
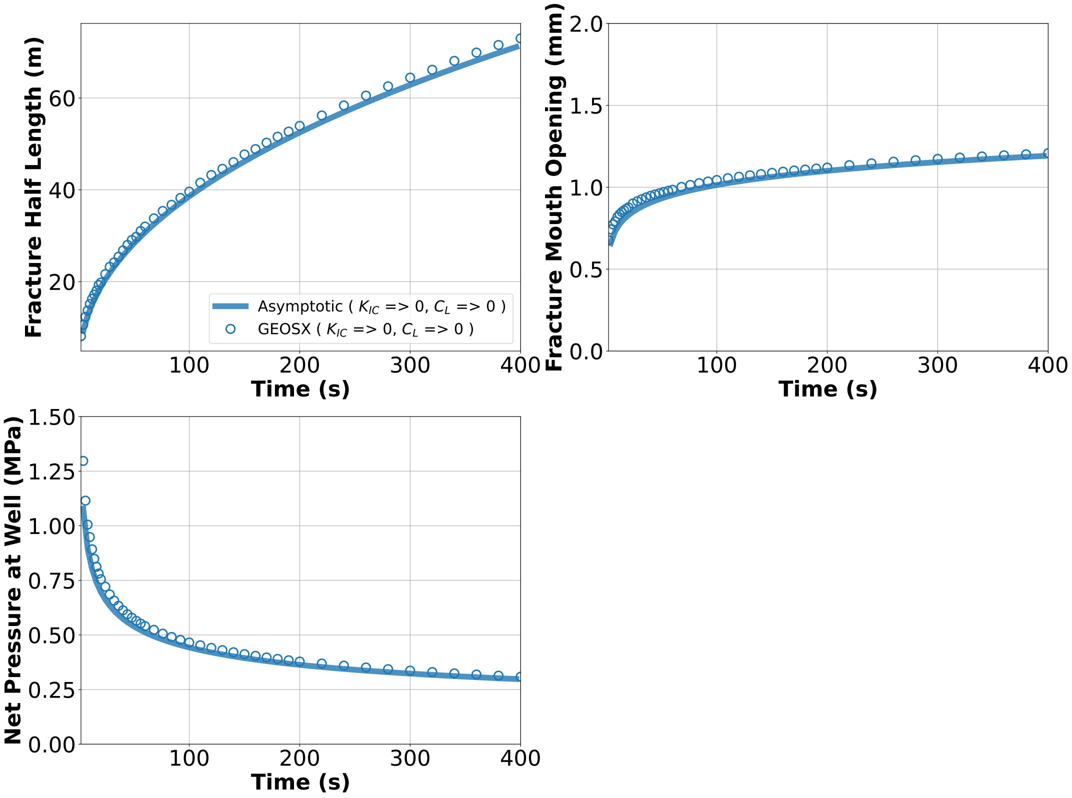


Fig. 1.33: Simulation result of σ_{zz} at $t = 400s$

As seen, GEOS predictions of the temporal evolution of fracture radius, wellbore aperture and pressure at fluid inlet are nearly identical to the asymptotic solutions.



To go further

Feedback on this example

For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

Viscosity-Storage-Dominated PKN Hydraulic Fracture

Context

In this example, we simulate the propagation of a Perkins–Kern–Nordgren (PKN) fracture in a viscosity-storage-dominated regime, a classic benchmark in hydraulic fracturing. The developed planar fracture displays an elliptical vertical cross-section. Unlike KGD and penny-shaped fractures, the growth height of a PKN fracture is constrained by mechanical barriers (such as bedding layers, sedimentary laminations, or weak interfaces), thus promoting lateral propagation. This problem is solved using the hydrofracture solver in GEOS to obtain the temporal evolutions of the fracture characteristics (length, aperture, and pressure). We validate these simulated values against existing analytical solutions (Kovalyshen and Detournay, 2010; Economides and Nolte, 2000).

Input file

This example uses no external input files. Everything we need is contained within two GEOS input files:

```
inputFiles/hydraulicFracturing/pknViscosityDominated_base.xml
```

```
inputFiles/hydraulicFracturing/pknViscosityDominated_benchmark.xml
```

Python scripts for post-processing and visualizing the simulation results are also prepared:

```
inputFiles/hydraulicFracturing/scripts/hydrofractureQueries.py
```

```
inputFiles/hydraulicFracturing/scripts/hydrofractureFigure.py
```

Description of the case

In this example, a hydraulic fracture initiates and propagates from the center of a 20m-thick layer. This layer is homogeneous and bounded by neighboring upper and lower layers. For viscosity-dominated fractures, more energy is necessary to move the fracturing fluid than to split the intact rock. If fluid leak-off is neglected, storage-dominated propagation occurs with most of the injected fluid confined within the open surfaces. To meet the requirements of the viscosity-storage-dominated assumptions, impermeable domain (no fluid leak-off), incompressible fluid with constant viscosity ($1.0cp$) and ultra-low rock toughness ($0.1MPa\sqrt{m}$) are chosen in the GEOS simulation. With these parameters, the fracture stays within the target layer; it extends horizontally and meets the conditions of the PKN fracture in a viscosity-storage-dominated regime.

We assume that the fluid injected in the fracture follows the lubrication equation resulting from mass conservation and Poiseuille's law. The fracture propagates by creating new surfaces if the stress intensity factor exceeds the local rock toughness K_{IC}. As the geometry of the PKN fracture exhibits symmetry, the simulation is reduced to a quarter-scale. For verification purposes, a plane strain deformation is considered in the numerical model.

We set up and solve a hydraulic fracture model to obtain the temporal solutions of the fracture half length l, the net pressure p_0 and the fracture aperture w_0 at the fluid inlet for the PKN fracture propagating in viscosity-storage-dominated regime. Kovalyshen and Detournay (2010) and Economides and Nolte (2000) derived the analytical solutions for this classic hydraulic fracture problem, used here to verify the results of the GEOS simulations:

$$l(t) = 0.3817(\frac{E_p Q_0^3 t^4}{\mu h^4})^{1/5}$$

$$w_0(t) = 3(\frac{\mu Q_0 l}{E_p})^{1/4}$$

$$p_0(t) = (\frac{16\mu Q_0 E_p^3 l}{\pi h^4})^{1/4}$$

where the plane modulus E_p is related to Young's modulus E and Poisson's ratio ν:

$$E_p = \frac{E}{1-\nu^2}$$

For this example, we focus on the `Mesh`, the `Constitutive`, and the `FieldSpecifications` tags.

Mesh

The following figure shows the mesh used in this problem.

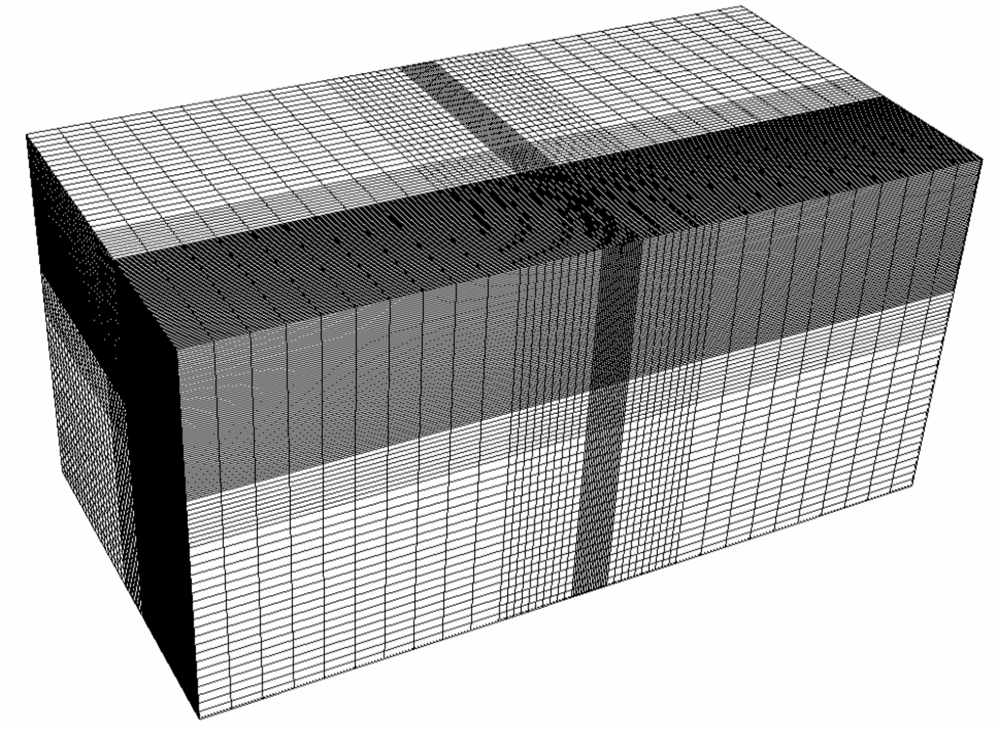


Fig. 1.34: Generated mesh

We use the internal mesh generator to create a computational domain ($400\,m \times 400\,m \times 800\,m$), as parametrized in the `InternalMesh` XML tag. The structured mesh contains 105 x 105 x 60 eight-node brick elements in the x, y, and z directions respectively. Such eight-node hexahedral elements are defined as `C3D8` elementTypes, and their collection forms a mesh with one group of cell blocks named here `cb1`. Local refinement is performed for the elements in the vicinity of the fracture plane.

```
<Mesh>
  <InternalMesh
    name="mesh1"
    elementTypes="{ C3D8 }"
    xCoords="{ 0, 150, 200, 400 }"
    yCoords="{ 0, 150, 200, 400 }"
    zCoords="{ -400, -100, -20, 20, 100, 400 }"
    nx="{ 75, 10, 20 }"
    ny="{ 75, 10, 20 }"
    nz="{ 10, 10, 20, 10, 10 }"
    cellBlockNames="{ cb1 }"/>
</Mesh>
```

The fracture plane is defined by a nodeset occupying a small region within the computational domain, where the fracture tends to open and propagate upon fluid injection:

```
  <Box
    name="core"
```

(continues on next page)

(continued from previous page)

```
      xMin="{ -500.1, -500.1, -0.1 }"
      xMax="{ 500.1, 10.1, 0.1 }"/>
```

Solid mechanics solver

GEOS is a multi-physics platform. Different combinations of physics solvers available in the code can be applied in different regions of the domain and be functional at different stages of the simulation. The `Solvers` tag in the XML file is used to list and parameterize these solvers.

Three elementary solvers are combined in the solver `hydrofracture` to model the coupling between fluid flow within the fracture, rock deformation, fracture deformation and propagation:

```
    <Hydrofracture
      name="hydrofracture"
      solidSolverName="lagsolve"
      flowSolverName="SinglePhaseFlow"
      surfaceGeneratorName="SurfaceGen"
      logLevel="1"
      targetRegions="{ Fracture }"
      contactRelationName="fractureContact"
      maxNumResolves="5"
      initialDt="0.1">
      <NonlinearSolverParameters
        newtonTol="1.0e-4"
        newtonMaxIter="10"
        maxTimeStepCuts="5"
        maxAllowedResidualNorm="1e+15"/>
      <LinearSolverParameters
        solverType="gmres"
        preconditionerType="mgr"
        logLevel="1"
        krylovAdaptiveTol="1"/>
    </Hydrofracture>
```

- Rock and fracture deformations are modeled by the solid mechanics solver `SolidMechanicsLagrangianSSLE`. In this solver, we define `targetRegions` that includes both the continuum region and the fracture region. The name of the contact constitutive behavior is specified in this solver by the `contactRelationName`.

```
    <SolidMechanicsLagrangianSSLE
      name="lagsolve"
      timeIntegrationOption="QuasiStatic"
      discretization="FE1"
      targetRegions="{ Domain, Fracture }"
      contactRelationName="fractureContact">
      <NonlinearSolverParameters
        newtonTol="1.0e-6"/>
      <LinearSolverParameters
        solverType="gmres"
        krylovTol="1.0e-10"/>
    </SolidMechanicsLagrangianSSLE>
```

- The single-phase fluid flow inside the fracture is solved by the finite volume method in the solver

SinglePhaseFVM.

```
  <SinglePhaseFVM
    name="SinglePhaseFlow"
    discretization="singlePhaseTPFA"
    targetRegions="{ Fracture }">
    <NonlinearSolverParameters
      newtonTol="1.0e-5"
      newtonMaxIter="10"/>
    <LinearSolverParameters
      solverType="gmres"
      krylovTol="1.0e-12"/>
  </SinglePhaseFVM>
```

- The solver SurfaceGenerator defines the fracture region and rock toughness rockToughness="0.1e6". With nodeBasedSIF="1", a node-based Stress Intensity Factor (SIF) calculation is chosen for the fracture propagation criterion.

```
  <SurfaceGenerator
    name="SurfaceGen"
    targetRegions="{ Domain }"
    nodeBasedSIF="1"
    rockToughness="0.1e6"
    mpiCommOrder="1"/>
```

Constitutive laws

For this problem, a homogeneous and isotropic domain with one solid material is assumed. Its mechanical properties and associated fluid rheology are specified in the Constitutive section. ElasticIsotropic model is used to describe the mechanical behavior of rock when subjected to fluid injection. The single-phase fluid model CompressibleSinglePhaseFluid is selected to simulate the response of water upon fracture propagation.

```
<Constitutive>
  <CompressibleSinglePhaseFluid
    name="water"
    defaultDensity="1000"
    defaultViscosity="0.001"
    referencePressure="0.0"
    compressibility="5e-12"
    referenceViscosity="1.0e-3"
    viscosibility="0.0"/>

  <ElasticIsotropic
    name="rock"
    defaultDensity="2700"
    defaultBulkModulus="20.0e9"
    defaultShearModulus="12.0e9"/>

  <CompressibleSolidParallelPlatesPermeability
    name="fractureFilling"
    solidModelName="nullSolid"
    porosityModelName="fracturePorosity"
```

(continues on next page)

(continued from previous page)

```
    permeabilityModelName="fracturePerm"/>

  <NullModel
    name="nullSolid"/>

  <PressurePorosity
    name="fracturePorosity"
    defaultReferencePorosity="1.00"
    referencePressure="0.0"
    compressibility="0.0"/>

  <ParallelPlatesPermeability
    name="fracturePerm"/>

  <FrictionlessContact
    name="fractureContact"
    penaltyStiffness="1.0e0"
    apertureTableName="apertureTable"/>
</Constitutive>
```

All constitutive parameters such as density, viscosity, bulk modulus, and shear modulus are specified in the International System of Units.

Time history function

In the Tasks section, PackCollection tasks are defined to collect time history information from fields. Either the entire field or specified named sets of indices in the field can be collected. In this example, pressureCollection, apertureCollection, hydraulicApertureCollection and areaCollection are specified to output the time history of fracture characterisctics (pressure, width and area). objectPath="ElementRegions/Fracture/FractureSubRegion" indicates that these PackCollection tasks are applied to the fracure element subregion.

```
<Tasks>
  <PackCollection
    name="pressureCollection"
    objectPath="ElementRegions/Fracture/FractureSubRegion"
    fieldName="pressure"/>

  <PackCollection
    name="apertureCollection"
    objectPath="ElementRegions/Fracture/FractureSubRegion"
    fieldName="elementAperture"/>

  <PackCollection
    name="hydraulicApertureCollection"
    objectPath="ElementRegions/Fracture/FractureSubRegion"
    fieldName="hydraulicAperture"/>

  <PackCollection
    name="areaCollection"
    objectPath="ElementRegions/Fracture/FractureSubRegion"
    fieldName="elementArea"/>
```

(continues on next page)

(continued from previous page)

```
    <!-- Collect aperture, pressure at the source for curve checks -->
    <PackCollection
      name="sourcePressureCollection"
      objectPath="ElementRegions/Fracture/FractureSubRegion"
      fieldName="pressure"
      setNames="{ source }"/>

    <PackCollection
      name="sourceHydraulicApertureCollection"
      objectPath="ElementRegions/Fracture/FractureSubRegion"
      fieldName="hydraulicAperture"
      setNames="{ source }"/>
  </Tasks>
```

These tasks are triggered using the `Event` manager with a `PeriodicEvent` defined for the recurring tasks. GEOS writes one file named after the string defined in the `filename` keyword and formatted as a HDF5 file (`pknViscosityDominated_output.hdf5`). This TimeHistory file contains the collected time history information from specified time history collector. This file includes datasets for the simulation time, fluid pressure, element aperture, hydraulic aperture and element area for the propagating hydraulic fracture. A Python script is prepared to read and query any specified subset of the time history data for verification and visualization.

Initial and boundary conditions

The next step is to specify:

- The initial values: the `waterDensity`, `separableFace` and the `ruptureState` of the propagating fracture have to be initialized,
- The boundary conditions: fluid injection rates and the constraints of the outer boundaries have to be set.

In this example, a mass injection rate `SourceFlux` (`scale="-6.625"`) is applied at the surfaces of the initial fracture. Only one fourth of the total injection rate is used because only a quarter of the fracture is modeled (the problem is symmetric). The value given for `scale` is $Q_0 \rho_f / 4$ (not $Q_0/4$). All the outer boundaries are subject to roller constraints. These boundary conditions are set through the `FieldSpecifications` section.

```
  <FieldSpecifications>
    <FieldSpecification
      name="waterDensity"
      initialCondition="1"
      setNames="{ fracture }"
      objectPath="ElementRegions"
      fieldName="water_density"
      scale="1000"/>

    <FieldSpecification
      name="separableFace"
      initialCondition="1"
      setNames="{ core }"
      objectPath="faceManager"
      fieldName="isFaceSeparable"
      scale="1"/>
```

(continues on next page)

(continued from previous page)

```
    <FieldSpecification
      name="frac"
      initialCondition="1"
      setNames="{ fracture }"
      objectPath="faceManager"
      fieldName="ruptureState"
      scale="1"/>

    <FieldSpecification
      name="yconstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="1"
      scale="0.0"
      setNames="{ yneg, ypos }"/>

    <FieldSpecification
      name="zconstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="2"
      scale="0.0"
      setNames="{ zneg, zpos }"/>

    <FieldSpecification
      name="xconstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="0"
      scale="0.0"
      setNames="{ xneg, xpos }"/>

    <SourceFlux
      name="sourceTerm"
      objectPath="ElementRegions/Fracture"
      scale="-6.625"
      setNames="{ source }"/>
  </FieldSpecifications>
```

The parameters used in the simulation are summarized in the following table.

Symbol	Parameter	Unit	Value
K	Bulk Modulus	[GPa]	20.0
G	Shear Modulus	[GPa]	12.0
K_{IC}	Rock Toughness	[MPa.m$^{1/2}$]	0.1
μ	Fluid Viscosity	[Pa.s]	1.0x10^{-3}
Q_0	Injection Rate	[m^3/s]	0.0265
t_{inj}	Injection Time	[s]	200
h_f	Fracture Height	[m]	20

Inspecting results

The following figure shows the distribution of σ_{zz} at $t = 200s$ within the computational domain..

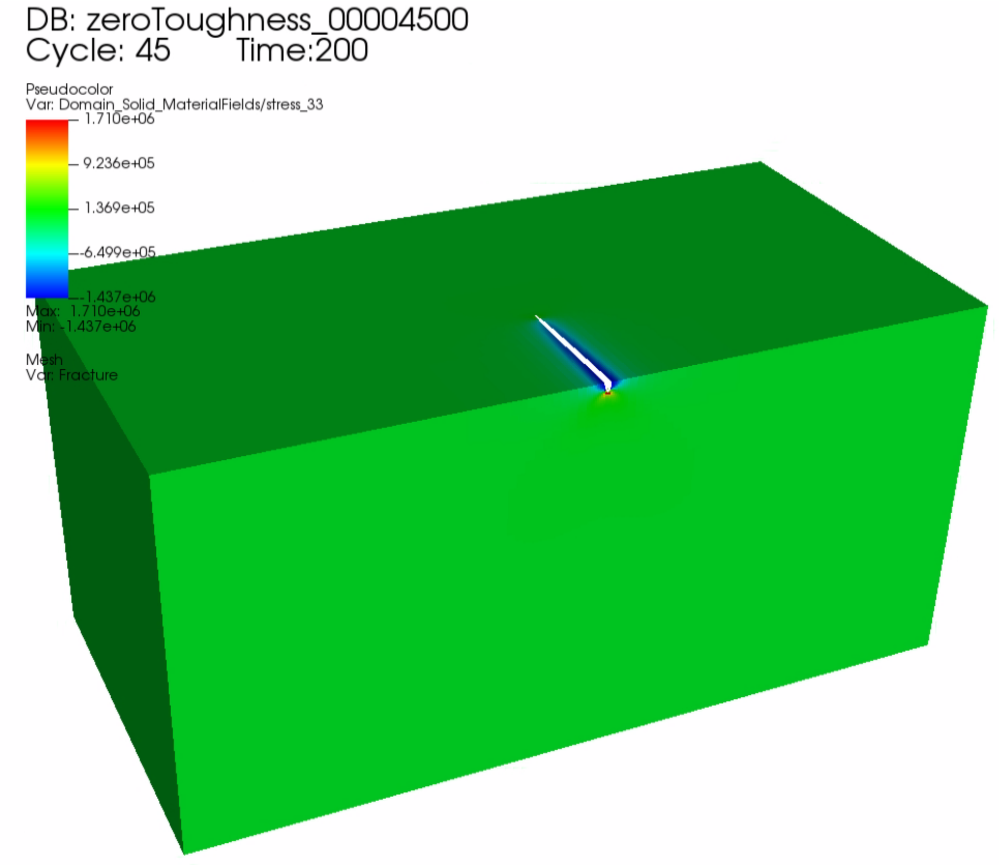


Fig. 1.35: Simulation result of σ_{zz} at $t = 200s$

First, by running the query script

```
python ./hydrofractureQueries.py pknViscosityDominated
```

the HDF5 output is postprocessed and temporal evolution of fracture characterisctics (fluid pressure and fracture width at fluid inlet and fracure half length) are saved into a txt file `model-results.txt`, which can be used for verification and visualization:

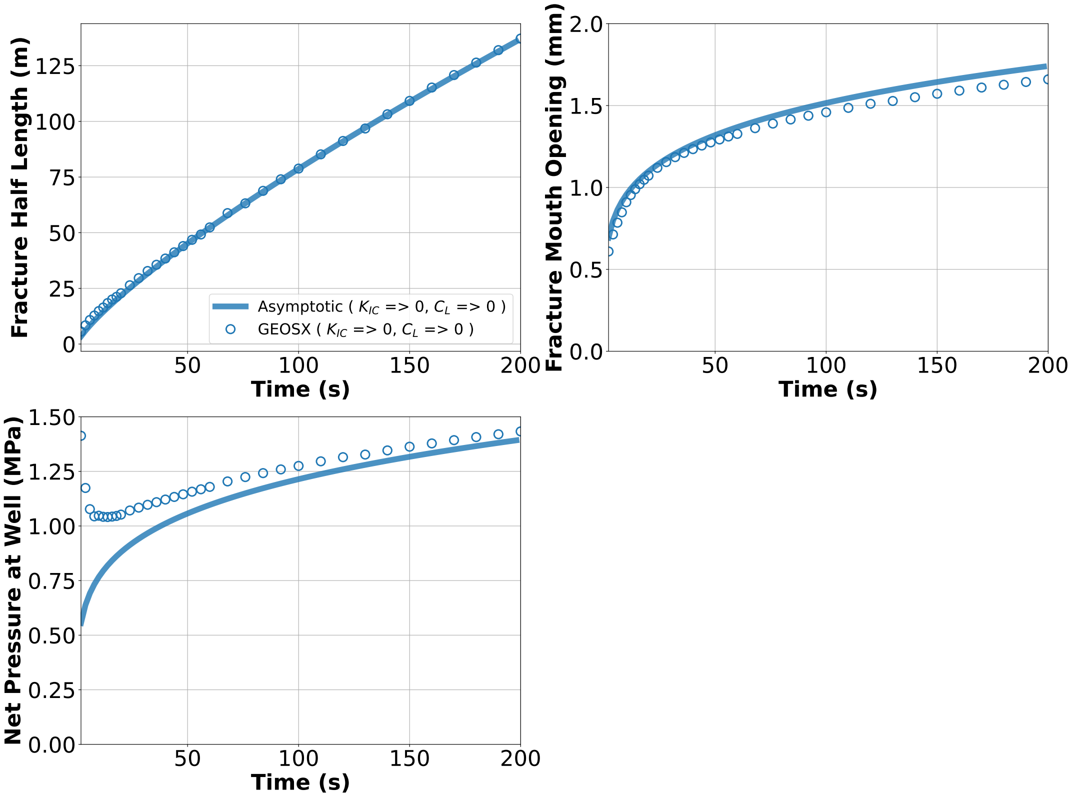
```
[['      time', '  pressure', '  aperture', '    length']]
         2 1.413e+06 0.0006093       5.6
         4 1.174e+06 0.0007132       8.4
         6 1.077e+06 0.0007849      10.8
         8 1.044e+06 0.0008482      12.8
        10 1.047e+06 0.0009098      14.8
```

Note: GEOS python tools `geosx_xml_tools` should be installed to run the query script (See *Python Tools Setup* for details).

Next, figure below shows the comparisons between the results from GEOS simulations (markers) and the corresponding analytical solutions (curves) for the example with viscosity-storage dominated assumptions, which is generated using the visualization script:

```
python ./pknViscosityDominatedFigure.py
```

The evolution in time of the fracture half-length, the near-wellbore fracture aperture, and the fluid pressure all correlate well with the analytical solutions.



To go further

Feedback on this example

For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

Proppant Slot Test

Context

In this example, a simulation is built up to model a proppant slot test. In this way, the implemented proppant model is validated by comparing numerical results with the corresponding experimental data. Furthermore, this calibrated proppant model can allow field engineers to customize stimulation design and optimize field operations in multiple engineering aspects (Huang et al., 2021).

Input file

This example uses no external input files and everything is contained within a single xml file that is located at:

```
inputFiles/proppant/ProppantSlotTest_base.xml
```

```
inputFiles/proppant/ProppantSlotTest_benchmark.xml
```

Description of the case

Chun et al. (2020) conducted slot tests on proppant transport with slickwater. As shown below, a 4 ft X 1 ft slot with 0.3 in gap width was constructed. Three fluid inlets with 0.5 in inner diameter were placed at the right side of the slot, which were three inches away from each other. One outlet was placed on the top side to allow pressure relief. The other one was located on the left side acting as a fluid sink. In their tests, to resemble a slickwater fracturing treatment, the proppant concentration was kept at 1.5 ppg and the viscosity of carrying fluid was approximately 1 cp. The slurry was mixed well and then injected into the flow channel at a constant injection rate of 6 gpm. A simulation case with the same settings is built up to mimic these slot tests. A vertical and impermeable fracture surface is assumed in this case, which eliminates the effect of fracture plane inclination and fluid leak-off. A static fracture with an uniform aperture of 0.3 in is defined and fracture propagation is not involved. 30/50 mesh proppant is injected via the three inlets and is flowed through the slot for 30 seconds.

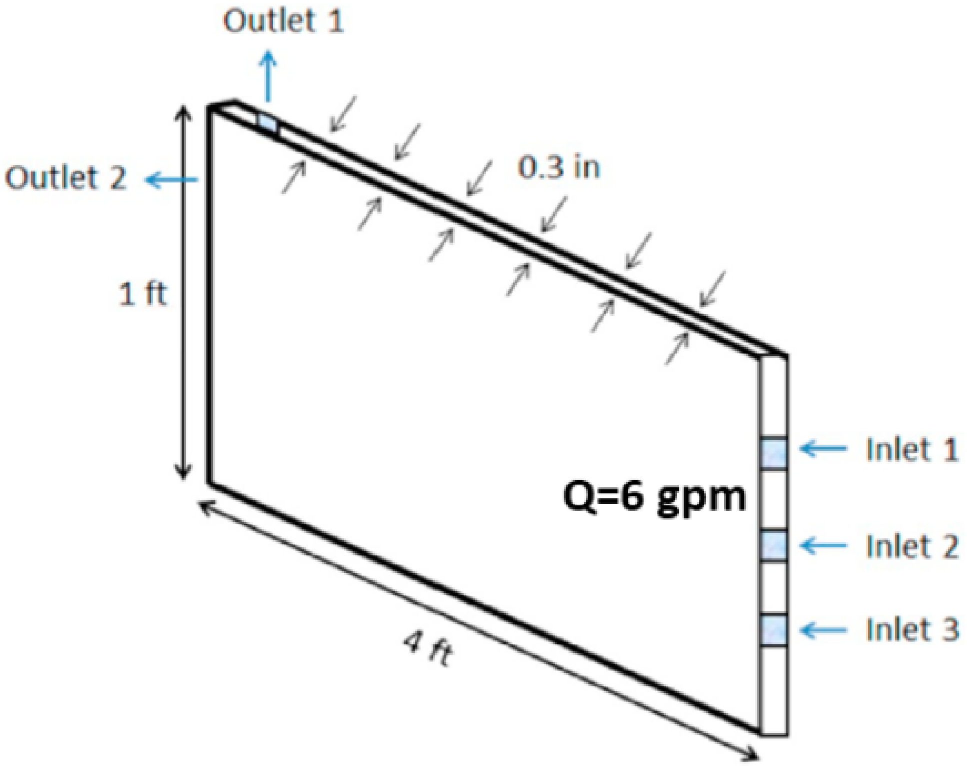


Fig. 1.36: Configuration of the slot for proppant transport experiment (after Chun et al., 2020)

To simulate proppant transport phenomenon, a proppant solver based on the assumption of multi-component single phase flow is used in this example. Proppant concentration and distribution within the slot are numerically calculated by solving the equations of proppant transport in hydraulic fractures. These numerical predictions are then validated against the corresponding testing results (Chun et al., 2020).

In this example, we focus our attention on the `Solvers`, `Constitutive` and `FieldSpecifications` tags.

Mesh

The following figure shows the mesh used for solving this problem.

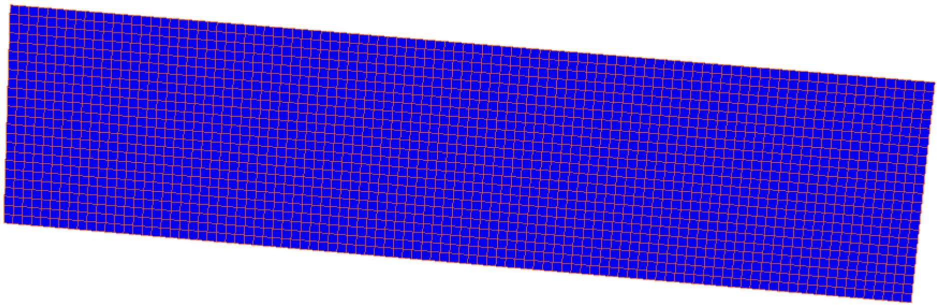


Fig. 1.37: Mesh for simulating the proppant slot tests.

We use the internal mesh generator `InternalMesh` to create a computational domain. This mesh contains 2 x 97 x 24 eight-node brick elements in the x, y and z directions, respectively. Here, a structured three-dimensional mesh is generated with `C3D8` as the elementTypes (eight-node hexahedral elements). This mesh is defined as a cell block with the name `cb1`.

```xml
<Mesh>
  <InternalMesh
    name="mesh"
    elementTypes="{ C3D8 }"
    xCoords="{ -1, 1 }"
    yCoords="{ 0, 1.2319 }"
    zCoords="{ 0, 0.3048 }"
    nx="{ 2 }"
    ny="{ 97 }"
    nz="{ 24 }"
    cellBlockNames="{ cb1 }"/>
</Mesh>
```

Proppant transport solver

GEOS is a multi-physics platform. Different combinations of physics solvers available in the code can be applied in different regions of the domain and be functional at different stages of the simulation. The `Solvers` tag in the XML file is used to define these solvers.

To specify a coupling between two different solvers, we define and characterize each single-physics solver separately. Then, we customize a *coupling solver* between these single-physics solvers as an additional solver. This approach allows for generality and flexibility in constructing multi-physics solvers. The order of specifying these solvers is not restricted in GEOS. Note that end-users should give each single-physics solver a meaningful and distinct name, as GEOS will recognize these single-physics solvers based on their customized names and create user-expected coupling.

As demonstrated in this example, to setup a coupled proppant transport solver, we need to define three different solvers in the XML file:

- the proppant transport solver for the fracture region, a solver of type `ProppantTransport` called here `ProppantTransport` (see *Proppant Transport Solver* for more information),

```
<ProppantTransport
  name="ProppantTransport"
  logLevel="1"
  updateProppantPacking="1"
  proppantDiameter="4.5e-4"
  frictionCoefficient="0.04"
  criticalShieldsNumber="0.0"
  maxProppantConcentration="0.62"
  discretization="singlePhaseTPFA"
  targetRegions="{ Fracture }">
  <NonlinearSolverParameters
    newtonTol="1.0e-6"
    newtonMaxIter="8"
    lineSearchAction="None"
    maxTimeStepCuts="5"/>
  <LinearSolverParameters
    solverType="gmres"
    krylovTol="1.0e-7"/>
</ProppantTransport>
```

- the single-phase flow solver, a solver of type `SinglePhaseProppantFVM` called here `SinglePhaseFVM`,

```
<SinglePhaseProppantFVM
  name="SinglePhaseFVM"
  logLevel="1"
  discretization="singlePhaseTPFA"
  targetRegions="{ Fracture }">
  <NonlinearSolverParameters
    newtonTol="1.0e-6"
    newtonMaxIter="8"
    lineSearchAction="None"
    newtonMinIter="0"/>
  <LinearSolverParameters
    solverType="gmres"
    preconditionerType="amg"
    krylovTol="1.0e-7"/>
</SinglePhaseProppantFVM>
```

- the coupling solver (`FlowProppantTransport`) that binds the two single-physics solvers above, which is named as `FlowProppantTransport`

```
<FlowProppantTransport
  name="FlowProppantTransport"
  proppantSolverName="ProppantTransport"
  flowSolverName="SinglePhaseFVM"
  targetRegions="{ Fracture }"
  logLevel="1"/>
```

In this example, let us focus on the coupling solver. This solver (`FlowProppantTransport`) describes the coupling process between proppant and flow transport within the `Fracture` region. In this way, the two single-physics solvers (`ProppantTransport` and `SinglePhaseFVM`) are sequentially called to solve the sub-problems (proppant transport and pressure problem, respectively) involved in this test case.

Constitutive laws

For this slot test, 30/50 mesh proppant is injected via the three inlets and flowing through the slot for 30 seconds. The viscosity of carrying fluid is 0.001 Pa.s to resemble slickwater fracturing. In this example, the solid and fluid materials are named as `sand` and `water` respectively. Proppant characterization and fluid rheology are specified in the `Constitutive` section:

```
<Constitutive>
  <ProppantSlurryFluid
    name="water"
    referencePressure="1e5"
    referenceDensity="1000"
    compressibility="0.0"
    maxProppantConcentration="0.62"
    referenceViscosity="0.001"
    referenceProppantDensity="2550.0"/>

  <ParticleFluid
    name="sand"
    particleSettlingModel="Stokes"
    hinderedSettlingCoefficient="4.5"
    proppantDensity="2550.0"
    proppantDiameter="4.5e-4"
    maxProppantConcentration="0.62"/>

  <ProppantSolidProppantPermeability
    name="fractureFilling"
    solidModelName="nullSolid"
    porosityModelName="fracturePorosity"
    permeabilityModelName="fracturePerm"/>

  <NullModel
    name="nullSolid"/>

  <ProppantPorosity
    name="fracturePorosity"
    defaultReferencePorosity="1.00"
    maxProppantConcentration="0.62"/>

  <ProppantPermeability
    name="fracturePerm"
    proppantDiameter="4.5e-4"
    maxProppantConcentration="0.62"/>
</Constitutive>
```

The constitutive parameters such as proppant density and proppant diameter are specified in the International System of Units.

Initial and boundary conditions

The next step is to specify fields, including:

- The initial value (fracture aperture, fluid pressure and proppant concentration within the fracture have to be initialized)
- The boundary conditions (fluid pressure and proppant concentration at fluid inlets and outlets)

These boundary conditions are set up through the `FieldSpecifications` section. At a constant injection rate, the slurry is equally flowing into the open channel through three inlets.

```
<FieldSpecifications>
  <FieldSpecification
    name="frac"
    initialCondition="1"
    setNames="{ fracture }"
    objectPath="faceManager"
    fieldName="ruptureState"
    scale="1"/>

  <FieldSpecification
    name="fracAp"
    initialCondition="1"
    objectPath="ElementRegions/Fracture"
    fieldName="elementAperture"
    scale="7.62e-3"
    setNames="{ fracture }"/>

  <FieldSpecification
    name="frac1"
    initialCondition="1"
    objectPath="ElementRegions/Fracture"
    fieldName="pressure"
    scale="0.0"
    component="0"
    setNames="{ fracture }"/>

  <FieldSpecification
    name="frac2"
    initialCondition="1"
    objectPath="ElementRegions/Fracture"
    fieldName="proppantConcentration"
    scale="0.0"
    component="0"
    setNames="{ fracture }"/>

  <FieldSpecification
    name="frac3"
    initialCondition="1"
    objectPath="ElementRegions/Fracture"
    fieldName="isProppantBoundary"
    component="0"
    setNames="{ fracture }"/>
```

(continues on next page)

(continued from previous page)

```
    <FieldSpecification
      name="frac4"
      initialCondition="1"
      objectPath="ElementRegions/Fracture"
      fieldName="isProppantBoundary"
      scale="1"
      component="0"
      setNames="{ left0 }"/>

    <SourceFlux
      name="left1a"
      objectPath="ElementRegions/Fracture"
      scale="-0.14"
      component="0"
      setNames="{ left1 }"/>

    <FieldSpecification
      name="left1b"
      objectPath="ElementRegions/Fracture"
      fieldName="proppantConcentration"
      scale="0.07"
      component="0"
      setNames="{ left1 }"/>

    <SourceFlux
      name="left2a"
      objectPath="ElementRegions/Fracture"
      scale="-0.14"
      component="0"
      setNames="{ left2 }"/>

    <FieldSpecification
      name="left2b"
      objectPath="ElementRegions/Fracture"
      fieldName="proppantConcentration"
      scale="0.07"
      component="0"
      setNames="{ left2 }"/>

    <SourceFlux
      name="left3a"
      objectPath="ElementRegions/Fracture"
      scale="-0.14"
      component="0"
      setNames="{ left3 }"/>

    <FieldSpecification
      name="left3b"
      objectPath="ElementRegions/Fracture"
      fieldName="proppantConcentration"
      scale="0.07"
      component="0"
```

(continues on next page)

(continued from previous page)

```
      setNames="{ left3 }"/>

    <FieldSpecification
      name="right1"
      objectPath="ElementRegions/Fracture"
      fieldName="pressure"
      scale="0.0"
      component="0"
      setNames="{ right }"/>

    <FieldSpecification
      name="right2"
      objectPath="ElementRegions/Fracture"
      fieldName="proppantConcentration"
      scale="0.0"
      component="0"
      setNames="{ right }"/>
  </FieldSpecifications>
```

Note: For static (non-propagating) fracture problems, the fields `ruptureState` and `elementAperture` should be provided in the initial conditions. `FieldName="pressure"` here means that the source flux term is added to the mass balance equation for pressure.

The parameters used in the simulation are summarized in the following table.

Symbol	Parameter	Unit	Value
d_p	Proppant Diameter	[m]	0.00045
f	Darcy Friction Coefficient	[-]	0.04
N_{sh}	Critical Shields Number	[-]	0.0
c_s	Max Fraction of Proppant	[-]	0.62
ρ_f	Fluid Density	[kg/m^3]	1000
μ_f	Fluid Viscosity	[Pa*s]	0.001
ρ_p	Proppant Density	[kg/m^3]	2550
λ_s	Hindered Settling Coefficient	[-]	4.5
c_p	Proppant Concentration in Slurry	[m^3/m^3]	0.07
L	Fracture Length	[m]	1.219
H	Fracture Height	[m]	0.3048
a	Fracture Aperture	[m]	0.00762
Q	Injection Rate	[m^3/s]	0.0003785

Inspecting results

The following figure shows the modelling prediction of proppant distribution at 10 s and 30 s, which are compared with the experiments in (Chun et al., 2020). Due to proppant settling in low viscosity fluid, a heterogeneous proppant distribution is obtained, which evolves with injection time. Three different zones (immobile proppant bed, suspended proppant and clean fluid) are visually identified for both the presented experiment and simulation.

As shown below, consistently, the modelling predictions (green curve) on proppant transport and distribution show a good agreement with the reported experimental data (red dot) at each time.

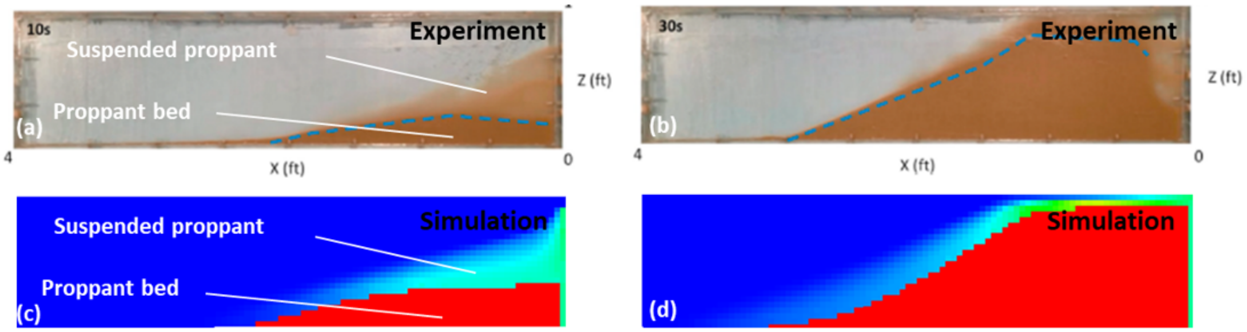
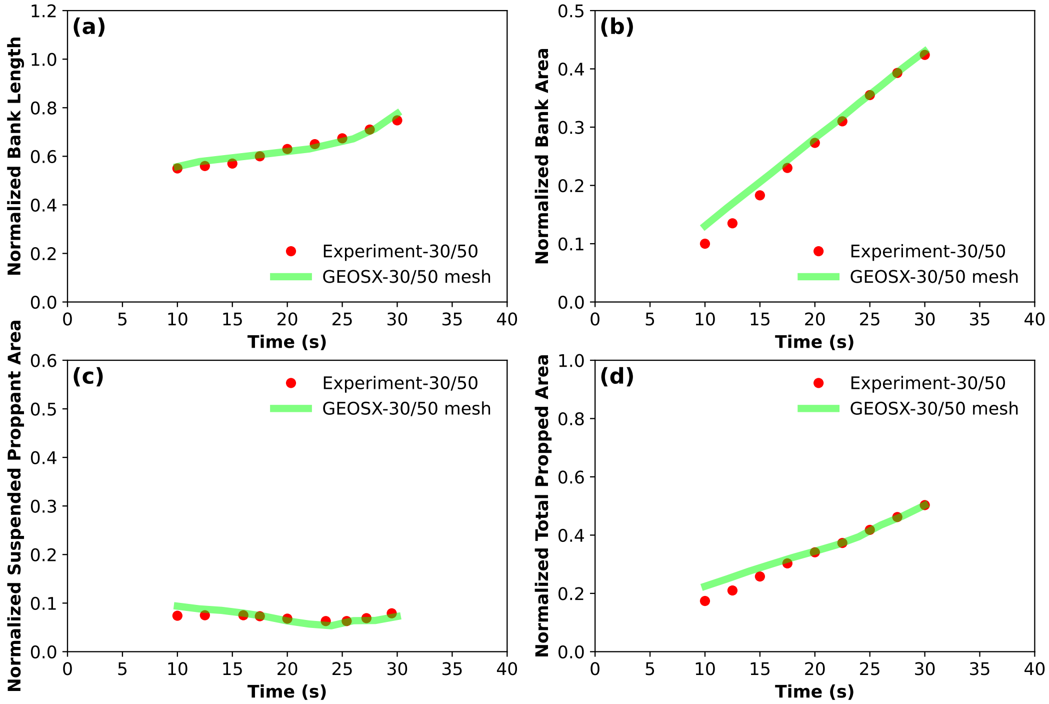


Fig. 1.38: Proppant distribution profile



To go further

Feedback on this example

This concludes the example on simulating a proppant slot test. For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

For more details

- More on proppant solver, please see *Proppant Transport Solver*.

Wellbore Problems

Kirsch Wellbore Problem

Context

In this example, we simulate a vertical elastic wellbore subjected to in-situ stress and the induced elastic deformation of the reservoir rock. Kirsch's solution to this problem provides the stress and displacement fields developing around a circular cavity, which is hereby employed to verify the accuracy of the numerical results. For this example, the `TimeHistory` function and python scripts are used to output and post-process multi-dimensional data (stress and displacement).

Input file

Everything required is contained within two GEOS input files located at:

```
inputFiles/solidMechanics/KirschProblem_base.xml
```

```
inputFiles/solidMechanics/KirschProblem_benchmark.xml
```

Description of the case

We solve a drained wellbore problem subjected to anisotropic horizontal stress (σ_{xx} and σ_{yy}) as shown below. This is a vertical wellbore drilled in an infinite, homogeneous, isotropic, and elastic medium. Far-field in-situ stresses and internal supporting pressure acting at the circular cavity cause a mechanical deformation of the reservoir rock and stress concentration in the near-wellbore region. For verification purpose, a plane strain condition is considered for the numerical model.

In this example, stress (σ_{rr}, $\sigma_{\theta\theta}$, and $\sigma_{r\theta}$) and displacement (u_r and u_θ) fields around the wellbore are calculated numerically. These numerical predictions are compared with the corresponding Kirsch solutions (Poulos and Davis, 1974).

$$\sigma_{rr} = \frac{\sigma_{xx} + \sigma_{yy}}{2}[1 - (\frac{a_0}{r})^2] + \frac{\sigma_{xx} - \sigma_{yy}}{2}[1 - 4(\frac{a_0}{r})^2 + 3(\frac{a_0}{r})^4]\cos{(2\theta)} + P_w(\frac{a_0}{r})^2$$

$$\sigma_{\theta\theta} = \frac{\sigma_{xx} + \sigma_{yy}}{2}[1 + (\frac{a_0}{r})^2] - \frac{\sigma_{xx} - \sigma_{yy}}{2}[1 + 3(\frac{a_0}{r})^4]\cos{(2\theta)} - P_w(\frac{a_0}{r})^2$$

$$\sigma_{r\theta} = -\frac{\sigma_{xx} - \sigma_{yy}}{2}[1 + 2(\frac{a_0}{r})^2 - 3(\frac{a_0}{r})^4]\sin{(2\theta)}$$

$$u_r = -\frac{(a_0)^2}{2Gr}[\frac{\sigma_{xx} + \sigma_{yy}}{2} + \frac{\sigma_{xx} - \sigma_{yy}}{2}(4(1-\nu) - (\frac{a_0}{r})^2)\cos{(2\theta)} - P_w]$$

$$u_\theta = \frac{(a_0)^2}{2Gr}\frac{\sigma_{xx} - \sigma_{yy}}{2}[2(1-2\nu) + (\frac{a_0}{r})^2]\sin{(2\theta)}$$

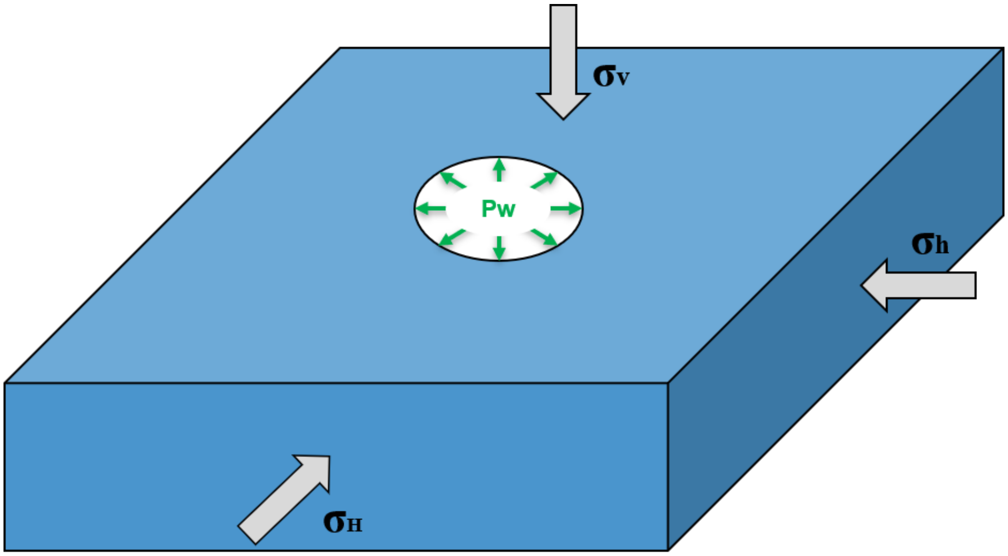


Fig. 1.39: Sketch of the wellbore problem

where a_0 is the intiial wellbore radius, r is the radial coordinate, ν is the Poisson's ratio, G is the shear modulus, P_w is the normal traction acting on the wellbore wall, the angle θ is measured with respect to x-z plane and defined as positive in counter-clockwise direction.

In this example, we focus our attention on the `Mesh`, the `Constitutive`, and the `FieldSpecifications` tags.

Mesh

Following figure shows the generated mesh that is used for solving this wellbore problem.

Let us take a closer look at the geometry of this wellbore problem. We use the internal wellbore mesh generator `InternalWellbore` to create a rock domain $(10\,m \ \times 5\,m \ \times 2\,m)$, with a wellbore of initial radius equal to 0.1 m. Only half of the domain is modeled by a `theta` angle from 0 to 180, assuming symmetry for the rest of the domain. Coordinates of `trajectory` defines the wellbore trajectory, a vertical well in this example. By turning on `autoSpaceRadialElems="{ 1 }"`, the internal mesh generator automatically sets number and spacing of elements in the radial direction, which overrides the values of `nr`. With `useCartesianOuterBoundary="0"`, a Cartesian aligned boundary condition is enforced on the outer blocks. This way, a structured three-dimensional mesh is created with 50 x 40 x 2 elements in the radial, tangential and z directions, respectively. All elements are eight-node hexahedral elements (`C3D8`) and refinement is performed to conform with the wellbore geometry. This mesh is defined as a cell block with the name `cb1`.

```
  <Mesh>
    <InternalWellbore
      name="mesh1"
      elementTypes="{ C3D8 }"
      radius="{ 0.1, 5.0 }"
      theta="{ 0, 180 }"
      zCoords="{ -1, 1 }"
      nr="{ 40 }"
      nt="{ 40 }"
      nz="{ 2 }"
      trajectory="{ { 0.0, 0.0, -1.0 },
```

(continues on next page)

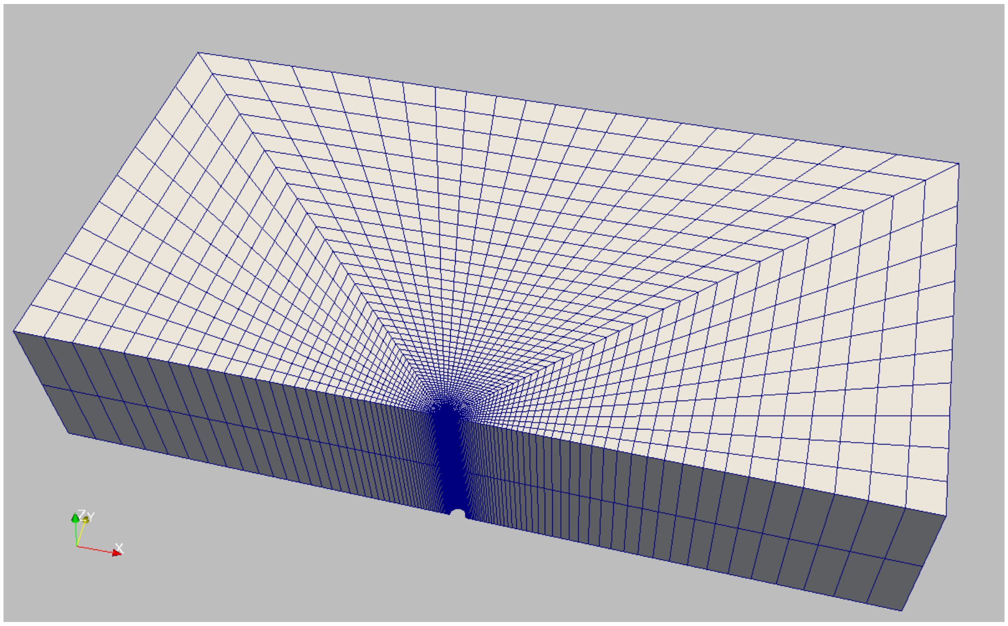


Fig. 1.40: Generated mesh for a vertical wellbore problem

(continued from previous page)

```
                    { 0.0, 0.0, 1.0 } }"
      autoSpaceRadialElems="{ 1 }"
      useCartesianOuterBoundary="0"
      cellBlockNames="{ cb1 }"/>
  </Mesh>
```

Solid mechanics solver

For a drained wellbore problem, the pore pressure variation is omitted. Therefore, we just need to define a solid mechanics solver, which is called `mechanicsSolver`. This solid mechanics solver (see *Solid Mechanics Solver*) is based on the Lagrangian finite element formulation. The problem is run as `QuasiStatic` without considering inertial effects. The computational domain is discretized by `FE1`, which is defined in the `NumericalMethods` section. The material is named `rock`, whose mechanical properties are specified in the `Constitutive` section.

```
  <Solvers gravityVector="{0.0, 0.0, 0.0}">
    <SolidMechanics_LagrangianFEM
      name="mechanicsSolver"
      timeIntegrationOption="QuasiStatic"
      logLevel="1"
      discretization="FE1"
      targetRegions="{Omega}">
      <NonlinearSolverParameters
        newtonTol = "1.0e-5"
        newtonMaxIter = "15"
      />
      <LinearSolverParameters
        directParallel="0"/>
    </SolidMechanics_LagrangianFEM>
```

(continues on next page)

(continued from previous page)

```
</Solvers>
```

Constitutive laws

For this drained wellbore problem, we simulate a linear elastic deformation around the circular cavity. A homogeneous and isotropic domain with one solid material is assumed, with mechanical properties specified in the `Constitutive` section:

```
<Constitutive>
  <ElasticIsotropic
    name="rock"
    defaultDensity="2700"
    defaultBulkModulus="5.0e8"
    defaultShearModulus="3.0e8"
  />
</Constitutive>
```

Recall that in the `SolidMechanics_LagrangianFEM` section, `rock` is the material in the computational domain. Here, the isotropic elastic model `ElasticIsotropic` simulates the mechanical behavior of `rock`.

The constitutive parameters such as the density, the bulk modulus, and the shear modulus are specified in the International System of Units.

Time history function

In the `Tasks` section, `PackCollection` tasks are defined to collect time history information from fields. Either the entire field or specified named sets of indices in the field can be collected. In this example, `stressCollection` and `displacementCollection` tasks are specified to output the resultant stresses (tensor stored as an array with Voigt notation) and total displacement field (stored as a 3-component vector) respectively.

```
<Tasks>
  <PackCollection
    name="stressCollection"
    objectPath="ElementRegions/Omega/cb1"
    fieldName="rock_stress"/>

  <PackCollection
    name="displacementCollection"
    objectPath="nodeManager"
    fieldName="totalDisplacement"/>
</Tasks>
```

These two tasks are triggered using the `Event` management, where `PeriodicEvent` are defined for these recurring tasks. GEOS writes two files named after the string defined in the `filename` keyword and formatted as HDF5 files (displacement_history.hdf5 and stress_history.hdf5). The TimeHistory file contains the collected time history information from each specified time history collector. This information includes datasets for the simulation time, element center or nodal position, and the time history information. Then, a Python script is prepared to access and plot any specified subset of the time history data for verification and visualization.

Initial and boundary conditions

The next step is to specify fields, including:

- The initial value (the in-situ stresses and traction at the wellbore wall have to be initialized),
- The boundary conditions (constraints of the outer boundaries have to be set).

Here, we specify anisotropic horizontal stress values (σ_{yy} = -9.0 MPa and σ_{xx} = -11.25 MPa) and a vertical stress (σ_{zz} = -15.0 MPa). A compressive traction (`WellLoad`) P_w = -2.0 MPa is loaded at the wellbore wall `rneg`. The remaining parts of the outer boundaries are subjected to roller constraints. These boundary conditions are set in the `FieldSpecifications` section.

```
<FieldSpecifications>
  <FieldSpecification
    name="Sxx"
    initialCondition="1"
    setNames="{ all }"
    objectPath="ElementRegions"
    fieldName="rock_stress"
    component="0"
    scale="-11.25e6"
  />

  <FieldSpecification
    name="Syy"
    initialCondition="1"
    setNames="{ all }"
    objectPath="ElementRegions"
    fieldName="rock_stress"
    component="1"
    scale="-9.0e6"
  />

  <FieldSpecification
    name="Szz"
    initialCondition="1"
    setNames="{ all }"
    objectPath="ElementRegions"
    fieldName="rock_stress"
    component="2"
    scale="-15.0e6"
  />

  <Traction
    name="WellLoad"
    setNames="{ rneg }"
    objectPath="faceManager"
    scale="-2.0e6"
    tractionType="normal"
  />

  <FieldSpecification
    name="xconstraint"
    objectPath="nodeManager"
```

(continues on next page)

(continued from previous page)

```
      fieldName="totalDisplacement"
      component="0"
      scale="0.0"
      setNames="{xneg, xpos}"
    />

    <FieldSpecification
      name="yconstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="1"
      scale="0.0"
      setNames="{tneg, tpos, ypos}"
    />

    <FieldSpecification
      name="zconstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="2"
      scale="0.0"
      setNames="{zneg, zpos}"
    />
  </FieldSpecifications>
```

With tractionType="normal", traction is applied to the wellbore wall rneg as a pressure specified as the scalar product of scale scale="-2.0e6" and the outward face normal vector. In this case, the loading magnitude of the traction does not change with time.

You may note :

- All initial value fields must have initialCondition field set to 1;
- The setName field points to the previously defined set to apply the fields;
- nodeManager and faceManager in the objectPath indicate that the boundary conditions are applied to the element nodes and faces, respectively;
- fieldName is the name of the field registered in GEOS;
- Component 0, 1, and 2 refer to the x, y, and z direction, respectively;
- And the non-zero values given by scale indicate the magnitude of the loading;
- Some shorthand, such as xneg and xpos, are used as the locations where the boundary conditions are applied in the computational domain. For instance, xneg means the face of the computational domain located at the left-most extent in the x-axis, while xpos refers to the face located at the right-most extent in the x-axis. Similar shorthands include ypos, yneg, zpos, and zneg;
- The mud pressure loading and in situ stresses have negative values due to the negative sign convention for compressive stress in GEOS.

The parameters used in the simulation are summarized in the following table.

Symbol	Parameter	Unit	Value
K	Bulk Modulus	[MPa]	500.0
G	Shear Modulus	[MPa]	300.0
σ_{yy}	Min Horizontal Stress	[MPa]	-9.0
σ_{xx}	Max Horizontal Stress	[MPa]	-11.25
σ_{zz}	Vertical Stress	[MPa]	-15.0
a_0	Initial Well Radius	[m]	0.1
P_w	Traction at Well	[MPa]	-2.0

Inspecting results

In the above examples, we request VTK output files that can be imported into Paraview to visualize the outcome. The following figure shows the distribution of σ_{xx} in the near wellbore region.

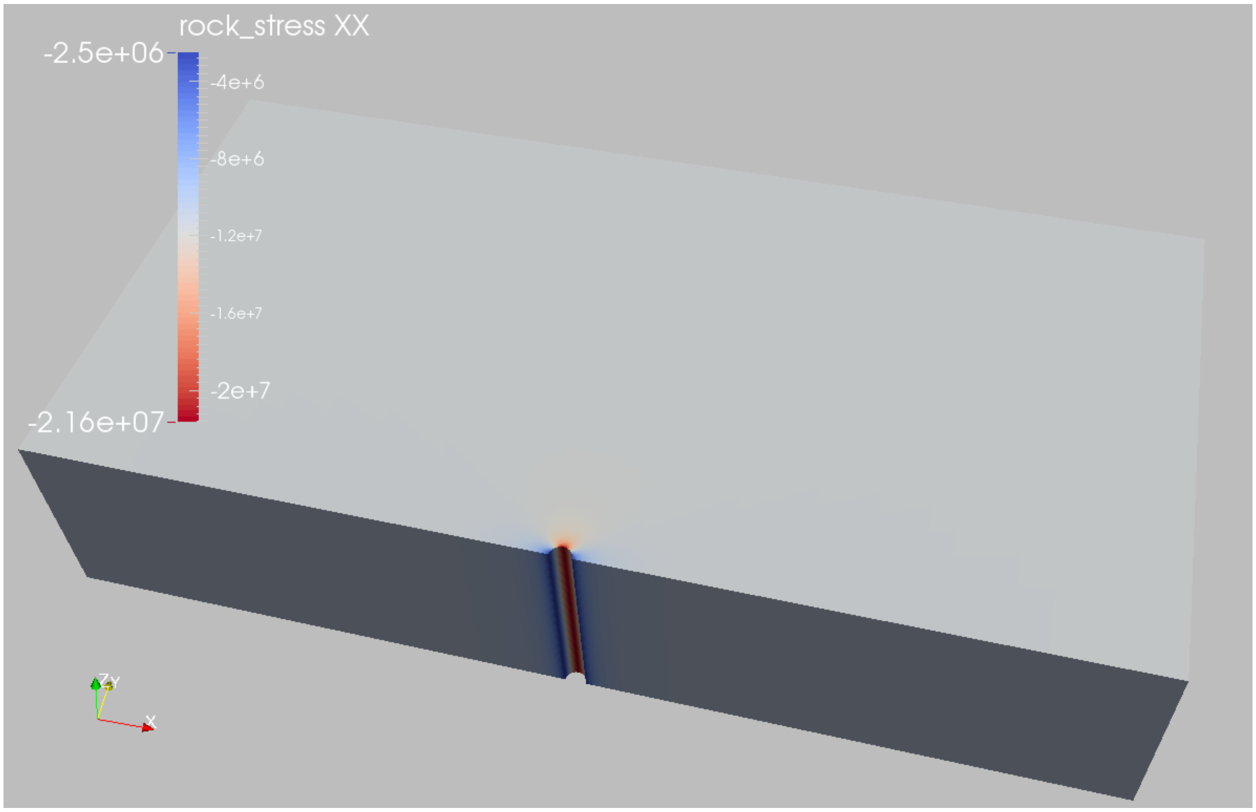
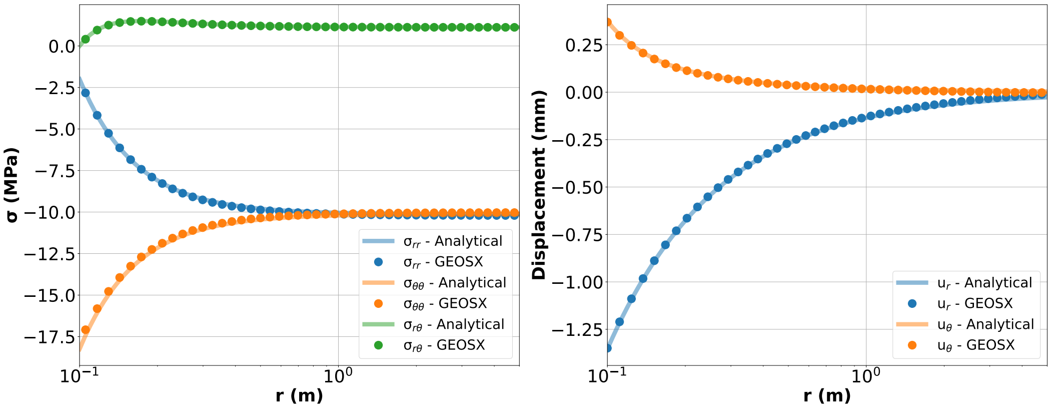


Fig. 1.41: Simulation result of σ_{xx}

We use time history function to collect time history information and run a Python script to query and plot the results. The figure below shows the comparisons between the numerical predictions (marks) and the corresponding analytical solutions (solid curves) with respect to the distributions of stress components and displacement at θ = 45 degrees. Predictions computed by GEOS match the analytical results.



To go further

Feedback on this example

For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

Cased Elastic Wellbore Problem

Problem description

This example uses the solid mechanics solver to handle a cased wellbore problem subjected to a pressure test. The completed wellbore is composed of a steel casing, a cement sheath and rock formation. Isotropic linear elastic behavior is assumed for all the three materials. No separation is allowed for the casing-cement and cement-rock contact interfaces.

Analytical results of the radial and hoop stresses, σ_{rr} and $\sigma_{\theta\theta}$, in casing, cement sheath and rock are expressed as (Hervé and Zaoui, 1995) :

$$\sigma_{rr} = (2\lambda + 2G)A - \frac{2GB}{r^2}$$

$$\sigma_{\theta\theta} = (2\lambda + 2G)A + \frac{2GB}{r^2}$$

where λ and G are the Lamé moduli, r is the radial coordinate, A and B are piecewise constants that are obtained by solving the boundary and interface conditions, as detailed in the post-processing script.

Input file

This benchmark example uses no external input files and everything required is contained within two GEOS xml files that are located at:

```
inputFiles/wellbore/CasedElasticWellbore_base.xml
```

and

```
inputFiles/wellbore/CasedElasticWellbore_benchmark.xml
```

The corresponding integrated test is

```
inputFiles/wellbore/CasedElasticWellbore_smoke.xml
```

In this example, we would focus our attention on the `Solvers`, `Mesh` and `Constitutive` tags.

Solid mechanics solver

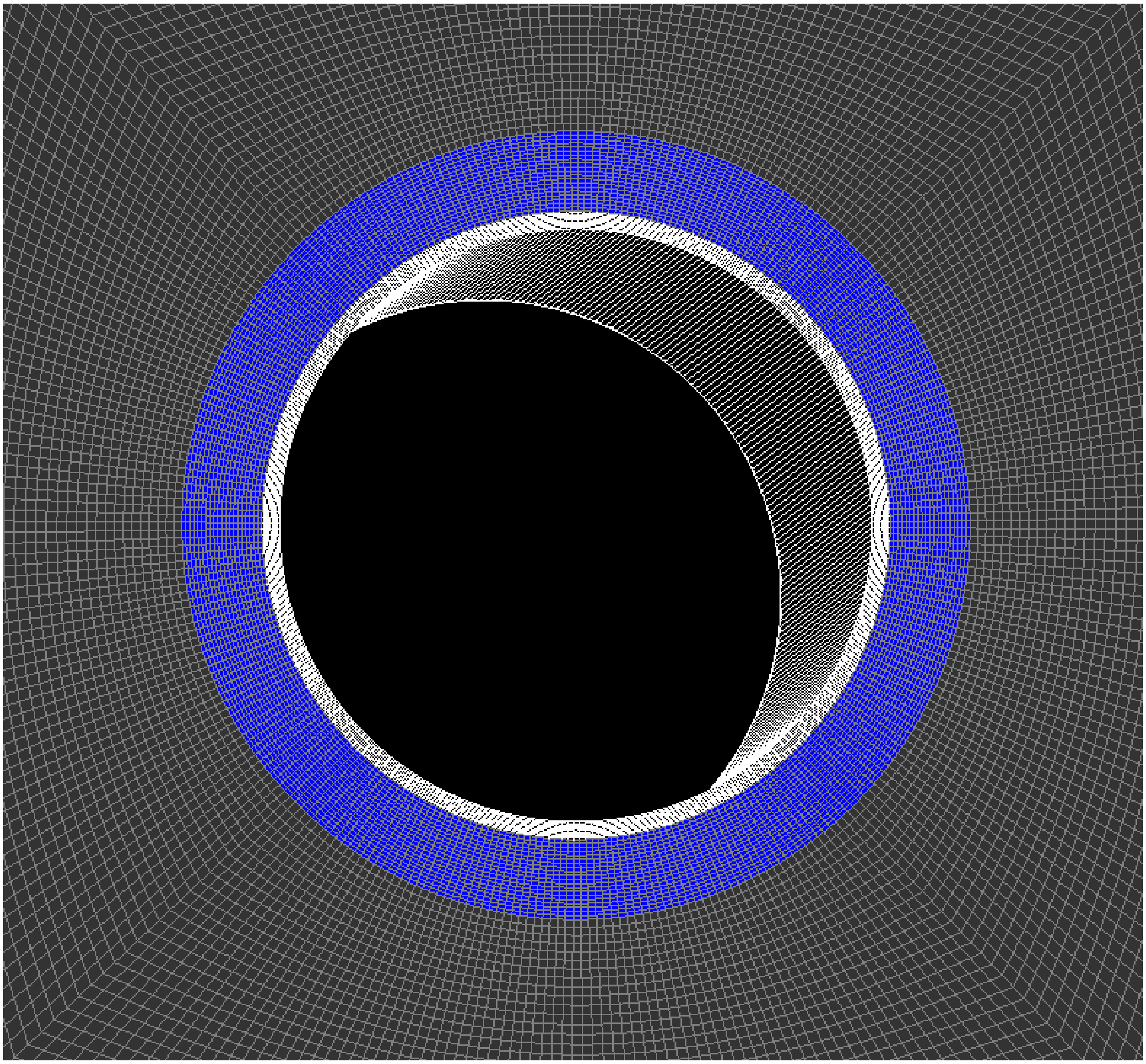
As fluid flow is not considered, only the solid mechanics `SolidMechanicsLagrangianSSLE` solver is required for solving this linear elastic problem. In this solver, the three regions and three materials associated to casing, cement sheath and rock are respectively defined by `targetRegions` and `solidMaterialNames`.

```
    <SolidMechanicsLagrangianSSLE
      name="lagsolve"
      timeIntegrationOption="QuasiStatic"
      discretization="FE1"
      logLevel="0"
      targetRegions="{ casing, cement, rock }">
```

Cased wellbore mesh

The internal wellbore mesh generator `InternalWellbore` is employed to create the mesh of this wellbore problem. The radii of the casing cylinder, the cement sheath cylinder and the far-field boundary of the surrounding rock formation are defined by a vector `radius`. In the tangent direction, `theta` angle is specified from 0 to 360 degree for a full geometry of the domain. Note that a half or a quarter of the domain can be defined by a `theta` angle from 0 to 180 or 90 degree, respectively. The trajectory of the well is defined by `trajectory`, which is vertical in this case. The `autoSpaceRadialElems` parameters allow optimally increasing the element size from local zone around the wellbore to the far-field zone. In this example, the auto spacing option is only applied for the rock formation. The `useCartesianOuterBoundary` transforms the far-field boundary to a squared shape to enforce a Cartesian aligned outer boundary, which eases the loading of the boundary conditions. The `cellBlockNames` and `elementTypes` define the regions and related element types associated to casing, cement sheath and rock.

```
  <Mesh>
    <InternalWellbore
      name="mesh1"
      elementTypes="{ C3D8, C3D8, C3D8 }"
      radius="{ 0.1, 0.106, 0.133, 2.0 }"
      theta="{ 0, 360 }"
      zCoords="{ 0, 1 }"
      nr="{ 10, 20, 10 }"
      nt="{ 320 }"
      nz="{ 1 }"
      trajectory="{ { 0.0, 0.0, 0.0 },
                    { 0.0, 0.0, 1.0 } }"
      autoSpaceRadialElems="{ 0, 0, 1 }"
      useCartesianOuterBoundary="2"
      cellBlockNames="{ casing, cement, rock }"
      />
  </Mesh>
```



Steel, cement, and rock constitutive laws

Isotropic linear elastic constitutive behavior is considered for all the three materials. Note that the default density is useless for this case.

```
    <ElasticIsotropic
      name="casing"
      defaultDensity="2700"
      defaultBulkModulus="175e9"
      defaultShearModulus="80.8e9"/>

    <ElasticIsotropic
      name="cement"
      defaultDensity="2700"
      defaultBulkModulus="10.3e9"
      defaultShearModulus="6.45e9"/>

    <ElasticIsotropic
      name="rock"
      defaultDensity="2700"
      defaultBulkModulus="5.5556e9"
      defaultShearModulus="4.16667e9"/>
```

Boundary conditions

Far-field boundary are subjected to roller constraints. The normal traction on the inner face of the casing is defined by `Traction` field specification. The nodeset generated by the internal wellbore generator for this face is named as `rneg`. The traction type is `normal` to mimic a casing test pressure that is applied normal to the casing inner face . The negative sign of the scale is attributed to the negative sign convention for compressive stress in GEOS.

```
  <FieldSpecifications>

    <FieldSpecification
      name="xConstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="0"
      scale="0.0"
      setNames="{ xneg, xpos }"/>

    <FieldSpecification
      name="yConstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="1"
      scale="0.0"
      setNames="{ yneg, ypos }"/>

    <FieldSpecification
      name="zconstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
```

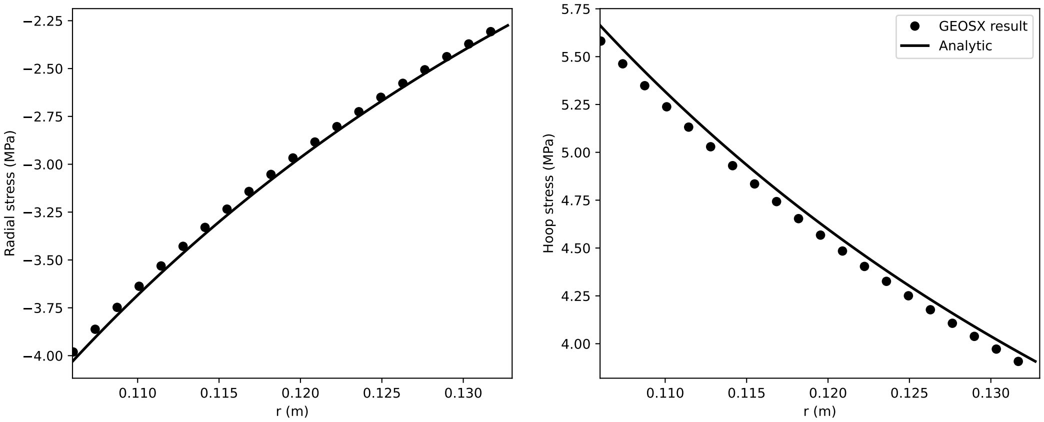
(continues on next page)

(continued from previous page)

```
      component="2"
      scale="0.0"
      setNames="{ zneg, zpos }"/>

    <Traction
      name="innerPressure"
      objectPath="faceManager"
      tractionType="normal"
      scale="-10.0e6"
      setNames="{ rneg }"/>
  </FieldSpecifications>
```

Results and benchmark

A good agreement between the GEOS results and analytical results is shown in the figure below:



To go further

Feedback on this example

This concludes the cased wellbore example. For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

Deviated Elastic Wellbore Problem

Problem description

This example uses the solid mechanics solver to handle a deviated wellbore problem with open hole completion. This wellbore is subjected to a mud pressure at wellbore wall and undrained condition is assumed (no fluid flow in the rock formation). A segment of the wellbore with isotropic linear elastic deformation is simulated in this case. Far field stresses and gravity effect are excluded. The main goal of this example is to validate the internal wellbore mesh generator and mechanics solver for the case of an inclined wellbore.

Analytical results of the radial and hoop stresses, σ_{rr} and $\sigma_{\theta\theta}$, around the wellbore are expressed as (Detournay and Cheng, 1988) :

$$\sigma_{rr} = p_0 \frac{a^2}{r^2}$$

$$\sigma_{\theta\theta} = -p_0 \frac{a^2}{r^2}$$

where p_0 is the applied mud pressure at wellbore wall, a is the wellbore radius and r is the radial coordinate.

Input file

This benchmark example uses no external input files and everything required is contained within two GEOS xml files that are located at:

```
inputFiles/wellbore/DeviatedElasticWellbore_base.xml
```

and

```
inputFiles/wellbore/DeviatedElasticWellbore_benchmark.xml
```

The corresponding xml file for the integrated test is

```
inputFiles/wellbore/DeviatedElasticWellbore_smoke.xml
```

In this example, we would focus our attention on the `Mesh` tag.

Solid mechanics solver

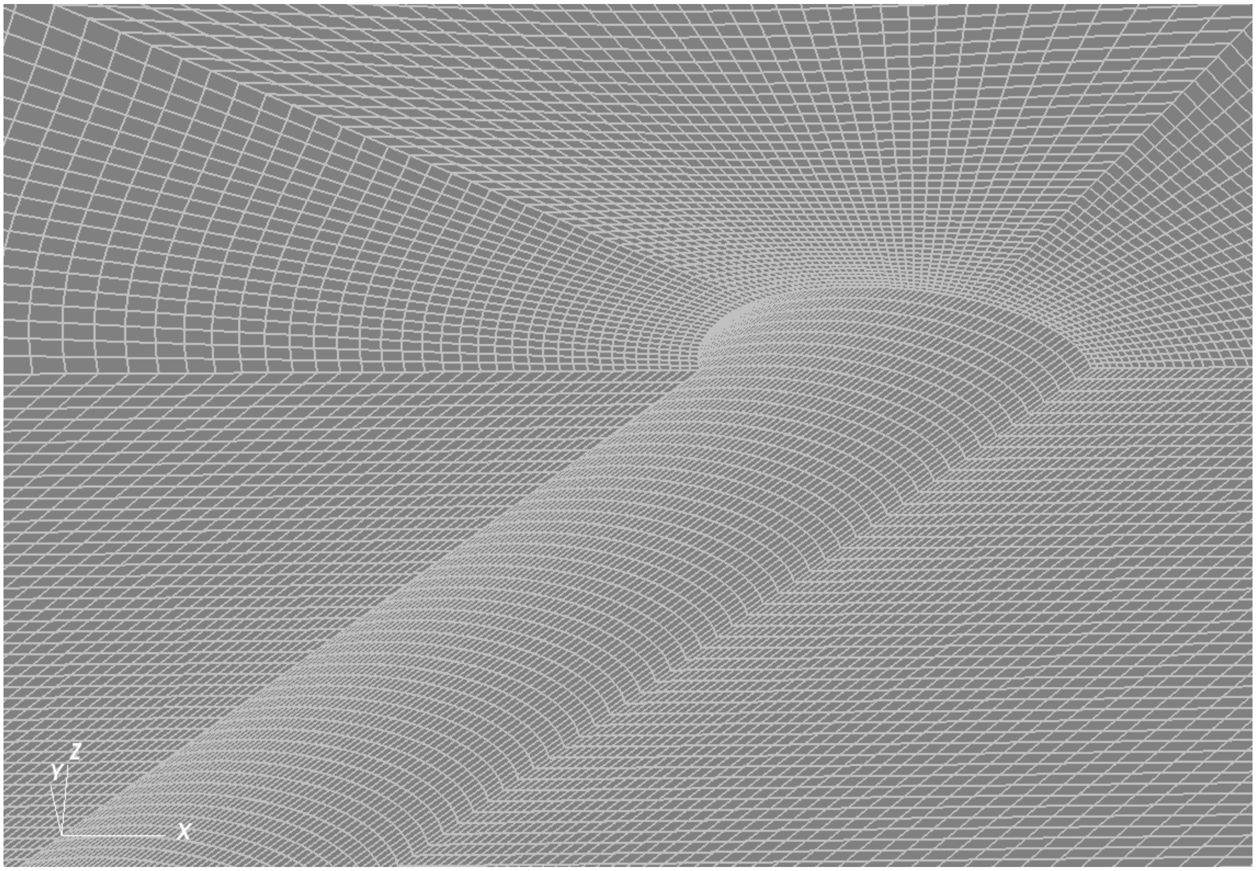
As fluid flow is not considered, only the solid mechanics solver `SolidMechanicsLagrangianSSLE` is required for solving this wellbore problem.

```
    <SolidMechanicsLagrangianSSLE
      name="lagsolve"
      timeIntegrationOption="QuasiStatic"
      discretization="FE1"
      logLevel="0"
      targetRegions="{ Omega }"
      >
```

Deviated wellbore mesh

The internal wellbore mesh generator `InternalWellbore` is employed to create the mesh of this wellbore problem. The radius of the wellbore and the size of the surrounding rock formation are defined by a vector `radius`. In the tangent direction, `theta` angle is specified from 0 to 180 degree for a half of the domain regarding its symmetry. Note that the whole domain could be specified with a `theta` angle from 0 to 360 degree, if modeling complicated scenarios. The trajectory of the well is defined by `trajectory`. In this example, the wellbore is inclined in the x-z plane by an angle of 45 degree. The `autoSpaceRadialElems` parameter allows optimally increasing the element size from local zone around the wellbore to the far-field zone, which is set to 1 to activate this option. The `useCartesianOuterBoundary` transforms the far-field boundary to a squared shape to enforce a Cartesian aligned outer boundary, which eases the loading of the far-field boundary conditions. In this example, this value is set to 0 for the single region along the radial direction.

```
<Mesh>
  <InternalWellbore
    name="mesh1"
    elementTypes="{ C3D8 }"
    radius="{ 0.1, 2 }"
    theta="{ 0, 180 }"
    zCoords="{ -0.5, 0.5 }"
    nr="{ 30 }"
    nt="{ 80 }"
    nz="{ 100 }"
    trajectory="{ { -0.5, 0.0, -0.5 },
                  {  0.5, 0.0,  0.5 } }"
    autoSpaceRadialElems="{ 1 }"
    useCartesianOuterBoundary="0"
    cellBlockNames="{ cb1 }"/>
</Mesh>
```



Constitutive law

Isotropic linear elastic constitutive behavior is considered for the rock around the wellbore. Note that the default density is useless in this specific example, as gravity effect is neglected.

```
  <ElasticIsotropic
    name="shale"
    defaultDensity="2700"
    defaultBulkModulus="5.5556e9"
    defaultShearModulus="4.16667e9"/>
```

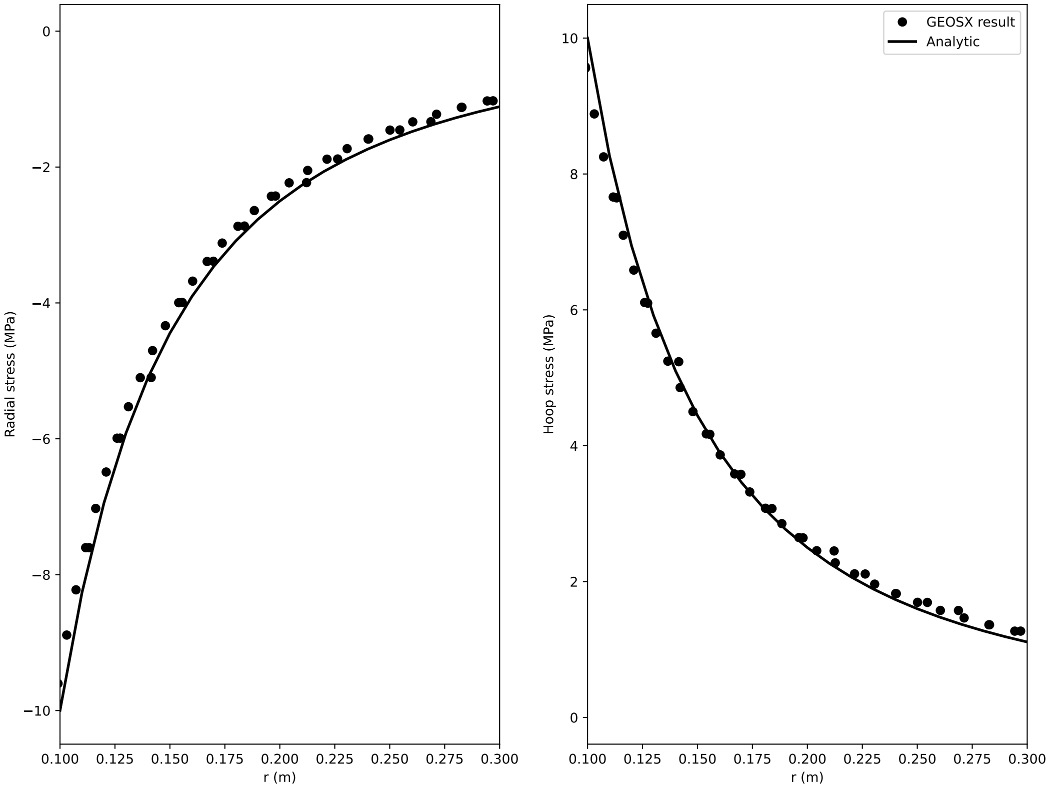
Boundary conditions

Far-field boundaries are subjected to roller constraints and in-situ stresses are not considered. The mud pressure on the wellbore wall is defined by `Traction` field specification. The nodeset generated by the internal wellbore generator for this face is named as `rneg`. The traction type is `normal` to mimic a pressure that is applied normal to the wellbore wall. The negative sign of the scale is attributed to the negative sign convention for compressive stresses in GEOS.

```
<FieldSpecifications>

  <FieldSpecification
    name="xConstraint"
    objectPath="nodeManager"
    fieldName="totalDisplacement"
    component="0"
    scale="0.0"
    setNames="{ xneg }"/>

  <FieldSpecification
    name="yConstraint"
    objectPath="nodeManager"
    fieldName="totalDisplacement"
    component="1"
    scale="0.0"
    setNames="{ tneg, tpos }"/>

  <FieldSpecification
    name="zconstraint"
    objectPath="nodeManager"
    fieldName="totalDisplacement"
    component="2"
    scale="0.0"
    setNames="{ zneg }"/>

  <Traction
    name="innerPressure"
    objectPath="faceManager"
    tractionType="normal"
    scale="-10.e6"
    setNames="{ rneg }"/>
</FieldSpecifications>
```

Results and benchmark

A good agreement between the GEOS results and the corresponding analytical solutions is shown in the figure below:



To go further

Feedback on this example

This concludes the deviated elastic wellbore example. For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

Extended Drucker-Prager Model for Wellbore Problems

Context

The main goal of this example is to learn how to use the internal wellbore mesh generator and an elasto-plastic model to handle wellbore problems in GEOS. The Extended Drucker-Prager model (see *Model: Extended Drucker-Prager*) is applied to solve for elastoplastic deformation within the vicinity of a vertical wellbore. For the presented example, an analytical solution is employed to verify the accuracy of the numerical results. The resulting model can be used as a base for more complex analysis (e.g., wellbore drilling, fluid injection and storage scenarios).

Objectives

At the end of this example you will know:

- how to construct meshes for wellbore problems with the internal mesh generator,
- how to specify initial and boundary conditions, such as in-situ stresses and variation of traction at the wellbore wall,
- how to use a plastic model for mechanical problems in the near wellbore region.

Input file

This example uses no external input files and everything required is contained within two xml files that are located at:

```
inputFiles/solidMechanics/ExtendedDruckerPragerWellbore_base.xml
```

```
inputFiles/solidMechanics/ExtendedDruckerPragerWellbore_benchmark.xml
```

The Python scripts for post-processing GEOS results, analytical restuls and validation plots are also provided in this example.

Description of the case

We simulate a drained wellbore problem subjected to isotropic horizontal stress (σ_h) and vertical stress (σ_v). By lowering the wellbore supporting pressure (P_w), the wellbore contracts, and the reservoir rock experiences elastoplastic deformation. A plastic zone develops in the near wellbore region, as shown below.

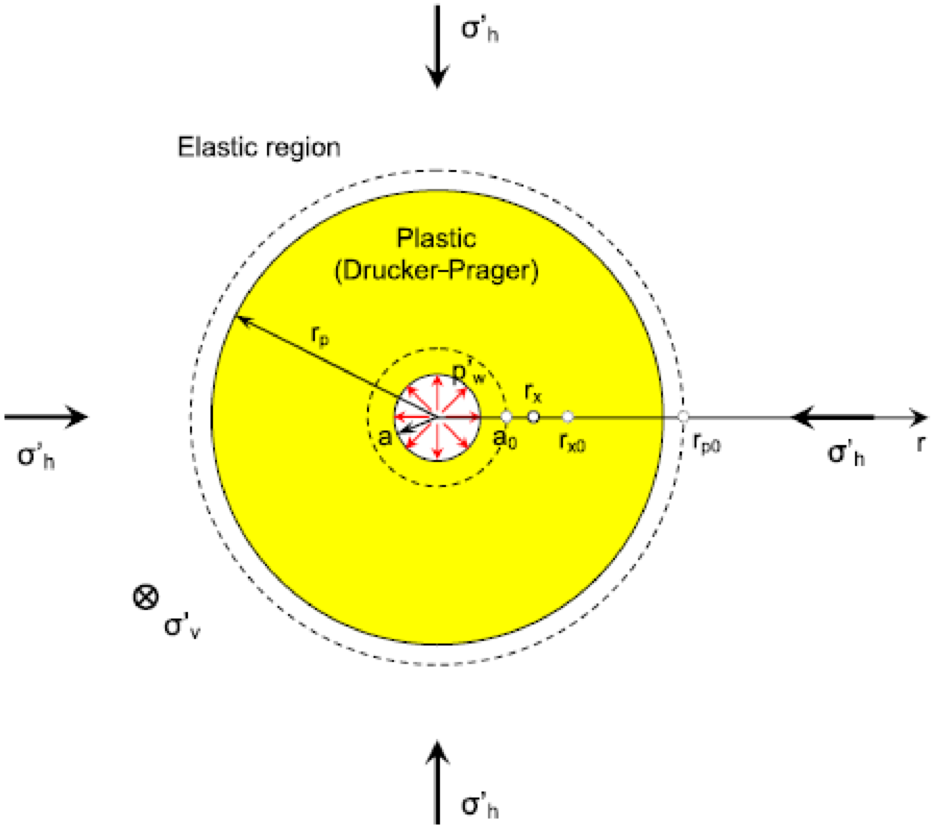


Fig. 1.42: Sketch of the wellbore problem (Chen and Abousleiman, 2017)

To simulate this phenomenon, the strain hardening Extended Drucker-Prager model with an associated plastic flow rule in GEOS is used in this example. Displacement and stress fields around the wellbore are numerically calculated. These numerical predictions are then compared with the corresponding analytical solutions (Chen and Abousleiman, 2017) from the literature.

All inputs for this case are contained inside a single XML file. In this example, we focus our attention on the `Mesh` tags, the `Constitutive` tags, and the `FieldSpecifications` tags.

Mesh

Following figure shows the generated mesh that is used for solving this 3D wellbore problem

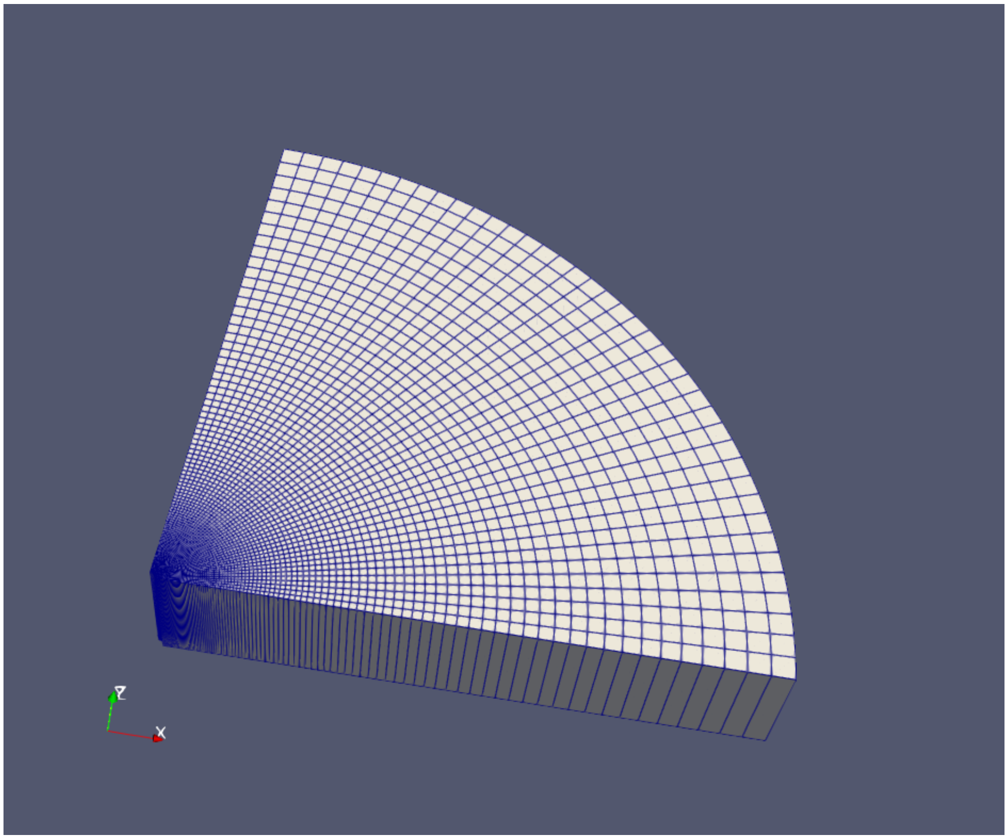


Fig. 1.43: Generated mesh for the wellbore problem

Let us take a closer look at the geometry of this wellbore problem. We use the internal mesh generator `InternalWellbore` to create a rock domain ($10\,m \ \times 10\,m \ \times 2\,m$), with a wellbore of initial radius equal to 0.1 m. Coordinates of `trajectory` defines the wellbore trajectory, which represents a vertical well in this example. By turning on `autoSpaceRadialElems="{ 1 }"`, the internal mesh generator automatically sets number and spacing of elements in the radial direction, which overrides the values of `nr`. In this way, a structured three-dimensional mesh is created. All the elements are eight-node hexahedral elements (`C3D8`) and refinement is performed to conform with the wellbore geometry. This mesh is defined as a cell block with the name `cb1`.

```
<Mesh>
  <InternalWellbore
    name="mesh1"
    elementTypes="{ C3D8 }"
```

(continues on next page)

(continued from previous page)

```
    radius="{ 0.1, 10.0 }"
    theta="{ 0, 90 }"
    zCoords="{ -1, 1 }"
    nr="{ 40 }"
    nt="{ 40 }"
    nz="{ 1 }"
    trajectory="{ { 0.0, 0.0, -1.0 },
                  { 0.0, 0.0, 1.0 } }"
    autoSpaceRadialElems="{ 1 }"
    cellBlockNames="{ cb1 }"/>
</Mesh>
```

Solid mechanics solver

For the drained wellbore problem, the pore pressure variation is omitted and can be subtracted from the analysis. Therefore, we just need to define a solid mechanics solver, which is called `mechanicsSolver`. This solid mechanics solver (see *Solid Mechanics Solver*) is based on the Lagrangian finite element formulation. The problem is run as `QuasiStatic` without considering inertial effects. The computational domain is discretized by `FE1`, which is defined in the `NumericalMethods` section. The material is named as `rock`, whose mechanical properties are specified in the `Constitutive` section.

```
<Solvers
  gravityVector="{ 0.0, 0.0, 0.0 }">
  <SolidMechanics_LagrangianFEM
    name="mechanicsSolver"
    timeIntegrationOption="QuasiStatic"
    logLevel="1"
    discretization="FE1"
    targetRegions="{ Omega }"
    >
    <LinearSolverParameters
      directParallel="0"/>
    <NonlinearSolverParameters
      newtonTol="1.0e-5"
      newtonMaxIter="15"/>
  </SolidMechanics_LagrangianFEM>
</Solvers>
```

Constitutive laws

For this drained wellbore problem, we simulate the elastoplastic deformation caused by wellbore contraction. A homogeneous domain with one solid material is assumed, whose mechanical properties are specified in the `Constitutive` section:

```
<Constitutive>
  <ExtendedDruckerPrager
    name="rock"
    defaultDensity="2700"
    defaultBulkModulus="0.5e9"
```

(continues on next page)

(continued from previous page)

```
    defaultShearModulus="0.3e9"
    defaultCohesion="0.0"
    defaultInitialFrictionAngle="15.27"
    defaultResidualFrictionAngle="23.05"
    defaultDilationRatio="1.0"
    defaultHardening="0.01"/>

</Constitutive>
```

Recall that in the `SolidMechanics_LagrangianFEM` section, `rock` is designated as the material in the computational domain. Here, Extended Drucker Prager model `ExtendedDruckerPrager` is used to simulate the elastoplastic behavior of `rock`. As for the material parameters, `defaultInitialFrictionAngle`, `defaultResidualFrictionAngle` and `defaultCohesion` denote the initial friction angle, the residual friction angle, and cohesion, respectively, as defined by the Mohr-Coulomb failure envelope. In this example, zero cohesion is considered to consist with the reference analytical results. As the residual friction angle `defaultResidualFrictionAngle` is larger than the initial one `defaultInitialFrictionAngle`, a strain hardening model is adopted, whose hardening rate is given as `defaultHardening="0.01"`. If the residual friction angle is set to be less than the initial one, strain weakening will take place. Setting `defaultDilationRatio="1.0"` corresponds to an associated flow rule. The constitutive parameters such as the density, the bulk modulus, and the shear modulus are specified in the International System of Units.

Initial and boundary conditions

The next step is to specify fields, including:

- The initial value (the in-situ stresses and traction at the wellbore wall have to be initialized)
- The boundary conditions (the reduction of wellbore pressure and constraints of the outer boundaries have to be set)

In this example, we need to specify isotropic horizontal stress (σ_h = -11.25 MPa) and vertical stress (σ_v = -15.0 MPa). To reach equilibrium, a compressive traction p_w = -11.25 MPa is instantaneously applied at the wellbore wall `rneg` at time t = 0 s, which will then be gradually reduced to a lower absolute value (-2.0 MPa) to let wellbore contract. The boundaries at `tneg` and `tpos` are subjected to roller constraints. The plane strain condition is ensured by fixing the vertical displacement at `zneg` and `zpos` The far-field boundary is fixed in horizontal displacement. These boundary conditions are set up through the `FieldSpecifications` section.

```
<FieldSpecifications>
  <FieldSpecification
    name="stressXX"
    initialCondition="1"
    setNames="{ all }"
    objectPath="ElementRegions/Omega/cb1"
    fieldName="rock_stress"
    component="0"
    scale="-11.25e6"/>

  <FieldSpecification
    name="stressYY"
    initialCondition="1"
    setNames="{ all }"
    objectPath="ElementRegions/Omega/cb1"
    fieldName="rock_stress"
```

(continues on next page)

(continued from previous page)

```
      component="1"
      scale="-11.25e6"/>

    <FieldSpecification
      name="stressZZ"
      initialCondition="1"
      setNames="{ all }"
      objectPath="ElementRegions/Omega/cb1"
      fieldName="rock_stress"
      component="2"
      scale="-15.0e6"/>

    <Traction
      name="ExternalLoad"
      setNames="{ rneg }"
      objectPath="faceManager"
      scale="1.0"
      tractionType="normal"
      functionName="timeFunction"/>

    <FieldSpecification
      name="xconstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="0"
      scale="0.0"
      setNames="{ tpos, rpos }"/>

    <FieldSpecification
      name="yconstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="1"
      scale="0.0"
      setNames="{ tneg, rpos }"/>

    <FieldSpecification
      name="zconstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="2"
      scale="0.0"
      setNames="{ zneg, zpos }"/>
  </FieldSpecifications>
```

With `tractionType="normal"`, traction is applied to the wellbore wall `rneg` as a pressure specified from the product of scale `scale="1.0"` and the outward face normal. A table function `timeFunction` is used to define the time-dependent traction `ExternalLoad`. The `coordinates` and `values` form a time-magnitude pair for the loading time history. In this case, the loading magnitude decreases linearly as the time evolves.

```
  <Functions>
    <TableFunction
```

(continues on next page)

(continued from previous page)

```
    name="timeFunction"
    inputVarNames="{ time }"
    coordinates="{ 0.0, 1.0, 1e99 }"
    values="{ -11.25e6, -2.0e6, -2.0e6 }"/>
</Functions>
```

You may note :

- All initial value fields must have `initialCondition` field set to 1;
- The `setName` field points to the previously defined box to apply the fields;
- `nodeManager` and `faceManager` in the `objectPath` indicate that the boundary conditions are applied to the element nodes and faces, respectively;
- `fieldName` is the name of the field registered in GEOS;
- Component `0`, `1`, and `2` refer to the x, y, and z direction, respectively;
- And the non-zero values given by `Scale` indicate the magnitude of the loading;
- Some shorthand, such as `tpos` and `xpos`, are used as the locations where the boundary conditions are applied in the computational domain. For instance, `tpos` means the portion of the computational domain located at the left-most in the x-axis, while `xpos` refers to the portion located at the right-most area in the x-axis. Similar shorthand include `ypos`, `tneg`, `zpos`, and `zneg`;
- The mud pressure loading has a negative value due to the negative sign convention for compressive stress in GEOS.

The parameters used in the simulation are summarized in the following table.

Symbol	Parameter	Unit	Value
K	Bulk modulus	[MPa]	500
G	Shear Modulus	[MPa]	300
c	Cohesion	[MPa]	0.0
ϕ_i	Initial Friction Angle	[degree]	15.27
ϕ_r	Residual Friction Angle	[degree]	23.05
m	Hardening Rate	[-]	0.01
σ_h	Horizontal Stress	[MPa]	-11.25
σ_v	Vertical Stress	[MPa]	-15.0
a_0	Initial Well Radius	[m]	0.1
p_w	Mud Pressure	[MPa]	-2.0

Inspecting results

In the above example, we requested hdf5 output files. We can therefore use python scripts to visualize the outcome. Below figure shows the comparisons between the numerical predictions (marks) and the corresponding analytical solutions (solid curves) with respect to the distributions of principal stress components, stress path on the wellbore surface, the supporting wellbore pressure and wellbore size. It is clear that the GEOS predictions are in excellent agreement with the analytical results. On the top-right figure, we added also a comparison between GEOS results for elasto-plastic material and the anlytical solutions of an elastic material. Note that the elastic solutions are differed from the elasto-plastic results even in the elastic zone (r/a>2).

For the same wellbore problem, using different constitutive models (plastic vs. elastic), obviously, distinct differences in rock deformation and distribution of resultant stresses is also observed and highlighted.

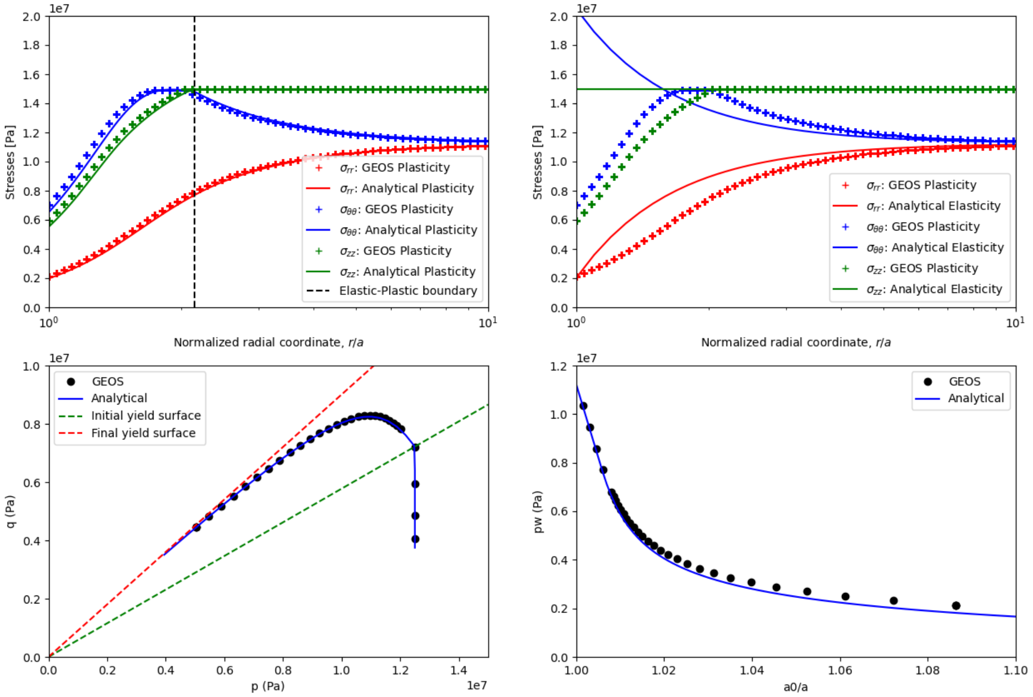


Fig. 1.44: Validation of GEOS results.

To go further

Feedback on this example

This concludes the example on Plasticity Model for Wellbore Problems. For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

For more details

- More on plasticity models, please see *Model: Extended Drucker-Prager*.
- More on functions, please see *Functions*.

Drucker-Prager Model with Hardening for Wellbore Problems

Context

This is an alternative to the example *Extended Drucker-Prager Model for Wellbore Problems*, and the Drucker-Prager constitutive with cohesion hardening (see *Model: Drucker-Prager*) is hereby considered. Analytical solutions to this problem are not provided from literature work, however they can be derived following (Chen and Abousleiman 2017). Details of those solutions are given in Python scripts associated to this example.

Input file

This example uses no external input files and everything required is contained within two xml files that are located at:

```
inputFiles/solidMechanics/DruckerPragerWellbore_base.xml
```

```
inputFiles/solidMechanics/DruckerPragerWellbore_benchmark.xml
```

The related integrated test is

```
inputFiles/solidMechanics/DruckerPragerWellbore_smoke.xml
```

The Drucker-Prager material properties are specified in the `Constitutive` section:

```
  <Constitutive>
    <DruckerPrager
      name="rock"
      defaultDensity="2700"
      defaultBulkModulus="0.5e9"
      defaultShearModulus="0.3e9"
      defaultCohesion="0.1e6"
      defaultFrictionAngle="15.27"
      defaultDilationAngle="15.0"
      defaultHardeningRate="10.0e6"/>

  </Constitutive>
```

Here, `rock` is designated as the material in the computational domain. Drucker Prager model `DruckerPrager` is used to simulate the elastoplastic behavior of `rock`. The material parameters, `defaultFrictionAngle`, `defaultDilationAngle` and `defaultCohesion` denote the friction angle, the dilation angle, and the cohesion, respectively. In this example, the hardening of the cohesion is described by a linear hardening law, which is governed by the parameter `defaultHardeningRate`. The constitutive parameters such as the density, the bulk modulus, and the shear modulus are specified in the International System of Units.

The parameters used in the simulation are summarized in the following table.

Symbol	Parameter	Unit	Value
K	Bulk modulus	[MPa]	500
G	Shear Modulus	[MPa]	300
c	Cohesion	[MPa]	0.1
ϕ	Friction Angle	[degree]	15.27
ψ	Dilation Angle	[degree]	15.0
h	Hardening Rate	[MPa]	10.0
σ_h	Horizontal Stress	[MPa]	-11.25
σ_v	Vertical Stress	[MPa]	-15.0
a_0	Initial Well Radius	[m]	0.1
p_w	Mud Pressure	[MPa]	-2.0

The validation of GEOS results against analytical results is shown in the figure below:

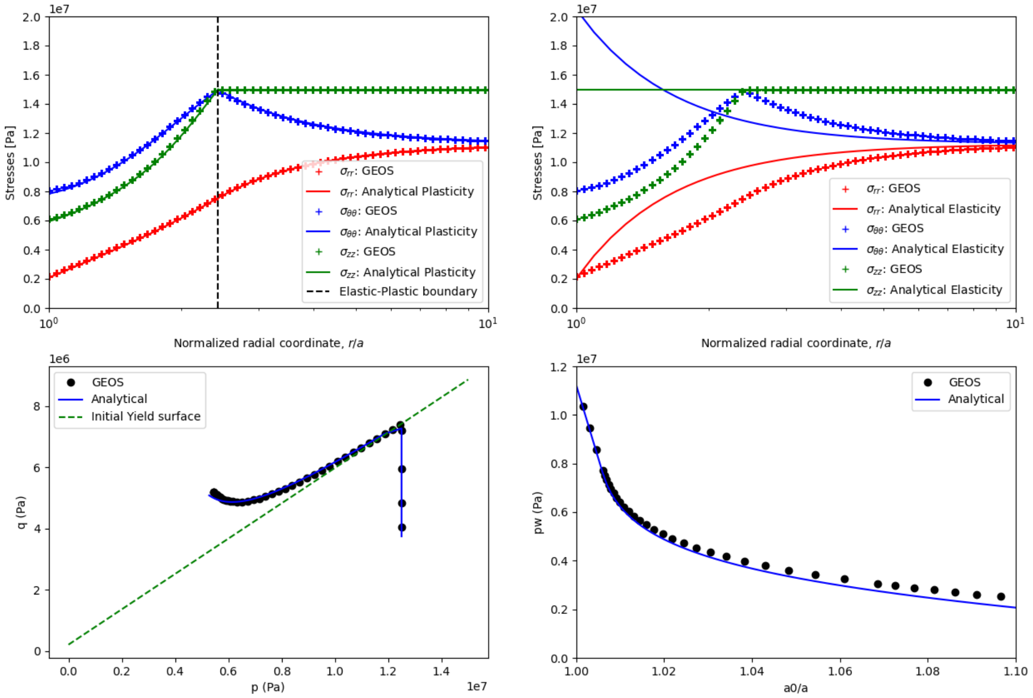


Fig. 1.45: Validation of GEOS results.

To go further

Feedback on this example

This concludes the example on Plasticity Model for Wellbore Problems. For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

Modified Cam-Clay Model for Wellbore Problems

Context

In this benchmark example, the Modified Cam-Clay model (see *Model: Modified Cam-Clay*) is applied to solve for elastoplastic deformation within the vicinity of a vertical wellbore. For the presented example, an analytical solution is employed to verify the accuracy of the numerical results. The resulting model can be used as a base for more complex analysis (e.g., wellbore drilling, fluid injection and storage scenarios).

Input file

Everything required is contained within two GEOS input files located at:

```
inputFiles/solidMechanics/ModifiedCamClayWellbore_base.xml
```

```
inputFiles/solidMechanics/ModifiedCamClayWellbore_benchmark.xml
```

Description of the case

We simulate a drained wellbore problem subjected to isotropic horizontal stress (σ_h) and vertical stress (σ_v), as shown below. By increasing the wellbore supporting pressure (P_w), the wellbore expands, and the formation rock experiences elastoplastic deformation. A plastic zone develops in the near wellbore region.

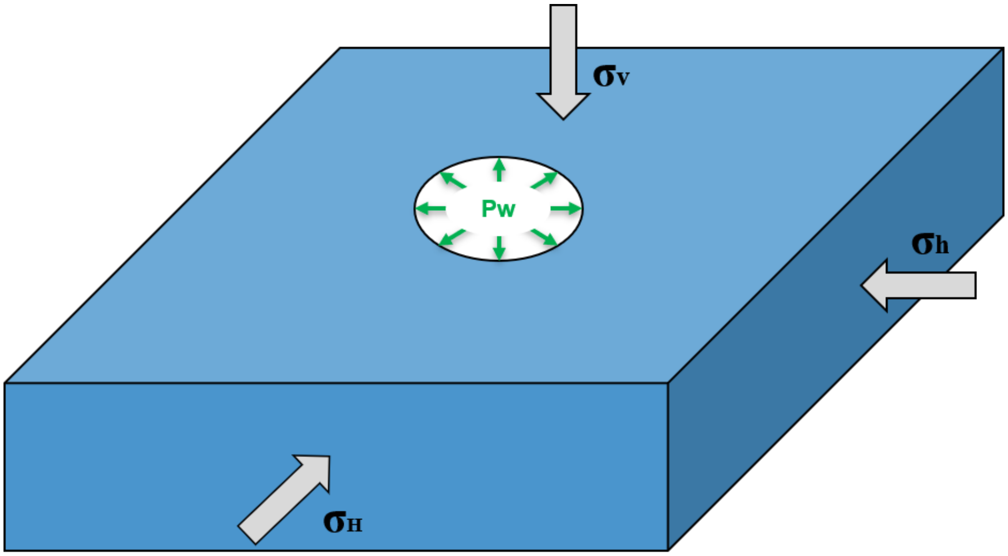


Fig. 1.46: Sketch of the wellbore problem

To simulate this phenomenon, the Modified Cam-Clay model is used in this example. Displacement and stress fields around the wellbore are numerically calculated. These numerical predictions are then compared with the corresponding analytical solutions (Chen and Abousleiman, 2013) from the literature.

In this example, we focus our attention on the `Mesh` tags, the `Constitutive` tags, and the `FieldSpecifications` tags.

Mesh

Following figure shows the generated mesh that is used for solving this wellbore problem.

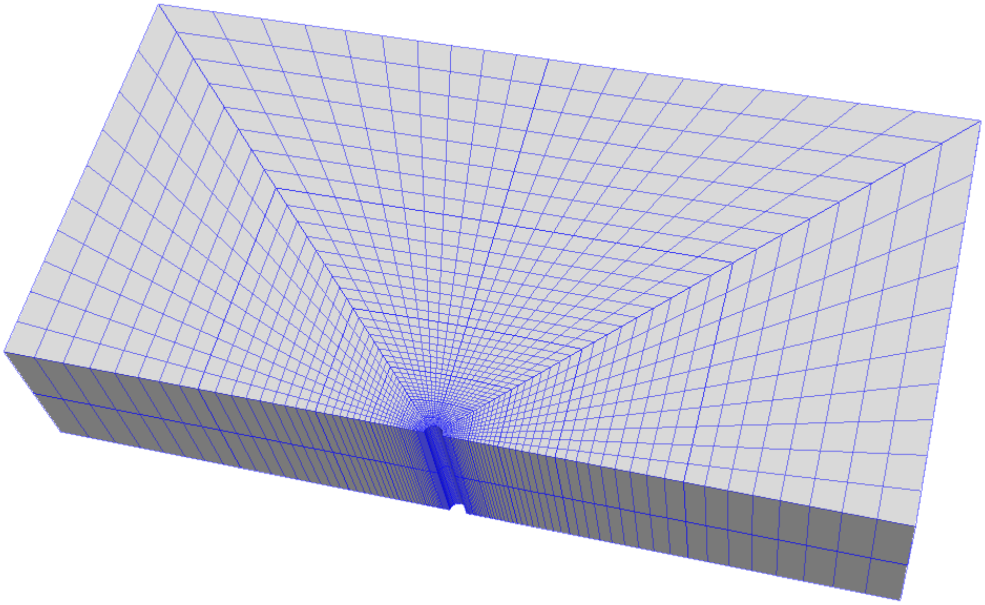


Fig. 1.47: Generated mesh for a vertical wellbore problem

Let us take a closer look at the geometry of this wellbore problem. We use the internal wellbore mesh generator `InternalWellbore` to create a rock domain ($10\,m\ \times 5\,m\ \times 2\,m$), with a wellbore of initial radius equal to 0.1 m. Coordinates of `trajectory` defines the wellbore trajectory, which represents a vertical well in this example. By turning on `autoSpaceRadialElems="{ 1 }"`, the internal mesh generator automatically sets number and spacing of elements in the radial direction, which overrides the values of `nr`. With `useCartesianOuterBoundary="0"`, a Cartesian aligned outer boundary on the outer block is enforced. In this way, a structured three-dimensional mesh is created with 50 x 40 x 2 elements in the radial, tangential and z directions, respectively. All the elements are eight-node hexahedral elements (`C3D8`) and refinement is performed to conform with the wellbore geometry. This mesh is defined as a cell block with the name `cb1`.

```
<Mesh>
  <InternalWellbore
    name="mesh1"
    elementTypes="{ C3D8 }"
    radius="{ 0.1, 5.0 }"
    theta="{ 0, 180 }"
    zCoords="{ -1, 1 }"
    nr="{ 40 }"
    nt="{ 40 }"
    nz="{ 2 }"
    trajectory="{ { 0.0, 0.0, -1.0 },
                  { 0.0, 0.0, 1.0 } }"
```

(continues on next page)

(continued from previous page)

```
    autoSpaceRadialElems="{ 1 }"
    useCartesianOuterBoundary="0"
    cellBlockNames="{ cb1 }"/>
</Mesh>
```

Solid mechanics solver

For the drained wellbore problem, the pore pressure variation is omitted. Therefore, we just need to define a solid mechanics solver, which is called `mechanicsSolver`. This solid mechanics solver (see *Solid Mechanics Solver*) is based on the Lagrangian finite element formulation. The problem is run as `QuasiStatic` without considering inertial effects. The computational domain is discretized by `FE1`, which is defined in the `NumericalMethods` section. The material is named as `rock`, whose mechanical properties are specified in the `Constitutive` section.

```
<Solvers
  gravityVector="{ 0.0, 0.0, 0.0 }">
  <SolidMechanics_LagrangianFEM
    name="mechanicsSolver"
    timeIntegrationOption="QuasiStatic"
    logLevel="1"
    discretization="FE1"
    targetRegions="{ Omega }"
    >
    <LinearSolverParameters
    directParallel="0"/>
    <NonlinearSolverParameters
      newtonTol="1.0e-5"
      newtonMaxIter="15"/>
  </SolidMechanics_LagrangianFEM>
</Solvers>
```

Constitutive laws

For this drained wellbore problem, we simulate the elastoplastic deformation caused by wellbore expansion. A homogeneous domain with one solid material is assumed, whose mechanical properties are specified in the `Constitutive` section:

```
<Constitutive>
  <ModifiedCamClay
    name="rock"
    defaultDensity="2700"
    defaultRefPressure="-1.2e5"
    defaultRefStrainVol="-0.0"
    defaultShearModulus="4.302e6"
    defaultPreConsolidationPressure="-1.69e5"
    defaultCslSlope="1.2"
    defaultVirginCompressionIndex="0.072676"
    defaultRecompressionIndex="0.014535"
  />
</Constitutive>
```

Recall that in the `SolidMechanics_LagrangianFEM` section, `rock` is designated as the material in the computational domain. Here, Modified Cam-Clay `ModifiedCamClay` is used to simulate the elastoplastic behavior of `rock`.

The following material parameters should be defined properly to reproduce the analytical example:

Name	Type	Default	Description
defaultCslSlope	real64	1	Slope of the critical state line
defaultDensity	real64	required	Default Material Density
defaultPreConsolidationPressure	real64	-1.5	Initial preconsolidation pressure
defaultRecompressionIndex	real64	0.002	Recompresion Index
defaultRefPressure	real64	-1	Reference Pressure
defaultRefStrainVol	real64	0	Reference Volumetric Strain
defaultShearModulus	real64	-1	Elastic Shear Modulus Parameter
defaultThermalExpansionCoefficient	real64	0	Default Linear Thermal Expansion Coefficient of the Solid Rock Frame
defaultVirginCompressionIndex	real64	0.005	Virgin compression index
name	string	required	A name is required for any non-unique nodes

The constitutive parameters such as the density, the bulk modulus, and the shear modulus are specified in the International System of Units.

Initial and boundary conditions

The next step is to specify fields, including:

- The initial value (the in-situ stresses and traction at the wellbore wall have to be initialized)
- The boundary conditions (the reduction of wellbore pressure and constraints of the outer boundaries have to be set)

In this tutorial, we need to specify isotropic horizontal stress (σ_h = -100 kPa) and vertical stress (σ_v = -160 kPa). To reach equilibrium, a compressive traction P_w = -100 kPa is instantaneously applied at the wellbore wall `rneg` at time t = 0 s, which will then be gradually increased to a higher value (-300 kPa) to let wellbore expand. The remaining parts of the outer boundaries are subjected to roller constraints. These boundary conditions are set up through the `FieldSpecifications` section.

```
<FieldSpecifications>
  <FieldSpecification
    name="stressXX"
    initialCondition="1"
    setNames="{ all }"
    objectPath="ElementRegions"
    fieldName="rock_stress"
    component="0"
    scale="-1.0e5"/>

  <FieldSpecification
    name="stressYY"
```

(continues on next page)

(continued from previous page)

```
      initialCondition="1"
      setNames="{ all }"
      objectPath="ElementRegions"
      fieldName="rock_stress"
      component="1"
      scale="-1.0e5"/>

    <FieldSpecification
      name="stressZZ"
      initialCondition="1"
      setNames="{ all }"
      objectPath="ElementRegions"
      fieldName="rock_stress"
      component="2"
      scale="-1.6e5"/>

    <Traction
      name="ExternalLoad"
      setNames="{ rneg }"
      objectPath="faceManager"
      scale="-1.0e5"
      tractionType="normal"
      functionName="timeFunction"/>

    <FieldSpecification
      name="xconstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="0"
      scale="0.0"
      setNames="{ xneg, xpos }"/>

    <FieldSpecification
      name="yconstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="1"
      scale="0.0"
      setNames="{ tneg, tpos, ypos }"/>

    <FieldSpecification
      name="zconstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="2"
      scale="0.0"
      setNames="{ zneg, zpos }"/>
  </FieldSpecifications>
```

With `tractionType="normal"`, traction is applied to the wellbore wall `rneg` as a pressure specified from the product of scale `scale="-1.0e5"` and the outward face normal. A table function `timeFunction` is used to define the time-dependent traction `ExternalLoad`. The `coordinates` and `values` form a time-magnitude pair for the loading time history. In this case, the loading magnitude increases linearly as the time evolves.

```
<Functions>
  <TableFunction
    name="timeFunction"
    inputVarNames="{ time }"
    coordinates="{ 0.0, 1.0 }"
    values="{ 1.0, 3.0 }"/>
</Functions>
```

You may note :

- All initial value fields must have `initialCondition` field set to `1`;
- The `setName` field points to the previously defined set to apply the fields;
- `nodeManager` and `faceManager` in the `objectPath` indicate that the boundary conditions are applied to the element nodes and faces, respectively;
- `fieldName` is the name of the field registered in GEOS;
- Component `0`, `1`, and `2` refer to the x, y, and z direction, respectively;
- And the non-zero values given by `scale` indicate the magnitude of the loading;
- Some shorthand, such as `xneg` and `xpos`, are used as the locations where the boundary conditions are applied in the computational domain. For instance, `xneg` means the face of the computational domain located at the left-most extent in the x-axis, while `xpos` refers to the face located at the right-most extent in the x-axis. Similar shorthands include `ypos`, `yneg`, `zpos`, and `zneg`;
- The mud pressure loading has a negative value due to the negative sign convention for compressive stress in GEOS.

The parameters used in the simulation are summarized in the following table.

Symbol	Parameter	Units	Value
P_r	Reference Pressure	[kPa]	120
G	Shear Modulus	[kPa]	4302
P_c	PreConsolidation Pressure	[kPa]	169
M	Slope of CSL	[-]	1.2
c_c	Virgin Compression Index	[-]	0.072676
c_r	Recompression Index	[-]	0.014535
σ_h	Horizontal Stress	[kPa]	-100
σ_v	Vertical Stress	[kPa]	-160
a_0	Initial Well Radius	[m]	0.1
P_w	Mud Pressure	[kPa]	-300

Inspecting results

In the above example, we requested silo-format output files. We can therefore import these into VisIt and use python scripts to visualize the outcome. The following figure shows the distribution of $\sigma_{\theta\theta}$ in the near wellbore region.

The figure below shows the comparisons between the numerical predictions (marks) and the corresponding analytical solutions (solid curves) with respect to the distributions of normal stress components, stress path, the supporting wellbore pressure and wellbore size. It is evident that the predictions well match the analytical results.

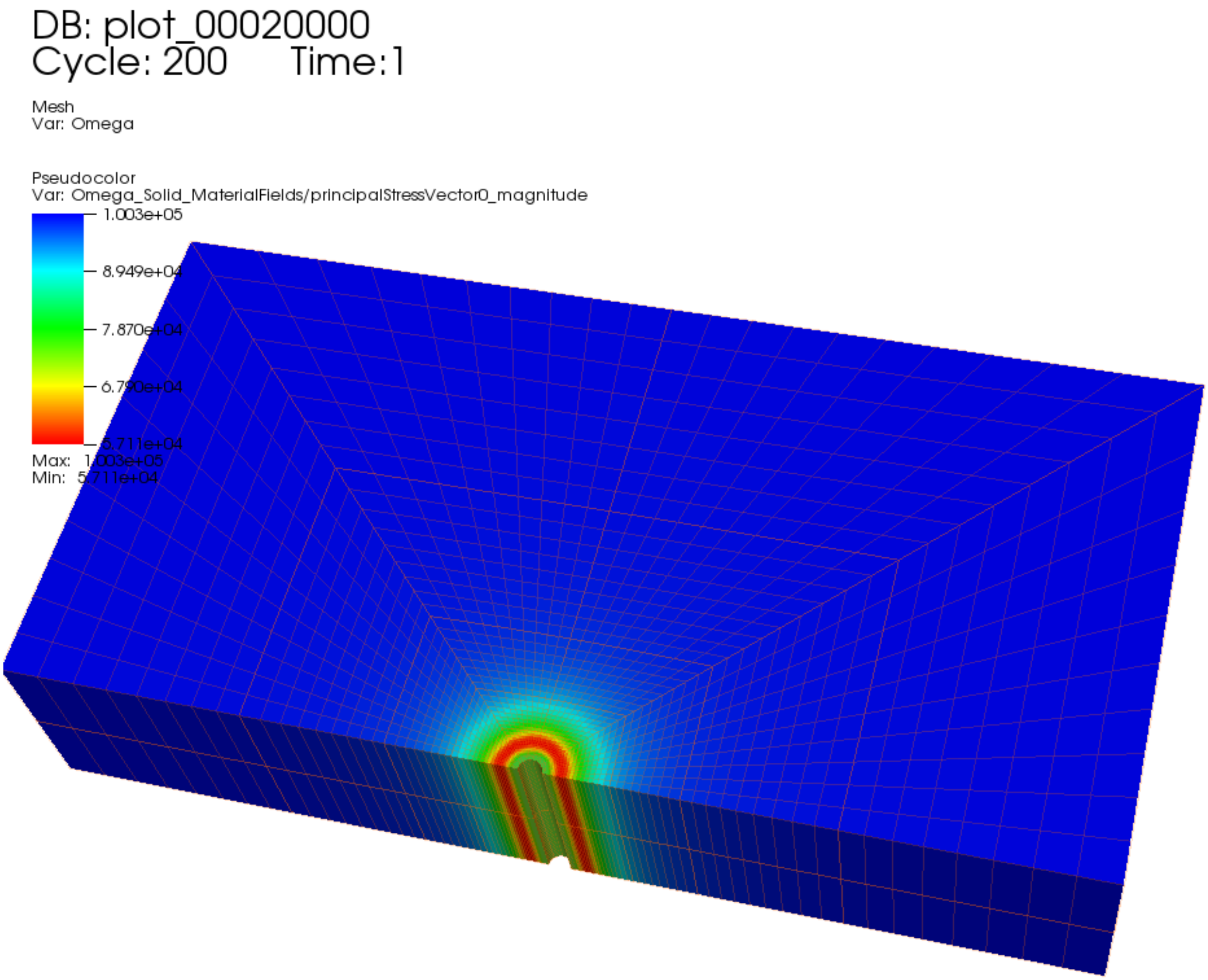
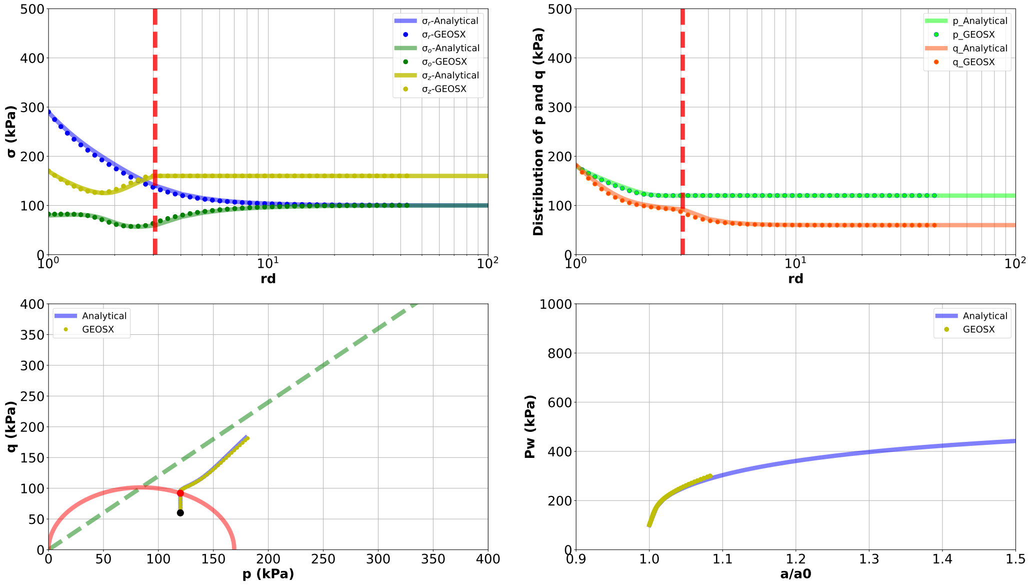


Fig. 1.48: Simulation result of $\sigma_{\theta\theta}$



To go further

Feedback on this example

For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

Deviated Poro-Elastic Wellbore Subjected to Fluid Injection

Problem description

This example aims to solve a typical injection problem of a deviated wellbore subjected to a fluid pressure loaded at wellbore wall. The problem geometry is generated with the internal wellbore mesh generator. Open hole completion and poroelastic deformation are assumed. The coupled poroelastic solver, which combines the solid mechanics solver and the single phase flow solver, is hereby employed to solve this specific problem. In-situ stresses and gravity effect are excluded from this example. Please refer to the case *Deviated Poro-Elastic Wellbore Subjected to In-situ Stresses and Pore Pressure* for in-situ stresses and pore pressure effects.

Analytical solutions of the pore pressure, the radial and hoop stresses in the near wellbore region are expressed in the Laplace space as (Detournay and Cheng, 1988) :

$$p = p_0 \frac{k_0(R\sqrt{s})}{s k_0(\sqrt{s})}$$

$$\sigma_{rr} = -b \frac{1-2\nu}{1-\nu} p_0 \frac{-R k_1(R\sqrt{s}) + k_1(\sqrt{s})}{R^2 \sqrt{s}^3 k_0(\sqrt{s})}$$

$$\sigma_{\theta\theta} = -b \frac{1-2\nu}{1-\nu} p - \sigma_{rr}$$

where s is the Laplace variable normalized by the fluid diffusion coefficient, k_0 and k_1 are respectively the modified Bessel functions of second kind of order 0 and 1, R is the dimensionless radial coordinate that is defined by the radial coordinate normalized by the wellbore radius, ν is the Poisson ratio and b is the Biot coefficient. Fluid pressure and stresses in time space are obtained from these analytical expressions by the inverse Laplace transform (see the attached Python script for more details).

Input file

Everything required is contained within two GEOS xml files that are located at:

```
inputFiles/wellbore/DeviatedPoroElasticWellbore_Injection_base.xml
```

```
inputFiles/wellbore/DeviatedPoroElasticWellbore_Injection_benchmark.xml
```

In this example, we would focus our attention on the `Solvers` and the `Mesh` tags.

Poroelastic solver

The coupled `Poroelastic` solver, that defines a coupling strategy between the solid mechanics solver `SolidMechanicsLagrangianSSLE` and the single phase flow solver `SinglePhaseFVM`, is required for solving this wellbore problem.

```
    <SinglePhasePoromechanics
      name="poroSolve"
      solidSolverName="lagsolve"
      flowSolverName="SinglePhaseFlow"
      logLevel="1"
      targetRegions="{ Omega }">
```

```
    <SolidMechanicsLagrangianSSLE
      name="lagsolve"
      timeIntegrationOption="QuasiStatic"
      discretization="FE1"
      logLevel="0"
      targetRegions="{ Omega }"
      >
```

```
    <SinglePhaseFVM
      name="SinglePhaseFlow"
      logLevel="1"
      discretization="singlePhaseTPFA"
      targetRegions="{ Omega }">
```

Deviated wellbore mesh

The internal wellbore mesh generator `InternalWellbore` is employed to create the mesh of this wellbore problem. The radius of the wellbore and the size of the surrounding rock formation are defined by a vector `radius`. In the tangent direction, `theta` angle is specified from 0 to 180 degree for a half of the domain regarding its symmetry. Note that the whole domain could be specified with a `theta` angle from 0 to 360 degree, if modeling complicated scenarios. The trajectory of the well is defined by `trajectory`. In this example, the wellbore is inclined in the x-z plane by an angle of 45 degree. The `autoSpaceRadialElems` parameter allows optimally increasing the element size from local zone around the wellbore to the far-field zone, which is set to 1 to activate this option. The `useCartesianOuterBoundary` transforms the far-field boundary to a squared shape to enforce a Cartesian aligned outer boundary, which eases the loading of the far-field boundary conditions. In this example, this value is set to 0 for the single region along the radial direction.

```
<Mesh>
  <InternalWellbore
    name="mesh1"
    elementTypes="{ C3D8 }"
    radius="{ 0.1, 4 }"
    theta="{ 0, 180 }"
    zCoords="{ -1, 1 }"
    nr="{ 30 }"
    nt="{ 80 }"
    nz="{ 10 }"
    trajectory="{ { -1.0, 0.0, -1.0 },
                  {  1.0, 0.0,  1.0 } }"
    autoSpaceRadialElems="{ 1 }"
    useCartesianOuterBoundary="0"
    cellBlockNames="{ cb1 }"/>
</Mesh>
```

Constitutive law

Isotropic elastic constitutive block `ElasticIsotropic`, with the specified bulk and shear elastic moduli, is considered for the rock around the wellbore. Fluid properties, such as dynamic viscosity and compressibility, are given in the `CompressibleSinglePhaseFluid` constitutive block. The grain bulk modulus, that is required for computing the Biot coefficient, as well as the default porosity are located in the `BiotPorosity` block. The constant permeability is given in the `ConstantPermeability` block.

```
  <PorousElasticIsotropic
    name="porousRock"
    solidModelName="rock"
    porosityModelName="rockPorosity"
    permeabilityModelName="rockPerm"/>

  <ElasticIsotropic
    name="rock"
    defaultDensity="0"
    defaultBulkModulus="11039657020.4"
    defaultShearModulus="8662741799.83"/>

  <!--  BiotCoefficient="0.771"
        BiotModulus=15.8e9 -->
  <CompressibleSinglePhaseFluid
    name="water"
    defaultDensity="1000"
    defaultViscosity="0.001"
    referencePressure="0e6"
    compressibility="1.78403329184e-10"
    viscosibility="0.0"/>

  <BiotPorosity
    name="rockPorosity"
```

(continues on next page)

(continued from previous page)

```
    grainBulkModulus="48208109259"
    defaultReferencePorosity="0.3"/>

  <ConstantPermeability
    name="rockPerm"
    permeabilityComponents="{ 1.0e-17, 1.0e-17, 1.0e-17 }"/>
```

Boundary conditions

Far-field boundaries are impermeable and subjected to roller constraints. The pressure on the wellbore wall is defined by face pressure field specification. The nodeset generated by the internal wellbore generator for this face is named as `rneg`. The negative sign of the scale denotes the fluid injection. Initial fluid pressure and the corresponding initial porosity are also given for the computational domain. In this example, uniform isotropic permeability is assumed.

```
<FieldSpecifications>
  <FieldSpecification
    name="initialPorosity"
    initialCondition="1"
    setNames="{all}"
    objectPath="ElementRegions/Omega/cb1"
    fieldName="rockPorosity_porosity"
    scale="0.3"/>

  <FieldSpecification
    name="initialPressure"
    initialCondition="1"
    setNames="{ all }"
    objectPath="ElementRegions/Omega/cb1"
    fieldName="pressure"
    scale="0e6"/>

  <FieldSpecification
    name="xConstraint"
    objectPath="nodeManager"
    fieldName="totalDisplacement"
    component="0"
    scale="0.0"
    setNames="{ xneg, xpos }"/>

  <FieldSpecification
    name="yConstraint"
    objectPath="nodeManager"
    fieldName="totalDisplacement"
    component="1"
    scale="0.0"
    setNames="{ tneg, tpos, ypos }"/>

  <FieldSpecification
    name="zconstraint"
    objectPath="nodeManager"
    fieldName="totalDisplacement"
```

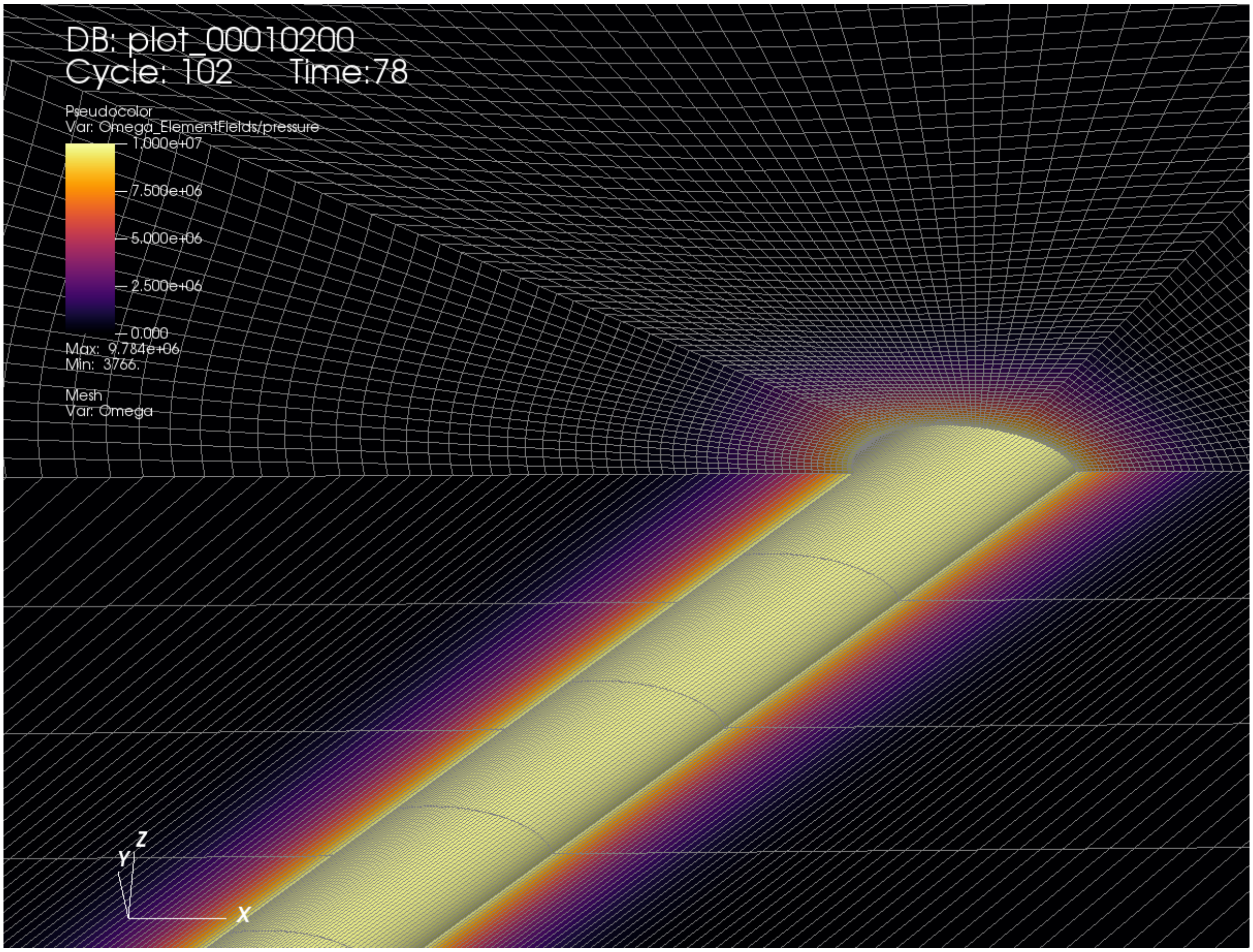
(continues on next page)

(continued from previous page)

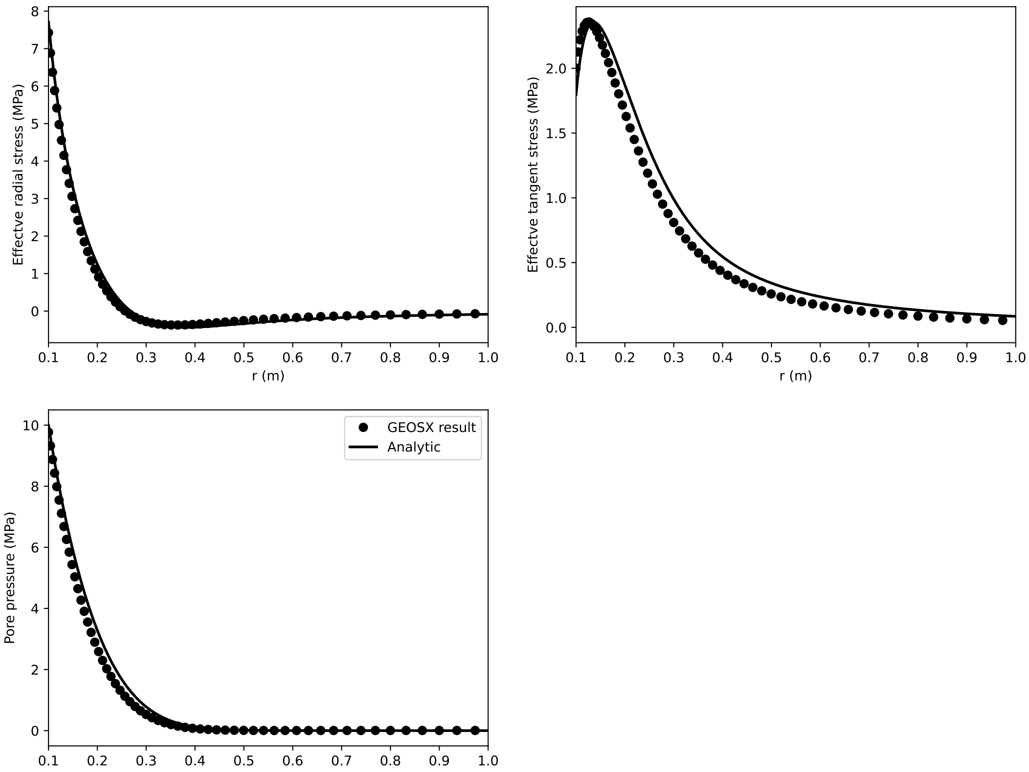
```
    component="2"
    scale="0.0"
    setNames="{ zneg, zpos }"/>

  <FieldSpecification
    name="innerPorePressure"
    objectPath="faceManager"
    fieldName="pressure"
    scale="10e6"
    setNames="{ rneg }"/>
</FieldSpecifications>
```

Results and benchmark

Result of the fluid pressure distribution after 78 s injection is shown in the figure below:



A good agreement between the GEOS results and the corresponding analytical solutions is shown in the figure below:


Effectve radial stress (MPa)
8
7
6
5
4
3
2
1
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
r (m)
Effectve tangent stress (MPa)
2.0
1.5
1.0
0.5
0.0
r (m)
Pore pressure (MPa)
10
8
6
4
2
0
GEOSX result
Analytic

To go further

Feedback on this example

This concludes the deviated poro-elastic wellbore example. For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

Deviated Poro-Elastic Wellbore Subjected to In-situ Stresses and Pore Pressure

Problem description

This example deals with the problem of drilling a deviated poro-elastic wellbore. This is an extension of the poroelastic wellbore example *Deviated Poro-Elastic Wellbore Subjected to Fluid Injection* with the consideration of in-situ stresses and in-situ pore pressure. Both pore pressure and mud pressure are supposed to be nil at the borehole wall following the consideration of (Abousleiman and Cui, 1998). Also, the in-situ horizontal stresses are anisotropic, i.e. $\sigma_hmax > \sigma_hmin$. The wellbore trajectory is deviated from the directions of the in-situ stresses. Analytical solutions of the pore pressure, the radial and hoop stresses in the near wellbore region are given by (Abousleiman and Cui, 1998). They are hereby used to verify the modeling predictions.

Input file

Everything required is contained within two GEOS xml files that are located at:

```
inputFiles/wellbore/DeviatedPoroElasticWellbore_Drilling_base.xml
```

```
inputFiles/wellbore/DeviatedPoroElasticWellbore_Drilling_benchmark.xml
```

This case is nearly identical to another example *Deviated Poro-Elastic Wellbore Subjected to Fluid Injection*, except for the `FieldSpecifications` tag. For this specific case, we need to consider following additional field specifications to define the in-situ stresses, in-situ pore pressure, as well as the zero pore pressure at the borehole wall.

```
    <FieldSpecification
      name="initialPorePressure"
      initialCondition="1"
      setNames="{all}"
      objectPath="ElementRegions/Omega/cb1"
      fieldName="pressure"
      scale="10e6"/>

    <FieldSpecification
      name="Sx"
      initialCondition="1"
      setNames="{all}"
      objectPath="ElementRegions/Omega/cb1"
      fieldName="rock_stress"
      component="0"
      scale="-21.9e6"/>

    <FieldSpecification
      name="Sy"
      initialCondition="1"
      setNames="{all}"
      objectPath="ElementRegions/Omega/cb1"
```

(continues on next page)

(continued from previous page)

```
      fieldName="rock_stress"
      component="1"
      scale="-12.9e6"/>

    <FieldSpecification
      name="Sz"
      initialCondition="1"
      setNames="{all}"
      objectPath="ElementRegions/Omega/cb1"
      fieldName="rock_stress"
      component="2"
      scale="-17.9e6"/>

    <FieldSpecification
      name="innerPorePressure"
      objectPath="faceManager"
      fieldName="pressure"
      scale="0e6"
      setNames="{ rneg }"/>
```

Results and benchmark

Pore pressure distribution after 78 s injection is shown in the figure below:

A good agreement between the GEOS results and the corresponding analytical solutions (Abousleiman and Cui, 1998) is shown in the figure below:

To go further

Feedback on this example

This concludes the deviated poro-elastic wellbore example with in-situ stresses and pore pressure effects. For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

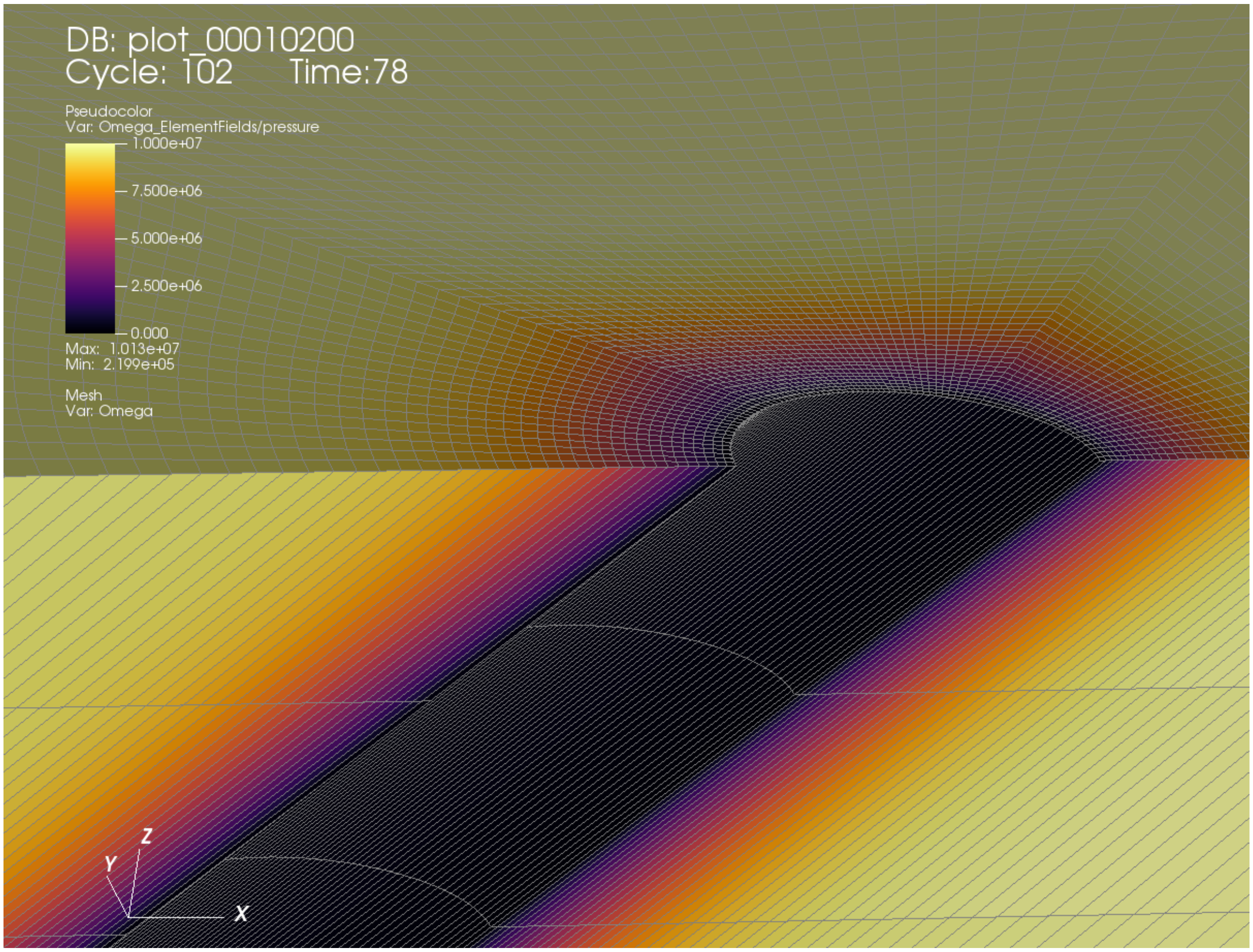
Vertical PoroElasto-Plastic Wellbore Problem

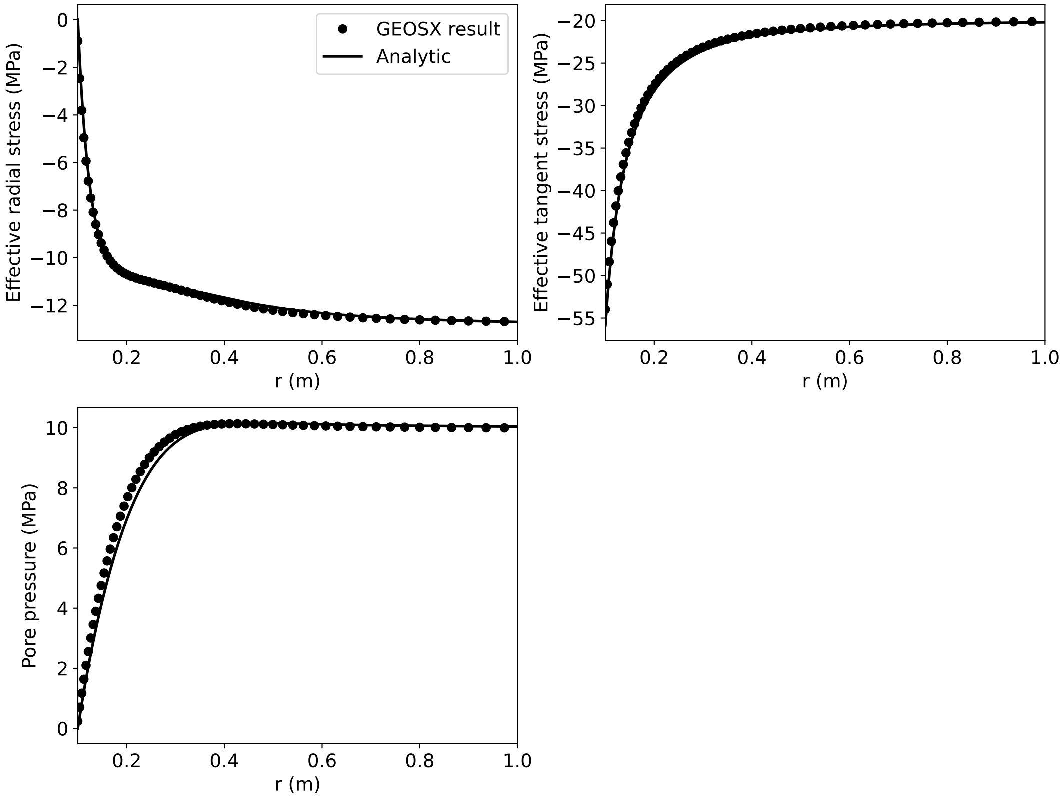
Context

The main objective of this example is to demonstrate how to use the internal wellbore mesh generator and poromechanical solvers in GEOS to tackle wellbore problems in porous media. In this example, a poroplastic model is applied to find the solution of rock deformation within the vicinity of a vertical wellbore, considering elastoplastic deformation, fluid diffusion and poromechanical coupling effect. To do so, a single phase flow solver is fully coupled with a Lagrangian mechanics solver and the Extended Drucker-Prager model (see *Model: Extended Drucker-Prager*) is chosen as the material model for the solid domain. We first solve this problem with a poroelastic model and verify the modeling results with the corresponding analytical solutions. Then, the verified case is modified to test a poroplastic version, whose results are compared with the ones obtained from the poroelastic case to highlight the impact of plasticity in this specific problem.

Objectives

At the end of this example you will know:


DB: plot_00010200
Cycle: 102 Time:78
Pseudocolor
Var: Omega_ElementFields/pressure
1.000e+07
7.500e+06
5.000e+06
2.500e+06
0.000
Max: 1.013e+07
Min: 2.199e+05
Mesh
Var: Omega
Z
Y
X


Effective radial stress (MPa)
0
−2
−4
−6
−8
−10
−12
GEOSX result
Analytic
0.2
0.4
0.6
0.8
1.0
r (m)
Effective tangent stress (MPa)
−20
−25
−30
−35
−40
−45
−50
−55
0.2
0.4
0.6
0.8
1.0
r (m)
Pore pressure (MPa)
10
8
6
4
2
0
0.2
0.4
0.6
0.8
1.0
r (m)

- how to construct meshes for wellbore problems with the internal wellbore mesh generator,
- how to specify initial and boundary conditions, such as reservoir properties, in-situ stresses, mixed loading (mechanical and fluid) at wellbore wall and far-field constraints,
- how to use multiple solvers in GEOS for predicting poroplastic deformations in the near wellbore region.

Input file

This example uses no external input files and everything required is contained within a single GEOS input file.

The xml input files for the test case with poroelasticity are located at:

```
inputFiles/poromechanics/PoroElasticWellbore_base.xml
inputFiles/poromechanics/PoroElasticWellbore_benchmark.xml
```

The xml input files for the test case with poroplasticity are located at:

```
inputFiles/poromechanics/PoroDruckerPragerWellbore_base.xml
inputFiles/poromechanics/PoroDruckerPragerWellbore_benchmark.xml
```

Description of the case

We simulate the wellbore problem subjected to anisotropic horizontal stress (σ_h and σ_H) and vertical stress (σ_v), as shown below. This is a vertical wellbore, which is drilled in a porous medium. By changing the wellbore supporting pressure, the mechanical deformation of the reservoir rock will be induced and evolve with time, due to fluid diffusion and coupling effect. Considering inelastic constitutive behavior, the reservoir rock in the near wellbore region will experience elastoplastic deformation and a plastic zone will be developed and expand with time. To setup the base case, a poroelastic version is employed to find the poroelastic solutions of this wellbore problem, which are verified with the analytical solution (Detournay and Cheng, 1993) from the literature. Following that, a poroplastic version is built and used to obtain the temporal and spatial solutions of pore pressure, displacement and stress fields around the wellbore, considering induced plastic deformation.

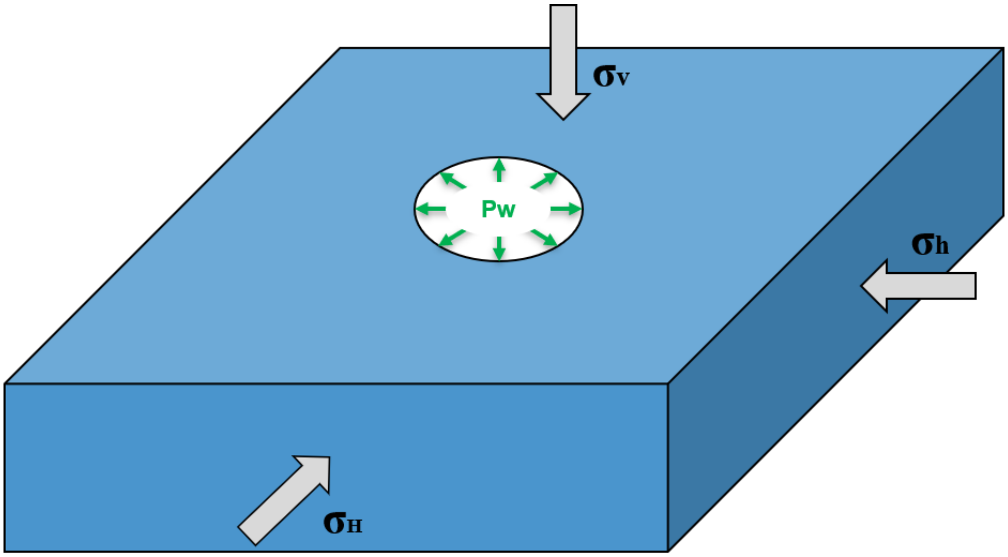


Fig. 1.49: Sketch of the wellbore problem

All inputs for this case are contained inside a single XML file. In this example, we focus our attention on the `Mesh` tags, the `Solver` tags, the `Constitutive` tags, and the `FieldSpecifications` tags.

Mesh

The following figure shows the generated mesh that is used for solving this wellbore problem

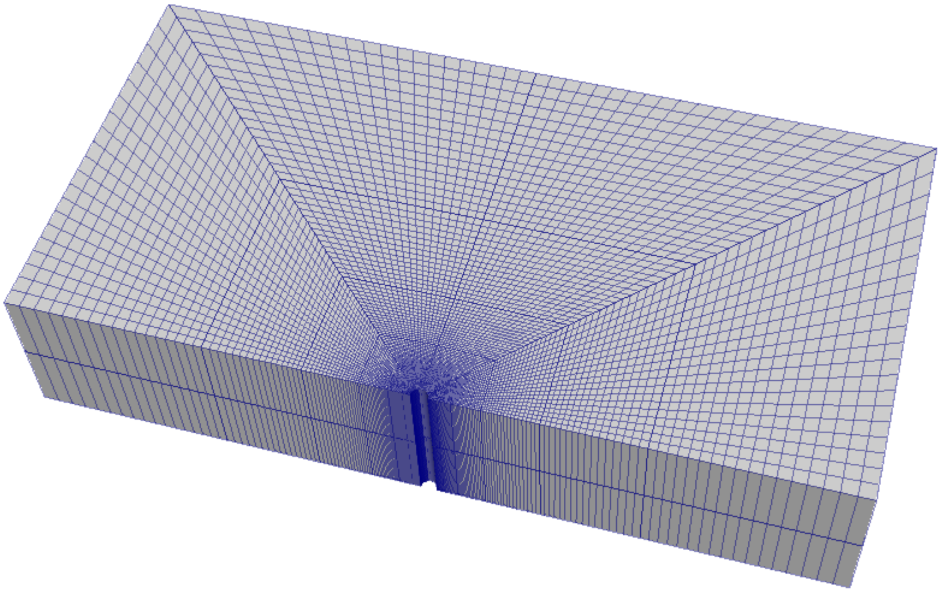


Fig. 1.50: Generated mesh for the wellbore problem

Let us take a closer look at the geometry of this wellbore problem. We use the internal mesh generator `InternalWellbore` to create a rock domain ($10\,m\ \times 5\,m\ \times 2\,m$), with a wellbore of initial radius equal to 0.1 m. Coordinates of `trajectory` defines the wellbore trajectory, which represents a perfect vertical well in this example. By turning on `autoSpaceRadialElems="{ 1 }"`, the internal mesh generator automatically sets number and spacing of elements in the radial direction, which overrides the values of `nr`. With `useCartesianOuterBoundary="0"`, a Cartesian aligned outer boundary on the outer block is enforced. In this way, a structured three-dimensional mesh is created with 100 x 80 x 2 elements in the radial, tangential and z directions, respectively. All the elements are eight-node hexahedral elements (`C3D8`) and refinement is performed to conform with the wellbore geometry. This mesh is defined as a cell block with the name `cb1`.

```
<Mesh>
  <InternalWellbore
    name="mesh1"
    elementTypes="{ C3D8 }"
    radius="{ 0.1, 5.0 }"
    theta="{ 0, 180 }"
    zCoords="{ -1, 1 }"
    nr="{ 40 }"
    nt="{ 80 }"
    nz="{ 2 }"
    trajectory="{ { 0.0, 0.0, -1.0 },
                  { 0.0, 0.0,  1.0 } }"
    autoSpaceRadialElems="{ 1 }"
    useCartesianOuterBoundary="0"
    cellBlockNames="{ cb1 }"/>
</Mesh>
```

Solid mechanics solver

GEOS is a multi-physics platform. Different combinations of physics solvers available in the code can be applied in different regions of the domain and be functional at different stages of the simulation. The Solvers tag in the XML file is used to list and parameterize these solvers.

To specify a coupling between two different solvers, we define and characterize each single-physics solver separately. Then, we customize a *coupling solver* between these single-physics solvers as an additional solver. This approach allows for generality and flexibility in constructing multi-physics solvers. The order of specifying these solvers is not restricted in GEOS. Note that end-users should give each single-physics solver a meaningful and distinct name, as GEOS will recognize these single-physics solvers based on their customized names and create user-expected coupling.

As demonstrated in this example, to setup a poromechanical coupling, we need to define three different solvers in the XML file:

- the mechanics solver, a solver of type SolidMechanics_LagrangianFEM called here mechanicsSolver (more information here: *Solid Mechanics Solver*),

```
<SolidMechanics_LagrangianFEM
  name="mechanicsSolver"
  timeIntegrationOption="QuasiStatic"
  logLevel="1"
  discretization="FE1"
  targetRegions="{ Omega }"
  >
  <NonlinearSolverParameters
    newtonTol = "1.0e-5"
    newtonMaxIter = "15"
  />
</SolidMechanics_LagrangianFEM>
```

- the single-phase flow solver, a solver of type SinglePhaseFVM called here SinglePhaseFlowSolver (more information on these solvers at *Singlephase Flow Solver*),

```
<SinglePhaseFVM
  name="SinglePhaseFlowSolver"
  logLevel="1"
  discretization="singlePhaseTPFA"
  targetRegions="{Omega}">
  <NonlinearSolverParameters
    newtonTol = "1.0e-6"
    newtonMaxIter = "8"
  />
</SinglePhaseFVM>
</Solvers>
```

- the coupling solver (SinglePhasePoromechanics) that will bind the two single-physics solvers above, which is named as PoromechanicsSolver (more information at *Poromechanics Solver*).

```
<Solvers gravityVector="{0.0, 0.0, 0.0}">
<SinglePhasePoromechanics
  name="PoromechanicsSolver"
  solidSolverName="mechanicsSolver"
  flowSolverName="SinglePhaseFlowSolver"
  logLevel="1"
```

(continues on next page)

(continued from previous page)

```
  targetRegions="{Omega}">
  <LinearSolverParameters
    solverType="direct"
    directParallel="0"
    logLevel="0"
  />
  <NonlinearSolverParameters
    newtonMaxIter = "40"
  />
</SinglePhasePoromechanics>
```

The two single-physics solvers are parameterized as explained in their corresponding documents.

In this example, let us focus on the coupling solver. This solver (`PoromechanicsSolver`) uses a set of attributes that specifically describe the coupling process within a poromechanical framework. For instance, we must point this solver to the designated fluid solver (here: `SinglePhaseFlowSolver`) and solid solver (here: `mechanicsSolver`). These solvers are forced to interact through the `porousMaterialNames="{porousRock}"` with all the constitutive models. We specify the discretization method (`FE1`, defined in the `NumericalMethods` section), and the target regions (here, we only have one, `Omega`). More parameters are required to characterize a coupling procedure (more information at *Poromechanics Solver*). In this way, the two single-physics solvers will be simultaneously called and executed for solving the wellbore problem here.

Discretization methods for multiphysics solvers

Numerical methods in multiphysics settings are similar to single physics numerical methods. In this problem, we use finite volume for flow and finite elements for solid mechanics. All necessary parameters for these methods are defined in the `NumericalMethods` section.

As mentioned before, the coupling solver and the solid mechanics solver require the specification of a discretization method called `FE1`. In GEOS, this discretization method represents a finite element method using linear basis functions and Gaussian quadrature rules. For more information on defining finite elements numerical schemes, please see the dedicated *Finite Element Discretization* section.

The finite volume method requires the specification of a discretization scheme. Here, we use a two-point flux approximation scheme (`singlePhaseTPFA`), as described in the dedicated documentation (found here: *Finite Volume Discretization*).

```
<NumericalMethods>
  <FiniteElements>
    <FiniteElementSpace
      name="FE1"
      order="1"/>
    </FiniteElements>
  <FiniteVolume>
    <TwoPointFluxApproximation
      name="singlePhaseTPFA"
    />
  </FiniteVolume>
</NumericalMethods>
```

Constitutive laws

For this test problem, the solid and fluid materials are named as `rock` and `water` respectively, whose mechanical properties are specified in the `Constitutive` section. In this example, different material models, linear elastic isotropic model (see *Model: Elastic Isotropic*) and Extended Drucker-Prager model (see *Model: Extended Drucker-Prager*), are used to solve the mechanical deformation, which is the only difference between the poroelastic and poroplastic cases in this example.

For the poroelastic case, `PorousElasticIsotropic` model is used to describe the linear elastic isotropic response of `rock` to loading. And the single-phase fluid model `CompressibleSinglePhaseFluid` is selected to simulate the flow of `water` upon injection:

```
<Constitutive>
  <PorousElasticIsotropic
    name="porousRock"
    solidModelName="rock"
    porosityModelName="rockPorosity"
    permeabilityModelName="rockPerm"
  />
  <ElasticIsotropic
    name="rock"
    defaultDensity="2700"
    defaultBulkModulus="1.1111e10"
    defaultShearModulus="8.3333e9"
  />
  <CompressibleSinglePhaseFluid
    name="water"
    defaultDensity="1000"
    defaultViscosity="0.001"
    referencePressure="0e6"
    referenceDensity="1000"
    compressibility="2.09028227021e-10"
    referenceViscosity="0.001"
    viscosibility="0.0"
  />
  <BiotPorosity
    name="rockPorosity"
    grainBulkModulus="1.0e27"
    defaultReferencePorosity="0.3"
  />
  <ConstantPermeability
    name="rockPerm"
    permeabilityComponents="{1.0e-20, 1.0e-20, 1.0e-20}"
  />
</Constitutive>
```

For the poroplastic case, `PorousExtendedDruckerPrager` model is used to simulate the elastoplastic behavior of `rock`. And the single-phase fluid model `CompressibleSinglePhaseFluid` is employed to handle the storage and flow of `water`:

```
<Constitutive>
  <PorousExtendedDruckerPrager
    name="porousRock"
    solidModelName="rock"
```

(continues on next page)

(continued from previous page)

```
    porosityModelName="rockPorosity"
    permeabilityModelName="rockPerm"
  />
  <ExtendedDruckerPrager
    name="rock"
    defaultDensity="2700"
    defaultBulkModulus="1.1111e10"
    defaultShearModulus="8.3333e9"
    defaultCohesion="1.0e6"
    defaultInitialFrictionAngle="15.27"
    defaultResidualFrictionAngle="23.05"
    defaultDilationRatio="1.0"
    defaultHardening="0.01"
  />
  <CompressibleSinglePhaseFluid
    name="water"
    defaultDensity="1000"
    defaultViscosity="0.001"
    referencePressure="0e6"
    referenceDensity="1000"
    compressibility="2.09028227021e-10"
    referenceViscosity="0.001"
    viscosibility="0.0"
  />
  <BiotPorosity
    name="rockPorosity"
    grainBulkModulus="1.0e27"
    defaultReferencePorosity="0.3"
  />
  <ConstantPermeability
    name="rockPerm"
    permeabilityComponents="{1.0e-20, 1.0e-20, 1.0e-20}"
  />
</Constitutive>
```

As for the material parameters, `defaultInitialFrictionAngle`, `defaultResidualFrictionAngle` and `defaultCohesion` denote the initial friction angle, the residual friction angle, and cohesion, respectively, as defined by the Mohr-Coulomb failure envelope. As the residual friction angle `defaultResidualFrictionAngle` is larger than the initial one `defaultInitialFrictionAngle`, a strain hardening model is automatically chosen, whose hardening rate is given as `defaultHardening="0.01"`. If the residual friction angle is set to be less than the initial one, strain weakening will take place. `defaultDilationRatio="1.0"` corresponds to an associated flow rule. If using an incompressible fluid, the user can lower the fluid compressibility `compressibility` to 0. The constitutive parameters such as the density, the bulk modulus, and the shear modulus are specified in the International System of Units. A stress-dependent porosity model `rockPorosity` and constant permeability `rockPerm` model are defined in this section.

Initial and boundary conditions

The next step is to specify fields, including:

- The initial value (the in-situ stresses and pore pressure have to be initialized)
- The boundary conditions (traction and fluid loading at the wellbore wall and constraints of the outer boundaries have to be set)

In this example, we need to specify anisotropic horizontal stress (σ_h = -9.0 MPa and σ_H = -11.0 MPa) and vertical stress (σ_v = -12.0 MPa). A compressive traction (`InnerMechanicalLoad`) P_w = -10 MPa and fluid loading (`InnerFluidLoad`) P_f = 10 MPa are applied at the wellbore wall `rneg`. The remaining parts of the outer boundaries are subjected to roller constraints. These boundary conditions are set up through the `FieldSpecifications` section.

```
<FieldSpecifications>
  <FieldSpecification
    name="stressXX"
    initialCondition="1"
    setNames="{all}"
    objectPath="ElementRegions/Omega/cb1"
    fieldName="rock_stress"
    component="0"
    scale="-9.0e6"
  />

  <FieldSpecification
    name="stressYY"
    initialCondition="1"
    setNames="{all}"
    objectPath="ElementRegions/Omega/cb1"
    fieldName="rock_stress"
    component="1"
    scale="-11.0e6"
  />

  <FieldSpecification
    name="stressZZ"
    initialCondition="1"
    setNames="{all}"
    objectPath="ElementRegions/Omega/cb1"
    fieldName="rock_stress"
    component="2"
    scale="-12.0e6"
  />

  <FieldSpecification
    name="initialPressure"
    initialCondition="1"
    setNames="{all}"
    objectPath="ElementRegions/Omega/cb1"
    fieldName="pressure"
    scale="0e6"
  />
```

(continues on next page)

(continued from previous page)

```
  <FieldSpecification
    name="xconstraint"
    objectPath="nodeManager"
    fieldName="totalDisplacement"
    component="0"
    scale="0.0"
    setNames="{xneg, xpos}"
  />

  <FieldSpecification
    name="yconstraint"
    objectPath="nodeManager"
    fieldName="totalDisplacement"
    component="1"
    scale="0.0"
    setNames="{tneg, tpos, ypos}"
  />

  <FieldSpecification
    name="zconstraint"
    objectPath="nodeManager"
    fieldName="totalDisplacement"
    component="2"
    scale="0.0"
    setNames="{zneg, zpos}"
  />

  <Traction
    name="InnerMechanicalLoad"
    setNames="{ rneg }"
    objectPath="faceManager"
    scale="-10.0e6"
    tractionType="normal"
    functionName="timeFunction"
  />

  <FieldSpecification
    name="InnerFluidLoad"
    setNames="{ rneg }"
    objectPath="faceManager"
    fieldName="pressure"
    scale="10e6"
    functionName="timeFunction"
  />
</FieldSpecifications>
```

With `tractionType="normal"`, traction is applied to the wellbore wall `rneg` as a pressure specified from the product of scale `scale="-10.0e6"` and the outward face normal. A table function `timeFunction` is used to define the time-dependent loading. The `coordinates` and `values` form a time-magnitude pair for the loading time history. In this case, the loading magnitude is given as:

```
<Functions>
```

(continues on next page)

(continued from previous page)

```
  <TableFunction
    name="timeFunction"
    inputVarNames="{time}"
    coordinates="{0.0, 0.1, 1e6}"
    values="{0.0, 1.0, 1.0}"
  />
</Functions>
```

You may note :

- All initial value fields must have `initialCondition` field set to `1`;
- The `setName` field points to the previously defined box to apply the fields;
- `nodeManager` and `faceManager` in the `objectPath` indicate that the boundary conditions are applied to the element nodes and faces, respectively;
- `fieldName` is the name of the field registered in GEOS;
- Component `0`, `1`, and `2` refer to the x, y, and z direction, respectively;
- And the non-zero values given by `scale` indicate the magnitude of the loading;
- Some shorthands, such as `xneg` and `xpos`, are used as the locations where the boundary conditions are applied in the computational domain. For instance, `xneg` means the portion of the computational domain located at the left-most in the x-axis, while `xpos` refers to the portion located at the right-most area in the x-axis. Similar shorthands include `ypos`, `yneg`, `zpos`, and `zneg`;
- The mud pressure loading has a negative value due to the negative sign convention for compressive stress in GEOS.

The parameters used in the simulation are summarized in the following table, which are specified in the `Constitutive` and `FieldSpecifications` sections.

Symbol	Parameter	Unit	Value
K	Bulk Modulus	[GPa]	11.11
G	Shear Modulus	[GPa]	8.33
C	Cohesion	[MPa]	1.0
ϕ_i	Initial Friction Angle	[degree]	15.27
ϕ_r	Residual Friction Angle	[degree]	23.05
c_h	Hardening Rate	[-]	0.01
σ_h	Min Horizontal Stress	[MPa]	-9.0
σ_H	Max Horizontal Stress	[MPa]	-11.0
σ_v	Vertical Stress	[MPa]	-12.0
a_0	Initial Well Radius	[m]	0.1
P_w	Traction at Well	[MPa]	-10.0
P_f	Fluid Pressure at Well	[MPa]	10.0
ρ_f	Fluid Density	[kg/m^3]	1000.0
μ	Fluid Viscosity	[Pa s]	0.001
c_f	Fluid Compressibility	[Pa^{-1}]	2.09*10^{-10}
κ	Matrix Permeability	[m^2]	1.0*10^{-20}
ϕ	Porosity	[-]	0.3

Inspecting results

As defined in the `Events` section, we run this simulation for 497640 seconds. In the above examples, we requested silo-format output files. We can therefore import these into VisIt and use python scripts to visualize the outcome. Please note that a non-dimensional time is used in the analytical solution, and the end time here leads to a non-dimensional end time of t* = 4.62.

Using the poroelastic solver, below figure shows the prediction of pore pressure distribution upon fluid injection.

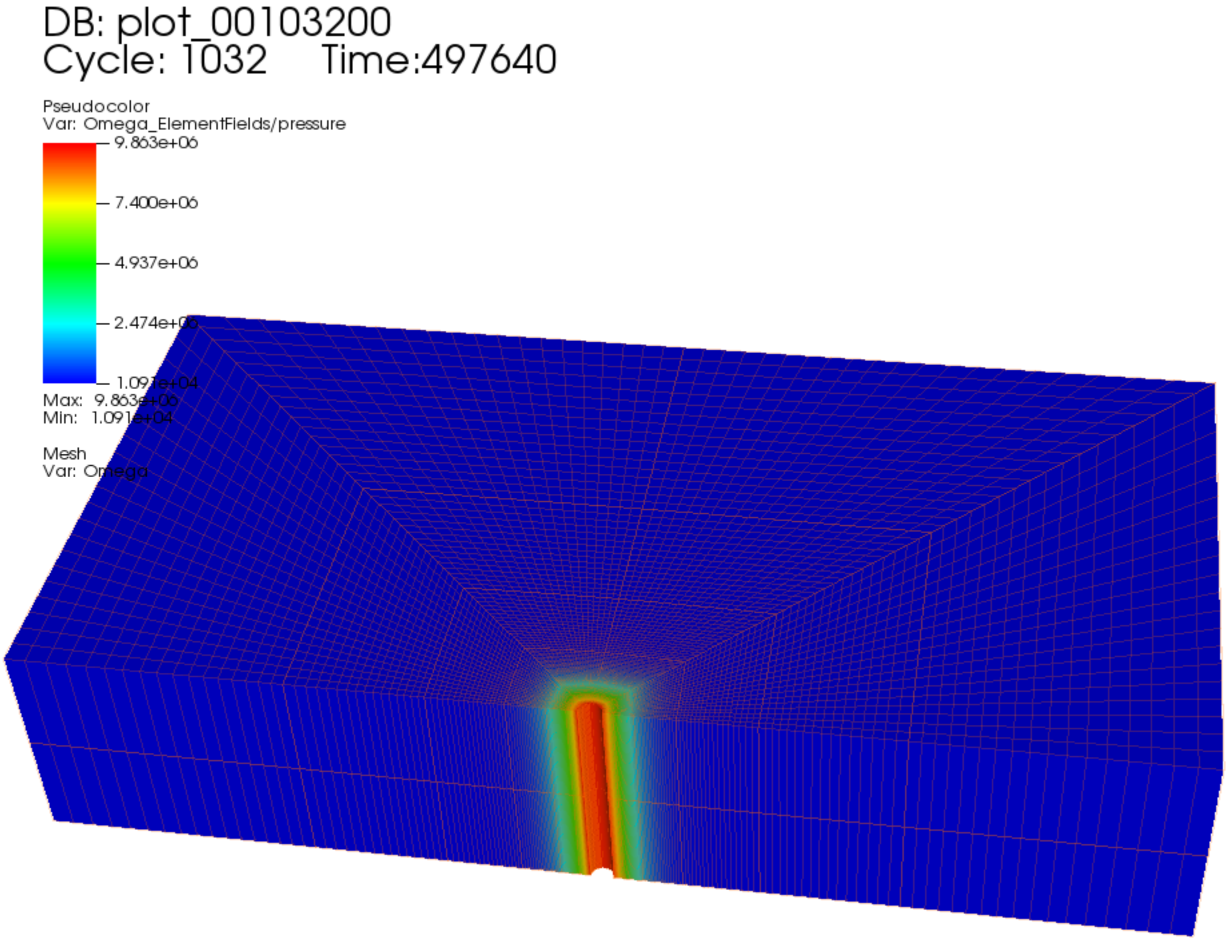


Fig. 1.51: Simulation result of pore pressure distribution

For the above poroelastic example, an analytical solution (Detournay and Cheng, 1993) is hereby employed to verify the accuracy of the numerical results. Following figure shows the comparisons between the numerical predictions (marks) and the corresponding analytical solutions (solid curves) with respect to the distributions of pore pressure, radial displacement, effective radial and tangential stresses along the minimum horizontal stress direction (x-axis). One can observe that GEOS results correlate very well with the analytical solutions for the poroelastic case.

For the same 3D wellbore problem, the poroplastic case is thereafter tested and compared with the poroelastic one. The figure below shows the distribution of σ_{yy} in the near wellbore region for both cases. As expected, a relaxation of the tangential stress along the direction of minimum horizontal stress is detected, which can be attributed to the plastic response of the rock.

By using python scripts, we can extract the simulation results along any direction and provide detailed comparisons between different cases. Here, the pore pressure, radial displacement, radial and tangential effective stresses along

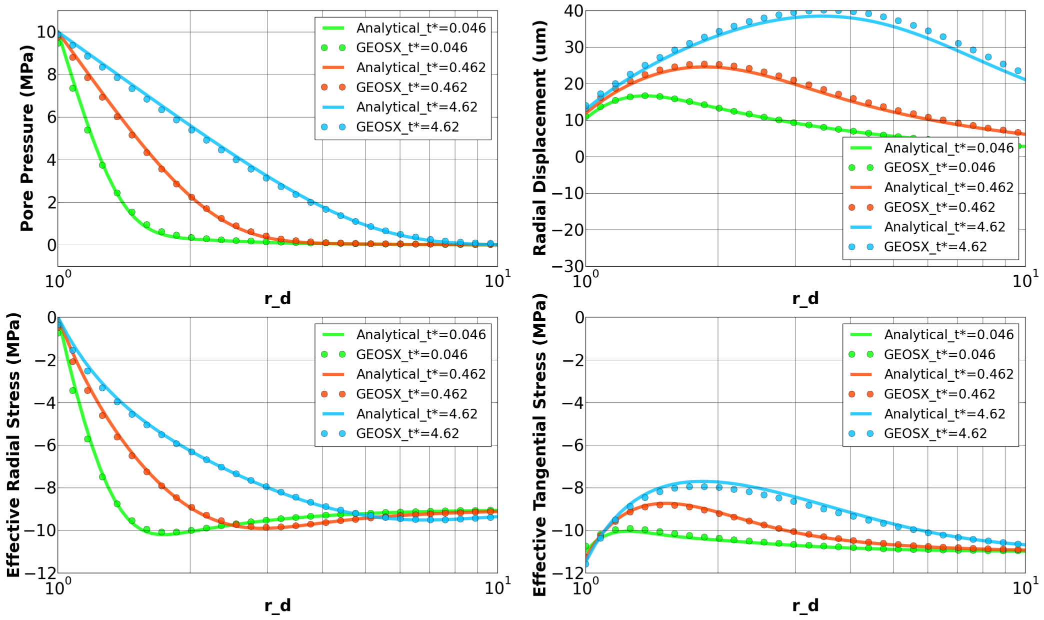


Fig. 1.52: Comparing GEOS results with analytical solutions

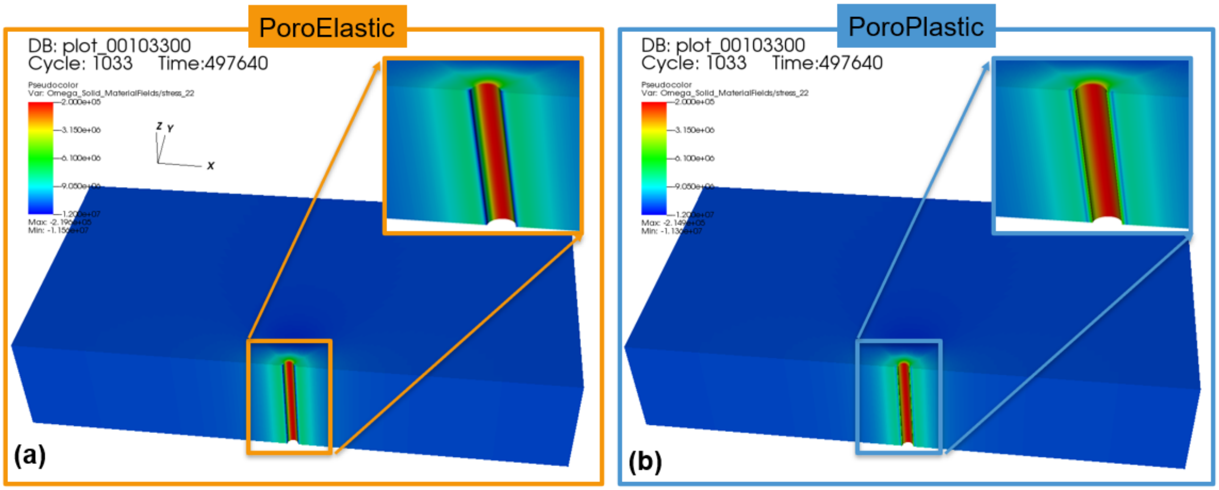


Fig. 1.53: Simulation result of Syy: PoroElastic vs. PoroPlastic

the direction of minimum horizontal stress are obtained at different time steps and plotted against the corresponding ones of the poroelastic case. Because of fluid diffusion and coupling effect, following figure shows that these solutions evolve with time for both cases. As mentioned above, a plastic zone is developed in the vicinity of the wellbore, due to stress concentration. As for the far field region, these two cases become almost identical, with the rock deformation governed by poroelasticity.

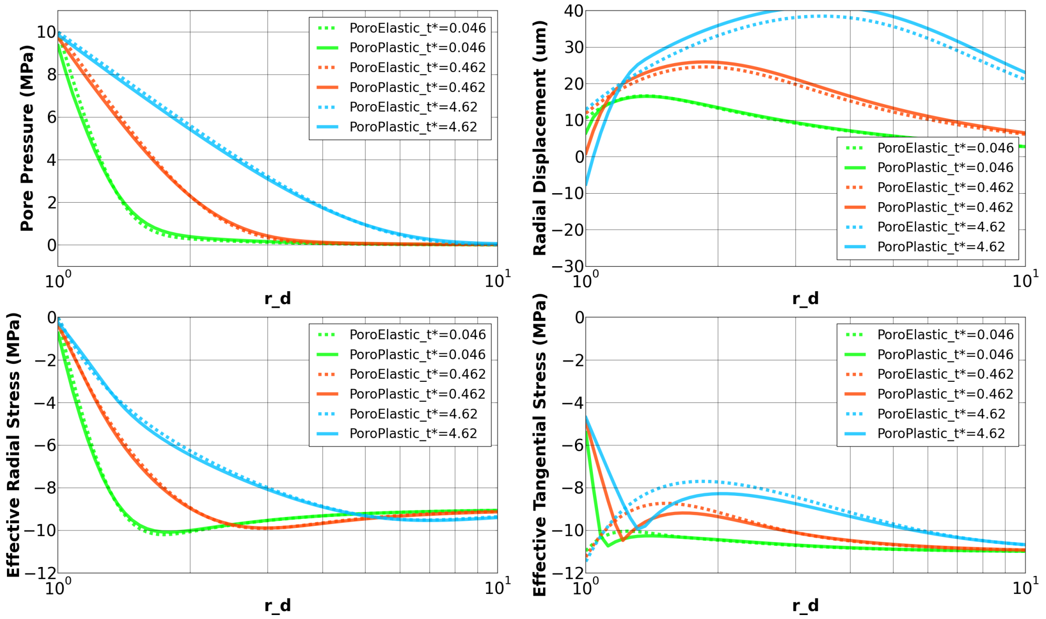


Fig. 1.54: Comparing the PoroPlastic case with the PoroElastic case at different times

To go further

Feedback on this example

This concludes the example on PoroPlasticity Model for Wellbore Problems. For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

For more details

- More on plasticity models, please see *Model: Extended Drucker-Prager*.
- More on multiphysics solvers, please see *Poromechanics Solver*.

Pure Thermal Diffusion Around a Wellbore

Problem description

This example uses the thermal single-phase flow solver to model a pure thermal diffusion problem around a wellbore. To mimic this specific problem, thermal convection and fluid flow are neglected by setting fluid pressure and fluid heat capacity to zero. With a uniform temperature applied on the inner surface of the wellbore, temperature field would radially diffuse as shown in the figure below:

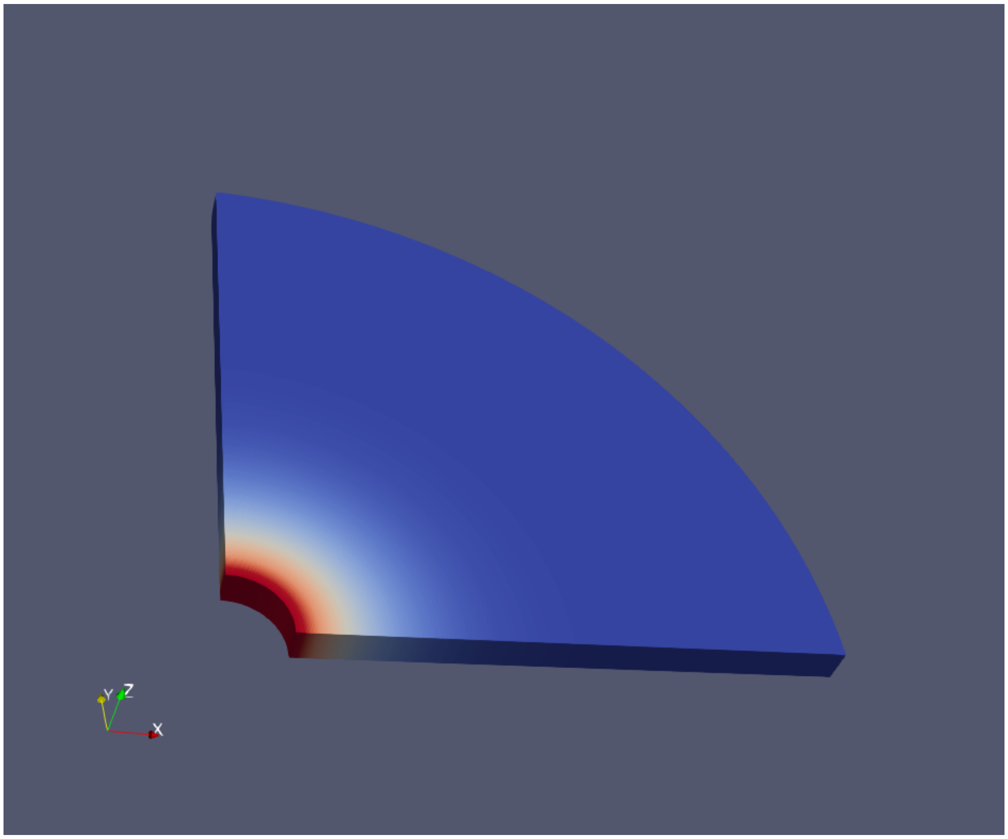


Fig. 1.55: Sketch of the radial thermal diffusion around a wellbore

Analytical results of the temperature profile along the radial direction is given by (Wang and Papamichos, 1994) :

$$T(r) = T_{in} \sqrt{\frac{R_{in}}{r}} erfc(\frac{r - R_{in}}{2\sqrt{c_T t}})$$

where r is the radial coordinate, T_{in} is the temperature applied on the surface of the wellbore at $r = R_{in}$, c_T is the thermal diffusion coefficient of rock, which is defined as the ratio between the thermal conductivity and the volumetric heat capacity of rock.

Input file

This benchmark example uses no external input file and everything required is contained within two GEOS xml files that are located at:

```
inputFiles/singlePhaseFlow/thermalCompressible_2d_base.xml
```

and

```
inputFiles/singlePhaseFlow/thermalCompressible_2d_benchmark.xml
```

The corresponding integrated test is

```
inputFiles/singlePhaseFlow/thermalCompressible_2d_smoke.xml
```

In this example, we would focus our attention on the `Constitutive` and `FieldSpecifications` tags.

Constitutive

The volumetric heat capacity of the medium around the wellbore is defined in the `SolidInternalEnergy` XML block as

```
    <SolidInternalEnergy
      name="rockInternalEnergy"
      volumetricHeatCapacity="4.56e6"
      referenceTemperature="0"
      referenceInternalEnergy="0"/>
```

The thermal conductivity of the medium around the wellbore is defined in the `SinglePhaseConstantThermalConductivity` XML block as

```
    <SinglePhaseConstantThermalConductivity
      name="thermalCond"
      thermalConductivityComponents="{ 1.66, 1.66, 1.66 }"/>
```

The volumetric heat capacity of fluid is set to a negligible value to exclude thermal convection effect. It is defined in the `ThermalCompressibleSinglePhaseFluid` XML block as

```
    <ThermalCompressibleSinglePhaseFluid
      name="fluid"
      defaultDensity="1000"
      defaultViscosity="0.001"
      referencePressure="0.0"
      referenceTemperature="0"
      compressibility="5e-10"
      thermalExpansionCoeff="3e-4"
```

(continues on next page)

(continued from previous page)

```
      viscosibility="0.0"
      volumetricHeatCapacity="1"
      referenceInternalEnergy="0.99"/>
```

FieldSpecifications

The initial temperature, the imposed temperature at the curved wellbore surface as well as the far-field temperature are defined as Dirichlet face boundary conditions using `faceManager` as

```
    <FieldSpecification
      name="initialTemperature"
      initialCondition="1"
      setNames="{ all }"
      objectPath="ElementRegions/region/cb"
      fieldName="temperature"
      scale="0"/>

    <FieldSpecification
      name="sinkTemperature"
      setNames="{ rpos }"
      objectPath="faceManager"
      fieldName="temperature"
      scale="0"/>

    <FieldSpecification
      name="sourceTemperature"
      setNames="{ rneg }"
      objectPath="faceManager"
      fieldName="temperature"
      scale="100.0"/>
```

Although a pure thermal diffusion problem is considered, it is also required to define specifications for fluid pressure, as thermal transfer is always coupled with fluid flow in GEOS. In this example, fluid pressure is set to zero everywhere to mimic a pure thermal diffusion problem as

```
    <FieldSpecification
      name="initialPressure"
      initialCondition="1"
      setNames="{ all }"
      objectPath="ElementRegions/region/cb"
      fieldName="pressure"
      scale="0e6"/>

    <FieldSpecification
      name="sinkPressure"
      setNames="{ rpos }"
      objectPath="faceManager"
      fieldName="pressure"
      scale="0e6"/>

    <FieldSpecification
```

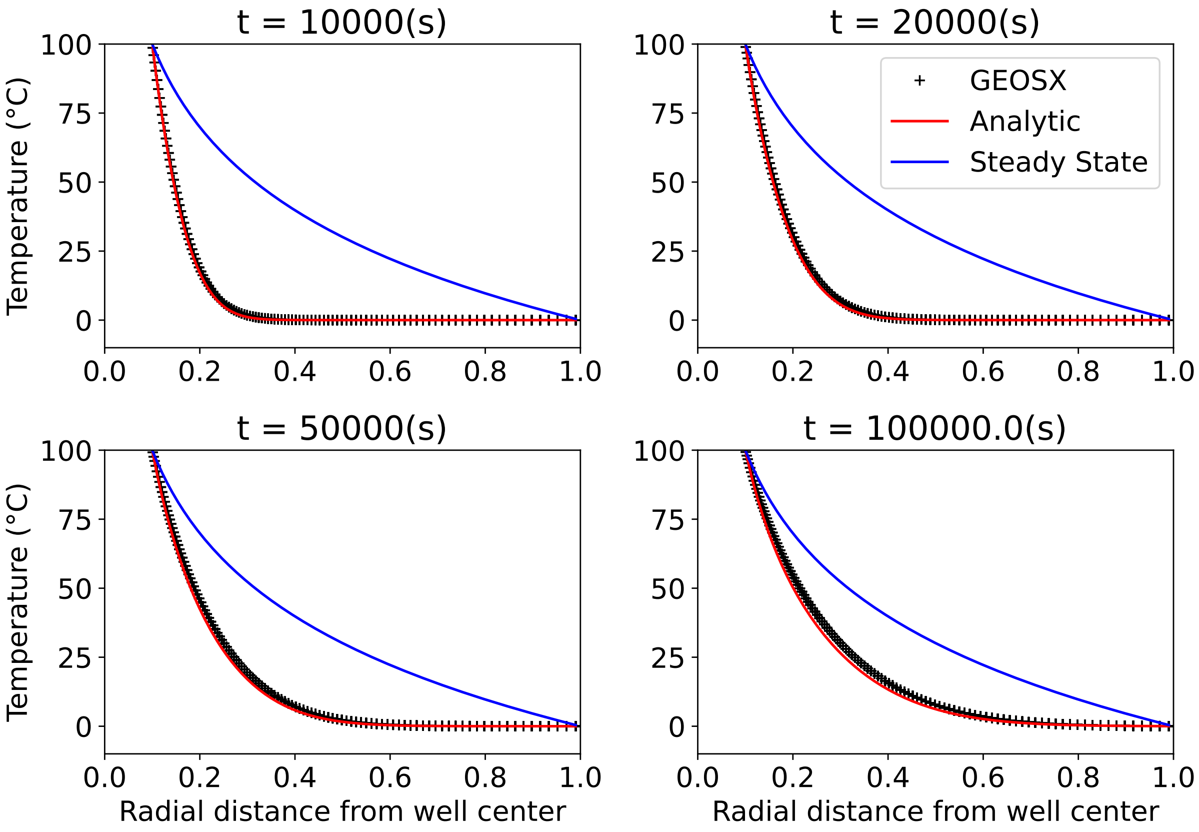
(continues on next page)

(continued from previous page)

```
      name="sourcePressure"
      setNames="{ rneg }"
      objectPath="faceManager"
      fieldName="pressure"
      scale="0e6"/>
```

Results and benchmark

A good agreement between the GEOS results and analytical results is shown in the figure below:



To go further

Feedback on this example

This concludes the example of pure thermal diffusion problem around a wellbore. For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

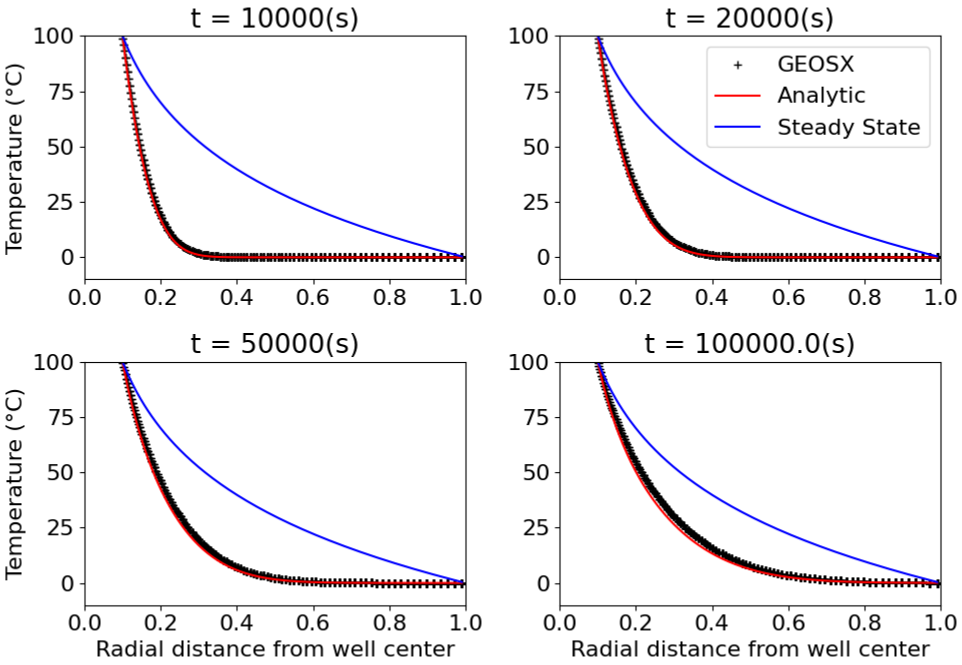


Fig. 1.56: Radial thermal diffusion around a wellbore: a validation against analytical results

Cased ThermoElastic Wellbore Problem

Problem description

This example uses the thermal option of the `SinglePhasePoromechanics` solver to handle a cased wellbore problem subjected to a uniform temperature change on the inner surface of the casing. The completed wellbore is composed of a steel casing, a cement sheath and rock formation. Isotropic linear thermoelastic behavior is assumed for all the three materials. No separation or thermal barrier is allowed for the casing-cement and cement-rock contact interfaces. Plane strain condition is also assumed.

Solution to this axisymmetric problem can be obtained in the cylindrical coordinate system by using an implicit 1D finite difference method (Jane and Lee 1999). Results of such analysis will be considered as reference solutions to validate GEOS results.

Input file

This benchmark example uses no external input files and everything required is contained within two GEOS xml files that are located at:

```
inputFiles/wellbore/CasedThermoElasticWellbore_base.xml
```

and

```
inputFiles/wellbore/CasedThermoElasticWellbore_benchmark.xml
```

The corresponding integrated test is

```
inputFiles/wellbore/CasedThermoElasticWellbore_smoke.xml
```

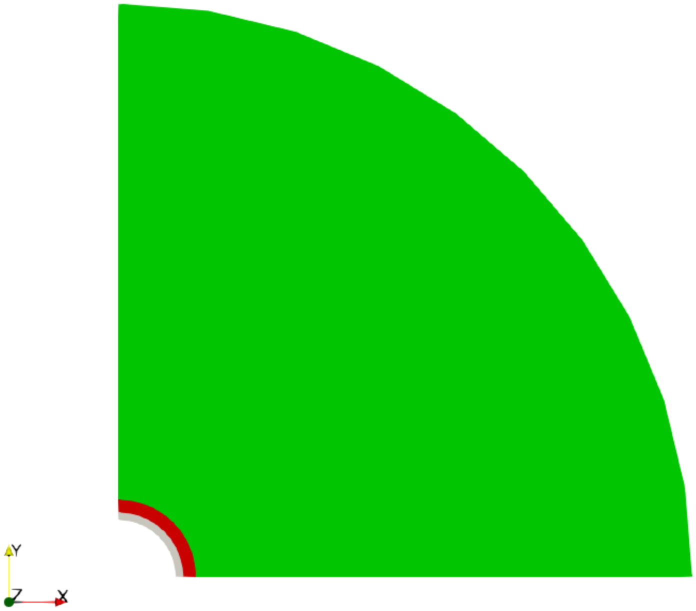


Fig. 1.57: Sketch of a cased thermoelastic wellbore

Geometry and mesh

The internal wellbore mesh generator `InternalWellbore` is employed to create the mesh of this wellbore problem. The radii of the casing cylinder, the cement sheath cylinder and the far-field boundary of the surrounding rock formation are defined by a vector `radius`. In the tangent direction, `theta` angle is specified from 0 to 90 degrees to simulate the problem on a quarter of the wellbore geometry. The problem is under plane strain condition and therefore we only consider radial thermal diffusion on a single horizontal layer. The trajectory of the well is defined by `trajectory`, which is vertical in this case. The `autoSpaceRadialElems` parameters allow for optimally increasing the element size from the wellbore to the far-field zone. In this example, the auto spacing option is only applied to the rock formation. The `useCartesianOuterBoundary` with a value 3 specified for the rock layer transforms the far-field boundary to a circular shape. The `cellBlockNames` and `elementTypes` define the regions and related element types associated to casing, cement sheath, and rock.

```
<Mesh>
  <InternalWellbore
    name="mesh1"
    elementTypes="{ C3D8, C3D8, C3D8 }"
    radius="{ 0.15707, 0.17780, 0.21272, 1.5707 }"
    theta="{ 0, 90 }"
    zCoords="{ 0, 0.1 }"
    nr="{ 5, 5, 5 }"
    nt="{ 10 }"
    nz="{ 1 }"
    trajectory="{ { 0.0, 0.0, 0.0 },
                  { 0.0, 0.0, 0.1 } }"
    autoSpaceRadialElems="{ 0, 0, 1 }"
    cellBlockNames="{ casing, cement, rock }"
    />
</Mesh>
```

Material properties

The bulk and shear drained elastic moduli of the materials as well as its drained linear thermal expansion coefficient relating stress change to temperature change are defined within the `Constitutive` tag as follows:

```
  <ElasticIsotropic
    name="casingSolid"
    defaultDensity="7500"
    defaultBulkModulus="159.4202899e9"
    defaultShearModulus="86.61417323e9"
    defaultThermalExpansionCoefficient="1.2e-5"/>

  <ElasticIsotropic
    name="cementSolid"
    defaultDensity="2700"
    defaultBulkModulus="2.298850575e9"
    defaultShearModulus="1.652892562e9"
    defaultThermalExpansionCoefficient="2.0e-5"/>

  <ElasticIsotropic
    name="rockSolid"
    defaultDensity="2700"
```

(continues on next page)

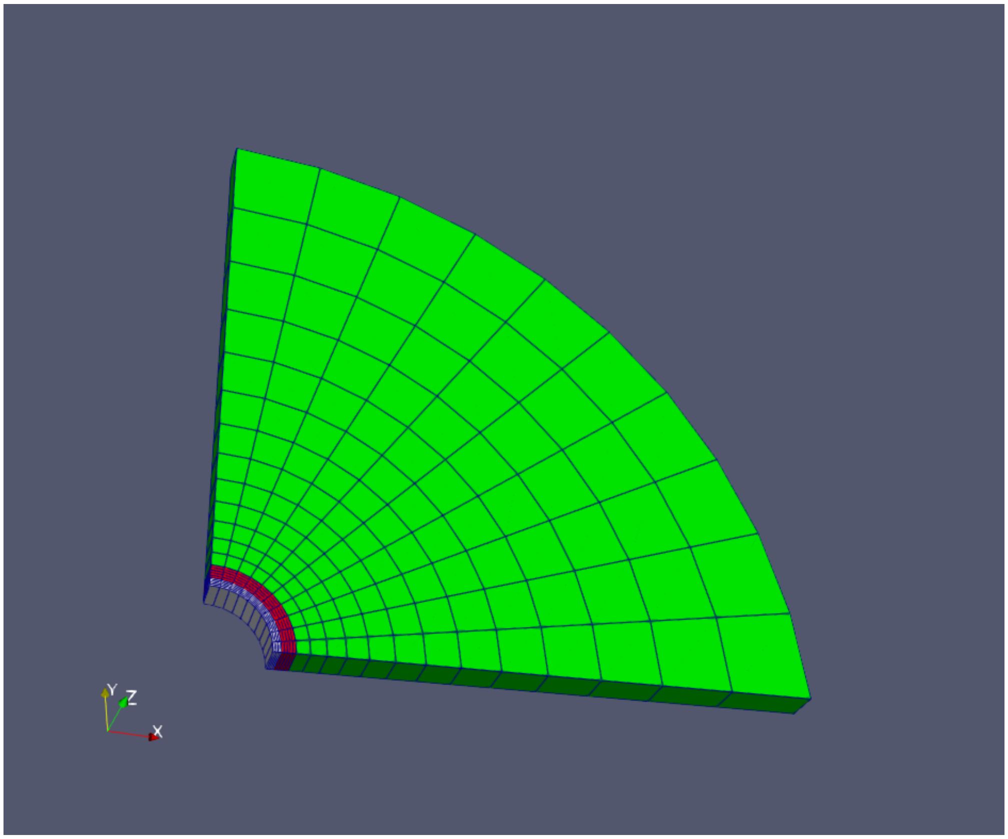


Fig. 1.58: An optimized mesh for the cased wellbore.

(continued from previous page)

```
      defaultBulkModulus="5.535714286e9"
      defaultShearModulus="3.81147541e9"
      defaultThermalExpansionCoefficient="2.0e-5"/>
```

Here the solid density is also defined but it is not used because the gravitational effect is ignored in this example. To mimic a thermoelastic coupling without fluid flow, a negligible porosity and a zero Biot coefficient are defined as:

```
    <BiotPorosity
      name="casingPorosity"
      defaultReferencePorosity="1e-6"
      grainBulkModulus="159.4202899e9"/>

    <BiotPorosity
      name="cementPorosity"
      defaultReferencePorosity="1e-6"
      grainBulkModulus="2.298850575e9"/>

    <BiotPorosity
      name="rockPorosity"
      defaultReferencePorosity="1e-6"
      grainBulkModulus="5.535714286e9"/>
```

In this XML block, the Biot coefficient is defined using the elastic bulk modulus K_s of the solid skeleton as $b_{Biot} = 1 - K/K_s$. In this example, we define a skeleton bulk modulus that is identical to the drained bulk modulus K defined above to enforce the Biot coefficient to zero.

The thermal conductivities and the volumetric heat capacities of casing, cement, and rock are defined by following XML blocks:

```
    <SinglePhaseConstantThermalConductivity
      name="casingThermalCond"
      thermalConductivityComponents="{ 15, 15, 15 }"/>

    <SinglePhaseConstantThermalConductivity
      name="cementThermalCond"
      thermalConductivityComponents="{ 1.0, 1.0, 1.0 }"/>

    <SinglePhaseConstantThermalConductivity
      name="rockThermalCond"
      thermalConductivityComponents="{ 1.66, 1.66, 1.66 }"/>
```

and

```
    <SolidInternalEnergy
      name="casingInternalEnergy"
      volumetricHeatCapacity="1.375e6"
      referenceTemperature="0"
      referenceInternalEnergy="0"/>

    <SolidInternalEnergy
      name="cementInternalEnergy"
      volumetricHeatCapacity="4.2e6"
      referenceTemperature="0"
```

(continues on next page)

(continued from previous page)

```
      referenceInternalEnergy="0"/>

    <SolidInternalEnergy
      name="rockInternalEnergy"
      volumetricHeatCapacity="4.56e6"
      referenceTemperature="0"
      referenceInternalEnergy="0"/>
```

An ultra low permeability is defined for the three layers to simulate a thermoelastic problem without the impact of fluid flow.

```
    <ConstantPermeability
      name="casingPerm"
      permeabilityComponents="{ 1.0e-100, 1.0e-100, 1.0e-100 }"/>

    <ConstantPermeability
      name="cementPerm"
      permeabilityComponents="{ 1.0e-100, 1.0e-100, 1.0e-100 }"/>

    <ConstantPermeability
      name="rockPerm"
      permeabilityComponents="{ 1.0e-100, 1.0e-100, 1.0e-100 }"/>
```

Also, a negligible volumetric heat capacity is defined for fluid to completely ignore the thermal convection effect such that only thermal transfer via the diffusion phenomenon is considered.

```
    <ThermalCompressibleSinglePhaseFluid
      name="fluid"
      defaultDensity="1000"
      defaultViscosity="1e-3"
      referencePressure="0.0"
      referenceTemperature="20.0"
      compressibility="5e-10"
      thermalExpansionCoeff="1e-10"
      viscosibility="0.0"
      volumetricHeatCapacity="1"
      referenceInternalEnergy="1"/>
```

Other fluid properties such as viscosity, thermal expansion coefficient, etc. are not relevant to this example because fluid flow is ignored and pore pressure is zero everywhere.

Boundary conditions

The mechanical boundary conditions are applied to ensure the axisymmetric plane strain conditions such as:

```
    <FieldSpecification
      name="tNegConstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="1"
      scale="0.0"
      setNames="{ tneg }"/>
```

(continues on next page)

(continued from previous page)

```
   <FieldSpecification
      name="tPosConstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="0"
      scale="0.0"
      setNames="{ tpos }"/>

    <FieldSpecification
      name="zconstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="2"
      scale="0.0"
      setNames="{ zneg, zpos }"/>
```

Besides, the far-field boundary is assumed to be fixed because the local changes on the wellbore must have negligible effect on the far-field boundary.

```
    <FieldSpecification
      name="rPosConstraint_x"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="0"
      scale="0.0"
      setNames="{ rpos }"/>

    <FieldSpecification
      name="rPosConstraint_y"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="1"
      scale="0.0"
      setNames="{ rpos }"/>
```

The traction-free condition on the inner surface of the casing is defined by:

```
    <Traction
      name="innerPressure"
      objectPath="faceManager"
      tractionType="normal"
      scale="0.0e6"
      setNames="{ rneg }"/>
```

The initial reservoir temperature (that is also the far-field boundary temperature) and the temperature of a cold fluid applied on the inner surface of the casing are defined as

```
    <FieldSpecification
      name="initialTemperature"
      initialCondition="1"
      setNames="{ all }"
      objectPath="ElementRegions"
```

(continues on next page)

(continued from previous page)

```
      fieldName="temperature"
      scale="100"/>

    <FieldSpecification
      name="farfieldTemperature"
      setNames="{ rpos }"
      objectPath="faceManager"
      fieldName="temperature"
      scale="100"/>

    <FieldSpecification
      name="innerTemperature"
      setNames="{ rneg }"
      objectPath="faceManager"
      fieldName="temperature"
      scale="-20.0"/>
```

It is important to remark that the initial effective stress of each layers must be set with accordance to the initial temperature: $\sigma_0 = 3K\alpha\delta T_0$ where σ_0 is the initial effective principal stress, δT_0 is the initial temperature change, K is the drained bulk modulus and α is the drained linear thermal expansion coefficient of the materials.

Zero pore pressure is set everywhere to simulate a thermoelastic problem in which fluid flow is ignored:

```
    <FieldSpecification
      name="zeroPressure"
      setNames="{ all }"
      objectPath="ElementRegions"
      fieldName="pressure"
      scale="0e6"/>

    <FieldSpecification
      name="sourcePressure"
      setNames="{ rneg }"
      objectPath="faceManager"
      fieldName="pressure"
      scale="0"/>

    <FieldSpecification
      name="sinkPressure"
      setNames="{ rpos }"
      objectPath="faceManager"
      fieldName="pressure"
      scale="0"/>
```

Collecting output data

It is convenient to collect data in hdf5 format that can be easily post-processed using Python. To collect the temperature field in the three layers for all the time steps, the following XML blocks need to be defined:

```
<PackCollection
  name="temperatureCollection_casing"
  objectPath="ElementRegions/casing/casing"
  fieldName="temperature"/>
<PackCollection
  name="temperatureCollection_cement"
  objectPath="ElementRegions/cement/cement"
  fieldName="temperature"/>
<PackCollection
  name="temperatureCollection_rock"
  objectPath="ElementRegions/rock/rock"
  fieldName="temperature"/>
```

```
<TimeHistory
  name="stressHistoryOutput_casing"
  sources="{ /Tasks/stressCollection_casing }"
  filename="stressHistory_casing"/>
<TimeHistory
  name="stressHistoryOutput_cement"
  sources="{ /Tasks/stressCollection_cement }"
  filename="stressHistory_cement"/>
<TimeHistory
  name="stressHistoryOutput_rock"
  sources="{ /Tasks/stressCollection_rock }"
  filename="stressHistory_rock"/>
```

Similarly, the following blocks are needed to collect the solid stress:

```
<PackCollection
  name="stressCollection_casing"
  objectPath="ElementRegions/casing/casing"
  fieldName="casingSolid_stress"/>
<PackCollection
  name="stressCollection_cement"
  objectPath="ElementRegions/cement/cement"
  fieldName="cementSolid_stress"/>
<PackCollection
  name="stressCollection_rock"
  objectPath="ElementRegions/rock/rock"
  fieldName="rockSolid_stress"/>
```

```
<TimeHistory
  name="temperatureHistoryOutput_casing"
  sources="{ /Tasks/temperatureCollection_casing }"
  filename="temperatureHistory_casing"/>
<TimeHistory
  name="temperatureHistoryOutput_cement"
  sources="{ /Tasks/temperatureCollection_cement }"
```

(continues on next page)

(continued from previous page)

```
      filename="temperatureHistory_cement"/>
    <TimeHistory
      name="temperatureHistoryOutput_rock"
      sources="{ /Tasks/temperatureCollection_rock }"
      filename="temperatureHistory_rock"/>
```

The displacement field can be collected for the whole domain using `nodeManager` as follows

```
    <PackCollection
      name="displacementCollection"
      objectPath="nodeManager"
      fieldName="totalDisplacement"/>
```

```
    <TimeHistory
      name="displacementHistoryOutput"
      sources="{ /Tasks/displacementCollection }"
      filename="displacementHistory"/>
```

Also, periodic events are required to trigger the collection of this data on the mesh. For example, the periodic events for collecting the displacement field are defined as:

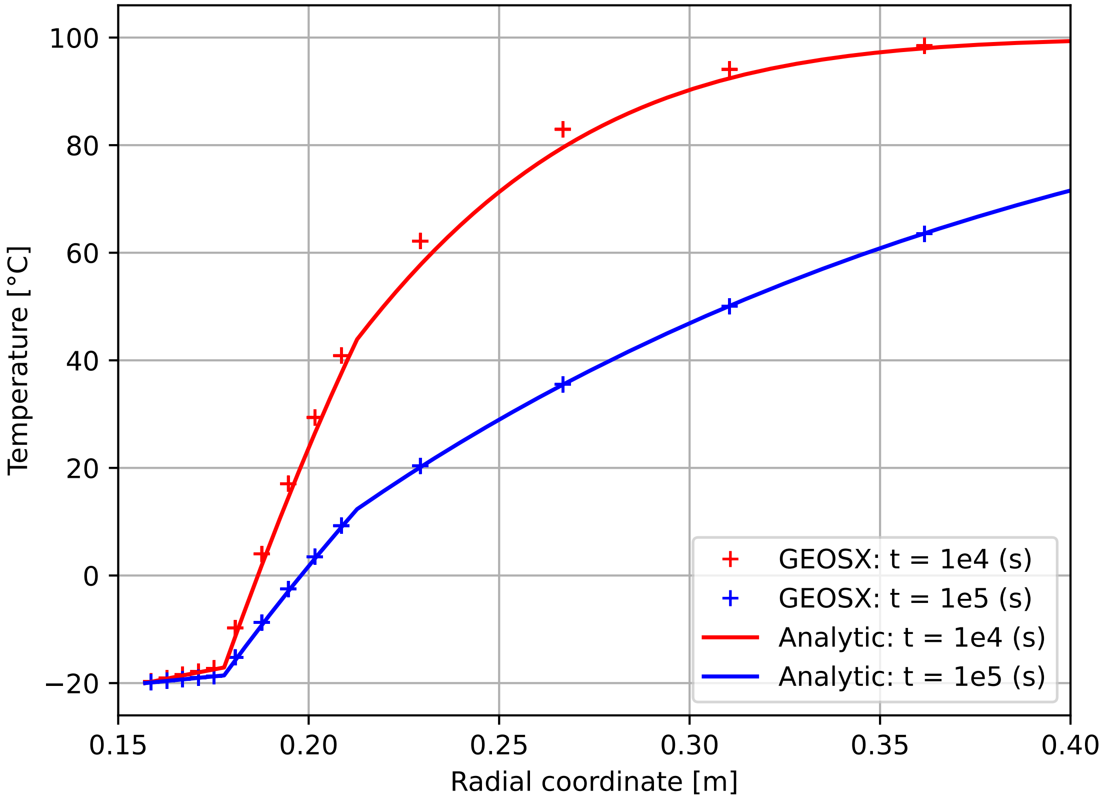
```
    <PeriodicEvent
      name="displacementHistoryCollection"
      endTime="1e5"
      forceDt="1e4"
      target="/Tasks/displacementCollection"/>
    <PeriodicEvent
      name="displacementTimeHistoryOutput"
      endTime="1e5"
      forceDt="1e4"
      target="/Outputs/displacementHistoryOutput"/>
```

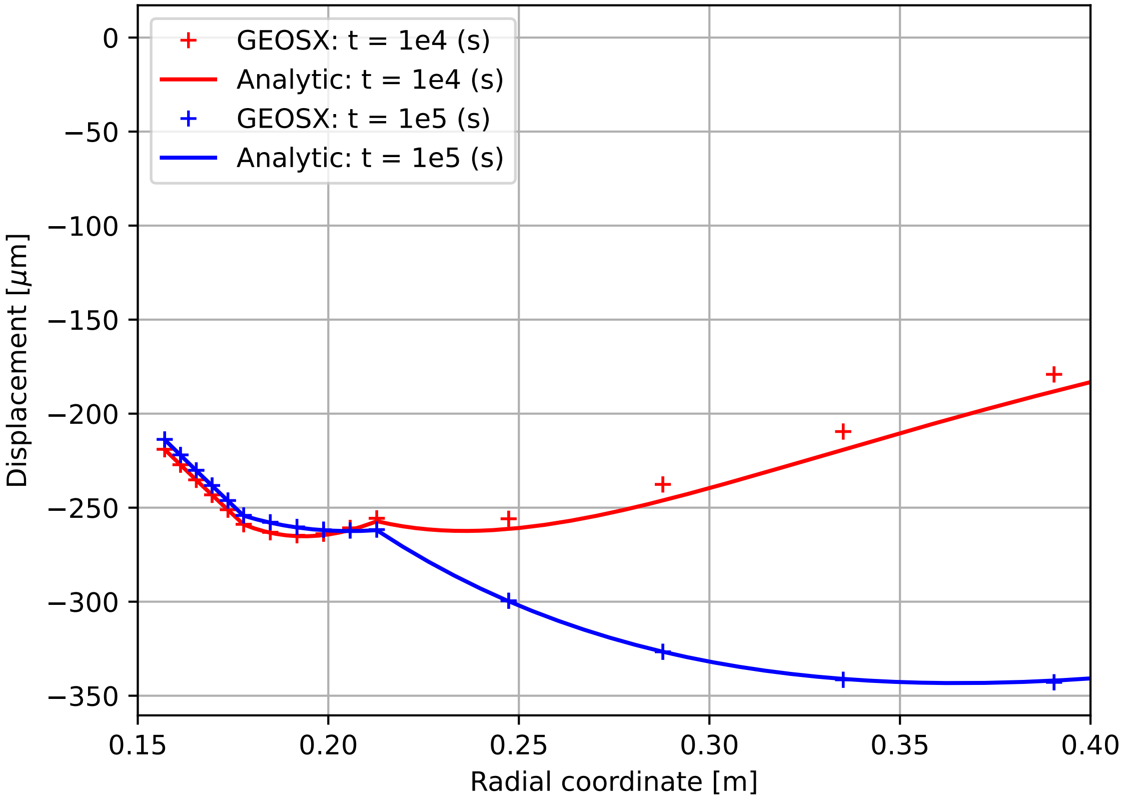
Results and benchmark

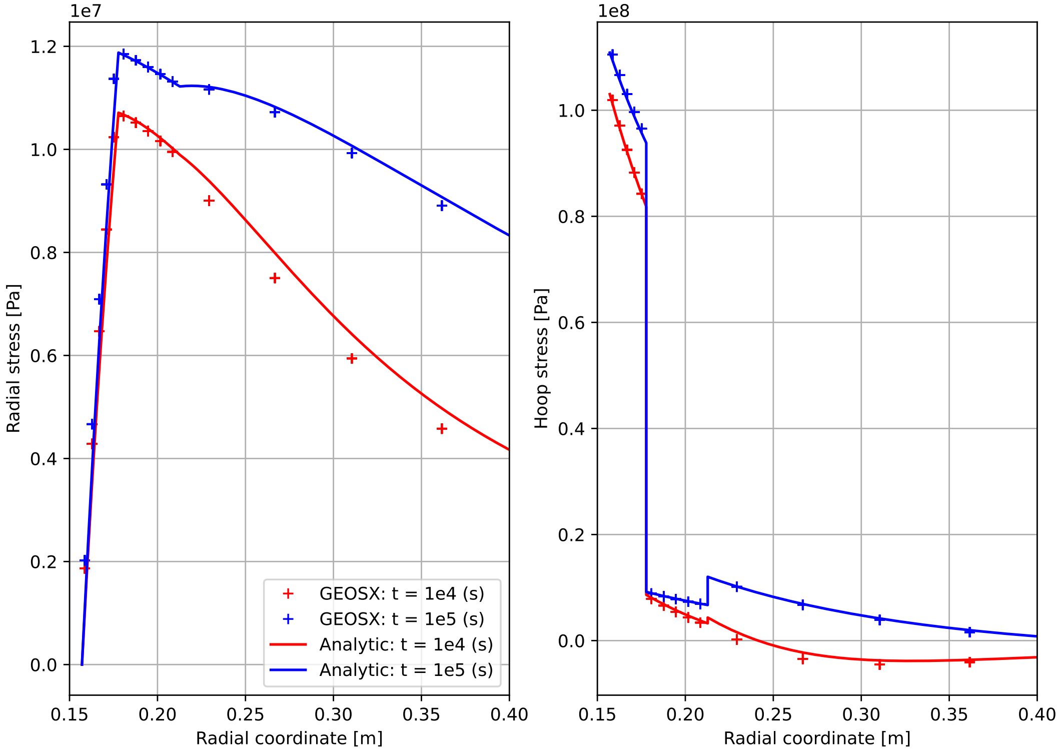
A good agreement between the GEOS results and analytical results for temperature distribution around the cased wellbore is shown in the figures below:

and the validation for the radial displacement around the cased wellbore is shown below:

The validations of the total radial and hoop stress (tangent stress) components computed by GEOS against reference results are shown in the figure below:


100
80
60
40
20
0
−20
Temperature [°C]
0.15
0.20
0.25
0.30
0.35
0.40
Radial coordinate [m]
GEOSX: t = 1e4 (s)
GEOSX: t = 1e5 (s)
Analytic: t = 1e4 (s)
Analytic: t = 1e5 (s)


GEOSX: t = 1e4 (s)
Analytic: t = 1e4 (s)
GEOSX: t = 1e5 (s)
Analytic: t = 1e5 (s)
Displacement [μm]
0
−50
−100
−150
−200
−250
−300
−350
Radial coordinate [m]
0.15
0.20
0.25
0.30
0.35
0.40


1e7
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Radial stress [Pa]
0.15
0.20
0.25
0.30
0.35
0.40
Radial coordinate [m]
GEOSX: t = 1e4 (s)
GEOSX: t = 1e5 (s)
Analytic: t = 1e4 (s)
Analytic: t = 1e5 (s)
1e8
1.0
0.8
0.6
0.4
0.2
0.0
Hoop stress [Pa]
0.15
0.20
0.25
0.30
0.35
0.40
Radial coordinate [m]

To go further

Feedback on this example

This concludes the cased wellbore example. For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

ThermoPoroElastic Wellbore Problem

Problem description

This example uses the thermal option of the `SinglePhasePoromechanics` solver to handle an open wellbore problem subjected to a uniform temperature change on its inner surface. Isotropic linear thermoporoelastic behavior is considered for the rock formation around the wellbore. Plane strain and axisymmetric conditions are assumed.

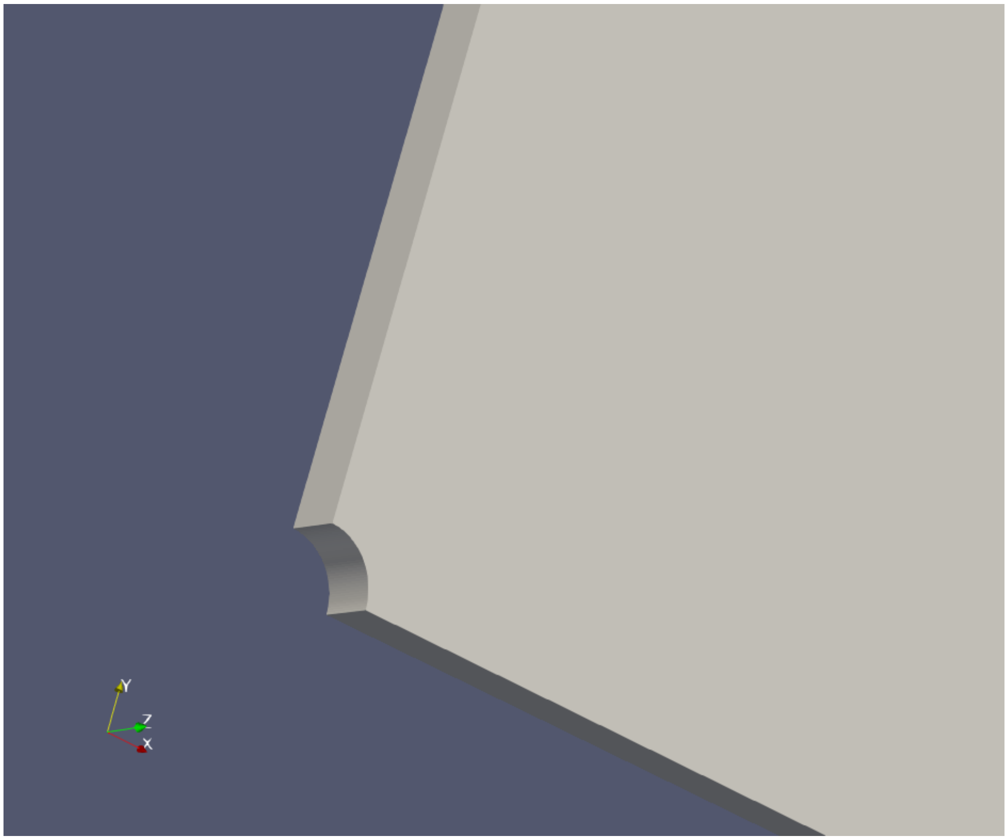


Fig. 1.59: Sketch of a thermoporoelastic wellbore

Analytical solutions to this problem were first derived by (Wang and Papamichos 1994) using a one-way coupling simplification. They are also reformulated for the full coupling assumption in the book of (Cheng 2016). These solutions will be considered to validate GEOS results.

Input file

This benchmark example uses no external input files and everything required is contained within two GEOS xml files that are located at:

```
inputFiles/wellbore/ThermoPoroElasticWellbore_base.xml
```

and

```
inputFiles/wellbore/ThermoPoroElasticWellbore_benchmark.xml
```

The corresponding integrated test is

```
inputFiles/wellbore/ThermoPoroElasticWellbore_smoke.xml
```

Geometry and mesh

The internal wellbore mesh generator `InternalWellbore` is employed to create the mesh of this wellbore problem. The radii of the open wellbore and the far-field boundary of the surrounding rock formation are defined by a vector `radius`. In the tangent direction, `theta` angle is specified from 0 to 90 degrees to simulate the problem on a quarter of the wellbore geometry. The problem is under plane strain condition and therefore we only consider thermal diffusion along the radial direction within a single horizontal layer. The trajectory of the well is defined by `trajectory`, which is vertical in this case. The `autoSpaceRadialElems` parameters allow for optimally increasing the element size from the near wellbore zone to the far-field one.

```
  <Mesh>
    <InternalWellbore
      name="mesh1"
      elementTypes="{ C3D8 }"
      radius="{ 0.1, 5.0 }"
      theta="{ 0, 90 }"
      zCoords="{ 0, 0.1 }"
      nr="{ 100 }"
      nt="{ 40 }"
      nz="{ 1 }"
      trajectory="{ { 0.0, 0.0, 0.0 },
                    { 0.0, 0.0, 0.1 } }"
      autoSpaceRadialElems="{ 1 }"
      cellBlockNames="{ rock }"
      />
  </Mesh>
```

Material properties

The bulk and shear drained elastic moduli of rock as well as its drained linear thermal expansion coefficient relating stress change to temperature variation are defined within the `Constitutive` tag as follows:

```
    <ElasticIsotropic
      name="rockSolid"
      defaultDensity="2700"
      defaultBulkModulus="20.7e9"
      defaultShearModulus="12.4e9"
      defaultThermalExpansionCoefficient="4e-5"/>
```

Here the solid density is also defined, but it is not used as the gravitational effect is ignored in this example. The porosity and the elastic bulk modulus K_s of the solid skeleton are defined as:

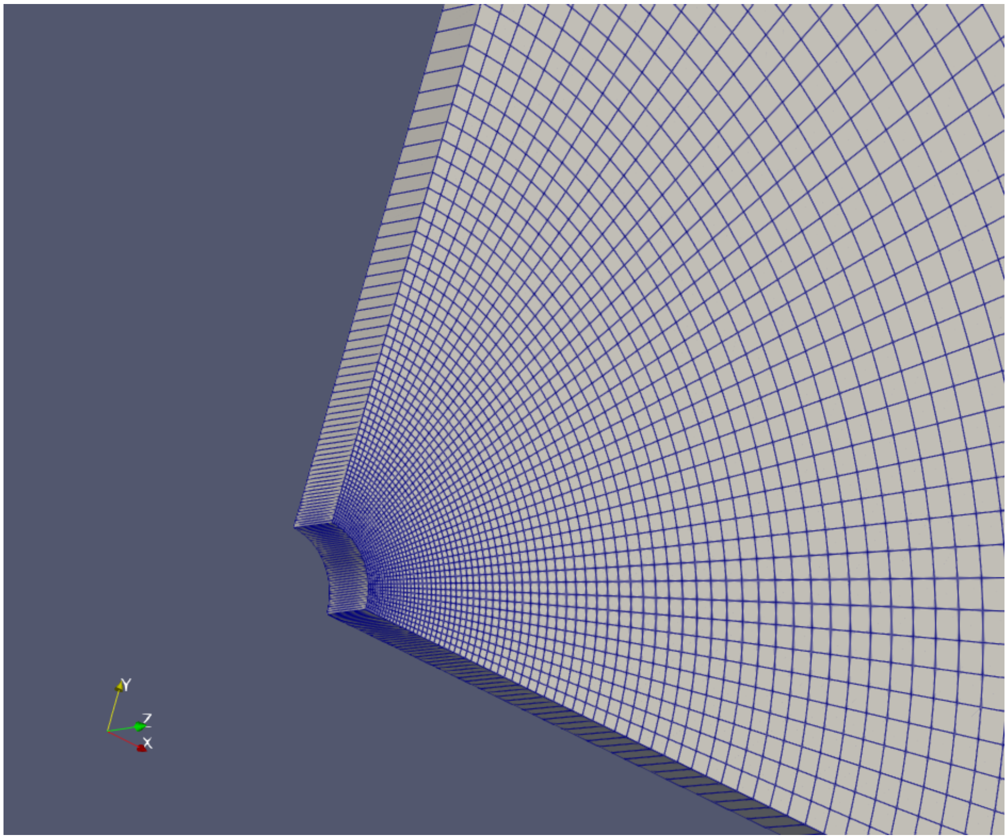


Fig. 1.60: An optimized mesh for the open wellbore.

```
<BiotPorosity
  name="rockPorosity"
  defaultReferencePorosity="0.001"
  grainBulkModulus="23.5e9"
  defaultThermalExpansionCoefficient="4e-5"/>
```

The thermal conductivities and the volumetric heat capacities of rock are defined by following XML blocks:

```
<SinglePhaseConstantThermalConductivity
  name="rockThermalCond"
  thermalConductivityComponents="{ 6.6, 6.6, 6.6 }"/>
```

and

```
<SolidInternalEnergy
  name="rockInternalEnergy"
  volumetricHeatCapacity="1.89e6"
  referenceTemperature="0"
  referenceInternalEnergy="0"/>
```

The permeability of rock is defined by:

```
<ConstantPermeability
  name="rockPerm"
  permeabilityComponents="{ 1.0e-21, 1.0e-21, 1.0e-21 }"/>
```

Fluid properties such as viscosity, thermal expansion coefficient, etc. are defined by the XML block below. A negligible volumetric heat capacity is defined for fluid to ignore the thermal convection effect. This way, only thermal transfer via the diffusion phenomenon is considered.

```
<ThermalCompressibleSinglePhaseFluid
  name="fluid"
  defaultDensity="1000"
  defaultViscosity="1e-3"
  referencePressure="0.0"
  referenceTemperature="20.0"
  compressibility="5e-10"
  thermalExpansionCoeff="3e-4"
  viscosibility="0.0"
  volumetricHeatCapacity="1"
  referenceInternalEnergy="1"/>
```

Boundary conditions

The mechanical boundary conditions are applied to ensure the axisymmetric plane strain conditions such as:

```
<FieldSpecification
  name="tNegConstraint"
  objectPath="nodeManager"
  fieldName="totalDisplacement"
  component="1"
  scale="0.0"
```

(continues on next page)

(continued from previous page)

```
      setNames="{ tneg }"/>

   <FieldSpecification
      name="tPosConstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="0"
      scale="0.0"
      setNames="{ tpos }"/>

    <FieldSpecification
      name="zconstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="2"
      scale="0.0"
      setNames="{ zneg, zpos }"/>
```

Besides, the far-field boundary is assumed to be fixed because the local changes on the wellbore must have negligible effect on the far-field boundary.

```
    <FieldSpecification
      name="rPosConstraint_x"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="0"
      scale="0.0"
      setNames="{ rpos }"/>

    <FieldSpecification
      name="rPosConstraint_y"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="1"
      scale="0.0"
      setNames="{ rpos }"/>
```

The traction-free condition on the inner surface of the wellbore is defined by:

```
    <Traction
      name="innerTraction"
      objectPath="faceManager"
      tractionType="normal"
      scale="0.0e6"
      setNames="{ rneg }"/>
```

The initial temperature (that is also the far-field boundary temperature) and the temperature applied on the inner surface of the wellbore are defined as

```
    <FieldSpecification
      name="initialTemperature"
      initialCondition="1"
      setNames="{ all }"
```

(continues on next page)

(continued from previous page)

```
      objectPath="ElementRegions"
      fieldName="temperature"
      scale="0"/>

    <FieldSpecification
      name="farfieldTemperature"
      setNames="{ rpos }"
      objectPath="faceManager"
      fieldName="temperature"
      scale="0"/>

    <FieldSpecification
      name="innerTemperature"
      setNames="{ rneg }"
      objectPath="faceManager"
      fieldName="temperature"
      scale="100.0"/>
```

It is important to remark that the initial effective stress of rock must be set with accordance to the initial temperature change: $\sigma_0 = 3K\alpha\delta T_0$ where σ_0 is the initial effective principal stress, δT_0 is the initial temperature change, K is the drained bulk modulus and α is the drained linear thermal expansion coefficient of the materials. In this example, the initial effective stresses are set to zero because the initial temperature change is set to zero.

The initial and boundary conditions for pore pressure are defined in the block below:

```
    <FieldSpecification
      name="initialPressure"
      initialCondition="1"
      setNames="{ all }"
      objectPath="ElementRegions"
      fieldName="pressure"
      scale="0e6"/>

    <FieldSpecification
      name="innerPressure"
      setNames="{ rneg }"
      objectPath="faceManager"
      fieldName="pressure"
      scale="0e6"/>

    <FieldSpecification
      name="farfieldPressure"
      setNames="{ rpos }"
      objectPath="faceManager"
      fieldName="pressure"
      scale="0"/>
```

Collecting output data

It is convenient to collect data in hdf5 format that can be easily post-processed using Python. To collect the temperature field for all the time steps, the following XML blocks need to be defined:

```
    <PackCollection
      name="temperatureCollection_rock"
      objectPath="ElementRegions/rock/rock"
      fieldName="temperature"/>
```

```
    <TimeHistory
      name="temperatureHistoryOutput_rock"
      sources="{ /Tasks/temperatureCollection_rock }"
      filename="temperatureHistory_rock"/>
```

Similarly, the following blocks are needed to collect the effective stress field across the domain:

```
    <PackCollection
      name="stressCollection_rock"
      objectPath="ElementRegions/rock/rock"
      fieldName="rockSolid_stress"/>
```

```
    <TimeHistory
      name="stressHistoryOutput_rock"
      sources="{ /Tasks/stressCollection_rock }"
      filename="stressHistory_rock"/>
```

The displacement field can be collected using `nodeManager` as follows

```
    <PackCollection
      name="displacementCollection"
      objectPath="nodeManager"
      fieldName="totalDisplacement"/>
```

```
    <TimeHistory
      name="displacementHistoryOutput"
      sources="{ /Tasks/displacementCollection }"
      filename="displacementHistory"/>
```

Also, periodic events are required to trigger the collection of this data during the entire simulation. For example, the periodic events for collecting the displacement field are defined as:

```
    <PeriodicEvent
      name="displacementHistoryCollection"
      beginTime="0"
      endTime="360"
      forceDt="60"
      target="/Tasks/displacementCollection"/>
    <PeriodicEvent
      name="displacementTimeHistoryOutput_1"
      beginTime="0"
      endTime="360"
      forceDt="60"
```

(continues on next page)

(continued from previous page)

```
      target="/Outputs/displacementHistoryOutput"/>
    <PeriodicEvent
      name="displacementHistoryCollection_2"
      beginTime="360"
      endTime="3700"
      forceDt="360"
      target="/Tasks/displacementCollection"/>
    <PeriodicEvent
      name="displacementTimeHistoryOutput_2"
      beginTime="360"
      endTime="3700"
      forceDt="360"
      target="/Outputs/displacementHistoryOutput"/>
```

Results and benchmark

A good agreement between the GEOS results and analytical results for temperature and pore pressure distribution around the wellbore is shown in the figures below:

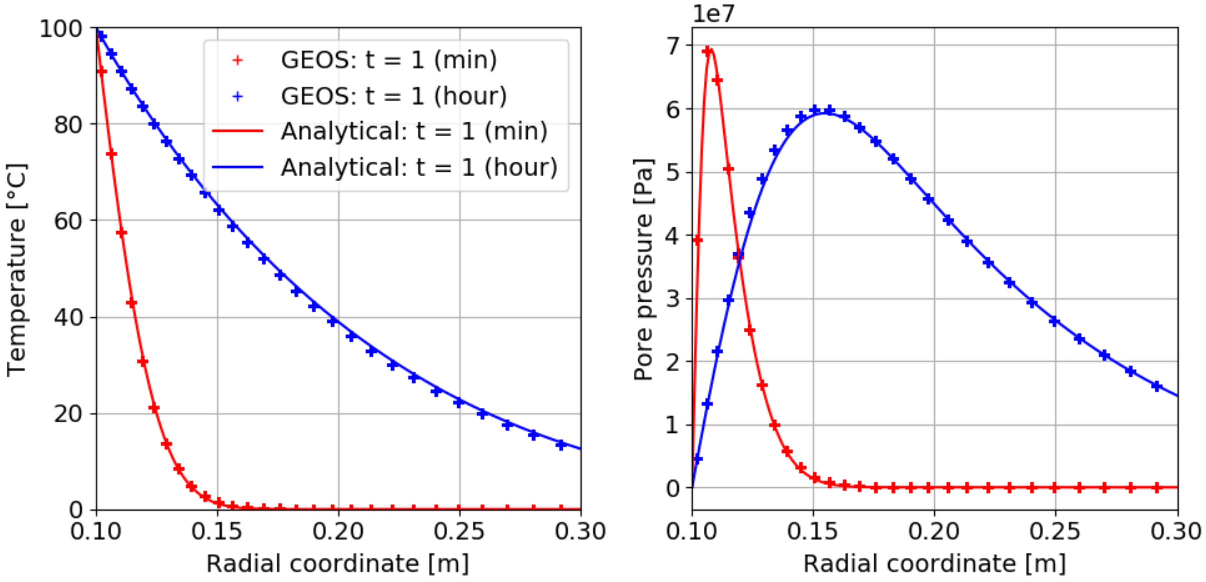


Fig. 1.61: Validation of temperature and pore pressure.

and the validation for the radial displacement around the cased wellbore is shown below:

The validations of the total radial and hoop stress (tangent stress) components computed by GEOS against reference results are shown in the figure below:

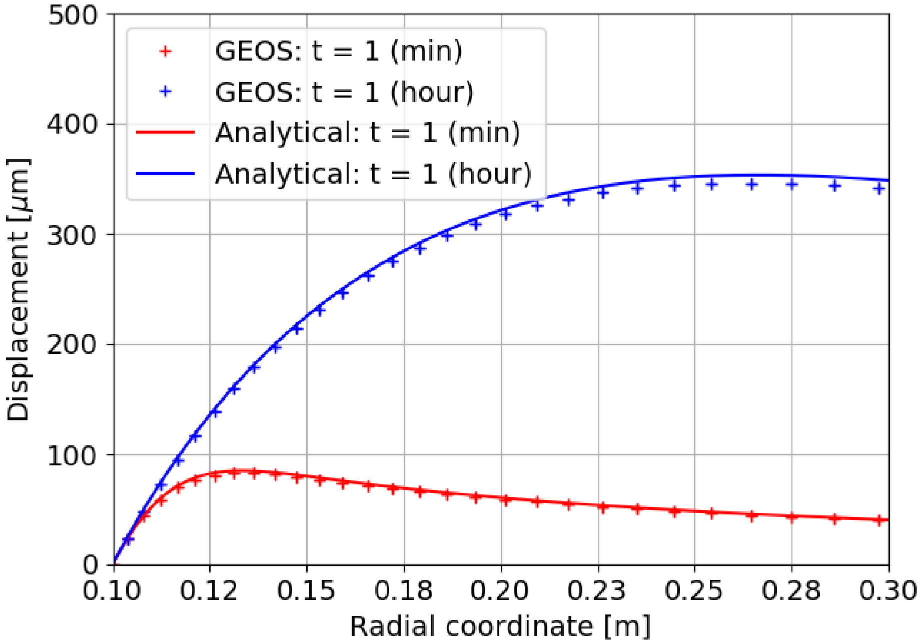


Fig. 1.62: Validation of the radial displacement.

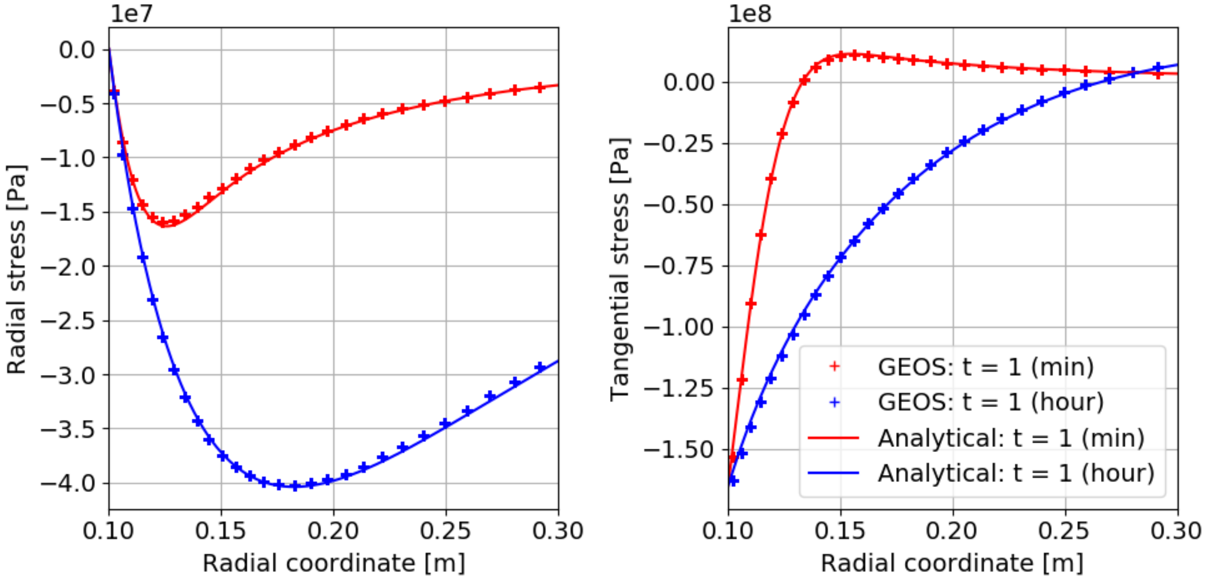


Fig. 1.63: Validation of the radial and tangent stresses.

To go further

Feedback on this example

This concludes the cased wellbore example. For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

Cased Elastic Wellbore with Imperfect Interfaces

Problem description

This example uses the `LagrangianContact` solver to handle a cased wellbore problem with imperfect contact interfaces. The completed wellbore is composed of a steel casing, a cement sheath, and rock formation. All the three materials are assumed to exhibit isotropic linear elastic behavior. The contact surfaces between these materials are simulated using a Lagrangian contact model.

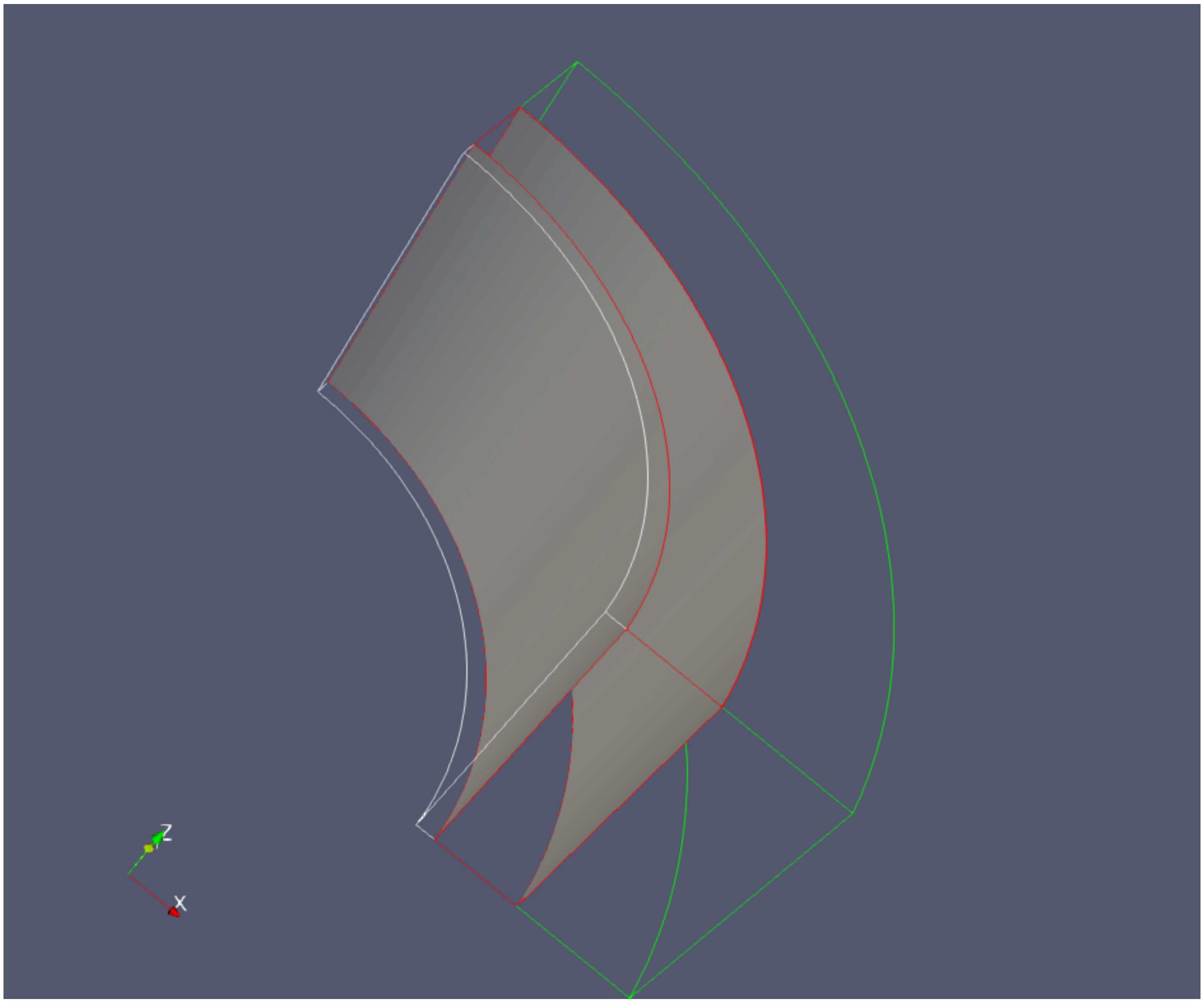


Fig. 1.64: A cased wellbore with imperfect casing-cement and cement-rock interfaces

Under a compressional loading in the radial direction, the imperfect contact interfaces behave like the perfect ones (see *Cased Elastic Wellbore Problem*). When a radial tension acting on the inner face of the wellbore, the casing debonds from the cement layer. Analytical results of the radial displacement u_r in the casing is expressed as (Hervé and Zaoui, 1995) :

$$u_r = Ar - \frac{B}{r}$$

where r is the radial coordinate, A and B are constants that are obtained by solving the boundary conditions, as detailed in the post-processing script. The outer face of the casing as well as the inner face of the cement layer are free of stress because of the debonding at the casing-cement interface. Therefore, the displacement jump at the cement-rock interface is nil, and the displacement jump across the casing-cement interface is equal to $u_r(r = r_{out_{c}asing})$, where $r_{out_{c}asing}$ is the outer radius of the casing.

Input file

This benchmark example does not use any external input files and everything required is contained within two GEOS XML files located at:

```
inputFiles/wellbore/CasedElasticWellbore_ImperfectInterfaces_base.xml
```

and

```
inputFiles/wellbore/CasedElasticWellbore_ImperfectInterfaces_benchmark.xml
```

The corresponding integrated test is

```
inputFiles/wellbore/CasedElasticWellbore_ImperfectInterfaces_smoke.xml
```

In this example, we should focus on following XML blocks:

Cylinder geometry

The nodesets that define the casing-cement and cement-rock interfaces are curved. In this example, we use the `Cylinder` geometry to select these nodesets. This geometry is defined by the centers of the two plane faces, `firstFaceCenter` and `secondFaceCenter`, and its inner and outer radii, `innerRadius` and `outerRadius`. Note that the inner radius is optional as it is only needed for defining a hollow cylinder (i.e. an annulus). The inner radius is required in this example to select only the nodes on the casing-cement and the cement-rock interfaces.

```
  <Geometry>
    <Cylinder
      name="casingCementInterface"
      firstFaceCenter="{ 0.0, 0.0,  -0.001 }"
      secondFaceCenter="{ 0.0, 0.0, 0.101 }"
      outerRadius="0.1061"
      innerRadius="0.1059"
    />

    <Cylinder
      name="cementRockInterface"
      firstFaceCenter="{ 0.0, 0.0,  -0.001 }"
      secondFaceCenter="{ 0.0, 0.0, 0.101 }"
      outerRadius="0.1331"
      innerRadius="0.1329"
    />
  </Geometry>
```

Events

In this example, we need to define a solo event for generating the imperfect contact surfaces as shown below:

```
    <SoloEvent
      name="preFracture"
      target="/Solvers/SurfaceGen"/>
```

where the surface generation solver is defined as follows:

```
    <SurfaceGenerator
      name="SurfaceGen"
      fractureRegion="Fracture"
      targetRegions="{ casing, cement, rock }"
      rockToughness="1.0e6"
      mpiCommOrder="1"/>
```

Here, `rockToughness` is defined by default but has been omitted in this simulation.

To collect the displacement jump across the imperfect interfaces, we also define two periodic events as shown below:

```
    <PeriodicEvent
      name="displacementJumpHistoryCollection"
      endTime="2.0"
      forceDt="0.1"
      target="/Tasks/displacementJumpCollection"/>
    <PeriodicEvent
      name="displacementJumpTimeHistoryOutput"
      endTime="2.0"
      forceDt="0.1"
      target="/Outputs/displacementJumpHistoryOutput"/>
```

The corresponding `Tasks` and `Outputs` targets must be defined in conjunction with these events.

Numerical Methods

The `stabilizationName` that is required in the `LagrangianContact` solver is defined by:

```
    <FiniteVolume>
      <TwoPointFluxApproximation
        name="TPFAstabilization"/>
    </FiniteVolume>
```

Contact region and material

The imperfect contact surfaces between casing, cement, and rock layers are defined as `Fracture` as shown below:

```
  <ElementRegions>
    <SurfaceElementRegion
      name="Fracture"
      faceBlock="faceElementSubRegion"
```

(continues on next page)

(continued from previous page)

```
      defaultAperture="0"
      materialList="{ fractureContact }"/>
```

Here, the `faceBlock` name, `faceElementSubRegion`, is needed to define `Tasks` for collecting displacement jumps across the contact surfaces. The `defaultAperture` defined in this block is the default hydraulic aperture that should not be confused with the mechanical aperture. For this purely mechanical problem, the default hydraulic aperture parameter is omitted. The fracture material given in the `materialList` is defined as follows:

```
    <Coulomb
      name="fractureContact"
      cohesion="0"
      frictionCoefficient="0.5"
      penaltyStiffness="1.0e8"
      apertureTableName="apertureTable"/>
```

For this purely mechanical problem, without fluid flow and shearing stress acting on the contact surface, all the parameters defined in this block are omitted.

Results and benchmark

The GEOS results of displacement jump across the casing-cement and cement-rock interfaces are shown in the figure below:

As expected, we observe a zero-displacement jump at the cement-rock interface under a tension stress on the inner surface of the casing. Indeed, the stress applied here does not cause any strain on the cement and rock layers after debonding has occurred at the casing-cement interface. The displacement jump at the casing-cement interface is homogeneous and varies over time because the tension stress on the inner surface of the casing varies with time, as defined in the XML file. A perfect comparison between GEOS results and theoretical results is shown in the figure below:

To go further

Feedback on this example

This concludes the cased wellbore example. For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

Viscoplasticity

Drucker-Prager Model: Triaxial Driver versus Semi-Analytical Solution

Problem description

This example uses the Triaxial Driver to simulate an elasto-plastic triaxial compression test of a Drucker-Prager solid. Constant lateral confining stress together with loading/unloading axial strain periods are imposed. Imposed axial strain range are high enough for allowing plastic yield in both loading and unloading period. This complicated scenario is used for verifying the numerical convergence and accuracy of the Drucker-Prager constitutive model implemented in GEOS.

Semi-analytical results for axial stress variations $\Delta\sigma_V$ and lateral strain variations $\Delta\varepsilon_V$ can be established for the imposed triaxial boundary conditions:

$$\Delta\sigma_V = \Delta\varepsilon_V E_{ep}$$

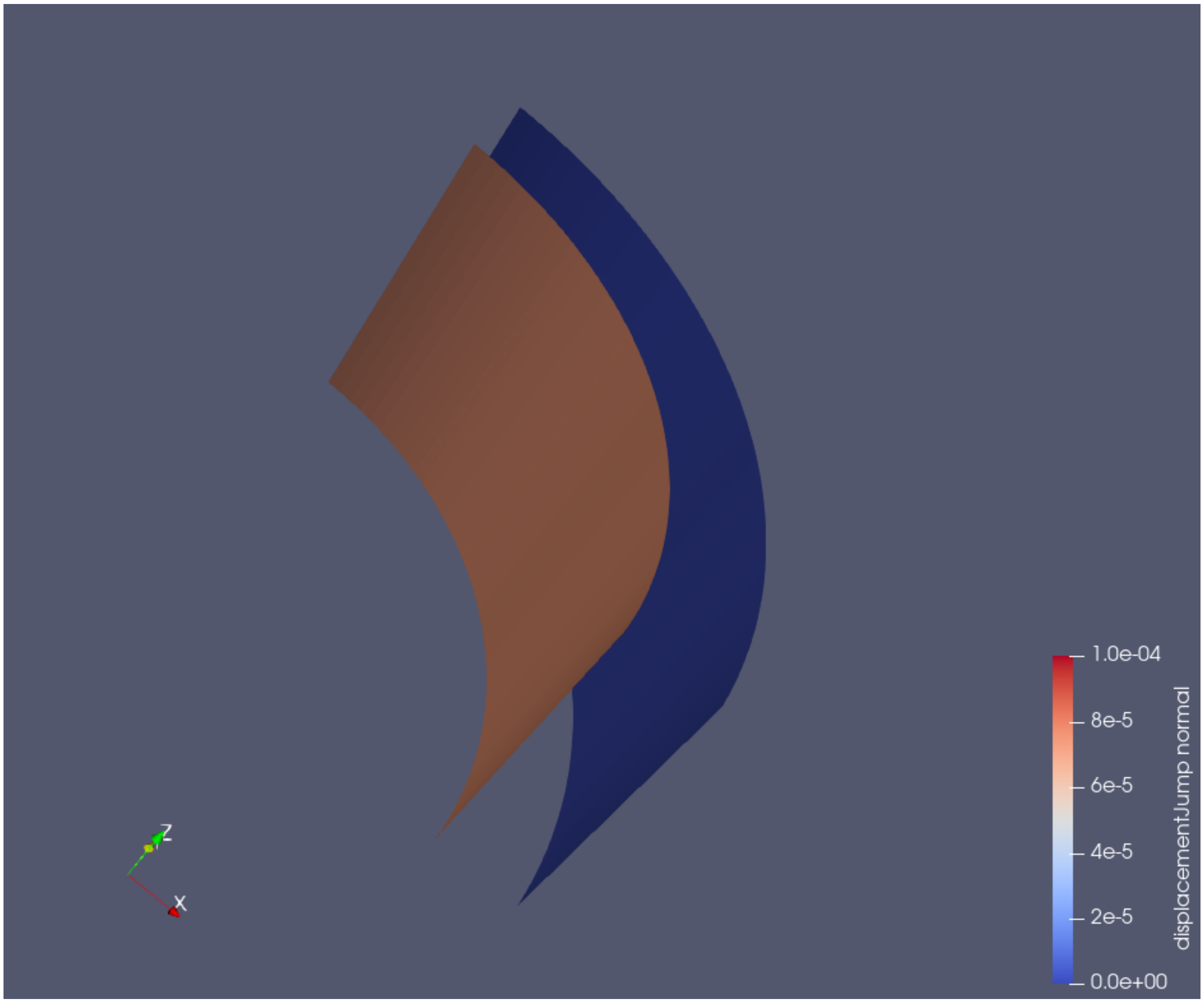
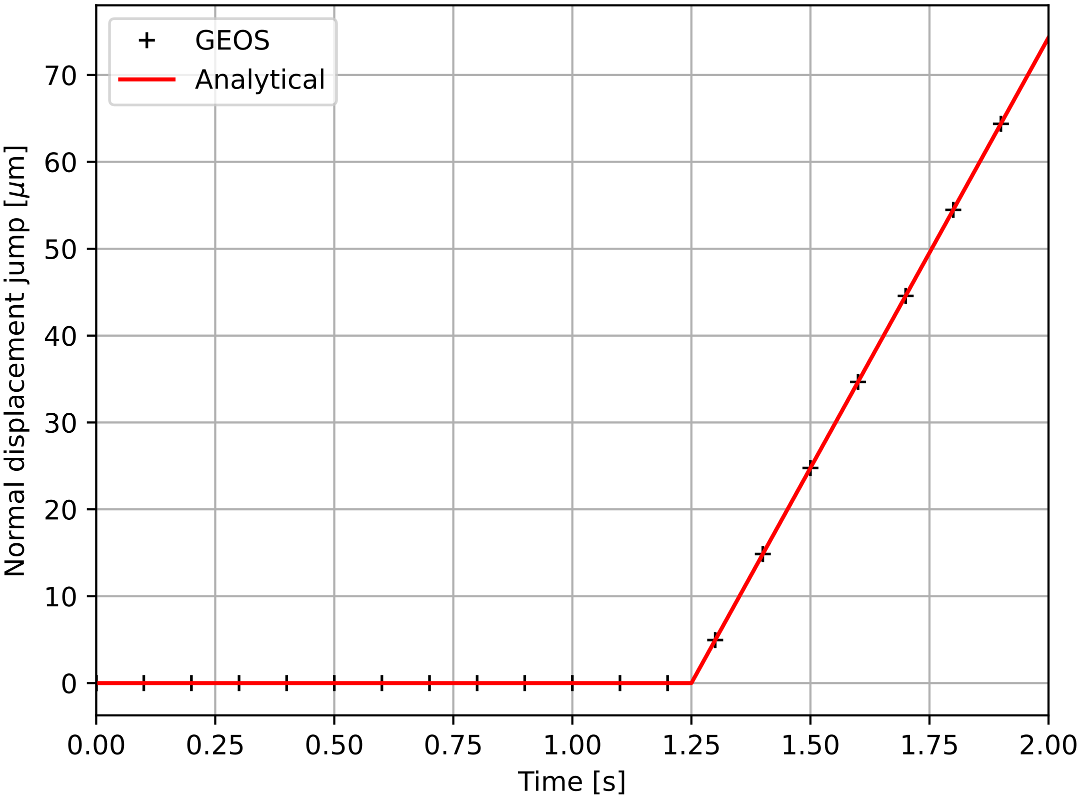


Fig. 1.65: Displacement jumps across the casing-cement and cement-rock interfaces


GEOS
Analytical
Normal displacement jump [μm]
0
10
20
30
40
50
60
70
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
Time [s]

$$\Delta\varepsilon_H = \Delta\varepsilon_V - \frac{\Delta\sigma_V}{E'_{ep}}$$

where E_{ep} and E'_{ep} are elasto-plastic Young moduli that can be obtained from the elastic Young and shear moduli (E and μ), the frictional parameter b, the dilation parameter b' and the hardening rate h of the Drucker-Prager model by:

$$\frac{1}{E_{ep}} = \frac{1}{E} + \frac{(b'-3)(b-3)}{9h}$$

$$\frac{1}{E'_{ep}} = \frac{1}{2\mu} - \frac{b-3}{2h}$$

These solutions are applied only when the plastic yield condition is satisfied. The cohesion parameter defining the plastic yield surface is updated with stress changes as

$$\Delta a = \frac{b-3}{3}\Delta\sigma_V$$

These solutions were established for a positive shear stress $q = -(\sigma_V - \sigma_H)$ (negative sign convention for compressional stress). For the case when the plastic yield occurs at a negative shear stress, we have:

$$\frac{1}{E_{ep}} = \frac{1}{E} + \frac{(b'+3)(b+3)}{9h}$$

$$\frac{1}{E'_{ep}} = \frac{1}{2\mu} + \frac{b+3}{2h}$$

and

$$\Delta a = \frac{b+3}{3}\Delta\sigma_V$$

These solutions are implemented in a Python script associated to this example for verifying GEOS results.

Input files

This validation example uses two GEOS xml files that are located at:

```
inputFiles/triaxialDriver/triaxialDriver_base.xml
```

and

```
inputFiles/triaxialDriver/triaxialDriver_DruckerPrager.xml
```

It also uses a set of table files located at:

```
inputFiles/triaxialDriver/tables/
```

A Python script for the semi-analytical solutions presented above as well as for post-processing the GEOS results is provided at:

```
src/docs/sphinx/advancedExamples/validationStudies/viscoplasticity/DruckerPrager/
↪TriaxialDriver_vs_SemiAnalytic_DruckerPrager.py
```

For this example, we focus on the `Task` and the `Constitutive` tags.

Task

The imposed axial strain loading/unloading periods, the constant lateral confining stress, and the initial stress are defined in the Task block as:

```
<Tasks>
  <TriaxialDriver
    name="triaxialDriver"
    material="DruckerPrager"
    mode="mixedControl"
    axialControl="strainFunction"
    radialControl="stressFunction"
    initialStress="-10.e6"
    steps="200"
    output="DruckerPragerResults.txt" />
</Tasks>
```

Constitutive laws

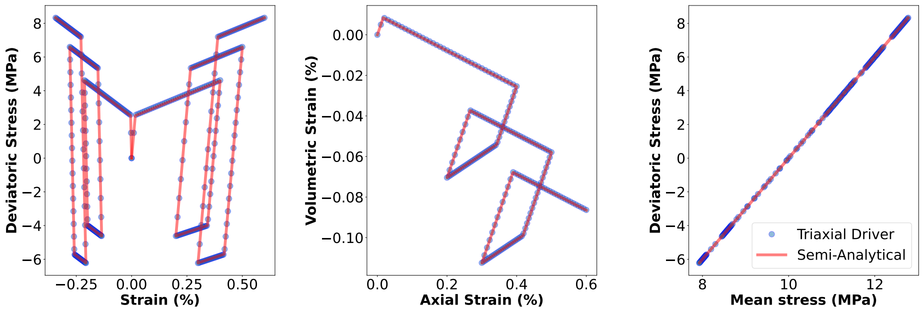
The elasto-plastic parameters are defined as:

```
  <DruckerPrager
    name="DruckerPrager"
    defaultDensity="2700"
    defaultBulkModulus="10.0e9"
    defaultShearModulus="6.0e9"
    defaultCohesion="0.1e6"
    defaultFrictionAngle="6.0"
    defaultDilationAngle="3.0"
    defaultHardeningRate="0.5e9"
  />
```

All constitutive parameters such as density, viscosity, and bulk and shear moduli are specified in the International System of Units.

A comparison between GEOS results and semi-analytical results

The simulation results are saved in a text file, named DruckerPragerResults.txt. A perfect comparison between the results given by the TriaxialDriver solver in GEOS and the semi-analytical results presented above is shown below.



To go further

Feedback on this example

For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

Visco Drucker-Prager Model: Triaxial Driver versus Semi-Analytical Solution

Problem description

This example uses the Triaxial Driver to simulate a triaxial compression test of a Visco Drucker-Prager solid. Constant lateral confining stress together with loading/unloading axial strain periods are imposed. Imposed axial strain range are high enough for allowing visco-plastic yield in both loading and unloading period. This complicated scenario is used for verifying the numerical convergence and accuracy of the Visco Drucker-Prager constitutive model implemented in GEOS.

Semi analytical result for axial stress variation $\Delta\sigma_V$ and lateral strain variation $\Delta\varepsilon_V$ can be established for the imposed triaxial boundary conditions following the theoretical basis of the Perzyna time dependent approach presented by (Runesson et al. 1999) as:

$$\Delta\sigma_V = (\Delta\varepsilon_V - \Delta\lambda\frac{b' - 3}{3})E$$

$$\Delta\varepsilon_H = \Delta\varepsilon_V - \frac{\Delta\sigma_V}{2\mu} + \frac{3}{2}\Delta\lambda$$

where E and μ are the elastic Young and shear moduli. The visco-plastic multiplier $\Delta\lambda$ can be approximated by:

$$\Delta\lambda = \frac{\Delta t}{t_*}\frac{F}{3\mu + Kbb' + h}$$

in which Δt is the time increment, t_* is the relaxation time, F is the stress function defining the visco-plastic yield surface, K is the elastic bulk modulus, b is the frictional parameter defining the visco-plastic yield surface, b' is the dilation parameter defining the plastic potential and h is the hardening rate. These solutions are applied only when plastic yield condition is satisfied. The cohesion parameter defining the plastic yield surface is updated with stress change as

$$\Delta a = h\Delta\lambda$$

These solutions were established for a positive shear stress $q = -(\sigma_V - \sigma_H)$ (negative sign convention for compression stress). For the case when the plastic yield occurs at a negative shear stress, we have

$$\Delta\sigma_V = (\Delta\varepsilon_V - \Delta\lambda\frac{b' + 3}{3})E$$

$$\Delta\varepsilon_H = \Delta\varepsilon_V - \frac{\Delta\sigma_V}{2\mu} - \frac{3}{2}\Delta\lambda$$

These solutions are implemented in a Python script associated to this example for verifying GEOS results.

Input files

This benchmark example uses two GEOS xml files that are located at:

```
inputFiles/triaxialDriver/triaxialDriver_base.xml
```

and

```
inputFiles/triaxialDriver/triaxialDriver_ViscoDruckerPrager.xml
```

It also uses a set of table files located at:

```
inputFiles/triaxialDriver/tables/
```

A Python script for the semi-analytical solutions presented above as well as for post-processing the GEOS results is provided at:

```
src/docs/sphinx/advancedExamples/validationStudies/viscoplasticity/ViscoDruckerPrager/
↪TriaxialDriver_vs_SemiAnalytic_ViscoDruckerPrager.py
```

For this example, we focus on the `Task` and the `Constitutive` tags.

Task

The imposed axial strain loading/unloading periods, the constant lateral confining stress as well as the initial stress are defined in the `Task` block as

```
  <Tasks>
    <TriaxialDriver
      name="triaxialDriver"
      material="ViscoDruckerPrager"
      mode="mixedControl"
      axialControl="strainFunction"
      radialControl="stressFunction"
      initialStress="-10.e6"
      steps="200"
      output="ViscoDruckerPragerResults.txt" />
  </Tasks>
```

Constitutive laws

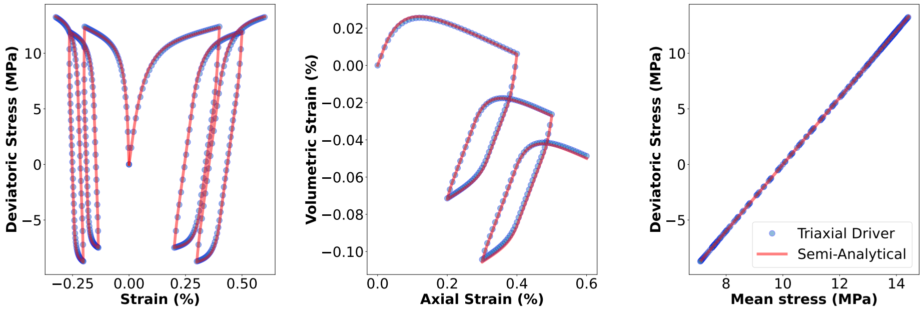
The elasto-visco-plastic parameters are defined as

```
    <ViscoDruckerPrager
      name="ViscoDruckerPrager"
      defaultDensity="2700"
      defaultBulkModulus="10.0e9"
      defaultShearModulus="6.0e9"
      defaultCohesion="0.1e6"
      defaultFrictionAngle="6.0"
      defaultDilationAngle="3.0"
      defaultHardeningRate="0.5e9"
      relaxationTime="0.1"
    />
```

All constitutive parameters such as density, viscosity, and bulk and shear moduli are specified in the International System of Units.

A comparison between GEOS results and semi-analytical results

The simulation results are saved in a text file, named `ViscoDruckerPragerResults.txt`. A comparison between the results given by the TriaxialDriver solver in GEOS and the approximated semi-analytical results presented above is shown below. Interestingly we observed that the Duvaut-Lions approach implemented in GEOS can fit perfectly with the Perzyna approach that was considered for deriving the analytical results. This consistency between these time dependence approaches is because of the linear hardening law of the considered constitutive model as already discussed by (Runesson et al. 1999) .



To go further

Feedback on this example

For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

Extended Drucker-Prager Model: Triaxial Driver versus Semi-Analytical Solution

Problem description

This example uses the Triaxial Driver to simulate an elasto-plastic triaxial compression test of an Extended Drucker-Prager solid. Constant lateral confining stress together with loading/unloading axial strain periods are imposed. Imposed axial strain range are high enough for allowing plastic yield in both loading and unloading period. This complicated scenario is used for verifying the numerical convergence and accuracy of the Extended Drucker-Prager constitutive model implemented in GEOS.

Semi-analytical results for axial stress variations $\Delta\sigma_V$ and lateral strain variations $\Delta\varepsilon_V$ can be established for the imposed triaxial boundary conditions:

$$\Delta\sigma_V = \Delta\varepsilon_V E_{ep}$$

$$\Delta\varepsilon_H = \Delta\varepsilon_V - \frac{\Delta\sigma_V}{E'_{ep}}$$

where E_{ep} and E'_{ep} are elasto-plastic Young moduli that can be obtained from the elastic Young and shear moduli (E and μ), the frictional parameter b and the dilation ratio θ of the Extended Drucker-Prager model by:

$$\frac{1}{E_{ep}} = \frac{1}{E} + \frac{(\theta b - 3)(b - 3)}{9h}$$

$$\frac{1}{E'_{ep}} = \frac{1}{2\mu} - \frac{b - 3}{2h}$$

The hardening rate h is defined by

$$h = \frac{\partial F}{\partial b} \frac{\partial b}{\partial \lambda}$$

These solutions are applied only when the plastic yield condition is satisfied. The cohesion parameter defining the plastic yield surface is updated with stress changes as:

$$\Delta\lambda = \frac{b-3}{3h} \Delta\sigma_V$$

These solutions were established for a positive shear stress $q = -(\sigma_V - \sigma_H)$ (negative sign convention for compressional stress). For the case when the plastic yield occurs at a negative shear stress, we have:

$$\frac{1}{E_{ep}} = \frac{1}{E} + \frac{(\theta b + 3)(b+3)}{9h}$$

$$\frac{1}{E'_{ep}} = \frac{1}{2\mu} + \frac{b+3}{2h}$$

and

$$\Delta\lambda = \frac{b+3}{3h} \Delta\sigma_V$$

These solutions are implemented in a Python script associated to this example for verifying GEOS results.

Input files

This validation example uses two GEOS xml files that are located at:

```
inputFiles/triaxialDriver/triaxialDriver_base.xml
```

and

```
inputFiles/triaxialDriver/triaxialDriver_ExtendedDruckerPrager.xml
```

It also uses a set of table files located at:

```
inputFiles/triaxialDriver/tables/
```

A Python script for the semi-analytical solutions presented above as well as for post-processing the GEOS results is provided at:

```
src/docs/sphinx/advancedExamples/validationStudies/viscoplasticity/ExtendedDruckerPrager/
↪TriaxialDriver_vs_SemiAnalytic_ExtendedDruckerPrager.py
```

For this example, we focus on the `Task` and the `Constitutive` tags.

Task

The imposed axial strain loading/unloading periods, the constant lateral confining stress, and the initial stress are defined in the `Task` block as:

```
  <Tasks>
    <TriaxialDriver
      name="triaxialDriver"
      material="ExtendedDruckerPrager"
```

(continues on next page)

(continued from previous page)

```
    mode="mixedControl"
    axialControl="strainFunction"
    radialControl="stressFunction"
    initialStress="-10.e6"
    steps="200"
    output="ExtendedDruckerPragerResults.txt" />
</Tasks>
```

Constitutive laws

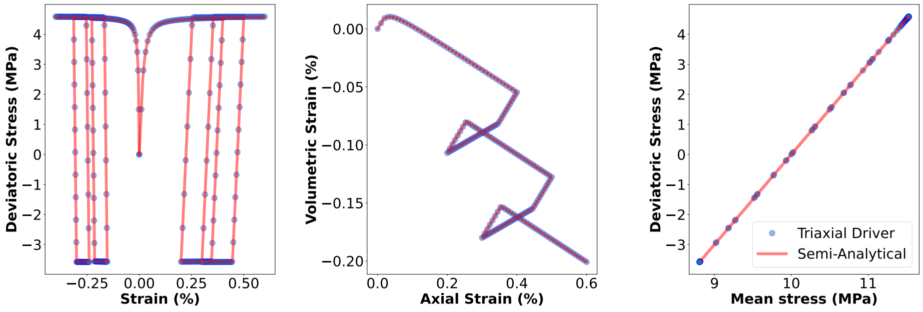
The elasto-plastic parameters are defined as:

```
  <ExtendedDruckerPrager
    name="ExtendedDruckerPrager"
    defaultDensity="2700"
    defaultBulkModulus="10.0e9"
    defaultShearModulus="6.0e9"
    defaultCohesion="0.1e6"
    defaultInitialFrictionAngle="6.0"
    defaultResidualFrictionAngle="10.0"
    defaultDilationRatio="0.5"
    defaultHardening="0.0001"
  />
```

All constitutive parameters such as density, viscosity, and bulk and shear moduli are specified in the International System of Units.

A comparison between GEOS results and semi-analytical results

The simulation results are saved in a text file, named `ExtendedDruckerPragerResults.txt`. A perfect comparison between the results given by the TriaxialDriver solver in GEOS and the semi-analytical results presented above is shown below



To go further

Feedback on this example

For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

Visco Extended Drucker-Prager Model: Triaxial Driver versus Semi-Analytical Solution

Problem description

This example uses the Triaxial Driver to simulate a triaxial compression test of a Visco Extended Drucker-Prager solid. Constant lateral confining stress together with loading/unloading axial strain periods are imposed. Imposed axial strain range are high enough for allowing plastic yield in both loading and unloading period. This complicated scenario is used for verifying the numerical convergence and accuracy of the Visco Extended Drucker-Prager constitutive model implemented in GEOS.

Semi analytical result for axial stress variation $\Delta\sigma_V$ and lateral strain variation $\Delta\varepsilon_V$ can be established for the imposed triaxial boundary conditions following the theoretical basis of the Perzyna time dependent approach presented by (Runesson et al. 1999) as:

$$\Delta\sigma_V = (\Delta\varepsilon_V - \Delta\lambda\frac{\theta b - 3}{3})E$$

$$\Delta\varepsilon_H = \Delta\varepsilon_V - \frac{\Delta\sigma_V}{2\mu} + \frac{3}{2}\Delta\lambda$$

where E and μ are the elastic Young and shear moduli. The visco-plastic multiplier $\Delta\lambda$ can be approximated by:

$$\Delta\lambda = \frac{\Delta t}{t_*}\frac{F}{3\mu + K\theta b^2 + h}$$

in which Δt is the time increment, t_* is the relaxation time, F is the stress function defining the visco-plastic yield surface, K is the elastic bulk modulus, b is the frictional parameter defining the visco-plastic yield surface, θ is the dilation ratio defining the plastic potential and h is the hardening rate. The hardening rate h is defined by

$$h = \frac{\partial F}{\partial b}\frac{\partial b}{\partial\lambda}$$

These solutions were established for a positive shear stress $q = -(\sigma_V - \sigma_H)$ (negative sign convention for compression stress). For the case when the plastic yield occurs at a negative shear stress, we have

$$\Delta\sigma_V = (\Delta\varepsilon_V - \Delta\lambda\frac{\theta b + 3}{3})E$$

$$\Delta\varepsilon_H = \Delta\varepsilon_V - \frac{\Delta\sigma_V}{2\mu} - \frac{3}{2}\Delta\lambda$$

These solutions are implemented in a Python script associated to this example for verifying GEOS results.

Input files

This validation example uses two GEOS xml files that are located at:

```
inputFiles/triaxialDriver/triaxialDriver_base.xml
```

and

```
inputFiles/triaxialDriver/triaxialDriver_ViscoExtendedDruckerPrager.xml
```

It also uses a set of table files located at:

```
inputFiles/triaxialDriver/tables/
```

A Python script for the semi-analytical solutions presented above as well as for post-processing the GEOS results is provided at:

```
src/docs/sphinx/advancedExamples/validationStudies/viscoplasticity/
↪ViscoExtendedDruckerPrager/TriaxialDriver_vs_SemiAnalytic_ViscoExtendedDruckerPrager.py
```

For this example, we focus on the `Task` and the `Constitutive` tags.

Task

The imposed axial strain loading/unloading periods, the constant lateral confining stress as well as the initial stress are defined in the `Task` block as

```
  <Tasks>
    <TriaxialDriver
      name="triaxialDriver"
      material="ViscoExtendedDruckerPrager"
      mode="mixedControl"
      axialControl="strainFunction"
      radialControl="stressFunction"
      initialStress="-10.e6"
      steps="200"
      output="ViscoExtendedDruckerPragerResults.txt" />
  </Tasks>
```

Constitutive laws

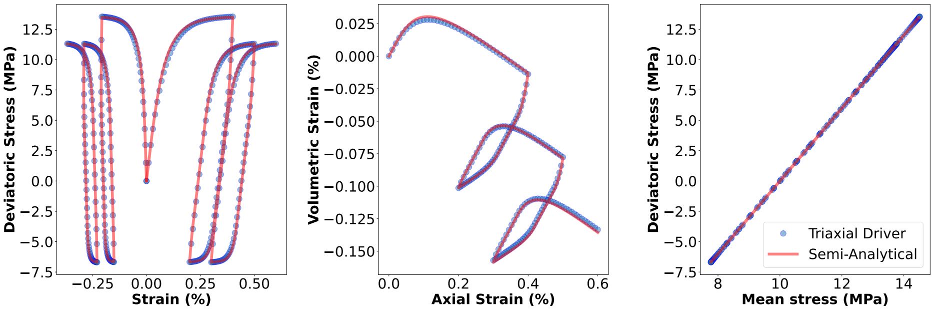
The elasto-visco-plastic parameters are defined as

```
    <ViscoExtendedDruckerPrager
      name="ViscoExtendedDruckerPrager"
      defaultDensity="2700"
      defaultBulkModulus="10.0e9"
      defaultShearModulus="6.0e9"
      defaultCohesion="0.1e6"
      defaultInitialFrictionAngle="6.0"
      defaultResidualFrictionAngle="10.0"
      defaultDilationRatio="0.5"
      defaultHardening="0.0001"
      relaxationTime="0.1"
    />
```

All constitutive parameters such as density, viscosity, and bulk and shear moduli are specified in the International System of Units.

A comparison between GEOS results and semi-analytical results

The simulation results are saved in a text file, named `ViscoExtendedDruckerPragerResults.txt`. A comparison between the results given by the TriaxialDriver solver in GEOS and the approximated semi-analytical results presented above is shown below. Interestingly we observed that the Duvaut-Lions approach implemented in GEOS can fit perfectly with the Perzyna approach that was considered for deriving the analytical results. This consistency between these time dependence approaches is because of the linear hardening law of the considered constitutive model as already discussed by (Runesson et al. 1999) .



To go further

Feedback on this example

For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

Modified CamClay Model: Triaxial Driver versus Semi-Analytical Solution

Problem description

This example uses the Triaxial Driver to simulate an elasto-plastic oedometric compression test of a Modified CamClay solid. Oedometric condition with zero lateral strain together with loading/unloading axial strain periods are imposed. Semi-analytical results for the mean and shear stress variations Δp and Δq can be derived from the imposed vertical strain variation by solving the following equation system:

$$\Delta\varepsilon_V = \Delta p(\frac{1}{K} + \frac{1}{h}\frac{\partial F}{\partial p}\frac{\partial G}{\partial p}) + \Delta q \frac{1}{h}\frac{\partial F}{\partial q}\frac{\partial G}{\partial p}$$

$$\Delta\varepsilon_V = \Delta p \frac{3}{2h}\frac{\partial F}{\partial p}\frac{\partial G}{\partial q} + \Delta q(\frac{1}{2\mu} + \frac{1}{h}\frac{\partial F}{\partial q}\frac{\partial G}{\partial q})$$

where K and μ are elastic bulk and shear moduli, F and G are the plastic yield surface and the plastic potential, and h the is hardening rate defined by:

$$h = -\frac{\partial F}{\partial \varepsilon_{vol}^{vp}}\frac{\partial G}{\partial p}$$

in which ε_{vol}^{vp} is the volumetric visco-plastic strain. These solutions are implemented in a Python script associated to this example for verifying GEOS results.

Input files

This validation example uses two GEOS xml files that are located at:

```
inputFiles/triaxialDriver/triaxialDriver_base.xml
```

and

```
inputFiles/triaxialDriver/triaxialDriver_ModifiedCamClay.xml
```

It also uses a set of table files located at:

```
inputFiles/triaxialDriver/tables/
```

A Python script for the semi-analytical solutions presented above as well as for post-processing the GEOS results is provided at:

```
src/docs/sphinx/advancedExamples/validationStudies/viscoplasticity/ModifiedCamClay/
↪TriaxialDriver_vs_SemiAnalytic_ModifiedCamClay.py
```

For this example, we focus on the `Task` and the `Constitutive` tags.

Task

The imposed axial strain loading/unloading periods, the zero lateral strain as well as the initial stress are defined in the `Task` block as

```
  <Tasks>
   <TriaxialDriver
      name="triaxialDriver"
      material="ModifiedCamClay"
      mode="strainControl"
      axialControl="strainFunction"
      radialControl="zeroStrain"
      initialStress="-1e5"
      steps="200"
      output="ModifiedCamClayResults.txt" />
  </Tasks>
```

Constitutive laws

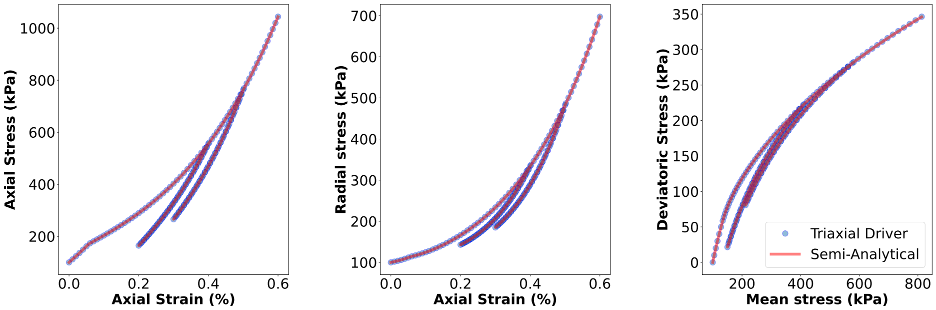
The elasto-plastic parameters are defined as

```
    <ModifiedCamClay
      name="ModifiedCamClay"
      defaultDensity="2700"
      defaultRefPressure="-1e5"
      defaultRefStrainVol="0.0"
      defaultShearModulus="5e7"
      defaultPreConsolidationPressure="-1.5e5"
      defaultCslSlope="1.2"
      defaultRecompressionIndex="0.002"
      defaultVirginCompressionIndex="0.003"
    />
```

All constitutive parameters such as density, viscosity, bulk and shear moduli are specified in the International System of Units.

A comparison between GEOS results and semi-analytical results

The simulation results are saved in a text file, named `ModifiedCamClayResults.txt`. A perfect comparison between the results given by the TriaxialDriver solver in GEOS and the semi-analytical results presented above is shown below:



To go further

Feedback on this example

For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

Visco Modified CamClay model: Triaxial Driver versus Semi-Analytical Solution

Problem description

This example uses the Triaxial Driver to simulate a visco-elasto-plastic oedometric compression test of a Visco Modified CamClay solid. Oedometric condition with zero lateral strain together with loading/unloading axial strain periods are imposed. Semi-analytical results for the mean and shear stress variations Δp and Δq can be established, considering the Perzyna approach, for the imposed oedometric boundary conditions as (Runesson et al. 1999) :

$$\Delta p = K(\Delta\varepsilon_V - \Delta\lambda\frac{\partial G}{\partial p})$$

$$\Delta q = 2\mu(\Delta\varepsilon_V - \Delta\lambda\frac{3}{2}\frac{\partial G}{\partial q})$$

where K and μ are elastic bulk and shear moduli, G is the plastic potential and $\Delta\lambda$ is the visco-plastic multiplier that can be approximated by:

$$\Delta\lambda = \frac{\Delta t}{t_*}\frac{F}{3\mu\frac{\partial F}{\partial q}\frac{\partial G}{\partial q} + K\frac{\partial F}{\partial p}\frac{\partial G}{\partial p} + h}$$

in which Δt is the time increment, t_* is the relaxation time, F is the stress function defining the visco-plastic yield surface and h is the hardening rate defined by:

$$h = -\frac{\partial F}{\partial\lambda}$$

These solutions are implemented in a Python script associated to this example for verifying GEOS results.

Input files

This validation example uses two GEOS xml files that are located at:

```
inputFiles/triaxialDriver/triaxialDriver_base.xml
```

and

```
inputFiles/triaxialDriver/triaxialDriver_ViscoModifiedCamClay.xml
```

It also uses a set of table files located at:

```
inputFiles/triaxialDriver/tables/
```

A Python script for the semi-analytical solutions presented above as well as for post-processing the GEOS results is provided at:

```
src/docs/sphinx/advancedExamples/validationStudies/viscoplasticity/ViscoModifiedCamClay/
↪TriaxialDriver_vs_SemiAnalytic_ViscoModifiedCamClay.py
```

For this example, we focus on the `Task` and the `Constitutive` tags.

Task

The imposed axial strain loading/unloading periods, the lateral zero strain, and the initial stress are defined in the `Task` block as:

```
  <Tasks>
   <TriaxialDriver
      name="triaxialDriver"
      material="ViscoModifiedCamClay"
      mode="strainControl"
      axialControl="strainFunction"
      radialControl="zeroStrain"
      initialStress="-1e5"
      steps="200"
      output="ViscoModifiedCamClayResults.txt" />
  </Tasks>
```

Constitutive laws

The elasto-visco-plastic parameters are defined as:

```
    <ViscoModifiedCamClay
      name="ViscoModifiedCamClay"
      defaultDensity="2700"
      defaultRefPressure="-1e5"
      defaultRefStrainVol="0.0"
      defaultShearModulus="5e7"
      defaultPreConsolidationPressure="-1.5e5"
      defaultCslSlope="1.2"
      defaultRecompressionIndex="0.002"
```

(continues on next page)

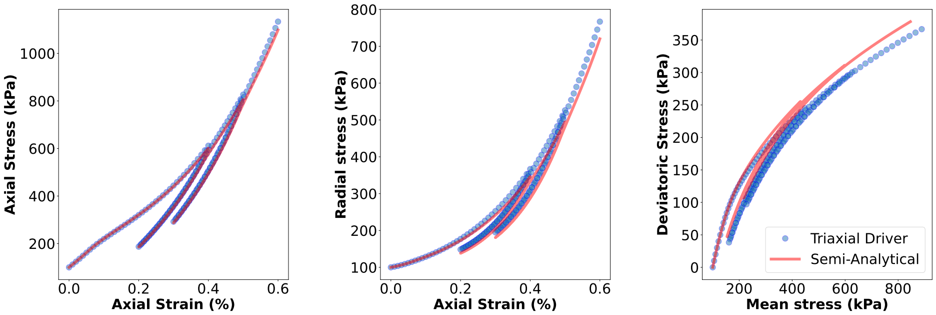
(continued from previous page)

```
      defaultVirginCompressionIndex="0.003"
      relaxationTime="0.1"
    />
```

All constitutive parameters such as density, viscosity, and the bulk and shear moduli are specified in the International System of Units.

A comparison between GEOS results and semi-analytical results

The simulation results are saved in a text file, named `ViscoModifiedCamClayResults.txt`. A comparison between the results given by the TriaxialDriver solver in GEOS and the semi-analytical results presented above is shown below. The discrepancy between these results may due to the difference between the Duvaut-Lions approach and the Perzyna approach for time dependant behavior when applying for the Modified CamClay model as discussed by Runesson et al. (1999).



To go further

Feedback on this example

For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

Verification of a Relaxation Test with Visco Extended Drucker-Prager Model

Context

In this example, we simulate a relaxation test with a Visco Extended Drucker-Prager solid. This problem is solved using a viscoplastic solver (see *Model: Viscoplasticity*) in GEOS to predict the time-dependent deformation of the sample when subject to loading conditions. We verify the numerical results obtained by GEOS against a semi-analytical solution (see *Visco Extended Drucker-Prager Model: Triaxial Driver versus Semi-Analytical Solution*).

Input file

The xml input files for the test case are located at:

```
inputFiles/solidMechanics/viscoExtendedDruckerPrager_relaxation_base.xml
inputFiles/solidMechanics/viscoExtendedDruckerPrager_relaxation_benchmark.xml
```

A Python script for post-processing the simulation results is provided:

```
src/docs/sphinx/advancedExamples/validationStudies/viscoplasticity/RelaxationTest/
↪relaxationTestFigure.py
```

Description of the case

We model the mechanical response of a viscoplastic slab subject to a displacement-controlled uniaxial loading and a constant confining stress, as shown below. The domain is homogeneous, isotropic, and isothermal. Before loading, the domain is initialized with isotropic stresses. Longitudinal compression is induced and governed by the normal displacement applied uniformly over the top surface. This compressive displacement is initially elevated to allow plastic hardening and then kept as a constant to mimic stress relaxation tests. In this example, fluid flow is not considered.

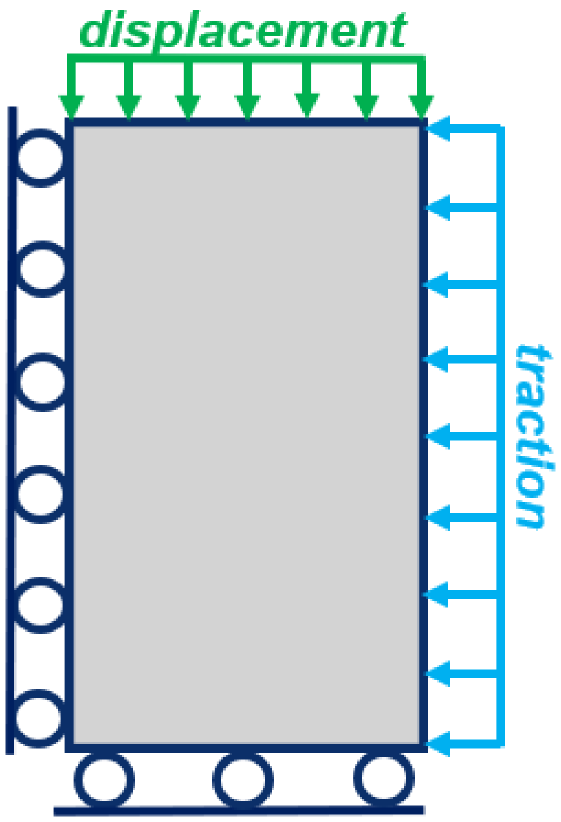


Fig. 1.66: Sketch of the problem

We set up and solve a Visco Extended Drucker-Prager model to obtain the spatial and temporal solutions of stresses and displacements across the domain upon loading. These numerical predictions are compared with the corresponding semi-analytical solutions derived from Runesson et al. (1999) (see *Visco Extended Drucker-Prager Model: Triaxial Driver versus Semi-Analytical Solution*).

For this example, we focus on the `Mesh`, the `Constitutive`, and the `FieldSpecifications` tags.

Mesh

The following figure shows the mesh used for solving this mechanical problem:

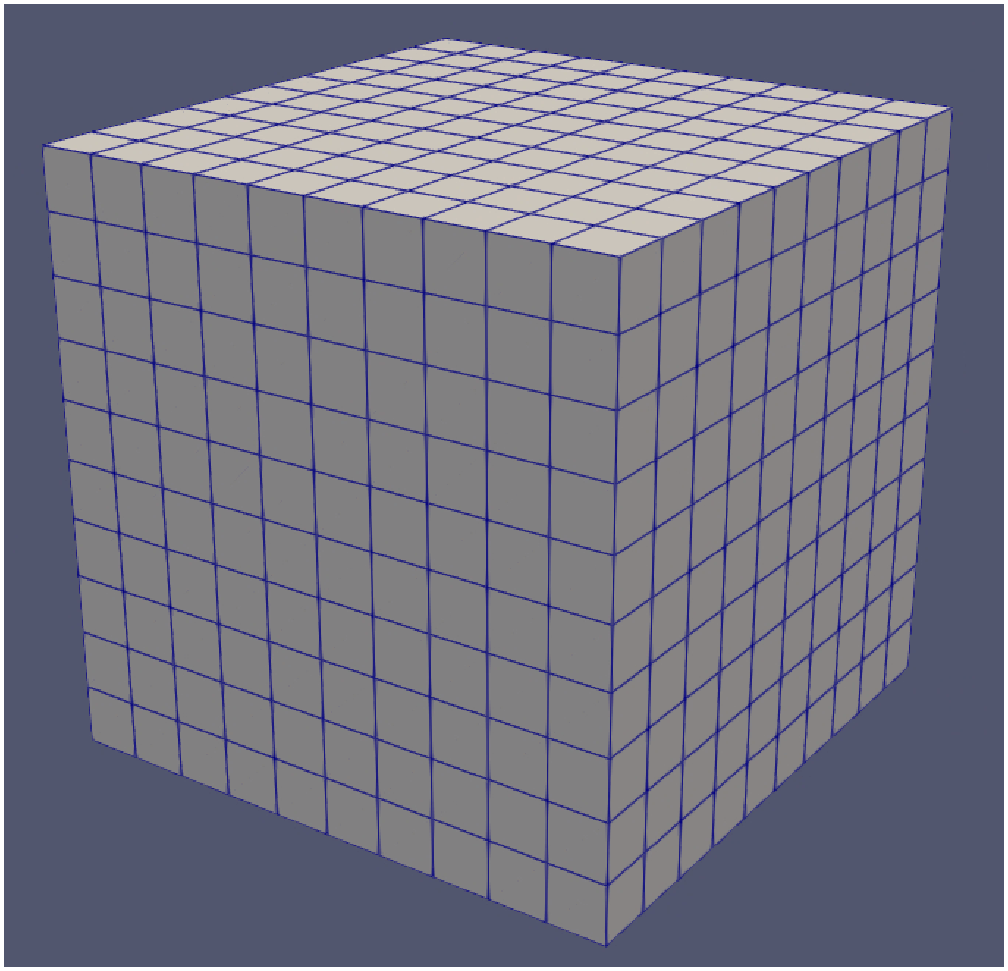


Fig. 1.67: Generated mesh

The mesh was created with the internal mesh generator and parametrized in the `InternalMesh` XML tag. It contains 10x10x10 eight-node brick elements in the x, y, and z directions respectively. Such eight-node hexahedral elements are defined as `C3D8` elementTypes, and their collection forms a mesh with one group of cell blocks named here `cellBlockNames`.

```xml
<Mesh>
  <InternalMesh
    name="mesh"
    elementTypes="{ C3D8 }"
    xCoords="{ 0, 0.9, 1.0 }"
    yCoords="{ 0, 0.9, 1.0 }"
    zCoords="{ 0, 0.9, 1.0 }"
    nx="{ 9, 1 }"
    ny="{ 9, 1 }"
    nz="{ 9, 1 }"
    cellBlockNames="{ cb1, cb2, cb3, cb4, cb5, cb6, cb7, cb8 }"/>
```

(continues on next page)

(continued from previous page)

```
</Mesh>
```

Solid mechanics solver

For the relaxation tests, pore pressure variations are neglected and subtracted from the analysis. Therefore, we define a solid mechanics solver, called here `mechanicsSolver`. This solid mechanics solver (see *Solid Mechanics Solver*) is based on the Lagrangian finite element formulation. The problem is run as `QuasiStatic` without considering inertial effects. The computational domain is discretized by `FE1`, defined in the `NumericalMethods` section. We use the `targetRegions` attribute to define the regions where the solid mechanics solver is applied. Here, we only simulate mechanical deformation in one region named as `Domain`, whose mechanical properties are specified in the `Constitutive` section.

```
<Solvers
  gravityVector="{ 0.0, 0.0, 0.0 }">
  <SolidMechanics_LagrangianFEM
    name="mechanicsSolver"
    timeIntegrationOption="QuasiStatic"
    logLevel="1"
    discretization="FE1"
    targetRegions="{ Domain }">
    <NonlinearSolverParameters
      newtonTol="1.0e-5"
      newtonMaxIter="15"/>
    <LinearSolverParameters
      solverType="direct"/>
  </SolidMechanics_LagrangianFEM>
</Solvers>
```

Constitutive laws

A homogeneous domain with one solid material is assumed, and its mechanical properties are specified in the `Constitutive` section:

```
<Constitutive>
  <ViscoExtendedDruckerPrager
    name="rock"
    defaultDensity="2700"
    defaultBulkModulus="10.0e9"
    defaultShearModulus="6.0e9"
    defaultCohesion="0.1e6"
    defaultInitialFrictionAngle="15.0"
    defaultResidualFrictionAngle="20.0"
    defaultDilationRatio="0.5"
    defaultHardening="0.0005"
    relaxationTime="5000.0"
  />
</Constitutive>
```

Recall that in the `SolidMechanics_LagrangianFEM` section, `rock` is designated as the material in the computational domain. Here, Visco Extended Drucker Prager model `ViscoExtendedDruckerPrager` is used to simulate the viscoplastic behavior of `rock`. As for the material parameters, `defaultInitialFrictionAngle`, `defaultResidualFrictionAngle` and `defaultCohesion` denote the initial friction angle, the residual friction angle, and cohesion, respectively, as defined by the Mohr-Coulomb failure envelope. As the residual friction angle `defaultResidualFrictionAngle` is larger than the initial one `defaultInitialFrictionAngle`, a strain hardening model is adopted, with a hardening rate given as `defaultHardening="0.0005"`. Finally, `relaxationTime` is a key parameter for characterizing the viscoplastic behavior of the solid called `rock`.

Constitutive parameters such as density, bulk modulus, and shear modulus are specified in the International System of Units.

Time history function

In the `Tasks` section, `PackCollection` tasks are defined to collect time history information from several property fields (time-series). We can collect either the entire collection of field properties or specified named sets. In this example, `stressCollection` is specified to output the time history of stresses `fieldName="rock_stress` for the selected subdomain `objectPath="ElementRegions/Domain/cb8"`. And `displacementCollection` is defined to output the time history of displacement `fieldName="totalDisplacement"` for the subset `setNames="{ topPoint }"`.

```
<Tasks>
  <PackCollection
    name="stressCollection"
    objectPath="ElementRegions/Domain/cb8"
    fieldName="rock_stress"/>

  <PackCollection
    name="displacementCollection"
    objectPath="nodeManager"
    fieldName="totalDisplacement"
    setNames="{ topPoint }"/>
</Tasks>
```

These two tasks are triggered using the `Event` management where `PeriodicEvent` are defined for these recurring tasks. GEOS writes two files named after the string defined in the `filename` keyword and formatted as HDF5 files (displacement_history.hdf5 and stress_history.hdf5). The TimeHistory file contains the collected time history information from each specified time history collector. This information includes datasets for the simulation time, element center or nodal position, and the time history information. We use a Python script to read and plot any specified subset of the time history data for verification and visualization.

Initial and boundary conditions

The next step is to specify fields, including:

- The initial value (the stresses have to be initialized),
- The boundary conditions (the displacement control, lateral confining stress, and constraints of the outer boundaries have to be set).

In this example, we specify isotropic stresses (σ_{xx} = -10.0 MPa, σ_{yy} = -10.0 MPa, and σ_{zz} = -10.0 MPa,).

A compressive traction (P_w = -10.0 MPa) is applied on the domain laterals `xpos` (all faces on the x-side of the domain, positive side) and `ypos` (all faces on the y-side of the domain, positive side).

The normal displacement (`axialload`) is instantaneously applied at the top surface `zpos` (all surfaces on the top) at time t = 0 s, and will gradually increase to a higher absolute value (-0.001) in 0.5 days to let the rock slab contract.

Then, it remains constant for 1.5 days to allow stress relaxation.

The remaining outer boundaries of the domain are subjected to roller constraints.

These boundary conditions are set up through the `FieldSpecifications` section.

The lateral traction and compressive displacement have negative values due to the negative sign convention for compressive stresses in GEOS.

```
<FieldSpecifications>
  <FieldSpecification
    name="xconstraint"
    objectPath="nodeManager"
    fieldName="totalDisplacement"
    component="0"
    scale="0.0"
    setNames="{ xneg }"/>

  <FieldSpecification
    name="yconstraint"
    objectPath="nodeManager"
    fieldName="totalDisplacement"
    component="1"
    scale="0.0"
    setNames="{ yneg }"/>

  <FieldSpecification
    name="zconstraint"
    objectPath="nodeManager"
    fieldName="totalDisplacement"
    component="2"
    scale="0.0"
    setNames="{ zneg }"/>

  <FieldSpecification
    name="stressXX"
    initialCondition="1"
    setNames="{all}"
    objectPath="ElementRegions/Domain"
    fieldName="rock_stress"
    component="0"
    scale="-10.0e6"
  />

  <FieldSpecification
    name="stressYY"
    initialCondition="1"
    setNames="{all}"
    objectPath="ElementRegions/Domain"
    fieldName="rock_stress"
    component="1"
    scale="-10.0e6"
  />

  <FieldSpecification
```

(continues on next page)

(continued from previous page)

```
      name="stressZZ"
      initialCondition="1"
      setNames="{all}"
      objectPath="ElementRegions/Domain"
      fieldName="rock_stress"
      component="2"
      scale="-10.0e6"
    />

    <Traction
      name="xconfinement"
      setNames="{ xpos }"
      objectPath="faceManager"
      scale="-10.0e6"
      tractionType="normal"
    />

    <Traction
      name="yconfinement"
      setNames="{ ypos }"
      objectPath="faceManager"
      scale="-10.0e6"
      tractionType="normal"
    />

    <FieldSpecification
      name="axialload"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="2"
      scale="-0.001"
      functionName="timeFunction"
      setNames="{ zpos }"/>
  </FieldSpecifications>
```

The parameters used in the simulation are summarized in the following table.

Symbol	Parameter	Unit	Value
K	Bulk modulus	[GPa]	10.0
G	Shear Modulus	[GPa]	6.0
c	Cohesion	[MPa]	0.1
ϕ_i	Initial Friction Angle	[degree]	15.0
ϕ_r	Residual Friction Angle	[degree]	20.0
m	Hardening Rate	[-]	0.0005
τ	Relaxation Time	[s]	5000.0
σ_h	Horizontal Stress	[MPa]	-10.0
σ_v	Vertical Stress	[MPa]	-10.0
P_w	Traction at lateral	[MPa]	-10.0

Inspecting results

In the example, we request hdf5 output files for time-series (time history). We use Python scripts to visualize the outcome. The following figure shows the final distribution of vertical displacement upon loading.

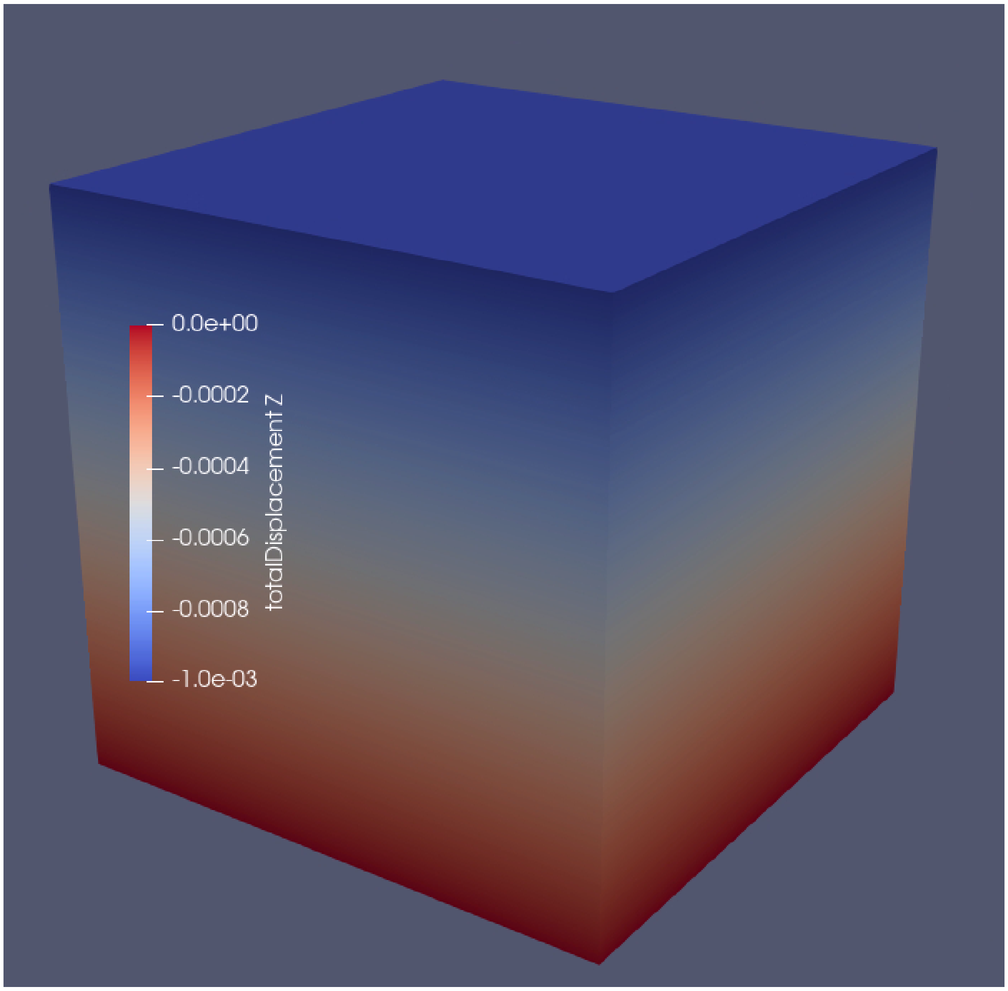
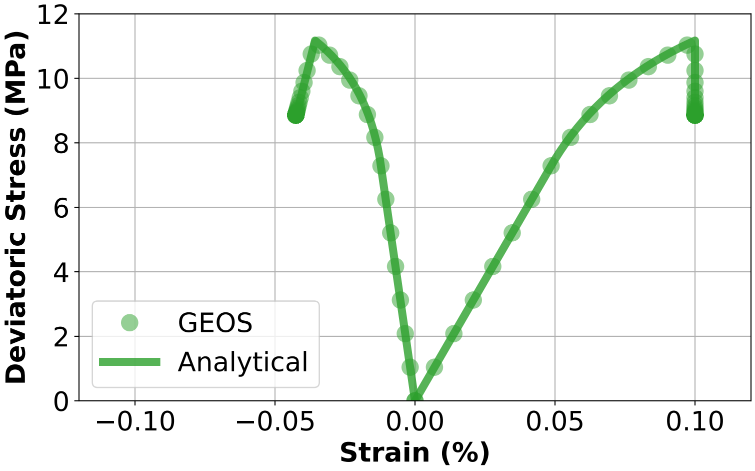
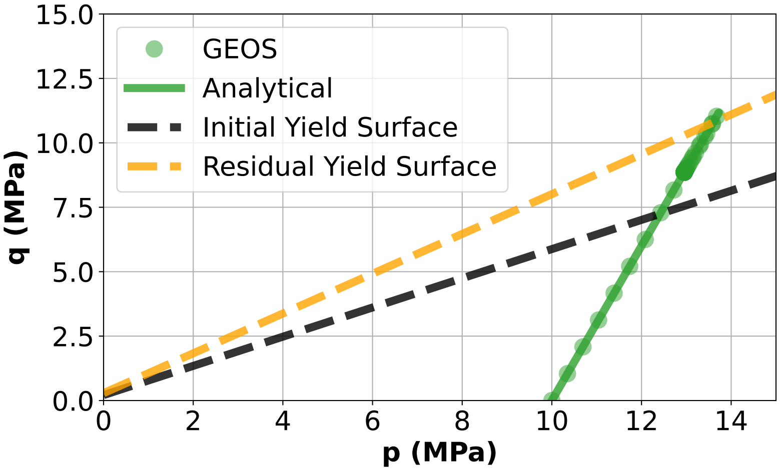
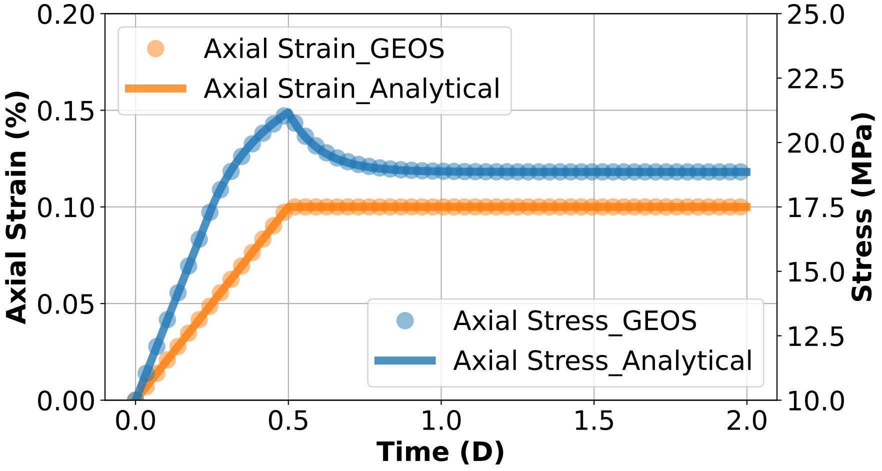


Fig. 1.68: Simulation result of vertical displacement

The figure below shows the comparisons between the numerical predictions (marks) and the corresponding analytical solutions (lines) with respect to stress–strain relationship, stress path on the top surface, the evolution of axial stress. Predictions computed by GEOS match the semi-analytical results. The bottom figure highlights the change in axial stress with time. Note that, if normal displacement remains constant, the axial stress decreases gradually to a residue value. This behavior is typically reported in the relaxation tests in laboratory.


Deviatoric Stress (MPa)
12
10
8
6
4
2
0
−0.10
−0.05
0.00
0.05
0.10
Strain (%)
GEOS
Analytical


q (MPa)
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0
2
4
6
8
10
12
14
p (MPa)
GEOS
Analytical
Initial Yield Surface
Residual Yield Surface


Axial Strain (%)
0.20
0.15
0.10
0.05
0.00
0.0
0.5
1.0
1.5
2.0
Time (D)
Stress (MPa)
25.0
22.5
20.0
17.5
15.0
12.5
10.0
Axial Strain_GEOS
Axial Strain_Analytical
Axial Stress_GEOS
Axial Stress_Analytical

To go further

Feedback on this example

For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

Poromechanics

Mandel's Problem

Context

In this example, we use the coupled solvers in GEOS to solve Mandel's 2D consolidation problem, a classic benchmark in poroelasticity. The analytical solution (Cheng and Detournay, 1988) is employed to verify the accuracy of the modeling predictions on induced pore pressure and the corresponding settlement. In this example, the `TimeHistory` function and a Python script are used to output and post-process multi-dimensional data (pore pressure and displacement field).

Input file

This example uses no external input files and everything required is contained within two GEOS input files located at:

```
inputFiles/poromechanics/PoroElastic_Mandel_base.xml
```

```
inputFiles/poromechanics/PoroElastic_Mandel_benchmark_fim.xml
```

Description of the case

We simulate the consolidation of a poroelastic slab between two rigid and impermeable plates subjected to a constant normal force. The slab is assumed to be fully saturated, homogeneous, isotropic, and infinitely long in the y-direction. We apply a uniform compressive load in the vertical direction. This force leads to a change of pore pressure and mechanical deformations of the sample, evolving with time due to fluid diffusion and coupling effects. The numerical model represents a plane strain deformation and lateral drainage without confinement, showing only a quarter of the computational domain in the x-z plane (the rest follows by symmetry).

In this example, we set up and solve a poroelastic model to obtain the temporal and spatial solutions of pore pressure ($p(x, z, t)$) and displacement field ($u_z(x, z, t)$) for Mandel's problem. These modeling predictions are validated against corresponding analytical solution (Cheng and Detournay, 1988).

$$p(x,z,t) = 2p_0 \sum_{n=1}^{\infty} \frac{\mathrm{sin}\alpha_n}{\alpha_n - \mathrm{sin}\alpha_n \mathrm{cos}\alpha_n} \left(\mathrm{cos}\frac{\alpha_n x}{a} - \mathrm{cos}\alpha_n \right) \exp\left(-\frac{{\alpha_n}^2 ct}{a^2} \right)$$

$$u_z(x,z,t) = \left[-\frac{F\left(1-\nu\right)}{2Ga} + \frac{F\left(1-\nu_u\right)}{Ga} \sum_{n=1}^{\infty} \frac{\mathrm{sin}\alpha_n \mathrm{cos}\alpha_n}{\alpha_n - \mathrm{sin}\alpha_n \mathrm{cos}\alpha_n} \exp\left(-\frac{{\alpha_n}^2 ct}{a^2} \right) \right] z$$

with α_n denoting the positive roots of the following equation:

$$\mathrm{tan}\alpha_n = \frac{1-\nu}{\nu_u - \nu}\alpha_n$$

Upon sudden application of the vertical load, the instantaneous overpressure ($p_0(x, z)$) and settlement ($u_{z,0}(x, z)$ and $u_{x,0}(x, z)$) across the sample are derived from the Skempton effect:

$$p_0(x,z) = \frac{1}{3a} B \left(1+\nu_u\right) F$$

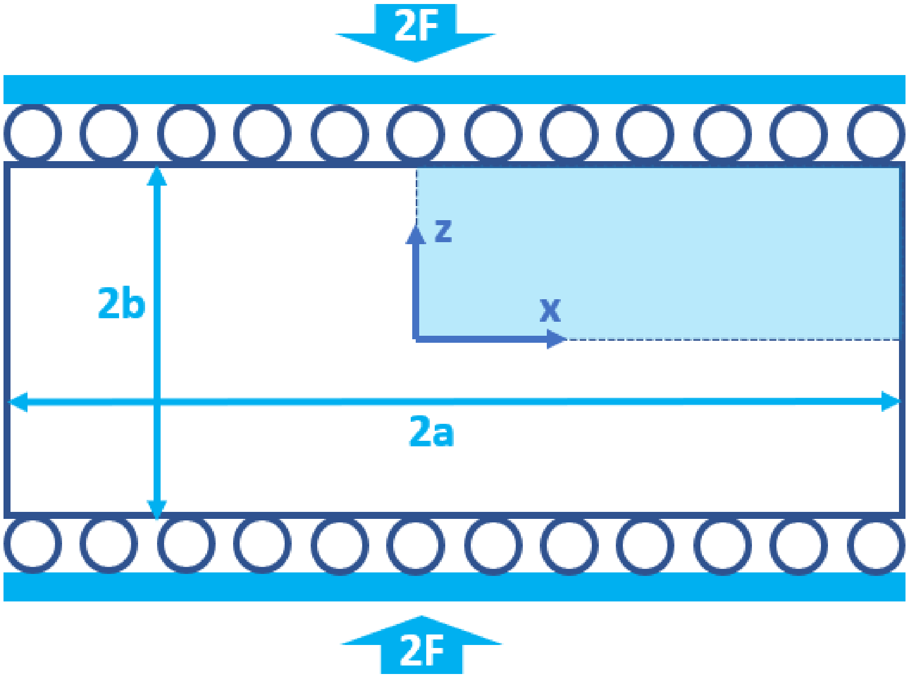


Fig. 1.69: Sketch of the problem

$$u_{z,0}(x,z) = -\frac{F\,(1-\nu_u)}{2G}\frac{z}{a}$$
$$u_{x,0}(x,z) = \frac{F\nu_u}{2G}\frac{x}{a}$$

where ν and ν_u are the drained and undrained Poisson's ratio respectively, c is the consolidation coefficient, B is Skempton's coefficient, G is the shear modulus, and F is the applied force.

For this example, we focus on the `Mesh`, the `Constitutive`, and the `FieldSpecifications` tags.

Mesh

The following figure shows the mesh used in this problem.

This mesh was created using the internal mesh generator as parametrized in the `InternalMesh` XML tag. The structured mesh contains 20 x 1 x 20 eight-node brick elements in the x, y, and z directions respectively. Such eight-node hexahedral elements are defined as `C3D8` elementTypes, and their collection forms a mesh with one group of cell blocks named here `cb1`.

```
<Mesh>
  <InternalMesh
    name="mesh1"
    elementTypes="{ C3D8 }"
    xCoords="{ 0.0, 1.0 }"
    yCoords="{ 0.0, 0.1 }"
    zCoords="{ 0.0, 1.0 }"
    nx="{ 20 }"
    ny="{ 1 }"
```

(continues on next page)

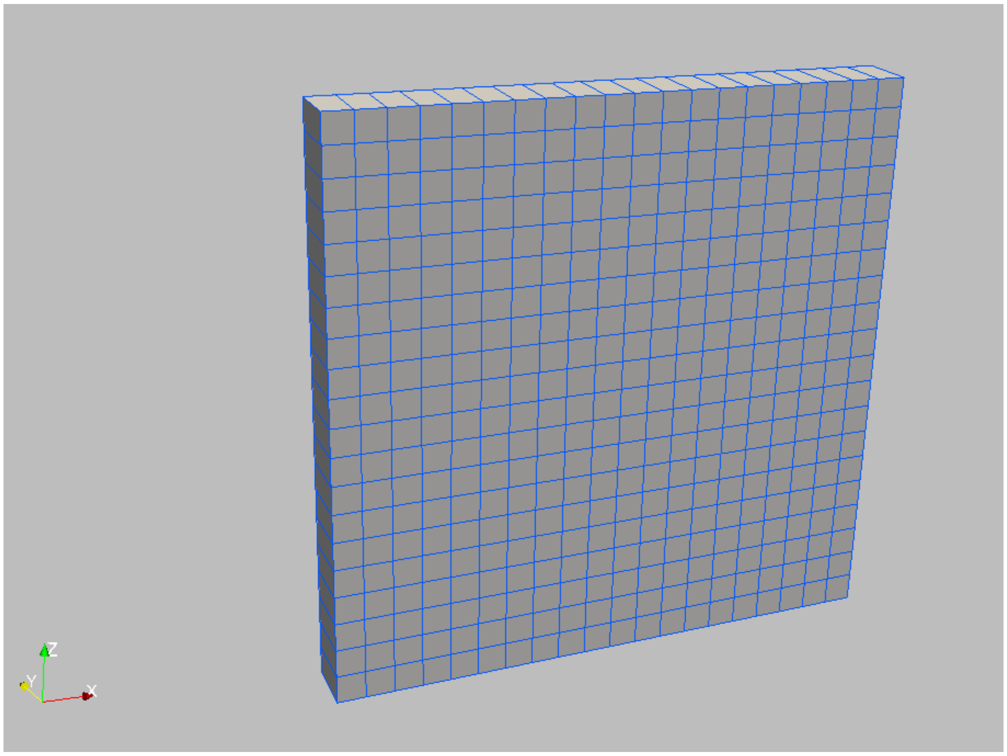


Fig. 1.70: Generated mesh

(continued from previous page)

```
    nz="{ 20 }"
    cellBlockNames="{ cb1 }"/>
</Mesh>
```

Solid mechanics solver

GEOS is a multi-physics platform. Different combinations of physics solvers available in the code can be applied in different regions of the domain and be functional at different stages of the simulation. The `Solvers` tag in the XML file is used to list and parameterize these solvers.

To specify a coupling between two different solvers, we define and characterize each single-physics solver separately. Then, we customize a *coupling solver* between these single-physics solvers as an additional solver. This approach allows for generality and flexibility in constructing multi-physics solvers. The order of specifying these solvers is not restricted in GEOS. Note that end-users should give each single-physics solver a meaningful and distinct name, as GEOS will recognize these single-physics solvers based on their customized names and create user-expected coupling.

As demonstrated in this example, to setup a poromechanical coupling, we need to define three different solvers in the XML file:

- the mechanics solver, a solver of type `SolidMechanicsLagrangianSSLE` called here `lagsolve` (more information here: *Solid Mechanics Solver*),
- the single-phase flow solver, a solver of type `SinglePhaseFVM` called here `SinglePhaseFlow` (more information on these solvers at *Singlephase Flow Solver*),
- the coupling solver (`SinglePhasePoromechanics`) that will bind the two single-physics solvers above, which is named as `poroSolve` (more information at *Poromechanics Solver*).

The two single-physics solvers are parameterized as explained in their corresponding documentation pages. We focus on the coupling solver in this example. The solver poroSolve uses a set of attributes that specifically describe the coupling process within a poromechanical framework. For instance, we must point this solver to the designated fluid solver (here: SinglePhaseFlow) and solid solver (here: lagsolve). These solvers interact through the porousMaterialNames="{ shale }" with all the constitutive models. We specify the discretization method (FE1, defined in the NumericalMethods section), and the target regions (here, we only have one, Domain). More parameters are required to characterize a coupling procedure (more information at *Poromechanics Solver*). In this way, the two single-physics solvers will be simultaneously called and executed for solving Mandel's problem here.

Constitutive laws

For this problem, we simulate the poroelastic deformation of a slab under uniaxial compression. A homogeneous and isotropic domain with one solid material is assumed, and its mechanical properties and associated fluid rheology are specified in the Constitutive section. PorousElasticIsotropic model is used to describe the mechanical behavior of shaleSolid when subjected to loading. The single-phase fluid model CompressibleSinglePhaseFluid is selected to simulate the response of water upon consolidation.

```xml
<Constitutive>
  <PorousElasticIsotropic
    name="shale"
    solidModelName="shaleSolid"
    porosityModelName="shalePorosity"
    permeabilityModelName="shalePerm"/>

  <ElasticIsotropic
    name="shaleSolid"
    defaultDensity="0"
    defaultBulkModulus="6.6667e7"
    defaultShearModulus="4.0e7"/>

  <BiotPorosity
    name="shalePorosity"
    grainBulkModulus="1.0e27"
    defaultReferencePorosity="0.375"/>

  <ConstantPermeability
    name="shalePerm"
    permeabilityComponents="{ 1.0e-12, 0.0, 1.0e-12 }"/>

  <CompressibleSinglePhaseFluid
    name="water"
    defaultDensity="1000"
    defaultViscosity="0.001"
    referencePressure="0.000"
    referenceDensity="1"
    compressibility="4.4e-10"
    referenceViscosity="0.001"
    viscosibility="0.0"/>
</Constitutive>
```

All constitutive parameters such as density, viscosity, bulk modulus, and shear modulus are specified in the International System of Units.

Time history function

In the Tasks section, PackCollection tasks are defined to collect time history information from fields. Either the entire field or specified named sets of indices in the field can be collected. In this example, pressureCollection and displacementCollection tasks are specified to output the time history of pore pressure fieldName="pressure" and displacement field fieldName="totalDisplacement" across the computational domain.

```
<Tasks>
  <PackCollection
    name="pressureCollection"
    objectPath="ElementRegions/Domain/cb1"
    fieldName="pressure"/>

  <PackCollection
    name="displacementCollection"
    objectPath="nodeManager"
    fieldName="totalDisplacement"/>
</Tasks>
```

These two tasks are triggered using the Event manager with a PeriodicEvent defined for these recurring tasks. GEOS writes two files named after the string defined in the filename keyword and formatted as HDF5 files (displacement_history.hdf5 and pressure_history.hdf5). The TimeHistory file contains the collected time history information from each specified time history collector. This information includes datasets for the simulation time, element center, and the time history information. A Python script is prepared to read and plot any specified subset of the time history data for verification and visualization.

Initial and boundary conditions

Next, we specify two fields:

- The initial value (the displacements, effective stress, and pore pressure have to be initialized, corresponding to the undrained response),
- The boundary conditions (the vertical displacement applied at the loaded boundary and the constraints of the outer boundaries have to be set).

In this example, the analytical z-displacement is applied at the top surface (zpos) of computational domain to enforce the rigid plate condition. The lateral surface (xpos) is traction-free and allows drainage. The remaining parts of the outer boundaries are subjected to roller constraints. These boundary conditions are set up through the FieldSpecifications section.

```
<FieldSpecifications>
  <FieldSpecification
    name="initialPressure"
    initialCondition="1"
    setNames="{ all }"
    objectPath="ElementRegions/Domain/cb1"
    fieldName="pressure"
    scale="4934.86"/>

  <FieldSpecification
    name="initial_sigma_x"
    initialCondition="1"
    setNames="{ all }"
```

(continues on next page)

(continued from previous page)

```
      objectPath="ElementRegions"
      fieldName="shaleSolid_stress"
      component="0"
      scale="4934.86"/>
    <FieldSpecification
      name="initial_sigma_y"
      initialCondition="1"
      setNames="{ all }"
      objectPath="ElementRegions"
      fieldName="shaleSolid_stress"
      component="1"
      scale="4934.86"/>
    <FieldSpecification
      name="initial_sigma_z"
      initialCondition="1"
      setNames="{ all }"
      objectPath="ElementRegions"
      fieldName="shaleSolid_stress"
      component="2"
      scale="4934.86"/>

    <FieldSpecification
      name="xInitialDisplacement"
      initialCondition="1"
      setNames="{ all }"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="0"
      scale="1.0"
      functionName="initialUxFunc"/>

    <FieldSpecification
      name="yInitialDisplacement"
      initialCondition="1"
      setNames="{ all }"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="1"
      scale="0.0"/>

    <FieldSpecification
      name="zInitialDisplacement"
      initialCondition="1"
      setNames="{ all }"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="2"
      scale="1.0"
      functionName="initialUzFunc"/>

    <FieldSpecification
      name="xnegconstraint"
```

(continues on next page)

(continued from previous page)

```
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="0"
      scale="0.0"
      setNames="{ xneg }"/>

    <FieldSpecification
      name="yconstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="1"
      scale="0.0"
      setNames="{ yneg, ypos }"/>

    <FieldSpecification
      name="zconstraint"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="2"
      scale="0.0"
      setNames="{ zneg }"/>

    <FieldSpecification
      name="NormalDisplacement"
      objectPath="nodeManager"
      fieldName="totalDisplacement"
      component="2"
      scale="-1.0e-5"
      setNames="{ zpos }"
      functionName="loadFunction"/>

    <FieldSpecification
      name="boundaryPressure"
      objectPath="faceManager"
      fieldName="pressure"
      scale="0.0"
      setNames="{ xpos }"/>
  </FieldSpecifications>
```

The parameters used in the simulation are summarized in the following table. Note that traction has a negative value, due to the negative sign convention for compressive stresses in GEOS.

Symbol	Parameter	Unit	Value
K	Bulk Modulus	[MPa]	66.667
G	Shear Modulus	[MPa]	40.0
F	Force per Unit Length	[N/m]	-10^{4}
ϕ	Porosity	[-]	0.375
K_s	Grain Bulk Modulus	[Pa]	10^{27}
ρ_f	Fluid density	[kg/m^3]	10^{3}
c_f	Fluid compressibility	[Pa^{-1}]	$4.4\text{x}10^{-10}$
κ	Permeability	[m^2]	10^{-12}
μ	Fluid viscosity	[Pa s]	10^{-3}
$2a$	Slab Length	[m]	2.0
$2b$	Slab Height	[m]	2.0

Inspecting results

We request VTK-format output files and use Paraview to visualize the results. The following figure shows the distribution of pore pressure ($p(x, z, t)$) at $t = 10s$ within the computational domain.

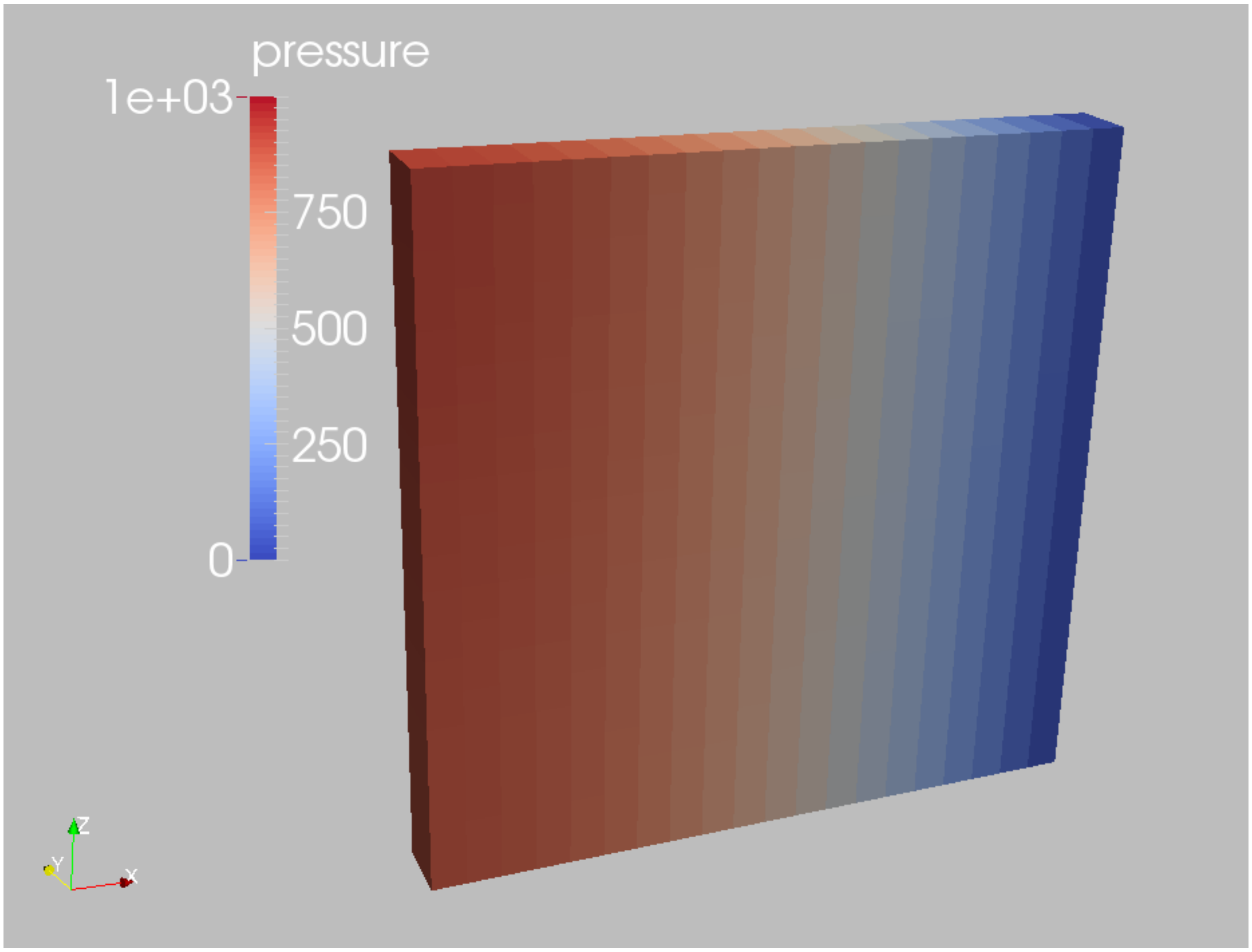


Fig. 1.71: Simulation result of pore pressure at $t = 10s$

The next figure shows the distribution of vertical displacement ($u_z(x, z, t)$) at $t = 10s$.

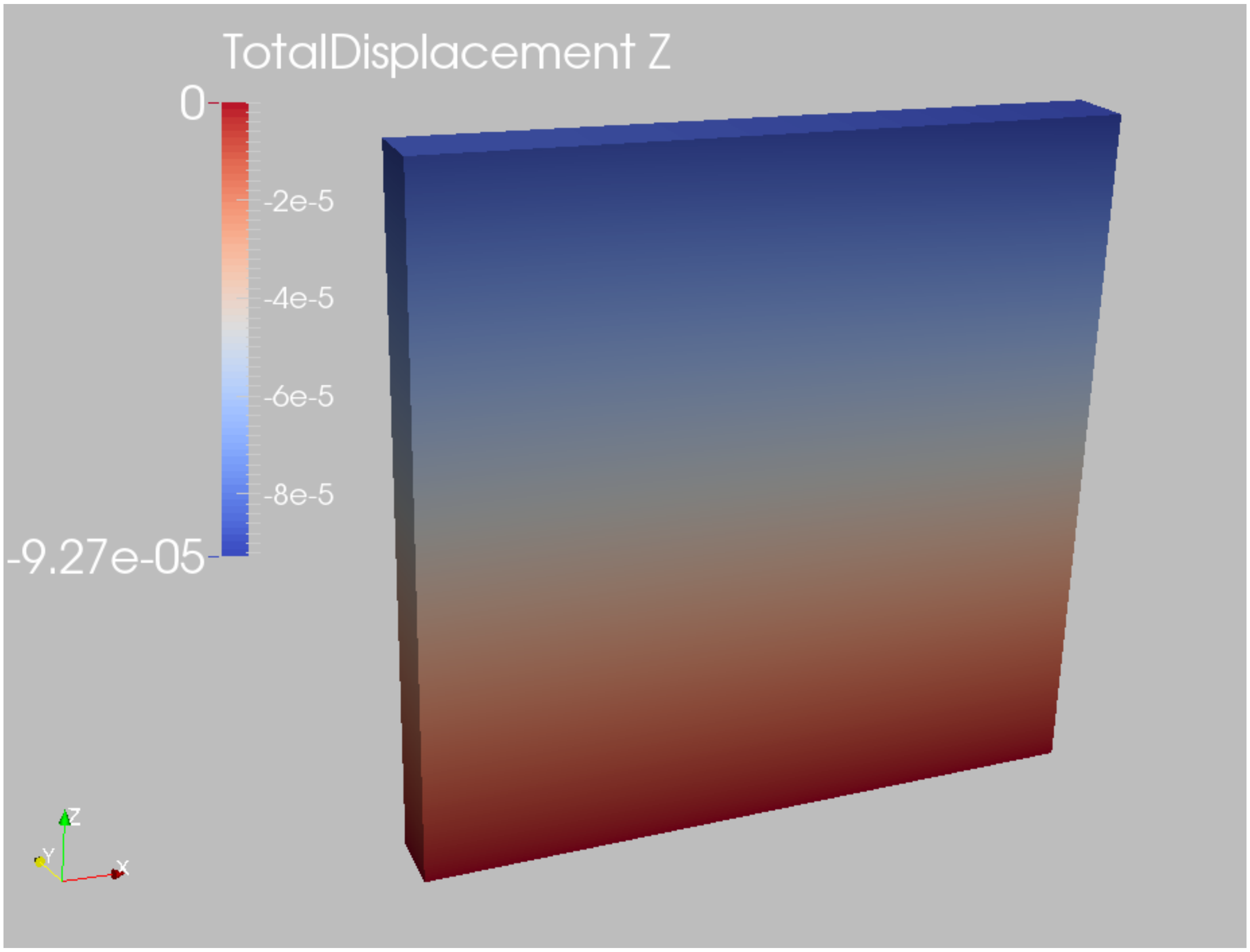


Fig. 1.72: Simulation result of vertical displacement at $t = 10s$

The figure below compares the results from GEOS (marks) and the corresponding analytical solution (lines) for the pore pressure along the x-direction and vertical displacement along the z-direction. GEOS reliably captures the short-term Mandel-Cryer effect and shows excellent agreement with the analytical solution at various times.

To go further

Feedback on this example

For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

Thermoporomechanics

Thermoporoelastic Consolidation

Context

Thermoporoelastic consolidation is a typical fully coupled problem which involves solid deformation, fluid flow and heat transfer in saturated porous media. In this example, we use the GEOS coupled solvers to solve a one-dimensional thermoporoelastic consolidation problem with a non-isothermal boundary condition, and we verify the accuracy of the results using the analytical solution provided in (Bai, 2005)

InputFile

This example uses no external input files and everything required is contained within two GEOS input files located at:

```
inputFiles/thermoPoromechanics/ThermoPoroElastic_consolidation_base.xml
```

```
inputFiles/thermoPoromechanics/ThermoPoroElastic_consolidation_benchmark_fim.xml
```

Description of the case

We simulate the consolidation of 1D thermoporoelastic column subjected to a surface traction stress of 1 Pa applied on the top surface, with a surface temperature of 50 degrees Celsius and a pore pressure of 0 Pa. The initial temperature of the saturated soil is 0 degrees Celsius. The soil column is insulated and sealed everywhere, except at the top surface. The problem setup is illustrated below.

The coupled dynamics experienced by the system are described in (Gao and Ghassemi, 2019) and summarized below. The model first experiences continuous settlement (contraction). Initially, the settlement caused by the drainage of the fluid (effective stress increase) and the compression of the solid matrix is larger than the expansion due to the increase of temperature in the region close to the surface on which a higher temperature is applied. As the temperature diffuses further into the domain, it gradually rebounds (expansion) and reaches a final status.

For this example, we focus on the `Solvers`, the `Constitutive`, and the `FieldSpecifications` tags of the GEOS input file.

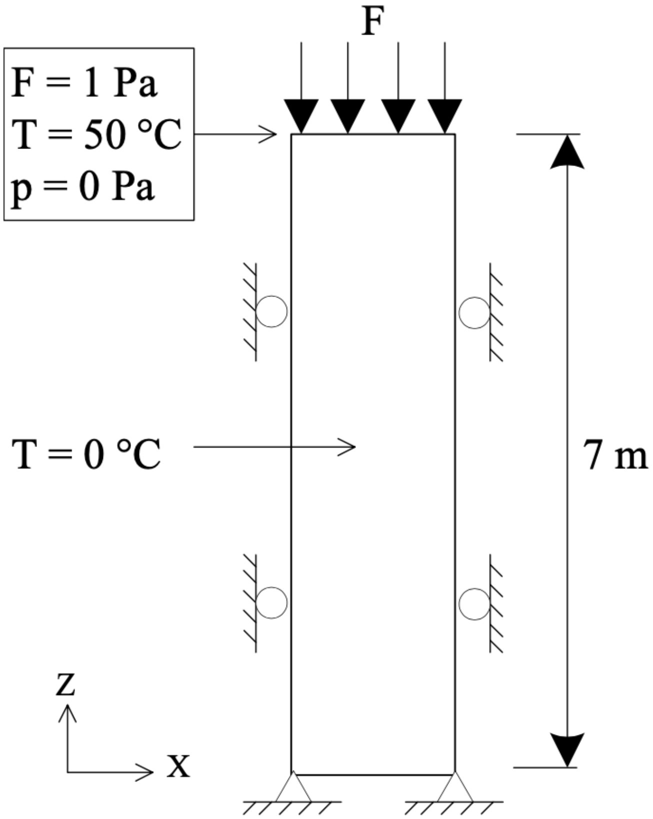


Fig. 1.73: Sketch of the problem (taken from (Gao and Ghassemi, 2019)).

Solvers

As demonstrated in this example, to setup a thermoporomechanical coupling, we need to define three different solvers in the **Solvers** part of the XML file:

- the mechanics solver, a solver of type `SolidMechanicsLagrangianSSLE` called here `solidMechSolver` (more information here: *Solid Mechanics Solver*),

```
<SolidMechanicsLagrangianSSLE
  name="solidMechSolver"
  timeIntegrationOption="QuasiStatic"
  logLevel="1"
  discretization="FE1"
  targetRegions="{ Domain }"/>
```

- the single-phase flow solver, a solver of type `SinglePhaseFVM` called here `flowSolver` (more information on these solvers at *Singlephase Flow Solver*),

```
<SinglePhaseFVM
  name="flowSolver"
  logLevel="1"
  discretization="tpfaFlow"
  temperature="273.0"
  isThermal="1"
  targetRegions="{ Domain }">
  <NonlinearSolverParameters
    newtonMaxIter="100"
    newtonMinIter="0"
    newtonTol="1.0e-6"/>
  <LinearSolverParameters
    directParallel="0"/>
</SinglePhaseFVM>
```

- the coupling solver (`SinglePhasePoromechanics`) that will bind the two single-physics solvers above, which is named as `thermoPoroSolver` (more information at *Poromechanics Solver*).

```
<SinglePhasePoromechanics
  name="thermoPoroSolver"
  solidSolverName="solidMechSolver"
  flowSolverName="flowSolver"
  isThermal="1"
  logLevel="1"
  targetRegions="{ Domain }">
  <NonlinearSolverParameters
    couplingType="FullyImplicit"
    newtonMaxIter="200"/>
  <LinearSolverParameters
    directParallel="0"/>
</SinglePhasePoromechanics>
```

To request the simulation of the temperature evolution, we set the `isThermal` flag of the coupling solver to 1. With this choice, the degrees of freedom are the cell-centered pressure, the cell-centered temperature, and the mechanical displacements at the mesh nodes. The governing equations consist of a mass conservation equation, an energy balance equation, and a linear momentum balance equation. In the latter, the total stress includes both a pore pressure and a temperature contribution. Note that in the coupling solver, we set the `couplingType` to `FullyImplicit` to require a fully coupled, fully implicit solution strategy.

Constitutive laws

A homogeneous and isotropic domain with one solid material is assumed, and its mechanical properties and associated fluid rheology are specified in the **Constitutive** section. We use the constitutive parameters specified in (Bai, 2005) listed in the following table.

Symbol	Parameter	Unit	Value
E	Young's modulus	[Pa]	6000.0
ν	Poisson's ratio	[-]	0.4
α	Thermal expansion coef.	[T^(-1)]	9.0x10^{-7}
ϕ	Porosity	[-]	0.20
b	Biot's coefficient	[-]	1.0
ρC	Heat capacity	[J/(m^3.K)]	167.2x10^{3}
μ	Fluid viscosity	[Pa.s]	1^{-3}
k^T	Thermal conductivity	[J/(m.s.K)]	836
k	Permeability	[m^2]	4.0x10^{-9}

The bulk modulus, the Young's modulus, and the thermal expansion coefficient are specified in the `ElasticIsotropic` solid model. Note that for now the solid density is constant and does not depend on temperature. Given that the gravity vector has been set to 0 in the XML file, the value of the solid density is not used in this simulation.

```
<ElasticIsotropic
  name="rockSolid"
  defaultDensity="2400"
  defaultBulkModulus="1e4"
  defaultShearModulus="2.143e3"
  defaultThermalExpansionCoefficient="3e-7"/>
```

The porosity and Biot's coefficient (computed from the `grainBulkModulus`) appear in the `BiotPorosity` model. In this model, the porosity is updated as a function of the strain increment, the change in pore pressure, and the change in temperature.

```
<BiotPorosity
  name="rockPorosity"
  grainBulkModulus="1.0e27"
  defaultReferencePorosity="0.2"
  defaultThermalExpansionCoefficient="3e-7"/>
```

The heat capacity is provided in the `SolidInternalEnergy` model. In the computation of the internal energy, the `referenceTemperature` is set to the initial temperature.

```
<SolidInternalEnergy
  name="rockInternalEnergy"
  volumetricHeatCapacity="1.672e5"
  referenceTemperature="0.0"
  referenceInternalEnergy="0.0"/>
```

The fluid density and viscosity are given in the `ThermalCompressibleSinglePhaseFluid`. Here, they are assumed to be constant and do not depend on pressure and temperature.

```
    <ThermalCompressibleSinglePhaseFluid
      name="water"
      defaultDensity="1000"
      defaultViscosity="1e-3"
      referencePressure="0.0"
      referenceTemperature="0.0"
      compressibility="0.0"
      thermalExpansionCoeff="0.0"
      viscosibility="0.0"
      volumetricHeatCapacity="1.672e2"
      referenceInternalEnergy="0.001"/>
```

Finally, the permeability and thermal conductivity are specified in the `ConstantPermeability` and `SinglePhaseConstantThermalConductivity`, respectively.

```
    <ConstantPermeability
      name="rockPerm"
      permeabilityComponents="{ 4.0e-9, 4.0e-9, 4.0e-9 }"/>

    <SinglePhaseConstantThermalConductivity
      name="thermalCond"
      thermalConductivityComponents="{ 836, 836, 836 }"/>
```

Initial and boundary conditions

To complete the specification of the problem, we specify two types of fields:

- The initial values (the displacements, effective stress, and pore pressure have to be initialized),
- The boundary conditions at the top surface (traction, pressure, and temperature) and at the other boundaries (zero-displacement).

This is done in the **FieldSpecifications** part of the XML file. The attribute `initialCondition` is set to 1 for the blocks specifying the initial pressure, temperature, and effective stress.

```
    <FieldSpecification
      name="initialPressure"
      initialCondition="1"
      setNames="{ all }"
      objectPath="ElementRegions/Domain/cb1"
      fieldName="pressure"
      scale="0.0"/>

    <FieldSpecification
      name="initialTemperature"
      initialCondition="1"
      setNames="{ all }"
      objectPath="ElementRegions/Domain/cb1"
      fieldName="temperature"
      scale="273.0"/>

    <FieldSpecification
      name="initialSigma_x"
```

(continues on next page)

(continued from previous page)

```
      initialCondition="1"
      setNames="{ all }"
      objectPath="ElementRegions/Domain/cb1"
      fieldName="rockSolid_stress"
      component="0"
      scale="2.457"/>
    <FieldSpecification
      name="initialSigma_y"
      initialCondition="1"
      setNames="{ all }"
      objectPath="ElementRegions/Domain/cb1"
      fieldName="rockSolid_stress"
      component="1"
      scale="2.457"/>
    <FieldSpecification
      name="initialSigma_z"
      initialCondition="1"
      setNames="{ all }"
      objectPath="ElementRegions/Domain/cb1"
      fieldName="rockSolid_stress"
      component="2"
      scale="2.457"/>
```

For the zero-displacement boundary conditions, we use the pre-defined set names *xneg* and *xpos*, *yneg*, *zneg* and *zpos* to select the boundary nodes. Note that here, we have considered a slab in the y-direction, which is why a displacement boundary condition is applied on *zpos* and not applied on *ypos*.

```
    <FieldSpecification
      name="xconstraint"
      fieldName="totalDisplacement"
      component="0"
      objectPath="nodeManager"
      setNames="{ xneg, xpos }"/>

    <FieldSpecification
      name="yconstraint"
      fieldName="totalDisplacement"
      component="1"
      objectPath="nodeManager"
      setNames="{ yneg }"/>

    <FieldSpecification
      name="zconstraint"
      fieldName="totalDisplacement"
      component="2"
      objectPath="nodeManager"
      setNames="{ zneg, zpos }"/>
```

On the top surface, we impose the traction boundary condition and the non-isothermal boundary condition specified in (Bai, 2005). We also fix the pore pressure to 0 Pa.

```
    <Traction
```

(continues on next page)

(continued from previous page)

```
      name="traction"
      objectPath="faceManager"
      tractionType="normal"
      scale="-1.0"
      setNames="{ ypos }"
      functionName="timeFunction"/>

    <FieldSpecification
      name="boundaryPressure"
      objectPath="faceManager"
      fieldName="pressure"
      scale="0.0"
      setNames="{ ypos }"/>

    <FieldSpecification
      name="boundaryTemperature"
      objectPath="faceManager"
      fieldName="temperature"
      scale="323.0"
      setNames="{ ypos }"/>
```

Inspecting results

We request an output of the displacements, pressure, and temperature using the **TimeHistory** feature of GEOS. The figures below compare the results from GEOS (dashed line) and the corresponding analytical solution (solid line) as a function of time at different locations of the slab. We obtain a very good match, confirming that GEOS can accurately capture the thermo-poromechanical coupling on this example. The first figure illustrates this good agreement for the pressure evolution.

The second figure confirms the good match with the analytical solution for the temperature.

The third figure shows that GEOS is also able to match the vertical displacement (settlement) analytical solution.

To go further

Feedback on this example

For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

1.4.2 Performance Benchmarks

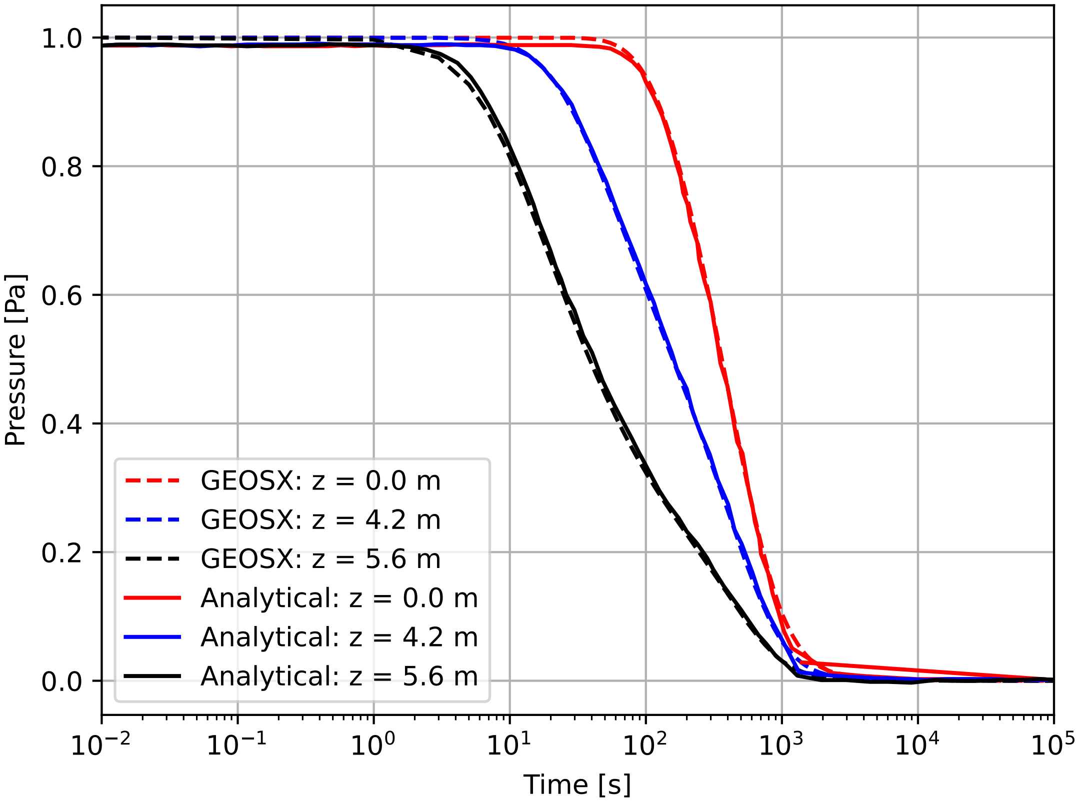
1.4.3 Application Studies

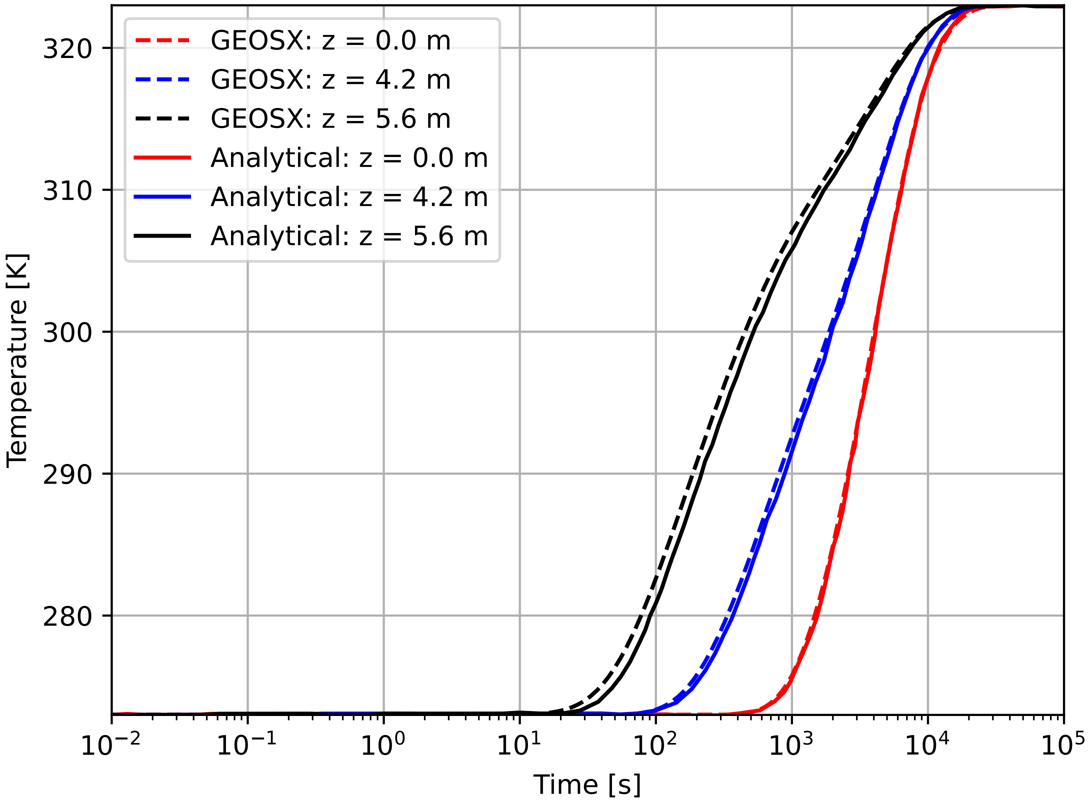
Enhanced Geothermal Systems

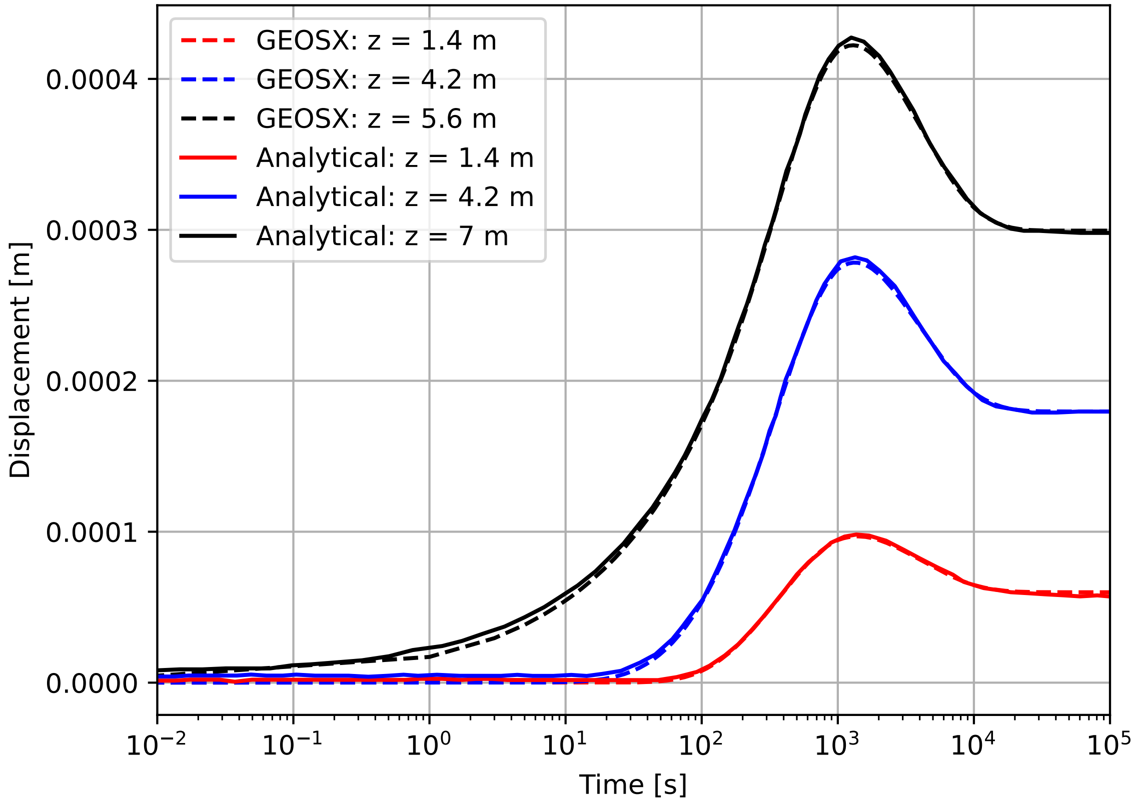
Simulation of Chilled-water Injection at EGS Collab Testbed 2

Context

Here we model the chilled-water injection that occurred at EGS Collab <https://www.energy.gov/eere/geothermal/egs-collab>_ Testbed 2.


Pressure [Pa]
1.0
0.8
0.6
0.4
0.2
0.0
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}
Time [s]
GEOSX: z = 0.0 m
GEOSX: z = 4.2 m
GEOSX: z = 5.6 m
Analytical: z = 0.0 m
Analytical: z = 4.2 m
Analytical: z = 5.6 m


GEOSX: z = 0.0 m
GEOSX: z = 4.2 m
GEOSX: z = 5.6 m
Analytical: z = 0.0 m
Analytical: z = 4.2 m
Analytical: z = 5.6 m
Temperature [K]
320
310
300
290
280
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}
Time [s]


GEOSX: z = 1.4 m
GEOSX: z = 4.2 m
GEOSX: z = 5.6 m
Analytical: z = 1.4 m
Analytical: z = 4.2 m
Analytical: z = 7 m
0.0004
0.0003
0.0002
0.0001
0.0000
Displacement [m]
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}
Time [s]

The simulations are performed with GEOS thermal single-phase flow solver. The mass and energy conservation equations are discretized using a finite-volume methods with using a two-point flux approximation and an upwinding scheme for all fluid properties.

Input file

The xml input files for the test case are located at:

```
inputFiles/thermalSinglePhaseFlowFractures/egsCollab_thermalFlow/egsCollab_thermalFlow_
↪base.xml
inputFiles/thermalSinglePhaseFlowFractures/egsCollab_thermalFlow/egsCollab_thermalFlow_
↪injection_200yInit_base.xml
inputFiles/thermalSinglePhaseFlowFractures/egsCollab_thermalFlow/egsCollab_thermalFlow_
↪injection_coarse.xml
```

Description of the case

We consider a 148.375 m x 75 m x 143.375 m domain and we employ a system of coordinates with origin in {0,0,0}. The drift and battery alcove are represented by the two rectangular prismatic regions highlighted in blue in the figure.

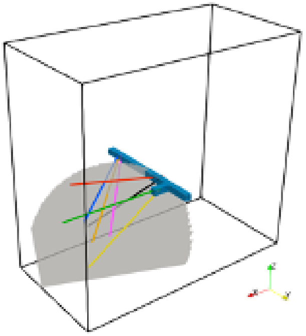
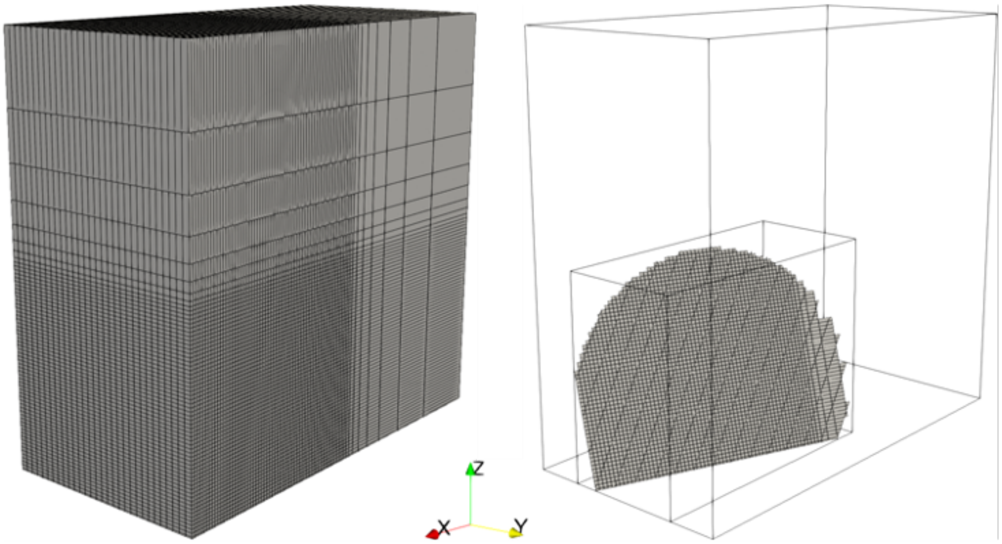


Fig. 1.74: Sketch of the problem

A single embedded planar fracture is considered (i.e., shaded surface in Fig. 1). The fracture unit normal vector is n={0.3569123763,0.9123044601,0.2007837838} and its center is in {100, 37.5, 30}. The fracture location and orientation are consistent with the microseismic data collected during flow testing. Finally, injection well TU, production wells TC and TN and monitoring wells AMU, AML, DMU, DML are also considered as shown in the Figure.

Mesh and embedded fracture geometry

The domain is discretized, in space, using a 68 x 30 x 65 structured mesh, for a total of 132600 elements in the rock matrix. The fracture is generated internally using the *EmbeddedSurfaceGenerator*. The resulting meshes for both the rock matrix and the fracture are shown in the figure below.



Boundary conditions

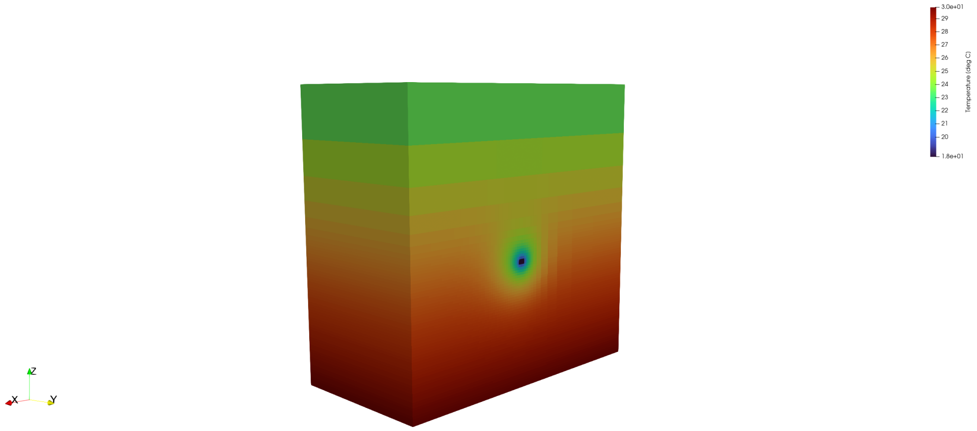
Pressure and temperature Dirichlet boundary conditions are considered at the top boundary (z = 143.375 m), i.e., p_top=2 MPa and T_top=23.0153 °C. The temperature gradient is –0.0488 °C/m whereas hydrostatic pore pressure distribution is assumed. The drift and the alcove are assumed to have constant pressure and temperature. Thus, Dirichlet boundary conditions are considered for both, i.e., T_drift=18.5 °C and T_alcove=20.637 °C.

Flow solver

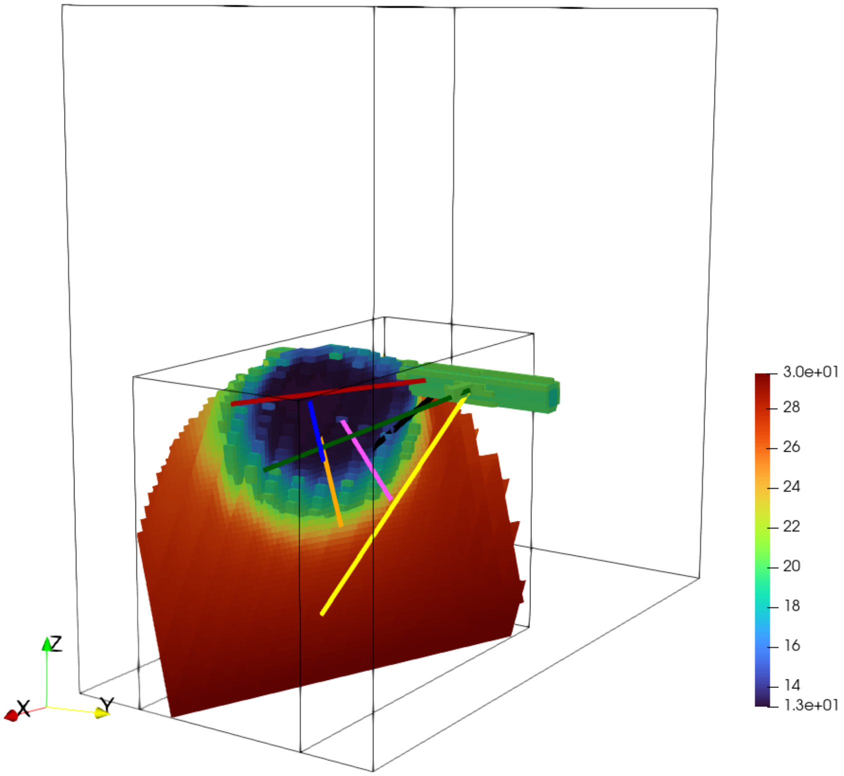
Constitutive laws

Initial conditions

To compute the initial distribution of pressure and temperature prior to the chilled-water injection, we run a simulation for 200 years considering the Dirichlet boundary conditions described in the previous paragraph. The computed initial temperature field is illustrated in the Figure below.



Chilled-water injection



Feedback on this example

For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

1.4.4 pygeosx Examples

The *pygeosx — GEOS in Python* enables users to query and/or manipulate the GEOS datastructure with python in real-time. The following examples show how to use *pygeosx* interface and the supplemental *pygeosx_tools_package*.

A quick warning: these examples are recommended for advanced users only. The pygeosx interface requires users to be familiar with the GEOS datastructure, and can trigger the code to crash if an object/variable is inappropriately modified.

In Situ Data Monitor

Objectives

At the end of this example you will know:

- how to run a problem using the pygeosx interface,
- how to process advanced xml features using pygeosx,
- how to extract and monitor values within the GEOS datastructure in real-time

Input files

This example requires two input xml files and one python script located at:

```
GEOS/examples/pygeosxExamples/hydraulicFractureWithMonitor
```

Description of the case

This example is derived from this basic example: *Hydraulic Fracturing*, which solves for the propagation of a single hydraulic fracture within a heterogeneous reservoir. The pygeosx interface is used to monitor the maximum hydraulic aperture and fracture extents over time.

XML Configuration

The input xml file for this example requires some modification in order to work with pygeosx. First, we use the advanced xml input features to include the base problem and override the *table_root* parameter that points to the table files. Note that these paths will need to be updated if you run this problem outside of the example directory.

```
  <Included>
    <File
      name="../../hydraulicFracturing/heterogeneousInSituProperties/heterogeneousInSitu_singleFracture.xml"/>
  </Included>

  <Parameters>
    <Parameter
      name="table_root"
      value="../../hydraulicFracturing/heterogeneousInSituProperties/tables"/>
  </Parameters>
```

Next, we add a new entry to the output block Python and an entry in the Events block. Whenever the python event is triggered, GEOS will pause and return to the controlling python script (in this case, every 10 cycles).

```
  <Outputs>
    <Python
      name="pythonOutput"/>
  </Outputs>

  <Events>
    <PeriodicEvent
      name="python"
      cycleFrequency="10"
```

(continues on next page)

(continued from previous page)

```
    target="/Outputs/pythonOutput"/>
</Events>
```

Python Script

Problems that use the pygeosx interface are driven by a custom python script. To begin, we import a number of packages and check whether this is a parallel run. The custom packages include pygeosx, which provides an interface to GEOS, and pygeosx_tools, which provides a number of common tools for working with the datastructure and dealing with parallel communication.

```
from mpi4py import MPI
import pygeosx
from pygeosx_tools import wrapper
from geosx_xml_tools.main import preprocess_parallel
import matplotlib.pyplot as plt
```

In the next step, we apply the xml preprocessor to resolve the advanced xml features. Note that this step will modify the input arguments to reflect the location of the compiled xml file, which is processed directly by GEOS. The script then initializes GEOS and receives the *problem* handle, which is the scripts view into the datastructure. There is an opportunity to interact with the GEOS before the initial conditions are set, which we do not use in this example.

```
    # Get the MPI rank
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()

    # Initialize the code and set initial conditions
    args = preprocess_parallel()
    problem = pygeosx.initialize(rank, args)
    pygeosx.apply_initial_conditions()
```

To extract information from the problem, you need to know the full path (or 'key') to the target object. These keys can be quite long, and can change depending on the xml input. In the next step, we use a method from the pygeosx_tools package to search for these keys using a list of keywords. If the keys are known beforehand, then this step could be skipped. Note that these functions will throw an error if they do not find a matching key, or if they find multiple matching keys.

```
    # Rather than specifying the wrapper paths explicitly,
    # search for them using a set of filters
    fracture_location_key = wrapper.get_matching_wrapper_path(problem, ['Fracture',
↪'elementCenter'])
    fracture_aperture_key = wrapper.get_matching_wrapper_path(problem, ['Fracture',
↪'effectiveAperture'])
```

Next, we setup a dictionary that will allow us to use pygeosx_tools to automatically query the problem. The root level of this dictionary contains the target keys (fracture location and aperture) and the required *time* key. These each point to a sub-dictionary that holds an axis label, a scale factor, and an empty list to hold the time history. The target dictionaries also hold an entry *fhandle*, which contains a matplotlib figure handle that we can use to display the results.

```
    # Setup values to record
    records = {fracture_location_key: {'label': 'Fracture Extents (m)',
                                       'scale': 1.0,
```

(continues on next page)

(continued from previous page)

```
                                       'history': [],
                                       'fhandle': plt.figure()},
               fracture_aperture_key: {'label': 'Aperture (mm)',
                                       'scale': 1e3,
                                       'history': [],
                                       'fhandle': plt.figure()},
               'time': {'label': 'Time (min)',
                        'scale': 1.0 / 60.0,
                        'history': []}}
```

After setting up the problem, we enter the main problem loop. Upon calling *pygeosx.run()*, the code will execute until a Python event is triggered in the Event loop. At those points, we have the option to interact with the problem before continuing processing. Here, we use pygeosx_tools to query the datastructure and occasionaly plot the results to the screen.

```
    # Setup values to record
    records = {fracture_location_key: {'label': 'Fracture Extents (m)',
                                       'scale': 1.0,
                                       'history': [],
                                       'fhandle': plt.figure()},
               fracture_aperture_key: {'label': 'Aperture (mm)',
                                       'scale': 1e3,
                                       'history': [],
                                       'fhandle': plt.figure()},
               'time': {'label': 'Time (min)',
                        'scale': 1.0 / 60.0,
                        'history': []}}
```

Manual Query

To obtain and manually inspect an object in the problem, you can use the methods in *pygeosx_tools.wrapper*. These are designed to handle any parallel communication that may be required in your analysis. For example, to get the fracture aperture as a numpy array, you could call:

```
from pygeosx_tools import wrapper

# (problem initialization / configuration)

# Grab aperture as a numpy array, using three different approaches

# Local copy (the write flag indicates that we do not plan to modify the result)
aperture_local = wrapper.get_wrapper(problem, fracture_aperture_key, write_flag=False)

# Global copy on the root rank
aperture_global = wrapper.gather_wrapper(problem, fracture_aperture_key)

# Global copy on the all ranks
aperture_global = wrapper.allgather_wrapper(problem, fracture_aperture_key)
```

Running the Problem

To run the problem, you must use the specific version of python where pygeosx is installed. This is likeley located here:

```
GEOS/[build_dir]/lib/PYGEOSX/bin/python
```

Note that you may need to manually install the pygeosx_tools package (and its pre-requisites) into this python distribution. To do so, you can run the following:

```
cd GEOS/[build_dir]/lib/PYGEOSX/bin
pip install --upgrade ../../../../src/coreComponents/python/modules/pygeosx_tools_
↪package/
```

To run the code, you will call the pygeosx run script with python, and supply the typical geosx command-line arguments and any parallel arguments. For example:

```
# Load the correct python environment
# If you are not using a bash shell, you may need to target one of
# the other activation scripts
source GEOS/[build_dir]/lib/PYGEOSX/bin/activate

# Move to the correct directory and run
cd /path/to/problem
srun -n 36 -ppdebug python hydraulicFractureWithMonitor.py -i hydraulicFracture.xml -x 6␣
↪-y 2 -z 3 -o hf_results
```

To go further

Feedback on this example

For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

For more details

- More on advanced xml features, please see *Advanced XML Features*.
- More on the pygeosx interface, please see *pygeosx — GEOS in Python*.

Initial Condition Modification

Objectives

At the end of this example you will know:

- how to modify GEOS arrays using pygeosx
- handle parallel communication with pygeosx_tools

Input files

This example requires an input xml and python script located at:

```
GEOS/examples/pygeosxExamples/sedovWithStressFunction
```

Description of the case

This example is derived from the sedov integrated test (*GEOS/src/coreComponents/physicsSolvers/solidMechanics/integratedTests/sedov..* which looks at the propagation of elastic waves due to an initial stress field. The pygeosx interface is used to modify the initial conditions of the problem to something of our choosing.

XML Configuration

As before, the basic sedov input xml file for this example requires some modification in order to work with pygeosx. First, we use the advanced xml input features to include the base problem (this path may need to be updated, depending on where you run the problem).

```
<Included>
  <File
    name="../../../inputFiles/solidMechanics/sedov.xml"/>
</Included>
```

Next, we add a new entry to the output block Python and an entry in the Events block. Whenever the python event is triggered, GEOS will pause and return to the controlling python script.

```
<Events>
  <PeriodicEvent
    name="python"
    cycleFrequency="5"
    target="/Outputs/pythonOutput"/>
</Events>

<Outputs>
  <Python
    name="pythonOutput"/>
</Outputs>
```

Python Script

Similar to the previous example, the python script begins by importing the required packages, applying the xml preprocessor, GEOS initialization, and key search.

```
    # Get the MPI rank
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()

    # Initialize the code and set initial conditions
    args = preprocess_parallel()
    problem = pygeosx.initialize(rank, args)
    pygeosx.apply_initial_conditions()

    # Rather than specifying the wrapper paths explicitly,
    # search for them using a set of filters
    location_key = wrapper.get_matching_wrapper_path(problem, ['Region2', 'elementCenter
↪'])
    stress_key = wrapper.get_matching_wrapper_path(problem, ['Region2', 'shale', 'stress
```

(continues on next page)

(continued from previous page)

```
↪'])
    ghost_key = wrapper.get_matching_wrapper_path(problem, ['Region2', 'cb1', 'ghostRank
↪'])
```

The next steps rely on a python function that we use to set stress. The argument to this function, x, is assumed to be a numpy array of element centers:

```
def stress_fn(x):
    """
    Function to set stress values

    Args:
        x (np.ndarray) the element centers

    Returns:
        np.ndarray: stress values
    """
    R = x[:, 0]**2 + x[:, 1]**2 + x[:, 2]**2
    return np.sin(2.0 * np.pi * R / np.amax(R))
```

In the following section, we zero out the initial stress and then set it based on *stress_fn*. While doing this, we use *wrapper.print_global_value_range* to check on the process.

```
    # Print initial stress
    wrapper.print_global_value_range(problem, stress_key, 'stress')

    # Zero out stress
    wrapper.set_wrapper_to_value(problem, stress_key, 0.0)
    wrapper.print_global_value_range(problem, stress_key, 'stress')

    # Set stress via a function
    wrapper.set_wrapper_with_function(problem, stress_key, location_key, stress_fn,␣
↪target_index=0)
    wrapper.set_wrapper_with_function(problem, stress_key, location_key, stress_fn,␣
↪target_index=1)
    wrapper.set_wrapper_with_function(problem, stress_key, location_key, stress_fn,␣
↪target_index=2)
    wrapper.print_global_value_range(problem, stress_key, 'stress')
```

Finally, we run the simulation. As an optional step, we extract numpy arrays from the datastructure using different parallel approaches:

```
    # Run the code
    while pygeosx.run() != pygeosx.COMPLETED:
        wrapper.print_global_value_range(problem, stress_key, 'stress')

        # Gather/allgather tests
        tmp = wrapper.gather_wrapper(problem, stress_key)
        print(wrapper.rank, 'gather', np.shape(tmp), flush=True)

        tmp = wrapper.allgather_wrapper(problem, stress_key)
        print(wrapper.rank, 'allgather', np.shape(tmp), flush=True)
```

(continues on next page)

(continued from previous page)

```
        tmp = wrapper.allgather_wrapper(problem, stress_key, ghost_key=ghost_key)
        print(wrapper.rank, 'allgather_ghost_filtered', np.shape(tmp), flush=True)
```

Running the Problem

To run the problem, you must use the specific version of python where pygeosx is installed. This is likeley located here:

```
GEOS/[build_dir]/lib/PYGEOSX/bin/python
```

Note that you may need to manually install the pygeosx_tools package (and its pre-requisites) into this python distribution. To do so, you can run the following:

```
cd GEOS/[build_dir]/lib/PYGEOSX/bin
pip install --upgrade ../../../../src/coreComponents/python/modules/pygeosx_tools_
↪package/
```

To run the code, you will call the pygeosx run script with python, and supply the typical geosx command-line arguments and any parallel arguments. For example:

```
# Load the correct python environment
# If you are not using a bash shell, you may need to target one of
# the other activation scripts
source GEOS/[build_dir]/lib/PYGEOSX/bin/activate

# Move to the correct directory and run
cd /path/to/problem
python run_sedov_problem.py -i modified_sedov.xml -o results
```

To go further

Feedback on this example

For any feedback on this example, please submit a GitHub issue on the project's GitHub page.

For more details

- More on advanced xml features, please see *Advanced XML Features*.
- More on the pygeosx interface, please see *pygeosx — GEOS in Python*.

1.5 User Guide

Welcome to the GEOS user guide.

1.5.1 Input Files

XML

GEOS is configured via one (or more) Extensible Markup Language (XML) files. These files contain a set of elements and attributes that closely follow the internal datastructure of GEOS. When running GEOS, these files are specified using the *-i* argument:

```
geosx -i input.xml
```

XML Components

The following illustrates some of the key features of a GEOS-format xml file:

```
<?xml version="1.0" ?>

<Problem>
    <BlockA
        someAttribute="1.234">

        <!-- Some comment -->
        <BlockB
          name="firstNamedBlock"
          anotherAttribute="0"/>
        <BlockB
          name="secondNamedBlock"
          anotherAttribute="1"/>
    </BlockA>
</Problem>
```

The two basic components of an xml file are blocks, which are specified using angle brackets (“<BlockA> </BlockA>”), and attributes that are attached to blocks (attributeName=”attributeValue”). Block and attributes can use any ASCII character aside from <, &, ‘, and “ (if necessary, use *&lt;*, *&amp;*, *&apos;*, or *&quot;*). Comments are indicated as follows: *<!– Some comment –>*.

At the beginning of a GEOS input file, you will find an optional xml declaration (*<?xml version=”1.0” ?>*) that is used to indicate the format to certain text editors. You will also find the root *Problem* block, where the GEOS configuration is placed. Note that, aside from these elements and commented text, the xml format requires that no other objects exist at the first level.

In the example above, there is a single element within the *Problem* block: *BlockA*. *BlockA* has an attribute *someAttribute*, which has a value of 1.234, and has three children: a commented string “Some comment” and two instances of *BlockB*. The *name* attribute is required for blocks that allow multiple instances, and should include a unique string to avoid potential errors. Where applicable these blocks will be executed in the order in which they are specified in input file.

Input Validation

The optional *xmlns:xsi* and *xsi:noNamespaceSchemaLocation* attributes in the Problem block can be used to indicate the type of document and the location of the xml schema to the text editor:

```
<Problem
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="/path/to/schema.xsd" />
```

The schema contains a list of xml blocks and attributes that are supported by GEOS, indicates whether a given object is optional or required, and defines the format of the object (string, floating point number, etc.). A copy of the schema is included in the GEOS source code (/path/to/GEOS/src/coreComponents/schema/schema.xsd). It can also be generated using GEOS: `geosx -s schema.xsd`

Many text editors can use the schema to help in the construction of an xml file and to indicate whether it is valid. Using a validation tool is highly recommended for all users. The following instructions indicate how to turn on validation for a variety of tools:

xmllint

xmllint is a command-line tool that is typically pre-installed on UNIX-like systems. To check whether an input file is valid, run the following command:

xmllint –schema /path/to/schema.xsd input_file.xml

Sublime Text

We recommend using the Exalt or SublimeLinter_xmllint plug-ins to validate xml files within sublime. If you have not done so already, install the sublime Package Control. To install the package, press `ctrl + shift + p`, type and select `Package Control: Install Package`, and search for `exalt` or `SublimeLinter` / `SublimeLinter-xmllint`. Note that, depending on the circumstances, these tools may indicate only a subset of the validation errors at a given time. Once resolved, the tools should re-check the document to look for any additional errors.

As an additional step for SublimLinter-xmllint, you will need to add a linter configuration. To do so, go to Preferences/Package Settings/SublimeLinter/Settings. In the right-hand side of the new window, add the xmllint configuration:

```
{
    "linters": {
        "xmllint":
        {
            "args": "--schema /path/to/schema.xsd",
            "styles": [
                {
                    "mark_style": "fill",
                    "scope": "region.bluish",
                    "types": ["error"],
                    "icon": "stop",
                }
            ]
        },
    }
}
```

VS Code

We recommend using the XML for validating xml files. After installing this extension, you can associate GEOS format xml files by adding the following entry to the user settings file (replacing *systemId* with the correct path to the schema file):

```
{
    "xml.fileAssociations": [

        {
            "pattern": "**.xml",
            "systemId": "/path/to/GEOS/src/coreComponents/schema/schema.xsd"
        }
    ]
}
```

Eclipse

The Eclipse Web Develop Tools includes features for validating xml files. To install them, go to Help -> Eclipse Marketplace, search for the Eclipse Web Developer Tools, install the package, and restart Eclipse. Finally, configure the xml validation preferences under Window -> Preferences -> XML -> XML Files -> Validation. Eclipse will automatically fetch the schema, and validate an active xml file. The editor will highlight any lines with errors, and underline the specific errors.

GEOS XML Tools

The geosx_xml_tools package, which is used to enable advanced features such as parameters, symbolic math, etc., contains tools for validating xml files. To do so, call the command-line script with the -s argument, i.e.: *preprocess_xml input_file.xml -s /path/to/schema.xsd*. After compiling the final xml file, pygeosx will fetch the designated schema, validate, and print any errors to the screen.

Note: Attributes that are using advanced xml features will likely contain characters that are not allowed by their corresponding type pattern. As such, file editors that are configured to use other validation methods will likely identify errors in the raw input file.

XML Schema

An XML schema definition (XSD) file lays out the expected structure of an input XML file. During the build process, GEOS automatically constructs a comprehensive schema from the code's data structure, and updates the version in the source (GEOS/src/coreComponents/schema/schema.xsd).

Schema Components

The first entry in the schema are a set of headers the file type and version. Following this, the set of available simple types for attributes are laid out. Each of these includes a variable type name, which mirrors those used in the main code, and a regular expression, which is designed to match valid inputs. These patterns are defined and documented in `DataTypes::typeRegex`. The final part of the schema is the file layout, beginning with the root `Problem`. Each complex type defines an element, its children, and its attributes. Each attribute defines the input name, type, default value, and/or usage. Comments preceding each attribute are used to relay additional information to the users.

Automatic Schema Generation

A schema may be generated by calling the main code with the -s argument , e.g.: `geosx -s schema.xsd` (Note: this is done automatically during the bulid process). To do this, GEOS does the following:

1) Initialize the GEOS data structure.
2) Initialize objects that are registered to catalogs via `ManagedGroup::ExpandObjectCatalogs()`.
3) Recursively write element and attribute definitions to the schema using information stored in GEOS groups and wrappers.
4) Define any expected deviations from the schema via `ManagedGroup::SetSchemaDeviations()`.

Advanced XML Features

The *geosx_xml_tools* python package adds a set of advanced features to the GEOS xml format: units, parameters, and symbolic expressions. See *Python Tools Setup* for details on setup instructions, and *GEOS XML Tools* for package API details.

Usage

An input file that uses advanced xml features requires preprocessing before it can be used with GEOS. The preprocessor writes a compiled xml file to the disk, which can be read directly by GEOS and serves as a permanent record for the simulation. There are three ways to apply the preprocessor:

1) Automatic Preprocessing: Substituting *geosx* for *geosx_preprocessed* when calling the code will automatically apply the preprocessor to the input xml file, and then pass the remaining arguments to GEOS. With this method, the compiled xml files will have the suffix '.preprocessed'. Before running the code, the compiled xml file will also be validated against the xml schema.

```
# Serial example
geosx_preprocessed -i input.xml

# Parallel example
srun -n 2 geosx_preprocessed -i input.xml -x 2
```

2) Manual Preprocessing: For this approach, xml files are preprocessed manually by the user with the *preprocess_xml* script. These files can then be submitted to GEOS separately:

```
# The -c argument is used to manually specify the compiled name
preprocess_xml -i input.xml -c input.xml.processed
geosx -i input.xml.processed

# Otherwise, a random name will be chosen by the tool
compiled_input=$(preprocess_xml input.xml)
geosx -i $compiled_input
```

3) Python / pygeosx: The preprocessor can also be applied directly in python or in pygeosx simulations. An example of this is method is provided here: *GEOS/examples/pygeosxExamples/hydraulicFractureWithMonitor/*.

Each of these options support specifying multiple input files via the command line (e.g. *geosx_preprocessed -i input_a.xml -i input_b.xml*). They also support any number of command-line parameter overrides (e.g. *geosx_preprocessed -i input_a.xml -p parameter_a alpha -p parameter_b beta*).

Included Files

Both the XML preprocessor and GEOS executable itself provide the capability to build complex multi-file input decks by including XML files into other XML files.

The files to be included are listed via the <*Included*> block. There maybe any number of such blocks. Each block contains a list of <*File name=”. . . ”/*> tags, each indicating a file to include. The *name* attribute must contain either an absolute or a relative path to the included file. If the path is relative, it is treated as relative to the location of the referring file. Included files may also contain includes of their own, i.e. it is possible to have *a.xml* include *b.xml* which in turn includes *c.xml*.

Note: When creating multi-file input decks, it is considered best practice to use relative file paths. This applies both to XML includes, and to other types of file references (for example, table file names). Relative paths keep input decks both relocatable within the file system and sharable between users.

XML preprocessor’s merging capabilities are more advanced than GEOS built-in ones. Both are outlined below.

XML preprocessor

The merging approach is applied recursively, allowing children to include their own files. Any potential conflicts are handled via the following scheme:

- **Merge two objects if:**
  - At the root level an object with the matching tag exists.
  - If the “name” attribute is present and an object with the matching tag and name exists.
  - Any preexisting attributes on the object are overwritten by the donor.
- Otherwise append the XML structure with the target.

GEOS

GEOS's built-in processing simply inserts the included files' content (excluding the root node) into the XML element tree, at the level of *<Included>* tag. Partial merging is handled implicitly by GEOS's data structure, which treats repeated top-level XML blocks as if they are one single block. This is usually sufficient for merging together top-level input sections from multiple files, such as multiple *<FieldSpecifications>* or *<Events>* sections, but more complex cases may require the use of preprocessor.

Note: While GEOS's XML processing is capable of handling any number of *<Included>* block at any level, the XML schema currently produced by GEOS only allows a single such block, and only directly within the *<Problem>* tag. Inputs that use multiple blocks or nest them deeper may run but will fail to validate against the schema. This is a known discrepancy that may be fixed in the future.

Parameters

Parameters are a convenient way to build a configurable and human-readable input XML. They are defined via a block in the XML structure. To avoid conflicts with other advanced features, parameter names can include upper/lower case letters and underscores. Parameters may have any value, including:

- Numbers (with or without units)
- A path to a file
- A symbolic expression
- Other parameters
- Etc.

They can be used as part of any input xml attribute as follows:

- x_par (preferred)
- $x_par
- $:x_par
- $:x_par$

Attributes can be used across Included files, but cannot be used to set the names of included files themselves. The following example uses parameters to set the root path for a table function, which is then scaled by another parameter:

```xml
<Parameters>
  <Parameter
    name="flow_scale"
    value="0.5"/>
  <Parameter
    name="table_root"
    value="/path/to/table/root"/>
</Parameters>

<FieldSpecifications>
  <SourceFlux
    name="sourceTerm"
    objectPath="ElementRegions/Region1/block1"
    scale="$flow_scale$"
```

(continues on next page)

(continued from previous page)

```
    functionName="flow_rate"
    setNames="{ source }"/>
</FieldSpecifications>

<Functions>
  <TableFunction
    name="flow_rate"
    inputVarNames="{time}"
    coordinateFiles="{$table_root$/time_flow.geos}"
    voxelFile="$table_root$/flow.geos"
    interpolation="linear"/>
</Functions>
```

Any number of parameter overrides can be issued from the command line using the *-p name value* argument in the preprocessor script. Note that if the override value contains any spaces, it may need to be surrounded by quotation marks (*-p name "paramter with spaces"*).

Units

The units for any input values to GEOS can be in any self-consistent system. In many cases, it is useful to override this behavior by explicitly specifying the units of the input. These are specified by appending a valid number with a unit definition in square braces. During pre-processing, these units are converted into base-SI units (we plan to support other unit systems in the future).

The unit manager supports most common units and SI prefixes, using both long- and abbreviated names (e.g.: c, centi, k, kilo, etc.). Units may include predefined composite units (dyne, N, etc.) or may be built up from sub-units using a python syntax (e.g.: [N], [kg*m/s**2]). Any (or no) amount of whitespace is allowed between the number and the unit bracket. The following shows a set of parameters with units specified:

```
<Parameters>
  <Parameter name="paramter_a" value="2[m]"/>
  <Parameter name="paramter_b" value="1.2 [cm]"/>
  <Parameter name="paramter_c" value="1.23e4 [bbl/day]"/>
  <Parameter name="paramter_d" value="1.23E-4 [km**2]"/>
</Parameters>
```

Please note that the preprocessor currently does not check whether any user-specified units are appropriate for a given input or symbolic expression.

Symbolic Expressions

Input XML files can also include symbolic mathematical expressions. These are placed within pairs of backticks (`), and use a limited python syntax. Please note that parameters and units are evaluated before symbolic expressions. While symbolic expressions are allowed within parameters, errors may occur if they are used in a way that results in nested symbolic expressions. Also, note that residual alpha characters (e.g. *sin()* are removed before evaluation for security. The following shows an example of symbolic expressions:

```
<Parameters>
  <Parameter name="a" value="2[m]"/>
  <Parameter name="b" value="1.2 [cm]"/>
  <Parameter name="c" value="3"/>
```

(continues on next page)

(continued from previous page)

```
  <Parameter name="d" value="1.23e-4"/>
</Parameters>
<Geometry>
  <Box
    name="perf"
    xMin="{`$a$ - 0.2*$b$`, -1e6, -1e6}"
    xMax="{`$c$**2 / $d$`, 1e6, 1e6}" />
</Geometry>
```

Validation

Unmatched special characters ($, [, `, etc.) in the final xml file indicate that parameters, units, or symbolic math were not specified correctly. If the prepreprocessor detects these, it will throw an error and exit. Additional validation of the compiled files can be completed with *preprocess_xml* by supplying the -s argument and the path to the GEOS schema.

1.5.2 Meshes

The purpose of this document is to explain how users and developers interact with mesh data. This section describes how meshes are handled and stored in GEOS.

There are two possible methods for generating a mesh: either by using GEOS's internal mesh generator (for Cartesian meshes only), or by importing meshes from various common mesh file formats. This latter options allows one to work with more complex geometries, such as unstructured meshes comprised of a variety of element types (polyhedral elements).

Internal Mesh Generation

Basic Example

The Internal Mesh Generator allows one to quickly build simple cartesian grids and divide them into several regions. The following attributes are supported in the input block for InternalMesh:

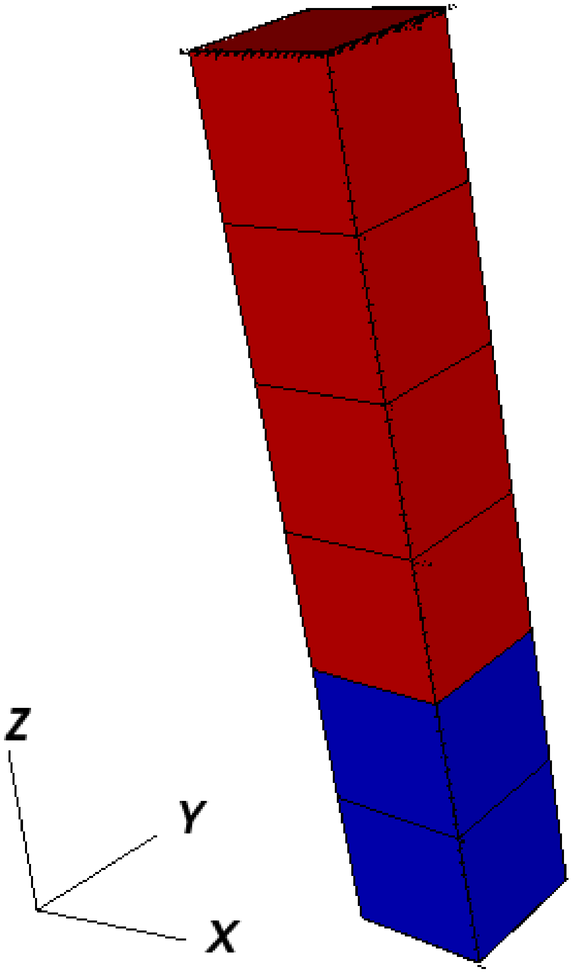
Name	Type	Default	Description
cellBlockNames	string_arra	required	Names of each mesh block
elementTypes	string_arra	required	Element types of each mesh block
name	string	required	A name is required for any non-unique nodes
nx	integer_array	required	Number of elements in the x-direction within each mesh block
ny	integer_array	required	Number of elements in the y-direction within each mesh block
nz	integer_array	required	Number of elements in the z-direction within each mesh block
positionTolerance	real64	1e-10	A position tolerance to verify if a node belong to a nodeset
trianglePattern	integer	0	Pattern by which to decompose the hex mesh into wedges
xBias	real64_arra	{1}	Bias of element sizes in the x-direction within each mesh block (dx_left=(1+b)*L/N, dx_right=(1-b)*L/N)
xCoords	real64_arra	required	x-coordinates of each mesh block vertex
yBias	real64_arra	{1}	Bias of element sizes in the y-direction within each mesh block (dy_left=(1+b)*L/N, dx_right=(1-b)*L/N)
yCoords	real64_arra	required	y-coordinates of each mesh block vertex
zBias	real64_arra	{1}	Bias of element sizes in the z-direction within each mesh block (dz_left=(1+b)*L/N, dz_right=(1-b)*L/N)
zCoords	real64_arra	required	z-coordinates of each mesh block vertex
InternalWell	node		*Element: InternalWell*

The following is an example XML <mesh> block, which will generate a vertical beam with two CellBlocks (one in red and one in blue in the following picture).

```
<Mesh>
  <InternalMesh name="mesh"
                elementTypes="C3D8"
                xCoords="0, 1"
                yCoords="0, 1"
                zCoords="0, 2, 6"
                nx="1"
                ny="1"
                nz="2, 4"
                cellBlockNames="cb1 cb2"/>
</Mesh>
```

- name the name of the mesh body
- elementTypes the type of the elements that will be generated.
- xCoord List of x coordinates of the boundaries of the CellBlocks
- yCoord List of y coordinates of the boundaries of the CellBlocks

- zCoord List of z coordinates of the boundaries of the CellBlocks
- nx List containing the number of cells in x direction within the CellBlocks
- ny List containing the number of cells in y direction within the CellBlocks
- nz List containing the number of cells in z direction within the CellBlocks
- cellBlockNames List containing the names of the CellBlocks



Mesh Bias

The internal mesh generator is capable of producing meshes with element sizes that vary smoothly over space. This is achieved by specifying xBias, yBias, and/or zBias fields. (Note: if present, the length of these must match nx, ny, and nz, respectively, and each individual value must be in the range (-1, 1).)

For a given element block, the average element size will be

$$dx_{average}[i] = \frac{xCoords[i+1] - xCoords[i]}{nx[i]},$$

the element on the left-most side of the block will have size

$$dx_{left}[i] = (1 + xBias[i]) \cdot dx_{average}[i],$$

and the element on the right-most side will have size

$$dx_{right}[i] = (1 - xBias[i]) \cdot dx_{average}[i].$$

The following are the two most common scenarios that occur while designing a mesh with bias:

1. The size of the block and the element size on an adjacent region are known. Assuming that we are to the left of the target block, the appropriate bias would be:

$$xBias[i] = 1 - \frac{nx[i] \cdot dx_{left}[i+1]}{xCoords[i+1] - xCoords[i]}$$

2. The bias of the block and the element size on an adjacent region are known. Again, assuming that we are to the left of the target block, the appropriate size for the block would be:

$$xCoords[i+1] - xCoords[i] = \frac{nx[i] \cdot dx_{left}[i+1]}{1 - xBias[i]}$$

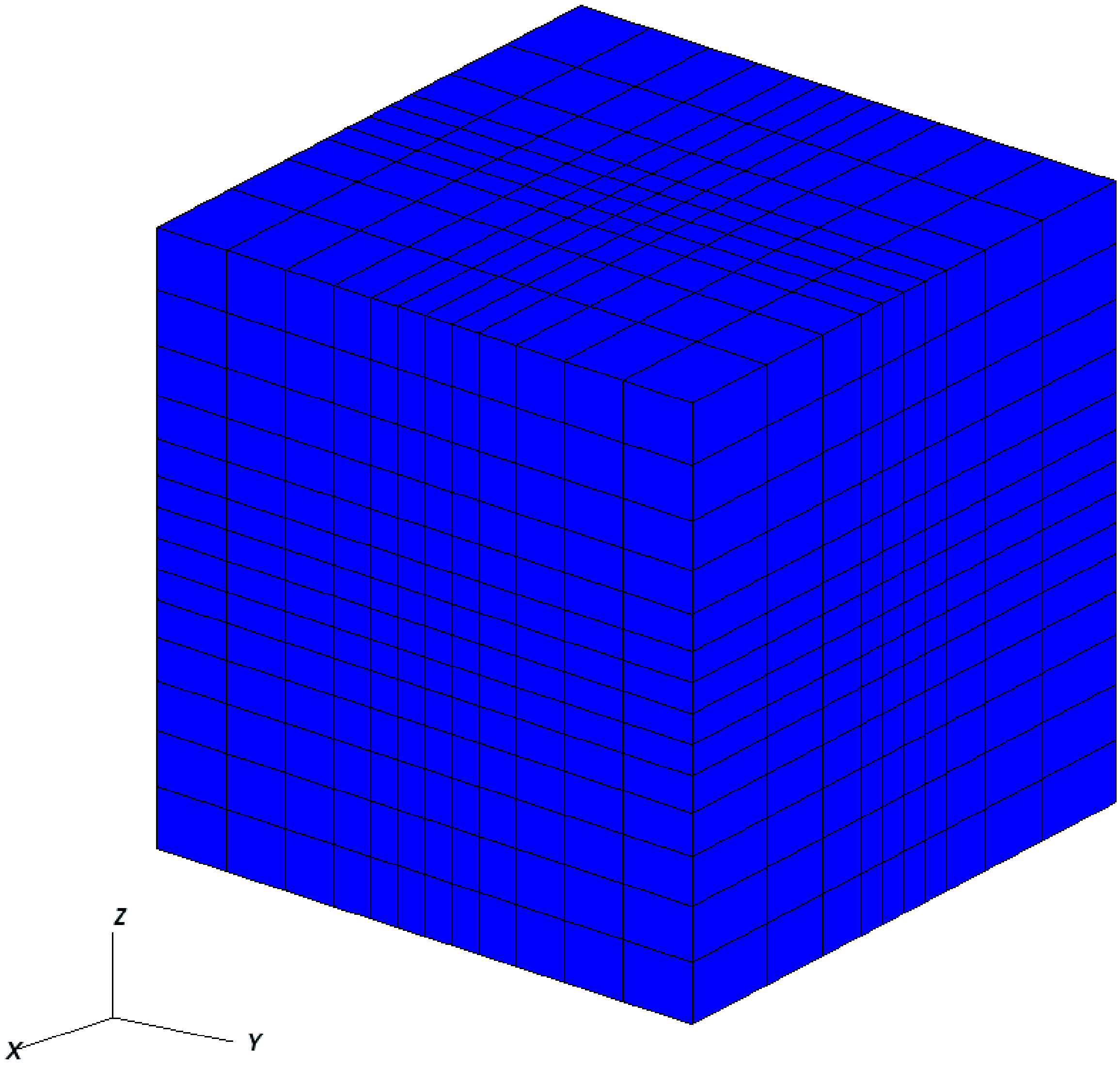
The following is an example of a mesh block along each dimension, and an image showing the corresponding mesh. Note that there is a core region of elements with zero bias, and that the transitions between element blocks are smooth.

```
<Mesh>
  <InternalMesh
    name="mesh1"
    elementTypes="{ C3D8 }"
    xCoords="{ -10, -1, 0, 1, 10 }"
    yCoords="{ -10, -1, 0, 1, 10 }"
    zCoords="{ -10, -1, 0, 1, 10 }"
    nx="{ 4, 1, 1, 4 }"
    ny="{ 5, 1, 1, 5 }"
    nz="{ 6, 1, 1, 6 }"
    xBias="{ 0.555, 0, 0, -0.555 }"
    yBias="{ 0.444, 0, 0, -0.444 }"
    zBias="{ 0.333, 0, 0, -0.333 }"
    cellBlockNames="{ cb1 }"/>
</Mesh>

<Solvers>
  <SolidMechanics_LagrangianFEM
    name="lagsolve"
    strainTheory="1"
    cflFactor="0.25"
```

(continues on next page)

(continued from previous page)

```
    discretization="FE1"
    targetRegions="{ Region2 }"
    />
  </Solvers>
```



Advanced Cell Block Specification

It's possible to generate more complex CellBlock using the InternalMeshGenerator. For instance, the staircase example is a model which is often used in GEOS as an integrated test. It defines CellBlocks in the three directions to generate a staircase-like model with the following code.

```
<Mesh>
  <InternalMesh name="mesh1"
                elementTypes="{C3D8}"
```

(continues on next page)

(continued from previous page)

```
                xCoords="{0, 5, 10}"
                yCoords="{0, 5, 10}"
                zCoords="{0, 2.5, 5, 7.5, 10}"
                nx="{5, 5}"
                ny="{5, 5}"
                nz="{3, 3, 3, 3}"
                cellBlockNames="{b00,b01,b02,b03,b04,b05,b06,b07,b08,b09,b10,b11,b12,b13,
↪b14,b15}"/>
</Mesh>

<ElementRegions>
   <CellElementRegion name="Channel"
                  cellBlocks="{b08,b00,b01,b05,b06,b14,b15,b11}"
                  materialList="{fluid1, rock, relperm}"/>
   <CellElementRegion name="Barrier"
                  cellBlocks="{b04,b12,b13,b09,b10,b02,b03,b07}"
                  materialList="{}"/>
</ElementRegions>
```

Thus, the generated mesh will be :

Using an External Mesh

Supported Formats

GEOS provides features to run simulations on unstructured meshes. It uses VTK to read the external meshes and its API to write it into the GEOS mesh data structure.

The supported mesh elements for volume elements consist of the following:

- 4-node tetrahedra,
- 5-node pyramids,
- 6-node wedges,
- 8-node hexahedra,
- n-gonal prisms (n = 7, . . . , 11).

The mesh can be divided in several regions. These regions are intended to support different physics or to define different constitutive properties. We usually use the `attribute` field is usually considered to define the regions.

Importing the Mesh

Importing regions

Several blocks are involved to import an external mesh into GEOS, defined in the XML input file. These are the `<Mesh>` block and the `<ElementRegions>` block.

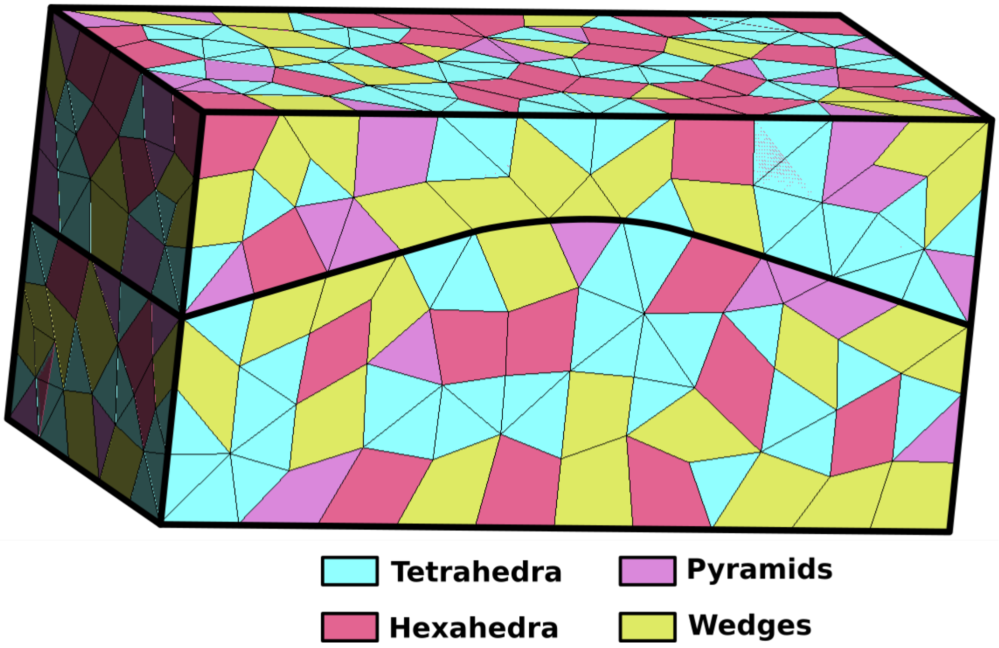
The mesh block has the following syntax:

```
<Mesh>
  <VTKMesh
    name="MyMeshName"
    logLevel="1"
    file="/path/to/the/mesh/file.vtk"/>
</Mesh>
```

We advise users to use absolute path to the mesh file, and strongly recommend the use of a logLevel of 1 or more to obtain some information about the mesh import. This information contains for example the list of regions that are imported with their names, which is particularly useful to fill the `cellBlocks` field of the `ElementRegions` block (see below). Some information about the imported surfaces is also provided.

GEOS uses `ElementRegions` to support different physics or to define different constitutive properties. The `ElementRegions` block can contain several `CellElementRegion` blocks. A `CellElementRegion` is defined as a set of `CellBlocks`. A `CellBlock` is an ensemble of elements with the same element geometry.

The naming of the imported `cellBlocks` depends on whether the data array called `regionAttribute` is present in the vtu file or not. This data array is used to define regions in the vtu file and assign the cells to a given region. The `regionAttribute` is a integer and not a string (unfortunately).



In the example presented above, the mesh is is composed of two regions (*Top* and *Bot*). Each region contains 4 `cellBlocks`.

- If the vtu file does not contain `regionAttribute`, then all the cells are grouped in a single region, and the cell block names are just constructed from the cell types (hexahedra, wedges, tetrahedra, etc). Then in the exemple above, the `ElementRegions` can be defined as bellow:

```
<ElementRegions>
  <CellElementRegion
    name="cellRegion"
    cellBlocks="{ hexahedra, wedges, tetrahedra, pyramids }"
    materialList="{ water, rock }" />
</ElementRegions>
```

- If the vtu file contains regionAttribute, then the cells are grouped by regions based on their individual (numeric) regionAttribute. In that case, the naming convention for the cellBlocks is regionAttribute_elementType. Let's assume that the top region of the exemple above is identified by the regionAttribute 1, and that the bottom region is identified with 2,
  - If we want the CellElementRegion to contain all the cells, we write:

```
<ElementRegions>
  <CellElementRegion
    name="cellRegion"
    cellBlocks="{ 1_hexahedra, 1_wedges, 1_tetrahedra, 1_pyramids, 3_hexahedra, 3_
↪wedges, 3_tetrahedra, 3_pyramids }"
    materialList="{ water, rock }" />
</ElementRegions>
```

  - If we want two CellElementRegion with the top and bottom regions separated, we write:

```
<ElementRegions>
  <CellElementRegion
    name="Top"
    cellBlocks="{ 1_hexahedra, 1_wedges, 1_tetrahedra, 1_pyramids }"
    materialList="{ water, rock }"/>
  <CellElementRegion
    name="Bot"
    cellBlocks="{ 3_hexahedra, 3_wedges, 3_tetrahedra, 3_pyramids }"
    materialList="{ water, rock }" />
</ElementRegions>
```

Warning: We remind the user that **all** the imported cellBlocks must be included in one of the CellElementRegion. Even if some cells are meant to be inactive during the simulation, they still have to be included in a CellElementRegion (this CellElementRegion should simply not be included as a targetRegion of any of the solvers involved in the simulation).

The keywords for the cellBlocks element types are :

- hexahedra
- tetrahedra
- wedges
- pyramids
- pentagonalPrisms
- hexagonalPrisms
- heptagonalPrisms
- octagonalPrisms
- nonagonalPrisms
- decagonalPrisms
- hendecagonalPrisms
- polyhedra

An example of a vtk file with all the physical regions defined is used in *Tutorial 3: Regions and Property Specifications*.

Importing surfaces

Surfaces are imported through point sets in GEOS. This feature is supported using only the `vtk` file format. In the same way than the regions, the surfaces of interests can be defined using the **`physical entity names`_**. The surfaces are automatically import in GEOS if they exist in the `vtk` file. Within GEOS, the point set will have the same name than the one given in the file. This name can be used again to impose boundary condition. For instance, if a surface is named "Bottom" and the user wants to impose a Dirichlet boundary condition of 0 on it, it can be easily done using this syntax.

```
<FieldSpecification
  name="zconstraint"
  objectPath="nodeManager"
  fieldName="Velocity"
  component="2"
  scale="0.0"
  setNames="{ Bottom }"/>
```

The name of the surface of interest appears under the keyword `setNames`. Again, an example of a `vtk` file with the surfaces fully defined is available within *Tutorial 3: Regions and Property Specifications* or *CO2 Plume Evolution With Hysteresis Effect on Relative Permeability*.

1.5.3 Physics Solvers

The <`Solvers`> section of the input file specifies one or several physics solvers to be included in the simulation.

Solution Strategy

All physics solvers share a common solution strategy for nonlinear time-dependent problems. Here, we briefly describe the nonlinear solver and the timestepping strategy employed.

Nonlinear Solver

At each time-step, the nonlinear system of discrete residual equations, i.e.

$$r(x) = 0$$

is solved by employing the Newton-Raphson method. Here, x is the vector of primary unknowns. Thus, each physics solver is responsible for assembling the Jacobian matrix J containing the analytical derivatives of the residual vector r with respect to the primary variables. Then, at each Newton iteration ν, the following linear system is solved

$$J^{\nu} \delta x^{\nu+1} = -r^{\nu},$$

where, $\delta x^{\nu+1}$ is the Newton update. This linear system can be solved with a variety of different linear solvers described in *Linear Solvers*. The Newton update, $\delta x^{\nu+1}$ is then applied to the primary variables:

$$x^{\nu+1} = x^{\nu} + \delta x^{\nu+1}.$$

This procedure is repeated until convergence is achieved or until the maximum number of iterations is reached.

Line Search

A line search method can be applied along with the Newton's method to facilitate Nonlinear convergence. After the Newton update, if the residual norm has increased instead of decreased, a line search algorithm is employed to correct the Newton update.

The user can choose between two different behaviors in case the line search fails to provide a reduced residual norm:

1. accept the solution and move to the next Newton iteration;
2. reject the solution and request a timestep cut;

Timestepping Strategy

The actual timestep size employed is determined by a combination of several factors. In particular, specific output events may have timestep requirements that force a specific timestep to be used. However, physics solvers do have the possibility of requesting a specific timestep size to the event manager based on their specific requirements. In particular, in case of fast convergence indicated by a small number of Newton iterations, i.e.

$$\text{numIterations} < \text{dtIncIterLimit} \cdot \text{newtonMaxIter},$$

the physics solver will require to double the timestep size. On the other hand, if a large number of nonlinear iterations are necessary to find the solution at timestep n

$$\text{numIterations} > \text{dtCutIterLimit} \cdot \text{newtonMaxIter},$$

the physics solver will request the next timestep, $n + 1$, to be half the size of timestep n. Here,

Additionally, in case the nonlinear solver fails to converge with the timestep provided by the event manager, the timestep size is cut, i.e.

$$\text{dt} = \text{timestepCutFactor} \cdot \text{dt},$$

and the nonlinear loop is repeated with the new timestep size.

Parameters

All parameters defining the behavior of the nonlinear solver and determining the timestep size requested by the physics solver are defined in the NonlinearSolverParameters and are presented in the following table.

Name	Type	Default	Description
allowNonConverged	integer	0	Allow non-converged solution to be accepted. (i.e. exit from the Newton loop without achieving the desired tolerance)
couplingType	geos_NonlinearSolverParam	FullyImplicit	Type of coupling. Valid options: * FullyImplicit * Sequential
lineSearchAction	geos_NonlinearSolverParam	Attempt	How the line search is to be used. Options are: * None - Do not use line search. * Attempt - Use line search. Allow exit from line search without achieving smaller residual than starting residual. * Require - Use line search. If smaller residual than starting resdual is not achieved, cut time step.
lineSearchCutFactor	real64	0.5	Line search cut factor. For instance, a value of 0.5 will result in the effective application of the last solution by a factor of (0.5, 0.25, 0.125, …)
lineSearchInterpolationType	geos_NonlinearSolverParam	Linear	Strategy to cut the solution update during the line search. Options are: * Linear * Parabolic
lineSearchMaxCuts	integer	4	Maximum number of line search cuts.
logLevel	integer	0	Log level
maxAllowedResidualNorm	real64	1e+09	Maximum value of residual norm that is allowed in a Newton loop
maxNumConfigurationAttempts	integer	10	Max number of times that the configuration can be changed
maxSubSteps	integer	10	Maximum number of time sub-steps allowed for the solver
maxTimeStepCuts	integer	2	Max number of time step cuts
newtonMaxIter	integer	5	Maximum number of iterations that are allowed in a

Solid Mechanics Solver

List of Symbols

$$
\begin{aligned}
i, j, k &\equiv \text{indices over spatial dimensions} \\
a, b, c &\equiv \text{indices over nodes} \\
l &\equiv \text{indices over volumetric elements} \\
q, r, s &\equiv \text{indices over faces} \\
n &\equiv \text{indices over time} \\
kiter &\equiv \text{iteration count for non-linear solution scheme} \\
\Omega &\equiv \text{Volume of continuum body} \\
\Omega_{crack} &\equiv \text{Volume of open crack} \\
\Gamma &\equiv \text{External surface of } \Omega \\
\Gamma_t &\equiv \text{External surface where tractions are applied} \\
\Gamma_u &\equiv \text{External surface where kinematics are specified} \\
\Gamma_{crack} &\equiv \text{entire surface of crack} \\
\Gamma_{cohesive} &\equiv \text{surface of crack subject to cohesive tractions} \\
\eta_0 &\equiv \text{set of all nodes} \\
\eta_f &\equiv \text{set of all nodes on flow mesh} \\
m &\equiv \text{mass} \\
\kappa_k &\equiv \text{all elements connected to element k} \\
\phi &\equiv \text{ porosity} \\
p_f &\equiv \text{ fluid pressure} \\
\mathbf{u} &\equiv \text{ displacement} \\
\mathbf{q} &\equiv \text{ volumetric flow rate} \\
\mathbf{T} &\equiv \text{ Cauchy stress} \\
\rho &\equiv \text{ density in the current configuration} \\
\mathbf{x} &\equiv \text{ current position} \\
\mathbf{w} &\equiv \text{ aperture, or gap vector}
\end{aligned}
$$

Introduction

The *SolidMechanics_LagrangianFEM* solver applies a Continuous Galerkin finite element method to solve the linear momentum balance equation. The primary variable is the displacement field which is discretized at the nodes.

Theory

Governing Equations

The *SolidMechanics_LagrangianFEM* solves the equations of motion as given by

$$T_{ij,j} + \rho(b_i - \ddot{x}_i) = 0,$$

which is a 3-dimensional expression for the well known expression of Newtons Second Law ($F = ma$). These equations of motion are discretized using the Finite Element Method, which leads to a discrete set of residual equations:

$$(R_{solid})_{ai} = \int\limits_{\Gamma_t} \Phi_a t_i dA - \int\limits_{\Omega} \Phi_{a,j} T_{ij} dV + \int\limits_{\Omega} \Phi_a \rho (b_i - \Phi_b \ddot{x}_{ib}) dV = 0$$

Quasi-Static Time Integration

The Quasi-Static time integration option solves the equation of motion after removing the inertial term, which is expressed by

$$T_{ij,j} + \rho b_i = 0,$$

which is essentially a way to express the equation for static equilibrium ($\Sigma F = 0$). Thus, selection of the Quasi-Static option will yield a solution where the sum of all forces at a given node is equal to zero. The resulting finite element discretized set of residual equations are expressed as

$$(R_{solid})_{ai} = \int\limits_{\Gamma_t} \Phi_a t_i dA - \int\limits_{\Omega} \Phi_{a,j} T_{ij} dV + \int\limits_{\Omega} \Phi_a \rho b_i dV = 0,$$

Taking the derivative of these residual equations wrt. the primary variable (displacement) yields

$$\frac{\partial (R^e_{solid})_{ai}}{\partial u_{bj}} = -\int\limits_{\Omega^e} \Phi_{a,k} \frac{\partial T_{ik}}{\partial u_{bj}} dV,$$

And finally, the expression for the residual equation and derivative are used to express a non-linear system of equations

$$\left(\frac{\partial (R^e_{solid})_{ai}}{\partial u_{bj}} \right) \Bigg|^{n+1}_{kiter} \left((u_{bj})|^{n+1}_{kiter+1} - (u_{bj})|^{n+1}_{kiter} \right) = -(R_{solid})_{ai}|^{n+1}_{kiter},$$

which are solved via the solver package.

Implicit Dynamics Time Integration (Newmark Method)

For implicit dynamic time integration, we use an implementation of the classical Newmark method. This update method can be posed in terms of a simple SDOF spring/dashpot/mass model. In the following, M represents the mass, C represent the damping of the dashpot, K represents the spring stiffness, and F represents some external load.

$$Ma^{n+1} + Cv^{n+1} + Ku^{n+1} = F_{n+1},$$

and a series of update equations for the velocity and displacement at a point:

$$\begin{aligned} u^{n+1} &= u^n + v^{n+1/2} \Delta t, \\ u^{n+1} &= u^n + \left(v^n + \frac{1}{2} \left[(1 - 2\beta) a^n + 2\beta a^{n+1} \right] \Delta t \right) \Delta t, \\ v^{n+1} &= v^n + \left[(1 - \gamma) a^n + \gamma a^{n+1} \right] \Delta t. \end{aligned}$$

As intermediate quantities we can form an estimate (predictor) for the end of step displacement and midstep velocity by assuming zero end-of-step acceleration.

$$
\begin{aligned}
\tilde{u}^{n+1} &= u^n + \left(v^n + \frac{1}{2}(1-2\beta)a^n \Delta t \right) \Delta t = u^n + \hat{\hat{u}} \\
\tilde{v}^{n+1} &= v^n + (1-\gamma)a^n \Delta t = v^n + \hat{\hat{v}}
\end{aligned}
$$

This gives the end of step displacement and velocity in terms of the predictor with a correction for the end step acceleration.

$$
\begin{aligned}
u^{n+1} &= \tilde{u}^{n+1} + \beta a^{n+1} \Delta t^2 \\
v^{n+1} &= \tilde{v}^{n+1} + \gamma a^{n+1} \Delta t
\end{aligned}
$$

The acceleration and velocity may now be expressed in terms of displacement, and ultimately in terms of the incremental displacement.

$$
\begin{aligned}
a^{n+1} &= \frac{1}{\beta \Delta t^2} \left(u^{n+1} - \tilde{u}^{n+1} \right) = \frac{1}{\beta \Delta t^2} \left(\hat{u} - \hat{\hat{u}} \right) \\
v^{n+1} &= \tilde{v}^{n+1} + \frac{\gamma}{\beta \Delta t} \left(u^{n+1} - \tilde{u}^{n+1} \right) = \tilde{v}^{n+1} + \frac{\gamma}{\beta \Delta t} \left(\hat{u} - \hat{\hat{u}} \right)
\end{aligned}
$$

plugging these into equation of motion for the SDOF system gives:

$$
M \left(\frac{1}{\beta \Delta t^2} \left(\hat{u} - \hat{\hat{u}} \right) \right) + C \left(\tilde{v}^{n+1} + \frac{\gamma}{\beta \Delta t} \left(\hat{u} - \hat{\hat{u}} \right) \right) + K u^{n+1} = F_{n+1}
$$

Finally, we assume Rayliegh damping for the dashpot.

$$
C = a_{mass} M + a_{stiff} K
$$

Of course we know that we intend to model a system of equations with many DOF. Thus the representation for the mass, spring and dashpot can be replaced by our finite element discretized equation of motion. We may express the system in context of a nonlinear residual problem

$$
\begin{aligned}
(R^e_{solid})_{ai} &= \int\limits_{\Gamma_t^e} \Phi_a t_i dA \\
&- \int\limits_{\Omega^e} \Phi_{a,j} \left(T_{ij}^{n+1} + a_{stiff} \left(\frac{\partial T_{ij}^{n+1}}{\partial \hat{u}_{bk}} \right)_{elastic} \left(\tilde{v}_{bk}^{n+1} + \frac{\gamma}{\beta \Delta t} \left(\hat{u}_{bk} - \hat{\hat{u}}_{bk} \right) \right) \right) dV \\
&+ \int\limits_{\Omega^e} \Phi_a \rho \left(b_i - \Phi_b \left(a_{mass} \left(\tilde{v}_{bi}^{n+1} + \frac{\gamma}{\beta \Delta t} \left(\hat{u}_{bi} - \hat{\hat{u}}_{bi} \right) \right) + \frac{1}{\beta \Delta t^2} \left(\hat{u}_{bi} - \hat{\hat{u}}_{bi} \right) \right) \right) dV, \\
\frac{\partial (R^e_{solid})_{ai}}{\partial \hat{u}_{bj}} &= - \int\limits_{\Omega^e} \Phi_{a,k} \left(\frac{\partial T_{ik}^{n+1}}{\partial \hat{u}_{bj}} + a_{stiff} \frac{\gamma}{\beta \Delta t} \left(\frac{\partial T_{ik}^{n+1}}{\partial \hat{u}_{bj}} \right)_{elastic} \right) dV \\
&- \left(\frac{\gamma a_{mass}}{\beta \Delta t} + \frac{1}{\beta \Delta t^2} \right) \int\limits_{\Omega^e} \rho \Phi_a \Phi_c \frac{\partial \hat{u}_{ci}}{\partial \hat{u}_{bj}} dV.
\end{aligned}
$$

Again, the expression for the residual equation and derivative are used to express a non-linear system of equations

$$
\left. \left(\frac{\partial (R^e_{solid})_{ai}}{\partial u_{bj}} \right) \right|^{n+1}_{kiter} \left((u_{bj})|^{n+1}_{kiter+1} - (u_{bj})|^{n+1}_{kiter} \right) = -(R_{solid})_{ai}|^{n+1}_{kiter},
$$

which are solved via the solver package. Note that the derivatives involving u and $\hat{u}$ are interchangable, as are differences between the non-linear iterations.

Explicit Dynamics Time Integration (Special Implementation of Newmark Method with gamma=0.5, beta=0)

For the Newmark Method, if gamma=0.5, beta=0, and the inertial term contains a diagonalized "mass matrix", the update equations may be carried out without the solution of a system of equations. In this case, the update equations simplify to a non-iterative update algorithm.

First the mid-step velocity and end-of-step displacements are calculated through the update equations

$$v^{n+1/2} = v^n + a^n \left(\frac{\Delta t}{2} \right), \text{ and}$$
$$u^{n+1} = u^n + v^{n+1/2} \Delta t.$$

Then the residual equation/s are calculated, and acceleration at the end-of-step is calculated via

$$\left(M + \frac{\Delta t}{2} C \right) a^{n+1} = F_{n+1} - C v^{n+1/2} - K u^{n+1}.$$

Note that the mass matrix must be diagonal, and damping term may not include the stiffness based damping coefficient for this method, otherwise the above equation will require a system solve. Finally, the end-of-step velocities are calculated from the end of step acceleration:

$$v^{n+1} = v^{n+1/2} + a^{n+1} \left(\frac{\Delta t}{2} \right).$$

Note that the velocities may be stored at the midstep, resulting one less kinematic update. This approach is typically referred to as the "Leapfrog" method. However, in GEOS we do not offer this option since it can cause some confusion that results from the storage of state at different points in time.

Parameters

In the preceding XML block, The *SolidMechanics_LagrangianFEM* is specified by the title of the subblock of the *Solvers* block. The following attributes are supported in the input block for *SolidMechanics_LagrangianFEM*:

Name	Type	Default	Description
cflFactor	real64	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
contactRelationName	string	NOCONTACT	Name of contact relation to enforce constraints on fracture boundary.
discretization	string	required	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
initialDt	real64	1e+99	Initial time-step value required by the solver to the event manager.
logLevel	integer	0	Log level
massDamping	real64	0	Value of mass based damping coefficient.
maxNumResolves	integer	10	Value to indicate how many resolves may be executed after some other event is executed. For example, if a SurfaceGenerator is specified, it will be executed after the mechanics solve. However if a new surface is generated, then the mechanics solve must be executed again due to the change in topology.
name	string	required	A name is required for any non-unique nodes
newmarkBeta	real64	0.25	Value of β in the Newmark Method for Implicit Dynamic time integration option. This should be pow(newmarkGamma+0.5,2.0)/4.0 unless you know what you are doing.
newmarkGamma	real64	0.5	Value of γ in the Newmark Method for Implicit Dynamic time integration option.
stiffnessDamping	real64	0	Value of stiffness based damping coefficient.
strainTheory	integer	0	

The following data are allocated and used by the solver:

Name	Type	Description
maxForce	real64	The maximum force contribution in the problem domain.
maxStableDt	real64	Value of the Maximum Stable Timestep for this solver.
meshTargets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >	MeshBody/Region combinations that the solver will be applied to.
LinearSolverParameters	node	*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node	*Datastructure: NonlinearSolverParameters*
SolverStatistics	node	*Datastructure: SolverStatistics*

Example

An example of a valid XML block is given here:

```
<Solvers>
  <SolidMechanics_LagrangianFEM
    name="lagsolve"
    strainTheory="1"
    cflFactor="0.25"
    discretization="FE1"
    targetRegions="{ Region2 }"
    />
</Solvers>
```

Contact Mechanics Solver

Governing Equations

GEOS contact solvers solve the the balance of linear momentum within a fractured solid, accounting for the continuity of stress across surfaces (i.e., fractures), i.e.

$$\nabla \cdot \sigma = 0$$

$$[[\sigma]] \cdot \mathbf{n} = 0$$

Where:

- σ is the stress tensor in the solid,
- $\mathbf{n}$ is the outward unit normal to the surface,
- $[[\sigma]]$ is the stress jump across the surface.

On each fracture surface, a no-interpenetration constraint is enforced. Additionally, tangential tractions can also be generated, which are modeled using a regularized Coulomb model to describe frictional sliding.

Solvers

There exist two broad classes of discretization methods that model fractures as lower dimensional entities (eg, 2D surfaces in a 3D domain): conforming grid methods and nonconforming (or embedded) methods. Both approaches have been implemented in GEOS in the following solvers:

Solid mechanics conforming fractures solver

Introduction

Theory

Under construction

Governing Equations

Under construction

Parameters

In the preceding XML block, The *LagrangianContactSolver* is specified by the title of the subblock of the *Solvers* block. The following attributes are supported in the input block for *LagrangianContactSolver*:

The following data are allocated and used by the solver:

Solid mechanics embedded fractures solver

Introduction

Discretization & soltuion strategy

The linear momentum balance equation is discretized using a low order finite element method. Moreover, to account for the influence of the fractures on the overall behavior, we utilize the enriched finite element method (EFEM) with a piece-wise constant enrichment. This method employs an element-local enrichment of the FE space using the concept of assumedenhanced strain [1-6].

Example

An example of a valid XML block is given here:

Parameters

In the preceding XML block, The *SolidMechanicsEmbeddedFractures* is specified by the title of the subblock of the *Solvers* block. Note that the *SolidMechanicsEmbeddedFractures* always relies on the existance of a The following attributes are supported in the input block for *SolidMechanicsEmbeddedFractures*:

Name	Type	Default	Description
cflFactor	real6	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
contactRelationName	string	require	Name of contact relation to enforce constraints on fracture boundary.
fractureRegionName	string	require	Name of the fracture region.
initialDt	real6	1e+9	Initial time-step value required by the solver to the event manager.
logLevel	integer	0	Log level
name	string	require	A name is required for any non-unique nodes
solidSolverName	string	require	Name of the solid mechanics solver in the rock matrix
targetRegions	string	require	Allowable regions that the solver may be applied to. Note that this does not indicate that the solver will be applied to these regions, only that allocation will occur such that the solver may be applied to these regions. The decision about what regions this solver will beapplied to rests in the EventManager.
useStaticCondensation	integer	0	Defines whether to use static condensation or not.
LinearSolverParameters	node	uniqu	*Element: LinearSolverParameters*
NonlinearSolverParameters	node	uniqu	*Element: NonlinearSolverParameters*

The following data are allocated and used by the solver:

Name	Type	Description
discretization	string	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
maxSta	real64	Value of the Maximum Stable Timestep for this solver.
meshTa gets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >	MeshBody/Region combinations that the solver will be applied to.
LinearSolverParameters	node	*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node	*Datastructure: NonlinearSolverParameters*
SolverStatistics	node	*Datastructure: SolverStatistics*

References

1. Simo JC, Rifai MS. A class of mixed assumed strain methods and the method of incompatible modes. *Int J Numer Methods Eng*. 1990;29(8):1595-1638. Available at: http://arxiv.org/abs/https://onlinelibrary.wiley.com/doi/pdf/10.1002/nme.1620290802.
2. Foster CD, Borja RI, Regueiro RA. Embedded strong discontinuity finite elements for fractured geomaterials with variable friction. *Int J Numer Methods Eng*. 2007;72(5):549-581. Available at: http://arxiv.org/abs/https://onlinelibrary.wiley.com/doi/pdf/10.1002/nme.2020.
3. Wells G, Sluys L. Three-dimensional embedded discontinuity model for brittle fracture. *Int J Solids Struct*. 2001;38(5):897-913. Available at: https://doi.org/10.1016/S0020-7683(00)00029-9.
4. Oliver J, Huespe AE, Sánchez PJ. A comparative study on finite elements for capturing strong discontinuities: E-fem vs x-fem. *Comput Methods Appl Mech Eng*. 2006;195(37-40):4732-4752. Available at: https://doi.org/10.1002/nme.4814.
5. Borja RI. Assumed enhanced strain and the extended finite element methods: a unification of concepts. *Comput Methods Appl Mech Eng*. 2008;197(33):2789-2803. Available at: https://doi.org/10.1016/j.cma.2008.01.019.
6. Wu J-Y. Unified analysis of enriched finite elements for modeling cohesive cracks. *Comput Methods Appl Mech Eng*. 2011;200(45-46):3031-3050. Available at: https://doi.org/10.1016/j.cma.2011.05.008.

Singlephase Flow Solver

Introduction

Here, we describe the single-phase flow solver. The role of this solver is to implement the fully implicit finite-volume discretization (mainly, accumulation and source terms, boundary conditions) of the equations governing compressible single-phase flow in porous media. This solver can be combined with the SinglePhaseWell class which handles the discrete multi-segment well model and provides source/sink terms for the fluid flow solver.

Theory

Governing Equations

This is a cell-centered Finite Volume solver for compressible single-phase flow in porous media. Fluid pressure as the primary solution variable. Darcy's law is used to calculate fluid velocity from pressure gradient. The solver currently only supports Dirichlet-type boundary conditions (BC) applied on cells or faces and Neumann no-flow type BC.

The following mass balance equation is solved in the domain:

$$\frac{\partial}{\partial t}(\phi\rho) + \boldsymbol{\nabla} \cdot (\rho \boldsymbol{u}) + q = 0,$$

where

$$\boldsymbol{u} = -\frac{1}{\mu}\boldsymbol{k}(\nabla p - \rho \boldsymbol{g})$$

and ϕ is porosity, ρ is fluid density, μ is fluid viscosity, $\boldsymbol{k}$ is the permeability tensor, $\boldsymbol{g}$ is the gravity vector, and q is the source function (currently not supported). The details on the computation of the density and the viscosity are given in *Compressible single phase fluid model*.

When the entire pore space is filled by a single phase, we can substitute the Darcy's law into the mass balance equation to obtain the single phase flow equation

$$\frac{\partial}{\partial t}(\phi\rho) - \boldsymbol{\nabla} \cdot \frac{\rho \boldsymbol{k}}{\mu}(\nabla p - \gamma \nabla z) + q = 0,$$

with $\gamma \nabla z = \rho \boldsymbol{g}$.

Discretization

Space Discretization

Let $\Omega \subset \mathbb{R}^n$, $n = 1, 2, 3$ be an open set defining the computational domain. We consider Ω meshed by element such that $\Omega = \cup_i V_i$ and integrate the single phase flow equation, described above, over each element V_i:

$$\int_{V_i} \frac{\partial}{\partial t}(\phi\rho) dV - \int_{V_i} \boldsymbol{\nabla} \cdot \frac{\rho \boldsymbol{k}}{\mu}(\nabla p - \gamma \nabla z) dV + \int_{V_i} q dV = 0.$$

Applying the divergence theorem to the second term leads to

$$\int_{V_i} \frac{\partial}{\partial t}(\phi\rho)_i - \oint_{S_i} \left(\frac{\rho \boldsymbol{k}}{\mu}(\nabla p - \gamma \nabla z) \right) \cdot \boldsymbol{n} dS + \int_{V_i} q dV = 0.$$

where S_i represents the surface area of the element V_i and $\boldsymbol{n}$ is a outward unit vector normal to the surface.

For the flux term, the (static) transmissibility is currently computed with a Two-Point Flux Approximation (TPFA) as described in *Finite Volume Discretization*.

The pressure-dependent mobility $\lambda = \frac{\rho}{\mu}$ at the interface is approximated using a first-order upwinding on the sign of the potential difference.

Time Discretization

Let $t_0 < t_1 < \cdots < t_N = T$ be a grid discretization of the time interval $[t_0, T]$, $t_0, T \in \mathbb{R}^+$. We use the backward Euler (fully implicit) method to integrate the single phase flow equation between two grid points t_n and t_{n+1}, $n < N$ to obtain the residual equation:

$$\int_{V_i} \frac{(\phi\rho)_i^{n+1} - (\phi\rho)_i^n}{\Delta t} - \oint_{S_i} \left(\frac{\rho \boldsymbol{k}}{\mu} (\nabla p - \gamma \nabla z) \right)^{n+1} \cdot \boldsymbol{n} dS + \int_{V_i} q^{n+1} dV = 0$$

where $\Delta t = t_{n+1} - t_n$ is the time-step. The expression of this residual equation and its derivative are used to form a linear system, which is solved via the solver package.

Parameters

The solver is enabled by adding a `<SinglePhaseFVM>` node in the Solvers section. Like any solver, time stepping is driven by events, see *Event Management*.

The following attributes are supported:

Name	Type	Default	Description
cflFactor	real6	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
discretization	string	require	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
initialDt	real6	1e+9	Initial time-step value required by the solver to the event manager.
isThermal	integer	0	Flag indicating whether the problem is thermal or not.
logLevel	integer	0	Log level
name	string	require	A name is required for any non-unique nodes
targetRegions	string	require	Allowable regions that the solver may be applied to. Note that this does not indicate that the solver will be applied to these regions, only that allocation will occur such that the solver may be applied to these regions. The decision about what regions this solver will beapplied to rests in the EventManager.
temperature	real6	0	Temperature
LinearSolverParameters	node	uniqu	*Element: LinearSolverParameters*
NonlinearSolverParameters	node	uniqu	*Element: NonlinearSolverParameters*

In particular:

- `discretization` must point to a Finite Volume flux approximation scheme defined in the Numerical Methods section of the input file (see *Finite Volume Discretization*)
- `fluidName` must point to a single phase fluid model defined in the Constitutive section of the input file (see *Constitutive Models*)
- `solidName` must point to a solid mechanics model defined in the Constitutive section of the input file (see *Constitutive Models*)
- `targetRegions` is used to specify the regions on which the solver is applied

Primary solution field label is `pressure`. Initial conditions must be prescribed on this field in every region, and boundary conditions must be prescribed on this field on cell or face sets of interest.

Example

```
<Solvers>
  <SinglePhaseFVM
    name="SinglePhaseFlow"
    logLevel="1"
    discretization="singlePhaseTPFA"
    targetRegions="{ mainRegion }">
    <NonlinearSolverParameters
      newtonTol="1.0e-6"
      newtonMaxIter="8"/>
    <LinearSolverParameters
      solverType="gmres"
      preconditionerType="amg"
      krylovTol="1.0e-10"/>
  </SinglePhaseFVM>
</Solvers>
```

We refer the reader to *this page* for a complete tutorial illustrating the use of this solver.

Compositional Multiphase Flow Solver

Introduction

This flow solver is in charge of implementing the finite-volume discretization (mainly, accumulation and flux terms, boundary conditions) of the equations governing compositional multiphase flow in porous media. The present solver can be combined with the *Compositional Multiphase Well Solver* which handles the discrete multi-segment well model and provides source/sink terms for the fluid flow solver.

Below, we first review the set of *Governing Equations*, followed by a discussion of the choice of *Primary Variables* used in the global variable formulation. Then we give an overview of the *Discretization* and, finally, we provide a list of the solver *Parameters* and an input *Example*.

Theory

Governing Equations

Mass Conservation Equations

Mass conservation for component c is expressed as:

$$\phi \frac{\partial}{\partial t} \left(\sum_\ell \rho_\ell \, y_{c\ell} \, S_\ell \right) + \nabla \cdot \left(\sum_\ell \rho_\ell \, y_{c\ell} \, \boldsymbol{u}_\ell \right) - \sum_\ell \rho_\ell \, y_{c\ell} \, q_\ell = 0,$$

where ϕ is the porosity of the medium, S_ℓ is the saturation of phase ℓ, $y_{c\ell}$ is the mass fraction of component c in phase ℓ, ρ_ℓ is the phase density, and t is time. We note that the formulation currently implemented in GEOS is isothermal.

Darcy's Law

Using the multiphase extension of Darcy's law, the phase velocity $\boldsymbol{u}_\ell$ is written as a function of the phase potential gradient $\nabla \Phi_\ell$:

$$\boldsymbol{u}_\ell := -\boldsymbol{k} \lambda_\ell \nabla \Phi_\ell = -\boldsymbol{k} \lambda_\ell \big(\nabla (p - P_{c,\ell}) - \rho_\ell g \nabla z \big).$$

In this equation, $\boldsymbol{k}$ is the rock permeability, $\lambda_\ell = k_{r\ell} / \mu_\ell$ is the phase mobility, defined as the phase relative permeability divided by the phase viscosity, p is the reference pressure, $P_{c,\ell}$ is the the capillary pressure, g is the gravitational acceleration, and z is depth. The evaluation of the relative permeabilities, capillary pressures, and viscosities is reviewed in the section about *Constitutive Models*.

Combining the mass conservation equations with Darcy's law yields a set of n_c equations written as:

$$\phi \frac{\partial}{\partial t} \bigg(\sum_\ell \rho_\ell \, y_{c\ell} \, S_\ell \bigg) - \nabla \cdot \boldsymbol{k} \bigg(\sum_\ell \rho_\ell \, y_{c\ell} \, \lambda_\ell \nabla \Phi_\ell \bigg) - \sum_\ell \rho_\ell \, y_{c\ell} \, q_\ell = 0.$$

Constraints and Thermodynamic Equilibrium

The volume constraint equation states that the pore space is always completely filled by the phases. The constraint can be expressed as:

$$\sum_\ell S_\ell = 1.$$

The system is closed by the following thermodynamic equilibrium constraints:

$$f_{c\ell} - f_{cm} = 0.$$

where $f_{c\ell}$ is the fugacity of component c in phase ℓ. The flash calculations performed to enforce the thermodynamical equilibrium are reviewed in the section about *Constitutive Models*.

To summarize, the compositional multiphase flow solver assembles a set of $n_c + 1$ equations in each element, i.e., n_c mass conservation equations and one volume constraint equation. A separate module discussed in the *Constitutive Models* is responsible for the enforcement of the thermodynamic equilibrium at each nonlinear iteration.

Number of equations	Equation type
n_c	Mass conservation equations
1	Volume constraint

Primary Variables

The variable formulation implemented in GEOS is a global variable formulation based on $n_c + 1$ primary variables, namely, one pressure, p, and n_c component densities, ρ_c. By default, we use molar component densities. A flag discussed in the section *Parameters* can be used to select mass component densities instead of molar component densities.

Number of primary variables	Variable type
1	Pressure
n_c	Component densities

Assembling the residual equations and calling the *Constitutive Models* requires computing the molar component fractions and saturations. This is done with the relationship:

$$z_c := \frac{\rho_c}{\rho_T},$$

where

$$\rho_T := \sum_c \rho_c.$$

These secondary variables are used as input to the flash calculations. After the flash calculations, the saturations are computed as:

$$S_\ell := \nu_\ell \frac{\rho_T}{\rho_\ell},$$

where ν_ℓ is the global mole fraction of phase ℓ and ρ_ℓ is the molar density of phase ℓ. These steps also involve computing the derivatives of the component fractions and saturations with respect to the pressure and component densities.

Discretization

Spatial Discretization

The governing equations are discretized using standard cell-centered finite-volume discretization.

In the approximation of the flux term at the interface between two control volumes, the calculation of the pressure stencil is general and will ultimately support a Multi-Point Flux Approximation (MPFA) approach. The current implementation of the transmissibility calculation is reviewed in the section about *Numerical Methods*.

The approximation of the dynamic transport coefficients multiplying the discrete potential difference (e.g., the phase mobilities) is performed with a first-order phase-per-phase single-point upwinding based on the sign of the phase potential difference at the interface.

Temporal Discretization

The compositional multiphase solver uses a fully implicit (backward Euler) temporal discretization.

Solution Strategy

The nonlinear solution strategy is based on Newton's method. At each Newton iteration, the solver assembles a residual vector, R, collecting the n_c discrete mass conservation equations and the volume constraint for all the control volumes.

Parameters

The following attributes are supported:

Name	Type	Default	Description
allowLocalCompDensityChopping	integer	1	Flag indicating whether local (cell-wise) chopping of negative compositions is allowed
cflFactor	real64	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
discretization	string	required	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
initialDt	real64	1e+99	Initial time-step value required by the solver to the event manager.
isThermal	integer	0	Flag indicating whether the problem is thermal or not.
logLevel	integer	0	Log level
maxCompFractionChange	real64	0.5	Maximum (absolute) change in a component fraction in a Newton iteration
maxRelativePressureChange	real64	0.5	Maximum (relative) change in pressure in a Newton iteration (expected value between 0 and 1)
maxRelativeTemperatureChange	real64	0.5	Maximum (relative) change in temperature in a Newton iteration (expected value between 0 and 1)
name	string	required	A name is required for any non-unique nodes
scalingType	geos_CompositionalMultipl	Global	Solution scaling type.Valid options: * Global * Local
solutionChangeScalingFactor	real64	0.5	Damping factor for solution change targets
targetPhaseVolFractionChangeInTimeStep	real64	0.2	Target (absolute) change in phase volume fraction in a time step

Example

```
<Solvers>
  <CompositionalMultiphaseFVM
    name="compflow"
    logLevel="1"
    discretization="fluidTPFA"
    targetRelativePressureChangeInTimeStep="1"
    targetPhaseVolFractionChangeInTimeStep="1"
    targetRegions="{ Channel }"
    temperature="300">
    <NonlinearSolverParameters
      newtonTol="1.0e-10"
      newtonMaxIter="40"/>
    <LinearSolverParameters
      directParallel="0"/>
  </CompositionalMultiphaseFVM>
</Solvers>
```

We refer the reader to *Multiphase Flow* for a complete tutorial illustrating the use of this solver.

Compositional Multiphase Well Solver

Introduction

Here, we present a description of the well solvers. These solvers are designed to be coupled with the flow solvers. Their specific task is to implement the multi-segment well discretization using the fluid model used in the corresponding flow solver – i.e., either single-phase flow or compositional multiphase flow. In particular, the perforation terms computed by the well solvers are a source/sink term for the discrete reservoir equations assembled by the flow solvers.

In the present description, we focus on the compositional multiphase well solver. The structure of the single-phase well solver is analogous and will not be described here for brevity.

Theory

Here, we give an overview of the well formulation implemented in GEOS. We review the set of *Discrete Equations*, and then we describe the *Primary variables* used the well solvers.

Discrete Equations

We assume that the well is discretized into segments denoted by the index i.

Mass Conservation

In well segment i, mass conservation for component c reads:

$$z^{upw}_{c,(i-1,i)} q_{(i-1,i)} - z^{upw}_{c,(i,i+1)} q_{(i,i+1)} + q^{perf}_{c,i} = 0,$$

where we have neglected the accumulation term. In the previous equation, $z^{upw}_{c,(i,j)}$ is the upwinded mass fraction of component c at the interface between segments i and j, $q_{(i,j)}$ is the total mass flux between segments i and j, and $q^{perf}_{c,i}$ is the source/sink term generated by the perforations – i.e., the connections between the well and the reservoir. The upwinded mass fractions are computed as:

$$z^{upw}_{c,(i,j)} = \begin{cases} z_{c,i} & \text{if } q_{(i,j)} > 0 \\ z_{c,j} & \text{otherwise.} \end{cases}$$

The perforation terms are obtained with:

$$q^{perf}_{c,i} = \begin{cases} WI z_{c,i} \rho_{m,i} \lambda^{res}_T \Delta\Phi & \text{if the well is upstream (i.e., } \Delta\Phi > 0) \\ WI x^{res}_{c,\ell} \rho^{res}_{\ell} \lambda^{res}_{\ell} \Delta\Phi & \text{otherwise,} \end{cases}$$

where $\Delta\Phi = p_i - p^{res} + \rho_{m,i} g \Delta d_{i,perf}$ is the potential difference between the segment center and the reservoir center. In the expression of the potential difference, the mixture density is computed as $\rho_{m,i} = \sum_{\ell} S_{\ell,i} \rho_{\ell,i}$. The well index, WI, is currently an input of the simulator. The superscript res means that the variable is evaluated at the center of the reservoir element.

Volume Constraint Equation

As in the *Compositional Multiphase Flow Solver*, the system is closed with a volume constraint equation.

Pressure Relations

In the current implementation of the well solver, we assume a hydrostatic equilibrium:

$$p_{i+1} - p_i = \rho_{m,(i,i+1)} g \Delta d_{i,i+1},$$

where $\rho_{m,(i,i+1)}$ is the arithmetic average of the mixture densities evaluated in segments i and $i+1$. Pressure drop components due to friction and acceleration are not implemented at the moment.

Pressure and Rate Controls

The well solver supports two types of control, namely, pressure control and rate control.

If pressure control is chosen, we add the following constraint for the pressure of the top segment of the well:

$$p_0 - p^{target} = 0$$

In this case, we check that at each iteration of the Newton solver, the rate at the top of the first segment is smaller than the maximum rate specified by the user. If this is not the case, we switch to rate control.

If rate control is used, we add the following constraint for the rate at the top of the first segment, denoted by $q_{(-1,0)}$:

$$q_{(-1,0)} - q^{target} = 0$$

For an injector well, if the pressure at the top segment becomes larger than the target pressure specified by the user, then we switch to pressure control. Similarly for a producer well, if the pressure at the top segment becomes smaller than the target pressure specified by the user, then we switch to pressure control.

To summarize, the compositional multiphase flow solver assembles a set of $n_c + 2$ equations, i.e., n_c mass conservation equations and 1 volume constraint equation in each segment, plus 1 pressure relation at the interface between a segment and the next segment in the direction of the well head. For the top segment, the pressure relation is replaced with the control equation.

Number of equations	Equation type
n_c	Mass conservation equations
1	Pressure relation or control equation
1	Volume constraint

Primary variables

The well variable formulation is the same as that of the *Compositional Multiphase Flow Solver*. In a well segment, in addition to the $n_c + 1$ primary variables of the *Compositional Multiphase Flow Solver*, namely, one pressure, p, and n_c component densities, ρ_c, we also treat the total mass flux at the interface with the next segment, denoted by q, as a primary variable.

Number of primary variables	Variable type
1	Pressure
1	Total mass flux at the interface with next segment
n_c	Component densities

Parameters

The following attributes are supported:

Name	Type	Default	Description
allowLocalCompDensityChopping	integer	1	Flag indicating whether local (cell-wise) chopping of negative compositions is allowed
cflFactor	real6	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
initialDt	real6	1e+9	Initial time-step value required by the solver to the event manager.
logLevel	integer	0	Log level
maxCompFractionChange	real6	1	Maximum (absolute) change in a component fraction between two Newton iterations
maxRelativePressureChange	real6	1	Maximum (relative) change in pressure between two Newton iterations (recommended with rate control)
name	string	require	A name is required for any non-unique nodes
targetRegions	string	require	Allowable regions that the solver may be applied to. Note that this does not indicate that the solver will be applied to these regions, only that allocation will occur such that the solver may be applied to these regions. The decision about what regions this solver will beapplied to rests in the EventManager.
useMass	integer	0	Use mass formulation instead of molar
LinearSolverParameters	node	uniqu	*Element: LinearSolverParameters*
NonlinearSolverParameters	node	uniqu	*Element: NonlinearSolverParameters*
WellControls	node		*Element: WellControls*

Example

```
<CompositionalMultiphaseWell
  name="compositionalMultiphaseWell"
  logLevel="1"
  targetRegions="{ wellRegion1, wellRegion2, wellRegion3 }">
  <WellControls
    name="wellControls1"
    type="producer"
    control="BHP"
    referenceElevation="0.5"
    targetBHP="4e6"
```

(continues on next page)

(continued from previous page)

```
        targetPhaseRate="1e-3"
        targetPhaseName="oil"/>
      <WellControls
        name="wellControls2"
        type="producer"
        control="phaseVolRate"
        referenceElevation="0.5"
        targetBHP="2e6"
        targetPhaseRate="2.5e-7"
        targetPhaseName="oil"/>
      <WellControls
        name="wellControls3"
        type="injector"
        control="totalVolRate"
        referenceElevation="0.5"
        targetBHP="4e7"
        targetTotalRate="5e-7"
        injectionTemperature="297.15"
        injectionStream="{ 0.1, 0.1, 0.1, 0.7 }"/>
    </CompositionalMultiphaseWell>
```

Poromechanics Solver

Introduction

This section describes the use of the poroelasticity models implemented in GEOS.

Theory

Governing Equations

In our model, the geomechanics (elasticity) equation is expressed in terms of the total stress σ:

$$\nabla \sigma + \rho_b \mathbf{g} = 0$$

where it relates to effective stress $\sigma\prime$ and pore pressure p through Biot's coefficient b:

$$\sigma = \sigma\prime - bp\mathbf{I}$$

The fluid mass conservation equation is expressed in terms of pore pressure and volumetric (mean) total stress:

$$\left(\frac{1}{M} + \frac{b^2}{K_{dr}} \right) \frac{\partial p}{\partial t} + \frac{b}{K_{dr}} \frac{\partial \sigma_v}{\partial t} + \nabla \cdot \mathbf{v}_f = f$$

where M is the Biot's modulus and K_{dr} is the drained bulk modulus.

Unlike the conventional reservoir model that uses Lagranges porosity, in the coupled geomechanics and flow model, Eulers porosity ϕ is adopted so the porosity variation is derived as:

$$\partial \phi = \left(\frac{b - \phi}{K_s} \right) \partial p + (b - \phi) \partial \epsilon_v$$

where K_s is the bulk modulus of the solid grain and ϵ_v is the volumetric strain.

Parameters

The poroelasticity model is implemented as a main solver listed in `<Solvers>` block of the input XML file that calls both SolidMechanicsLagrangianSSLE and SinglePhaseFlow solvers. In the main solver, it requires the specification of solidSolverName, flowSolverName, and couplingTypeOption.

The following attributes are supported:

Name	Type	Default	Description
cflFactor	real6	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
flowSolverName	string	require	Name of the flow solver used by the coupled solver
initialDt	real6	1e+9	Initial time-step value required by the solver to the event manager.
isThermal	integer	0	Flag indicating whether the problem is thermal or not. Set isThermal="1" to enable the thermal coupling
logLevel	integer	0	Log level
name	string	require	A name is required for any non-unique nodes
solidSolverName	string	require	Name of the solid solver used by the coupled solver
targetRegions	string	require	Allowable regions that the solver may be applied to. Note that this does not indicate that the solver will be applied to these regions, only that allocation will occur such that the solver may be applied to these regions. The decision about what regions this solver will beapplied to rests in the EventManager.
LinearSolverParameters	node	uniqu	*Element: LinearSolverParameters*
NonlinearSolverParameters	node	uniqu	*Element: NonlinearSolverParameters*

- `couplingTypeOption`: defines the coupling scheme.

The solid constitutive model used here is PoroLinearElasticIsotropic, which derives from ElasticIsotropic and includes an additional parameter: Biot's coefficient. The fluid constitutive model is the same as SinglePhaseFlow solver. For the parameter setup of each individual solver, please refer to the guideline of the specific solver.

An example of a valid XML block for the constitutive model is given here:

```
<Constitutive>
  <PorousElasticIsotropic
    name="porousRock"
    solidModelName="skeleton"
```

(continues on next page)

(continued from previous page)

```
      porosityModelName="skeletonPorosity"
      permeabilityModelName="skeletonPerm"/>

    <ElasticIsotropic
      name="skeleton"
      defaultDensity="0"
      defaultYoungModulus="1.0e4"
      defaultPoissonRatio="0.2"/>

    <CompressibleSinglePhaseFluid
      name="fluid"
      defaultDensity="1"
      defaultViscosity="1.0"
      referencePressure="0.0"
      referenceDensity="1"
      compressibility="0.0e0"
      referenceViscosity="1"
      viscosibility="0.0"/>

    <BiotPorosity
      name="skeletonPorosity"
      grainBulkModulus="1.0e27"
      defaultReferencePorosity="0.3"/>

    <ConstantPermeability
      name="skeletonPerm"
      permeabilityComponents="{ 1.0e-4, 1.0e-4, 1.0e-4 }"/>
  </Constitutive>
```

Example

```
    <SinglePhasePoromechanics
      name="PoroelasticitySolver"
      solidSolverName="LinearElasticitySolver"
      flowSolverName="SinglePhaseFlowSolver"
      logLevel="1"
      targetRegions="{ Domain }">
      <LinearSolverParameters
        solverType="gmres"
        preconditionerType="mgr"/>
    </SinglePhasePoromechanics>

    <SolidMechanicsLagrangianSSLE
      name="LinearElasticitySolver"
      timeIntegrationOption="QuasiStatic"
      logLevel="1"
      discretization="FE1"
      targetRegions="{ Domain }"/>

    <SinglePhaseFVM
```

(continues on next page)

(continued from previous page)

```
    name="SinglePhaseFlowSolver"
    logLevel="1"
    discretization="singlePhaseTPFA"
    targetRegions="{ Domain }"/>
</Solvers>
```

Proppant Transport Solver

Introduction

The *ProppantTransport* solver applies the finite volume method to solve the equations of proppant transport in hydraulic fractures. The behavior of proppant transport is described by a continuum formulation. Here we briefly outline the usage, governing equations and numerical implementation of the proppant transport model in GEOS.

Theory

The following mass balance and constitutive equations are solved inside fractures,

Proppant-fluid Slurry Flow

$$\frac{\partial}{\partial t}(\rho_m) + \boldsymbol{\nabla} \cdot (\rho_m \boldsymbol{u_m}) = 0,$$

where the proppant-fluid mixture velocity $\boldsymbol{u_m}$ is approximated by the Darcy's law as,

$$\boldsymbol{u}_m = -\frac{K_f}{\mu_m}(\nabla p - \rho_m \boldsymbol{g}),$$

and p is pressure, ρ_m and μ_m are density and viscosity of the mixed fluid , respectively, and $\boldsymbol{g}$ is the gravity vector. The fracture permeability K_f is determined based on fracture aperture a as

$$K_f = \frac{a^2}{12}$$

Proppant Transport

$$\frac{\partial}{\partial t}(c) + \boldsymbol{\nabla} \cdot (c\boldsymbol{u}_p) = 0,$$

in which c and $\boldsymbol{u}_p$ represent the volume fraction and velocity of the proppant particles.

Multi-component Fluid Transport

$$\frac{\partial}{\partial t}[\rho_i \omega_i (1-c)] + \boldsymbol{\nabla} \cdot [\rho_i \omega_i (1-c) \boldsymbol{u}_f] = 0.$$

Here $\boldsymbol{u}_f$ represents the carrying fluid velocity. ρ_i and ω_i denote the density and concentration of *i-th* component in fluid, respectively. The fluid density ρ_f can now be readily written as

$$\rho_f = \sum_{i=1}^{N_c} \rho_i \omega_i,$$

where N_c is the number of components in fluid. Similarly, the fluid viscosity μ_f can be calculated by the mass fraction weighted average of the component viscosities.

The density and velocity of the slurry fluid are further expressed as,

$$\rho_m = (1-c)\rho_f + c\rho_p,$$

and

$$\rho_m \boldsymbol{u}_m = (1-c)\rho_f \boldsymbol{u}_f + c\rho_p \boldsymbol{u}_p,$$

in which ρ_f and $\boldsymbol{u}_f$ are the density and velocity of the carrying fluid, and ρ_p is the density of the proppant particles.

Proppant Slip Velocity

The proppant particle and carrying fluid velocities are related by the slip velocity $\boldsymbol{u}_{slip}$,

$$\boldsymbol{u}_{slip} = \boldsymbol{u}_p - \boldsymbol{u}_f.$$

The slip velocity between the proppant and carrying fluid includes gravitational and collisional components, which take account of particle settling and collision effects, respectively.

The gravitational component of the slip velocity $\boldsymbol{u}_{slipG}$ is written as a form as

$$\boldsymbol{u}_{slipG} = F(c)\boldsymbol{u}_{settling},$$

where $\boldsymbol{u}_{settling}$ is the settling velocity for a single particle, d_p is the particle diameter, and $F(c)$ is the correction factor to the particle settling velocity in order to account for hindered settling effects as a result of particle-particle interactions,

$$F(c) = e^{-\lambda_s c},$$

with the hindered settling coefficient λ_s as an empirical constant set to 5.9 by default (Barree & Conway, 1995).

The settling velocity for a single particle, $\boldsymbol{u}_{settling}$, is calculated based on the Stokes drag law by default,

$$\boldsymbol{u}_{settling} = (\rho_p - \rho_f)\frac{{d_p}^2}{18\mu_f}\boldsymbol{g}.$$

Single-particle settling under intermediate Reynolds-number and turbulent flow conditions can also be described respectively by the Allen's equation (Barree & Conway, 1995),

$$\boldsymbol{u}_{settling} = 0.2 d_p^{1.18}\left[\frac{g(\rho_p - \rho_f)}{\rho_f}\right]^{0.72}\left(\frac{\rho_f}{\mu_f}\right)^{0.45}\boldsymbol{e},$$

and Newton's equation(Barree & Conway, 1995),

$$\boldsymbol{u}_{settling} = 1.74{d_p}^{0.5} \left[\frac{g(\rho_p - \rho_f)}{\rho_f} \right]^{0.5} \boldsymbol{e}.$$

$\boldsymbol{e}$ is the unit gravity vector and d_p is the particle diameter.

The collisional component of the slip velocity is modeled by defining λ, the ratio of the particle velocity to the volume averaged mixture velocity as a function of the proppant concentration. From this the particle slip velocity in horizontal direction is related to the mixed fluid velocity by,

$$\boldsymbol{u}_{slipH} = \frac{\lambda - 1}{1 - c} \boldsymbol{v}_m$$

with $\boldsymbol{v}_m$ denoting volume averaged mixture velocity. We use a simple expression of λ proposed by Barree & Conway (1995) to correct the particle slip velocity in horizontal direction,

$$\lambda = \left[\alpha - |c - c_{slip}|^{\beta} \right]$$

where α and β are empirical constants, c_{slip} is the volume fraction exhibiting the greatest particle slip. By default the model parameters are set to the values given in (Barree & Conway, 1995): $\alpha = 1.27$, $c_{slip} = 0.1$ and $\beta = 1.5$. This model can be extended to account for the transition to the particle pack as the proppant concentration approaches the jamming transition.

Proppant Bed Build-up and Load Transport

In addition to suspended particle flow the GEOS has the option to model proppant settling into an immobile bed at the bottom of the fracture. As the proppant cannot settle further down the proppant bed starts to form and develop at the element that is either at the bottom of the fracture or has an underlying element already filled with particles. Such an "inter-facial" element is divided into proppant flow and immobile bed regions based on the proppant-pack height.

Although proppant becomes immobile fluid can continue to flow through the settled proppant pack. The pack permeability *K* is defined based on the Kozeny-Carmen relationship:

$$K = \frac{(sd_p)^2}{180} \frac{\phi^3}{(1 - \phi)^2}$$

and

$$\phi = 1 - c_s$$

where ϕ is the porosity of particle pack and c_s is the saturation or maximum fraction for proppant packing, s is the sphericity and d_p is the particle diameter.

The growth of the settled pack in an "inter-facial" element is controlled by the interplay between proppant gravitational settling and shear-force induced lifting as (Hu et al., 2018),

$$\frac{dH}{dt} = \frac{c u_{settling} F(c)}{c_s} - \frac{Q_{lift}}{A c_s},$$

where H, t, c_s, Q_{lift}, and A represent the height of the proppant bed, time, saturation or maximum proppant concnetration in the proppant bed, proppant-bed load (wash-out) flux, and cross-sectional area, respectively.

The rate of proppant bed load transport (or wash out) due to shear force is calculated by the correlation proposed by Wiberg and Smith (1989) and McClure (2018),

$$Q_{lift} = a \left(d_p \sqrt{\frac{g d_p (\rho_p - \rho_f)}{\rho_f}} \right) (9.64 N_{sh}^{0.166})(N_{sh} - N_{sh,c})^{1.5}.$$

a is fracture aperture, and N_{sh} is the Shields number measuring the relative importance of the shear force to the gravitational force on a particle of sediment (Miller et al., 1977; Biot & Medlin, 1985; McClure, 2018) as

$$N_{sh} = \frac{\tau}{d_p g(\rho_p - \rho_f)},$$

and

$$\tau = 0.125 f \rho_f u_m^2$$

where τ is the shear stress acting on the top of the proppant bed and f is the Darcy friction coefficient. $N_{sh,c}$ is the critical Shields number for the onset of bed load transport.

Proppant Bridging and Screenout

Proppant bridging occurs when proppant particle size is close to or larger than fracture aperture. The aperture at which bridging occurs, h_b, is defined simply by

$$h_b = \lambda_b d_p,$$

in which λ_b is the bridging factor.

Slurry Fluid Viscosity

The viscosity of the bulk fluid, μ_m, is calculated as a function of proppant concentration as (Keck et al., 1992),

$$\mu_m = \mu_f \left[1 + 1.25 \left(\frac{c}{1 - c/c_s} \right) \right]^2.$$

Note that continued model development and improvement are underway and additional empirical correlations or functions will be added to support the above calculations.

Spatial Discretization

The above governing equations are discretized using a cell-centered two-point flux approximation (TPFA) finite volume method. We use an upwind scheme to approximate proppant and component transport across cell interfaces.

Solution Strategy

The discretized non-linear slurry flow and proppant/component transport equations at each time step are separately solved by the Newton-Raphson method. The coupling between them is achieved by a time-marching sequential (operator-splitting) solution approach.

Parameters

The solver is enabled by adding a `<ProppantTransport>` node and a `<SurfaceGenerator>` node in the Solvers section. Like any solver, time stepping is driven by events, see *Event Management*.

The following attributes are supported:

Name	Type	Default	Description
bridgingFactor	real6	0	Bridging factor used for bridging/screen-out calculation
cflFactor	real6	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
criticalShieldsNumber	real6	0	Critical Shields number
discretization	string	require	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
frictionCoefficient	real6	0.03	Friction coefficient
initialDt	real6	1e+9	Initial time-step value required by the solver to the event manager.
isThermal	integer	0	Flag indicating whether the problem is thermal or not.
logLevel	integer	0	Log level
maxProppantConcentration	real6	0.6	Maximum proppant concentration
name	string	require	A name is required for any non-unique nodes
proppantDensity	real6	2500	Proppant density
proppantDiameter	real6	0.000	Proppant diameter
targetRegions	string	require	Allowable regions that the solver may be applied to. Note that this does not indicate that the solver will be applied to these regions, only that allocation will occur such that the solver may be applied to these regions. The decision about what regions this solver will beapplied to rests in the EventManager.
updateProppantPacking	integer	0	Flag that enables/disables proppant-packing update
LinearSolverParameters	node	uniqu	*Element: LinearSolverParameters*
NonlinearSolverParameters	node	uniqu	*Element: NonlinearSolverParameters*

In particular:

- discretization must point to a Finite Volume flux approximation scheme defined in the Numerical Methods section of the input file (see *Finite Volume Discretization*)
- proppantName must point to a particle fluid model defined in the Constitutive section of the input file (see *Constitutive Models*)
- fluidName must point to a slurry fluid model defined in the Constitutive section of the input file (see *Constitutive Models*)
- solidName must point to a solid mechanics model defined in the Constitutive section of the input file (see *Constitutive Models*)
- targetRegions attribute is currently not supported, the solver is always applied to all regions.

Primary solution field labels are proppantConcentration and pressure. Initial conditions must be prescribed on these field in every region, and boundary conditions must be prescribed on these fields on cell or face sets of interest. For static (non-propagating) fracture problems, the fields ruptureState and elementAperture should be provided in the initial conditions.

In addition, the solver declares a scalar field named referencePorosity and a vector field named permeability, that contains principal values of the symmetric rank-2 permeability tensor (tensor axis are assumed aligned with the global coordinate system). These fields must be populated via *Element: FieldSpecification* section and permeability should be supplied as the value of coefficientName attribute of the flux approximation scheme used.

Example

First, we specify the proppant transport solver itself and apply it to the fracture region:

```
    <ProppantTransport
      name="ProppantTransport"
      logLevel="1"
      discretization="singlePhaseTPFA"
      targetRegions="{ Fracture }">
      <NonlinearSolverParameters
        newtonTol="1.0e-8"
        newtonMaxIter="8"
        lineSearchAction="None"/>
      <LinearSolverParameters
        directParallel="0"/>
    </ProppantTransport>
```

Then, we specify a compatible flow solver (currently a specialized SinglePhaseProppantFVM solver must be used):

```
    <SinglePhaseProppantFVM
      name="SinglePhaseFVM"
      logLevel="1"
      discretization="singlePhaseTPFA"
      targetRegions="{ Fracture }">
      <NonlinearSolverParameters
        newtonTol="1.0e-8"
        newtonMaxIter="8"
        lineSearchAction="None"/>
      <LinearSolverParameters
        solverType="gmres"
```

(continues on next page)

(continued from previous page)

```
      krylovTol="1.0e-12"/>
  </SinglePhaseProppantFVM>
```

Finally, we couple them through a coupled solver that references the two above:

```
  <FlowProppantTransport
    name="FlowProppantTransport"
    proppantSolverName="ProppantTransport"
    flowSolverName="SinglePhaseFVM"
    targetRegions="{ Fracture }"
    logLevel="1">
    <NonlinearSolverParameters
      newtonMaxIter="8"
      lineSearchAction="None"
      couplingType="Sequential"/>
  </FlowProppantTransport>
```

References

- R. D. Barree & M. W. Conway. "Experimental and numerical modeling of convective proppant transport", JPT. Journal of petroleum technology, 47(3):216-222, 1995.
- M. A. Biot & W. L. Medlin. "Theory of Sand Transport in Thin Fluids", Paper presented at the SPE Annual Technical Conference and Exhibition, Las Vegas, NV, 1985.
- X. Hu, K. Wu, X. Song, W. Yu, J. Tang, G. Li, & Z. Shen. "A new model for simulating particle transport in a low-viscosity fluid for fluid-driven fracturing", AIChE J. 64 (9), 35423552, 2018.
- R. G. Keck, W. L. Nehmer, & G. S. Strumolo. "A new method for predicting friction pressures and rheology of proppant-laden fracturing fluids", SPE Prod. Eng., 7(1):21-28, 1992.
- M. McClure. "Bed load proppant transport during slickwater hydraulic fracturing: insights from comparisons between published laboratory data and correlations for sediment and pipeline slurry transport", J. Pet. Sci. Eng. 161 (2), 599610, 2018.
- M. C. Miller, I. N. McCave, & P. D. Komar. "Threshold of sediment motion under unidirectional currents", Sedimentology 24 (4), 507527, 1977.
- P. L. Wiberg & J. D. Smith. "Model for calculating bed load transport of sediment", J. Hydraul. Eng. 115 (1), 101123, 1989.

1.5.4 Constitutive Models

Constitutive models describe relations between various physical quantities. In a physics simulation they are used to model the response or state of material (solid, fluid, or a mixture) as a function of input variables. There are many types of constitutive models available in GEOS. These models are grouped together based on their input/output interface.

Solid Models

A solid model governs the relationship between deformation and stress in a solid (or porous) material. GEOS provides interfaces for both small and large deformation models, as well as specific implementations of a number of well known models.

Deformation Theories

Table of Contents

Introduction

The solid mechanics solvers in GEOS work in a time-discrete setting, in which the system state at time t^n is fully known, and the goal of the solution procedure is to advance forward one timestep to $t^{n+1} = t^n + \Delta t$. As part of this process, calls to a solid model must be made to compute the updated stress σ^{n+1} resulting from incremental deformation over the timestep. History-dependent models may also need to compute updates to one or more internal state variables Q^{n+1}.

The exact nature of the incremental update will depend, however, on the kinematic assumptions made. Appropriate measures of deformation and stress depend on assumptions of infinitesimal or finite strain, as well as other factors like rate-dependence and material anisotropy.

This section briefly reviews three main classes of solid models in GEOS, grouped by their kinematic assumptions. The presentation is deliberately brief, as much more extensive presentations can be found in almost any textbook on linear and nonlinear solid mechanics.

Small Strain Models

Let u denote the displacement field, and ∇u its gradient. In small strain theory, ones assumes the displacement gradients $\nabla u \ll 1$. In this case, it is sufficient to use the linearized strain tensor

$$\epsilon = \frac{1}{2}\left(\nabla u + u \nabla\right)$$

as the deformation measure. Higher-order terms present in finite strain theories are neglected. For inelastic problems, this strain is additively decomposed into elastic and inelastic components as

$$\epsilon = \epsilon^e + \epsilon^i.$$

Inelastic strains can arise from a number of sources: plasticity, damage, etc. Most constitutive models (including nonlinear elastic and inelastic models) can then be generically expressed in rate form as

$$\dot{\sigma} = c : \dot{\epsilon}^e$$

where $\dot{\sigma}$ is the Cauchy stress rate and c is the tangent stiffness tensor. Observe that the stress rate is driven by the elastic component $\dot{\epsilon}^e$ of the strain rate.

In the time-discrete setting (as implemented in the code) the incremental constitutive update for stress is computed from a solid model update routine as

$$\sigma^{n+1} = \sigma(\Delta\epsilon, \Delta t, Q^n),$$

where $\Delta\epsilon = \epsilon^{n+1} - \epsilon^n$ is the incremental strain, Δt is the timestep size (important for rate-dependent models), and Q^n is a collection of material state variables (which may include the previous stress and strain).

For path and rate independent models, such as linear elasticity, a simpler constitutive update may be formulated in terms of the total strain:

$$\sigma^{n+1} = \sigma(\epsilon^{n+1}).$$

GEOS will use this latter form in specific, highly-optimized solvers when we know in advance that a linear elastic model is being applied. The more general interface is the the default, however, as it can accommodate a much wider range of constitutive behavior within a common interface.

When implicit timestepping is used, the solid models must also provide the stiffness tensor,

$$c^{n+1} = \frac{\partial\sigma^{n+1}}{\partial\epsilon^{n+1}},$$

in order to accurately linearize the governing equations. In many works, this fourth-order tensor is referred to as the algorithmic or consistent tangent, in the sense that it must be "consistent" with the discrete timestepping scheme being used (Simo and Hughes 1987). For inelastic models, it depends not only on the intrinsic material stiffness, but also the incremental nature of the loading process. The correct calculation of this stiffness can have a dramatic impact on the convergence rate of Newton-type solvers used in the implicit solid mechanics solvers.

Finite Deformation Models with Hypo-Materials

In the finite deformation regime, there are two broad classes of constitutive models frequently used:

- Hypo-elastic models (and inelastic extensions)
- Hyper-elastic models (and inelastic extensions)

Hypo-materials typically rely on a rate-form of the constitutive equations expressed in the spatial configuration. Let $v(x,t)$ denote the spatial velocity field. It can be decomposed into symmetric and anti-symmetric components as

$$d = \frac{1}{2}\left(\nabla v + v\nabla\right) \qquad \text{and} \qquad w = \frac{1}{2}\left(\nabla v - v\nabla\right),$$

where d is the rate of deformation tensor and w is the spin tensor. A hypo-material model can be written in rate form as

$$\mathring{\tau} = c : d^e$$

where $\mathring{\tau}$ is an objective rate of the Kirchoff stress tensor, c is the tangent stiffness tensor, and d^e is the elastic component of the deformation rate. We see that the structure is similar to the rate form in the small strain regime, except the rate of Cauchy stress is replaced with an objective rate of Kirchoff stress, and the linearized strain rate is replaced with the rate of deformation tensor.

The key difference separating most hypo-models is the choice of the objective stress rate. In GEOS, we adopt the incrementally objective integration algorithm proposed by Hughes & Winget (1980). This method relies on the concept of an incrementally rotating frame of reference in order to preserve objectivity of the stress rate. In particular, the stress update sequence is

$$\begin{aligned} \Delta R &= (I - \frac{1}{2}\Delta t w)^{-1}(I + \frac{1}{2}\Delta t w) && \text{(compute incremental rotation)}, \\ \bar{\tau}^n &= \Delta R \tau^n \Delta R^T && \text{(rotate previous stress)}, \\ \tau^{n+1} &= \bar{\tau}^n + \Delta\tau && \text{(call constitutive model to update stress)}. \end{aligned}$$

First, the previous timestep stress is rotated to reflect any rigid rotations occuring over the timestep. If the model has tensor-valued state variables besides stress, these must also be rotated. Then, a standard constitutive update routine can be called, typically driven by the incremental strain $\Delta\epsilon = \Delta t d$. In fact, an identical update routine as used for small strain models can be re-used at this point.

Note: Hypo-models suffer from several well known deficiencies. Most notably, the energy dissipation in a closed loading cycle of a hypo-elastic material is not guaranteed to be zero, as one might desire from thermodynamic considerations.

Finite Deformation Models with Hyper-Materials

Hyper-elastic models (and inelastic extensions) attempt to correct the thermodynamic deficiencies of their hypo-elastic cousins. The constitutive update can be generically expressed at

$$S^{n+1} = S(\Delta\mathbf{F}, Q^n, \Delta t),$$

where S is the second Piola-Kirchoff stress and $\Delta\mathbf{F}$ is the incremental deformation gradient. Depending on the model, the deformation gradient can be converted to different deformation measures as needed. Similarly, different stress tensors can be recovered through appropriate push-forward and pull-back operations.

In a hyperelastic material, the elastic response is expressed in terms of a stored strain-energy function that serves as the potential for stress, e.g.

$$\mathbf{S} = \frac{\partial\psi(C)}{\partial C},$$

where ψ is the stored energy potential, and C is the right Cauchy-Green deformation tensor. This potential guarantees that the energy dissipated or gained in a closed elastic cycle is zero.

Voigt Notation

In GEOS we express rank-two symmetric tensors using Voigt notation. Stress tensors are represented as an "unrolled" six-component vector storing only the unique component values. For strain tensors, note that *engineering strains* are used such that the shear components of strain are multiplied by a factor of two. With this representation, the strain energy density is, conveniently, the inner product of the stress and strain vectors.

Voigt representation of common rank-2 symmetric tensors are:

$$\sigma = \begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{23} \\ \sigma_{13} \\ \sigma_{12} \end{bmatrix}, \mathbf{S} = \begin{bmatrix} S_{11} \\ S_{22} \\ S_{33} \\ S_{23} \\ S_{13} \\ S_{12} \end{bmatrix}, \epsilon = \begin{bmatrix} \epsilon_{11} \\ \epsilon_{22} \\ \epsilon_{33} \\ 2\epsilon_{23} \\ 2\epsilon_{13} \\ 2\epsilon_{12} \end{bmatrix}, \mathbf{D} = \begin{bmatrix} D_{11} \\ D_{22} \\ D_{33} \\ 2D_{23} \\ 2D_{13} \\ 2D_{12} \end{bmatrix}, \mathbf{E} = \begin{bmatrix} E_{11} \\ E_{22} \\ E_{33} \\ 2E_{23} \\ 2E_{13} \\ 2E_{12} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} B_{11} \\ B_{22} \\ B_{33} \\ 2B_{23} \\ 2B_{13} \\ 2B_{12} \end{bmatrix}, \mathbf{C} = \begin{bmatrix} C_{11} \\ C_{22} \\ C_{33} \\ 2C_{23} \\ 2C_{13} \\ 2C_{12} \end{bmatrix},$$

where σ is the Cauchy stress, $\mathbf{S}$ is the second Piola-Kirchhoff stress, ϵ is the small strain tensor, $\mathbf{D}$ is the rate of deformation tensor, $\mathbf{E}$ is the Lagrangian finite strain tensor, $\mathbf{B}$ is the left Cauchy-Green tensor, $\mathbf{C}$ is the right Cauchy-Green deformation tensor.

Note: The factor of two in the shear components of strain (and strain-like) quantities is a frequent source of confusion, even for expert modelers. It can be particularly challenging to use in nuanced situations like stiffness tensor calculations or invariant decompositions. If you plan to implement new models within GEOS, please pay extra attention to this detail. We also provide many common operations in centralized helper functions to avoid re-inventing the wheel.

Plasticity Notation

Table of Contents

Overview

According to the theory of plasticity in the small strain regime, the total strain $\boldsymbol{\epsilon}$ can be additively split into elastic ($\boldsymbol{\epsilon}^e$) and plastic ($\boldsymbol{\epsilon}^p$) strains:

$$\boldsymbol{\epsilon} = \boldsymbol{\epsilon}^e + \boldsymbol{\epsilon}^p.$$

The plastic strain tensor is obtained from the flow rule:

$$\dot{\boldsymbol{\epsilon}}^p = \dot{\lambda} \frac{\partial g}{\partial \boldsymbol{\sigma}},$$

in which $\dot{\lambda} \geq 0$ is the magnitude of plastic strain rate and g is the plastic potential. The elastic strain is related to Cauchy stress tensor in rate form as:

$$\dot{\boldsymbol{\sigma}} = c^e : \dot{\boldsymbol{\epsilon}}^e,$$

where c^e is the fourth order elastic stiffness tensor. The Cauchy stress tensor is related to the total strain as

$$\dot{\boldsymbol{\sigma}} = \boldsymbol{c}^{ep} : \dot{\boldsymbol{\epsilon}},$$

in which c^{ep} is the fourth order elasto-plastic stiffness tensor.

Two-Invariant Models

Two-invariant plasticity models use the first invariant of the Cauchy stress tensor and the second invariant of the deviatoric stress tensor to describe the yield surface.

Here we use the following stress invariants to define the yield surface: the von Mises stress $q = \sqrt{3J_2} = \sqrt{3/2}\|\boldsymbol{s}\|$ and mean normal stress $p = I_1/3$. Here, I_1 and J_2 are the first invariant of the stress tensor and second invariant of the deviatoric stress, defined as

$$I_1 = tr(\boldsymbol{\sigma})/3\,, \quad J_2 = \frac{1}{2}\|\boldsymbol{s}\|^2\,, \quad \boldsymbol{s} = \boldsymbol{\sigma} - p\mathbf{1}\,,$$

in which $\mathbf{1}$ is the identity tensor.

Similarly, we can define invariants of strain tensor, namely, volumetric strain ϵ_v and deviatoric strain ϵ_s.

$$\epsilon_v = tr(\boldsymbol{\epsilon})\,, \quad \epsilon_s = \sqrt{\frac{2}{3}}\|\boldsymbol{e}\|\,, \quad \text{where} \quad \boldsymbol{e} = \boldsymbol{\epsilon} - \frac{1}{3}\epsilon_v\mathbf{1}.$$

Stress and strain tensors can then be recomposed from the invariants as:

$$\boldsymbol{\sigma} = p\,\mathbf{1} + \sqrt{\frac{2}{3}}q\,\hat{\boldsymbol{n}}$$

$$\boldsymbol{\epsilon} = \frac{1}{3}\epsilon_v\mathbf{1} + \sqrt{\frac{3}{2}}\epsilon_s\hat{\boldsymbol{n}}$$

in which $\hat{\boldsymbol{n}} = \boldsymbol{e}/\|\boldsymbol{e}\|$.

The following two-invariant models are currently implemented in GEOS:

- *DruckerPrager*
- *J2Plasticity*
- *ModifiedCamClay*
- *DelftEgg*

Three-Invariant Models

Several three-invariant models are under active development, but are not yet available in develop. If you are interested in helping to add additional material models, please submit a feature request.

Triaxial Driver

Table of Contents

Introduction

When calibrating solid material parameters to experimental data, it can be a hassle to launch a full finite element simulation to mimic experimental loading conditions. Instead, GEOS provides a `TriaxialDriver` allowing the user to run loading tests on a single material point. This makes it easy to understand the material response and fit it to lab data. The driver itself is launched like any other GEOS simulation, but with a particular XML structure:

```
./bin/geosx -i myTest.xml
```

XML Structure

A typical XML file to run the triaxial driver will have the following key elements. We present the whole file first, before digging into the individual blocks.

```xml
<?xml version="1.0" ?>

<Problem>
  <Tasks>
    <TriaxialDriver
      name="triaxialDriver"
      material="drucker"
      mode="mixedControl"
      axialControl="strainFunction"
      radialControl="stressFunction"
      initialStress="-1.0"
      steps="40"
      output="none"
      baseline="testTriaxial_druckerPragerExtended.txt"
      logLevel="0"/>
  </Tasks>

  <Events
    maxTime="1">
    <SoloEvent
      name="triaxialDriver"
      target="/Tasks/triaxialDriver"/>
  </Events>

  <Constitutive>
    <ExtendedDruckerPrager
      name="drucker"
      defaultDensity="2700"
      defaultBulkModulus="500"
      defaultShearModulus="300"
      defaultCohesion="0.0"
      defaultInitialFrictionAngle="15.27"
```

(continues on next page)

(continued from previous page)

```
      defaultResidualFrictionAngle="23.05"
      defaultDilationRatio="1.0"
      defaultHardening="0.001"/>
  </Constitutive>

  <Functions>
    <TableFunction
      name="strainFunction"
      inputVarNames="{ time }"
      coordinates="{ 0.0, 3.0, 4.0, 7.0, 8.0 }"
      values="{ 0, -0.003, -0.002, -0.005, -0.004 }"/>

    <TableFunction
      name="stressFunction"
      inputVarNames="{ time }"
      coordinates="{ 0.0, 8.0 }"
      values="{ -1.0, -1.0 }"/>
  </Functions>

  <!-- Mesh is not used, but GEOSX throws an error without one.  Will resolve soon-->
  <Mesh>
    <InternalMesh
      name="mesh1"
      elementTypes="{ C3D8 }"
      xCoords="{ 0, 1 }"
      yCoords="{ 0, 1 }"
      zCoords="{ 0, 1 }"
      nx="{ 1 }"
      ny="{ 1 }"
      nz="{ 1 }"
      cellBlockNames="{ cellBlock01 }"/>
  </Mesh>

  <ElementRegions>
    <CellElementRegion
      name="dummy"
      cellBlocks="{ cellBlock01 }"
      materialList="{ dummy }"/>
  </ElementRegions>
</Problem>
```

The first thing to note is that the XML structure is identical to a standard GEOS input deck. In fact, once the constitutive block is calibrated, one could start adding solver and discretization blocks to the same file to create a proper field simulation. This makes it easy to go back and forth between calibration and simulation.

The `TriaxialDriver` is added as a `Task`, a particular type of executable event often used for simple actions. It is added as a `SoloEvent` to the event queue. This leads to a trivial event queue, since all we do is launch the driver and then quit.

Internally, the triaxial driver uses a simple form of time-stepping to advance through the loading steps, allowing for both rate-dependent and rate-independent models to be tested. This timestepping is handled independently from the more complicated time-stepping pattern used by physics `Solvers` and coordinated by the `EventManager`. In particular, in the XML file above, the `maxTime` parameter in the `Events` block is an event manager control, controlling when/if certain events occur. Once launched, the triaxial driver internally determines its own max time and timestep size using a combination of the strain function's time coordinates and the requested number of loadsteps. It is therefore helpful to think of the driver as an instantaneous *event* (from the event manager's point of view), but one which has a separate, internal clock.

The key parameters for the TriaxialDriver are:

Name	Type	Default	Description
axialControl	string	required	Function controlling axial stress or strain (depending on test mode)
baseline	path	none	Baseline file
initialStress	real64	required	Initial stress (scalar used to set an isotropic stress state)
logLevel	integer	0	Log level
material	string	required	Solid material to test
mode	string	required	Test mode [stressControl, strainControl, mixedControl]
name	string	required	A name is required for any non-unique nodes
output	string	none	Output file
radialControl	string	required	Function controlling radial stress or strain (depending on test mode)
steps	integer	required	Number of load steps to take

Note: GEOS uses the *engineering* sign convention where compressive stresses and strains are *negative*. This is one of the most frequent issues users make when calibrating material parameters, as stress- and strain-like quantities often need to be negative to make physical sense. You may note in the XML above, for example, that `stressFunction` and `strainFunction` have negative values for a compressive test.

Test Modes

The most complicated part of the driver is understanding how the stress and strain functions are applied in different testing modes. The driver mimics laboratory core tests, with loading controlled in the axial and radial directions. These conditions may be either strain-controlled or stress-controlled, with the user providing time-dependent functions to describe the loading. The following table describes the available test modes in detail:

mode	**axial loading**	**radial loading**	**initial stress**
`strainContr`	axial strain controlled with `axialControl`	radial strain controlled with `radialControl`	isotropic stress using `initialStress`
`stressContr`	axial stress controlled with `axialControl`	radial stress controlled with `radialControl`	isotropic stress using `initialStress`
`mixedContrc`	axial strain controlled with `axialControl`	radial stress controlled with `radialControl`	isotropic stress using `initialStress`

Note that a classical triaxial test can be described using either the `stressControl` or `mixedControl` mode. We recommend using the `mixedControl` mode when possible, because this almost always leads to well-posed loading conditions. In a pure stress controlled test, it is possible for the user to request that the material sustain a load beyond its intrinsic strength envelope, in which case there is no feasible solution and the driver will fail to converge. Imagine, for example, a perfectly plastic material with a yield strength of 10 MPa, but the user attempts to load it to 11 MPa.

A volumetric test can be created by setting the axial and radial control functions to the same time history function. Similarly, an oedometer test can be created by setting the radial strain to zero.

The user should be careful to ensure that the initial stress set via the `initialStress` value is consistent any applied stresses set through axial or radial loading functions. Otherwise, the material may experience sudden and unexpected deformation at the first timestep because it is not in static equilibrium.

Output Format

The output key is used to identify a file to which the results of the simulation are written. If this key is omitted, or the user specifies output="none", file output will be suppressed. The file is a simple ASCII format with a brief header followed by test data:

```
# column 1 = time
# column 2 = axial_strain
# column 3 = radial_strain_1
# column 4 = radial_strain_2
# column 5 = axial_stress
# column 6 = radial_stress_1
# column 7 = radial_stress_2
# column 8 = newton_iter
# column 9 = residual_norm
0.0000e+00  0.0000e+00 0.0000e+00 0.0000e+00 -1.0000e+00 -1.0000e+00 -1.0000e+00 0.
↪0000e+00 0.0000e+00
1.6000e-01 -1.6000e-04 4.0000e-05 4.0000e-05 -1.1200e+00 -1.0000e+00 -1.0000e+00 2.
↪0000e+00 0.0000e+00
3.2000e-01 -3.2000e-04 8.0000e-05 8.0000e-05 -1.2400e+00 -1.0000e+00 -1.0000e+00 2.
↪0000e+00 0.0000e+00
...
```

This file can be readily plotted using any number of plotting tools. Each row corresponds to one timestep of the driver, starting from initial conditions in the first row.

We note that the file contains two columns for radial strain and two columns for radial stress. For an isotropic material, the stresses and strains along the two radial axes will usually be identical. We choose to output this way, however, to accommodate both anisotropic materials and true-triaxial loading conditions. In these cases, the stresses and strains in the radial directions could potentially differ.

These columns can be added and subtracted to produce other quantities of interest, like mean stress or deviatoric stress. For example, we can plot the output produce stress / strain curves (in this case for a plastic rather than simple elastic material):

Fig. 1.75: Stress/strain behavior for a plastic material.

In this plot, we have reversed the sign convention to be consistent with typical experimental plots. Note also that the strainFunction includes two unloading cycles, allowing us to observe both plastic loading and elastic unloading.

Model Convergence

The last two columns of the output file contain information about the convergence behavior of the material driver. In triaxial mode, the mixed nature of the stress/strain control requires using a Newton solver to converge the solution. This last column reports the number of Newton iterations and final residual norm. Large values here would be indicative of the material model struggling (or failing) to converge. Convergence failures can result from several reasons, including:

1. Inappropriate material parameter settings
2. Overly large timesteps
3. Infeasible loading conditions (i.e. trying to load a material to a physically-unreachable stress point)
4. Poor model implementation

We generally spend a lot of time vetting the material model implementations (#4). When you first encounter a problem, it is therefore good to explore the other three scenarios first. If you find something unusual in the model implementation or are just really stuck, please submit an issue on our issue tracker so we can help resolve any bugs.

Unit Testing

The development team also uses the Triaxial Driver to perform unit testing on the various material models within GEOS. The optional argument `baseline` can be used to point to a previous output file that has been validated (e.g. against analytical or experimental benchmarks). If such a file is specified, the driver will perform a loading run and then compare the new results against the baseline. In this way, any regressions in the material models can be quickly identified.

Developers of new models are encouraged to add their own baselines to `src/coreComponents/constitutive/unitTests`. Adding additional tests is straightforward:

1. Create a new xml file for your test in `src/coreComponents/constitutive/unitTests`. There are several examples is this directory already to use as a template. We suggest using the naming convention `testTriaxial_myTest.xml`, so that all triaxial tests will be grouped together alphabetically. Set the `output` file to `testTriaxial_myTest.txt`, and run your test. Validate the results however is appropriate.
2. This output file will now become your new baseline. Replace the `output` key with `baseline` so that the driver can read in your file as a baseline for comparison. Make sure there is no remaining `output` key, or set `output=none`, to suppress further file output. While you can certainly write a new output for debugging purposes, during our automated unit tests we prefer to suppress file output. Re-run the triaxial driver to confirm that the comparison test passes.
3. Modify `src/coreComponents/constitutive/unitTests/CMakeLists.txt` to enable your new test in the unit test suite. In particular, you will need to add your new XML file to the existing list in the `gtest_triaxial_xmls` variable:

```
set( gtest_triaxial_xmls
     testTriaxial_elasticIsotropic.xml
     testTriaxial_druckerPragerExtended.xml
     testTriaxial_myTest.xml
   )
```

4. Run `make` in your build directory to make sure the CMake syntax is correct
5. Run `ctest -V -R Triax` to run the triaxial unit tests. Confirm your test is included and passes properly.

If you run into troubles, do not hesitate to contact the development team for help.

Model: Elastic Isotropic

Overview

This model may be used for solid materials with a linear elastic isotropic behavior. The relationship between stress and strain is given by Hooke's Law, expressed as:

$$\sigma_{ij} = \lambda \epsilon_{kk} + 2\mu\epsilon_{ij},$$

where σ_{ij} is the ij component of the Cauchy stress tensor, ϵ_{ij} is the ij component of the strain tensor, λ is the first Lamé elastic constant, and μ is the elastic shear modulus.

Hooke's Law may also be expressed using Voigt notation for stress and strain vectors as:

$$\sigma = C \cdot \epsilon,$$

or,

$$\begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{23} \\ \sigma_{13} \\ \sigma_{12} \end{bmatrix} = \begin{bmatrix} 2\mu+\lambda & \lambda & \lambda & 0 & 0 & 0 \\ \lambda & 2\mu+\lambda & \lambda & 0 & 0 & 0 \\ \lambda & \lambda & 2\mu+\lambda & 0 & 0 & 0 \\ 0 & 0 & 0 & \mu & 0 & 0 \\ 0 & 0 & 0 & 0 & \mu & 0 \\ 0 & 0 & 0 & 0 & 0 & \mu \end{bmatrix} \begin{bmatrix} \epsilon_{11} \\ \epsilon_{22} \\ \epsilon_{33} \\ 2\epsilon_{23} \\ 2\epsilon_{13} \\ 2\epsilon_{12} \end{bmatrix}.$$

Variations

For finite deformation solvers, the elastic isotropic model can be called within a hypo-elastic update routine. See *Finite Deformation Models with Hypo-Materials*

Parameters

The following attributes are supported. Note that any two elastic constants can be provided, and the other two will be internally calculated. The "default" keyword in front of certain properties indicates that this is the default value adopted for a region unless the user separately specifies a heterogeneous field via the `FieldSpecification` mechanism.

Name	Type	Default	Description
defaultBulkModulus	real64	-1	Default Bulk Modulus Parameter
defaultDensity	real64	required	Default Material Density
defaultPoissonRatio	real64	-1	Default Poisson's Ratio
defaultShearModulus	real64	-1	Default Shear Modulus Parameter
defaultThermalExpansionCoefficient	real64	0	Default Linear Thermal Expansion Coefficient of the Solid Rock Frame
defaultYoungModulus	real64	-1	Default Young's Modulus
name	string	required	A name is required for any non-unique nodes

Example

A typical `Constititutive` block will look like:

```xml
<Constitutive>
  <ElasticIsotropic
     name="shale"
     defaultDensity="2700"
     defaultBulkModulus="60.0e6"
     defaultShearModulus="30.0e6" />
</Constitutive>
```

Model: Elastic Isotropic Pressure Dependent

Overview

This model may be used for solid materials with a pressure-dependent elastic isotropic behavior. The relationship between stress and strain is given by a hyperelastic law. The elastic constitutive equations for the volumetric and deviatoric stresses and strain are expressed as:

$$p = p_0 \exp \left(\frac{\epsilon_{v0} - \epsilon_v^e}{c_r} \right), \quad q = 3\mu\epsilon_s^e$$

where p and q are the volumetric and deviatoric components of the Cauchy stress tensor. ϵ_v^e and ϵ_s^e are the volumetric and deviatoric components of the strain tensor. ϵ_{v0} and p_0 are the initial volumetric strain and initial pressure. C_r denotes the elastic compressibility index, and μ is the elastic shear modulus. In this model, the shear modulus is constant and the bulk modulus, K, varies linearly with pressure as follows:

$$K = -\frac{p}{c_r}$$

Parameters

The following attributes are supported. Note that two elastic constants c_r and μ, as well as the initial volumetric strain and initial pressure need to be provided. The "default" keyword in front of certain properties indicates that this is the default value adopted for a region unless the user separately specifies a heterogeneous field via the `FieldSpecification` mechanism.

Example

A typical `Constititutive` block will look like:

```xml
<Constitutive>
  <ElasticIsotropicPressureDependent
    name="elasticPressure"
    defaultDensity="2700"
    defaultRefPressure="-1.0"
    defaultRefStrainVol="1"
    defaultRecompressionIndex="0.003"
    defaultShearModulus="200"/>
</Constitutive>
```

Model: Elastic Transverse-Isotropic

Overview

This model may be used for solid materials with a linear elastic, transverse-isotropic behavior. This is most readily expressed in Voight notation as

$$\begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{23} \\ \sigma_{13} \\ \sigma_{12} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} & C_{13} & 0 & 0 & 0 \\ C_{12} & C_{11} & C_{13} & 0 & 0 & 0 \\ C_{13} & C_{13} & C_{33} & 0 & 0 & 0 \\ 0 & 0 & 0 & C_{44} & 0 & 0 \\ 0 & 0 & 0 & 0 & C_{44} & 0 \\ 0 & 0 & 0 & 0 & 0 & (C_{11}-C_{12})/2 \end{bmatrix} \begin{bmatrix} \epsilon_{11} \\ \epsilon_{22} \\ \epsilon_{33} \\ 2\epsilon_{23} \\ 2\epsilon_{13} \\ 2\epsilon_{12} \end{bmatrix}.$$

This system contains five independent constants. These constants are calculated from the input parameters indicated below.

Parameters

The following attributes are supported. The "default" keyword in front of certain properties indicates that this is the default value adopted for a region unless the user separately specifies a heterogeneous field via the `FieldSpecification` mechanism.

Name	Type	Default	Description
defaultC11	real64	-1	Default Stiffness Parameter C11
defaultC13	real64	-1	Default Stiffness Parameter C13
defaultC33	real64	-1	Default Stiffness Parameter C33
defaultC44	real64	-1	Default Stiffness Parameter C44
defaultC66	real64	-1	Default Stiffness Parameter C66
defaultDensity	real64	required	Default Material Density
defaultPoissonRatioAxialTransverse	real64	-1	Default Axial-Transverse Poisson's Ratio
defaultPoissonRatioTransverse	real64	-1	Default Transverse Poisson's Ratio
defaultShearModulusAxialTransverse	real64	-1	Default Axial-Transverse Shear Modulus
defaultThermalExpansionCoefficient	real64	0	Default Linear Thermal Expansion Coefficient of the Solid Rock Frame
defaultYoungModulusAxial	real64	-1	Default Axial Young's Modulus
defaultYoungModulusTransverse	real64	-1	Default Transverse Young's Modulus
name	string	required	A name is required for any non-unique nodes

Example

A typical `Constititutive` block will look like:

```
<Constitutive>
  <ElasticTransverseIsotropic
     name="shale"
     defaultDensity="2700"
     defaultPoissonRatioAxialTransverse="0.20"
     defaultPoissonRatioTransverse="0.30"
     defaultYoungModulusAxial="50.0e6"
     defaultYoungModulusTransverse="60.0e6"
     defaultShearModulusAxialTransverse="30.0e6" />
</Constitutive>
```

Model: Elastic Orthotropic

Overview

This model may be used for solid materials with a linear elastic, orthotropic behavior. This is most readily expressed in Voight notation as

$$\begin{bmatrix}\sigma_{11}\\ \sigma_{22}\\ \sigma_{33}\\ \sigma_{23}\\ \sigma_{13}\\ \sigma_{12}\end{bmatrix} = \begin{bmatrix}C_{11} & C_{12} & C_{13} & 0 & 0 & 0\\ C_{12} & C_{22} & C_{23} & 0 & 0 & 0\\ C_{13} & C_{23} & C_{33} & 0 & 0 & 0\\ 0 & 0 & 0 & C_{44} & 0 & 0\\ 0 & 0 & 0 & 0 & C_{55} & 0\\ 0 & 0 & 0 & 0 & 0 & C_{66}\end{bmatrix} \begin{bmatrix}\epsilon_{11}\\ \epsilon_{22}\\ \epsilon_{33}\\ 2\epsilon_{23}\\ 2\epsilon_{13}\\ 2\epsilon_{12}\end{bmatrix}.$$

This system contains nine independent constants. These constants are calculated from the input parameters indicated below.

Parameters

The following attributes are supported. The "default" keyword in front of certain properties indicates that this is the default value adopted for a region unless the user separately specifies a heterogeneous field via the `FieldSpecification` mechanism.

Name	Type	Default	Description
defaultC11	real64	-1	Default C11 Component of Voigt Stiffness Tensor
defaultC12	real64	-1	Default C12 Component of Voigt Stiffness Tensor
defaultC13	real64	-1	Default C13 Component of Voigt Stiffness Tensor
defaultC22	real64	-1	Default C22 Component of Voigt Stiffness Tensor
defaultC23	real64	-1	Default C23 Component of Voigt Stiffness Tensor
defaultC33	real64	-1	Default C33 Component of Voigt Stiffness Tensor
defaultC44	real64	-1	Default C44 Component of Voigt Stiffness Tensor
defaultC55	real64	-1	Default C55 Component of Voigt Stiffness Tensor
defaultC66	real64	-1	Default C66 Component of Voigt Stiffness Tensor
defaultDensity	real64	required	Default Material Density
defaultE1	real64	-1	Default Young's Modulus E1
defaultE2	real64	-1	Default Young's Modulus E2
defaultE3	real64	-1	Default Young's Modulus E3
defaultG12	real64	-1	Default Shear Modulus G12
defaultG13	real64	-1	Default Shear Modulus G13
defaultG23	real64	-1	Default Shear Modulus G23
defaultNu12	real64	-1	Default Poission's Ratio Nu12
defaultNu13	real64	-1	Default Poission's Ratio Nu13
defaultNu23	real64	-1	Default Poission's Ratio Nu23
defaultThermalExpansionCoefficient	real64	0	Default Linear Thermal Expansion Coefficient of the Solid Rock Frame
name	string	required	A name is required for any non-unique nodes

Example

A typical `Constititutive` block will look like:

```
<Constitutive>
  <ElasticOrthotropic
     name="shale"
     defaultDensity="2700"
     defaultNu12="0.20"
     defaultNu13="0.25"
     defaultNu23="0.30"
     defaultE1="40.0e6"
     defaultE2="50.0e6"
     defaultE3="60.0e6"
     defaultG12="20.0e6"
     defaultG13="30.0e6"
     defaultG23="40.0e6" />
</Constitutive>
```

Model: Drucker-Prager

Overview

This model may be used to represent a solid material with plastic response to loading according to the Drucker-Prager yield criterion below:

$$f(p,q) = q + b\,p - a = 0.$$

The material behavior is linear elastic (see *Model: Elastic Isotropic*) for $f < 0$, and plastic for $f = 0$. The two material parameters a and b are derived by approximating the Mohr-Coulomb surface with a cone. Figure 3 shows the Mohr-Coulomb yield surface and circumscribing Drucker-Prager surface in principal stress space. The Drucker-Prager yield surface has a circular cross-section in the deviatoric plane that passes through the tension or compression corners of the Mohr-Coulomb yield surface, as shown in the Figure 4. The material parameters a and b are derived as:

$$a = \frac{6\,c\,\cos\phi}{3 \pm \sin\phi}, \quad b = \frac{6\,\sin\phi}{3 \pm \sin\phi}$$

where plus signs are for circles passing through the tension corners, and minus signs are for circles passing through compression corners. Also, ϕ and c denote friction angle and cohesion, respectively, as defined by the Mohr-Coulomb failure envelope shown in Figure 5. In GEOS, we use a compression corner fit (minus signs) to convert the user-specified friction angle and cohesion to a and b.

We consider a non-associative plastic potential to determine the direction of plastic flow.

$$g(p,q) = q + b'\,p,$$

where $b' \leq b$ is the dilatancy parameter. Setting $b' = b$ leads to associative flow rule, while for $b' < b$ non-associative flow is obtained. The parameter b' is related to dilation angle as:

$$b' = \frac{6\,\sin\psi}{3 \pm \sin\psi},$$

where $\psi \leq \phi$ is the dilation angle. If $\psi > 0$, then the plastic flow is dilative. Again, we use a compression corner fit (minus sign).

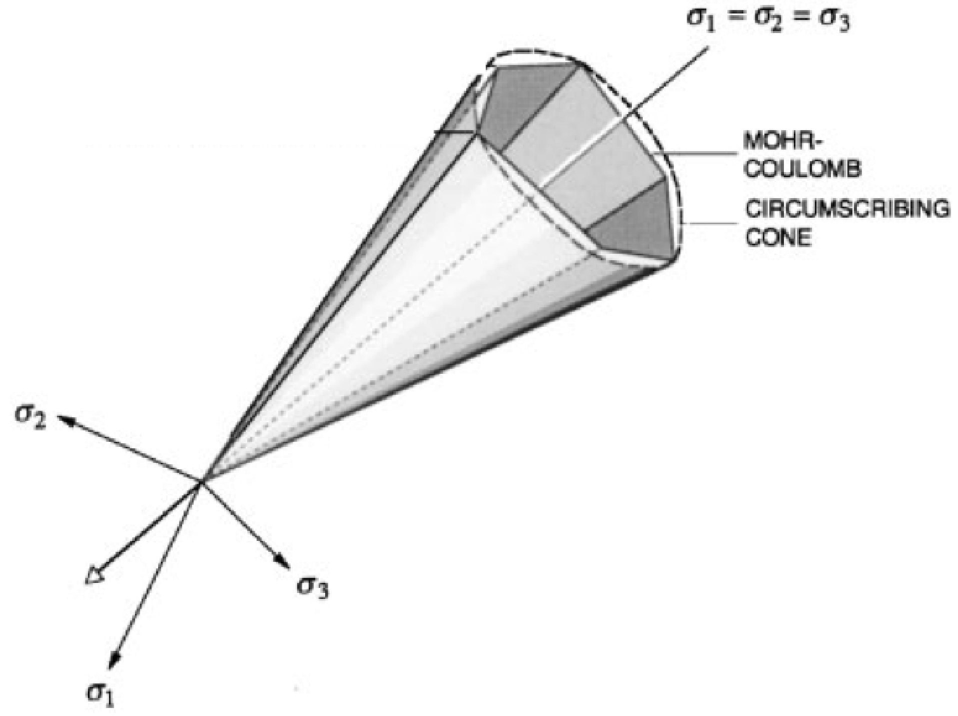


Fig. 1.76: Mohr-Coulomb and Drucker-Prager yield surfaces in principal stress axes (Borja, 2002).

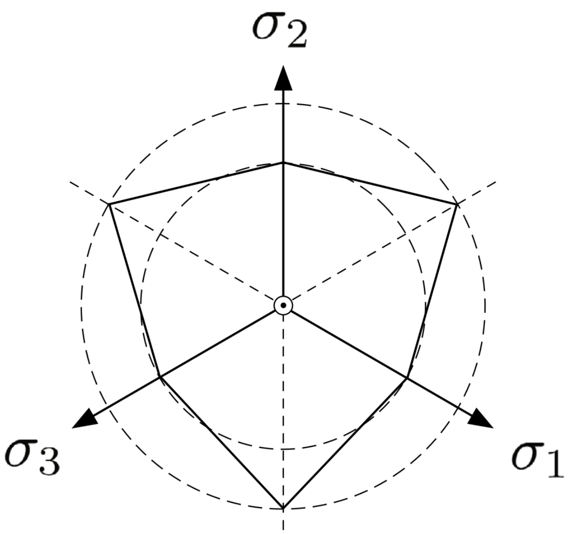


Fig. 1.77: Mohr-Coulomb and Drucker-Prager yield surfaces on the deviatoric plane (Borja, 2013).

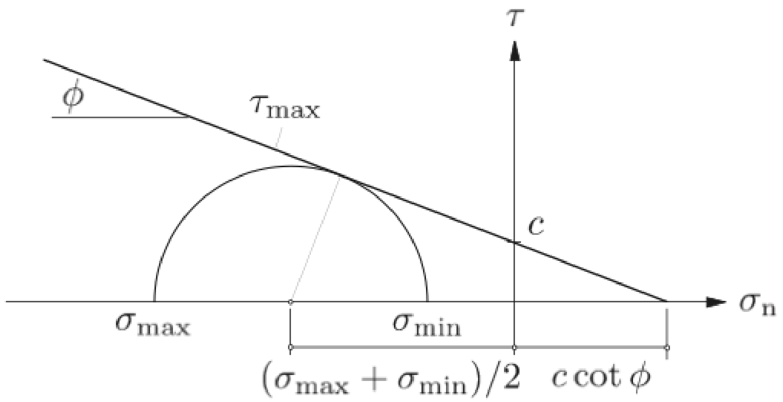


Fig. 1.78: The Mohr-Coulomb failure envelope (Borja, 2013).

A hardening rule is defined which determines how the yield surface will change as a result of plastic deformations. Here we use linear hardening for the cohesion parameter, a,

$$\dot{a} = h\,\dot{\lambda},$$

where h is the hardening parameter. A positive hardening parameter will allow the cohesion to grow, shifting the cohesion intercept vertically on the q-axis. A negative hardening parameter will cause the cohesion to shrink, though negative cohesion values are not allowed. Once all cohesion has been lost, the cohesion will remain at zero, so the cone vertex is fixed at the origin. In either case, the friction and dilation angles remain constant. See the *DruckerPragerExtended* model for an alternative version of hardening behavior.

Parameters

The following attributes are supported:

Name	Type	Default	Description
defaultBulkModulus	real64	-1	Default Bulk Modulus Parameter
defaultCohesion	real64	0	Initial cohesion
defaultDensity	real64	required	Default Material Density
defaultDilationAngle	real64	30	Dilation angle (degrees)
defaultFrictionAngle	real64	30	Friction angle (degrees)
defaultHardeningRate	real64	0	Cohesion hardening/softening rate
defaultPoissonRatio	real64	-1	Default Poisson’s Ratio
defaultShearModulus	real64	-1	Default Shear Modulus Parameter
defaultThermalExpansionCoefficient	real64	0	Default Linear Thermal Expansion Coefficient of the Solid Rock Frame
defaultYoungModulus	real64	-1	Default Young’s Modulus
name	string	required	A name is required for any non-unique nodes

Example

```
<Constitutive>
  <DruckerPrager name="drucker"
                           defaultDensity="2700"
                           defaultBulkModulus="1000.0"
                           defaultShearModulus="1000.0"
                           defaultFrictionAngle="30.0"
                           defaultDilationAngle="20.0"
                           defaultHardeningRate="0.0"
                           defaultCohesion="10.0" />
</Constitutive>
```

Variant: J2 plasticity

J2 yield criterion can be obtained as a special case of the Drucker-Prager model by setting the friction and dilation angles to zero, i.e. $\phi = \psi = 0$.

Model: Extended Drucker-Prager

Overview

This model implements a more sophisticated version of the Drucker-Prager model (see *Model: Drucker-Prager*) allowing for both cohesion and friction hardening / softening. We implement the specific hardening model reported in Liu et al. (2020). The yield surface is given by

$$f(p,q) = q + b\left(p - \frac{a_i}{b_i}\right) = 0,$$

where b is the current yield surface slope, b_i is the initial slope, and a_i is the initial cohesion intercept in p-q space. The vertex of the Drucker-Prager cone is fixed at $p = a_i/b_i$. Let λ denote the accumulated plastic strain measure. The current yield surface slope is given by the hyperbolic relationship

$$b = b_i + \frac{\lambda}{m+\lambda}\left(b_r - b_i\right)$$

with m a parameter controlling the hardening rate. Here, b_r is the residual yield surface slope. If $b_r < b_i$, hardening behavior will be observed, while for $b_r < b_i$ softening behavior will occur.

In the resulting model, the yield surface begins at an initial position defined by the initial cohesion and friction angle. As plastic deformation occurs, the friction angle hardens (or softens) so that it asymptoptically approaches a residual friction angle. The vertex of the cone remains fixed in p-q space, but the cohesion intercept evolves in tandem with the friction angle. See *Liu et al. (2020) <https://doi.org/10.1007/s00603-019-01992-5>* for complete details.

In order to allow for non-associative behavior, we define a "dilation ratio" parameter $\theta \in [0,1]$ such that $b' = \theta b$, where b' is the slope of the plastic potential surface, while b is the slope of the yield surface. Choosing $\theta = 1$ leads to associative behavior, while $\theta = 0$ implies zero dilatancy.

Parameters

The supported attributes will be documented soon.

Example

```
<Constitutive>
  <ExtendedDruckerPrager
    name="edp"
    defaultDensity="2700"
    defaultBulkModulus="500"
    defaultShearModulus="300"
    defaultCohesion="0.0"
    defaultInitialFrictionAngle="15.0"
    defaultResidualFrictionAngle="23.0"
    defaultDilationRatio="1.0"
    defaultHardening="0.001"
  />
</Constitutive>
```

Model: Modified Cam-Clay

This model may be used to represent a solid material with plastic response to loading according to the Modified Cam-Clay (MCC) critical state model. The MCC yield function is defined as:

$$f = q^2 + M^2 p(p - p_c) = 0,$$

where p_c is the preconsolidation pressure, and M is the slope of the critical state line (CSL). M can be related to the critical state friction angle ϕ_{cs} as

$$M = \frac{6 \sin \phi_{cs}}{3 - \sin \phi_{cs}}.$$

Here f represents the yield surface, as shown in Figure 6.

Here we use a hyper-elastic constitutive law using the following elastic rate constitutive equation

$$\dot{p} = -\frac{p}{c_r} \dot{\epsilon}_v^e,$$

where $c_r > 0$ is the elastic compressibility index. The tangential elastic bulk modulus is $K = -\frac{p}{c_r}$ and varies linearly with pressure. We assume a constant shear modulus, and can write stress invariants p and q as

$$p = p_0 \exp\left(\frac{\epsilon_{v0} - \epsilon_v^e}{c_r}\right), \quad q = 3\mu\epsilon_s^e,$$

where p_0 is the reference pressure and ϵ_{v0} is the reference volumetric strain. The hardening law is derived from the linear relationship between logarithm of specific volume and logarithm of preconsolidation pressure, as show in Figure 7.

The hardening law describes evolution of the preconsolidation pressure p_c as

$$\dot{p_c} = -\frac{tr(\dot{\boldsymbol{\epsilon}}^p)}{c_c - c_r} p_c,$$

where c_c is the virgin compressibility index and we have $0 < c_r < c_c$.

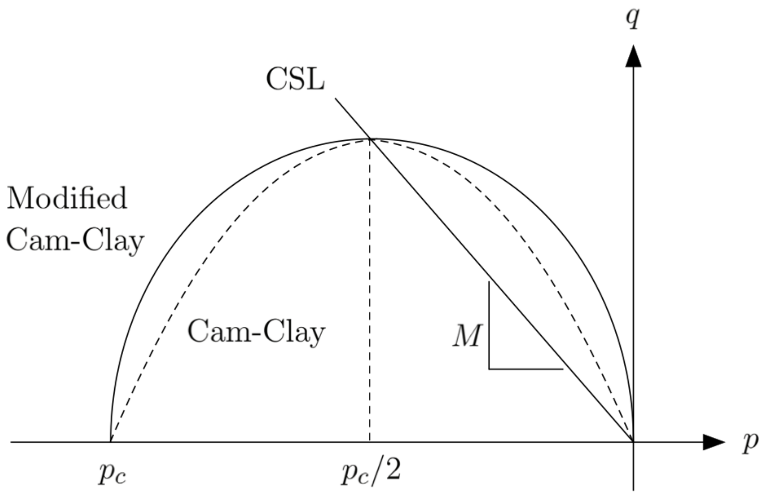


Fig. 1.79: Cam-Clay and Modified Cam-Clay yield surfaces in p-q space (Borja, 2013).

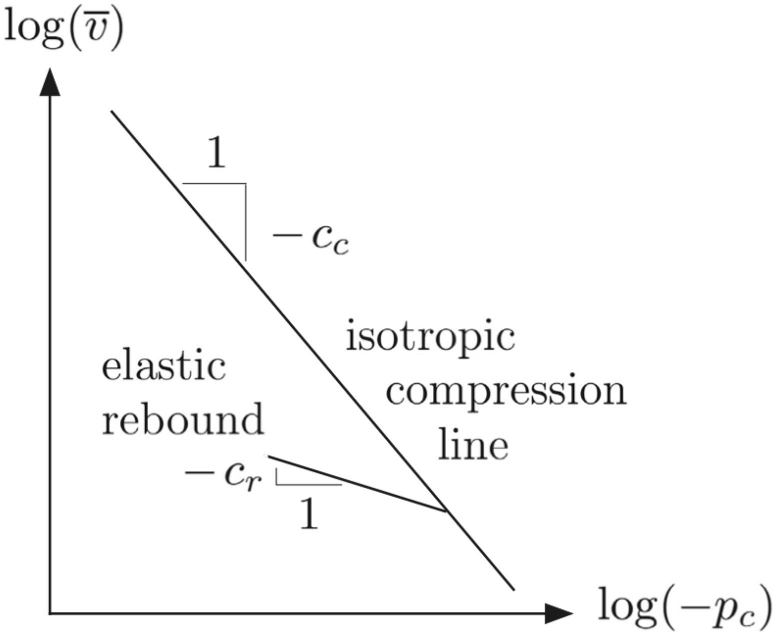


Fig. 1.80: Bilogarithmic hardening law derived from isotropic compression tests (Borja, 2013).

Parameters

The supported attributes will be documented soon.

Example

```
<Constitutive>
  <ModifiedCamClay name="mcc"
                   defaultDensity="2700"
                   defaultRefPressure="-1.0"
                   defaultRefStrainVol="0"
                   defaultShearModulus="200.0"
                   defaultPreConsolidationPressure="-1.5"
                   defaultCslSlope="1.2"
                   defaultRecompressionIndex="0.002"
                   defaultVirginCompressionIndex="0.003" />
</Constitutive>
```

Model: Delft Egg

The Delft-Egg plasticity model uses a generalization of the Modified Cam-Clay yield surface, defined as

$$f = q^2 - M^2 \left[\alpha^2 \, p \left(\frac{2\alpha}{\alpha+1} p_c - p \right) - \frac{\alpha^2(\alpha-1)}{\alpha+1} p_c^2 \right] = 0 \quad (\text{for } p_c > \frac{\alpha}{\alpha+1})$$

$$f = q^2 - M^2 \, p \left(\frac{2\alpha}{\alpha+1} p_c - p \right) = 0 \quad (\text{for } p_c \leq \frac{\alpha}{\alpha+1})$$

where $\alpha \geq 1$ is the shape parameter. For $\alpha = 1$, this model leads to a Modified Cam-Clay (MCC) type model with an ellipsoidal yield surface. For $\alpha > 1$, an egg-shaped yield surface is obtained. The additional parameter makes it easier to fit the cap behavior of a broader range of soils and rocks.

Because Delft-Egg is frequently used for hard rocks, GEOS uses a linear model for the elastic response, rather than the hyper-elastic model used for MCC. This is a slight deviation from the original formulation proposed in the reference above. For reservoir applications, the ability to use a simpler linear model was a frequent user request.

Parameters

The supported attributes will be documented soon.

Example

```
<Constitutive>
   <DelftEgg
    name="DE"
    defaultDensity="2700"
    defaultBulkModulus="10.0e9"
    defaultShearModulus="6.0e9"
    defaultPreConsolidationPressure="-20.0e6"
```

(continues on next page)

(continued from previous page)

```
    defaultShapeParameter="6.5"
    defaultCslSlope="1.2"
    defaultVirginCompressionIndex="0.005"
    defaultRecompressionIndex="0.001"/>
</Constitutive>
```

Damage Models

The damage models are in active development, and documentation will be added when they are ready for production release.

Model: Viscoplasticity

The classical Perzyna-type viscoplasticity models are not suitable for rate-dependent viscoplastic models with non-smooth multisurface, because of using unclearly defined nested viscoplastic loading surfaces. As an alternative, the Duvaut-Lions viscoplastic theory, precludes these difficulties by excluding the concept of nested viscoplastic loading surfaces. The viscoplastic constitutive equation that relates the stress $\boldsymbol{\sigma}$ and the viscoplastic strain rate $\boldsymbol{\epsilon}^{\dot{vp}}$ is given by:

$$\boldsymbol{\sigma} - \bar{\boldsymbol{\sigma}} = \frac{1}{t_*} c : \boldsymbol{\epsilon}^{\dot{vp}}$$

Here, $\bar{\boldsymbol{\sigma}}$ represents the inviscid stress, which is the rate-independent elasto-plastic stress part that can be solved by using elasto-plastic solvers (such as Drucker-Prager, CamClay, etc.). c is the tangent stiffness tensor and t_* is the relaxation time, which is measured in units of time. The viscoplastic strain rate $\dot{\boldsymbol{\epsilon}}^{vp}$ can be approximated using the following finite difference formula:

$$\boldsymbol{\epsilon}^{\dot{vp}} = \frac{1}{\Delta t}(\Delta \boldsymbol{\epsilon} - \Delta \boldsymbol{\epsilon}^{elas})$$

Here, Δt is the time increment, $\Delta \boldsymbol{\epsilon}$ is the total strain increment, and $\Delta \boldsymbol{\epsilon}^{elas}$ is the elastic part of the strain increment. Note that the elastic strain increment is related to the stress increment through Hook's law.

$$\Delta \boldsymbol{\sigma} = c : \Delta \boldsymbol{\epsilon}^{elas}$$

With some arrangements, we can obtain the following formula to update the stress tensor of the Duvaut-Lions elasto-viscoplastic materials:

$$\boldsymbol{\sigma} = r_t \hat{\boldsymbol{\sigma}} + (1 - r_t)\bar{\boldsymbol{\sigma}}$$

Here, the time ratio r_t is calculated from the relaxation time t_* and the time increment Δt as:

$$r_t = \frac{1}{1 + \Delta t / t_*}$$

Assuming elastic behavior, the trial stress tensor $\hat{\boldsymbol{\sigma}}$ is computed using the strain increment:

$$\hat{\boldsymbol{\sigma}}^{t+\Delta t} = \boldsymbol{\sigma}^t + c^{t+\Delta t} : \Delta \boldsymbol{\epsilon}^{t+\Delta t}$$

The tangent stiffness tensor is updated using the following equivalent approximation:

$$c^{t+\Delta t} = r_t c^e + (1 - r_t) c^t$$

Here, c^e is the elastic stiffness tensor.

The name of the viscoplastic solver is formed by adding the prefix *Visco* to the name of the elasto-plastic solver used to compute the inviscid stress $\overline{\boldsymbol{\sigma}}$. For example, the solver *Visco Drucker-Prager* corresponds to cases where the inviscid stress is computed by the *Drucker-Prager* solver. It is interesting to note that equivalent viscoelastic solutions can also be obtained using the Duvault-Lions algorithm by updating the inviscid stress with an elastic solver.

Fluid Models

These models provide density, viscosity, and composition relationships for single fluids and fluid mixtures.

Compressible single phase fluid model

Overview

This model represents a compressible single-phase fluid with constant compressibility and pressure-dependent viscosity. These assumptions are valid for slightly compressible fluids, such as water, and some types of oil with negligible amounts of dissolved gas.

Specifically, fluid density is computed as

$$\rho(p) = \rho_0 e^{c_\rho (p - p_0)}$$

where c_ρ is compressibility, p_0 is reference pressure, ρ_0 is density at reference pressure. Similarly,

$$\mu(p) = \mu_0 e^{c_\mu (p - p_0)}$$

where c_μ is viscosibility (viscosity compressibility), μ_0 is reference viscosity.

Either exponent may be approximated by linear (default) or quadratic terms of Taylor series expansion. Currently there is no temperature dependence in the model, although it may be added in future.

Parameters

The model is represented by `<CompressibleSinglePhaseFluid>` node in the input.

The following attributes are supported:

Name	Type	Default	Description
compressibility	real64	0	Fluid compressibility
defaultDensity	real64	required	Default value for density.
defaultViscosity	real64	required	Default value for viscosity.
densityModelType	geos_constitutive_Exponent	linear	Type of density model. Valid options: * exponential * linear * quadratic
name	string	required	A name is required for any non-unique nodes
referenceDensity	real64	1000	Reference fluid density
referencePressure	real64	0	Reference pressure
referenceViscosity	real64	0.001	Reference fluid viscosity
viscosibility	real64	0	Fluid viscosity exponential coefficient
viscosityModelType	geos_constitutive_Exponent	linear	Type of viscosity model. Valid options: * exponential * linear * quadratic

Example

```
<Constitutive>
  <CompressibleSinglePhaseFluid name="water"
                                referencePressure="2.125e6"
                                referenceDensity="1000"
                                compressibility="1e-19"
                                referenceViscosity="0.001"
                                viscosibility="0.0"/>
</Constitutive>
```

Black-oil fluid model

Overview

In the black-oil model three pseudo-components, oil (o), gas (g) and water (w) are considered. These are assumed to be partitioned across three fluid phases, named liquid (l), vapor (v) and aqueous (a).

Phase behavior is characterized by the following quantities which are used to relate properties of the fluids in the reservoir to their properties at surface conditions.

- B_o: oil formation volume factor

- B_g: gas formation volume factor
- R_s: gas/oil ratio
- R_v: oil/gas ratio

By tables, that tabulate saturated and undersaturated oil and gas properties as functions of pressure and solution ratios.

Dead oil

In **dead-oil** each component occupies only one phase. Thus, the following partition matrix determines the components distribution within the three phases:

$$\begin{bmatrix} y_{gv} & y_{gl} & y_{ga} \\ y_{ov} & y_{ol} & y_{oa} \\ y_{wv} & y_{wl} & y_{wa} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

and the phase densities are

$$\begin{aligned} \rho_l &= \frac{\rho_o^{STC}}{B_o} \\ \rho_v &= \frac{\rho_g^{STC}}{B_g}. \end{aligned}$$

Live oil

The live oil fluid model make no assumptions about the partitioning of the hydrocarbon components and the following composition matrix can be used

$$\begin{bmatrix} y_{gv} & y_{gl} & y_{ga} \\ y_{ov} & y_{ol} & y_{oa} \\ y_{wv} & y_{wl} & y_{wa} \end{bmatrix} = \begin{bmatrix} \frac{\rho_g^{STC}}{\rho_g^{STC} + \rho_o^{STC} r_s} & \frac{\rho_g^{STC} R_s}{\rho_o^{STC} + \rho_g^{STC} R_s} & 0 \\ \frac{\rho_o^{STC} r_s}{\rho_g^{STC} + \rho_o^{STC} r_s} & \frac{\rho_o^{STC}}{\rho_o^{STC} + \rho_g^{STC} R_s} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

whereas the densities of the two hydrocarbon phases are

$$\begin{aligned} \rho_l &= \frac{\rho_o^{STC} + \rho_g^{STC} R_s}{B_o} \\ \rho_v &= \frac{\rho_g^{STC} + \rho_o^{STC} R_v}{B_g} \end{aligned}$$

See Petrowiki for more information.

Parameters

Both types are represented by `<BlackOilFluid>` node in the input. Under the hood this is a wrapper around `PVTPackage` library, which is included as a submodule. In order to use the model, GEOS must be built with `-DENABLE_PVTPACKAGE=ON` (default).

The following attributes are supported:

Name	Type	Default	Description
componentMolarWeight	real64_array	required	Component molar weights
componentNames	string_array	{}	List of component names
hydrocarbonFormationVolFactorTableNames	string_array	{}	List of formation volume factor TableFunction names from the Functions block. The user must provide one TableFunction per hydrocarbon phase, in the order provided in "phaseNames". For instance, if "oil" is before "gas" in "phaseNames", the table order should be: oilTableName, gasTableName
hydrocarbonViscosityTableNames	string_array	{}	List of viscosity TableFunction names from the Functions block. The user must provide one TableFunction per hydrocarbon phase, in the order provided in "phaseNames". For instance, if "oil" is before "gas" in "phaseNames", the table order should be: oilTableName, gasTableName
name	string	required	A name is required for any non-unique nodes
phaseNames	string_array	required	List of fluid phases
surfaceDensities	real64_array	required	List of surface mass densities for each phase
tableFiles	path_array	{}	List of filenames with input PVT tables (one per phase)
waterCompressibility	real64	0	Water compressibility
waterFormationVolumeFactor	real64	0	Water formation volume factor
waterReferencePressure	real64	0	Water reference pressure
waterViscosity	real64	0	Water viscosity

Supported phase names are:

Value	Comment
oil	Oil phase
gas	Gas phase
water	Water phase

Example

```
<Constitutive>
  <BlackOilFluid name="fluid1"
                 fluidType="LiveOil"
                 phaseNames="{ oil, gas, water }"
                 surfaceDensities="{ 800.0, 0.9907, 1022.0 }"
                 componentMolarWeight="{ 114e-3, 16e-3, 18e-3 }"
                 tableFiles="{ pvto.txt, pvtg.txt, pvtw.txt }"/>
</Constitutive>
```

Compositional multiphase fluid model

Overview

This model represents a full composition description of a multiphase multicomponent fluid. Phase behavior is modeled by an Equation of State (EOS) and partitioning of components into phases is computed based on instantaneous chemical equilibrium via a two- or three-phase flash. Each component (species) is characterized by molar weight and critical properties that serve as input parameters for the EOS. See Petrowiki for more information.

Parameters

The model represented by <CompositionalMultiphaseFluid> node in the input. Under the hood this is a wrapper around PVTPackage library, which is included as a submodule. In order to use the model, GEOS must be built with -DENABLE_PVTPACKAGE=ON (default).

The following attributes are supported:

Name	Type	Default	Description
componentAcentricFactor	real64_array	required	Component acentric factors
componentBinaryCoeff	real64_array2d	{{0}}	Table of binary interaction coefficients
componentCriticalPressure	real64_array	required	Component critical pressures
componentCriticalTemperature	real64_array	required	Component critical temperatures
componentMolarWeight	real64_array	required	Component molar weights
componentNames	string_array	required	List of component names
componentVolumeShift	real64_array	{0}	Component volume shifts
equationsOfState	string_array	required	List of equation of state types for each phase
name	string	required	A name is required for any non-unique nodes
phaseNames	string_array	required	List of fluid phases

Supported phase names are:

Value	Comment
oil	Oil phase
gas	Gas phase
water	Water phase

Supported Equation of State types:

Value	Comment
PR	Peng-Robinson EOS
SRK	Soave-Redlich-Kwong EOS

Example

```
<Constitutive>
  <CompositionalMultiphaseFluid name="fluid1"
                                phaseNames="{ oil, gas }"
                                equationsOfState="{ PR, PR }"
                                componentNames="{ N2, C10, C20, H2O }"
                                componentCriticalPressure="{ 34e5, 25.3e5, 14.6e5, 220.
↪5e5 }"
                                componentCriticalTemperature="{ 126.2, 622.0, 782.0, 647.
↪0 }"
                                componentAcentricFactor="{ 0.04, 0.443, 0.816, 0.344 }"
                                componentMolarWeight="{ 28e-3, 134e-3, 275e-3, 18e-3 }"
                                componentVolumeShift="{ 0, 0, 0, 0 }"
                                componentBinaryCoeff="{ { 0, 0, 0, 0 },
                                                      { 0, 0, 0, 0 },
                                                      { 0, 0, 0, 0 },
                                                      { 0, 0, 0, 0 } }"/>
</Constitutive>
```

CO2-brine model

Summary

The CO2-brine model implemented in GEOS includes two components (CO2 and H2O) that are transported by one or two fluid phases (the brine phase and the CO2 phase). We refer to the brine phase with the subscript ℓ and to the CO2 phase with the subscript g (although the CO2 phase can be in supercritical, liquid, or gas state). The water component is only present in the brine phase, while the CO2 component can be present in the CO2 phase as well as in the brine phase. Thus, considering the molar phase component fractions, $y_{c,p}$ (i.e., the fraction of the molar mass of phase p represented by component c) the following partition matrix determines the component distribution within the two phases:

$$\begin{bmatrix} y_{H2O,\ell} & y_{CO2,\ell} \\ 0 & 1 \end{bmatrix}$$

The update of the fluid properties is done in two steps:

1) The phase fractions (ν_p) and phase component fractions ($y_{c,p}$) are computed as a function of pressure (p), temperature (T), component fractions (z_c), and a constant salinity.

2) The phase densities (ρ_p) and phase viscosities (μ_p) are computed as a function of pressure, temperature, the updated phase component fractions, and a constant salinity.

Once the phase fractions, phase component fractions, phase densities, phase viscosities–and their derivatives with respect to pressure, temperature, and component fractions–have been computed, the *Compositional Multiphase Flow Solver* proceeds to the assembly of the accumulation and flux terms. Note that the current implementation of the flow solver is isothermal and that the derivatives with respect to temperature are therefore discarded.

The models that are used in steps 1) and 2) are reviewed in more details below.

Step 1: Computation of the phase fractions and phase component fractions (flash)

At initialization, GEOS performs a preprocessing step to construct a two-dimensional table storing the values of CO2 solubility in brine as a function of pressure, temperature, and a constant salinity. The user can parameterize the construction of the table by specifying the salinity and by defining the pressure (p) and temperature (T) axis of the table in the form:

FlashModel	CO2Solubility	p_{min}	p_{max}	Δp	T_{min}	T_{max}	ΔT	Salinity

Note that the pressures are in Pascal, temperatures are in Kelvin, and the salinity is a molality (moles of NaCl per kg of brine). The temperature must be between 283.15 and 623.15 Kelvin. The table is populated using the model of Duan and Sun (2003). Specifically, we solve the following nonlinear CO2 equation of state (equation (A1) in Duan and Sun, 2003) for each pair (p, T) to obtain the reduced volume, V_r.

$$\frac{p_r V_r}{T_r} = 1 + \frac{a_1 + a_2/T_r^2 + a_3/T_r^3}{V_r} + \frac{a_4 + a_5/T_r^2 + a_6/T_r^3}{V_r^2} + \frac{a_7 + a_8/T_r^2 + a_9/T_r^3}{V_r^4}$$
$$+ \frac{a_{10} + a_{11}/T_r^2 + a_{12}/T_r^3}{V_r^5} + \frac{a_{13}}{T_r^3 V_r^2}\big(a_{14} + \frac{a_{15}}{V_r^2}\big) \exp(-\frac{a_{15}}{V_r^2})$$

where $p_r = p/p_{crit}$ and $T_r = T/T_{crit}$ are respectively the reduced pressure and the reduced temperature. We refer the reader to Table (A1) in Duan and Sun (2003) for the definition of the coefficients a_i involved in the previous equation. Using the reduced volume, V_r, we compute the fugacity coefficient of CO2, $\ln_\phi(p, T)$, using equation (A6) of Duan and Sun (2003). To conclude this preprocessing step, we use the fugacity coefficient of CO2 to compute and store the solubility of CO2 in brine, s_{CO2}, using equation (6) of Duan and Sun (2003):

$$\ln \frac{y_{CO2} P}{s_{CO2}} = \frac{\Phi_{CO2}}{RT} - \ln_\phi(p, T) + \sum_c 2\lambda_c m + \sum_a 2\lambda_a m + \sum_{a,c} \zeta_{a,c} m^2$$

where Φ_{CO2} is the chemical potential of the CO2 component, R is the gas constant, and m is the salinity. The mole fraction of CO2 in the vapor phase, y_{CO2}, is computed with equation (4) of Duan and Sun (2003). Note that the first, third, fourth, and fifth terms in the equation written above are approximated using equation (7) of Duan and Sun (2003) as recommended by the authors.

During the simulation, Step 1 starts with a look-up in the precomputed table to get the CO2 solubility, s_{CO2}, as a function of pressure and temperature. Then, we compute the phase fractions as:

$$\nu_\ell = \frac{1 + s_{CO2}}{1 + z_{CO2}/(1 - z_{CO2})}$$
$$\nu_g = 1 - \nu_\ell$$

We conclude Step 1 by computing the phase component fractions as:

$$y_{CO2,\ell} = \frac{s_{CO2}}{1 + s_{CO2}}$$
$$y_{H2O,\ell} = 1 - y_{CO2,\ell}$$
$$y_{CO2,g} = 1$$
$$y_{H2O,g} = 0$$

Step 2: Computation of the phase densities and phase viscosities

CO2 phase density and viscosity

In GEOS, the computation of the CO2 phase density and viscosity is entirely based on look-up in precomputed tables. The user defines the pressure (in Pascal) and temperature (in Kelvin) axis of the density table in the form:

DensityFun	SpanWagnerCO2Density	p_{min}	p_{max}	Δp	T_{min}	T_{max}	ΔT

This correlation is valid for pressures less than 8×10^8 Pascal and temperatures less than 1073.15 Kelvin. Using these parameters, GEOS internally constructs a two-dimensional table storing the values of density as a function of pressure and temperature. This table is populated as explained in the work of Span and Wagner (1996) by solving the following nonlinear Helmholtz energy equation for each pair (p, T) to obtain the value of density, ρ_g:

$$\frac{p}{RT\rho_g} = 1 + \delta\phi^r_{\delta}(\delta, \tau)$$

where R is the gas constant, $\delta := \rho_g/\rho_{crit}$ is the reduced CO2 phase density, and $\tau := T_{crit}/T$ is the inverse of the reduced temperature. The definition of the residual part of the energy equation, denoted by ϕ^r_{δ}, can be found in equation (6.5), page 1544 of Span and Wagner (1996). The coefficients involved in the computation of ϕ^r_{δ} are listed in Table (31), page 1544 of Span and Wagner (1996). These calculations are done in a preprocessing step.

The pressure and temperature axis of the viscosity table can be parameterized in a similar fashion using the format:

ViscosityFun	FenghourCO2Viscosity	p_{min}	p_{max}	Δp	T_{min}	T_{max}	ΔT

This correlation is valid for pressures less than 3×10^8 Pascal and temperatures less than 1493.15 Kelvin. This table is populated as explained in the work of Fenghour and Wakeham (1998) by computing the CO2 phase viscosity, μ_g, as follows:

$$\mu_g = \mu_0(T) + \mu_{excess}(\rho_g, T) + \mu_{crit}(\rho_g, T)$$

The “zero-density limit” viscosity, $\mu_0(T)$, is computed as a function of temperature using equations (3), (4), and (5), as well as Table (1) of Fenghour and Wakeham (1998). The excess viscosity, $\mu_{excess}(\rho_g, T)$, is computed as a function of temperature and CO2 phase density (computed as explained above) using equation (8) and Table (3) of Fenghour and Wakeham (1998). We currently neglect the critical viscosity, μ_{crit}. These calculations are done in a preprocessing step.

During the simulation, the update of CO2 phase density and viscosity is simply done with a look-up in the precomputed tables.

Brine density and viscosity using Phillips correlation

The computation of the brine density involves a tabulated correlation presented in Phillips et al. (1981). The user specifies the (constant) salinity and defines the pressure and temperature axis of the brine density table in the form:

DensityFun	PhillipsBrineDensity	p_{min}	p_{max}	Δp	T_{min}	T_{max}	ΔT	Salinity

The pressure must be in Pascal and must be less than 5×10^7 Pascal. The temperature must be in Kelvin and must be between 283.15 and 623.15 Kelvin. The salinity is a molality (moles of NaCl per kg of brine). Using these parameters, GEOS performs a preprocessing step to construct a two-dimensional table storing the brine density, $\rho_{\ell,table}$ for the specified salinity as a function of pressure and temperature using the expression:

$$\begin{aligned}\rho_{\ell,table} &= A + Bx + Cx^2 + Dx^3 \\ x &= c_1 \exp(a_1 m) + c_2 \exp(a_2 T) + c_3 \exp(a_3 P)\end{aligned}$$

We refer the reader to Phillips et al. (1981), equations (4) and (5), pages 14 and 15 for the definition of the coefficients involved in the previous equation. This concludes the preprocessing step.

Then, during the simulation, the brine density update proceeds in two steps. First, a table look-up is performed to retrieve the value of density, $\rho_{\ell,table}$. Then, in a second step, the density is modified using the method of Garcia (2001) to account for the presence of CO2 dissolved in brine as follows:

$$\rho_{\ell} = \rho_{\ell,table} + M_{CO2} c_{CO2} - c_{CO2} \rho_{\ell,table} V_{\phi}$$

where M_{CO2} is the molecular weight of CO2, c_{CO2} is the concentration of CO2 in brine, and V_{ϕ} is the apparent molar volume of dissolved CO2. The CO2 concentration in brine is obtained as:

$$c_{CO2} = \frac{y_{CO2,\ell} \rho_{\ell,table}}{M_{H2O}(1 - y_{CO2,\ell})}$$

where M_{H2O} is the molecular weight of water. The apparent molar volume of dissolved CO2 is computed as a function of temperature using the expression:

$$V_{\phi} = 37.51 - 9.585 \times 10^{-2} T + 8.740 \times 10^{-4} T^2 - 5.044 \times 10^{-7} T^3$$

The brine viscosity is controlled by a salinity parameter provided by the user in the form:

ViscosityFun	PhillipsBrineViscosity	Salinity

During the simulation, the brine viscosity is updated as a function of temperature using the analytical relationship of Phillips et al. (1981):

$$\mu_{\ell} = aT + b$$

where the coefficients a and b are defined as:

$$\begin{aligned}a &= \mu_w(T) \times 0.000629(1.0 - \exp(-0.7m)) \\ b &= \mu_w(T)(1.0 + 0.0816m + 0.0122m^2 + 0.000128m^3)\end{aligned}$$

where μ_w is the pure water viscosity computed as a function of temperature, and m is the user-defined salinity (in moles of NaCl per kg of brine).

Brine density and viscosity using Ezrokhi correlation

Brine density ρ_l is computed from pure water density ρ_w at specified pressure and temperature corrected by Ezrokhi correlation presented in Zaytsev and Aseyev (1993):

$$\begin{aligned}log_{10}(\rho_l) &= log_{10}(\rho_w(P,T)) + A(T) x_{CO2,\ell} \\ A(T) &= a_0 + a_1 T + a_2 T^2,\end{aligned}$$

where a_0, a_1, a_2 are correlation coefficients defined by user:

DensityFun	EzrokhiBrineDensity	a_0	a_1	a_2

While $x_{CO2,\ell}$ is mass fraction of CO2 component in brine, computed from molar fractions as

$$x_{CO2,\ell} = \frac{M_{CO2} y_{CO2,\ell}}{M_{CO2} y_{CO2,\ell} + M_{H2O} y_{H2O,\ell}},$$

Pure water density is computed according to:

$$\rho_w = \rho_{w,sat}(T) e^{c_w * (P - P_{w,sat}(T))},$$

where c_w is water compressibility defined as a constant $4.5 \times 10^{-10} Pa^{-1}$, while $\rho_{w,sat}(T)$ and $P_{w,sat}(T)$ are density and pressure of saturated water at a given temperature. Both are obtained through internally constructed tables tabulated as functions of temperature and filled with the steam table data from Engineering ToolBox (2003, 2004).

Brine viscosity μ_ℓ is computed from pure water viscosity μ_w similarly:

$$log_{10}(\mu_l) = log_{10}(\mu_w(P,T)) + B(T) x_{CO2,\ell}$$
$$B(T) = b_0 + b_1 T + b_2 T^2,$$

where b_0, b_1, b_2 are correlation coefficients defined by user:

ViscosityFun	EzrokhiBrineViscosity	b_0	b_1	b_2

Mass fraction of CO2 component in brine $x_{CO2,\ell}$ is exactly as in density calculation. The dependency of pure water viscosity from pressure is ignored, and it is approximated as saturated pure water viscosity:

$$\mu_w(P,T) = \mu_{w,sat}(T),$$

which is tabulated using internal table as a function of temperature based on steam table data Engineering ToolBox (2004).

Parameters

The models are represented by `<CO2BrinePhillipsFluid>`, `<CO2BrineEzrokhiFluid>` nodes in the input.

The following attributes are supported:

Name	Type	Default	Description
componentMolarWeight	real64_array	{0}	Component molar weights
componentNames	string_array	{}	List of component names
flashModelParaFile	path	required	Name of the file defining the parameters of the flash model
name	string	required	A name is required for any non-unique nodes
phaseNames	string_array	{}	List of fluid phases
phasePVTParaFiles	path_array	required	Names of the files defining the parameters of the viscosity and density models

Supported phase names are:

Value	Comment
gas	CO2 phase
water	Water phase

Supported component names are:

Value	Component
co2,CO2	CO2 component
water,liquid	Water component

Example

```xml
<Constitutive>
    <CO2BrinePhillipsFluid
      name="fluid"
      phaseNames="{ gas, water }"
      componentNames="{ co2, water }"
      componentMolarWeight="{ 44e-3, 18e-3 }"
      phasePVTParaFiles="{ pvtgas.txt, pvtliquid.txt }"
      flashModelParaFile="co2flash.txt"/>
</Constitutive>
```

```xml
<Constitutive>
    <CO2BrineEzrokhiFluid
      name="fluid"
      phaseNames="{ gas, water }"
      componentNames="{ co2, water }"
      componentMolarWeight="{ 44e-3, 18e-3 }"
      phasePVTParaFiles="{ pvtgas.txt, pvtliquid.txt }"
      flashModelParaFile="co2flash.txt"/>
</Constitutive>
```

In the XML code listed above, “co2flash.txt” parameterizes the CO2 solubility table constructed in Step 1. The file “pvtgas.txt” parameterizes the CO2 phase density and viscosity tables constructed in Step 2, the file “pvtliquid.txt” parameterizes the brine density and viscosity tables according to Phillips or Ezrokhi correlation, depending on chosen fluid model.

References

- Z. Duan and R. Sun, An improved model calculating CO2 solubility in pure water and aqueous NaCl solutions from 273 to 533 K and from 0 to 2000 bar., Chemical Geology, vol. 193.3-4, pp. 257-271, 2003.
- R. Span and W. Wagner, A new equation of state for carbon dioxide covering the fluid region from the triple-point temperature to 1100 K at pressure up to 800 MPa, J. Phys. Chem. Ref. Data, vol. 25, pp. 1509-1596, 1996.
- A. Fenghour and W. A. Wakeham, The viscosity of carbon dioxide, J. Phys. Chem. Ref. Data, vol. 27, pp. 31-44, 1998.
- S. L. Phillips et al., A technical databook for geothermal energy utilization, Lawrence Berkeley Laboratory report, 1981.
- J. E. Garcia, Density of aqueous solutions of CO2. No. LBNL-49023. Lawrence Berkeley National Laboratory, Berkeley, CA, 2001.
- Zaytsev, I.D. and Aseyev, G.G. Properties of Aqueous Solutions of Electrolytes, Boca Raton, Florida, USA CRC Press, 1993.

- Engineering ToolBox, Water - Density, Specific Weight and Thermal Expansion Coefficients, 2003
- Engineering ToolBox, Water - Dynamic (Absolute) and Kinematic Viscosity, 2004

PVT Driver

Table of Contents

Introduction

When calibrating fluid material parameters to experimental or other reference data, it can be a hassle to launch a full flow simulation just to confirm density, viscosity, and other fluid properties are behaving as expected. Instead, GEOS provides a `PVTDriver` allowing the user to test fluid property models for a well defined set of pressure, temperature, and composition conditions. The driver itself is launched like any other GEOS simulation, but with a particular XML structure:

```
./bin/geosx -i myFluidTest.xml
```

This driver will work for any multi-phase fluid model (e.g. black-oil, co2-brine, compositional multiphase) enabled within GEOS.

XML Structure

A typical XML file to run the driver will have several key elements. Here, we will walk through an example file included in the source tree at

```
src/coreComponents/unitTests/constitutiveTests/testPVT_docExample.xml
```

The first thing to note is that the XML file structure is identical to a standard GEOS input deck. In fact, once the constitutive block is calibrated, one could start adding solver and discretization blocks to the same file to create a proper field simulation. This makes it easy to go back and forth between calibration and simulation.

The first step is to define a parameterized fluid model to test. Here, we create a particular type of CO2-Brine mixture:

```
  <Constitutive>
    <CO2BrinePhillipsFluid
      name="co2Mixture"
      phaseNames="{ gas, water }"
      componentNames="{ co2, water }"
      componentMolarWeight="{ 44e-3, 18e-3 }"
```

(continues on next page)

(continued from previous page)

```
    phasePVTParaFiles="{ testPVT_data/carbonDioxidePVT.txt, testPVT_data/brinePVT.txt }
↪"
    flashModelParaFile="testPVT_data/carbonDioxideFlash.txt"/>
</Constitutive>
```

We also define two time-history functions for the pressure (Pascal units) and temperature (Kelvin units) conditions we want to explore.

```
<Functions>
  <TableFunction
    name="pressureFunction"
    inputVarNames="{ time }"
    coordinates="{ 0.0, 1.0 }"
    values="{ 1e6, 50e6 }"/>

  <TableFunction
    name="temperatureFunction"
    inputVarNames="{ time }"
    coordinates="{ 0.0, 1.0 }"
    values="{ 350, 350 }"/>
</Functions>
```

Note that the time-axis here is just a pseudo-time, allowing us to parameterize arbitrarily complicated paths through a (pressure,temperature) diagram. The actual time values have no impact on the resulting fluid properties. Here, we fix the temperature at 350K and simply ramp up pressure from 1 MPa to 50 MPa:

A PVTDriver is then added as a Task, a particular type of executable event often used for simple actions.

```
<Tasks>
  <PVTDriver
    name="testCO2"
    fluid="co2Mixture"
    feedComposition="{ 1.0, 0.0 }"
    pressureControl="pressureFunction"
    temperatureControl="temperatureFunction"
    steps="49"
    output="pvtOutput.txt"
    logLevel="1"/>
</Tasks>
```

The driver itself takes as input the fluid model, the pressure and temperature control functions, and a "feed composition." The latter is the mole fraction of each component in the mixture to be tested. The steps parameter controls how many steps are taken along the parametric (P,T) path. Results will be written in a simple ASCII table format (described below) to the file output. The logLevel parameter controls the verbosity of log output during execution.

The driver task is added as a SoloEvent to the event queue. This leads to a trivial event queue, since all we do is launch the driver and then quit.

```
<Events
  maxTime="1">
  <SoloEvent
    name="eventA"
    target="/Tasks/testCO2"/>
</Events>
```

Internally, the driver uses a simple form of time-stepping to advance through the (P,T) steps. This timestepping is handled independently of the more complicated time-stepping pattern used by physics `Solvers` and coordinated by the `EventManager`. In particular, in the XML file above, the `maxTime` parameter in the `Events` block is an event manager control, controlling when/if certain events occur. Once launched, the PVTDriver internally determines its own max time and timestep size using a combination of the input functions' time coordinates and the requested number of loadsteps. It is therefore helpful to think of the driver as an instantaneous *event* (from the event manager's point of view), but one which has a separate, internal clock.

Parameters

The key XML parameters for the PVTDriver are summarized in the following table:

Name	Type	Default	Description
baseline	path	none	Baseline file
feedComposition	real64_array	required	Feed composition array [mol fraction]
fluid	string	required	Fluid to test
logLevel	integer	0	Log level
name	string	required	A name is required for any non-unique nodes
output	string	none	Output file
pressureControl	string	required	Function controlling pressure time history
steps	integer	required	Number of load steps to take
temperatureControl	string	required	Function controlling temperature time history

Output Format

The `output` key is used to identify a file to which the results of the simulation are written. If this key is omitted, or the user specifies `output="none"`, file output will be suppressed. The file is a simple ASCII format with a brief header followed by test data:

```
# column 1 = time
# column 2 = pressure
# column 3 = temperature
# column 4 = density
# columns 5-6 = phase fractions
# columns 7-8 = phase densities
# columns 9-10 = phase viscosities
0.0000e+00 1.0000e+06 3.5000e+02 1.5581e+01 1.0000e+00 4.1138e-11 1.5581e+01 1.0033e+03␣
↪1.7476e-05 9.9525e-04
2.0408e-02 2.0000e+06 3.5000e+02 3.2165e+01 1.0000e+00 4.1359e-11 3.2165e+01 1.0050e+03␣
↪1.7601e-05 9.9525e-04
4.0816e-02 3.0000e+06 3.5000e+02 4.9901e+01 1.0000e+00 4.1563e-11 4.9901e+01 1.0066e+03␣
↪1.7778e-05 9.9525e-04
...
```

Note that the number of columns will depend on how may phases are present. In this case, we have a two-phase, two-component mixture. The total density is reported in column 4, while phase fractions, phase densities, and phase viscosities are reported in subsequent columns. The phase order will match the one defined in the input XML (here, the co2-rich phase followed by the water-rich phase). This file can be readily plotted using any number of plotting tools. Each row corresponds to one timestep of the driver, starting from initial conditions in the first row.

Unit Testing

The development team also uses the PVTDriver to perform unit testing on the various fluid models within GEOS. The optional argument `baseline` can be used to point to a previous output file that has been validated (e.g. against experimental benchmarks or reference codes). If such a file is specified, the driver will perform a testing run and then compare the new results against the baseline. In this way, any regressions in the fluid models can be quickly identified.

Developers of new models are encouraged to add their own baselines to `src/coreComponents/constitutive/unitTests`. Adding additional tests is straightforward:

1. Create a new xml file for your test in `src/coreComponents/constitutive/unitTests` or (easier) add extra blocks to the existing XML at `src/coreComponents/constitutive/unitTests/testPVT.xml`. For new XMLs, we suggest using the naming convention `testPVT_myTest.xml`, so that all tests will be grouped together alphabetically. Set the `output` file to `testPVT_myTest.txt`, and run your test. Validate the results however is appropriate. If you have reference data available for this validation, we suggest archiving it in the `testPVT_data/` subdirectory, with a description of the source and formatting in the file header. Several reference datasets are included already as examples. This directory is also a convenient place to store auxiliary input files like PVT tables.

2. This output file will now become your new baseline. Replace the `output` key with `baseline` so that the driver can read in your file as a baseline for comparison. Make sure there is no remaining `output` key, or set `output=none`, to suppress further file output. While you can certainly write a new output for debugging purposes, during our automated unit tests we prefer to suppress file output. Re-run the driver to confirm that the comparison test passes.

3. Modify `src/coreComponents/constitutive/unitTests/CMakeLists.txt` to enable your new test in the unit test suite. In particular, you will need to add your new XML file to the existing list in the `gtest_pvt_xmls` variable. Note that if you simply added your test to the existing `testPVT.xml` file, no changes are needed.

```
set( gtest_pvt_xmls
     testPVT.xml
     testPVT_myTest.xml
   )
```

4. Run `make` in your build directory to make sure the CMake syntax is correct
5. Run `ctest -V -R PVT` to run the PVT unit tests. Confirm your test is included and passes properly.

If you run into troubles, do not hesitate to contact the development team for help.

Relative Permeability Models

There are two ways to specify relative permeabilities in GEOS. The user can either select an analytical relative permeability model (e.g., Brooks-Corey or Van Genuchten) or provide relative permeability tables. This is explained in the following sections.

Brooks-Corey relative permeability model

Overview

The following paragraphs explain how the Brooks-Corey model is used to compute the phase relative permeabilities as a function of volume fraction (i.e., saturation) with the expression:

$$k_{r\ell} = k_{r\ell,max} S_{\ell,scaled}^{\lambda_\ell},$$

where the scaled volume fraction of phase ℓ is computed as:

$$S_{\ell,scaled} = \frac{S_\ell - S_{\ell,min}}{1 - \sum_{m=1}^{n_p} S_{m,min}}.$$

The minimum phase volume fractions $S_{\ell,min}$ are model parameters specified by the user.

Parameters

The relative permeability constitutive model is listed in the <`Constitutive`> block of the input XML file. The relative permeability model must be assigned a unique name via `name` attribute. This name is used to assign the model to regions of the physical domain via a `materialList` attribute of the <`ElementRegions`> node.

The following attributes are supported:

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes
phaseMinVolumeFraction	real64_array	{0}	Minimum volume fraction value for each phase
phaseNames	string_array	required	List of fluid phases
phaseRelPermExponent	real64_array	{1}	Minimum relative permeability power law exponent for each phase
phaseRelPermMaxValue	real64_array	{0}	Maximum relative permeability value for each phase

Below are some comments on the model parameters.

- `phaseNames` - The number of phases can be either two or three. Note that for three-phase flow, this model does not apply a special treatment to the intermediate phase relative permeability (no Stone or Baker interpolation). Supported phase names are:

Value	Phase
oil	Oil phase
gas	Gas phase
water	Water phase

- `phaseMinVolFraction` - The list of minimum volume fractions $S_{\ell,min}$ for each phase is specified in the same order as in `phaseNames`. Below this volume fraction, the phase is assumed to be immobile.
- `phaseRelPermExponent` - The list of exponents λ_ℓ for each phase is specified in the same order as in `phaseNames`.
- `phaseMaxValue` - The list of maximum values $k_{r\ell,max}$ for each phase is specified in the same order as in `phaseNames`.

Examples

For a two-phase water-gas system (for instance in the CO2-brine fluid model), a typical relative permeability input looks like:

```
<Constitutive>
   ...
   <BrooksCoreyRelativePermeability
     name="relPerm"
     phaseNames="{ water, gas }"
     phaseMinVolumeFraction="{ 0.02, 0.015 }"
     phaseRelPermExponent="{ 2, 2.5 }"
     phaseRelPermMaxValue="{ 0.8, 1.0 }"/>
   ...
</Constitutive>
```

For a three-phase oil-water-gas system (for instance in the Black-Oil fluid model), a typical relative permeability input looks like:

```
<Constitutive>
   ...
   <BrooksCoreyRelativePermeability
     name="relPerm"
     phaseNames="{ water, oil, gas }"
     phaseMinVolumeFraction="{ 0.02, 0.1, 0.015 }"
     phaseRelPermExponent="{ 2, 2, 2.5 }"
     phaseRelPermMaxValue="{ 0.8, 1.0, 1.0 }"/>
   ...
</Constitutive>
```

Three-phase relative permeability model

Overview

For the simulation of three-phase flow in porous media, it is common to use a specific treatment (i.e., different from the typical two-phase procedure) to evaluate the oil relative permeability. Specifically, the three-phase oil relative permeability is obtained by interpolation of oil-water and oil-gas experimental data measured independently in two-phase displacements.

Let $k_{rw,wo}$ and $k_{ro,wo}$ be the water-oil two-phase relative permeabilities for the water phase and the oil phase, respectively. Let $k_{rg,go}$ and $k_{ro,go}$ be the oil-gas two-phase relative permeabilities for the gas phase and the oil phase, respectively. In the current implementation, the two-phase relative permeability data is computed analytically using the *Brooks-Corey relative permeability model*.

The water and gas three-phase relative permeabilities are simply given by two-phase data and only depend on S_w and S_g, respectively. That is,

$$k_{rw,wog}(S_w) = k_{rw,wo}(S_w),$$

$$k_{rg,wog}(S_g) = k_{rg,go}(S_g).$$

The oil three-phase relative permeability is obtained using a variant of the saturation-weighted interpolation procedure initially proposed by Baker. Specifically, we compute:

$$k_{ro,wog}(S_w, S_g) = \frac{(S_w - S_{w,min})k_{ro,wo}(S_w) + S_g k_{rg,go}(S_g)}{(S_w - S_{w,min}) + S_g}.$$

This procedure provides a simple but effective formula avoiding the problems associated with the other interpolation methods (negative values).

Parameters

The relative permeability constitutive model is listed in the <Constitutive> block of the input XML file. The relative permeability model must be assigned a unique name via name attribute. This name is used to assign the model to regions of the physical domain via a materialList attribute of the <ElementRegion> node.

The following attributes are supported:

Name	Type	Default	Description
gasOilRelPermExponent	real64_array	{1}	Rel perm power law exponent for the pair (gas phase, oil phase) at residual water saturation The expected format is "{ gasExp, oilExp }", in that order
gasOilRelPermMaxValue	real64_array	{0}	Maximum rel perm value for the pair (gas phase, oil phase) at residual water saturation The expected format is "{ gasMax, oilMax }", in that order
name	string	required	A name is required for any non-unique nodes
phaseMinVolumeFraction	real64_array	{0}	Minimum volume fraction value for each phase
phaseNames	string_array	required	List of fluid phases
waterOilRelPermExponent	real64_array	{1}	Rel perm power law exponent for the pair (water phase, oil phase) at residual gas saturation The expected format is "{ waterExp, oilExp }", in that order
waterOilRelPermMaxValue	real64_array	{0}	Maximum rel perm value for the pair (water phase, oil phase) at residual gas saturation The expected format is "{ waterMax, oilMax }", in that order

Below are some comments on the model parameters.

- `phaseNames` - The number of phases should be 3. Supported phase names are:

Value	Phase
oil	Oil phase
gas	Gas phase
water	Water phase

- `phaseMinVolFraction` - The list of minimum volume fractions $S_{\ell,min}$ for each phase is specified in the same order as in `phaseNames`. Below this volume fraction, the phase is assumed to be immobile.
- `waterOilRelPermExponent` - The list of exponents $\lambda_{\ell,wo}$ for the two-phase water-oil relative permeability data, with the water exponent first and the oil exponent next. These exponents are then used to compute $k_{r\ell,wo}$ in the *Brooks-Corey relative permeability model*.
- `waterOilRelPermMaxValue` - The list of maximum values $k_{r\ell,wo,max}$ for the two-phase water-oil relative permeability data, with the water max value first and the oil max value next. These exponents are then used to compute $k_{r\ell,wo}$ in the *Brooks-Corey relative permeability model*.
- `gasOilRelPermExponent` - The list of exponents $\lambda_{\ell,go}$ for the two-phase gas-oil relative permeability data, with the gas exponent first and the oil exponent next. These exponents are then used to compute $k_{r\ell,go}$ in the *Brooks-Corey relative permeability model*.
- `gasOilRelPermMaxValue` - The list of maximum values $k_{r\ell,go,max}$ for the two-phase gas-oil relative permeability data, with the gas max value first and the oil max value next. These exponents are then used to compute $k_{r\ell,go}$ in the *Brooks-Corey relative permeability model*.

Example

```
<Constitutive>
   ...
 <BrooksCoreyBakerRelativePermeability name="relperm"
                                       phaseNames="{oil, gas, water}"
                                       phaseMinVolumeFraction="{0.05, 0.05, 0.05}"
                                       waterOilRelPermExponent="{2.5, 1.5}"
                                       waterOilRelPermMaxValue="{0.8, 0.9}"
                                       gasOilRelPermExponent="{3, 3}"
                                       gasOilRelPermMaxValue="{0.4, 0.9}"/>
   ...
</Constitutive>
```

Table relative permeability

Overview

The user can specify the relative permeabilities using tables describing a piecewise-linear relative permeability function of volume fraction (i.e., saturation) for each phase. Depending on the number of fluid phases, this model is used as follows:

- For two-phase flow, the user must specify two relative permeability tables, that is, one for the wetting-phase relative permeability, and one for the non-wetting phase relative permeability. During the simulation, the relative permeabilities are then obtained by interpolating in the tables as a function of phase volume fraction.
- For three-phase flow, following standard reservoir simulation practice, the user must specify four relative permeability tables. Specifically, two relative permeability tables are required for the pair wetting-phase–intermediate

phase (typically, water-oil), and two relative permeability tables are required for the pair non-wetting-phase–intermediate phase (typically, gas-oil). During the simulation, the relative permeabilities of the wetting and non-wetting phases are computed by interpolating in the tables as a function of their own phase volume fraction. The intermediate phase relative permeability is obtained by interpolating the two-phase relative permeabilities using the Baker interpolation procedure.

Parameters

The relative permeability constitutive model is listed in the `<Constitutive>` block of the input XML file. The relative permeability model must be assigned a unique name via `name` attribute. This name is used to assign the model to regions of the physical domain via a `materialList` attribute of the `<ElementRegions>` node.

The following attributes are supported:

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes
nonWettingIntermediateRelPermTableNames	string_array	{}	List of relative permeability tables for the pair (non-wetting phase, intermediate phase) The expected format is “{ nonWettingPhaseRelPermTableName, intermediatePhaseRelPermTableName }”, in that order Note that this input is only used for three-phase flow. If you want to do a two-phase simulation, please use instead wettingNonWettingRelPermTableNames to specify the table names
phaseNames	string_array	required	List of fluid phases
wettingIntermediateRelPermTableNames	string_array	{}	List of relative permeability tables for the pair (wetting phase, intermediate phase) The expected format is “{ wettingPhaseRelPermTableName, intermediatePhaseRelPermTableName }”, in that order Note that this input is only used for three-phase flow. If you want to do a two-phase simulation, please use instead wettingNonWettingRelPermTableNames to specify the table names
wettingNonWettingRelPermTableNames	string_array	{}	List of relative permeability tables for the pair (wetting phase, non-wetting phase) The expected format is “{ wettingPhaseRelPermTable-

Below are some comments on the model parameters.

- phaseNames - The number of phases can be either two or three. For three-phase flow, this model applies a Baker interpolation to the intermediate phase relative permeability. Supported phase names are:

Value	Phase
oil	Oil phase
gas	Gas phase
water	Water phase

- wettingNonWettingRelPermTableNames - The list of relative permeability table names for two-phase systems, starting with the name of the wetting-phase relative permeability table, followed by the name of the non-wetting phase relative permeability table. Note that this keyword is only valid for two-phase systems, and is not allowed for three-phase systems (for which the user must specify instead wettingIntermediateRelPermTableNames and nonWettingIntermediateRelPermTableNames).
- wettingIntermediateRelPermTableNames - The list of relative permeability table names for the pair wetting-phase–intermediate-phase, starting with the name of the wetting-phase relative permeability table, and continuing with the name of the intermediate phase relative permeability table. Note that this keyword is only valid for three-phase systems, and is not allowed for two-phase systems (for which the user must specify instead wettingNonWettingRelPermTableNames).
- nonWettingIntermediateRelPermTableNames - The list of relative permeability table names for the pair non-wetting-phase–intermediate-phase, starting with the name of the non-wetting-phase relative permeability table, and continuing with the name of the intermediate phase relative permeability table. Note that this keyword is only valid for three-phase systems, and is not allowed for two-phase systems (for which the user must specify instead wettingNonWettingRelPermTableNames).

Note: We remind the user that the relative permeability must be a strictly increasing function of phase volume fraction. GEOS throws an error when this condition is not satisfied.

Examples

For a two-phase water-gas system (for instance in the CO2-brine fluid model), a typical relative permeability input looks like:

```
<Constitutive>
   ...
   <TableRelativePermeability
     name="relPerm"
     phaseNames="{ water, gas }"
     wettingNonWettingRelPermTableNames="{ waterRelativePermeabilityTable,␣
↪gasRelativePermeabilityTable }"/>
   ...
</Constitutive>
```

Note: The name of the wetting-phase relative permeability table must be specified before the name of the non-wetting phase relative permeability table.

For a three-phase oil-water-gas system (for instance in the Black-Oil fluid model), a typical relative permeability input looks like:

```
<Constitutive>
   ...
   <TableRelativePermeability
     name="relPerm"
     phaseNames="{ water, oil, gas }"
     wettingIntermediateRelPermTableNames="{ waterRelativePermeabilityTable,␣
↪oilRelativePermeabilityTableForWO }"
     nonWettingIntermediateRelPermTableNames="{ gasRelativePermeabilityTable,␣
↪oilRelativePermeabilityTableForGO }"/>
   ...
</Constitutive>
```

Note: For the wetting-phase–intermediate-phase pair, the name of the wetting-phase relative permeability table must be specified first. For the non-wetting-phase–intermediate-phase pair, the name of the non-wetting-phase relative permeability table must be specified first. If the results look incoherent, this is something to double-check.

The tables mentioned above by name must be defined in the `<Functions>` block of the XML file using the `<TableFunction>` keyword.

Capillary Pressure Models

There are two ways to specify capillary pressures in GEOS. The user can either select an analytical capillary pressure model (e.g., Brooks-Corey or Van Genuchten) or provide capillary pressure tables. This is explained in the following sections.

Brooks-Corey capillary pressure model

Overview

In GEOS, the oil-phase pressure is assumed to be the primary pressure. The following paragraphs explain how the Brooks-Corey capillary pressure model is used to compute the water-phase and gas-phase pressures as:

$$p_w = p_o - P_{c,w}(S_w),$$

and

$$p_g = p_o + P_{c,g}(S_g).$$

In the Brooks-Corey model, the water-phase capillary pressure is computed as a function of the water-phase volume fraction with the following expression:

$$P_{c,w}(S_w) = p_{e,w} S_{w,scaled}^{-1/\lambda_w},$$

where the scaled water-phase volume fraction is computed as:

$$S_{w,scaled} = \frac{S_w - S_{w,min}}{1 - S_{w,min} - S_{o,min} - S_{g,min}}.$$

The gas capillary pressure is computed analogously.

Parameters

The capillary pressure constitutive model is listed in the <Constitutive> block of the input XML file. The capillary pressure model must be assigned a unique name via name attribute. This name is used to assign the model to regions of the physical domain via a materialList attribute of the <ElementRegions> node.

The following attributes are supported:

Name	Type	Default	Description
capPressureEpsilon	real64	1e-06	Wetting-phase saturation at which the max cap. pressure is attained; used to avoid infinite cap. pressure values for saturations close to zero
name	string	required	A name is required for any non-unique nodes
phaseCapPressureExponentInv	real64_a	{2}	Inverse of capillary power law exponent for each phase
phaseEntryPressure	real64_a	{1}	Entry pressure value for each phase
phaseMinVolumeFraction	real64_a	{0}	Minimum volume fraction value for each phase
phaseNames	string_aı	required	List of fluid phases

Below are some comments on the model parameters:

- phaseNames - The number of phases can be either 2 or 3. The names entered for this attribute should match the phase names specified in the relative permeability block, either in *Brooks-Corey relative permeability model* or in *Three-phase relative permeability model*. The capillary pressure model assumes that oil is always present. Supported phase names are:

Value	Phase
oil	Oil phase
gas	Gas phase
water	Water phase

- phaseMinVolFraction - The list of minimum volume fractions $S_{\ell,min}$ for each phase is specified in the same order as in phaseNames. Below this volume fraction, the phase is assumed to be immobile. The values entered for this attribute have to match those of the same attribute in the relative permeability block.
- phaseCapPressureExponentInv - The list of exponents λ_{ℓ} for each phase is specified in the same order as in phaseNames. The parameter corresponding to the oil phase is currently not used.
- phaseEntryPressure - The list of entry pressures $p_{e,\ell}$ for each phase is specified in the same order as in phaseNames. The parameter corresponding to the oil phase is currently not used.
- capPressureEpsilon - This parameter is used for both the water-phase and gas-phase capillary pressure. To avoid extremely large, or infinite, capillary pressure values, we set $P_{c,w}(S_w) := P_{c,w}(\epsilon)$ whenever $S_w < \epsilon$. The gas-phase capillary pressure is treated analogously.

Example

```
<Constitutive>
   ...
   <BrooksCoreyCapillaryPressure name="capPressure"
                                 phaseNames="{oil, gas}"
                                 phaseMinVolumeFraction="{0.01, 0.015}"
                                 phaseCapPressureExponentInv="{0, 6}"
                                 phaseEntryPressure="{0, 1e8}"
                                 capPressureEpsilon="1e-8"/>
   ...
</Constitutive>
```

Van Genuchten capillary pressure model

Overview

In GEOS, the oil-phase pressure is assumed to be the primary pressure. The following paragraphs explain how the Van Genuchten capillary pressure model is used to compute the water-phase and gas-phase pressures as:

$$p_w = p_o - P_{c,w}(S_w),$$

and

$$p_g = p_o + P_{c,g}(S_g),$$

The Van Genuchten model computes the water-phase capillary pressure as a function of the water-phase volume fraction as:

$$P_c(S_w) = \alpha_w (S_{w,scaled}^{-1/m_w} - 1)^{(1-m_w)/2},$$

where the scaled water-phase volume fraction is computed as:

$$S_{\mathit{w,scaled}} = \frac{S_w - S_{\mathit{w,min}}}{1 - S_{\mathit{w,min}} - S_{\mathit{o,min}} - S_{\mathit{g,min}}}.$$

The gas-phase capillary pressure is computed analogously.

Parameters

The capillary pressure constitutive model is listed in the `<Constitutive>` block of the input XML file. The capillary pressure model must be assigned a unique name via `name` attribute. This name is used to assign the model to regions of the physical domain via a `materialList` attribute of the `<ElementRegions>` node.

The following attributes are supported:

Name	Type	Default	Description
capPressureEpsilon	real64	1e-06	Saturation at which the extremum capillary pressure is attained; used to avoid infinite capillary pressure values for saturations close to 0 and 1
name	string	required	A name is required for any non-unique nodes
phaseCapPressureExponentInv	real64_a	{0.5}	Inverse of capillary power law exponent for each phase
phaseCapPressureMultiplier	real64_a	{1}	Entry pressure value for each phase
phaseMinVolumeFraction	real64_a	{0}	Minimum volume fraction value for each phase
phaseNames	string_a	required	List of fluid phases

Below are some comments on the model parameters:

- `phaseNames` - The number of phases can be either 2 or 3. The phase names entered for this attribute should match the phase names specified in the relative permeability block, either in *Brooks-Corey relative permeability model* or in *Three-phase relative permeability model*. The capillary model assumes that oil is always present. Supported phase names are:

Value	Phase
oil	Oil phase
gas	Gas phase
water	Water phase

- `phaseMinVolFraction` - The list of minimum volume fractions $S_{\ell,min}$ for each phase is specified in the same order as in `phaseNames`. Below this volume fraction, the phase is assumed to be immobile. The values entered for this attribute have to match those of the same attribute in the relative permeability block.
- `phaseCapPressureExponentInv` - The list of exponents m_ℓ for each phase is specified in the same order as in `phaseNames`. The parameter corresponding to the oil phase is not used.
- `phaseCapPressureMultiplier` - The list of multipliers α_ℓ for each phase is specified in the same order as in `phaseNames`. The parameter corresponding to the oil phase is not used.
- `capPressureEpsilon` - The parameter ϵ. This parameter is used for both the water-phase and gas-phase capillary pressure. To avoid extremely large, or infinite, capillary pressure values, we set $P_{c,w}(S_w) := P_{c,w}(\epsilon)$ whenever $S_w < \epsilon$. The gas-phase capillary pressure is treated analogously.

Example

```
<Constitutive>
  ...
  <VanGenuchtenCapillaryPressure name="capPressure"
                                 phaseNames="{ water, oil }"
                                 phaseMinVolumeFraction="{ 0.1, 0.015 }"
                                 phaseCapPressureExponentInv="{ 0.55, 0 }"
                                 phaseCapPressureMultiplier="{ 1e6, 0 }"
```

(continues on next page)

(continued from previous page)

```
                                  capPressureEpsilon="1e-7"/>
  ...
</Constitutive>
```

Table capillary pressure

Overview

The user can specify the capillary pressures using tables describing a piecewise-linear capillary pressure function of volume fraction (i.e., saturation) for each phase, except the reference phase for which capillary pressure is assumed to be zero. Depending on the number of fluid phases, this model is used as follows:

- For two-phase flow, the user must specify one capillary pressure table. During the simulation, the capillary pressure of the non-reference phase is computed by interpolating in the table as a function of the non-reference phase saturation.
- For three-phase flow, the user must specify two capillary pressure tables. One capillary pressure table is required for the pair wetting-phase–intermediate-phase (typically, water-oil), and one capillary pressure table is required for the pair non-wetting-phase–intermediate-phase (typically, gas-oil). During the simulation, the former is used to compute the wetting-phase capillary pressure as a function of the wetting-phase volume fraction and the latter is used to compute the non-wetting-phase capillary pressure as a function of the non-wetting-phase volume fraction. The intermediate phase is assumed to be the reference phase, and its capillary pressure is set to zero.

Below is a table summarizing the choice of reference pressure for the various phase combinations:

Phases present in the model	Reference phase
oil, water, gas	Oil phase
oil, water	Oil phase
oil, gas	Oil phase
water, gas	Gas phase

In all cases, the user-provided capillary pressure is used in GEOS to compute the phase pressure using the formula:

$$P_c = p_{nw} - p_w.$$

where p_{nw} and p_w are respectively the non-wetting-phase and wetting-phase pressures.

Parameters

The capillary pressure constitutive model is listed in the `<Constitutive>` block of the input XML file. The capillary pressure model must be assigned a unique name via `name` attribute. This name is used to assign the model to regions of the physical domain via a `materialList` attribute of the `<ElementRegions>` node.

The following attributes are supported:

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes
nonWettingIntermediateCapPressureTableName	string		Capillary pressure table [Pa] for the pair (non-wetting phase, intermediate phase) Note that this input is only used for three-phase flow. If you want to do a two-phase simulation, please use instead wettingNonWettingCapPressureTableName to specify the table names
phaseNames	string_array	required	List of fluid phases
wettingIntermediateCapPressureTableName	string		Capillary pressure table [Pa] for the pair (wetting phase, intermediate phase) Note that this input is only used for three-phase flow. If you want to do a two-phase simulation, please use instead wettingNonWettingCapPressureTableName to specify the table names
wettingNonWettingCapPressureTableName	string		Capillary pressure table [Pa] for the pair (wetting phase, non-wetting phase) Note that this input is only used for two-phase flow. If you want to do a three-phase simulation, please use instead wettingIntermediateCapPressureTableName and nonWettingIntermediateCapPressureTableName to specify the table names

Below are some comments on the model parameters.

- phaseNames - The number of phases can be either two or three. Supported phase names are:

Value	Phase
oil	Oil phase
gas	Gas phase
water	Water phase

- wettingNonWettingCapPressureTableName - The name of the capillary pressure table for two-phase systems. Note that this keyword is only valid for two-phase systems, and is not allowed for three-phase systems (for which the user must specify instead wettingIntermediateCapPressureTableName and nonWettingIntermediateCapPressureTableName). This capillary pressure must be a strictly decreasing function of the water-phase volume fraction (for oil-water systems and gas-water systems), or a strictly increasing function of the gas-phase volume fraction (for oil-gas systems).
- wettingIntermediateCapPressureTableName - The name of the capillary pressure table for the pair wetting-phase–intermediate-phase. This capillary pressure is applied to the wetting phase, as a function of the wetting-phase volume fraction. Note that this keyword is only valid for three-phase systems, and is not allowed for two-phase systems (for which the user must specify instead wettingNonWettingCapPressureTableName). This capillary pressure must be a strictly decreasing function of the wetting-phase volume fraction.
- nonWettingIntermediateCapPressureTableName - The name of the capillary pressure table for the pair non-wetting-phase–intermediate-phase. Note that this keyword is only valid for three-phase systems, and is not allowed for two-phase systems (for which the user must specify instead wettingNonWettingCapPressureTableName). This capillary pressure must be a strictly increasing function of the non-wetting-phase volume fraction.

Examples

For a two-phase water-gas system (for instance in the CO2-brine fluid model), a typical capillary pressure input looks like:

```
<Constitutive>
   ...
   <TableCapillaryPressure
     name="capPressure"
     phaseNames="{ water, gas }"
     wettingNonWettingCapPressureTableNames="waterCapillaryPressureTable"/>
   ...
</Constitutive>
```

For a three-phase oil-water-gas system (for instance in the Black-Oil fluid model), a typical capillary pressure input looks like:

```
<Constitutive>
   ...
   <TableCapillaryPressure
     name="capPressure"
     phaseNames="{ water, oil, gas }"
     wettingIntermediateCapPressureTableName="waterCapillaryPressureTable"
     nonWettingIntermediateCapPressureTableName="gasCapillaryPressureTable"/>
   ...
</Constitutive>
```

The tables mentioned above by name must be defined in the <Functions> block of the XML file using the <TableFunction> keyword.

Porosity models

Pressure dependent porosity

Overview

This model assumes a simple exponential law for the porosity as function of pressure, i.e.

$$\phi = \phi_{ref} \, exp(c \cdot (p - p_{ref}))$$

where ϕ_{ref} is the reference porosity at reference pressure, p_{ref} , p is the pressure and, c is the compressibility.

Parameters

The following attributes are supported:

Name	Type	Default	Description
compressibility	real64	required	Solid compressibility
defaultReferencePorosity	real64	required	Default value of the reference porosity
name	string	required	A name is required for any non-unique nodes
referencePressure	real64	required	Reference pressure for solid compressibility

Example

```
<Constitutive>
  ...
  <PressurePorosity name="rockPorosity"
                    referencePressure="1.0e27"
                    defaultReferencePorosity="0.3"
                    compressibility="1.0e-9"/>
  ...
</Constitutive>
```

Biot Porosity Model

Overview

According to the poroelasticity theory, the porosity (pore volume), ϕ, can be computed as

$$\phi = \phi_{ref} + \alpha(\epsilon_v - \epsilon_{v,\,ref}) + (p - p_{ref})/N.$$

Here, ϕ_{ref} is the porosity at a reference state with pressure p_{ref} and volumetric strain $\epsilon_{v,\,ref}$. Additionally, α is the Biot coefficient, ϵ_v is the volumetric strain, p is the fluid pressure and $N = \frac{K_s}{\alpha - \phi_{ref}}$, where K_s is the grain bulk modulus.

Parameters

The Biot Porosity Model can be called in the `<Constitutive>` block of the input XML file. This porosity model must be assigned a unique name via the `name` attribute. This name is used to assign the model to regions of the physical domain via a `materialList` attribute in the `<ElementRegions>` block.

The following attributes are supported:

Name	Type	Default	Description
defaultReferencePorosity	real64	required	Default value of the reference porosity
defaultThermalExpansionCoefficient	real64	0	Default thermal expansion coefficient
grainBulkModulus	real64	required	Grain bulk modulus
name	string	required	A name is required for any non-unique nodes

Example

```
<Constitutive>
   ...
   <BiotPorosity name="rockPorosity"
                 grainBulkModulus="1.0e27"
                 defaultReferencePorosity="0.3"/>
   ...
</Constitutive>
```

Permeability models

Constant Permeability Model

Overview

This model is used to define a diagonal permeability tensor that does not depend on any primary variable.

Parameters

The following attributes are supported:

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes
permeabilityComponents	R1Tensor	required	xx, yy and zz components of a diagonal permeability tensor.

Example

```
<Constitutive>
   ...
   <ConstantPermeability name="matrixPerm"
                         permeabilityComponents="{1.0e-12, 1.0e-12, 1.0e-12}"/>
   ...
</Constitutive>
```

Exponential Decay Permeability Model

Overview

This stress-dependent permeability model assumes a simple exponential law for the fracture permeability as function of the effective normal stress acting on the fracture surface (Gutierrez et al., 2000):

$$k = k_i \text{exp}(-C {\sigma_n}')$$

where k_i is the initial unstressed fracture permeability; C is an empirical constant; ${\sigma_n}'$ is the effective normal stress.

Parameters

The Exponential Decay Permeability model can be called in the `<Constitutive>` block of the input XML file. This permeability model must be assigned a unique name via the `name` attribute. This name is used to assign the model to regions of the physical domain via a `permeabilityModelName` attribute in the `<CompressibleSolidExponentialDecayPermeability>` block.

The following attributes are supported:

Name	Type	Default	Description
empiricalConstant	real64	required	an empirical constant.
initialPermeability	R1Tensor	required	initial permeability of the fracture.
name	string	required	A name is required for any non-unique nodes

Example

```
<Constitutive>
   ...
   <ExponentialDecayPermeability
     name="fracturePerm"
     empiricalConstant="0.27"
     initialPermeability="{1e-15, 1e-15, 1e-15}"/>
   ...
</Constitutive>
```

Kozeny-Carman Permeability Model

Overview

In the Kozeny-Carman model (see ref), the permeability of a porous medium is governed by several key parameters, including porosity, grain size, and grain shape:

$$k = \frac{(s_\epsilon D_p)^2 \phi^3}{150(1-\phi)^2}$$

where s_ϵ is the sphericity of the particles, D_p is the particle diameter, ϕ is the porosity of the porous medium.

Parameters

The Kozeny-Carman Permeability Model can be called in the `<Constitutive>` block of the input XML file. This permeability model must be assigned a unique name via the `name` attribute. This name is used to assign the model to regions of the physical domain via a `materialList` attribute in the `<ElementRegions>` block.

The following attributes are supported:

Name	Type	Default	Description
anisotropy	R1Tensor	{1,1,1}	Anisotropy factors for three permeability components.
name	string	required	A name is required for any non-unique nodes
particleDiameter	real64	required	Diameter of the spherical particles.
sphericity	real64	required	Sphericity of the particles.

Example

```
<Constitutive>
   ...
   <CarmanKozenyPermeability name="matrixPerm"
                             particleDiameter="0.0002"
                             sphericity="1.0"/>
   ...
</Constitutive>
```

Parallel Plates Permeability Model

Overview

The parallel plates permeability model defines the relationship between the hydraulic fracture aperture and its corresponding permeability following the classic lubrication model (Witherspoon et al.) . In this model, the two fracture walls are assumed to be smooth and parallel to each other and separated by a uniform aperture.

$$k = \frac{a^3}{12}$$

where a denotes the hydraulic fracture aperture.

Remark: k, dimensionally, is not a permeability (as it is expressed in m^3).

Parameters

The Parallel Plates Permeability Model can be called in the `<Constitutive>` block of the input XML file. This permeability model must be assigned a unique name via the `name` attribute. This name is used to assign the model to regions of the physical domain via a `materialList` attribute in the `<ElementRegions>` block.

The following attributes are supported:

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes

Slip Dependent Permeability Model

Overview

The slip dependent permeability model defines the relationship between the relative shear displacement and fracture permeability. In this model, fractrues/faults are represented as slip interfaces.

$$k = k_i \left[(M_{mult} - 1)\text{tanh}\left(3\frac{U_s}{U_{s_{threshold}}}\right) + 1 \right]$$

where k_i is the initial fracture permeability; M_{mult} is the maximum permeability multiplier; U_s is the relative shear displacement; $U_{s_{threshold}}$ is the slip threshold.

Parameters

The Slip Dependent Permeability model can be called in the `<Constitutive>` block of the input XML file. This permeability model must be assigned a unique name via the `name` attribute. This name is used to assign the model to regions of the physical domain via a `permeabilityModelName` attribute in the `<CompressibleSolidSlipDependentPermeability>` block.

The following attributes are supported:

Name	Type	Default	Description
initialPermeability	R1Tensor	required	initial permeability of the fracture.
maxPermMultiplier	real64	required	Maximum permeability multiplier.
name	string	required	A name is required for any non-unique nodes
shearDispThreshold	real64	required	Threshold of shear displacement.

Example

```
<Constitutive>
   ...
   <SlipDependentPermeability
     name="fracturePerm"
     shearDispThreshold="0.005"
     maxPermMultiplier="1000.0"
     initialPermeability="{1e-15, 1e-15, 1e-15}"/>
```

(continues on next page)

(continued from previous page)

```
  ...
</Constitutive>
```

Willis-Richards Permeability Model

Overview

In the Willis-Richards permeability model, the stress-aperture relationship is derived based on Barton–Bandis constitutive model. In this model, fracture hydraulic aperture is assumed to be a function of effective normal stress acting on the fracture surface and shear displacement along the fracture surface (Willis-Richards et al., 1996).

$$a = \frac{a_m + U_s \tan(\phi_{dil})}{1 + 9\frac{\sigma_n}{\sigma_{ref}}}$$

Based on the assumption of parallel plates, the correlation between fracture hydraulic aperture and its corresponding permeability is defined as:

$$k = \frac{a^2}{12}$$

where

a is the fracture hydraulic aperture; a_m is the fracture aperture at zero contact stress; U_s is the relative shear displacement; ϕ_{dil} is the shear dilation angle; σ_n is the effective normal stress acting on the fracture surface; σ_{ref} is the effective normal stress that causes a 90% reduction in the fracture hydraulic aperture.

Parameters

The Willis-Richards permeability model is called in the `<Constitutive>` block of the input XML file. This permeability model must be assigned a unique name via the `name` attribute. This name is used to attach the model to regions of the physical domain via a `permeabilityModelName` attribute in the `<CompressibleSolidWillisRichardsPermeability>` block.

The following attributes are supported:

Name	Type	Default	Description
dilationCoefficient	real64	required	Dilation coefficient (tan of dilation angle).
maxFracAperture	real64	required	Maximum fracture aperture at zero contact stress.
name	string	required	A name is required for any non-unique nodes
refClosureStress	real64	required	Effective normal stress causes 90% reduction in aperture.

Example

```
<Constitutive>
  ...
  <WillisRichardsPermeability
    name="fracturePerm"
    maxFracAperture="0.005"
    dilationCoefficient="0.01"
```

(continues on next page)

(continued from previous page)

```
     refClosureStress="1.0e7"/>
   ...
</Constitutive>
```

Porous Solids

Overview

Simulation of fluid flow in porous media and of poromechanics, requires to define, along with fluid properties, the hydrodynamical properties of the solid matrix. Thus, for porous media flow and and poromecanical simulation in GEOS, two types of composite constitutive models can be defined to specify the characteristics of a porous material: (1) a *CompressibleSolid* model, used for flow-only simulations and which assumes that all poromechanical effects can be represented by the pressure dependency of the porosity; (2) a *PorousSolid* model which, instead, allows to couple any solid model with a *BiotPorosity* model and to include permeability's dependence on the mechanical response.

Both these composite models require the names of the solid, porosity and permeability models that, combined, define the porous material. The following sections outline how these models can be defined in the Constitutive block of the xml input files and which type of submodels they allow for.

CompressibleSolid

This composite constitutive model requires to define a *NullModel* as solid model (since no mechanical properties are used), a *PressurePorosity* model and any type of *Permeability* model.

To define this composite model the keyword *CompressibleSolid* has to be appended to the name of the permeability model of choice, as shown in the following example for the *ConstantPermeability* model.

```
<Constitutive>
  <CompressibleSolidConstantPermeability name="porousRock"
                                         solidModelName="nullSolid"
                                         porosityModelName="rockPorosity"
                                         permeabilityModelName="rockPermeability"/>

 <NullModel name="nullSolid"/>

 <PressurePorosity name="rockPorosity"
                   referencePressure="1.0e27"
                   defaultReferencePorosity="0.3"
                   compressibility="1.0e-9"/>

 <ConstantPermeability name="rockPermeability"
                       permeabilityComponents="{ 1.0e-4, 1.0e-4, 1.0e-4 }"/>

</Constitutive>
```

PorousSolid

To run poromechanical problems, the total stress is decomposed into an “effective stress” (driven by mechanical deformations) and a pore fluid pressure component, following the Biot theory of poroelasticity. For single-phase flow, or multiphase problems with no capillarity, this decomposition reads

$$\sigma_{ij} = \sigma\prime_{ij} - bp\delta_{ij}$$

where σ_{ij} is the ij component of the total stress tensor, $\sigma\prime_{ij}$ is the ij component of the effective (Cauchy) stress tensor, b is Biot’s coefficient, p is fluid pressure, and δ is the Kronecker delta.

The *PorousSolid* models simply append the keyword Porous in front of the solid model they contain, e.g., PorousElasticIsotropic, PorousDruckerPrager, and so on. Additionally, they require to define a *BiotPorosity* model and a *ConstantPermeability* model. For example, a Poroelastic material with a certain permeability can be defined as

```
<Constitutive>
  <PorousElasticIsotropic name="porousRock"
                          porosityModelName="rockPorosity"
                          solidModelName="rockSkeleton"
                          permeabilityModelName="rockPermeability"/>

  <ElasticIsotropic name="rockSkeleton"
                    defaultDensity="0"
                    defaultYoungModulus="1.0e4"
                    defaultPoissonRatio="0.2"/>

  <BiotPorosity name="rockPorosity"
                grainBulkModulus="1.0e27"
                defaultReferencePorosity="0.3"/>

  <ConstantPermeability name="rockPermeability"
                     permeabilityComponents="{ 1.0e-4, 1.0e-4, 1.0e-4 }"/>
</Constitutive>
```

Note that any of the previously described solid models is used by the *PorousSolid* model to compute the effective stress, leading to either poro-elastic, poro-plastic, or poro-damage behavior depending on the specific model chosen.

In an input XML file, constitutive models are listed in the `<Constitutive>` block. Each parameterized model has its own XML tag, and each must be assigned a unique name via the `name` attribute. Names are used to assign models to regions of the physical domain via the `materialList` attribute of the `<CellElementRegion>` node (see *Element: ElementRegions*). In some cases, physics solvers must also be assigned specific constitutive models to use (see *Physics Solvers*).

A typical `<Constitutive>` and `<ElementRegions>` block will look like:

```
<Problem>

  <Constitutive>

    <!-- Define a compressible, single-phase fluid called "water"-->
    <CompressibleSinglePhaseFluid
      name="water"
      referencePressure="2.125e6"
      referenceDensity="1000"
      compressibility="1e-19"
```

(continues on next page)

(continued from previous page)

```
      referenceViscosity="0.001"
      viscosibility="0.0"/>

  </Constitutive>

  <ElementRegions>

    <!--Add water to the material list for region 1-->
    <CellElementRegion
       name="region1"
       cellBlocks="{ hexahedra, wedges, tetrahedra, pyramids }"
       materialList="{ water }"/>

  </ElementRegions>

  ... remainder of problem definition here ...

</Problem>
```

1.5.5 Initial and Boundary Conditions

Hydrostatic Equilibrium Initial Condition

Overview

The user can request an initialization procedure enforcing a hydrostatic equilibrium for flow simulations and for coupled flow and mechanics simulations. The hydrostatic initialization is done by placing one or more **HydrostaticEquilibrium** tag(s) in the **FieldSpecifications** block of the XML input file. This initialization procedure is described below in the context of single-phase and compositional multiphase flow. At the end of this document, we compare the hydrostatic equilibrium method to another initialization method, based on the input of x-y-z tables.

Single-phase flow parameters

For single-phase flow, the **HydrostaticEquilibrium** initialization procedure requires the following user input parameters:

- `datumElevation`: the elevation (in meters) at which the datum pressure is enforced. The user must ensure that the datum elevation is within the elevation range defined by the input mesh. GEOS issues a warning if this is not the case.
- `datumPressure`: the pressure value (in Pascal) enforced by GEOS at the datum elevation.
- `objectPath`: the path defining the groups on which the hydrostatic equilibrium is computed. We recommend using `ElementRegions` to apply the hydrostatic equilibrium to all the cells in the mesh. Alternatively, the format `ElementRegions/NameOfRegion/NameOfCellBlock` can be used to select only a cell block on which the hydrostatic equilibrium is computed.

Note: In GEOS, the z-axis is positive going upward, this is why the attributes listed in this page are expressed as a function of elevation, not depth.

Using these parameters and the pressure-density constitutive relationship, GEOS uses a fixed-point iteration scheme to populate a table of hydrostatic pressures as a function of elevation. The fixed-point iteration scheme uses two optional attributes: `equilibriumTolerance`, the absolute tolerance to declare that the algorithm has converged, and `maxNumberOfEquilibrationTolerance`, the maximum number of iterations for a given elevation in the fixed point iteration scheme.

In addition, the elevation spacing of the hydrostatic pressure table is set with the optional `elevationIncrementInHydrostaticPressureTable` parameter (in meters), whose default value is 0.6096 meters. Then, once the table is fully constructed, the hydrostatic pressure in each cell is obtained by interpolating in the hydrostatic pressure table using the elevation at the center of the cell.

Note: The initialization algorithm assumes that the `gravityVector` (defined in the **Solvers** XML tag) is aligned with the z-axis. If this is not the case, GEOS terminates the simulation when the **HydrostaticEquilibrium** tag is detected in the XML file.

Compositional multiphase flow parameters

For compositional multiphase flow, the **HydrostaticEquilibrium** initialization procedure follows the same logic but requires more input parameters. In addition to the required `datumElevation`, `datumPressure`, and `objectPath` parameters listed above, the user must specify:

- `componentNames`: the names of the components present in the fluid model. This field is used to make sure that the components provided to **HydrostaticEquilibrium** are consistent with the components listed in the fluid model of the **Constitutive** block.
- `componentFractionVsElevationTableNames`: the names of n_c tables (where n_c is the number of components) specifying the component fractions as a function of elevation. There must be one table name per component, and the table names must be listed in the same order as the components in `componentNames`.
- `temperatureVsElevationTableName`: the names of the table specifying the temperature (in Kelvin) as a function of elevation.
- `initialPhaseName`: the name of the phase initially saturating the domain. The other phases are assumed to be at residual saturation at the beginning of the simulation.

These parameters are used with the fluid density model (depending for compositional flow on pressure, component fractions, and in some cases, temperature) to populate the hydrostatic pressure table, and later initialize the pressure in each cell.

Note: The current initialization algorithm has an important limitation and does not support initial phase contacts (e.g., water-oil, gas-oil, or water-gas contacts). The implementation assumes only one mobile phase in the initial system, identified by the `initialPhaseName` attribute. The other phases are assumed at residual saturation. As a result, the system may not be at equilibrium if there is initially more than one mobile phase in the system (for instance if the domain is saturated with gas at the top, and water at the bottom, for instance).

Note: As in the single-phase flow case, GEOS terminates the simulation if **HydrostaticEquilibrium** tag is present in an XML file defining a `gravityVector` not aligned with the z-axis.

The full list of parameters is provided below:

Name	Type	Default	Description
bcApplicationTableName	string		Name of table that specifies the on/off application of the boundary condition.
beginTime	real64	-1e+99	Time at which the boundary condition will start being applied.
componentFractionVsElevationTableNames	string_ar	{}	Names of the tables specifying the (component fraction vs elevation) relationship for each component
componentNames	string_ar	{}	Names of the fluid components
datumElevation	real64	required	Datum elevation [m]
datumPressure	real64	required	Datum pressure [Pa]
direction	R1Tenso	{0,0,0	Direction to apply boundary condition to.
elevationIncrementInHydrostaticPressureTable	real64	0.6096	Elevation increment [m] in the hydrostatic pressure table constructed internally
endTime	real64	1e+99	Time at which the boundary condition will stop being applied.
equilibrationTolerance	real64	0.001	Tolerance in the fixed-point iteration scheme used for hydrostatic initialization
functionName	string		Name of function that specifies variation of the boundary condition.
initialPhaseName	string		Name of the phase initially saturating the reservoir
logLevel	integer	0	Log level
maxNumberOfEquilibrationIterations	integer	5	Maximum number of equilibration iterations
name	string	required	A name is required for any non-unique nodes
objectPath	string		Path to the target field
scale	real64	0	Scale factor for value of the boundary condition.
temperatureVsElevationTableName	string		Name of the table specifying the (temperature [K] vs elevation) relationship

Examples

For single-phase flow, a typical hydrostatic equilibrium input looks like:

```
<FieldSpecifications>

  <HydrostaticEquilibrium
    name="equil"
    objectPath="ElementRegions"
    datumElevation="5"
    datumPressure="1e6"/>

</FieldSpecifications>
```

For compositional multiphase flow, using for instance the CO2-brine flow model, a typical hydrostatic equilibrium input looks like:

```
<FieldSpecifications>

```

(continues on next page)

(continued from previous page)

```
   <HydrostaticEquilibrium
     name="equil"
     objectPath="ElementRegions"
     datumElevation="28.5"
     datumPressure="1.1e7"
     initialPhaseName="water"
     componentNames="{ co2, water }"
     componentFractionVsElevationTableNames="{ initCO2CompFracTable,
                                               initWaterCompFracTable }"
     temperatureVsElevationTableName="initTempTable"/>

</FieldSpecifications>
```

In this case, a possible way to provide the three required tables is:

```
<Functions>

  <TableFunction
    name="initCO2CompFracTable"
    coordinates="{ 0.0, 10.0, 20.0, 30.0 }"
    values="{ 0.04, 0.045, 0.05, 0.055 }"/>

  <TableFunction
    name="initWaterCompFracTable"
    coordinates="{ 0.0, 10.0, 20.0, 30.0 }"
    values="{ 0.96, 0.955, 0.95, 0.945 }"/>

  <TableFunction
    name="initTempTable"
    coordinates="{ 0.0, 15.0, 30.0 }"
    values="{ 358.15, 339.3, 333.03 }"/>

</Functions>
```

Note that the spacing of the two component fraction tables must be the same, but the spacing of the temperature table can be different.

Expected behavior and comparison with another initialization method

As illustrated in *Tutorial 3: Regions and Property Specifications*, users can also use multiple **FieldSpecification** tags to impose initial fields, such as the pressure, component fractions, and temperature fields. To help users select the initialization method that best meets their needs, we summarize and compare below the two possible ways to initialize complex, non-uniform initial fields for compositional multiphase simulations in GEOS.

Initialization using HydrostaticEquilibrium

This is the initialization procedure that we have described in the first sections of this page. In **HydrostaticEquilibrium**, the initial component fractions and temperatures are provided as a function of elevation only, and the hydrostatic pressure is computed internally before the simulation starts. The typical input was illustrated for a CO2-brine fluid model in the previous paragraph.

Expected behavior:

- If **FieldSpecification** tags specifying initial pressure, component fractions, and/or temperature are included in an XML input file that also contains the **HydrostaticEquilibrium** tag, the **FieldSpecification** tags are ignored by GEOS. In other words, only the pressure, component fractions, and temperature fields defined with the **HydrostaticEquilibrium** tag as a function of elevation are taken into account.
- In the absence of source/sink terms and wells, the initial flow residual should be smaller than $10^{-}6$. Similarly, in coupled simulations, the residual of the mechanical problem should be close to zero.

Initialization using FieldSpecification tags

This is the initialization method illustrated in *Tutorial 3: Regions and Property Specifications*. The user can impose initial pressure, component fractions, and temperature fields using **FieldSpecification** tags, such as, for a two-component CO2-brine case:

```
<FieldSpecifications>

  <FieldSpecification
    name="initialPressure"
    initialCondition="1"
    setNames="{ all }"
    objectPath="ElementRegions"
    fieldName="pressure"
    scale="1"
    functionName="initialPressureTableXYZ"/>

  <FieldSpecification
    name="initialCO2CompFraction"
    initialCondition="1"
    setNames="{ all }"
    objectPath="ElementRegions"
    fieldName="globalCompFraction"
    component="0"
    scale="1"
    functionName="initialCO2CompFracTableXYZ"/>

  <FieldSpecification
    name="initialWaterCompFrac"
    initialCondition="1"
    setNames="{ all }"
    objectPath="ElementRegions"
    fieldName="globalCompFraction"
    component="1"
    scale="1"
    functionName="initialWaterCompFracTableXYZ"/>
```

(continues on next page)

(continued from previous page)

```
  <FieldSpecification
    name="initialTemperature"
    initialCondition="1"
    setNames="{ all }"
    objectPath="ElementRegions"
    fieldName="temperature"
    scale="1"
    functionName="initialTemperatureTableXYZ"/>

</FieldSpecifications>
```

In this input method, `initialPressureTableXYZ`, `initialCO2CompFracTableXYZ`, `initialWaterCompFracTableXYZ`, and `initialTemperatureTableXYZ` are tables describing these initial fields as a function of the x, y, and z spatial coordinates. Then, the cell-wise values are determined by interpolating in these tables using the coordinates of the center of the cells.

Expected behavior:

- In this approach, it is the responsibility of the user to make sure that these initial fields satisfy a hydrostatic equilibrium. If not, the model will equilibrate itself during the first time steps of the simulation, possibly causing large initial changes in pressure, component fractions, and temperature.
- If the initial state imposed by the **FieldSpecification** tags is not at equilibrium, the displacements produced by coupled flow and mechanics simulations should be interpreted with caution, as these displacements are computed with respect to a non-equilibrium initial state.
- This method is suited to impose initial fields in complex cases currently not supported by the **HydrostaticEquilibrium** tag (e.g., in the presence of phase contacts, capillary pressure, etc). Specifically, the user can equilibrate the model using other means (such as using another simulator, or running a few steps of GEOS), retrieve the equilibrated values, convert them into x-y-z tables, and impose them in the new GEOS simulations using **FieldSpecification** tags.

Aquifer Boundary Condition

Overview

Aquifer boundary conditions allow simulating flow between the computational domain (the reservoir) and one or multiple aquifers. In GEOS, we use a Carter-Tracy aquifer model parameterized in **Aquifer** tags of the **FieldSpecifications** XML input file blocks.

Aquifer model

An aquifer A is a source of volumetric flow rate q^A_f, where f is the index of a face connecting the aquifer and the reservoir. We use a Carter-Tracy model in GEOS to compute this volumetric flow rate.

Once q^A_f is computed, the aquifer mass contribution F^A_f is assembled and added to the mass conservation equations of the reservoir cell K connected to face f. The computation of F^A_f depends on the sign of the volumetric flow rate q^A_f.

The upwinding procedure is done as follows: if the sign of q^A_f indicates that flow goes from the aquifer to the reservoir, the aquifer contribution to the conservation equation of component c is:

$$F^A_{f,c} = \rho^A_w y^A_{w,c} q^A_f$$

where ρ_w^A is the aquifer mass/molar water phase density and $y_{w,c}^A$ is the aquifer mass/molar fraction of component c in the water phase. We assume that the aquifer is fully saturated with the water phase.

If the sign of q_f^A indicates that flow goes from the reservoir into the aquifer, the aquifer contribution to the mass/molar conservation equation of component c is computed as:

$$F_{f,c}^A = \sum_{\ell=1}^{n_p} (\rho_\ell S_\ell y_{\ell,c})_K q_f^A$$

where n_p is the number of fluid phases, $(\rho_\ell)_K$ is the reservoir cell mass/molar density of phase ℓ, $(S_\ell)_K$ is the reservoir cell saturation of phase ℓ, and $(y_{\ell,c})_K$ is the reservoir cell mass/molar fraction of component c in phase ℓ.

In the next section, we review the computation of the aquifer volumetric flow rate q_f^A.

Carter-Tracy analytical aquifer

The Carter-Tracy aquifer model is a simplified approximation to a fully transient model (see R. D. Carter and G. W. Tracy, An improved method for calculating water influx, Transactions of the AIME, 1960).

Although the theory was developed for a radially symmetric reservoir surrounded by an annular aquifer, this method applies to any geometry where the dimensionless pressure can be expressed as a function of a dimensionless time.

The two main parameters that govern the behavior of the aquifer are the time constant and the influx constant. These two parameters are precomputed at the beginning of the simulation and are later used to compute the aquifer volumetric flow rate.

Time constant

The time constant, T_c, has the dimension of time (in seconds).

It is computed as:

$$T_c = \frac{\mu_w^A \phi^A c_t^A (r_0^A)^2}{k^A}$$

where μ_w^A is the aquifer water phase viscosity, ϕ^A is the aquifer porosity, c_t^A is the aquifer total compressibility (fluid and rock), r_0^A is the inner radius of the aquifer, and k^A is the aquifer permeability.

The time constant is used to convert time (t, in seconds) into dimensionless time, t_D using the following expression:

$$t_D = \frac{t}{T_c}$$

Influx constant

The influx constant, β, has the dimension of $m^3.Pa^{-1}$.

It is computed as:

$$\beta = 6.283 h^A \theta^A \phi^A c_t^A (r_0^A)^2$$

where h^A is the aquifer thickness, θ^A is the aquifer angle, ϕ^A is the aquifer porosity, c_t^A is the aquifer total compressibility (fluid and rock), and r_0^A is the inner radius of the aquifer.

Aquifer volumetric flow rate

Let us consider a reservoir cell K connected to aquifer A through face f, and the corresponding aquifer volumetric flow rate q^A_f over time interval $[t^n, t^{n+1}]$.

The computation of q^A_f proceeds as follows:

$$q^A_f = \alpha^A_f (a - b(p_K(t^{n+1}) - p_K(t^n)))$$

where α^A_f is the area fraction of face$f$, and $p_K(t^{n+1})$ and $p_K(t^n)$ are the pressures in cell K at time t^{n+1} and time t^n, respectively.

The area fraction of face f with area $|f|$ is computed as:

$$\alpha^A_f = \frac{|f|}{\sum_{f_i \in A} |f_i|}$$

The coefficient a is computed as:

$$a = \frac{1}{T_c} \frac{\beta \Delta \Phi^A_K(t^n_D) - W^A(t^n_D) P'_D(t^{n+1}_D)}{P_D(t^{n+1}_D) - t^{n+1}_D P'_D(t^{n+1}_D)}$$

and the coefficient b is given by the formula:

$$b = \frac{1}{T_c} \frac{\beta}{P_D(t^{n+1}_D) - t^{n+1}_D P'_D(t^{n+1}_D)}$$

where $\Delta \Phi^A_K(t^n_D) := p^A - p_K(t^n) - \rho^A_w g(z_K - z^A)$ is the potential difference between the reservoir cell and the aquifer at time t^n, $P_D(t_D)$ is the dimensionless pressure evaluated at dimensionless time t_D, $P'_D(t_D)$ is the derivative of the dimensionless pressure with respect to dimensionless time, evaluated at dimensionless time t_D.

The functional relationship of dimensionless pressure, P_D, as a function of dimensionless time is provided by the user. A default table is also available, as shown below. The cumulative aquifer flow rate, $W^A(t^n_D)$, is an explicit quantity evaluated at t^n_D and updated at the end of each converged time step using the formula:

$$W^A(t^{n+1}_D) = W^A(t^n_D) + (t^{n+1} - t^n) \sum_{f \in A} q^A_f$$

with $W^A(0) := 0$.

Parameters

The main Carter-Tracy parameters and the expected units are listed below:

- *aquiferPorosity*: the aquifer porosity ϕ^A.
- *aquiferPermeability*: the aquifer permeability k^A (in m2).
- *aquiferInitialPressure*: the aquifer initial pressure p^A (in Pa), used to compute $\Delta \Phi^A_K$.
- *aquiferWaterViscosity*: the aquifer water viscosity μ^A_w (in Pa.s).
- *aquiferWaterDensity*: the aquifer water mass/molar density ρ^A_w (in kg/m3 or mole/m3).
- *aquiferWaterPhaseComponentNames*: the name of the components in the water phase. These names must match the component names listed in the fluid model of the **Constitutive** block. This parameter is ignored in single-phase flow simulations.

- *aquiferWaterPhaseComponentFraction*: the aquifer component fractions in the water phase, $y^A_{w,c}$. The components must be listed in the order of the components in *aquiferWaterPhaseComponentNames*. This parameter is ignored in single-phase flow simulations.
- *aquiferTotalCompressibility*: the aquifer total compressibility (for the fluid and the solid) c^A_t (in 1/Pa).
- *aquiferElevation*: the elevation of the aquifer (in m).
- *aquiferThickness*: the thickness of the aquifer (in m).
- *aquiferInnerRadius*: the aquifer inner radius (in m).
- *aquiferAngle*: the angle subtended by the aquifer boundary from the center of reservoir (in degrees, must be between 0 and 360).
- *allowAllPhasesIntoAquifer*: flag controlling the behavior of the aquifer when there is flow from the reservoir to the aquifer. If the flag is equal to 1, all phases can flow into the aquifer. If the flag is equal to 0, only the water phase can flow into the aquifer. The default value of this optional parameter is 0.
- *pressureInfluenceFunctionName*: the name of the table providing the dimensionless pressure as a function of dimensionless time. This table must be defined as a **TableFunction** in the **Functions** block of the XML file. If this optional parameter is omitted, a default pressure influence table is used.
- *setNames*: the names of the face sets on which the aquifer boundary condition is applied.

Note: Following the GEOS convention, the z-coordinate is increasing upward. This convention must be taken into account when providing the *aquiferElevation*. In other words, the z-value is not a depth.

The full list of parameters is provided below:

Name	Type	Default	Description
allowAllPhasesIntoAquifer	integer	0	Flag to allow all phases to flow into the aquifer. This flag only matters for the configuration in which flow is from reservoir to aquifer. - If the flag is equal to 1, then all phases, including non-aqueous phases, are allowed to flow into the aquifer. - If the flag is equal to 0, then only the water phase is allowed to flow into the aquifer. If you are in a configuration in which flow is from reservoir to aquifer and you expect non-aqueous phases to saturate the reservoir cells next to the aquifer, set this flag to 1. This keyword is ignored for single-phase flow simulations
aquiferAngle	real64	required	Angle subtended by the aquifer boundary from the center of the reservoir [degress]
aquiferElevation	real64	required	Aquifer elevation (positive going upward) [m]
aquiferInitialPressure	real64	required	Aquifer initial pressure [Pa]
aquiferInnerRadius	real64	required	Aquifer inner radius [m]
aquiferPermeability	real64	required	Aquifer permeability [m^2]
aquiferPorosity	real64	required	Aquifer porosity
aquiferThickness	real64	required	Aquifer thickness [m]
aquiferTotalCompressibility	real64	required	Aquifer total compressibility (rock and fluid) [Pa^-1]
aquiferWaterDensity	real64	required	Aquifer water density [kg.m^-3]
aquiferWaterPhaseComponentFraction	real64_array	{0}	Aquifer water phase component fraction. This keyword is ignored for single-phase flow simulations.
aquiferWaterPhaseComponentNames	string_array	{}	Aquifer water phase component names. This key-

Examples

Setting up the **Aquifer** boundary condition requires two additional pieces of information in the XML input file: a set of faces to specify where the aquifer boundary conditions will apply, and an aquifer tag that specifies the physical characteristics of the aquifer and determines how the boundary condition is applied.

1) To specify a set of faces: on simple grids, in the **Geometry** block of the XML file, we can define a **Box** that selects and assigns a name to a set of faces. To be included in a set, the faces must be fully enclosed in the **Box** (all vertices of a face must be inside the box for the face to be included to the set). The name of this box is a user-defined string, and it will be used in the aquifer tag to locate the face set. Here is an example of XML code to create such a face set from a box:

```
<Geometry>
  ...
  <Box
    name="aquifer"
    xMin="{ 999.99, 199.99, 3.99 }"
    xMax="{ 1010.01, 201.01, 6.01 }"/>
  ...
</Geometry>
```

Note: This step captures *faces*, not *cells*. For now, the user must ensure that the box actually contains faces (GEOS will proceed even if the face set is empty).

For more complex meshes, sch as those imported using the **VTKMesh**, using a **Box** to perform a face selection is challenging. We recommend using a surface in the `vtk` file instead, which will be used to locate the face set.

2) To specify the aquifer characteristics: in the **FieldSpecifications** block of the XML file, we include an **Aquifer** tag. For single-phase flow, the aquifer definition looks like:

```
<FieldSpecifications>
  ...
  <Aquifer
    name="aquiferBC"
    aquiferPorosity="2e-1"
    aquiferPermeability="3e-13"
    aquiferInitialPressure="9e6"
    aquiferWaterViscosity="0.00089"
    aquiferWaterDensity="962.81"
    aquiferTotalCompressibility="1e-10"
    aquiferElevation="4"
    aquiferThickness="18"
    aquiferInnerRadius="2000"
    aquiferAngle="20"
    setNames="{ aquifer }"/>
  ...
</FieldSpecifications>
```

For compositional multiphase flow, the user must include additional parameters to specify the water composition. We have additional influx controls over the aquifer with `allowAllPhasesIntoAquifer`. This is illustrated below for the CO2-brine fluid model:

```
<FieldSpecifications>
  ...
  <Aquifer
    name="aquiferBC"
    aquiferPorosity="2e-1"
    aquiferPermeability="3e-13"
    aquiferInitialPressure="9e6"
    aquiferWaterViscosity="0.00089"
    aquiferWaterDensity="962.81"
    aquiferWaterPhaseComponentFraction="{ 0.0, 1.0 }"
    aquiferWaterPhaseComponentNames="{ co2, water }"
    aquiferTotalCompressibility="1e-10"
    aquiferElevation="4"
    aquiferThickness="18"
    aquiferInnerRadius="2000"
    aquiferAngle="20"
    allowAllPhasesIntoAquifer="1"
    setNames="{ aquifer }"/>
  ...
</FieldSpecifications>
```

Finally, for both single-phase and multiphase flow, if a `pressureInfluenceFunctionName` attribute is specified in the **Aquifer** tag, a **TableFunction** must be included in the **Functions** block of the XML file as follows:

```
<Functions>
  ...
  <TableFunction
    name="pressureInfluenceFunction"
    coordinates="{ 0.01, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9,␣
↪1.0, 1.5,
                   2.0, 2.5, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 15.0, 20.0, 25.0,␣
↪30.0, 40.0,
                   50.0, 60.0, 70.0, 80.0, 90.0, 100.0, 200.0, 800.0, 1600.0, 3200.0,␣
↪6400.0, 12800.0 }"
    values="{ 0.112, 0.229, 0.315, 0.376, 0.424, 0.469, 0.503, 0.564, 0.616, 0.659, 0.
↪702, 0.735, 0.772, 0.802,
              0.927, 1.02, 1.101, 1.169, 1.275, 1.362, 1.436, 1.5, 1.556, 1.604, 1.651,␣
↪1.829, 1.96, 2.067, 2.147,
              2.282, 2.388, 2.476, 2.55, 2.615, 2.672, 2.723, 3.0537, 3.7468, 4.0934, 4.
↪44, 4.7866, 5.1331 }"/>
  ...
</Functions>
```

Note: The values provided in the table above are the default values used internally in GEOS when the user does not specify the pressure influence function in the XML file.

1.5.6 Event Management

The goal of the GEOS event manager is to be flexible with regards to event type, application order, and method of triggering. The event manager is configured via the `Event` block in an input .xml file, i.e.:

```
<Events maxTime="1.0e-2">
  <PeriodicEvent name="event_a"
                 target="/path/to/event"
                 forceDt="1" />
  <HaltEvent name="event_b"
             target="/path/to/halt_target"
             maxRunTime="1e6" />
</Events>
```

Event Execution Rules

The EventManager will repeatedly iterate through a list of candidate events specified via the Events block **in the order they are defined in the xml**. When certain user-defined criteria are met, they will trigger and perform a task. The simulation `cycle` denotes the number of times the primary event loop has completed, `time` denotes the simulation time at the beginning of the loop, and `dt` denotes the global timestep during the loop.

During each cycle, the EventManager will do the following:

1. Loop through each event and obtain its timestep request by considering:

    a. The maximum dt specified via the target's GetTimestepRequest method

    b. The time remaining until user-defined points (e.g. application start/stop times)

    c. Any timestep overrides (e.g. user-defined maximum dt)

    d. The timestep request for any of its children

2. Set the cycle dt to the smallest value requested by any event

3. Loop through each event and:

    a. Calculate the event `forecast`, which is defined as the expected number of cycles until the event is expected to execute.

    b. `if (forecast == 1)` the event will signal its target to prepare to execute. This is useful for preparing time-consuming I/O operations.

    c. `if (forecast <= 0)` the event will call the Execute method on its target object

4. Check to see if the EventManager exit criteria have been met

After exiting the main event loop, the EventManager will call the `Cleanup` method for each of its children (to produce final plots, etc.). Note: if the code is resuming from a restart file, the EventManager will pick up exactly where it left off in the execution loop.

Event Manager Configuration

Event

The children of the Event block define the events that may execute during a simulation. These may be of type `HaltEvent`, `PeriodicEvent`, or `SoloEvent`. The exit criteria for the global event loop are defined by the attributes `maxTime` and `maxCycle` (which by default are set to their max values). If the optional logLevel flag is set, the EventManager will report additional information with regards to timestep requests and event forecasts for its children.

Name	Type	Default	Description
logLevel	integer	0	Log level
maxCycle	integer	2147483647	Maximum simulation cycle for the global event loop.
maxTime	real64	1.79769e+30	Maximum simulation time for the global event loop.
minTime	real64	0	Start simulation time for the global event loop.
timeOutputFormat	geos_EventManager_TimeOutputF	seconds	Format of the time in the GEOS log.
HaltEvent	node		*Element: HaltEvent*
PeriodicEvent	node		*Element: PeriodicEvent*
SoloEvent	node		*Element: SoloEvent*

PeriodicEvent

This is the most common type of event used in GEOS. As its name suggests, it will execute periodically during a simulation. It can be triggered based upon a user-defined `cycleFrequency` or `timeFrequency`.

If cycleFrequency is specified, the event will attempt to execute every X cycles. Note: the default behavior for a PeriodicEvent is to execute every cycle. The event forecast for this case is given by: `forecast = cycleFrequency - (cycle - lastCycle)` .

If timeFrequency is specified, the event will attempt to execute every X seconds (this will override any cycle-dependent behavior). By default, the event will attempt to modify its timestep requests to respect the timeFrequency (this can be turned off by specifying targetExactTimestep=”0”). The event forecast for this case is given by: `if (dt > 0), forecast = (timeFrequency - (time - lastTime)) / dt, otherwise forecast=max`

By default, a PeriodicEvent will execute throughout the entire simulation. This can be restricted by specifying the beginTime and/or endTime attributes. Note: if either of these values are set, then the event will modify its timestep requests so that a cycle will occur at these times (this can be turned off by specifying targetExactStartStop=”0”).

The timestep request event is typically determined via its target. However, this value can be overridden by setting the `forceDt` or `maxEventDt` attributes.

Name	Type	Default	Description
beginTime	real6·	0	Start time of this event.
cycleFrequency	integer	1	Event application frequency (cycle, default)
endTime	real6·	1e+1(	End time of this event.
finalDtStretch	real6·	0.001	Allow the final dt request for this event to grow by this percentage to match the endTime exactly.
forceDt	real6·	-1	While active, this event will request this timestep value (ignoring any children/targets requests).
function	string		Name of an optional function to evaluate when the time/cycle criteria are met.If the result is greater than the specified eventThreshold, the function will continue to execute.
logLevel	integer	0	Log level
maxEventDt	real6·	-1	While active, this event will request a timestep <= this value (depending upon any child/target requests).
name	string	required	A name is required for any non-unique nodes
object	string		If the optional function requires an object as an input, specify its path here.
set	string		If the optional function is applied to an object, specify the setname to evaluate (default = everything).
stat	integer	0	If the optional function is applied to an object, specify the statistic to compare to the eventThreshold.The current options include: min, avg, and max.
target	string		Name of the object to be executed when the event criteria are met.
targetExactStartStop	integer	1	If this option is set, the event will reduce its timestep requests to match any specified beginTime/endTimes exactly.
targetExactTimestep	integer	1	If this option is set, the event will reduce its timestep requests to match the specified timeFrequency perfectly: dt_request = min(dt_request, t_last + time_frequency - time)).
threshold	real6·	0	If the optional function is used, the event will execute if the value returned by the function exceeds this threshold.
timeFrequency	real6·	-1	Event application frequency (time). Note: if this value is specified, it will override any cycle-based behavior.
HaltEvent	node		*Element: HaltEvent*
PeriodicEvent	node		*Element: PeriodicEvent*
SoloEvent	node		*Element: SoloEvent*

SoloEvent

This type of event will execute once once the event loop reaches a certain cycle (targetCycle) or time (targetTime). Similar to the PeriodicEvent type, this event will modify its timestep requests so that a cycle occurs at the exact time requested (this can be turned off by specifying targetExactTimestep=”0”). The forecast calculations follow an similar approach to the PeriodicEvent type.

Name	Type	Default	Description
beginTime	real64	0	Start time of this event.
endTime	real64	1e+10	End time of this event.
finalDtStretch	real64	0.001	Allow the final dt request for this event to grow by this percentage to match the endTime exactly.
forceDt	real64	-1	While active, this event will request this timestep value (ignoring any children/targets requests).
logLevel	integer	0	Log level
maxEventDt	real64	-1	While active, this event will request a timestep <= this value (depending upon any child/target requests).
name	string	required	A name is required for any non-unique nodes
target	string		Name of the object to be executed when the event criteria are met.
targetCycle	integer	-1	Targeted cycle to execute the event.
targetExactStartStop	integer	1	If this option is set, the event will reduce its timestep requests to match any specified beginTime/endTimes exactly.
targetExactTimestep	integer	1	If this option is set, the event will reduce its timestep requests to match the specified execution time exactly: dt_request = min(dt_request, t_target - time)).
targetTime	real64	-1	Targeted time to execute the event.
HaltEvent	node		*Element: HaltEvent*
PeriodicEvent	node		*Element: PeriodicEvent*
SoloEvent	node		*Element: SoloEvent*

HaltEvent

This event type is designed to track the wall clock. When the time exceeds the value specified via maxRunTime, the event will trigger and set a flag that instructs the main EventManager loop to cleanly exit at the end of the current cycle. The event for cast for this event type is given by: `forecast = (maxRuntime - (currentTime - startTime)) / realDt`

Name	Type	Default	Description
beginTime	real64	0	Start time of this event.
endTime	real64	1e+100	End time of this event.
finalDtStretch	real64	0.001	Allow the final dt request for this event to grow by this percentage to match the endTime exactly.
forceDt	real64	-1	While active, this event will request this timestep value (ignoring any children/targets requests).
logLevel	integer	0	Log level
maxEventDt	real64	-1	While active, this event will request a timestep <= this value (depending upon any child/target requests).
maxRuntime	real64	required	The maximum allowable runtime for the job.
name	string	required	A name is required for any non-unique nodes
target	string		Name of the object to be executed when the event criteria are met.
targetExactStartStop	integer	1	If this option is set, the event will reduce its timestep requests to match any specified beginTime/endTimes exactly.
HaltEvent	node		*Element: HaltEvent*
PeriodicEvent	node		*Element: PeriodicEvent*
SoloEvent	node		*Element: SoloEvent*

Other Event Features

Event Progress Indicator

Because the event manager allows the user to specify the order of events, it could introduce ambiguity into the timestamps of output files. To resolve this, we pass two arguments to the target's Execute method:

1. eventCounter (integer) - the application index for the event (or sub-event)
2. eventProgress (real64) - the percent completion of the event loop, paying attention to events whose targets are associated with physics (from the start of the event, indicated via target->GetTimestepBehavior())

For example, consider the following Events block:

```xml
<Events maxTime="1.0e-2">
  <PeriodicEvent name="outputs"
                 timeFrequency="1e-6"
                 targetExactTimestep="0"
                 target="/Outputs/siloOutput">
  <PeriodicEvent name="solverApplications_a"
                 forceDt="1.0e-5"
                 target="/Solvers/lagsolve" />
  <PeriodicEvent name="solverApplications_b"
                 target="/Solvers/otherSolver" />
  <PeriodicEvent name="restarts"
                 timeFrequency="5.0e-4"
                 targetExactTimestep="0"
                 target="/Outputs/restartOutput"/>
</Events>
```

In this case, the events solverApplications_a and solverApplications_b point target physics events. The eventCounter, eventProgress pairs will be: outputs (0, 0.0), solverApplications_a (1, 0.0), solverApplications_b (2, 0.5), and restarts (3, 1.0). These values are supplied to the target events via their Execute methods for use. For example, for the name of a silo output file will have the format: “%s_%06d%02d” % (name, cycle, eventCounter), and the time listed in the file will be `time = time + dt*eventProgress`

Nested Events

The event manager allows its child events to be nested. If this feature is used, then the manager follows the basic execution rules, with the following exception: When its criteria are met, an event will first execute its (optional) target. It will then estimate the forecast for its own sub-events, and execute them following the same rules as in the main loop. For example:

```
<Events maxTime="1.0e-2">
  <PeriodicEvent name="event_a"
                 target="/path/to/target_a" />

  <PeriodicEvent name="event_b"
                 timeFrequency="100">

    <PeriodicEvent name="subevent_b_1"
                   target="/path/to/target_b_1"/>

    <PeriodicEvent name="subevent_b_2"
                   target="/path/to/target_b_2"/>
  <PeriodicEvent/>
</Events>
```

In this example, event_a will trigger during every cycle and call the Execute method on the object located at /path/to/target_a. Because it is time-driven, event_b will execute every 100 s. When this occurs, it will execute it will execute its own target (if it were defined), and then execute subevent_b_1 and subevent_b_2 in order. Note: these are both cycle-driven events which, by default would occur every cycle. However, they will not execute until each of their parents, grandparents, etc. execution criteria are met as well.

1.5.7 Tasks Manager

The GEOS tasks manager allows a user to specify tasks to be executed. These tasks are compatible targets for the *Event Management*.

The tasks manager is configured via the `Tasks` block in an input .xml file, i.e.:

```
<Tasks>
  <PackCollection name="historyCollection" objectPath="nodeManager" fieldName="Velocity"␣
↪/>
</Tasks>
```

Tasks Manager Configuration

Task

The children of the Tasks block define different Tasks to be triggered by events specified in the *Event Management* during the execution of the simulation. At present the only supported task is the `PackCollection` used to collect time history data for output by a TimeHistory output.

Name	Type	Default	Description
CompositionalMultiphaseStatistics	node		*Element: CompositionalMultiphaseStatistics*
MultiphasePoromechanicsInitialization	node		*Element: MultiphasePoromechanicsInitialization*
PVTDriver	node		*Element: PVTDriver*
PackCollection	node		*Element: PackCollection*
ReactiveFluidDriver	node		*Element: ReactiveFluidDriver*
RelpermDriver	node		*Element: RelpermDriver*
SinglePhasePoromechanicsInitialization	node		*Element: SinglePhasePoromechanicsInitialization*
SinglePhaseStatistics	node		*Element: SinglePhaseStatistics*
SolidMechanicsStateReset	node		*Element: SolidMechanicsStateReset*
SolidMechanicsStatistics	node		*Element: SolidMechanicsStatistics*
TriaxialDriver	node		*Element: TriaxialDriver*

PackCollection

The `PackCollection` Task is used to collect time history information from fields. Either the entire field or specified named sets of indices in the field can be collected.

Name	Type	Default	Description
disableCoordCollection	integer	0	Whether or not to create coordinate meta-collectors if collected objects are mesh objects.
fieldName	string	required	The name of the (packable) field associated with the specified object to retrieve data from
name	string	required	A name is required for any non-unique nodes
objectPath	string	required	The name of the object from which to retrieve field values.
onlyOnSetChange	integer	0	Whether or not to only collect when the collected sets of indices change in any way.
setNames	string_arra	{}	The set(s) for which to retrieve data.

Note: The time history information collected via this task is buffered internally until it is output by a linked TimeHistory Output.

Triggering the Tasks

Tasks can be triggered using the *Event Management*. Recurring tasks sould use a <PeriodicEvent> and one-time tasks should use a <*SoloEvent*>:

```
<PeriodicEvent name="historyCollectEvent"
               timeFrequency="1.0"
               targetExactTimeset="1"
               target="/Tasks/historyCollection" />
```

The keyword target has to match the name of a Task specified as a child of the <Tasks> block.

1.5.8 Functions

Functions are the primary avenue for specifying values that change in space, time, or any other dimension. These are specified in the Functions block, and may be referenced by name throughout the rest of the .xml file. For example:

```
<Functions>
  <TableFunction name="q"
                 inputVarNames="time"
                 coordinates="0 60 1000"
                 values="0 1 1" />
</Functions>
<FieldSpecifications>
  <SourceFlux name="sourceTerm"
              objectPath="ElementRegions/Region1/block1"
              scale="0.001"
              functionName="q"
              setNames="{source}"/>
</FieldSpecifications>
```

Function Inputs and Application

The inputs to each function type are specified via the inputVarName attribute. These can either be the name of an array (e.g. “Pressure”) or the special keyword “time” (time at the beginning of the current cycle). If any of the input variables are vectors (e.g. “referencePosition”), the components will be given as function arguments in order.

In the .xml file, functions are referenced by name. Depending upon the application, the functions may be applied in one of three ways:

1. Single application: The function is applied to get a single scalar value. For example, this could define the flow rate applied via a BC at a given time.
2. Group application: The function is applied to a (user-specified) ManagedGroup of size N. When called, it will iterate over the inputVarNames list and build function inputs from the group’s wrappers. The resulting value will be a wrapper of size N. For example, this could be used to apply a user-defined constitutive relationship or specify a complicated boundary condition.
3. Statistics mode: The function is applied in the same manner as the group application, except that the function will return an array that contains the minimum, average, and maximum values of the results.

Function Types

There are three types of functions available for use: `TableFunction`, `SymbolicFunction`, and `CompositeFunction`. **Note: the symbolic and composite function types are currently only available for x86-64 systems.**

TableFunction

A table function uses a set of pre-computed values defined at points on a structured grid to represent an arbitrary-dimensional function. Typically, the axes of the table will represent time and/or spatial dimensions; however, these can be applied to represent phase diagrams, etc.

Name	Type	Default	Description
coordinateFiles	path_array	{}	List of coordinate file names for ND Table
coordinates	real64_array	{0}	Coordinates inputs for 1D tables
inputVarNames	string_array	{}	Name of fields are input to function.
interpolation	geos_TableFunction_Interp	linear	Interpolation method. Valid options: * linear * nearest * upper * lower
name	string	required	A name is required for any non-unique nodes
values	real64_array	{0}	Values for 1D tables
voxelFile	path		Voxel file name for ND Table

1D Table

For 1D tables, the function may be defined using the coordinates and values attributes. These represent the location of the grid nodes (ordered from smallest to largest) and the function values at those points, respectively. For example, the following function defines a simple ramp function with a rise-time of 60 seconds:

```xml
<TableFunction name="q"
               inputVarNames="time"
               coordinates="0 60 1000"
               values="0 1 1" />
```

ND Table

For ND tables, the grid coordinates and values may be defined using a set of .csv files. The `coordinateFiles` attribute specifies the file names that define the coordinates for each axis. The values in each coordinate file must be comma-delimited and ordered from smallest to largest. The dimensionality of the table is defined by the number of coordinate files (coordinateFiles=”x.csv” would indicate a 1D table, coordinateFiles=”x.csv y.csv z.csv t.csv” would indicate a 4D table, etc.). The `voxelFile` attribute specifies name of the file that defines the value of the function at each point along the grid. These values must be comma-delimited (line-breaks are allowed) and be specified in Fortran order, i.e., column-major order (where the index of the first dimension changes the fastest, and the index of the last dimension changes slowest).

The following would define a simple 2D function `c = a + 2*b`:

```
<TableFunction name="c"
               inputVarNames="a b"
               coordinateFiles="a.csv b.csv"
               voxelFile="c.csv" />
```

- a.csv: “0, 1”
- b.csv: “0, 0.5, 1”
- c.csv: “0, 1, 1, 2, 2, 3”

Interpolation Methods

There are four interpolation methods available for table functions. Within the table axes, these will return a value:

- linear: using piecewise-linear interpolation
- upper: equal to the value of the next table vertex
- nearest: equal to the value of the nearest table vertex
- lower: equal to the value of the previous table vertex

Outside of the table axes, these functions will return the edge-values. The following figure illustrates how each of these methods work along a single dimension, given identical table values:

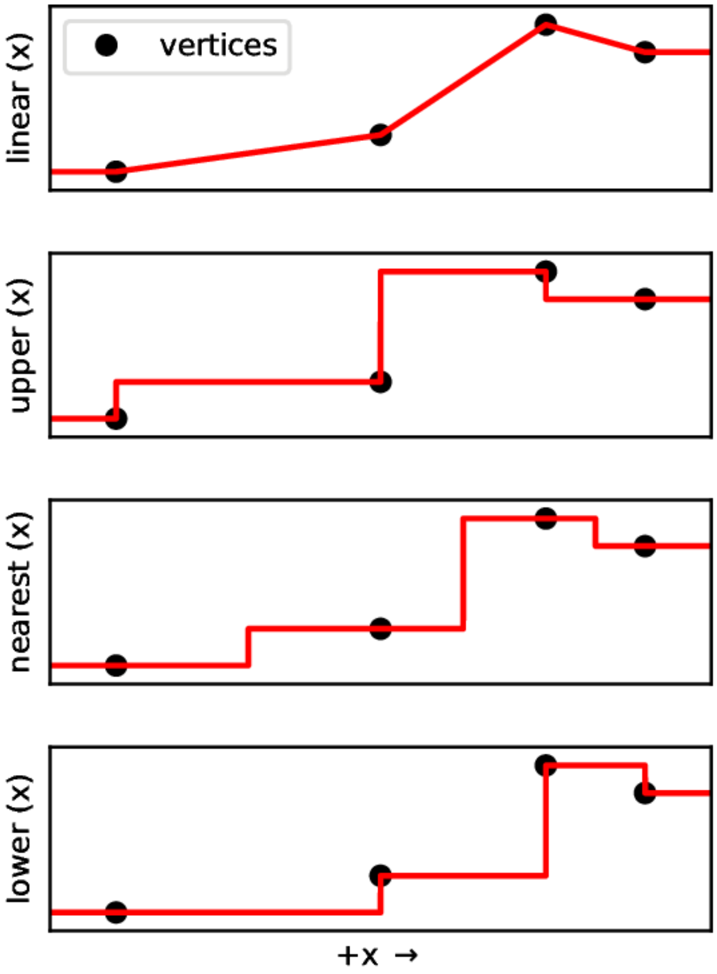


Table Generation Example

The following is an example of how to generate the above tables in Python:

```
import numpy as np

# Define table axes
a = np.array([0.0, 1.0])
b = np.array([0.0, 0.5, 1.0])

# Generate table values (note: the indexing argument is important)
A, B = np.meshgrid(a, b, indexing='ij')
C = A + 2.0*B

# Write axes, value files
np.savetxt('a.csv', a, fmt='%1.2f', delimiter=',')
np.savetxt('b.csv', b, fmt='%1.2f', delimiter=',')
values = np.reshape(C, (-1), order='F')
```

(continues on next page)

(continued from previous page)

```
np.savetxt('c.csv', values, fmt='%1.2f', delimiter=',')
```

SymbolicFunction

This function leverages the symbolic expression library mathpresso to define and evaluate functions. These functions are processed using an x86-64 JIT compiler, so are nearly as efficient as natively compiled C++ expressions.

Name	Type	Default	Description
expression	string	required	Symbolic math expression
inputVarNames	string_array	{}	Name of fields are input to function.
name	string	required	A name is required for any non-unique nodes
variableNames	string_array	required	List of variables in expression. The order must match the evaluate argument

The `variableNames` attribute defines a set of single-character names for the inputs to the symbolic function. There should be a definition for each scalar input and for each component of a vector input. For example if `inputVarName="time, ReferencePosition"`, then `variableNames="t, x, y, z"`. The `expression` attribute defines the symbolic expression to be executed. Aside from the following exceptions, the syntax mirrors python:

- The function string cannot contain any spaces
- The power operator is specified using the C-style expression (e.g. pow(x,3) instead of x**3)

The following would define a simple 2D function `c = a + 2*b`:

```
<SymbolicFunction name="c"
                  inputVarNames="a b"
                  variableNames="x y"
                  expression="x+(2*y)"/>
```

CompositeFunction

This function is derived from the symbolic function. However, instead of using the time or object as inputs, it is used to combine the outputs of other functions using a symbolic expression.

Name	Type	Default	Description
expression	string		Composite math expression
functionNames	string_array	{}	List of source functions. The order must match the variableNames argument.
inputVarNames	string_array	{}	Name of fields are input to function.
name	string	required	A name is required for any non-unique nodes
variableNames	string_array	{}	List of variables in expression

The `functionNames` attribute defines the set of input functions to use (these may be of any type, and may each have any number of inputs). The `variableNames` attribute defines a set of single-character names for each function. The `expression` attribute defines the symbolic expression, and follows the same rules as above. The `inputVarNames` attribute is ignored for this function type.

The following would define a simple 1D table function `f(t) = 1 + t`, a 3D symbolic function `g(x, y, z) = x**2 + y**2 + z**2`, and a 4D composite function `h = sin(f(t)) + g(x, y, z)`:

```xml
<Functions>
  <TableFunction name="f"
                 inputVarNames="time"
                 coordinates="0 1000"
                 values="1 1001" />

  <SymbolicFunction name="g"
                    inputVarNames="ReferencePosition"
                    variableNames="x y z"
                    expression="pow(x,2)+pow(y,2)+pow(z,2)"/>

  <CompositeFunction name="h"
                     inputVarNames="ignored"
                     functionNames="f g"
                     variableNames="x y"
                     expression="sin(x)+y"/>
</Events>
```

1.5.9 Linear Solvers

Introduction

Any physics solver relying on standard finite element and finite volume techniques requires the solution of algebraic linear systems, which are obtained upon linearization and discretization of the governing equations, of the form:

$$\mathsf{A}\mathsf{x} = \mathsf{b}$$

with a A a square sparse matrix, x the solution vector, and b the right-hand side. For example, in a classical linear elastostatics problem A is the stiffness matrix, and x and b are the displacement and nodal force vectors, respectively.

This solution stage represents the most computationally expensive portion of a typical simulation. Solution algorithms generally belong to two families of methods: direct methods and iterative methods. In GEOS both options are made available wrapping around well-established open-source linear algebra libraries, namely HYPRE, PETSc, SuperLU, and Trilinos.

Direct methods

The major advantages are their reliability, robustness, and ease of use. However, they have large memory requirements and exhibit poor scalability. Direct methods should be used in a prototyping stage, for example when developing a new formulation or algorithm, when the dimension of the problem, namely the size of matrix A, is small. Irrespective of the selected direct solver implementation, three stages can be idenitified:

(1) **Setup Stage**: the matrix is first analyzed and then factorized

(2) **Solve Stage**: the solution to the linear systems involving the factorized matrix is computed

(3) **Finalize Stage**: the systems involving the factorized matrix have been solved and the direct solver lifetime ends

The default option in GEOS relies on SuperLU, a general purpose library for the direct solution of large, sparse, nonsymmetric systems of linear equations, that is called taking advantage of the interface provided in HYPRE.

Iterative methods

As the problem size (number of computational cells) increases, global iterative solution strategies are the method of choice—typically nonsymmetric Krylov solvers. Because of the possible poor conditioning of A, preconditioning is essential to solve such systems efficiently. *''Preconditioning is simply a means of transforming the original linear system into one which has the same solution, but which is likely to be easier to solve with an iterative solver"* [Saad (2003)].

The design of a robust and efficient preconditioner is based on a trade-off between two competing objectives:

- **Robustness**: reducing the number of iterations needed by the preconditioned solver to achieve convergence;
- **Efficiency**: limiting the time required to construct and apply the preconditioner.

Assuming a preconditioning matrix M is available, three standard approaches are used to apply the preconditioner:

(1) **Left preconditioning**: the preconditioned system is $\mathsf{M}^{-1}\mathsf{A}\mathsf{x} = \mathsf{M}^{-1}\mathsf{b}$

(2) **Right preconditioning**: the preconditioned system is $\mathsf{A}\mathsf{M}^{-1}\mathsf{y} = \mathsf{b}$, with $\mathsf{x} = \mathsf{M}^{-1}\mathsf{y}$

(3) **Split preconditioning**: the preconditioned system is $\mathsf{M}_L^{-1}\mathsf{A}\mathsf{M}_R^{-1}\mathsf{y} = \mathsf{M}_L^{-1}\mathsf{b}$, with $\mathsf{x} = \mathsf{M}_R^{-1}\mathsf{y}$

Summary

The following table summarizes the available input parameters for the linear solver.

Name	Type	Default	Description
amgAggressiveCoarseningLevels	integer	0	AMG number of levels for aggressive coarsening
amgAggressiveCoarseningPaths	integer	1	AMG number of paths for aggressive coarsening
amgAggressiveInterpType	geos_LinearSolverParamete	multipass	AMG aggressive interpolation algorithm. Available options are: `default|extendedIStage2|standardStage2|extendedStage2|multipass|modifiedExtended|modifiedExtendedI|modifiedExtendedE|modifiedMultipass`
amgCoarseSolver	geos_LinearSolverParamete	direct	AMG coarsest level solver/smoother type. Available options are: `default|jacobi|l1jacobi|fgs|sgs|l1sgs|chebyshev|direct|bgs`
amgCoarseningType	geos_LinearSolverParamete	HMIS	AMG coarsening algorithm. Available options are: `default|CLJP|RugeStueben|Falgout|PMIS|HMIS`
amgInterpolationMaxNonZeros	integer	4	AMG interpolation maximum number of nonzeros per row
amgInterpolationType	geos_LinearSolverParamete	extendedI	AMG interpolation algorithm. Available options are: `default|modifiedClassical|direct|multipass|extendedI|standard|extended|directBAMG|modifiedExtended|modifiedExtendedI|modifiedExtendedE`
amgNullSpaceType	geos_LinearSolverParamete	constantModes	AMG near null space approximation. Available options are:`constantModes|rigidBodyModes`

continues on next page

Table 1.1 – continued from previous page

Name	Type	Default	Description
amgNumFunctions	integer	1	AMG number of functions
amgNumSweeps	integer	1	AMG smoother sweeps
amgRelaxWeight	real64	1	AMG relaxation factor for the smoother
amgSeparateComponents	integer	0	AMG apply separate component filter for multi-variable problems
amgSmootherType	geos_LinearSolverParamete	l1sgs	AMG smoother type. Available options are: `default|jacobi|l1jacobi|fgs|bgs|sgs|l1sgs|chebyshev|ilu0|ilut|ic0|ict`
amgThreshold	real64	0	AMG strength-of-connection threshold
directCheckResidual	integer	0	Whether to check the linear system solution residual
directColPerm	geos_LinearSolverParamete	metis	How to permute the columns. Available options are: `none|MMD_AtplusA|MMD_AtA|colAMD|metis|parmetis`
directEquil	integer	1	Whether to scale the rows and columns of the matrix
directIterRef	integer	1	Whether to perform iterative refinement
directParallel	integer	1	Whether to use a parallel solver (instead of a serial one)
directReplTinyPivot	integer	1	Whether to replace tiny pivots by sqrt(epsilon)*norm(A)
directRowPerm	geos_LinearSolverParamete	mc64	How to permute the rows. Available options are: `none|mc64`
iluFill	integer	0	ILU(K) fill factor
iluThreshold	real64	0	ILU(T) threshold factor
krylovAdaptiveTol	integer	0	Use Eisenstat-Walker adaptive linear tolerance
krylovMaxIter	integer	200	Maximum iterations allowed for an iterative solver
krylovMaxRestart	integer	200	Maximum iterations before restart (GMRES only)

continues on next page

Table 1.1 – continued from previous page

Name	Type	Default	Description
krylovTol	real64	1e-06	Relative convergence tolerance of the iterative method If the method converges, the iterative solution x_k is such that the relative residual norm satisfies: $\lVert b - Ax_k \rVert_2$ < `krylovTol` * $\lVert b \rVert_2$
krylovWeakestTol	real64	0.001	Weakest-allowed tolerance for adaptive method
logLevel	integer	0	Log level
preconditionerType	geos_LinearSolverParamete	iluk	Preconditioner type. Available options are: `none|jacobi|l1jacobi|fgs|sgs|l1sgs|chebyshev|iluk|ilut|icc|ict|amg|mgr|block|direct|bgs`
solverType	geos_LinearSolverParamete	direct	Linear solver type. Available options are: `direct|cg|gmres|fgmres|bicgstab|preconditioner`
stopIfError	integer	1	Whether to stop the simulation if the linear solver reports an error

Preconditioner descriptions

This section provides a brief description of the available preconditioners.

- **None**: no preconditioning is used, i.e., $M^{-1} = I$.
- **Jacobi**: diagonal scaling preconditioning, with $M^{-1} = D^{-1}$, with D the matrix diagonal. Further details can be found in:
  - HYPRE documentation,
  - PETSc documentation,
  - Trilinos documentation.
- **ILUK**: incomplete LU factorization with fill level k of the original matrix: $M^{-1} = U^{-1}L^{-1}$. Further details can be found in:
  - HYPRE documentation,
  - PETSc documentation,

  - Trilinos documentation.
- **ILUT**: a dual threshold incomplete LU factorization: $\mathsf{M}^{-1} = \mathsf{U}^{-1}\mathsf{L}^{-1}$. Further details can be found in:
  - HYPRE documentation,
  - not yet available through PETSc interface,
  - Trilinos documentation.
- **ICC**: incomplete Cholesky factorization of a symmetric positive definite matrix: $\mathsf{M}^{-1} = \mathsf{L}^{-T}\mathsf{L}^{-1}$. Further details can be found in:
  - not yet available through *hypre* interface,
  - PETSc documentation,
  - Trilinos documentation.
- **AMG**: algebraic multigrid (can be classical or aggregation-based according to the specific package). Further details can be found in:
  - HYPRE documentation,
  - PETSc documentation,
  - Trilinos documentation.
- **MGR**: multigrid reduction. Available through *hypre* interface only. Specific documentation coming soon. Further details can be found in MGR documentation.
- **Block**: custom preconditioner designed for a 2 x 2 block matrix.

HYPRE MGR Preconditioner

MGR stands for multigrid reduction, a multigrid method that uses the interpolation, restriction operators, and the Galerkin triple product, to reduce a linear system to a smaller one, similar to a Schur complement approach. As such, it is designed to target block linear systems resulting from discretizations of multiphysics problems. GEOS uses MGR through an implementation in HYPRE. More information regarding MGR can be found here. Currently, MGR strategies are implemented for hydraulic fracturing, poroelastic, compositional flow with and without wells. More multiphysics solvers with MGR will be enabled in the future.

To use MGR for a specific block system, several components need to be specified.

(1) **The number of reduction levels and the coarse points (corresponding to fields) for each level**. For example, for single-phase hydraulic fracturing, there are two fields, i.e. displacement and fluid pressure, a two-level MGR strategy can be used with the fluid pressure being the coarse degrees of freedom.

(2) **Interpolation/restriction operators and the coarse-grid computation strategy**. A simple but effective strategy is to use Jacobi diagonalization for interpolation and injection for restriction. For most cases, a Galerkin coarse grid strategy can be used, but for special cases such as poroelastic, a non-Galerkin approach is preferable.

(3) **Global smoother**. Depending on the problem, a global relaxation step could be beneficial. Some options include ILU(k), (block) Jacobi, (block) Gauss-Seidel.

(4) **Solvers for F-relaxation and coarse-grid correction**. These solvers should be chosen carefully for MGR to be effective. The choice of these solvers should correspond to the properties of the blocks specified by the C- and F-points. For example, if the A_{FF} block is hyperbolic, a Jacobi smoother is sufficient while for an elliptic operator an AMG V-cycle might be required. For the single-phase hydraulic fracturing case, an AMG V-cycle is needed for both F-relaxation and coarse-grid correction.

Note that these are only general guidelines for designing a working MGR recipe. For complicated multiphysics problems, experimentation with different numbers of levels, choices of C- and F-points, and smoothers/solvers, etc., is typically needed to find the best strategy. Currently, these options are only available to developers. We are working on exposing these functionalities to the users in future releases.

Block preconditioner

This framework allows the user to design a block preconditioner for a 2 x 2 block matrix. The key component is the Schur complement $\mathsf{S} = \mathsf{A}_{11} - \mathsf{A}_{10}\widetilde{\mathsf{A}}_{00}^{-1}\mathsf{A}_{01}$ computation, that requires an approximation of the leading block. Currently, available options for $\widetilde{\mathsf{A}}_{00}^{-1}$ are:

- diagonal with diagonal values (essentially, a Jacobi preconditioner);
- diagonal with row sums as values (e.g., used for CPR-like preconditioners).

Once the Schur complement is computed, to properly define the block preconditioner we need:

- the preconditioner for A_{00} (any of the above listed single-matrix preconditioner);
- the preconditioner for S (any of the above listed single-matrix preconditioner);
- the application strategy. This can be:
  - diagonal: none of the coupling terms is used;
  - upper triangular: only the upper triangular coupling term is used;
  - lower-upper triangular: both coupling terms are used.

Moreover, a block scaling is available. Feasible options are:

- none: keep the original scaling;
- Frobenius norm: equilibrate Frobenius norm of the diagonal blocks;
- user provided.

1.5.10 Numerical Methods

This section describes the specification of numerical methods used by solvers.

Name	Type	Default	Description
FiniteElements	node	unique	*Element: FiniteElements*
FiniteVolume	node	unique	*Element: FiniteVolume*

Finite Element Discretization

We are currently refactoring the finite element infrastructure, and will update the documentation soon to reflect the new structure.

Finite Volume Discretization

Two different finite-volume discretizations are available to simulate single-phase flow in GEOSX, namely, a standard cell-centered TPFA approach, and a hybrid finite-volume scheme relying on both cell-centered and face-centered degrees of freedom. The key difference between these two approaches is the computation of the flux, as detailed below.

Standard cell-centered TPFA FVM

This is the standard scheme implemented in the *SinglePhaseFVM* flow solver. It only uses cell-centered degrees of freedom and implements a Two-Point Flux Approximation (TPFA) for the computation of the flux. The numerical flux is obtained using the following expression for the mass flux between cells K and L:

$$F_{KL} = \Upsilon_{KL} \frac{\rho^{upw}}{\mu^{upw}} \big(p_K - p_L - \rho^{avg} g (d_K - d_L) \big),$$

where p_K is the pressure of cell K, d_K is the depth of cell K, and Υ_{KL} is the standard TPFA transmissibility coefficient at the interface. The fluid density, ρ^{upw}, and the fluid viscosity, μ^{upw}, are upwinded using the sign of the potential difference at the interface.

This is currently the only available discretization in the *Compositional Multiphase Flow Solver*.

Hybrid FVM

This discretization scheme overcomes the limitations of the standard TPFA on non K-orthogonal meshes. The hybrid finite-volume scheme–equivalent to the well-known hybrid Mimetic Finite Difference (MFD) scheme–remains consistent with the pressure equation even when the mesh does not satisfy the K-orthogonality condition. This numerical scheme is currently implemented in the *SinglePhaseHybridFVM* solver.

The hybrid FVM scheme uses both cell-centered and face-centered pressure degrees of freedom. The one-sided face flux, $F_{K,f}$, at face f of cell K is computed as:

$$F_{K,f} = \frac{\rho^{upw}}{\mu^{upw}} \widetilde{F}_{K,f},$$

where $\widetilde{F}_{K,f}$ reads:

$$\widetilde{F}_{K,f} = \sum_{f'} \Upsilon_{ff'} \big(p_K - \pi_f - \rho_K g (d_K - d_f) \big).$$

In the previous equation, p_K is the cell-centered pressure, π_f is the face-centered pressure, d_K is the depth of cell K, and d_f is the depth of face f. The fluid density, ρ^{upw}, and the fluid viscosity, μ^{upw}, are upwinded using the sign of $\widetilde{F}_{K,f}$. The local transmissibility Υ of size $n_{\mathit{local\,faces}} \times n_{\mathit{local\,faces}}$ satisfies:

$$N K = \Upsilon C$$

Above, N is a matrix of size $n_{\mathit{local\,faces}} \times 3$ storing the normal vectors to each face in this cell, C is a matrix of size $n_{\mathit{local\,faces}} \times 3$ storing the vectors from the cell center to the face centers, and K is the permeability tensor. The local transmissibility matrix, Υ, is currently computed using the quasi-TPFA approach described in Chapter 6 of this book. The scheme reduces to the TPFA discretization on K-orthogonal meshes but remains consistent when the mesh does not satisfy this property. The mass flux $F_{K,f}$ written above is then added to the mass conservation equation of cell K.

In addition to the mass conservation equations, the hybrid FVM involves algebraic constraints at each mesh face to enforce mass conservation. For a given interior face f between two neighboring cells K and L, the algebraic constraint reads:

$$\widetilde{F}_{K,f} + \widetilde{F}_{L,f} = 0.$$

We obtain a numerical scheme with n_{cells} cell-centered degrees of freedom and n_{faces} face-centered pressure degrees of freedom. The system involves n_{cells} mass conservation equations and n_{faces} face-based constraints. The linear systems can be efficiently solved using the MultiGrid Reduction (MGR) preconditioner implemented in the Hypre linear algebra package.

The implementation of the hybrid FVM scheme for *Compositional Multiphase Flow Solver* is in progress.

1.5.11 Parallel Partitioning

Parallel GEOS simulations involves multiple `partitions` and there are `ghost` objects in each partition. Users need to understand these concepts to effectively design models and visualize results.

Partition and ghosting : simple examples

A model, or more strictly, a computational domain, is stored in a distributed fashion among many processors. In the following simple example, the computational domain has 10 cells and the simulation involves two processors. The first processor, “partition 0” (“0” is called the “rank” of the processor) owns the first five cells (0 to 4) while “partition 1” owns 5 to 9. When the whole domain is divided into partitions, each partition will number the cells and other objects such as nodes, faces, and edges in the partition . Therefore, in both partitions, the cells IDs start from zero. Element 0 in partition 1 is cell 5 (or the sixthc cell) of the original domain. In parallel computing, each partition does not only need to know information about its own cells, but it also needs to know information about some cells owned by the neighbor partitions if these cells are directly connected to objects in this partition. For example, cell 0 in partition 1 (i.e. cell 5 in the original whole domain) is connected to cell 4 in partition 0. Therefore, partition 0 will keep a copy of this cell (including the data associated with this cell) which is synchronized with the corresponding information in the partition that actually owns this cell.

In summary, a partition owns a number of cells and other objects (e.g. faces) and also keeps copies of objects from neighbor partitions. Partitioning is automatically handled by GEOS once the user specifies how the domain should be divided.

The following figure show the partitioning of a simple mesh. Real nodes appear as solid red circles in the owning partition and ghost nodes are shown as hollow circles.

This concept of ghosting and communications between owned cells and ghost cells can also be applied to the other types of elements in GEOS (Faces, Edges, Nodes). The next figure summarizes the way nodes, edges, faces and cells are ghosted.

Specifying partitioning pattern

Cartesian partitioning

In the command line to run GEOS, the user can specify the partitioning pattern by adding the following switches:

- `-x, --x-partitions` - Number of partitions in the x-direction
- `-y, --y-partitions` - Number of partitions in the y-direction
- `-z, --z-partitions` - Number of partitions in the z-direction

Graph-based partitioning

The Graph-based partitioning is used only when importing exernal meshes using the VTKMesh (see *Tutorial 3: Regions and Property Specifications* section for more details using external meshes). While importing themesh, vtk computes the graph of connectivity between all the volume elements of the mesh. The partitioning is then done using whether a KD-tree or the PTSCOTCH, METIS, PARMETIS libraries. The graph is not weighted so the expected result is as mesh divided in n parts, with n being the number of MPI ranks used for simulation containing a similar amount of cells.

Ghost ranks

Each object (node, edge, face, or cell) has a ghost rank attribute, stored in the ghostRank field. If a object does not appear in any other partition as a ghost, its ghost rank is a large negative number, -2.14e9 in a typical system. If a object is real (owned by the current partition) but exists in other partitions as ghosts, its ghost rank is -1. The ghost rank of a ghost object is the rank of the partition that owns the corresponding real object.

Considerations for visualization

In VisIt, a partition is called a domain. The ID of a domain is the rank of the corresponding partition in GEOS plus one. VisIt would display all elements/objects regardless if they are real or ghosts. As information about a ghost is synchronized with the real object, VisIt just overlaying the same images on top of each other. The user would not perceive the overlapping between partitions unless the models are shown as semi-transparent entities. Note that if ghosts are not hidden, results from a query operation, such as summation of variable values, would be wrong due to double-counting. Therefore, it is a good practice or habit to hide ghost objects using ghostRank as a filter.

If the visualization method involves interpolation, such as interpolating a zonal field into a nodal field or generating contours, the interpretation near partition boundaries is not accurate.

1.5.12 Outputs

This section describes how outputs are handled by GEOS

The outputs are defined in a <Outputs> XML block.

There are three available formats to output the results of a simulation: SILO, VTK, and Time History output into simple dataset HDF5 files which are consumable by post-processing scripts..

Defining an output

SILO Output

The SILO output is defined through the <Silo> XML node (subnode of <Outputs> XML block) as shown here:

```xml
<Outputs>
  <Silo name="siloOutput"/>
</Outputs>
```

The parameter options are listed in the following table:

Name	Type	Default	Description
childDirectory	string		Child directory path
fieldNames	string_a	{}	Names of the fields to output. If this attribute is specified, GEOSX outputs all (and only) the fields specified by the user, regardless of their plotLevel
name	string	required	A name is required for any non-unique nodes
onlyPlotSpecifiedFieldNames	integer	0	If this flag is equal to 1, then we only plot the fields listed in *fieldNames*. Otherwise, we plot all the fields with the required *plotLevel*, plus the fields listed in *fieldNames*
parallelThreads	integer	1	Number of plot files.
plotFileRoot	string	plot	(no description available)
plotLevel	integer	1	(no description available)
writeCellElementMesh	integer	1	(no description available)
writeEdgeMesh	integer	0	(no description available)
writeFEMFaces	integer	0	(no description available)
writeFaceElementMesh	integer	1	(no description available)

VTK Output

The VTK output is defined through the `<VTK>` XML node (subnode of `<Outputs> XML block`) as shown here:

```
<Outputs>
  <VTK name="vtkOutput"/>
</Outputs>
```

The parameter options are listed in the following table:

Name	Type	Default	Description
childDirectory	string		Child directory path
fieldNames	string_array	{}	Names of the fields to output. If this attribute is specified, GEOSX outputs all the fields specified by the user, regardless of their *plotLevel*
format	geos_vtk_VTI	binary	Output data format. Valid options: `binary`, `ascii`
name	string	require	A name is required for any non-unique nodes
onlyPlotSpecifiedFieldNames	integer	0	If this flag is equal to 1, then we only plot the fields listed in *fieldNames*. Otherwise, we plot all the fields with the required *plotLevel*, plus the fields listed in *fieldNames*
outputRegionType	geos_vtk_VTI	all	Output region types. Valid options: `cell`, `well`, `surface`, `all`
parallelThreads	integer	1	Number of plot files.
plotFileRoot	string	VTK	Name of the root file for this output.
plotLevel	integer	1	Level detail plot. Only fields with lower of equal plot level will be output.
writeFEMFaces	integer	0	(no description available)
writeGhostCells	integer	0	Should the vtk files contain the ghost cells or not.

TimeHistory Output

The TimeHistory output is defined through the `<TimeHistory>` XML node (subnode of `<Outputs> XML block`) as shown here:

```
<Outputs>
  <TimeHistory name="timeHistoryOutput" sources="{/Tasks/collectionTask}" filename=
↪"timeHistory" />
</Outputs>
```

The parameter options are listed in the following table:

Name	Type	Default	Description
childDirectory	string		Child directory path
filename	string	TimeHistory	The filename to which to write time history output.
format	string	hdf	The output file format for time history output.
name	string	required	A name is required for any non-unique nodes
parallelThreads	integer	1	Number of plot files.
sources	string_array	required	A list of collectors from which to collect and output time history information.

In order to properly collect and output time history information the following steps must be accomplished:

1. Specify one or more collection tasks using the *Tasks Manager*.
2. Specify a *TimeHistory Output* using the collection task(s) as source(s).
3. Specify an event in the *Event Management* to trigger the collection task(s).
4. Specify an event in the *Event Management* to trigger the output.

Note: Currently if the collection and output events are triggered at the same simulation time, the one specified first will also trigger first. Thus in order to output time history for the current time in this case, always specify the time history collection events prior to the time history output events.

Triggering the outputs

The outputs can be triggered using the *Event Management*. It is recommended to use a <PeriodicEvent> to output results with a defined frequency:

```
<PeriodicEvent name="outputs"
               timeFrequency="5000.0"
               targetExactTimestep="1"
               target="/Outputs/siloOutput" />
```

The keyword target has to match with the name of the <Silo>, <VTK>, or <TimeHistory> node.

Visualisation of the outputs

We suggest the use of VisIT, Paraview, and MatPlotLib to visualize the outputs.

Visualizing Silo outputs with VisIT

If the <Silo> XML node was defined, GEOS writes the results in a folder called siloFiles.

In VisIT :

1. File > Open file…
2. On the right panel, browse to the siloFiles folder.
3. On the left panel, select the file(s) you want to visualize. Usually, one file is written according the frequency defined in the timeFrequency keyword of the Event that has triggered the output.
4. To load fields, use the “Add” button and browse to the fields you want to plot.
5. To plot fields, use the “Draw” button.

Please consult the VisIT documentation for further explanations on its usage.

Visualizing VTK outputs with VisIT

If the <VTK> XML node was defined, GEOS writes the results in a folder named after the `plotFileRoot` attribute (default = `vtkOutputs`). For problems with multiple active regions (e.g. *Hydraulic Fracturing*), additional work may be required to ensure that vtk files can be read by VisIt. Options include:

1. Using a VisIt macro / python script to convert default multi-region vtk files (see `GEOS/src/coreComponents/python/visitMacros/visitVTKConversion.py`).
2. Using the `outputRegionType` attribute to output separate sets of files per region. For example:

```xml
<Problem>
  <Events>
    <!-- Use nested events to trigger both vtk outputs -->
    <PeriodicEvent
      name="outputs"
      timeFrequency="2.0"
      targetExactTimestep="0">
      <PeriodicEvent
        name="outputs_cell"
        target="/Outputs/vtkOutput_cell"/>
      <PeriodicEvent
        name="outputs_surface"
        target="/Outputs/vtkOutput_surface"/>
    </PeriodicEvent>
  </Events>

  <Outputs>
    <VTK
      name="vtkOutput_cell"
      outputRegionType="cell"
      plotFileRoot="vtk_cell"/>

    <VTK
      name="vtkOutput_surface"
      outputRegionType="surface"
      plotFileRoot="vtk_surface"/>
  </Outputs>
</Problem>
```

In VisIt, the prepared files can be visualized as follows:

1. Click the Open icon (or select File/Open from the top menu).
2. Enter the vtk output folder in the left-hand panel (these will be separate if you use option 2 above).
3. Select the desired `*.vtm` group in the right-hand panel (by double-clicking or selecting OK).
4. (Repeat for any additional datasets/regions.)
5. In the main window, select the desired dataset from the Active Sources dropdown menu.
6. Click on the Add icon, and select a parameter to visualize from the mesh, psuedocolor, or vector plot options
7. At some point you may be prompted to create a Database Correlation. If you select Yes, then VisIt will create a new time slider, which can be used to change the time state while keeping the various aspects of the simulation synchronized.
8. Finally, click the Draw button and use the time slider to visualize the output.

Please consult the VisIT documentation for further explanations on its usage.

Visualizing VTK outputs with Paraview

If the <VTK> XML node was defined, GEOS writes a folder and a .pvd file named after the string defined in name keyword.

The .pvd file contains references to the .pvtu files. One .pvtu file is output according the frequency defined in the timeFrequency keyword of the Event that has triggered the output.

One .pvtu contains references to .vtu files. There is as much .vtu file as there were MPI processes used for the computation.

All these files can be opened with paraview. To have the whole results for every output time steps, you can open the .pvd file.

Visualizing TimeHistory outputs with MatPlotLib

If the <TimeHistory> XML node was defined, GEOS writes a file named after the string defined in the filename keyword and formatted as specified by the string defined in the format keyword (only HDF5 is currently supported).

The TimeHistory file contains the collected time history information from each specified time history collector. This information includes datasets for the time itself, any metadata sets describing index association with specified collection sets, and the time history information itself.

It is recommended to use MatPlotLib and format-specific accessors (like H5PY for HDF5) to access and easily plot the time history datat.

1.5.13 pygeosx — GEOS in Python

GEOS can be manipulated and executed through a Python script.

High-level control of GEOS is managed through the top-level pygeosx functions, like initialize and run. GEOS's data can be manipulated by getting pylvarray views of LvArray objects living in GEOS's data repository. These pylvarray views are fetched by calling Wrapper.value() after getting a Wrapper out of the data repository.

> **Warning:** The pygeosx module provides plenty of opportunities to crash Python. See the Segmentation Faults section below.

Only Python 3 is supported.

Module Functions

pygeosx.**initialize**(*rank*, *args*)

Initialize GEOS for the first time, with a rank and command-line arguments.

This function should only be called **once**. To reinitialize, use the reinit function.

Generally the rank is obtained from the mpi4py module and the arguments are obtained from sys.argv.

Returns a Group representing the ProblemManager instance.

pygeosx.**reinit**(*args*)

Reinitialize GEOS with a new set of command-line arguments.

Returns a `Group` representing the `ProblemManager` instance.

pygeosx.**apply_initial_conditions**()

Apply the initial conditions.

pygeosx.**finalize**()

Finalize GEOS. After this no calls into pygeosx or to MPI are allowed.

pygeosx.**run**()

Enter the GEOS event loop.

Runs until hitting a breakpoint defined in the input deck, or until the simulation is complete.

Returns one of the state constants defined below.

GEOS State

pygeosx.**UNINITIALIZED**

pygeosx.**INITIALIZED**

pygeosx.**READY_TO_RUN**

This state indicates that GEOS still has time steps left to run.

pygeosx.**COMPLETED**

This state indicates that GEOS has completed the current simulation.

Module Classes

class pygeosx.**Group**

Python interface to geos::dataRepository::Group.

Used to get access to other groups, and ultimately to get wrappers and convert them into Python views of C++ objects.

groups()

Return a list of the subgroups.

wrappers()

Return a list of the wrappers.

get_group(*path*)

get_group(*path*, *default*)

Return the `Group` at the relative path `path`; `default` is optional. If no group exists and `default` is not given, raise a `ValueError`; otherwise return `default`.

get_wrapper(*path*)

get_wrapper(*path*, *default*)

Return the `Wrapper` at the relative path `path`; `default` is optional. If no Wrapper exists and `default` is not given, raise a `ValueError`; otherwise return `default`.

register(*callback*)

Register a callback on the physics solver.

The callback should take two arguments: the CRSMatrix and the array.

Raise `TypeError` if the group is not the Physics solver.

class pygeosx.**Wrapper**

Python interface to geos::dataRepository::WrapperBase.

Wraps a generic C++ object. Use `repr` to get a description of the type.

value()

Return a view of the wrapped value, or `None` if it cannot be exported to Python.

A breakdown of the possible return types:

- Instance of a pylvarray class If the wrapped type is one of the LvArray types that have a Python wrapper type.
- 1D numpy.ndarray If the wrapped type is a numeric constant. The returned array is a shallow copy and has a single entry.
- str If the wrapped type is a std::string this returns a copy of the string.
- list of str If the wrapped type is a *LvArray::Array< std::string, 1, …* > or a *std::vector< std::string* >. This is a copy.
- None If the wrapped type is not covered by any of the above.

Segmentation Faults

Improper use of this module and associated programs can easily cause Python to crash. There are two main causes of crashes. Both can be avoided by following some general guidelines.

Stale Numpy Views

The `pylvarray` classes (which may be returned from `Wrapper.value()`) provide various ways to get Numpy views of their data. However, those views are only valid as long as the LvArray object's buffer is not reallocated. The buffer may be reallocated by invoking methods (the ones that require the `pylarray.RESIZEABLE` permission) or by calls into `pygeosx`. It is strongly recommended that you do not keep Numpy views of LvArray objects around after calls to `pygeosx`.

```
my_array = pygeosx.get_wrapper("path").value()
view = my_array.to_numpy()
my_array.resize(1000)
print(view) # segfault
```

Destroyed LvArray C++ objects

As mentioned earlier, the classes defined in this module cannot be created in Python; pygeosx must create an LvArray object in C++, then create a pylvarray view of it. However, the Python view will only be valid as long as the underlying LvArray C++ object is kept around. If that is destroyed, the Python object will be left holding an invalid pointer and subsequent attempts to use the Python object will cause undefined behavior. Unfortunately, pygeosx may destroy LvArray objects without warning. It is therefore strongly recommended that you do not keep pylvarray objects around after calls to pygeosx. The following code snippet, for instance, could segfault:

```
my_array = pygeosx.get_wrapper("path").value()
pygeosx.run()
view = my_array.to_numpy() # segfault
```

1.5.14 Indices and tables

- genindex
- modindex
- search

1.6 Developer Guide

Welcome to the GEOS developer guide.

1.6.1 Contributing

Code style

Introduction

GEOS is written in standard c++17. In general, target platforms are:

- Linux
- Mac OS X

Currently, our CI/CD system tests on these platforms:

- Ubuntu 18.04, with gcc 8.0 and clang 8.0.0 + cuda10.1.243
- Centos 7.6.1810, with gcc 8.3.1 + cuda10.1.243
- Centos 7.7, with clang 9.0.0
- Mac OS X, with xcode 11.2

Naming Conventions

File Names

- File names should be PascalCase.
- C++ header files are always named with a file extension of *.hpp.
- C++ header implementation files, which contain templated or inline function definitions, are always named *Helpers.hpp.
- C++ source files are always named with a file extension of *.cpp.
- C++ class declarations and definitions are contained files with identical names, except for the extensions.
- C++ free function headers and source files are declared/defined in files with identical names, except for the extension.

For example, a class named “Foo” may be declared in a file named “Foo.hpp”, with inline/templated functions defined in “FooHelpers.hpp”, with the source implementation contained in Foo.cpp.

Warning: There should not be identical filenames that only differ by case. Some filesystems are not case-sensitive, and worse, some filesystems such as MacOSX are case-preserving but not case sensitive.

Function Names

Function and member function names should be camelCase.

Variable Names

Variables should be camelCase.

Member Names

Member data should be camelCase prefix with “m_” (i.e. double m_dataVariable;)

Class/Struct Names

Please use PascalCase for typenames (i.e. classes)

```
class MyClass;
```

```
class MyClass
{
  double m_doubleDataMember;
  int m_integerDataMember;
}
```

Alias/Typedef Names

Alias and typedefs should be the case of the underlying type that they alias. If no clear format is apparent, as is the case with *double*, then use camelCase

Namespace Names

Namespaces names are all lower camel case.

Example

One example of would be a for a class named “Foo”, the declaration would be in a header file named “Foo.hpp”

```
/*
 * Foo.hpp
 */

namespace bar
{

class Foo
{
public:
  Foo();
private:
  double m_myDouble;
}
}
```

and a source file named “Foo.cpp”

```
/*
 * Foo.cpp
 */
namespace bar
{
  Foo::Foo():
    m_myDouble(0.0)
  {
    // some constructor stuff
  }
}
```

Const Keyword

1. All functions and accessors should be declared as “const” functions unless modification to the class is required.
2. In the case of accessors, both a “const” and “non-const” version should be provided.
3. The const keyword should be placed in the location read by the compiler, which is right to left.

The following examples are provided:

```
int a=0; // regular int
int const b = 0; // const int
int * const c = &a; // const pointer to non const int
int const * const d = &b; // const pointer to const int
int & e = a; // reference to int
int const & f = b; // reference to const int
```

Code Format

GEOS applies a variant of the BSD/Allman Style. Key points to the GEOS style are:

1. Opening braces (i.e. “{”) go on the next line of any control statement, and are not indented from the control statement.
2. NO TABS. Only spaces. In case it isn’t clear … NO TABS!
3. 2-space indentation

```
for( int i=0 ; i<10 ; ++i )
{
  std::cout << "blah" << std::endl;
}
```

4. Try to stay under 100 character line lengths. To achieve this apply these rules in order
5. Align function declaration/definitions/calls on argument list
6. Break up return type and function definition on new line
7. Break up scope resolution operators

```
void
SolidMechanics_LagrangianFEM::
TimeStepExplicit( real64 const& time_n,
                  real64 const& dt,
                  const int cycleNumber,
                  DomainPartition * const domain )
{
  code here
}
```

As part of the continuous integration testing, this GEOS code style is enforced via the uncrustify tool. While quite extensive, uncrustify does not enforce every example of the preferred code style. In cases where uncrusitfy is unable to enforce code style, it will ignore formatting rules. In these cases it is acceptable to proceed with pull requests, as there is no logical recourse.

Header Guards

Header guard names should consist of the name *GEOS*, followed by the component name (e.g. dataRepository), and finally the name of the header file. All characters in the macro should be capitalized.

Git Workflow

The GEOS project is hosted on github here. For instructions on how to clone and build GEOS, please refer to the *Quick Start Guide*. Consider consulting https://try.github.io/ for practical references on how to use git.

Git Credentials

Those who want to contribute to GEOS should setup SSH keys for authentication, and connect to github through SSH as discussed in this article. Before going further, you should test your ssh connection. If it fails (perhaps because of your institution's proxy), you may consider the personal access token option as an alternative.

Downloading the Code

Once you have created an `ssh-key` and you have added it to your *Github* account you can download the code through SSH. The following steps clone the repository into `your_geosx_dir`:

```
git clone git@github.com:GEOS-DEV/GEOS.git your_geosx_dir
cd your_geosx_dir
git lfs install
git submodule init
git submodule update
```

If all goes well, you should have a complete copy of the GEOS source at this point. The most common errors people encounter here have to do with Github not recognizing their authentication settings.

Branching Model

The branching model used in GEOS is a modified Gitflow approach, with some modifications to the merging strategy, and the treatment of release branches, and hotfix branches.

In GEOS, there are two main branches, `release` and `develop`. The `develop` branch serves as the main branch for the development of new features. The `release` branch serves as the "stable release" branch. The remaining branch types are described in the following subsections.

Note: The early commits in GEOS (up to version 0.2) used a pure Gitflow approach for merging feature branches into develop. This was done without cleaning the commit history in each feature branch prior to the merge into develop, resulting in an overly verbose history. Furthermore, as would be expected, having many active feature branches resulted in a fairly wide (spaghetti) history. At some point in the development process, we chose to switch primarily to a squash-merge approach which results in a linear develop history. While this fixes the spaghetti history, we do potentially lose important commit history during the development process. Options for merging are discussed in the following sections.

Feature Branches

New developments (new features or modifications to features) are branched off of develop into a feature branch. The naming of feature branches should follow feature/[developer]/[branch-description] if you expect that only a single developer will contribute to the branch, or feature/[branch-description] if you expect it will be a collaborative effort. For example, if a developer named neo were to add or modify a code feature expecting that they would be the only contributor, they would create a branch using the following commands to create the local branch and push it to the remote repository:

```
git checkout -b feature/neo/freeYourMind
git push -u origin feature/neo/freeYourMind
```

However if the branch is a collaborative branch amongst many developers, the appropriate commands would be:

```
git checkout -b feature/freeYourMind
git push -u origin feature/freeYourMind
```

When feature branches are ready to be merged into develop, a Pull Request should be created to perform the review and merging process.

An example lifecycle diagram for a feature branch:

```
create new feature branch:
git checkout -b feature/neo/freeYourMind

A-------B-------C (develop)
         \
          \
          BA      (feature/neo/freeYourMind)

Add commits to 'feature/neo/freeYourMind' and merge back into develop:

A-------B--------C-------D--------E (develop)
         \              /
          \            /
          BA----BB----BC            (feature/neo/freeYourMind)
```

See below for details about *Submitting a Pull Request*.

Bugfix Branches

Bugfix branches are used to fix bugs that are present in the develop branch. A similar naming convention to that of the feature branches is used, replacing “feature” with “bugfix” (i.e. bugfix/neo/squashAgentSmith). Typically, bugfix branches are completed by a single contributor, but just as with the feature branches, a collaborative effort may be required resulting a dropping the developer name from the branch name.

When bugfix branches are ready to be merged into develop, a Pull Request should be created to perform the review and merging process. See below for details about *Submitting a Pull Request*.

Release Candidate Branches

When develop has progressed to a point where we would like to create a new release, we will create a release candidate branch with the name consisting of release_major.minor.x number, where the x represents the sequence of patch tags that will be applied to the branch. For instance if we were releasing version 1.2.0, we would name the branch release_1.2.x. Once the release candidate is ready, it is merged back into develop. Then the develop branch is merged into the release branch and tagged. From that point the release branch exists to provide a basis for maintaining a stable release version of the code. Note that the absence of hotfix branches, the history for release and develop would be identical.

An example lifecycle diagram for a release candidate branch:

```
                                  v1.2.0          (tag)
                                  G               (release)
                                  ^
                                  |
A----B-----C----D-----E-----F-----G------------   (develop)
      \          \         /
       \          \       /
       BA----BB----BC----BD                       (release_1.2.x)
```

Hotfix Branches

A hotfix branch fixes a bug in the release branch. It uses the same naming convention as a bugfix branch. The main difference with a bugfix branch is that the primary target branch is the release branch instead of develop. As a soft policy, merging a hotfix into a release branch should result in a patch increment for the release sequence of tags. So if a hotfix was merged into release with a most recent tag of 1.2.1, the merged commit would be tagged with 1.2.2. Finally, at some point prior to the next major/minor release, the release branch should be merged back into develop to incorporate any hotfix changes into develop.

An example lifecycle diagram for hotfix branchs:

```
     v1.2.0       v1.2.1       v1.2.2         v1.3.0 (tag)
     B------------H1-----------H2             I      (release)
     ^\          /| \         / \             ^
     | \        /  \ \       /   \            |
     |  BA-----BB   \ H1A--H1B    \           |      (hotfix/xyz)
     |               \             \          |
A----B-----C-----D----E------F------G----H----I---   (develop)
```

Documentation Branches

A docs branch is focused on writing and improving the documentation for GEOS. The use of the docs branch name root applies to both sphinx documentation and doxygen documentation. The docs branch follows the same naming conventions as described in the *Feature Branches* section. The html produced by a documentation branch should be proofread using sphinx/doxygen prior to merging into develop.

Keeping Your Branch Current

Over the course of a long development effort in a single `feature` branch, a developer may need to either merge `develop` into their `feature` branch, or rebase their `feature` branch on `develop`. We do not have a mandate on how you keep your branch current, but we do have guidelines on the branch history when merging your branch into `develop`. Typically, merging `develop` into your branch is the easiest approach, but will lead to a complex relationship with `develop` with multiple interactions… which can lead to a confusing history. Conversely, rebasing your branch onto `develop` is more difficult, but will lead to a linear history within the branch. For a complex history, we will perform a squash merge into `develop`, thereby the work from the branch will appear as a single commit in `develop`. For clean branch histories where the individual commits are meaningful and should be preserved, we have the option to perform a merge commit in with the PR is merged into `develop`, with the addition of a merge commit, thus maintaining the commit history.

Branching off of a Branch

During the development processes, sometimes it is appropriate to create a branch off of a branch. For instance, if there is a large collaborative development effort on the branch `feature/theMatrix`, and a developer would like to add a self-contained and easily reviewable contribution to that effort, he/she should create a branch as follows:

```
git checkout feature/theMatrix
git checkout -b feature/smith/dodgeBullets
git push -u origin feature/smith/dodgeBullets
```

If `feature/smith/dodgeBullets` is intended to be merged into `feature/theMatrix`, and the commit history of `feature/theMatrix` is not changed via `git rebase`, then the process of merging the changes back into `feature/theMatrix` is fairly standard.

However, if `feature/theMatrix` is merged into `develop` via a `squash merge`, and then `smith` would like to merge `feature/smith/dodgeBullets` into `develop`, there is a substantial problem due to the diverged history of the branches. Specifically, `feature/smith/dodgeBullets` branched off a commit in `feature/theMatrix` that does not exist in `develop` (because it was squash-merged). For simplicity, let us assume that the commit hash that `feature/smith/dodgeBullets` originated from is `CC`, and that there were commits `CA, CB, CC, CD` in `feature/theMatrix`. When `feature/theMatrix` was squash-merged, all of the changes appear in `develop` as commit `G`. To further complicate the situation, perhaps a complex PR was merged after `G`, resulting in `E` on develop. The situation is illustrated by:

```
A----B----C----D----E----F----G----E (develop)
           \                 /
            CA---CB---CC---CD        (feature/theMatrix)
                       \
                       CCA--CCB--CCC (feature/smith/dodgeBullets)
```

In order to successfully merge `feature/smith/dodgeBullets` into `develop`, all commits present in `feature/smith/dodgeBullets` after `CC` must be included, while discarding `CA, CB`, which exist in `feature/smith/dodgeBullets` as part of its history, but not in `develop`.

One "solution" is to perform a `git rebase --onto` of `feature/smith/dodgeBullets` onto `develop`. Specifically, we would like to rebase `CCA, CCB, CCC` onto *G*, and proceed with our development of `feature/smith/dodgeBullets`. This would look like:

```
git checkout develop
git pull
git checkout feature/smith/dodgeBullets
git rebase -onto G CC
```

As should be apparent, we have specified the starting point as `G`, and the point at which we replay the commits in `feature/smith/dodgeBullets` as all commits AFTER `CC`. The result is:

```
A----B----C----D----E----F----G----E (develop)
                               \
                              CCA'--CCB'--CCC' (feature/smith/dodgeBullets)
```

Now you may proceed with standard methods for keeping `feature/smith/dodgeBullets` current with `develop`.

Submitting a Pull Request

Once you have created your branch and pushed changes to Github, you can create a Pull Request on Github. The PR creates a central place to review and discuss the ongoing work on the branch. Creating a pull request early in the development process is preferred as it allows for developers to collaborate on the branch more readily.

Note: When initially creating a pull request (PR) on GitHub, always create it as a *draft* PR while work is ongoing and the PR is not ready for testing, review, and merge consideration.

When you create the initial draft PR, please ensure that you apply appropriate labels. Applying labels allows other developers to more quickly filter the live PRs and access those that are relevant to them. Always add the *new* label upon PR creation, as well as to the appropriate *type*, *priority*, and *effort* labels. In addition, please also add any appropriate *flags*.

Note: If your branch and PR will resolve any open issues, be sure to *link* them to the PR to ensure they are appropriately resolved once the PR is merged. In order to *link* the issue to the PR for automatic resolution, you must use one of the keywords followed by the issue number (e.g. resolves #1020) in either the main description of the PR, or a commit message. Entries in PR comments that are not the main description or a commit message will be ignored, and the issue will not be automatically closed. A complete list of keywords are:

- close
- closes
- closed
- fix
- fixes
- fixed
- resolve
- resolves
- resolved

For more details, see the Github Documentation.

Once you are satisfied with your work on the branch, you may promote the PR out of draft status, which will allow our integrated testing suite to execute on the PR branch to ensure all tests are passing prior to merging.

Once the tests are passing – or in some cases immediately – add the *flag: ready for review* label to the PR, and be sure to tag any relevant developers to review the PR. The PR *must* be approved by reviewers in order to be merged.

Note that whenever a pull request is merged into `develop`, commits are either `squashed`, or preserved depending on the cleanliness of the history.

Keeping Submodules Current

Whenever you switch between branches locally, pull changes from `origin` and/or `merge` from the relevant branches, it is important to update the submodules to move the `head` to the proper `commit`.

```
git submodule update --recursive
```

You may also wish to modify your *git pull* behavior to update your submodules recursively for you in one command, though you forfeit some control granularity to do so. The method for accomplishing this varies between git versions, but as of git 2.15 you should be able to globally configure git to accomplish this via:

```
git config --global submodule.recurse true
```

In some cases, code changes will require to rebaseline the `Integrated Tests`. If that is the case, you will need to modify the `integrated tests submodule`. Instructions on how to modify a submodule are presented in the following section.

Working on the Submodules

Sometimes it may be necessary to modify one of the submodules. In order to do so, you need to create a pull request on the submodule repository. The following steps can be followed in order to do so.

Move to the folder of the `submodule` that you intend to modify.

```
cd submodule-folder
```

Currently the `submodule` is in detached head mode, so you first need to move to the main branch (either `develop` or `master`) on the submodule repository, pull the latest changes, and then create a new branch.

```
git checkout <main-branch>
git pull
git checkout -b <branch-name>
```

You can perform some work on this branch, *add* and *commit* the changes and then push the newly created branch to the `submodule repository` on which you can eventually create a pull request using the same process discussed above in *Submitting a Pull Request*.

```
git push --set-upstream origin <branch-name>
```

Resolving Submodule Changes in Primary Branch PRs

When you conduct work on a submodule during work on a primary GEOS branch with an open PR, the merging procedure requires that the submodule referenced by the GEOS PR branch be consistent with the submodule in the main branch of the project. This is checked and enforced via our CI.

Thus, in order to merge a PR that includes modifications to submodules, the various PRs for each repository should be staged and finalized, to the point they are all ready to be merged, with higher-level PRs in the merge hierarchy having the correct submodule references for the current main branch for their repository.

Starting from the bottom of the submodule hierarchy, the PRs are resolved, after which the higher-level PRs with reference to a resolved PR must update their submodule references to point to the new main branch of the submodule with the just-resolved PR merged. After any required automated tests pass, the higher-level PRs can then be merged.

The name of the main branch of each submodule is presented in the table below.

Submodule	Main branch
blt	develop
LvArray	develop
integratedTests	develop
hdf5_interface	master
PVTPackage	master

Sphinx Documentation

Generating the documentation

- To generate the documentation files, you will need to install Sphinx using

  ```
  sudo apt install python-sphinx
  ```

  Then you can generate the documentation files with the following command

  ```
  cd GEOS/build-your-platform-release
  make geosx_docs
  ```

- That will create a new folder

  ```
  GEOS/build-your-platform-release/html/docs/sphinx
  ```

which contains all the html files generated.

Documenting the code

The documentation is generated from restructured text files (`.rst`). Most files can be found in `src/docs/sphinx`. Files which are specific to parts of the code, like those describing a specific class, can instead be found in `docs` subdirectory in the folder containing the source code.

Information about how to write `rst` files can be found here .

Doxygen Documentation

Developer documentation of code is provided in the form of Doxygen-style comment blocks. Doxygen is a tool for generating html/xml/latex documentation for C++ code from specially marked code comments. Having concise but high quality documentation of public APIs helps both users and developers of these APIs. We use Doxygen and Ctest to enforce documentation coverage. If Doxygen produces any warnings, your pull request will fail CI checks! See *Git Workflow* for more on pull requests and CI.

Accessing

There are two ways to access Doxygen documentation.

Build locally

Prior to configuring a GEOS build, have Doxygen installed:

```
sudo apt install doxygen
```

Note: Eventually, doxygen (version 1.8.13) is provided within the *thirdPartyLibs* repository.

Configure GEOS and go the build directory:

```
cd GEOS/build-your-platform-release
```

Build doxygen docs only:

```
make geosx_doxygen
```

Or build all docs:

```
make geosx_docs
```

Open in browser:

```
google-chrome html/doxygen_output/html/index.html
```

On readthedocs

Go to GEOS documentation, select the version of interest, and follow the Doxygen link at the left-hand-side.

Guidelines

What to document

The following entities declared in project header files within *geosx* namespace require documentation:

- all classes and structs, including public nested ones
- global functions, variables and type aliases
- public and protected member functions, variables and type aliases in classes
- preprocessor macros

Exceptions are made for:

- overrides of virtual functions in derived types
- implementation details nested in namespace *internal*
- template specializations in some cases

How to document

The following rules and conventions are used. Some are stricter than others.

1. We use @-syntax for all Doxygen commands (e.g. *@brief* instead of *\brief*).
2. Entities such as type aliases and member variables that typically only require a brief description, can have a single-line documentation starting with *///*.
   - *@brief* is not required for single-line comments.
3. Entities such as classes and functions that typically require either detailed explanation or parameter documentation, are documented with multiline comment blocks.
   - *@brief* is required for comment blocks.
4. Brief and detailed descriptions should be complete sentences (i.e. start with a capital letter and end with a dot).
5. Prefer concise wording in *@brief*, e.g. "Does X." instead of "This is a function that does X."
6. All functions parameters and return values must be explicitly documented via *@param* and *@return*.
   - An exception to this rule seem to be copy/move constructor/assignment, where parameter documentation can be omitted.
7. Add *[in]* and *[out]* tags to function parameters, as appropriate.
8. Function and template parameter descriptions are not full sentences (i.e. not capitalized nor end with a dot).
9. For hierarchies with virtual inheritance, document base virtual interfaces rather than overriding implementations.
10. Documented functions cannot use *GEOS_UNUSED_ARG()* in their declarations.
11. For empty virtual base implementations that use *GEOS_UNUSED_ARG(x)* to remove compiler warnings, use one of two options:
    - move empty definition away (e.g. out of class body) and keep *GEOS_UNUSED_ARG(x)* in definition only;
    - put *GEOS_UNUSED_VAR(x)* into the inline empty body.
12. For large classes, logically group functions using member groups via *///@{* and *///@}* and give them group names and descriptions (if needed) via a *@name* comment block. Typical groups may include:
    - constructors/destructor/assignment operators;
    - getter/setter type functions;
    - overridable virtual functions;
    - any other logically coherent groups (functions related to the same aspect of class behavior).
13. In-header implementation details (e.g. template helpers) often shouldn't appear in user documentation. Wrap these into *internal* namespace.
14. Use */// @cond DO_NOT_DOCUMENT* and */// @endcond* tags to denote a section of public API that should not be documented for some reason. This should be used rarely and selectively. An example is in-class helper structs that must be public but that user should not refer to explicitly.

Example

```
/// This is a documented macro
#define USEFUL_MACRO

/**
 * @brief Short description.
 * @tparam    T type of input value
 * @param[in] x input value explanation
 * @return      return value explanation
 *
 * Detailed description goes here.
 *
 * @note A note warning users of something unexpected.
 */
template<typename T>
int Foo( T const & x );

/**
* @brief Class for showing Doxygen.
* @tparam T type of value the class operates on
*
* This class does nothing useful except show how to use Doxygen.
*/
template<typename T>
class Bar
{
public:

  /// A documented member type alias.
  using size_type = typename std::vector<T>::size_type;

  /**
   * @name Constructors/destructors.
   */
  ///@{

  /**
   * @brief A documented constructor.
   * @param value to initialize the object
   */
  explicit Bar( T t );

  /**
   * @brief A deleted, but still documented copy constructor.
   * @param an optionally documented parameter
   */
  Bar( Bar const & source ) = delete;

  /**
   * @brief A defaulted, but still documented move constructor.
   * @param an optionally documented parameter
   */
```

(continues on next page)

(continued from previous page)

```
  Bar( Bar const & source ) = default;

  /**
   * @brief A documented desctructor.
   * virtual ~Bar() = default;
   */

  ///@}

  /**
   * @name Getters for stored value.
   */
  ///@{

  /**
   * @brief A documented public member function.
   * @return a reference to contained value
   */
  T & getValue();

  /**
   * @copydoc getValue()
   */
  T const & getValue() const;

  ///@}

protected:

  /**
   * @brief A documented protected pure virtual function.
   * @param[in]  x the input value
   * @param[out] y the output value
   *
   * Some detailed explanation for users and implementers.
   */
  virtual void doSomethingOverridable( int const x, T & y ) = 0;

  /// @cond DO_NOT_DOCUMENT
  // Some stuff we don't want showing up in Doxygen
  struct BarHelper
  {};
  /// @endcond

private:

  /// An optionally documented (not enforced) private member.
  T m_value;

};
```

Current Doxygen

Link to Doxygen

Unit Testing

Unit testing is integral to the GEOS development process. While not all components naturally lend themselves to unit testing (for example a physics solver) every effort should be made to write comprehensive quality unit tests.

Each sub-directory in `coreComponents` should have a `unitTests` directory containing the test sources. Each test consists of a `cpp` file whose name begins with `test` followed by a name to describe the test. Please read over the LvArray unit test documentation as it gives an intro to the Google Test framework and a set of best practices.

GEOS Specific Recommendations

An informative example is `testSinglePhaseBaseKernels` which tests the single phase flow mobility and accumulation kernels on a variety of inputs.

```
TEST( SinglePhaseBaseKernels, mobility )
{
  int constexpr NTEST = 3;

  real64 const dens[NTEST]        = { 800.0, 1000.0, 1500.0 };
  real64 const dDens_dPres[NTEST] = { 1e-5, 1e-10, 0.0    };
  real64 const visc[NTEST]        = { 5.0, 2.0, 1.0    };
  real64 const dVisc_dPres[NTEST] = { 1e-7, 0.0, 0.0    };

  for( int i = 0; i < NTEST; ++i )
  {
    SCOPED_TRACE( "Input # " + std::to_string( i ) );

    real64 mob;
    real64 dMob_dPres;

    MobilityKernel::compute( dens[i], dDens_dPres[i], visc[i], dVisc_dPres[i], mob, dMob_
↪dPres );

    // compute etalon
    real64 const mob_et = dens[i] / visc[i];
    real64 const dMob_dPres_et = mob_et * (dDens_dPres[i] / dens[i] - dVisc_dPres[i] /␣
↪visc[i]);

    EXPECT_DOUBLE_EQ( mob, mob_et );
    EXPECT_DOUBLE_EQ( dMob_dPres, dMob_dPres_et );
  }
}
```

[Source: coreComponents/physicsSolvers/fluidFlow/unitTests/testSinglePhaseBaseKernels.cpp]

What makes this such a good test is that it depends on very little other than kernels themselves. There is no need to involve the data repository or parse an XML file. Sometimes however this is not possible, or at least not without a significant duplication of code. In this case it is better to embed the XML file into the test source as a string instead of creating a separate XML file and passing it to the test as a command line argument or hard coding the path. One example of this is `testLaplaceFEM` which tests the laplacian solver. The embedded XML is shown below.

```
char const * xmlInput =
  R"xml(
  <Problem>
    <Solvers gravityVector="{ 0.0, 0.0, -9.81 }">
      <CompositionalMultiphaseFVM name="compflow"
                                   logLevel="0"
                                   discretization="fluidTPFA"
                                   targetRegions="{region}"
                                   temperature="297.15"
                                   useMass="1">

        <NonlinearSolverParameters newtonTol="1.0e-6"
                                   newtonMaxIter="2"/>
        <LinearSolverParameters solverType="gmres"
                                krylovTol="1.0e-10"/>
      </CompositionalMultiphaseFVM>
    </Solvers>
    <Mesh>
      <InternalMesh name="mesh"
                    elementTypes="{C3D8}"
                    xCoords="{0, 3}"
                    yCoords="{0, 1}"
                    zCoords="{0, 1}"
                    nx="{3}"
                    ny="{1}"
                    nz="{1}"
                    cellBlockNames="{cb1}"/>
    </Mesh>
    <NumericalMethods>
      <FiniteVolume>
        <TwoPointFluxApproximation name="fluidTPFA"/>
      </FiniteVolume>
    </NumericalMethods>
    <ElementRegions>
      <CellElementRegion name="region" cellBlocks="{cb1}" materialList="{fluid, rock,␣
↪relperm, cappressure}" />
    </ElementRegions>
    <Constitutive>
      <CompositionalMultiphaseFluid name="fluid"
                                    phaseNames="{oil, gas}"
                                    equationsOfState="{PR, PR}"
                                    componentNames="{N2, C10, C20, H2O}"
                                    componentCriticalPressure="{34e5, 25.3e5, 14.6e5,␣
↪220.5e5}"
                                    componentCriticalTemperature="{126.2, 622.0, 782.0,␣
↪647.0}"
                                    componentAcentricFactor="{0.04, 0.443, 0.816, 0.344}"
                                    componentMolarWeight="{28e-3, 134e-3, 275e-3, 18e-3}"
                                    componentVolumeShift="{0, 0, 0, 0}"
                                    componentBinaryCoeff="{ {0, 0, 0, 0},
```

(continues on next page)

(continued from previous page)

```
                                                            {0, 0, 0, 0},
                                                            {0, 0, 0, 0},
                                                            {0, 0, 0, 0} }"/>
      <CompressibleSolidConstantPermeability name="rock"
          solidModelName="nullSolid"
          porosityModelName="rockPorosity"
          permeabilityModelName="rockPerm"/>
     <NullModel name="nullSolid"/>
     <PressurePorosity name="rockPorosity"
                       defaultReferencePorosity="0.05"
                       referencePressure = "0.0"
                       compressibility="1.0e-9"/>
      <BrooksCoreyRelativePermeability name="relperm"
                                       phaseNames="{oil, gas}"
                                       phaseMinVolumeFraction="{0.1, 0.15}"
                                       phaseRelPermExponent="{2.0, 2.0}"
                                       phaseRelPermMaxValue="{0.8, 0.9}"/>
      <BrooksCoreyCapillaryPressure name="cappressure"
                                    phaseNames="{oil, gas}"
                                    phaseMinVolumeFraction="{0.2, 0.05}"
                                    phaseCapPressureExponentInv="{4.25, 3.5}"
                                    phaseEntryPressure="{0., 1e8}"
                                    capPressureEpsilon="0.0"/>
    <ConstantPermeability name="rockPerm"
                          permeabilityComponents="{2.0e-16, 2.0e-16, 2.0e-16}"/>
    </Constitutive>
    <FieldSpecifications>
      <FieldSpecification name="initialPressure"
                 initialCondition="1"
                 setNames="{all}"
                 objectPath="ElementRegions/region/cb1"
                 fieldName="pressure"
                 functionName="initialPressureFunc"
                 scale="5e6"/>
      <FieldSpecification name="initialComposition_N2"
                 initialCondition="1"
                 setNames="{all}"
                 objectPath="ElementRegions/region/cb1"
                 fieldName="globalCompFraction"
                 component="0"
                 scale="0.099"/>
      <FieldSpecification name="initialComposition_C10"
                 initialCondition="1"
                 setNames="{all}"
                 objectPath="ElementRegions/region/cb1"
                 fieldName="globalCompFraction"
                 component="1"
                 scale="0.3"/>
      <FieldSpecification name="initialComposition_C20"
                 initialCondition="1"
                 setNames="{all}"
                 objectPath="ElementRegions/region/cb1"
```

(continues on next page)

(continued from previous page)

```
                 fieldName="globalCompFraction"
                 component="2"
                 scale="0.6"/>
      <FieldSpecification name="initialComposition_H2O"
                 initialCondition="1"
                 setNames="{all}"
                 objectPath="ElementRegions/region/cb1"
                 fieldName="globalCompFraction"
                 component="3"
                 scale="0.001"/>
    </FieldSpecifications>
    <Functions>
      <TableFunction name="initialPressureFunc"
                     inputVarNames="{elementCenter}"
                     coordinates="{0.0, 3.0}"
                     values="{1.0, 0.5}"/>
    </Functions>
  </Problem>
  )xml";
```

[Source: coreComponents/physicsSolvers/fluidFlow/unitTests/testCompMultiphaseFlow.cpp]

MPI

Often times it makes sense to write a unit test that is meant to be run with multiple MPI ranks. This can be accomplished by simply adding the NUM_MPI_TASKS parameter to blt_add_test in the CMake file. For example

```
blt_add_test( NAME testWithMPI
              COMMAND testWithMPI
              NUM_MPI_TASKS ${NUMBER_OF_MPI_TASKS} )
```

With this addition make test or calling ctest directly will run testWithMPI via something analogous to mpirun -n NUMBER_OF_MPI_TASKS testWithMPI.

Contributing Input Files

As part of the development cycle, functional input files should be introduced into the repository in order to provide 1) testing, 2) examples of proper use of the code. The input files should be modulerized such that the majority of the input resides in a base file, and the variations in input are contained in specific files for integrated/smoke testing, and benchmarking. For example if we had a single input file named myProblem.xml, we would break it up into myProblem_base.xml, myProblem_smoke.xml`', and ``myProblem_benchmark.xml. Each of the smoke/benchark files should include the base file using an include block as follows:

```
<Included>
  <File name="./myProblem_base.xml"/>
</Included>
```

The files should be placed in the appropriate application specific subdirectory under the GEOS/inputFiles directory. For example, the beamBending problem input files reside in the inputFiles/solidMechanics directory. The files then be linked to from the appropriate location in the integratedTests repository as described in the following section.

Integrated Tests

About

integratedTests is a submodule of *GEOS* residing at the top level of the directory structure. It defines a set of tests that will run *GEOS* with various *.xml* files and partitioning schemes using the Automated Test System (ATS). For each scenario, test performance is evaluated by comparing output files to baselines recorded in this repository.

Structure

The *integratedTests* repository includes a python package (*integratedTests/scripts/geos_ats_package*) and a directory containing test definitions (*integratedTests/tests/allTests*). A typical test directory will include an *.ats* file, which defines the behavior of tests or points to child test directories, symbolic links to any required *.xml* files, and a directory containing baseline files.

```
- integratedTests/
  - scripts/
    - geos_ats_package
- tests/
  - allTests/
    - main.ats/
    - sedov/
      - baselines/
        - sedov_XX/
          - <baseline-files>
      - sedov.ats
      - sedov.xml
```

High level integrated test results are recorded in the GEOS build directory: */path/to/GEOS/build-xyz/integratedTests/TestsResults*. These include *.log* and *.err* files for each test and a *test_results.html* summary file, which can be inspected in your browser.

Note: Baseline files are stored using the git LFS (large file storage) plugin. If you followed the suggested commands in the quickstart guide, then your environment is likely already setup.

However, if lfs is not yet activated, then the contents of baseline files may look something like this:

0to100_restart_000000100/rank_0000003.hdf5 version https://git-lfs.github.com/spec/v1 oid sha256:09bbe1e92968852cb915b971af35b0bba519fae7cf5934f3abc7b709ea76308c size 1761208

If this is the case, try running the following commands: `git lfs install` and `git lfs pull`.

How to Run the Tests

In most cases, integrated tests processes can be triggered in the GEOS build directory with the following commands:

- *make ats_environment* : Setup the testing environment (Note: this step is run by default for the other make targets). This process will install packages required for testing into the python environment defined in your current host config file. Depending on how you have built GEOS, you may be prompted to manually run the *make pygeosx* command and then re-run this step.
- *make ats_run* : Run all of the available tests (see the below note on testing resources).

- *make ats_clean* : Remove any unnecessary files created during the testing process (.vtk, .hdf5 files, etc.)
- *make ats_rebaseline* : Selectively update the baseline files for tests.
- *make ats_rebaseline_failed* : Automatically update the baseline files for any failed tests.

Note: The *make_ats_environment* step will attempt to collect python packages github and pypi, so it should be run from a machine with internet access.

Note: Running the integrated tests requires significant computational resources. If you are on a shared system, we recommend that you only run *make ats_run* within an allocation.

Note: We forward any arguments included in the *ATS_ARGUMENTS* cmake variable to the testing system. For example, on LLNL Lassen builds we select a couple of runtime options: set(ATS_ARGUMENTS "–ats jsrun_omp –ats jsrun_bind=packed" CACHE STRING "")

Note: When running test or creating new baselines on LC systems, we recommend that you use the *quartz-gcc-12-release* configuration

Override Test Behavior

For cases where you need additional control over the integrated tests behavior, you can use this script in your build directory: */path/to/GEOS/build-xyz/integratedTests/geos_ats.sh*. To run the tests, simply call this script with any desired arguments (see the output of *geos_ats.sh –help* for additional details.) Common options for this script include:

- -a/–action : The type of action to run. Common options include: *run*, *veryclean*, *rebaseline*, and *rebaselinefailed*.
- -r/–restartCheckOverrides : Arguments to pass to the restart check function. Common options include: *skip_missing* (ignores any new/missing values in restart files) and *exclude parameter1 parameter2* (ignore these values in restart files).
- –machine : Set the ats machine type name.
- –ats : Pass an argument to the underlying ats framework. Running *geos_ats.sh –ats help* will show you a list of available options for your current machine.

Machine Definitions

On many machines, ATS will automatically identify your machine's configuration and optimize it's performance. If the tests fail to run or to properly leverage your machine's resources, you may need to manually configure the machine. If you know the appropriate name for your machine in ATS (or the geos_ats package), then you can run *./geos_ats.sh –machine machine_name –ats help* to see a list of potential configuration options.

The *openmpi* machine is a common option for non-LC systems. For a system with 32 cores/node, an appropriate run command might look like:

```
./geos_ats.sh --machine openmpi --ats openmpi_numnodes 32 --ats openmpi_args=--report-
↪bindings --ats openmpi_args="--bind-to none" --ats openmpi_install "/path/to/openmpi/
↪installation"
```

Note: In this example, the path given by *openmpi_install* should include *bin/mpirun*.

Note: When you have identified a set of arguments that work for your machine, we recommend recording in the *ATS_ARGUMENTS* cmake variable in your system host config file.

Test Filtering

An arbitrary number of filter arguments can be supplied to ATS to limit the number of tests to be run. Filter arguments should refer to an ATS test variable and use a python-syntax (e.g.: "'some_string' in ats_variable" or "ats_variable<10"). These can be set via command-line arguments (possible via the *ATS_ARGUMENTS* variable):

```
./geos_ats.sh --ats f "np==1" --ats f "'SinglePhaseFVM' in solvers"
```

or via an environment variable (*ATS_FILTER*):

```
export ATS_FILTER="np==1,'SinglePhaseFVM' in solvers"
```

Common ATS variables that you can filter tests include:

- np : The number of parallel processes for the test
- label : The name of the test case (e.g.: "sedov_01")
- collection : The name of the parent test folder (e.g.: "contactMechanics")
- checks : A comma-separated list of checks (e.g.: "curve,restart")
- solvers : A comma-separated list of solver types (e.g.: "SinglePhaseFVM,SurfaceGenerator")
- outputs : A comma-separated list of output types (e.g.: "Restart,VTK")
- constitutive_models : A comma-separated list of constitutive model types (e.g.: "CompressibleSinglePhaseFluid,ElasticIsotropic")

Inspecting Test Results

While the tests are running, the name and size of the active test will be periodically printed out to the screen. Test result summaries will also be periodically written to the screen and files in */path/to/GEOS/build-xyz/integratedTests/TestsResults*. For most users, we recommend inspecting the *test_results.html* file in your browser (e.g.: *firefox integratedTests/TestsResults/test_results.html*). Tests will be organized by their status variable, which includes:

- *RUNNING* : The test is currently running
- *NOT RUN* : The test is waiting to start
- *PASSED* : The test and associated checks succeeded
- *FAIL RUN* : The test was unable to run (this often occurs when there is an error in the .ats file)
- *FAIL CHECK* : The test ran to completion, but failed either its restart or curve check
- *SKIPPED* : The test was skipped (likely due to lack of computational resources)

If each test ends up in the *PASSED* category, then you are likely done with the integrated testing procedure. However, if tests end up in any other category, it is your responsibility to address the failure. If you identify that a failure is due to an expected change in the code (e.g.: adding a new parameter to the xml structure or fixing a bug in an algorithm), you can follow the *rebaselining procedure*. Otherwise, you will need to track down and potentially fix the issue that triggered the failure.

Test Output

Output files from the tests will be stored in the TestResults directory (*/path/to/GEOS/build-xyz/integratedTests/TestsResults*) or in a subdirectory next to their corresponding *.xml* file (*integratedTests/tests/allTests/testGroup/testName_xx*). Using the serial beam bending test as an example, key output files include:

- *beamBending_01.data* : Contains the standard output for all test steps.
- *beamBending_01.err* : Contains the standard error output for all test steps.
- *displacement_history.hdf5* : Contains time history information that is used as an input to the curve check step.
- *totalDisplacement_trace.png* : A figure displaying the results of the curve check step.
- *beamBending.geos.out* : Contains the standard output for only the geos run step.
- *beamBending_restart_000000010.restartcheck* which holds all of the standard output for only the *restartcheck* step.
- *beamBending_restart_000000010.0.diff.hdf5* which mimmics the hierarchy of the restart file and has links to the

See *Restart Check* and *Curve Check* for further details on the test checks and output files.

Restart Check

This check compares a restart file output at the end of a run against a baseline. The python script that evaluates the diff is included in the *geos_ats* package, and is located here: *integratedTests/scripts/geos_ats_package/geos_ats/helpers/restart_check.py*. The script compares the two restart files and writes out a *.restart_check* file with the results, as well as exiting with an error code if the files compare differently. This script takes two positional arguments and a number of optional keyword arguments:

- file_pattern : Regex specifying the restart file. If the regex matches multiple files the one with the greater string is selected. For example *restart_100.hdf5* wins out over *restart_088.hdf5*.
- baseline_pattern : Regex specifying the baseline file.
- -r/–relative : The relative tolerance for floating point comparison, the default is 0.0.
- -a/–absolute : The absolute tolerance for floating point comparison, the default is 0.0.
- -e/–exclude : A list of regex expressions that match paths in the restart file tree to exclude from comparison. The default is [.*/commandLine].
- -w/-Werror : Force warnings to be treated as errors, default is false.
- -m/–skip-missing : Ignore values that are missing from either the baseline or target file.

The itself starts off with a summary of the arguments. The script begins by recording the arguments to the *.restart_check* file header, and then compares the *.root* restart files to their baseline. If these match, the script will compare the linked *.hdf5* data files to their baseline. If the script encounters any differences it will output an error message, and record a summary to the *.restart_check* file.

The restart check step can be run in parallel using mpi via

```
mpirun -n NUM_PROCESSES python -m mpi4py restartcheck.py ...
```

In this case rank zero reads in the restart root file and then each rank parses a subset of the data files creating a *.$RANK.restartcheck* file. Rank zero then merges the output from each of these files into the main *.restartcheck* file and prints it to standard output.

Scalar Error Example

An error message for scalar values looks as follows

```
Error: /datagroup_0000000/sidre/external/ProblemManager/domain/ConstitutiveManager/shale/
↪YoungsModulus
  Scalar values of types float64 and float64 differ: 22500000000.0, 10000022399.9.
```

Where the first value is the value in the test's restart file and the second is the value in the baseline.

Array Error Example

An example of an error message for arrays is

```
Error: /datagroup_0000000/sidre/external/ProblemManager/domain/MeshBodies/mesh1/Level0/
↪nodeManager/TotalDisplacement
  Arrays of types float64 and float64 have 1836 values of which 1200 have differing␣
↪values.
  Statistics of the differences greater than 0:
    max_index = (1834,), max = 2.47390764755, mean = 0.514503482629, std = 0.70212888881
```

This means that the max absolute difference is 2.47 which occurs at value 1834. Of the values that are not equal the mean absolute difference is 0.514 and the standard deviation of the absolute difference is 0.702.

When the tolerances are non zero the comparison is a bit more complicated. From the *FileComparison.compareFloatArrays* method documentation

```
Entries x1 and x2 are  considered equal iff
    |x1 - x2| <= ATOL or |x1 - x2| <= RTOL * |x2|.
To measure the degree of difference a scaling factor q is introduced. The goal is now to␣
↪minimize q such that
    |x1 - x2| <= ATOL * q or |x1 - x2| <= RTOL * |x2| * q.
If RTOL * |x2| > ATOL
    q = |x1 - x2| / (RTOL * |x2|)
else
    q = |x1 - x2| / ATOL.
If the maximum value of q over all the entries is greater than 1.0 then the arrays are␣
↪considered different and an error message is produced.
```

An sample error message is

```
Error: /datagroup_0000000/sidre/external/ProblemManager/domain/MeshBodies/mesh1/Level0/
↪nodeManager/TotalDisplacement
  Arrays of types float64 and float64 have 1836 values of which 1200 fail both the␣
↪relative and absolute tests.
    Max absolute difference is at index (1834,): value = 2.07474948094, base_value = 4.
```

(continues on next page)

(continued from previous page)

```
↪54865712848
    Max relative difference is at index (67,): value = 0.00215842135281, base_value = 0.
↪00591771127792
  Statistics of the q values greater than 1.0 defined by the absolute tolerance: N = 1200
    max = 16492717650.3, mean = 3430023217.52, std = 4680859258.74
  Statistics of the q values greater than 1.0 defined by the relative tolerance: N = 0
```

The *.diff.hdf5* File

Each error generated in the *restartcheck* step creates a group with three children in the *_diff.df5* file. For example the error given above will generate a hdf5 group

```
/FILENAME/datagroup_0000000/sidre/external/ProblemManager/domain/MeshBodies/mesh1/Level0/
↪nodeManager/TotalDisplacement
```

with datasets *baseline*, *run* and *message* where *FILENAME* is the name of the restart data file being compared. The *message* dataset contains a copy of the error message while *baseline* is a symbolic link to the baseline dataset and *run* is a sumbolic link to the dataset genereated by the run. This allows for easy access to the raw data underlying the diff without data duplication. For example if you want to extract the datasets into python you could do this:

```python
import h5py
file_path = "beamBending_restart_000000003_diff.hdf5"
path_to_data = "/beamBending_restart_000000011_0000000.hdf5/datagroup_0000000/sidre/
↪external/ProblemManager/domain/MeshBodies/mesh1/Level0/nodeManager/TotalDisplacement"
f = h5py.File("file_path", "r")
error_message = f["path_to_data/message"]
run_data = f["path_to_data/run"][:]
baseline_data = f["path_to_data/baseline"][:]

# Now run_data and baseline_data are numpy arrays that you may use as you see fit.
rtol = 1e-10
atol = 1e-15
absolute_diff = np.abs(run_data - baseline_data) < atol
hybrid_diff = np.close(run_data, baseline_data, rtol, atol)
```

When run in parallel each rank creates a *.$RANK.diff.hdf5* file which contains the diff of each data file processed by that rank.

Curve Check

This check compares time history (*.hdf5*) curves generated during GEOS execution against baseline and/or analytic solutions. In contrast to restart checks, curve checks are designed to be flexible with regards to things like mesh construction, time stepping, etc. The python script that evaluates the diff is included in the *geos_ats* package, and is located here: *integratedTests/scripts/geos_ats_package/geos_ats/helpers/curve_check.py*. The script renders the curve check results as a figure, and will throw an error if curves are out of tolerance. This script takes two positional arguments and a number of optional keyword arguments:

- filename : Path to the time history file.
- baseline : Path to the baseline file.
- -c/–curve : Add a curve to the check (value) or (value, setname). Multiple curves are allowed.

- -s/–script : Python script instructions for curve comparisons (path, function, value, setname)
- -t/–tolerance : The tolerance for each curve check diffs (||x-y||/N). Default is 0.
- -w/-Werror : Force warnings to be treated as errors, default is false.
- -o/–output : Output figures to this directory. Default is ./curve_check_figures
- -n/–n-column : Number of columns to use for the output figure. Default is 1.
- -u/–units-time : Time units for plots. Options include milliseconds, seconds (default), minutes, hours, days, years

The curve check script begins by checking the target time history file for expected key values. These include the time array ("value Time"), location array ("value ReferencePosition setname" or "value elementCenter setname"), and value array ("value setname"). Any missing values will be recorded as errors in the output.

The script will then run and record any user-requested python script instructions. To do this, python will attempt to import the file given by *path* and evaluate the target function, which should accept the time history data as keyword arguments. Note: to avoid side effects, please ensure that any imported scripts are appropriately guarded if they also allow direct execution:

```
if __name__ == '__main__':
  main()
```

This script will then check the size of the time history items, and will attempt to interpolate them if they do not match (currently, we only support interpolation in time). Finally, the script will compare the time history values to the baseline values and any script-generated values. If any curves do not match ($||x\text{-}y||/N > tol$), this will be recorded as an error.

Item Not Found Errors

The following error would indicate that the requested baseline file was not found:

```
baseline file not found: /path/to/baseline/file
```

This type of error can occur if you are adding a new test, or if you time history output failed.

The following errors would indicate that values were not found in time history files:

```
Value not found in target file: value
Set not found in target file: setname
Could not find location string for parameter: value, search...
```

The following error would indicate that a given curve exceeded its tolerance compared to script-generated values:

```
script_value_setname diff exceeds tolerance: ||t-b||/N=100.0, script_tolerance=1.0
```

Adding and Modifying Tests

ATS Configuration File

Files with the *.ats* extension are used to configure the integratedTests. They use a Python 3.x syntax, and have a set of ATS-related methods loaded into the scope (TestCase, geos, source, etc.). The root configuration file (*integratedTests/tests/allTests/main.ats*) finds and includes any test definitions in its subdirectories. The remaining configuration files typically add one or more tests with varying partitioning and input xml files to ATS.

The *integratedTests/tests/allTests/sedov/sedov.ats* file shows how to add three groups of tests. This file begins by defining a set of common parameters, which are used later:

It then enters over the requested partitioning schemes:

and registers a unique test case with the *TestCase* method, which accepts the following arguments:

- name : The name of the test. The expected convention for this variable is 'testName_N' (N = number of ranks) or 'testName_X_Y_Z' (X, Y, and Z ranks per dimension)
- desc : A brief description of the test
- label : The test label (typically 'auto')
- owner : The point(s) of contact for the test
- independent : A flag indicating whether the test is dependent on another test (typically True)
- steps: A tuple containing the test steps (minimum length = 1)

Test steps are run sequentially, and are created with the *geos* method. If a given test step fails, then it will produce an error and any subsequent steps will be canceled. This method accepts the following keyword arguments:

- deck : The name of the input xml file.
- np : The number of parallel processes required to run the step.
- ngpu : The number of GPU devices required to run the step. Note: this argument is typically ignored for geos builds/machines that are not configured to use GPU's. In addition, we typically expect that np=ngpu.
- x_partitions : The number of partitions to use along the x-dimension
- y_partitions : The number of partitions to use along the y-dimension
- z_partitions : The number of partitions to use along the z-dimension
- name : The header to use for output file names
- restartcheck_params : (optional) If this value is defined, run a restart check with these parameters (specified as a dictionary).
- curvecheck_params : (optional) If this value is defined, run a curve check with these parameters (specified as a dictionary).
- restart_file : (optional) The name of a restart file to resume from. To use this option, there must be a previous step that produces the selected restart file.
- baseline_pattern : (optional) The regex for the baseline files to use (required if the name of the step differs from the baseline)
- allow_rebaseline : A flag that indicates whether this step can be rebaselined. This is typically only true for the first step in a test case.

Note that a given *.ats* file can create any number of tests and link to any number of input xml files. For any given test step, we expect that at least one restart or curve check be defined.

Creating a New Test Directory

To add a new set of tests, create a new folder in the *integratedTests/tests/allTests** directory. This folder needs to include at least one *.ats* file to be included in the integrated tests. Using the sedov example, after creating *sedov.ats* the directory should look like

```
- integratedTests/tests/allTests/sedov/
  - sedov.ats
  - sedov.xml (this file should be a symbolic link to a GEOS input file located␣
↪somewhere within */path/to/GEOS/inputFiles*)
```

At this point you should run the integrated tests (in the build directory run: *make ats_run*). Assuming that the new *geos* step for your test case was successful, the subsequent *restartcheck* step will fail because the baselines have not yet been created. At this point the directory should look like this:

```
- integratedTests/tests/allTests/sedov/
  - sedov/
    - <geosx files>...
    - <ats files>...
  - sedov.ats
  - sedov.xml
  - <ats files>...
```

You can then follow the steps in the next section to record the initial baseline files.

Rebaselining Tests

Occasionally you may need to add or update baseline files in the repository (possibly due to feature changes in the code). This process is called rebaselining. We suggest the following workflow:

Step 1

In the GEOS repository, create or checkout a branch with your modifications:

```
cd /path/to/GEOS
git checkout -b user/feature/newFeature
```

Add your changes, confirm that they produce the expected results, and get approval for a pull request. If your branch needs to be rebaselined, make sure to indicate this in your pull request with the appropriate Label.

Step 2

Go to the integratedTests submodule, checkout and pull develop, and create a branch with the same name as the one in the main GEOS repository:

```
cd /path/to/GEOS/integratedTests
git checkout develop
git pull
git checkout -b user/feature/newFeature
```

Step 3

Go back to your GEOS build directory and run the integrated tests:

```
# Note: on shared machines, run these commands in an allocation
cd /path/to/GEOS/build-dir/
make ats_run
```

Inspect the test results that fail and determine which need to be **legitimately** rebaselined. Arbitrarily changing baselines defeats the purpose of the integrated tests. In your PR discussion, please identify which tests will change and any unusual behavior.

Step 4

We can then rebaseline the tests. In most cases, you will want to rebaseline all of the tests marked as **FAILED**. To do this you can run this command in the build directory:

```
make ats_rebaseline_failed
```

Otherwise, you can run the following command, and select whether tests should be rebaselined one at a time via a `[y/n]` prompt.

```
make ats_rebaseline_failed
```

Make sure to only answer `y` to the tests that you actually want to rebaseline, otherwise correct baselines for already passing tests will still be updated and bloat your pull request and repository size.

Step 5

Confirm that the new baselines are working as expected. You can do this by cleaning the test directories and re-running the tests:

```
# Note: on shared machines, run these commands in an allocation
cd /path/to/GEOS/build-dir/
make ats_clean
make ats_run
```

At this point you should pass all the integratedTests.

Step 6

Clean out unnecessary files and add new ones to the branch:

```
cd /path/to/GEOS/build-dir/
make ats_clean

# Check for new or modified files
cd /path/to/GEOS/integratedTests
git status

# Add new or modified files
git add file_a file_b ...
git commit -m "Updating baselines"
git push origin user/feature/newFeature
```

Step 6

If you haven't already done so, create a merge request for your integratedTests branch. Once you have received approval for your PR and are ready to continue, you can click merge the branch by clicking the button on github.

You should then checkout the develop branch of integratedTests and pull the new changes.

```
cd /path/to/GEOS/integratedTests
git checkout develop
git pull
```

You then need to update the integratedTests 'hash' in your associated GEOS branch.

```
cd /path/to/GEOS/
git add integratedTests
git commit -m "Updating the integratedTests hash"
git push origin user/feature/newFeature
```

At this point, you will need to wait for the CI/CD tests to run on github. After they succeed, you can merge your branch into develop using the button on github.

Tips

Parallel Tests: On some development machines geosxats won't run parallel tests by default (e.g. on an linux laptop or workstation), and as a result many baselines will be skipped. We highly recommend running tests and rebaselining on an MPI-aware platform.

Filtering Checks: A common reason for rebaselining is that you have changed the name of an XML node in the input files. While the baselines may be numerically identical, the restarts will fail because they contain different node names. In this situation, it can be useful to add a filter to the restart check script using the *geos_ats.sh* script (see the *-e* and *-m* options in Override Test Behavior)

Benchmarks

In addition to the integrated tests which track code correctness we have a suite of benchmarks that track performance.

Running the benchmarks

Because performance is system specific we currently only support running the benchmarks on the LLNL machines Quartz and Lassen. If you are on either of these machines the script `benchmarks/runBenchmarks.py` can be used to run the benchmarks.

```
> python ../benchmarks/runBenchmarks.py --help
usage: runBenchmarks.py [-h] [-t TIMELIMIT] [-o TIMINGCOLLECTIONDIR]
                        [-e ERRORCOLLECTIONDIR]
                        geosxPath outputDirectory

positional arguments:
  geosxPath             The path to the GEOS executable to benchmark.
  outputDirectory       The parent directory to run the benchmarks in.

optional arguments:
```

(continues on next page)

(continued from previous page)

```
  -h, --help            show this help message and exit
  -t TIMELIMIT, --timeLimit TIMELIMIT
                        Time limit for the entire script in minutes, the
                        default is 60.
  -o TIMINGCOLLECTIONDIR, --timingCollectionDir TIMINGCOLLECTIONDIR
                        Directory to copy the timing files to.
  -e ERRORCOLLECTIONDIR, --errorCollectionDir ERRORCOLLECTIONDIR
                        Directory to copy the output from any failed runs to.
```

At a minimum you need to pass the script the path to the GEOS executable and a directory to run the benchmarks in. This directory will be created if it doesn't exist. The script will collect a list of benchmarks to be run and submit a job to the system's scheduler for each benchmark. This means that you don't need to be in an allocation to run the benchmarks. Note that this is different from the integrated tests where you need to already be in an allocation and an internal scheduler is used to run the individual tests. Since a benchmark is a measure of performance to get consistent results it is important that each time a benchmark is run it has access to the same resources. Using the system scheduler guarantees this.

In addition to whatever outputs the input would normally produce (plot files, restart files, . . .) each benchmark will produce an output file `output.txt` containing the standard output and standard error of the run and a `.cali` file containing the Caliper timing data in a format that Spot can read.

Note: A future version of the script will be able to run only a subset of the benchmarks.

Specifying a benchmark

A group of benchmarks is specified with a standard GEOS input XML file with an extra `Benchmarks` block added at the top level. This block is ignored by GEOS itself and only used by the `runBenchmarks.py` script.

```
  <Benchmarks>
    <quartz>
      <Run
        name="OMP"
        nodes="1"
        tasksPerNode="1"
        timeLimit="10"
        autoPartition="On"/>
      <Run
        name="MPI_OMP"
        autoPartition="On"
        timeLimit="10"
        nodes="1"
        tasksPerNode="2"
        scaling="strong"
        scaleList="{ 1, 2, 4, 8 }"/>
      <Run
        name="MPI"
        autoPartition="On"
        timeLimit="10"
        nodes="1"
        tasksPerNode="36"
```

(continues on next page)

(continued from previous page)

```
      scaling="strong"
      scaleList="{ 1, 2, 4, 8 }"/>
  </quartz>

  <lassen>
    <Run
      name="OMP_CUDA"
      nodes="1"
      tasksPerNode="1"
      autoPartition="On"
      timeLimit="10"/>
    <Run
      name="MPI_OMP_CUDA"
      autoPartition="On"
      timeLimit="10"
      nodes="1"
      tasksPerNode="4"
      scaling="strong"
      scaleList="{ 1, 2, 4, 8 }"/>
  </lassen>
</Benchmarks>
```

[Source: benchmarks/SSLE-small.xml]

The `Benchmarks` block consists of a block for each machine the benchmarks are to run on. Currently the only options are `quartz`, `lassen`, and `crusher`.

The `Run` block

Each machine block contains a number of `Run` blocks each of which specify a family of benchmarks to run. Each `Run` block must have the following required attributes

- `name`: The name of the family of benchmarks, must be unique among all the other `Run` blocks on that system.
- `nodes`: An integer which specifies the base number of nodes to run the benchmark with.
- `tasksPerNode`: An integer that specifies the number of tasks to launch per node.

Each `Run` block may contain the following optional attributes

- `threadsPerTask`: An integer specifying the number of threads to allocate each task.
- `timeLimit`: An integer specifying the time limit in minutes to pass to the system scheduler when submitting the benchmark.
- `args`: containing any extra command line arguments to pass to GEOS.
- `autoPartition`: Either `On` or `Off`, not specifying `autoPartition` is equivalent to `autoPartition="Off"`. When auto partitioning is enabled the script will compute the number of `x`, `y` and `z` partitions such that the the resulting partition is close to a perfect cube as possible, ie with 27 tasks `x = 3, y = 3, z = 3` and with 36 tasks `x = 4, y = 3, z = 3`. This is optimal when the domain itself is a cube, but will be suboptimal otherwise.
- `strongScaling`: A list of unique integers specifying the factors to scale the number of nodes by. If `N` number are provided then `N` benchmarks are run and benchmark `i` uses `nodes * strongScaling[ i ]` nodes. Not specifying `strongScaling` is equivalent to `strongScaling="{ 1 }"`.

Looking at the example Benchmarks block above on Lassen one benchmark from the OMP_CUDA family will be run with one node and one task. Four benchmarks from the MPI_OMP_CUDA family will be run with one, two, four and eight nodes and four tasks per node.

Note that specifying a time limit for each benchmark family can greatly decrease the time spent waiting in the scheduler's queue. A good rule of thumb is that the time limit should be twice as long as it takes to run the longest benchmark in the family.

Adding a benchmark problem

To add a new group of benchmarks you need to create an XML input file describing the problem to be run. Then you need to add the Benchmarks block described above which specifies the specific benchmarks. Finally add a symbolic link to the input file in benchmarks and run the benchmarks to make sure everything works as expected.

Viewing the results

Each night the NightlyTests repository runs the benchmarks on both Quartz and Lassen, the timingFiles directory contains all of the resulting caliper output files. If you're on LC then these files are duplicated at /usr/gapps/GEOSX/timingFiles/ and if you have LC access you can view them in Spot. You can also open these files in Python and analyse them (See *Opening Spot caliper files in Python*).

If you want to run the benchmarks on your local branch and compare the results with develop you can use the benchmarks/compareBenchmarks.py python script. This requires that you run the benchmarks on your branch and on develop. It will print out a table with the initialization time speed up and run time speed up, so a run speed up of of 2x means your branch runs twice as fast as develop where as a initialization speed up of 0.5x means the set up takes twice as long.

Note: A future version of the script will be able to pull timing results straight from the .cali files so that if you have access to the NightlyTests timing files you won't need to run the benchmarks on develop. Furthermore it will be able to provide more detailed information than just initialization and run times.

Basic profiling with CALIPER

GEOS is equipped with Caliper timers. We integrate Caliper into GEOS by marking source-code sections of interest such as computational kernels or initialization steps. Caliper is included in the GEOS TPL library and is built by adding the following cmake configuration to a host-config file.

```
option( ENABLE_CALIPER "Enables CALIPER" On )
```

GEOS/Caliper Annotation Macros

The following macros may be used to annotate GEOS:

- GEOS_MARK_SCOPE(name) - Marks a scope with the given name.
- GEOS_MARK_FUNCTION - Marks a function with the name of the function. The name includes the namespace the function is in but not any of the template arguments or parameters. Therefore overloaded function all show up as one entry. If you would like to mark up a specific overload use GEOS_MARK_SCOPE with a unique name.
- GEOS_MARK_BEGIN(name) - Marks the beginning of a user defined code region.

- `GEOS_MARK_END(name)` - Marks the end of user defined code region.

The 2 first macros also generate annotations for NVTX is ENABLE_CUDA_NVTOOLSEXT is activated through CMake.

Configuring Caliper

Caliper configuration is done by specifying a string to initialize Caliper with via the *-t* option. A few options are listed below but we refer the reader to Caliper Config for the full Caliper tutorial.

- `-t runtime-report,max_column_width=200` Will make Caliper print aggregated timing information to standard out, with a column width large enought that it doesn't truncate most function names.
- `-t runtime-report,max_column_width=200,profile.cuda` Does the same as the above, but also instruments CUDA API calls. This is only an option when building with CUDA.
- `-t runtime-report,aggregate_across_ranks=false` Will make Caliper write per rank timing information to standard out. This isn't useful when using more than one rank but it does provide more information for single rank runs.
- `-t spot()` Will make Caliper output a *.cali* timing file that can be viewed in the Spot web server.

Using Adiak

Adiak is a library that allows the addition of meta-data to the Caliper Spot output, it is enabled with Caliper. This meta-data allows you to easily slice and dice the timings available in the Spot web server. To export meta-data use the *adiak::value* function.

See Adiak API for the full Adiak documentation.

Using Spot

To use Spot you will need an LC account and a directory full of *.cali* files you would like to analyse. Point your browser to Spot and open up the directory containing the timing files.

Opening Spot caliper files in Python

An example Python program for analyzing Spot Caliper files in Python is provided below. Note that it requires `pandas` and `hatchet` both of which can be installed with a package manager. In addition it requires that `cali-query` is in the `PATH` variable, this is built with Caliper so we can just point it into the TPLs.

```python
import sys
import subprocess
import json
import os

import pandas as pd
from IPython.display import display, HTML

# Import hatchet, on LC this can be done by adding hatchet to PYTHONPATH
sys.path.append('/usr/gapps/spot/live/hatchet')
import hatchet as ht
```

(continues on next page)

(continued from previous page)

```
# Add cali-query to PATH
cali_query_path = "/usr/gapps/GEOSX/thirdPartyLibs/2020-06-12/install-quartz-gcc@8.1.0-
↪release/caliper/bin"
os.environ["PATH"] += os.pathsep + cali_query_path

CALI_FILES = [
{ "cali_file": "/usr/gapps/GEOSX/timingFiles/200612-04342891243.cali", "metric_name":
↪"avg#inclusive#sum#time.duration"},
{ "cali_file": "/usr/gapps/GEOSX/timingFiles/200611-044740108300.cali", "metric_name":
↪"avg#inclusive#sum#time.duration"},
]

grouping_attribute = "prop:nested"
default_metric = "avg#inclusive#sum#time.duration"
query = "select %s,sum(%s) group by %s format json-split" % (grouping_attribute, default_
↪metric, grouping_attribute)

gf1 = ht.GraphFrame.from_caliper(CALI_FILES[0]['cali_file'], query)
gf2 = ht.GraphFrame.from_caliper(CALI_FILES[1]['cali_file'], query)

# Print the tree representation using the default metric
# Also print the resulting dataframe with metadata
print(gf1.tree(color=True, metric="sum#"+default_metric))
display(HTML(gf1.dataframe.to_html()))

# Print the tree representation using the default metric
# Also print the resulting dataframe with metadata
print(gf2.tree(color=True, metric="sum#"+default_metric))
display(HTML(gf2.dataframe.to_html()))

# Compute the speedup between the first two cali files (exlusive and inclusive metrics␣
↪only)
gf3 = (gf1 - gf2) / gf2
print(gf3.tree(color=True, metric="sum#"+default_metric))

# Compute the difference between the first two cali files (exclusive and inclusive␣
↪metrics only)
# Print the resulting tree
gf4 = gf1 - gf2
print(gf4.tree(color=True, metric="sum#"+default_metric))

# Compute the sum of the first two cali files (exclusive and inclusive metrics only)
# Print the resulting tree
gf5 = gf1 + gf2
print(gf5.tree(color=True, metric="sum#"+default_metric))
```

[Unsupported] Developing inside Docker with precompiled TPL binaries

For development purposes, you may want to use the publicly available docker images instead of compiling them yourself. While this is possible and this page will help you in through this journey, please note that *this is not officially supported by the GEOS team that reserves the right to modify its workflow or delete elements on which you may have build your own workflow*.

There are multiple options to use the exposed docker images.

- A lot of IDE now provide remote development modes (e.g. CLion, VS Code, Eclipse Che and surely others). Depending on your choice, please read their documentation carefully so you can add their own requirements on top the TPL images that are already available.
- Another option is to develop directly inside the container (*i.e.* not remotely). Install your favorite development inside the image (be mindful of X display issues), connect to the running container and start hacking!
- It is also possible to develop directly in the cloud using GitHub codespaces. This product will let you buy a machine in the cloud with an environment already configured to build and run `geos`. To make it work, create a branch and/or fork the repository, then before creating your `codespace` instance, control the version of the TPL you need (defined by `build.args.GEOS_TPL_TAG` in .devcontainer/devcontainer.json). (You may most probably stick to the `GEOSX_TPL_TAG` of the `.github/workflows/ci_tests.yml` file). The submodules are automatically cloned (expect for the `integratedTests` which you may need to `init` yourself if you really need them). You do not need to run the `scripts/config-build.py` scripts since `cmake` and `vscode` are already configured. Last, run `cmake` through the `vscode` interface and start hacking!

You must first install docker on your machine. Note that there now exists a rootless install that may help you in case you are not granted extended permissions on your environment. Also be aware that nvidia provides its own nvidia-docker that grants access to GPUs.

Once you've installed docker, you must select from our docker registry the target environment you want to develop into.

- You can select the distribution you are comfortable with, or you may want to mimic (to some extend) a production environment.
- Our containers are built with a relative CPU agnosticism (still `x86_64`), so you should be fine.
- Our GPU containers are built for a dedicated `compute capability` that may not match yours. Please dive into our configuration files and refer to the official nvidia page to see what matches your needs.
- There may be risks of kernel inconsistency between the container and the host, but if you have relatively modern systems (and/or if you do not interact directly with the kernel like `perf`) it should be fine.
- You may have noticed that our docker containers are tagged like `224-965`. Please refer to *Continuous Integration process* for further information.

Now that you've selected your target environment, you must be aware that just running a TPL docker image is not enough to let you develop. You'll have to add extra tools.

The following *example* is for our `ubuntu` flavors. You'll notice the arguments `IMG`, `VERSION`, `ORG`. While surely overkill for most cases, if you develop in GEOS on a regular basis you'll appreciate being able to switch containers easily. For example, simply create the image `remote-dev-ubuntu20.04-gcc9:224-965` by running

```
export VERSION=224-965
export IMG=ubuntu20.04-gcc9
export REMOTE_DEV_IMG=remote-dev-${IMG}
docker build --build-arg ORG=geosx --build-arg IMG=${IMG} --build-arg VERSION=${VERSION}␣
↪-t ${REMOTE_DEV_IMG}:${VERSION} -f /path/to/Dockerfile .
```

And the `Dockerfile` is the following (comments are embedded)

```
# Define you base image for build arguments
ARG IMG
ARG VERSION
ARG ORG
FROM ${ORG}/${IMG}:${VERSION}

# Uninstall some packages, install others.
# I use those for clion, but VS code would have different requirements.
# Use yum's equivalent commands for centos/red-hat images.
# Feel free to adapt.
RUN apt-get update
RUN apt-get remove --purge -y texlive graphviz
RUN apt-get install --no-install-recommends -y openssh-server gdb rsync gdbserver ninja-build

# You may need to define your time zone. This is a way to do it. Please adapt to your own needs.
RUN ln -fs /usr/share/zoneinfo/America/Los_Angeles /etc/localtime && \
    dpkg-reconfigure -f noninteractive tzdata

# You will need cmake to build GEOSX.
ARG CMAKE_VERSION=3.23.3
RUN apt-get install -y --no-install-recommends curl ca-certificates && \
    curl -fsSL https://cmake.org/files/v${CMAKE_VERSION%.[0-9]*}/cmake-${CMAKE_VERSION}-linux-x86_64.tar.gz | tar --directory=/usr/local --strip-components=1 -xzf - && \
    apt-get purge --auto-remove -y curl ca-certificates
RUN apt-get autoremove -y

# You'll most likely need ssh/sshd too (e.g. CLion and VSCode allow remote dev through ssh).
# This is the part where I configure sshd.

# The default user is root. If you plan your docker instance to be a disposable environment,
# with no sensitive information that a split between root and normal user could protect,
# then this is a choice which can make sense. Make your own decision.
RUN echo "PermitRootLogin prohibit-password" >> /etc/ssh/sshd_config
RUN echo "PermitUserEnvironment yes" >> /etc/ssh/sshd_config
RUN mkdir -p -m 700 /root/.ssh
# Put your own public key here!
RUN echo "ssh-rsa AAAAB... your public ssh key here ...EinP5Q== somebody@somewhere.org" > /root/.ssh/authorized_keys

# Some important variables are provided through the environment.
# You need to explicitly tell sshd to forward them.
# Using these variables and not paths will let you adapt to different installation locations in different containers.
# Feel free to adapt to your own convenience.
RUN touch /root/.ssh/environment &&\
    echo "CC=${CC}" >> /root/.ssh/environment &&\
    echo "CXX=${CXX}" >> /root/.ssh/environment &&\
    echo "MPICC=${MPICC}" >> /root/.ssh/environment &&\
    echo "MPICXX=${MPICXX}" >> /root/.ssh/environment &&\
```

(continues on next page)

(continued from previous page)

```
    echo "MPIEXEC=${MPIEXEC}" >> /root/.ssh/environment &&\
    echo "OMPI_CC=${CC}" >> /root/.ssh/environment &&\
    echo "OMPI_CXX=${CXX}" >> /root/.ssh/environment &&\
    echo "GEOSX_TPL_DIR=${GEOSX_TPL_DIR}" >> /root/.ssh/environment
# If you decide to work as root in your container, you may consider adding
# `OMPI_ALLOW_RUN_AS_ROOT=1` and `OMPI_ALLOW_RUN_AS_ROOT_CONFIRM=1`
# to your environment. This will prevent you from adding the `--allow-run-as-root` option
# when you run mpi. Of course, weigh the benefits and risks and make your own decision.

# Default ssh port 22 is exposed. For _development_ purposes,
# it can be useful to expose other ports for remote tools.
EXPOSE 22 11111 64010-64020
# sshd's option -D prevents it from detaching and becoming a daemon.
# Otherwise, sshd would not block the process and `docker run` would quit.
RUN mkdir -p /run/sshd
ENTRYPOINT ["/usr/sbin/sshd", "-D"]
```

Now that you've created the image, you must instantiate it as a container. I like to do

```
docker run --cap-add=SYS_PTRACE -d --name ${REMOTE_DEV_IMG}-${VERSION} -p 64000:22 -p 11111:11111 -p 64010-64020:64010-64020 ${REMOTE_DEV_IMG}:${VERSION}
```

that creates the container `remote-dev-ubuntu20.04-gcc9-224-965`, running instance of `remote-dev-ubuntu20.04-gcc9:224-965`.

- Note that you'll have to access your remote development instance though port `64000` (forwarded to standard port `22` by docker).
- Additional port `11111` and ports from `64010` to `64020` will be open if you need them (remote paraview connection , multiple instances of gdbserver, . . .).
- Please be aware of how to retrieve your code back: you may want to bind mount volumes and store you code there (`-v`/`--volume=` options of docker run).
- Change `docker` to `nvidia-docker` and add the `--gpus=...` option for GPUs.

You can stop and restart your container with

```
docker stop ${REMOTE_DEV_IMG}-${VERSION}
docker start ${REMOTE_DEV_IMG}-${VERSION}
```

Now hack.

[Unsupported] Installing GEOS on Windows machines using Docker

In this section, we will install GEOS on a Windows machine using a `Docker` container with a precompiled version of GEOS's third party libraries (TPL). These steps are an adaptation of *ref:UsingDocker* for the Windows environment. In the following sections, we will be using *Ubuntu* based image as an example.

1. Install *Docker Desktop*

On your Windows machine, follow these steps. Download the most recent installer for *Docker Desktop*. Before installation please check the current status of Windows Subsystem for Linux (WSL) on your machine as Docker will use WSL2 as a backend. To do that, open a PowerShell(Admin)

```
PS > wsl --install
PS > wsl --status
PS > wsl --set-default-version 2
PS > wsl --status
```

The first command should install *WSL2*, download an *Ubuntu* distribution for it and ask for a restart. The following commands are used to check the status, and if the WSL is still the default one, change it to WSL2. More details on the installation procedure can be found here.

Once the WSL2 is set as default, proceed with the *Docker Desktop* installation.

2. Start *Docker Desktop*

When launching Docker Desktop for the first time, you should be prompted with a message informing you that it uses WSL2. Using PowerShell, you can check that Docker and WSL2 are actually running in the background:

```
PS > Get-Process docker
Handles  NPM(K)    PM(K)      WS(K)     CPU(s)     Id  SI ProcessName
-------  ------    -----      -----     ------     --  -- -----------
    123      11    26084      25608       0.42  13960   1 docker

PS > Get-Process wsl
Handles  NPM(K)    PM(K)      WS(K)     CPU(s)     Id  SI ProcessName
-------  ------    -----      -----     ------     --  -- -----------
    146       8     1412       6848       0.05  14816   1 wsl
    146       7     1356       6696       0.02  15048   1 wsl
    145       7     1368       6704       0.02  15100   1 wsl
    145       7     1352       6716       0.14  15244   1 wsl
    146       7     1396       6876       0.02  16156   1 wsl
```

You should be able to see one docker process and several *wsl* processes.

3. Preparing *DockerFile*

Let us now prepare the installation, picking a destination folder and editing our Dockerfile:

```
PS > cd D:/
PS > mkdir install-geosx-docker
PS > cd install-geosx-docker/
PS > notepad.exe Dockerfile
```

Let us edit the Dockerfile, which is the declarative file for out container:

```
# Define you base image for build arguments
ARG IMG
ARG VERSION
```

(continues on next page)

(continued from previous page)

```
ARG ORG
FROM ${ORG}/${IMG}:${VERSION}

# Uninstall some packages, install others.
# I use those for clion, but VS code would have different requirements.
# Use yum's equivalent commands for centos/red-hat images.
# Feel free to adapt.
RUN apt-get update
RUN apt-get remove --purge -y texlive graphviz
RUN apt-get install --no-install-recommends -y openssh-server gdb rsync gdbserver ninja-build

# You may need to define your time zone. This is a way to do it. Please adapt to your own needs.
RUN ln -fs /usr/share/zoneinfo/America/Los_Angeles /etc/localtime && \
    dpkg-reconfigure -f noninteractive tzdata

# You will need cmake to build GEOSX.
ARG CMAKE_VERSION=3.23.3
RUN apt-get install -y --no-install-recommends curl ca-certificates && \
    curl -fsSL https://cmake.org/files/v${CMAKE_VERSION%.[0-9]*}/cmake-${CMAKE_VERSION}-linux-x86_64.tar.gz | tar --directory=/usr/local --strip-components=1 -xzf - && \
    apt-get purge --auto-remove -y curl ca-certificates
RUN apt-get autoremove -y

# You'll most likely need ssh/sshd too (e.g. CLion and VSCode allow remote dev through ssh).
# This is the part where I configure sshd.

# The default user is root. If you plan your docker instance to be a disposable environment,
# with no sensitive information that a split between root and normal user could protect,
# then this is a choice which can make sense. Make your own decision.
RUN echo "PermitRootLogin prohibit-password" >> /etc/ssh/sshd_config
RUN echo "PermitUserEnvironment yes" >> /etc/ssh/sshd_config
RUN mkdir -p -m 700 /root/.ssh
# Put your own public key here!
RUN echo "ssh-rsa AAAAB... your public ssh key here ...EinP5Q== somebody@somewhere.org" > /root/.ssh/authorized_keys

# Some important variables are provided through the environment.
# You need to explicitly tell sshd to forward them.
# Using these variables and not paths will let you adapt to different installation locations in different containers.
# Feel free to adapt to your own convenience.
RUN touch /root/.ssh/environment &&\
    echo "CC=${CC}" >> /root/.ssh/environment &&\
    echo "CXX=${CXX}" >> /root/.ssh/environment &&\
    echo "MPICC=${MPICC}" >> /root/.ssh/environment &&\
    echo "MPICXX=${MPICXX}" >> /root/.ssh/environment &&\
    echo "MPIEXEC=${MPIEXEC}" >> /root/.ssh/environment &&\
    echo "OMPI_CC=${CC}" >> /root/.ssh/environment &&\
```

(continues on next page)

(continued from previous page)

```
    echo "OMPI_CXX=${CXX}" >> /root/.ssh/environment &&\
    echo "GEOSX_TPL_DIR=${GEOSX_TPL_DIR}" >> /root/.ssh/environment
# If you decide to work as root in your container, you may consider adding
# `OMPI_ALLOW_RUN_AS_ROOT=1` and `OMPI_ALLOW_RUN_AS_ROOT_CONFIRM=1`
# to your environment. This will prevent you from adding the `--allow-run-as-root` option
# when you run mpi. Of course, weigh the benefits and risks and make your own decision.

# Default ssh port 22 is exposed. For _development_ purposes,
# it can be useful to expose other ports for remote tools.
EXPOSE 22 11111 64010-64020
# sshd's option -D prevents it from detaching and becoming a daemon.
# Otherwise, sshd would not block the process and `docker run` would quit.
RUN mkdir -p /run/sshd
ENTRYPOINT ["/usr/sbin/sshd", "-D"]
```

This file enriches a base image already containing the GEOS's TPL as well as extra utils, such as `cmake` and preparing for ssh connexion. In the end, we will be able to run it in a detached mode, and connect to it to run and develop in GEOS.

There are two things you may have noticed reading through the `Dockerfile` :

- It has environment variables to be passed to it to select the proper image to pull, namely `${ORG}`, `${IMG}` and `${VERSION}`, we'll then have to declare them

```
PS> $env:VERSION='224-965'
PS> $env:IMG='ubuntu20.04-gcc9'
PS> $env:REMOTE_DEV_IMG="remote-dev-${env:IMG}"
```

Please note the preposition of `env:` in the windows formalisme. The `${ORG}` variable will be hard-coded as `geosx`. The last variable will be use as the image name. `224-965` refers to a specific version of the TPLs which may not be up to date. Please refer to *Continuous Integration process* for further info.

- You'll need to generate a ssh-key to be able to access the container without the need for defining a password. This can be done from the *PowerShell*,

```
PS > ssh-keygen.exe
PS > cat [path-to-gen-key]/[your-key].pub
```

The first command will prompt you with a message asking you to complete the desired path for the key as well as a passphrase, with confirmation. More details on ssh-key generation.

4. Build the image and run the container

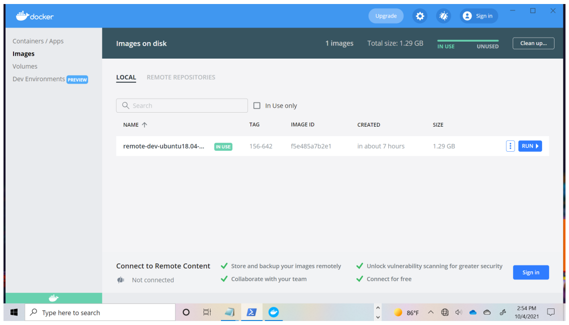
The preliminary tasks are now done. Let us build the image that will be containerized.

```
PS> cd [path-to-dockerfile-folder]/
PS > docker build --build-arg ORG=geosx --build-arg IMG=${env:IMG} --build-arg VERSION=$
↪{env:VERSION} -t ${env:REMOTE_DEV_IMG}:${env:VERSION} -f Dockerfile .
```

As described above, we are passing our environment variables in the building stage, which offer the flexibility of changing the version or image by a simple redefinition. A log updating or pulling the different layers should be displayed afterwards and on the last line the *image id*. We can check that the image is created using `PowerShell` CLI:

```
PS > docker images
```

or using the *Docker Desktop*



Images on disk — 1 images — Total size: 1.29 GB

LOCAL — REMOTE REPOSITORIES

□ In Use only

NAME ↑	TAG	IMAGE ID	CREATED	SIZE
remote-dev-ubuntu18.04-... IN USE	156-642	f5e485a7b2e1	in about 7 hours	1.29 GB

Connect to Remote Content — Not connected

✓ Store and backup your images remotely

✓ Collaborate with your team

✓ Unlock vulnerability scanning for greater security

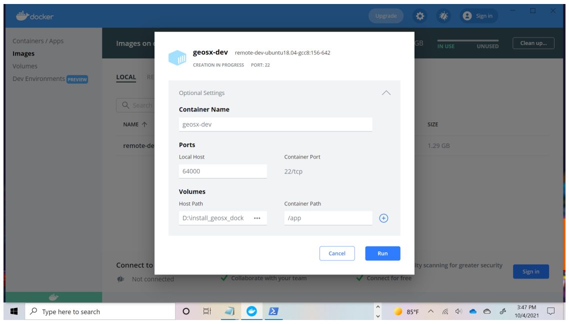
✓ Connect for free

Now that we have the image build, let us run a container from,

```
PS > docker run --cap-add=SYS_PTRACE  -d --name ${env:REMOTE_DEV_IMG}-${env:VERSION} -p 64000:22 --mount 'type=bind,source=D:/install_geosx_docker/,target=/app' ${env:REMOTE_DEV_IMG}:${env:VERSION}
```

Note that in addition to the detached flag (`-d`) and the name tage (`--name`), we provide `Docker` with the port the container should be associated to communicate with ssh port 22, as well as a binding between a host mount point (`D:/install_geosx_docker/`) and a container mount point (`/app`) to have a peristent storage for our development/geosx builds. More details on the –mount options

A similar step can be achieved using the *Docker Desktop* GUI in the image tabs, clicking on the *run* button and filling the same information in the interface,



geosx-dev remote-dev-ubuntu18.04-gcc8:156-642

CREATION IN PROGRESS PORT: 22

Optional Settings

Container Name: geosx-dev

Ports — Local Host: 64000 — Container Port: 22/tcp

Volumes — Host Path: D:\install_geosx_dock — Container Path: /app

Coming back to our `PowerShell` terminal, we can check that our container is running and trying to ssh to it.

```
PS > docker ps -a
CONTAINER ID   IMAGE                                COMMAND                CREATED              STATUS          PORTS                                   NAMES
1efffac66c4c   remote-dev-ubuntu20.04-gcc9:224-965   "/usr/sbin/sshd -D"   Less than a second ago   Up 18 seconds   0.0.0.0:64000->22/tcp, :::64000->22/tcp   remote-dev-ubuntu20.04-gcc9-224-965

PS > ssh root@localhost -p 64000
Enter passphrase for key 'C:\***********/.ssh/id_rsa':
Welcome to Ubuntu 18.04.5 LTS (GNU/Linux 5.10.16.3-microsoft-standard-WSL2 x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
```

(continues on next page)

(continued from previous page)

```
 * Support:        https://ubuntu.com/advantage
This system has been minimized by removing packages and contents that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

root@b105f9ead860:~# cd /app && ls
```

We are now logged into our container and can start *Quick Start Guide*.

Note: Note that :

1. You might be prompted that you miss certificates to clone, this can be resolved by installing *ca-certificates* and updating them
2. It might occur that *git-lfs* is missing then install it

```
PS > apt install ca-certificates && update-ca-certificates
PS > apt install git-lfs
```

From there you should be able to develop in your container or access it from an IDE, e.g. VSCode or MSVC19.

5. Running a case

Once the code is configured and compiled, let us check the status of the build,

```
root@b105f9ead860:~# cd [path-to-build]/ && ./bin/geosx --help
```

Trying to launch a case using `mpirun`, you might get the following warning

```
root@b105f9ead860:/tmp# mpirun -np 4 /app/code/GEOS/build-environment-debug/bin/geosx -i␣
↪[geosx-case].xml -x 4 -y1 -z1
--------------------------------------------------------------------------
mpirun has detected an attempt to run as root.
Running at root is *strongly* discouraged as any mistake (e.g., in
defining TMPDIR) or bug can result in catastrophic damage to the OS
file system, leaving your system in an unusable state.

You can override this protection by adding the --allow-run-as-root
option to your cmd line. However, we reiterate our strong advice
against doing so - please do so at your own risk.
--------------------------------------------------------------------------
```

A possible workaround is to create a new user account and a run folder from this account

```
root@b105f9ead860:~# adduser runner
root@b105f9ead860:~# su runner
runner@b105f9ead860:/root$ mkdir -p /tmp/geosx && cd /tmp/geosx/
runner@b105f9ead860:/tmp/geosx$ cp [path-to-case]/[geosx-case].xml .
runner@b105f9ead860:/tmp/geosx$ ${MPIEXEC} -np 4 /app/code/GEOS/build-environment-debug/
↪bin/geosx -i [geosx-case].xml -x 4 -y 1 -z 1
```

1.6.2 Code Components

The main code components are described here.

Data Repository

The GEOS "Data Repository" is intended to provide the building blocks for the code structure within GEOS. The "Data Repository" provides a general capability to store arbitrary data and objects in a hierarchical structure, similar to a standard file system.

The components/classes of the data structure that a developer will require some knowledge of are:

Mapped Vector

A superposition of a contiguous and an associative container.

Description

The container stores pointers to objects (which are themselves heap-allocated). Each element may be optionally *owned* by the container, in which case it will be deleted upon removal or container destruction. The pointers are stored in a contiguous memory allocation, and thus are accessible through an integral index lookup. In addition, there is a map that provides a key lookup capability to the container if that is the preferred interface.

The container template has four type parameters:

- `T` is the object type pointed to by container entries
- `T_PTR` is a pointer-to-`T` type which must be either `T *` (default) or `std::unique_ptr<T>`
- `KEY_TYPE` is the type of key used in associative lookup
- `INDEX_TYPE` is the type used in index lookup

Element access

`MappedVector` provides three main types of data access using `[]` operator:

- **Index lookup** is the fastest way of element random access if the ordinal index is known.
- **Key lookup** is similar to key lookup of any associative container and incurs similar cost.
- **KeyIndex lookup** uses a special type, `KeyIndex`, that contains both a key and an index. Initially the index is unknown and the key is used for the lookup. The `KeyIndex` is modified during lookup, storing the index located. If the user persists the `KeyIndex` object, they may reuse it in subsequent accesses and get the benefit of direct index access.

In addition to these, an STL-conformant iterator interface is available via `begin()` and `end()` methods. The type iterated over is a key-pointer pair (provided as *value_type* alias).

API documentation

MappedVector

Group

`dataRepository::Group` serves as a base class for most objects in GEOS. In GEOS, the `Group` may be thought of as an analogy to the file folder in a hierachical filesystem-like structure. As such, a `Group` is used as a container class that holds a collection of other Groups, or sub-Groups, a pointer to the parent of the Group, and a collection of Wrappers. The `Group` also defines a general capability to create and traverse/access the objects in the hierarchy. The Wrappers contained in a Group may be of arbitrary type, but in the case of an LvArray object, a Group size and capacity may be translated down to array, thereby keeping a collection of wrapped Array objects that will resize in unison with the Group. Each group has a string "name" that defines its key in the parent Group. This key string must be unique in the scope of the parent Group.

Implementation Details

Some noteworthy implementation details inside the declaration of `dataRepository::Group` are:

```
/// The default key type for entries in the hierarchy.
using keyType = string;

/// The default index type for entries the hierarchy.
using indexType = localIndex;
```

- In the GEOS repository, the `keyType` is specified to be a `string` for all collection objects, while the `indexType` is specified to be a `localIndex`. The types are set in the `common/DataTypes.hpp` file, but are typically a `string` and a `std::ptrdiff_t` respectively.

```
  /// The template specialization of MappedVector to use for the collection of sub-Group␣
↪objects.
  using subGroupMap = MappedVector< Group, Group *, keyType, indexType >;

  /// The template specialization of MappedVector to use for the collection wrappers␣
↪objects.
  using wrapperMap = MappedVector< WrapperBase, WrapperBase *, keyType, indexType >;
```

- The `subGroupMap` and `wrapperMap` aliases represent the type of container that the collection of sub-`Group` s and `Wrapper` s are stored in for each `Group`. These container types are template specializations of the `MappedVector` class, which store a pointer to a type, and provides functionality for a key or index based lookup. More details may be found in the documentation for `MappedVector`.

```
  /// The parent Group that contains "this" Group in its "sub-Group" collection.
  Group * m_parent = nullptr;

  /// Specification that this group will have the same m_size as m_parent.
  integer m_sizedFromParent;

```

(continues on next page)

(continued from previous page)

```
  /// The container for the collection of all wrappers continued in "this" Group.
  wrapperMap m_wrappers;

  /// The container for the collection of all sub-groups contained in "this" Group.
  subGroupMap m_subGroups;

  /// The size/length of this Group...and all Wrapper<> that are are specified to have the same size as their
  /// owning group.
  indexType m_size;

  /// The capacity for wrappers in this group...and all Wrapper<> that are specified to have the same size as their
  /// owning group.
  indexType m_capacity;

  /// The name/key of this Group in its parent collection of sub-Groups.
  string m_name;

  /// Verbosity flag for group logs
  integer m_logLevel;
```

- The `m_parent` member is a pointer to the `Group` that contains the current `Group` as part of its collection of sub-`Group` s.

> **Warning:** The existence of the non-const `m_parent` gives the current `Group` access to alter the parent Group. Special care should be taken to avoid using this access whenever possible. Remember…with great power comes great responsibility.

- The `m_wrappers` member is the collection of Wrappers contained in the current `Group`.
- The `m_subGroups` member is the collection of `Group` s contained in the current `Group`.
- The `m_size` and `m_capacity` members are used to set the size and capacity of any objects contained in the `m_wrappers` collection that have been specified to be set by their owning `Group`. This is typically only useful for Array types and is implemented within the `WrapperBase` object.
- The `m_name` member is the key of this Group in the collection of `m_parent->m_subGroups`. This key is unique in the scope of `m_parent`, so some is required when constructing the hierarchy.

Interface Functions

The public interface for `dataRepository::Group` provides functionality for constructing a hierarchy, and traversing that hierarchy, as well as accessing the contents of objects stored in the `Wrapper` containers stored within a `Group`.

Adding New Groups

To add new sub-`Group` s there are several `registerGroup` functions that add a new `Group` under the calling `Group` scope. A listing of these functions is provided:

```
/**
 * @name Sub-group registration interface
 */
///@{

/**
 * @brief Register a new Group as a sub-group of current Group.
 *
 * @tparam T The type of the Group to add/register. This should be a type that derives from Group.
 * @param[in] name      The name of the group to use as a string key.
 * @param[in] newObject A unique_ptr to the object that is being registered.
 * @return              A pointer to the newly registered Group.
 *
 * Registers a Group or class derived from Group as a subgroup of this Group and takes ownership.
 */
template< typename T = Group >
T & registerGroup( string const & name, std::unique_ptr< T > newObject )
{
  newObject->m_parent = this;
  return dynamicCast< T & >( *m_subGroups.insert( name, newObject.release(), true ) );
}

/**
 * @brief @copybrief registerGroup(string const &,std::unique_ptr<T>)
 *
 * @tparam T The type of the Group to add/register. This should be a type that derives from Group.
 * @param[in] name        The name of the group to use as a string key.
 * @param[in] newObject   A unique_ptr to the object that is being registered.
 * @return                A pointer to the newly registered Group.
 *
 * Registers a Group or class derived from Group as a subgroup of this Group but does not take ownership.
 */
template< typename T = Group >
T & registerGroup( string const & name, T * newObject )
{ return dynamicCast< T & >( *m_subGroups.insert( name, newObject, false ) ); }


/**
```

(continues on next page)

(continued from previous page)

```
   * @brief @copybrief registerGroup(string const &,std::unique_ptr<T>)
   *
   * @tparam T The type of the Group to add/register. This should be a type that derives␣
↪from Group.
   * @param[in] name The name of the group to use as a string key.
   * @return         A pointer to the newly registered Group.
   *
   * Creates and registers a Group or class derived from Group as a subgroup of this␣
↪Group.
   */
  template< typename T = Group >
  T & registerGroup( string const & name )
  { return registerGroup< T >( name, std::make_unique< T >( name, this ) ); }

  /**
   * @brief @copybrief registerGroup(string const &,std::unique_ptr<T>)
   *
   * @tparam T The type of the Group to add/register. This should be a type that derives␣
↪from Group.
   * @param keyIndex A KeyIndexT object that will be used to specify the name of
   *   the new group. The index of the KeyIndex will also be set.
   * @return A pointer to the newly registered Group, or @c nullptr if no group was␣
↪registered.
   *
   * Creates and registers a Group or class derived from Group as a subgroup of this␣
↪Group.
   */
  template< typename T = Group >
  T & registerGroup( subGroupMap::KeyIndex const & keyIndex )
  {
    T & rval = registerGroup< T >( keyIndex.key(), std::make_unique< T >( keyIndex.key(),
↪ this ) );
    keyIndex.setIndex( m_subGroups.getIndex( keyIndex.key() ) );
    return rval;
  }

  /**
   * @brief @copybrief registerGroup(string const &,std::unique_ptr<T>)
   *
   * @tparam T The type of the Group to add/register. This should be a type that derives␣
↪from Group.
   * @tparam TBASE The type whose type catalog will be used to look up the new sub-group␣
↪type
   * @param[in] name        The name of the group to use as a string key.
   * @param[in] catalogName The catalog name of the new type.
   * @return                A pointer to the newly registered Group.
   *
   * Creates and registers a Group or class derived from Group as a subgroup of this␣
↪Group.
   */
  template< typename T = Group, typename TBASE = Group >
  T & registerGroup( string const & name, string const & catalogName )
```

(continues on next page)

(continued from previous page)

```
{
  std::unique_ptr< TBASE > newGroup = TBASE::CatalogInterface::Factory( catalogName,␣
↪name, this );
  return registerGroup< T >( name, std::move( newGroup ) );
}

/**
 * @brief Removes a child group from this group.
 * @param name the name of the child group to remove from this group.
 */
void deregisterGroup( string const & name );

/**
 * @brief Creates a new sub-Group using the ObjectCatalog functionality.
 * @param[in] childKey The name of the new object type's key in the
 *                     ObjectCatalog.
 * @param[in] childName The name of the new object in the collection of
 *                      sub-Groups.
 * @return A pointer to the new Group created by this function.
 */
virtual Group * createChild( string const & childKey, string const & childName );

///@}
```

These functions all take in a `name` for the new `Group`, which will be used as the key when trying to access the `Group` in the future. Some variants create a new `Group`, while some variants take in an existing `Group` . The template argument is to specify the actaul type of the `Group` as it it is most likely a type that derives from `Group` that is we would like to create in the repository. Please see the doxygen documentation for a detailed description of each option.

Getting Groups

The collection of functions to retrieve a `Group` and their descriptions are taken from source and shown here:

```
/**
 * @name Sub-group retrieval methods.
 *
 * This collection of functions are used to get a sub-Group from the current group.␣
↪Various methods
 * for performing the lookup are provided (localIndex, string, KeyIndex), and each␣
↪have their
 * advantages and costs. The lowest cost lookup is the "localIndex" lookup. The␣
↪KeyIndex lookup
 * will add a cost for checking to make sure the index stored in KeyIndex is valid (a␣
↪string
 * compare, and a hash if it is incorrect). The string lookup is the full cost hash␣
↪lookup every
 * time that it is called.
 *
 * The template parameter specifies the "type" that the caller expects to lookup, and␣
↪thus attempts
 * to cast the pointer that is stored in m_subGroups to a pointer of the desired type.␣
```

(continues on next page)

(continued from previous page)

```
↪If this
   * cast fails, then a @p nullptr is returned. If no template parameter is specified␣
↪then a default
   * type of Group is assumed.
   */
  ///@{

  /**
   * @brief Return a pointer to a sub-group of the current Group.
   * @tparam T The type of subgroup.
   * @tparam KEY The type of the lookup.
   * @param key The key used to perform the lookup.
   * @return A pointer to @p T that refers to the sub-group, if the Group does not exist␣
↪or it
   *   has an incompatible type a @c nullptr is returned.
   */
  template< typename T = Group, typename KEY = void >
  T * getGroupPointer( KEY const & key )
  { return dynamicCast< T * >( m_subGroups[ key ] ); }

  /**
   * @copydoc getGroupPointer(KEY const &)
   */
  template< typename T = Group, typename KEY = void >
  T const * getGroupPointer( KEY const & key ) const
  { return dynamicCast< T const * >( m_subGroups[ key ] ); }

  /**
   * @brief Return a reference to a sub-group of the current Group.
   * @tparam T The type of subgroup.
   * @tparam KEY The type of the lookup.
   * @param key The key used to perform the lookup.
   * @return A reference to @p T that refers to the sub-group.
   * @throw std::domain_error If the Group does not exist is thrown.
   */
  template< typename T = Group, typename KEY = void >
  T & getGroup( KEY const & key )
  {
    Group * const child = m_subGroups[ key ];
    GEOS_THROW_IF( child == nullptr,
                   "Group " << getDataContext() << " has no child named " << key <<␣
↪std::endl
                            << dumpSubGroupsNames(),
                   std::domain_error );

    return dynamicCast< T & >( *child );
  }

  /**
   * @copydoc getGroup( KEY const & )
   */
  template< typename T = Group, typename KEY = void >
```

(continues on next page)

(continued from previous page)

```
  T const & getGroup( KEY const & key ) const
  {
    Group const * const child = m_subGroups[ key ];
    GEOS_THROW_IF( child == nullptr,
                   "Group " << getDataContext() << " has no child named " << key <<␣
↪std::endl
                            << dumpSubGroupsNames(),
                   std::domain_error );

    return dynamicCast< T const & >( *child );
  }

  /**
   * @brief Retrieve a group from the hierarchy using a path.
   * @tparam T type of subgroup
   * @param[in] path a unix-style string (absolute, relative paths valid)
   *                 to lookup the Group to return. Absolute paths search
   *                 from the tree root, while relative - from current group.
   * @return A reference to @p T that refers to the sub-group.
   * @throw std::domain_error If the Group doesn't exist.
   */
  template< typename T = Group >
  T & getGroupByPath( string const & path )
  { return dynamicCast< T & >( const_cast< Group & >( getBaseGroupByPath( path ) ) ); }

  /**
   * @copydoc getGroupByPath(string const &)
   */
  template< typename T = Group >
  T const & getGroupByPath( string const & path ) const
  { return dynamicCast< T const & >( getBaseGroupByPath( path ) ); }
```

Register Wrappers

```
  /**
   * @name Wrapper registration interface
   */
  ///@{

  /**
   * @brief Create and register a Wrapper around a new object.
   * @tparam T The type of the object allocated.
   * @tparam TBASE The type of the object that the Wrapper holds.
   * @param[in] name the name of the wrapper to use as a string key
   * @param[out] rkey a pointer to a index type that will be filled with the new
   *   Wrapper index in this Group
   * @return A reference to the newly registered/created Wrapper
   */
  template< typename T, typename TBASE=T >
  Wrapper< TBASE > & registerWrapper( string const & name,
```

(continues on next page)

(continued from previous page)

```
                                          wrapperMap::KeyIndex::index_type * const rkey =␣
↪nullptr );

  /**
   * @copybrief registerWrapper(string const &,wrapperMap::KeyIndex::index_type * const)
   * @tparam T the type of the wrapped object
   * @tparam TBASE the base type to cast the returned wrapper to
   * @param[in] viewKey The KeyIndex that contains the name of the new Wrapper.
   * @return A reference to the newly registered/created Wrapper
   */
  template< typename T, typename TBASE=T >
  Wrapper< TBASE > & registerWrapper( Group::wrapperMap::KeyIndex const & viewKey );

  /**
   * @brief Register a Wrapper around a given object and take ownership.
   * @tparam T the type of the wrapped object
   * @param[in] name the name of the wrapper to use as a string key
   * @param[in] newObject an owning pointer to the object that is being registered
   * @return A reference to the newly registered/created Wrapper
   * @note Not intended to register a @p WrapperBase instance. Use dedicated member␣
↪function instead.
   */
  template< typename T >
  Wrapper< T > & registerWrapper( string const & name, std::unique_ptr< T > newObject );

  /**
   * @brief Register a Wrapper around an existing object, does not take ownership of the␣
↪object.
   * @tparam T the type of the wrapped object
   * @param[in] name the name of the wrapper to use as a string key
   * @param[in] newObject a pointer to the object that is being registered
   * @return A reference to the newly registered/created Wrapper
   * @note Not intended to register a @p WrapperBase instance. Use dedicated member␣
↪function instead.
   */
  template< typename T >
  Wrapper< T > & registerWrapper( string const & name,
                                  T * newObject );

  /**
   * @brief Register and take ownership of an existing Wrapper.
   * @param wrapper A pointer to the an existing wrapper.
   * @return An un-typed pointer to the newly registered/created wrapper
   */
  WrapperBase & registerWrapper( std::unique_ptr< WrapperBase > wrapper );

  /**
   * @brief Removes a Wrapper from this group.
   * @param name the name of the Wrapper to remove from this group.
   */
  void deregisterWrapper( string const & name );
```

(continues on next page)

(continued from previous page)

```
///@}
```

Getting Wrappers/Wrapped Objects

```
/**
 * @name Untyped wrapper retrieval methods
 *
 * These functions query the collection of Wrapper objects for the given
 * index/name/KeyIndex and returns a WrapperBase pointer to the object if
 * it exists. If it is not found, nullptr is returned.
 */
///@{

/**
 * @brief Return a reference to a WrapperBase stored in this group.
 * @tparam KEY The lookup type.
 * @param key The value used to lookup the wrapper.
 * @return A reference to the WrapperBase that resulted from the lookup.
 * @throw std::domain_error if the wrapper doesn't exist.
 */
template< typename KEY >
WrapperBase const & getWrapperBase( KEY const & key ) const
{
  WrapperBase const * const wrapper = m_wrappers[ key ];
  GEOS_THROW_IF( wrapper == nullptr,
                 "Group " << getDataContext() << " has no wrapper named " << key <<␣
↪std::endl
                           << dumpWrappersNames(),
                 std::domain_error );

  return *wrapper;
}

/**
 * @copydoc getWrapperBase(KEY const &) const
 */
template< typename KEY >
WrapperBase & getWrapperBase( KEY const & key )
{
  WrapperBase * const wrapper = m_wrappers[ key ];
  GEOS_THROW_IF( wrapper == nullptr,
                 "Group " << getDataContext() << " has no wrapper named " << key <<␣
↪std::endl
                           << dumpWrappersNames(),
                 std::domain_error );

  return *wrapper;
}

/**
```

(continues on next page)

(continued from previous page)

```
 * @brief
 * @param name
 * @return
 */
indexType getWrapperIndex( string const & name ) const
{ return m_wrappers.getIndex( name ); }

/**
 * @brief Get access to the internal wrapper storage.
 * @return a reference to wrapper map
 */
wrapperMap const & wrappers() const
{ return m_wrappers; }

/**
 * @copydoc wrappers() const
 */
wrapperMap & wrappers()
{ return m_wrappers; }

/**
 * @brief Return the number of wrappers.
 * @return The number of wrappers.
 */
indexType numWrappers() const
{ return m_wrappers.size(); }

/**
 * @return An array containing all wrappers keys
 */
std::vector< string > getWrappersNames() const;

///@}

/**
 * @name Typed wrapper retrieval methods
 *
 * These functions query the collection of Wrapper objects for the given
 * index/key and returns a Wrapper<T> pointer to the object if
 * it exists. The template parameter @p T is used to perform a cast
 * on the WrapperBase pointer that is returned by the lookup, into
 * a Wrapper<T> pointer. If the wrapper is not found, or the
 * WrapperBase pointer cannot be cast to a Wrapper<T> pointer, then nullptr
 * is returned.
 */
///@{

/**
 * @brief Check if a wrapper exists
 * @tparam LOOKUP_TYPE the type of key used to perform the lookup.
 * @param[in] lookup a lookup value used to search the collection of wrappers
 * @return @p true if wrapper exists (regardless of type), @p false otherwise
```

(continues on next page)

(continued from previous page)

```
 */
template< typename LOOKUP_TYPE >
bool hasWrapper( LOOKUP_TYPE const & lookup ) const
{ return m_wrappers[ lookup ] != nullptr; }

/**
 * @brief Retrieve a Wrapper stored in this group.
 * @tparam T the object type contained in the Wrapper
 * @tparam LOOKUP_TYPE the type of key used to perform the lookup
 * @param[in] index    a lookup value used to search the collection of wrappers
 * @return A reference to the Wrapper<T> that resulted from the lookup.
 * @throw std::domain_error if the Wrapper doesn't exist.
 */
template< typename T, typename LOOKUP_TYPE >
Wrapper< T > const & getWrapper( LOOKUP_TYPE const & index ) const
{
  WrapperBase const & wrapper = getWrapperBase( index );
  return dynamicCast< Wrapper< T > const & >( wrapper );
}

/**
 * @copydoc getWrapper(LOOKUP_TYPE const &) const
 */
template< typename T, typename LOOKUP_TYPE >
Wrapper< T > & getWrapper( LOOKUP_TYPE const & index )
{
  WrapperBase & wrapper = getWrapperBase( index );
  return dynamicCast< Wrapper< T > & >( wrapper );
}

/**
 * @brief Retrieve a Wrapper stored in this group.
 * @tparam T the object type contained in the Wrapper
 * @tparam LOOKUP_TYPE the type of key used to perform the lookup
 * @param[in] index a lookup value used to search the collection of wrappers
 * @return A pointer to the Wrapper<T> that resulted from the lookup, if the Wrapper
 *   doesn't exist or has a different type a @c nullptr is returned.
 */
template< typename T, typename LOOKUP_TYPE >
Wrapper< T > const * getWrapperPointer( LOOKUP_TYPE const & index ) const
{ return dynamicCast< Wrapper< T > const * >( m_wrappers[ index ] ); }

/**
 * @copydoc getWrapperPointer(LOOKUP_TYPE const &) const
 */
template< typename T, typename LOOKUP_TYPE >
Wrapper< T > * getWrapperPointer( LOOKUP_TYPE const & index )
{ return dynamicCast< Wrapper< T > * >( m_wrappers[ index ] ); }

///@}

/**
```

(continues on next page)

(continued from previous page)

```
   * @name Wrapper data access methods.
   *
   * These functions can be used to get referece/pointer access to the data
   * stored by wrappers in this group. They are essentially just shortcuts for
   * @p Group::getWrapper() and @p Wrapper<T>::getReference().
   * An additional template parameter can be provided to cast the return pointer
   * or reference to a base class pointer or reference (e.g. Array to ArrayView).
   */
  ///@{

  /**
   * @brief Look up a wrapper and get reference to wrapped object.
   * @tparam T return value type
   * @tparam WRAPPEDTYPE wrapped value type (by default, same as return)
   * @tparam LOOKUP_TYPE type of value used for wrapper lookup
   * @param lookup       value for wrapper lookup
   * @return reference to @p T
   * @throw A std::domain_error if the Wrapper does not exist.
   */
  template< typename T, typename LOOKUP_TYPE >
  GEOS_DECLTYPE_AUTO_RETURN
  getReference( LOOKUP_TYPE const & lookup ) const
  { return getWrapper< T >( lookup ).reference(); }

  /**
   * @copydoc getReference(LOOKUP_TYPE const &) const
   */
  template< typename T, typename LOOKUP_TYPE >
  T & getReference( LOOKUP_TYPE const & lookup )
  { return getWrapper< T >( lookup ).reference(); }
```

Looping Interface

```
  /**
   * @name Functor-based subgroup iteration
   *
   * These functions loop over sub-groups and executes a functor that uses the sub-group␣
↪as an
   * argument. The functor is only executed if the group can be cast to a certain type␣
↪specified
   * by the @p ROUPTYPE/S pack. The variadic list consisting of @p GROUPTYPE/S will be␣
↪used recursively
   * to check if the group is able to be cast to the one of these types. The first type␣
↪in the
   * @p GROUPTYPE/S list will be used to execute the functor, and the next sub-group␣
↪will be processed.
   */
  ///@{

  /**
```

(continues on next page)

(continued from previous page)

```
 * @brief Apply the given functor to subgroups that can be casted to one of specified␣
↪types.
 * @tparam GROUPTYPE  the first type that will be used in the attempted casting of␣
↪group.
 * @tparam GROUPTYPES a variadic list of types that will be used in the attempted␣
↪casting of group.
 * @tparam LAMBDA     the type of functor to call
 * @param[in] lambda  the functor to call on subgroups
 */
template< typename GROUPTYPE = Group, typename ... GROUPTYPES, typename LAMBDA >
void forSubGroups( LAMBDA && lambda )
{
  for( auto & subGroupIter : m_subGroups )
  {
    applyLambdaToContainer< GROUPTYPE, GROUPTYPES... >( *subGroupIter.second, [&](␣
↪auto & castedSubGroup )
    {
      lambda( castedSubGroup );
    } );
  }
}

/**
 * @copydoc forSubGroups(LAMBDA &&)
 */
template< typename GROUPTYPE = Group, typename ... GROUPTYPES, typename LAMBDA >
void forSubGroups( LAMBDA && lambda ) const
{
  for( auto const & subGroupIter : m_subGroups )
  {
    applyLambdaToContainer< GROUPTYPE, GROUPTYPES... >( *subGroupIter.second, [&](␣
↪auto const & castedSubGroup )
    {
      lambda( castedSubGroup );
    } );
  }
}


/**
 * @brief Apply the given functor to subgroups that can be casted to one of specified␣
↪types.
 * @tparam GROUPTYPE  the first type that will be used in the attempted casting of␣
↪group.
 * @tparam GROUPTYPES a variadic list of types that will be used in the attempted␣
↪casting of group.
 * @tparam LAMBDA     the type of functor to call
 * @param[in] lambda  the functor to call on subgroups
 */
template< typename GROUPTYPE = Group, typename ... GROUPTYPES, typename LAMBDA >
void forSubGroupsIndex( LAMBDA && lambda )
{
```

(continues on next page)

(continued from previous page)

```
    localIndex counter = 0;
    for( auto & subGroupIter : m_subGroups )
    {
      applyLambdaToContainer< GROUPTYPE, GROUPTYPES... >( *subGroupIter.second,
                                                          [&]( auto & castedSubGroup )
      {
        lambda( counter, castedSubGroup );
      } );
      ++counter;
    }
  }

  /**
   * @copydoc forSubGroupsIndex(LAMBDA &&)
   */
  template< typename GROUPTYPE = Group, typename ... GROUPTYPES, typename LAMBDA >
  void forSubGroupsIndex( LAMBDA && lambda ) const
  {
    localIndex counter = 0;
    for( auto const & subGroupIter : m_subGroups )
    {
      applyLambdaToContainer< GROUPTYPE, GROUPTYPES... >( *subGroupIter.second,
                                                          [&]( auto const &␣
↪castedSubGroup )
      {
        lambda( counter, castedSubGroup );
      } );
      ++counter;
    }
  }

  /**
   * @copybrief forSubGroups(LAMBDA &&)
   * @tparam GROUPTYPE        the first type that will be used in the attempted casting␣
↪of group.
   * @tparam GROUPTYPES       a variadic list of types that will be used in the␣
↪attempted casting of group.
   * @tparam LOOKUP_CONTAINER type of container of subgroup lookup keys (names or␣
↪indices), must support range-based for
   * loop
   * @tparam LAMBDA           type of functor callable with an index in lookup container␣
↪and a reference to casted
   * subgroup
   * @param[in] subGroupKeys  container with subgroup lookup keys (e.g. names or␣
↪indices) to apply the functor to
   * @param[in] lambda        the functor to call
   */
  template< typename GROUPTYPE = Group, typename ... GROUPTYPES, typename LOOKUP_
↪CONTAINER, typename LAMBDA >
  void forSubGroups( LOOKUP_CONTAINER const & subGroupKeys, LAMBDA && lambda )
  {
    localIndex counter = 0;
```

(continues on next page)

(continued from previous page)

```
    for( auto const & subgroup : subGroupKeys )
    {
      applyLambdaToContainer< GROUPTYPE, GROUPTYPES... >( getGroup( subgroup ), [&](␣
↪auto & castedSubGroup )
      {
        lambda( counter, castedSubGroup );
      } );
      ++counter;
    }
  }

  /**
   * @copybrief forSubGroups(LAMBDA &&)
   * @tparam GROUPTYPE        the first type that will be used in the attempted casting␣
↪of group.
   * @tparam GROUPTYPES       a variadic list of types that will be used in the␣
↪attempted casting of group.
   * @tparam LOOKUP_CONTAINER type of container of subgroup lookup keys (names or␣
↪indices), must support range-based for
   * loop
   * @tparam LAMBDA           type of functor callable with an index in lookup container␣
↪and a reference to casted
   * subgroup
   * @param[in] subGroupKeys  container with subgroup lookup keys (e.g. names or␣
↪indices) to apply the functor to
   * @param[in] lambda        the functor to call
   */
  template< typename GROUPTYPE = Group, typename ... GROUPTYPES, typename LOOKUP_
↪CONTAINER, typename LAMBDA >
  void forSubGroups( LOOKUP_CONTAINER const & subGroupKeys, LAMBDA && lambda ) const
  {
    localIndex counter = 0;
    for( auto const & subgroup : subGroupKeys )
    {
      applyLambdaToContainer< GROUPTYPE, GROUPTYPES... >( getGroup( subgroup ), [&](␣
↪auto const & castedSubGroup )
      {
        lambda( counter, castedSubGroup );
      } );
      ++counter;
    }
  }
  ///@}

  /**
   * @name Functor-based wrapper iteration
   *
   * These functions loop over the wrappers contained in this group, and executes a␣
↪functor that
   * uses the Wrapper as an argument. The functor is only executed if the Wrapper can be␣
↪casted to
   * a certain type specified by the @p TYPE/S pack. The variadic list consisting of
```

(continues on next page)

(continued from previous page)

```
   * @p TYPE/S will be used recursively to check if the Wrapper is able to be casted to␣
↪the
   * one of these types. The first type in the @p WRAPPERTYPE/S list will be used to␣
↪execute the
   * functor, and the next Wrapper will be processed.
   */
  ///@{

  /**
   * @brief Apply the given functor to wrappers.
   * @tparam LAMBDA the type of functor to call
   * @param[in] lambda  the functor to call
   */
  template< typename LAMBDA >
  void forWrappers( LAMBDA && lambda )
  {
    for( auto & wrapperIter : m_wrappers )
    {
      lambda( *wrapperIter.second );
    }
  }

  /**
   * @copydoc forWrappers(LAMBDA &&)
   */
  template< typename LAMBDA >
  void forWrappers( LAMBDA && lambda ) const
  {
    for( auto const & wrapperIter : m_wrappers )
    {
      lambda( *wrapperIter.second );
    }
  }

  /**
   * @brief Apply the given functor to wrappers that can be cast to one of specified␣
↪types.
   * @tparam TYPE   the first type that will be used in the attempted casting of Wrapper
   * @tparam TYPES  a variadic list of types that will be used in the attempted casting␣
↪of Wrapper
   * @tparam LAMBDA the type of functor to call
   * @param[in] lambda  the functor to call
   */
  template< typename TYPE, typename ... TYPES, typename LAMBDA >
  void forWrappers( LAMBDA && lambda )
  {
    for( auto & wrapperIter : m_wrappers )
    {
      applyLambdaToContainer< Wrapper< TYPE >, Wrapper< TYPES >... >( *wrapperIter.
↪second,
                                                                      std::forward<␣
↪LAMBDA >( lambda ));
```

(continues on next page)

(continued from previous page)

```
  }
 }

 /**
  * @brief Apply the given functor to wrappers that can be cast to one of specified␣
↪types.
  * @tparam TYPE   the first type that will be used in the attempted casting of Wrapper
  * @tparam TYPES  a variadic list of types that will be used in the attempted casting␣
↪of Wrapper
  * @tparam LAMBDA the type of functor to call
  * @param[in] lambda  the functor to call
  */
 template< typename TYPE, typename ... TYPES, typename LAMBDA >
 void forWrappers( LAMBDA && lambda ) const
 {
  for( auto const & wrapperIter : m_wrappers )
  {
   applyLambdaToContainer< Wrapper< TYPE >, Wrapper< TYPES >... >( *wrapperIter.
↪second,
                                                                  std::forward<␣
↪LAMBDA >( lambda ));
  }
 }

 ///@}
```

Wrapper

This class encapsulates an object for storage in a Group and provides an interface for performing some common operations on that object.

Description

In the filesystem analogy, a Wrapper may be thought of as a file that stores actual data. Each Wrapper belong to a single Group much like a file belongs to a filesystem directory. In general, more than one wrapper in the tree may refer to the same wrapped object, just like symlinks in the file system may refer to the same file. However, only one wrapper should be *owning* the data (see below).

In the XML input file, Wrapper correspond to attribute of an XML element representing the containing Group. See *XML Input* for the relationship between XML input files and Data Repository.

Wrapper<T> is templated on the type of object it encapsulates, thus providing strong type safety when retrieving the objects. As each Wrapper class instantiation will be a distinct type, Wrapper derives from a non-templated WrapperBase class that defines a common interface. WrapperBase is the type of pointer that is stored in the MappedVector container within a Group.

WrapperBase provides several interface functions that delegate the work to the wrapped object if it supports the corresponding method signature. This allows a collection of heterogeneous wrappers (i.e. over different types) to be treated uniformly. Examples include:

- size()
- resize(newSize)

- `reserve(newCapacity)`
- `capacity()`
- `move(LvArray::MemorySpace)`

A `Wrapper` may be *owning* or *non-owning*, depending on how it's constructed. An *owning* `Wrapper` will typically either take a previously allocated object via `std::unique_ptr<T>` or no pointer at all and itself allocate the object. It will delete the wrapped object when destroyed. A *non-owning* `Wrapper` may be used to register with the data repository objects that are not directly heap-allocated, for example data members of other objects. It will take a raw pointer as input and not delete the wrapped object when destroyed.

Attributes

Each instance of `Wrapper` has a set of attributes that control its function in the data repository. These attributes are:

- **InputFlags**

  A strongly typed enum that defines the relationship between the Wrapper and the XML input. Possible values are:

Value	Explanation
`FALSE`	Data is not read from XML input (default).
`OPTIONAL`	Data is read from XML if an attribute matching Wrapper's name is found.
`REQUIRED`	Data is read from XML and an error is raised if the attribute is not found.

  Other values of `InputFlags` enumeration are reserved for `Group` objects.

Note: A runtime error will occur when attempting to read from XML a wrapped type `T` that does not have `operator>>` defined.

- **RestartFlags**

  Enumeration that describes how the Wrapper interacts with restart files.

Value	Explanation
`NO_WRITE`	Data is not written into restart files.
`WRITE`	Data is written into restart files but not read upon restart.
`WRITE_AND_READ`	Data is both written and read upon restart (default).

Note: A runtime error will occur when attempting to write a wrapped type `T` that does not support buffer packing. Therefore, when registering custom types (i.e. not a basic C++ type or an *LvArray* container) we recommend setting the flag to `NO_WRITE`. A future documentation topic will explain how to extend buffer packing capabilities to custom user-defined types.

- **PlotLevel**

  Enumeration that describes how the Wrapper interacts with plot (visualization) files.

Value	Explanation
`LEVEL_0`	Data always written to plot files.
`LEVEL_1`	Data written to plot when `plotLevel>=1` is specified in input.
`LEVEL_2`	Data written to plot when `plotLevel>=2` is specified in input.
`LEVEL_3`	Data written to plot when `plotLevel>=3` is specified in input.
`NOPLOT`	Data never written to plot files.

Note: Only data stored in *LvArray*'s `Array<T>` containers is currently written into plot files.

Default Values

`Wrapper` supports setting a default value for its wrapped object. The default value is used if a wrapper with `InputFlags::OPTIONAL` attribute does not match an attribute in the input file. For *LvArray* containers it is also used as a default value for new elements upon resizing the container.

Default value can be set via one of the following two methods:

- `setDefaultValue` sets the default value but does not affect the actual value stored in the wrapper.
- `setApplyDefaultValue` sets the default value *and* applies it to the stored value.

Note: A runtime error is raised if a default value is not set for a wrapper with `InputFlags::OPTIONAL` attribute.

The type `DefaultValue<T>` is used to store the default value for the wrapper.

Todo: `DefaultValue` is actually not a type but an alias for another internal struct. As such, it cannot currently be specialized for a user's custom type.

API documentation

Wrapper

ObjectCatalog

The "ObjectCatalog" is a collection of classes that acts as a statically initialized factory. It functions in a similar manner to a classic factory method, except that there is no maintained list of derived objects that is required to create new objects. In other words, there is no case-switch/if-elseif block to maintain. Instead, the `ObjectCatalog` creates a "catalog" of derived objects using a `std::unordered_map`. The "catalog" is filled when new types are declared through the declaration of a helper class named `CatalogEntryConstructor`.

The key functional features of the "ObjectCatalog" concept may be summarized as:

- Anonymous addition of new objects to the catalog. Because we use a statically initialized singleton map object to store the catalog, no knowledge of the contents of the catalog is required in the main code. Therefore if a proprietary/sensitive catalog entry is desired, it is only required that the object definition be outside of the main repository and tied into the build system through some non-specific mechanism (i.e. a link in the src/externalComponents directory) and the catalog entry will be registered in the catalog without sharing any knowledge of its existence. Then a proprietary input file may refer to the object to call for its creation.

- Zero maintenance catalog. Again, because we use a singleton map to store the catalog, there is no updating of code required to add new entries into the catalog. The only modifications required are the actual source files of the catalog entry, as described in the *Usage* section below.

Implementation Details

There are three key objects that are used to provide the ObjectCatalog functionality.

CatalogInterface

The `CatalogInterface` class provides the base definitions and interface for the ObjectCatalog concept. It is templated on the common base class of all derived objects that are creatable by the "ObjectCatalog". In addition, `CatalogInterface` is templated on a variadic parameter pack that allows for an arbitrary constructor argument list as shown in the declaration shown below:

```
template< typename BASETYPE, typename ... ARGS >
class CatalogInterface
```

The `CatalogInterface` also defines the actual catalog type using the template arguments:

```
  typedef std::unordered_map< std::string,
                              std::unique_ptr< CatalogInterface< BASETYPE, ARGS... > > >␣
↪CatalogType;
```

The `CatalogInterface::CatalogType` is a `std::unordered_map` with a string "key" and a value type that is a pointer to the CatalogInterface that represents a specific combination of `BASETYPE` and constructor arguments.

After from setting up and populating the catalog, which will be described in the "Usage" section, the only interface with the catalog will typically be when the `Factory()` method is called. The definition of the method is given as:

```
  static std::unique_ptr< BASETYPE > factory( std::string const & objectTypeName,
                                              ARGS... args )
  {
    // We stop the simulation if the product is not found
    if( !hasKeyName( objectTypeName ) )
    {
      std::list< typename CatalogType::key_type > keys = getKeys();
      string const tmp = stringutilities::join( keys.cbegin(), keys.cend(), ",\n" );

      string errorMsg = "Could not find keyword \"" + objectTypeName + "\" in this␣
↪context. ";
      errorMsg += "Please be sure that all your keywords are properly spelled or that␣
↪input file parameters have not changed.\n";
      errorMsg += "All available keys are: [\n" + tmp + "\n]";
      GEOS_ERROR( errorMsg );
    }

    // We also stop the simulation if the builder is not here.
    CatalogInterface< BASETYPE, ARGS... > const * builder = getCatalog().at(␣
↪objectTypeName ).get();
```

(continues on next page)

(continued from previous page)

```
    if( builder == nullptr )
    {
      const string errorMsg = "\"" + objectTypeName + "\" could be found. But the␣
↪builder is invalid.\n";
      GEOS_ERROR( errorMsg );
    }

    return builder->allocate( args ... );
  }
```

It can be seen that the static `Factory` method is simply a wrapper that calls the virtual `Allocate` method on a the catalog which is returned by `getCatalog()`. The usage of the `Factory` method will be further discussed in the *Usage* section.

Note: The method for organizing constructing new objects relies on a common constructor list between the derived type and the `BASETYPE`. This means that there is a single catalog for each combination of `BASETYPE` and the variadic parameter pack representing the constructor arguments. In the future, we can investigate removing this restriction and allowing for construction of a hierarchy of objects with an arbitrary constructor parameter list.

CatalogEntry

The `CatalogEntry` class derives from `CatalogInterface` and adds the a `TYPE` template argument to the arguments of the `CatalogInterface`.

```
template< typename BASETYPE, typename TYPE, typename ... ARGS >
class CatalogEntry final : public CatalogInterface< BASETYPE, ARGS... >
```

The `TYPE` template argument is the type of the object that you would like to be able to create with the "ObjectCatalog". `TYPE` must be derived from `BASETYPE` and have a constructor that matches the variadic parameter pack specified in the template parameter list. The main purpose of the `CatalogEntry` is to override the `CatalogInterface::Allocate()` virtual function s.t. when key is retrieved from the catalog, then it is possible to create a new `TYPE`. The `CatalogEntry::Allocate()` function is a simple creation of the underlying `TYPE` as shown by its definition:

```
  virtual std::unique_ptr< BASETYPE > allocate( ARGS... args ) const override
  {
#if OBJECTCATALOGVERBOSE > 0
    GEOS_LOG( "Creating type " << LvArray::system::demangle( typeid(TYPE).name())
                               << " from catalog of " << LvArray::system::demangle(␣
↪typeid(BASETYPE).name()));
#endif
#if ( __cplusplus >= 201402L )
    return std::make_unique< TYPE >( args ... );
#else
    return std::unique_ptr< BASETYPE >( new TYPE( args ... ) );
#endif
  }
```

CatalogEntryConstructor

The `CatalogEntryConstructor` is a helper class that has a sole purpose of creating a new `CatalogEntry` and adding it to the catalog. When a new `CatalogEntryConstructor` is created, a new `CatalogEntry` entry is created and inserted into the catalog automatically.

Usage

Creating A New Catalog

When creating a new “ObjectCatalog”, it typically is done within the context of a specific `BASETYPE`. A simple example of a class hierarchy in which we would like to use the “ObjectCatalog” to use to generate new objects is given in the unit test located in `testObjectCatalog.cpp`.

The base class for this example is defined as:

```
class Base
{
public:
  Base( int & junk, double const & junk2 )
  {
    GEOS_LOG( "calling Base constructor with arguments ("<<junk<<" "<<junk2<<")" );
  }

  virtual ~Base()
  {
    GEOS_LOG( "calling Base destructor" );
  }

  using CatalogInterface = dataRepository::CatalogInterface< Base, int &, double const &␣
↪>;
  static CatalogInterface::CatalogType & getCatalog()
  {
    static CatalogInterface::CatalogType catalog;
    return catalog;
  }

  virtual string getCatalogName() = 0;
};
```

There a couple of things to note in the definition of `Base`:

- `Base` has a convenience alias to use in place of the fully templated `CatalogInterface` name.
- `Base` defines a `getCatalog()` function that returns a static instantiation of a `CatalogInterface::CatalogType`. The `CatalogInterface::getCatalog()` function actually calls this function within the base class. This means that the base class actually owns the catalog, and the `CatalogInterface` is only operating on that `Base::getCatalog()`, and that the definition of this function is required.

Adding A New Type To The Catalog

Once a Base class is defined with the required features, the next step is to add a new derived type to the catalog defined in Base. There are three requirements for the new type to be registered in the catalog:

- The derived type must have a constructor with the arguments specified by the variadic parameter pack specified in the catalog.
- There must be a static function `static string catalogName()` that returns the name of the type that will be used to as keyname when it is registered Base's catalog.
- The new type must be registered with the catalog held in Base. To accomplish this, a convenience macro `REGISTER_CATALOG_ENTRY()` is provided. The arguments to this macro are the name type of Base, the type of the derived class, and then the variadic pack of constructor arguments.

A pair of of simple derived class that have the required methods are used in the unit test.

```
class Derived1 : public Base
{
public:
  Derived1( int & junk, double const & junk2 ):
    Base( junk, junk2 )
  {
    GEOS_LOG( "calling Derived1 constructor with arguments ("<<junk<<" "<<junk2<<")" );
  }

  ~Derived1()
  {
    GEOS_LOG( "calling Derived1 destructor" );
  }
  static string catalogName() { return "derived1"; }
  string getCatalogName() { return catalogName(); }

};
REGISTER_CATALOG_ENTRY( Base, Derived1, int &, double const & )
```

```
class Derived2 : public Base
{
public:
  Derived2( int & junk, double const & junk2 ):
    Base( junk, junk2 )
  {
    GEOS_LOG( "calling Derived2 constructor with arguments ("<<junk<<" "<<junk2<<")" );
  }

  ~Derived2()
  {
    GEOS_LOG( "calling Derived2 destructor" );
  }
  static string catalogName() { return "derived2"; }
  string getCatalogName() { return catalogName(); }

};
REGISTER_CATALOG_ENTRY( Base, Derived2, int &, double const & )
```

Allocating A New Object From The Catalog

The test function in the unit test shows how to allocate a new object of one of the derived types from `Factory` method. Note the call to `Factory` is scoped by `Base::CatalogInterface`, which is an alias to the full templated instantiation of `CatalogInterface`. The arguments for `Factory`

```
TEST( testObjectCatalog, testRegistration )
{
  GEOS_LOG( "EXECUTING MAIN" );
  int junk = 1;
  double junk2 = 3.14;

  // allocate a new Derived1 object
  std::unique_ptr< Base >
  derived1 = Base::CatalogInterface::factory( "derived1", junk, junk2 );

  // allocate a new Derived2 object
  std::unique_ptr< Base >
  derived2 = Base::CatalogInterface::factory( "derived2", junk, junk2 );

  EXPECT_STREQ( derived1->getCatalogName().c_str(),
                Derived1::catalogName().c_str() );

  EXPECT_STREQ( derived2->getCatalogName().c_str(),
                Derived2::catalogName().c_str() );
  GEOS_LOG( "EXITING MAIN" );
}
```

The unit test creates two new objects of type `Derived1` and `Derived2` using the catalogs `Factory` method. Then the test checks to see that the objects that were created are of the correct type. This unit test has some extra output to screen to help with understanding of the sequence of events. The result of running this test is:

```
$ tests/testObjectCatalog
Calling constructor for CatalogEntryConstructor< Derived1 , Base , ... >
Calling constructor for CatalogInterface< Base , ... >
Calling constructor for CatalogEntry< Derived1 , Base , ... >
Registered Base catalog component of derived type Derived1 where Derived1::catalogName()␣
↪= derived1
Calling constructor for CatalogEntryConstructor< Derived2 , Base , ... >
Calling constructor for CatalogInterface< Base , ... >
Calling constructor for CatalogEntry< Derived2 , Base , ... >
Registered Base catalog component of derived type Derived2 where Derived2::catalogName()␣
↪= derived2
Running main() from gtest_main.cc
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from testObjectCatalog
[ RUN      ] testObjectCatalog.testRegistration
EXECUTING MAIN
Creating type Derived1 from catalog of Base
calling Base constructor with arguments (1 3.14)
calling Derived1 constructor with arguments (1 3.14)
Creating type Derived2 from catalog of Base
```

(continues on next page)

(continued from previous page)

```
calling Base constructor with arguments (1 3.14)
calling Derived2 constructor with arguments (1 3.14)
EXITING MAIN
calling Derived2 destructor
calling Base destructor
calling Derived1 destructor
calling Base destructor
[       OK ] testObjectCatalog.testRegistration (0 ms)
[----------] 1 test from testObjectCatalog (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (0 ms total)
[  PASSED  ] 1 test.
Calling destructor for CatalogEntryConstructor< Derived2 , Base , ... >
Calling destructor for CatalogEntryConstructor< Derived1 , Base , ... >
Calling destructor for CatalogEntry< Derived2 , Base , ... >
Calling destructor for CatalogInterface< Base , ... >
Calling destructor for CatalogEntry< Derived1 , Base , ... >
Calling destructor for CatalogInterface< Base , ... >
```

In the preceding output, it is clear that the static catalog in `Base::getCatalog()` is initialized prior the execution of main, and destroyed after the completion of main. In practice, there have been no indicators of problems due to the use of a statically initialized/deinitialized catalog.

XML Input

In this document, you will learn how GEOS classes interact with external information parsed from XML files, and how to add a new XML block that can be interpreted by GEOS. Flow solvers and relative permeability are used as examples.

GEOS data structure overview

Group : the base class of GEOS

All GEOS classes derive from a base class called `dataRepository::Group`. The `Group` class provides a way to organize all GEOS objects in a filesystem-like structure. One could think of `Group` s as *file folders* that can bear data (stored in `Wrapper` s), have a parent folder (another `Group`), and have possibly multiple subfolders (referred to as the subgroups). Below, we briefly review the data members of the `Group` class that are essential to understand the correspondence between the GEOS data structure and the XML input. For more details, we refer the reader to the extensive documentation of the *Data Repository*, including the *Group* class documentation.

In the code listing below, we see that each `Group` object is at minimum equipped with the following member properties:

- A pointer to the parent `Group` called `m_parent` (member classes are prefixed by `m_`),
- The `Group` ‘s own data, stored for flexibility in an array of generic data `Wrapper` s called `m_wrappers`,
- A map of one or many children (also of type `Group`) called `m_subGroups`.
- The `m_size` and `m_capacity` members, that are used to set the size and capacity of any objects contained.
- The name of the `Group`, stored as a `string` in `m_name`. This name can be seen as the object unique ID.

```
  /// The parent Group that contains "this" Group in its "sub-Group" collection.
  Group * m_parent = nullptr;

  /// Specification that this group will have the same m_size as m_parent.
  integer m_sizedFromParent;

  /// The container for the collection of all wrappers continued in "this" Group.
  wrapperMap m_wrappers;

  /// The container for the collection of all sub-groups contained in "this" Group.
  subGroupMap m_subGroups;

  /// The size/length of this Group...and all Wrapper<> that are are specified to have␣
↪the same size as their
  /// owning group.
  indexType m_size;

  /// The capacity for wrappers in this group...and all Wrapper<> that are specified to␣
↪have the same size as their
  /// owning group.
  indexType m_capacity;

  /// The name/key of this Group in its parent collection of sub-Groups.
  string m_name;

  /// Verbosity flag for group logs
  integer m_logLevel;
  //END_SPHINX_INCLUDE_02

  /// Restart flag for this group... and subsequently all wrappers in this group.
```

[Source: src/coreComponents/dataRepository/Group.hpp]

A few words about the ObjectCatalog

What is an ObjectCatalog and why do we need it?

Some classes need external information (physical and/or algorithmic parameters for instance) provided by the user to be instantiated. This is the case when the `m_input_flags` data member of one of the `Group` ‘s `Wrapper` s has an entry set to `REQUIRED` (we will illustrate this below). In this situation, the required information must be supplied in the XML input file, and if it is absent, an error is raised by GEOS.

To connect the external (XML) and internal (C++) data structures, GEOS uses an **ObjectCatalog** that maps keys (of type `string`) to the corresponding classes (one unique key per mapped class). These string keys, referred to as `catalogName` s, are essential to transfer the information from the XML file to the factory functions in charge of object instantiation (see below).

What is a CatalogName?

The `catalogName` of an object is a *key* (of type `string`) associated with this object’s class. On the one hand, in the XML file, the key is employed by the user as an XML tag to specify the type of object (e.g., the type of solver, constitutive model, etc) to create and use during the simulation. On the other hand, internally, the key provides a way to access the appropriate factory function to instantiate an object of the desired class.

Most of the time, the `catalogName` and the C++ class name are identical. This helps make the code easier to debug and allows the XML/C++ correspondence to be evident. But strictly speaking, the `catalogName` can be anything, as long as it refers uniquely to a specific class. The `catalogName` must not be confused with the object's *name* (`m_name` is a data member of the class that stores the object's unique ID, not its class key). You can have several objects of the same class and hence the same `catalogName`, but with different names (i.e. unique ID): several fluid models, several solvers, etc.

How can I add my new externally-accessible class to the ObjectCatalog?

Let us consider a flow solver class derived from `FlowSolverBase`, that itself is derived from `SolverBase`. To instantiate and use this solver, the developer needs to make the derived flow solver class reachable from the XML file, via an XML tag. Internally, this requires adding the derived class information to `ObjectCatalog`, which is achieved with two main ingredients: 1) a `CatalogName()` method in the class that lets GEOS know *what* to search for in the internal `ObjectCatalog` to instantiate an object of this class, 2) a macro that specifies *where* to search in the `ObjectCatalog`.

1. To let GEOS know what to search for in the catalog to instantiate an object of the derived class, the developer must equip the class with a `CatalogName()` method that returns a `string`. In this document, we have referred to this returned `string` as the object's `catalogName`, but in fact, the method `CatalogName()` is what matters since the `ObjectCatalog` contains all the `CatalogName()` return values. Below, we illustrate this with the `CompositionalMultiphaseFlow` solver. The first code listing defines the class name, which in this case is the same as the `catalogName` shown in the second listing.

```
/**
 * @class CompositionalMultiphaseBase
 *
 * A compositional multiphase solver
 */
class CompositionalMultiphaseBase : public FlowSolverBase
{
```

```
```

[Source: src/coreComponents/physicsSolvers/fluidFlow/CompositionalMultiphaseBase.hpp]

2. To let GEOS know where to search in the `ObjectCatalog`, a macro needs to be added at the end of the .cpp file implementing the class. This macro (illustrated below) must contain the type of the base class (in this case, `SolverBase`), and the name of the derived class (continuing with the example used above, this is `CompositionalMultiphaseFlow`). As a result of this construct, the `ObjectCatalog` is not a flat list of `string` s mapping the C++ classes. Instead, the `ObjectCatalog` forms a tree that reproduces locally the structure of the class diagram, from the base class to the derived classes.

```
REGISTER_CATALOG_ENTRY( SolverBase, CompositionalMultiphaseFVM, string const &, Group *␣
↪const )
```

[Source: src/coreComponents/physicsSolvers/fluidFlow/CompositionalMultiphaseFVM.cpp]

> Summary: All GEOS objects form a filesystem-like structure. If an object needs to be accessible externally, it must be registered in the `ObjectCatalog`. This is done by adding `CatalogName()` method that returns a `string` key to the object's class, and by adding the appropriate macro. The catalog has the same tree structure as the class diagram.

Registration: parsing XML input files to instantiate GEOS objects

In this section, we describe with more details the connection between **internal GEOS objects** and **external XML tags** parsed from parameter files. We call this process *Registration*. The registration process works in three steps:

1. The XML document is parsed. Each time a new XML tag is found, the current local scope of the `ObjectCatalog` is inspected. The goal is to find a `catalogName string` that matches the XML tag.
2. If it is the case (the current local scope of the `ObjectCatalog` contains a `catalogName` identical to the XML tag), then the code creates a new instance of the class that the `catalogName` refers to. This new object is inserted in the `Group` tree structure at the appropriate location, as a subgroup.
3. By parsing the XML attributes of the tag, the new object properties are populated. Some checks are performed to ensure that the data supplied is conform, and that all the required information is present.

Let's look at this process in more details.

Creating a new object and giving it a Catalog name

Consider again that we are registering a flow solver deriving from `FlowSolverBase`, and assume that this solver is called `CppNameOfMySolver`. This choice of name is not recommended (we want names that reflect what the solver does!), but for this particular example, we just need to know that this name is the class name inside the C++ code.

To specify parameters of this new solver from an XML file, we need to be sure that the XML tag and the `catalogName` of the class are identical. Therefore, we equip the `CppNameOfMySolver` class with a `CatalogName()` method that returns the solver `catalogName` (=XML name). Here, this method returns the `string` "XmlNameOfMySolver".

We have deliberately distinguished the class name from the catalog/XML name for the sake of clarity in this example. It is nevertheless a best practice to use the same name for the class and for the `catalogName`. This is the case below for the existing `CompositionalMultiphaseFVM` class.

```
class CompositionalMultiphaseFVM : public CompositionalMultiphaseBase
{
```

```
  /**
   * @brief name of the solver in the object catalog
   * @return string that contains the catalog name to generate a new object through the object catalog.
   */
  static string catalogName() { return "CompositionalMultiphaseFVM"; }
```

[Source: src/coreComponents/physicsSolvers/fluidFlow/CompositionalMultiphaseFVM.hpp]

Parsing XML and searching the ObjectCatalog in scope

Now that we have implemented a `CatalogName()` method returning a specific key (of type `string`), we can have a block in our XML input file with a tag that corresponds to the `catalogName` "XmlNameOfMySolver". This is how the XML block would look like.

```
<Problem>
  <Solvers
    gravityVector="{ 0.0, 0.0, -9.81 }">
    <XmlNameOfMySolver name="nameOfThisSolverInstance"
                       verboseLevel="1"
```

(continues on next page)

(continued from previous page)

```
                             gravityFlag="1"
                             temperature="297.15" />
      <LinearSolverParameters newtonTol="1.0e-6"
                              maxIterNewton="15"
                              useDirectSolver="1"/>
    </XmlNameOfMySolver>
  </Solvers>
</Problem>
```

Here, we see that the XML structure defines a parent node "Problem", that has (among many others) a child node "Solvers". In the "Solvers" block, we have placed the new solver block as a child node of the "Solvers" block with the XML tag corresponding to the catalogName of the new class. We will see in details next how the GEOS internal structure constructed from this block mirrors the XML file structure.

Instantiating the new solver

Above, we have specified an XML block with the tag "XmlNameOfMySolver". Now, when reading the XML file and encountering an "XmlNameOfMySolver" solver block, we add a new instance of the class CppNameOfMySolver in the filesystem structure as explained below.

We saw that in the XML file, the new solver block appeared as child node of the XML block "Solvers". The internal construction mirrors this XML structure. Specifically, the new object of class CppNameOfMySolver is registered as a subgroup (to continue the analogy used so far, as a subfolder) of its parent Group, the class PhysicsSolverManager (that has a catalogName "Solvers"). To do this, the method CreateChild of the PhysicsSolverManager class is used.

```
// Variable values in this example:
// --------------------------------
// childKey = "XmlNameOfMySolver" (string)
// childName = "nameOfThisSolverInstance" (string)
// SolverBase::CatalogInterface = the Catalog attached to the base Solver class
// hasKeyName = bool method to test if the childKey string is present in the Catalog
// registerGroup = method to create a new instance of the solver and add it to the group␣
↪tree
```

```
Group * PhysicsSolverManager::createChild( string const & childKey, string const &␣
↪childName )
{
  Group * rval = nullptr;
  if( SolverBase::CatalogInterface::hasKeyName( childKey ) )
  {
    GEOS_LOG_RANK_0( "Adding Solver of type " << childKey << ", named " << childName );
    rval = &registerGroup( childName,
                           SolverBase::CatalogInterface::factory( childKey, childName,␣
↪this ) );
  }
  return rval;
}
```

[Source: src/coreComponents/physicsSolvers/PhysicsSolverManager.cpp]

In the code listing above, we see that in the PhysicsSolverManager class, the ObjectCatalog is searched to find the catalogName "CompositionalMultiphaseFlow" in the scope of the SolverBase class. Then, the factory function of the base class SolverBase is called. The catalogName (stored in childKey) is passed as an argument of the factory function to ensure that it instantiates an object of the desired derived class.

As explained above, this is working because 1) the XML tag matches the catalogName of the CompositionalMultiphaseFlow class and 2) a macro is placed at the end of the .cpp file implementing the CompositionalMultiphaseFlow class to let the ObjectCatalog know that CompositionalMultiphaseFlow is a derived class of SolverBase.

Note that several instances of the same type of solver can be created, as long as they each have a different name.

Filling the objects with data (wrappers)

After finding and placing the new solver Group in the filesystem hierarchy, properties are read and stored. This is done by registering *data wrappers*. We refer to the documentation of the *Data Repository* for additional details about the Wrapper s. The method used to do that is called registerWrapper and is placed in the class constructor when the data is required in the XML file. Note that some properties are registered at the current (derived) class level, and other properties can also be registered at a base class level.

Here, the only data (=wrapper) that is defined at the level of our CppNameOfMySolver class is temperature, and everything else is registered at the base class level. We register a property of temperature, corresponding to the member class m_temperature of CppNameOfMySolver. The registration also checks if a property is required or optional (here, it is required), and provides a brief description that will be used in the auto-generated code documentation.

```
  this->registerWrapper( viewKeyStruct::inputTemperatureString(), &m_inputTemperature ).
    setInputFlag( InputFlags::REQUIRED ).
    setDescription( "Temperature" );
//END_SPHINX_INCLUDE_00
```

[Source: src/coreComponents/physicsSolvers/fluidFlow/CompositionalMultiphaseBase.cpp]

This operation is done recursively if XML tags are nested.

To summarize:

- Every class in GEOS derive from a Group in a filesystem-like structure. A Group must have a parent Group, can have data (in Wrapper s), and can have one or many children (the subgroups). There is an ObjectCatalog in which the classes derived from Group are identified by a key called the catalogName.
- When parsing XML input files, GEOS inspects each object's scope in the ObjectCatalog to find classes with the same catalogName as the XML tag. Once it finds an XML tag in the ObjectCatalog, it registers it inside the filesystem structure.
- In the registration process, properties from the XML file are parsed and used to allocate member data Wrapper s and fully instantiate the Group class.
- If XML tags are nested, subgroups are allocated and processed in a nested manner.

The correspondence between XML and class hierarchy is thus respected, and the internal object hierarchy mirrors the XML structure.

Example: adding a new relative permeability model

This example is taken from the class BrooksCoreyRelativePermeability, derived from RelativePermeabilityBase.

Implement a CatalogName function (.hpp):

As explained above we add the class to the ObjectCatalog in two steps. First we implement the CatalogName function:

```
  static string catalogName() { return "BrooksCoreyRelativePermeability"; }
```

[source: src/coreComponents/constitutive/relativePermeability/BrooksCoreyRelativePermeability.hpp]

Then in the .cpp file we add the macro to register the catalog entry:

```
REGISTER_CATALOG_ENTRY( ConstitutiveBase, BrooksCoreyRelativePermeability, string const &
↪, Group * const )
```

[source: src/coreComponents/constitutive/relativePermeability/BrooksCoreyRelativePermeability.cpp]

Now every time a "BrooksCoreyRelativePermeability" string is encountered inside a Relative Permeability catalog, we will instantiate a class BrooksCoreyRelativePermeability.

Declare the Wrapper s keys (.hpp):

When attaching properties (i.e. data Wrapper s) to a class, a similar registration process must be done. Every property is accessed through its ViewKey namespace. In this namespace, we define string s that correspond to the tags of XML attributes of the "BrooksCoreyRelativePermeability" block.

```
  struct viewKeyStruct : RelativePermeabilityBase::viewKeyStruct
  {
    static constexpr char const * phaseMinVolumeFractionString() { return
↪"phaseMinVolumeFraction"; }
    static constexpr char const * phaseRelPermExponentString() { return
↪"phaseRelPermExponent"; }
    static constexpr char const * phaseRelPermMaxValueString() { return
↪"phaseRelPermMaxValue"; }
    static constexpr char const * volFracScaleString() { return "volFracScale"; }
  } vieKeysBrooksCoreyRelativePermeability;
```

[source: src/coreComponents/constitutive/relativePermeability/BrooksCoreyRelativePermeability.hpp]

Declare data members (.hpp):

The data members are defined in the class. They will ultimately contain the data read from the XML file (other data members not read from the XML file can also exist).

```
  array1d< real64 > m_phaseMinVolumeFraction;
  array1d< real64 > m_phaseRelPermExponent;
  array1d< real64 > m_phaseRelPermMaxValue;

  real64 m_volFracScale;
```

[source: src/coreComponents/constitutive/relativePermeability/BrooksCoreyRelativePermeability.hpp]

Implement the data registration process (`registerWrapper`):

The registration process done in the class constructor puts everything together. It connects the attributes values in the XML file to class member data. For instance, in the listing below, the first `registerWrapper` call means that we want to read in the XML file the attribute value corresponding to the attribute tag ‘’phaseMinVolumeFraction” defined in the .hpp file, and that we want to store the read values into the `m_phaseMinVolumeFraction` data members. We see that this input is not required. If it is absent from the XML file, the default value is used instead. The short description that completes the registration will be added to the auto-generated documentation.

```
BrooksCoreyRelativePermeability::BrooksCoreyRelativePermeability( string const & name,
                                                                  Group * const parent )
  : RelativePermeabilityBase( name, parent )
{
  registerWrapper( viewKeyStruct::phaseMinVolumeFractionString(), &m_
↪phaseMinVolumeFraction ).
    setApplyDefaultValue( 0.0 ).
    setInputFlag( InputFlags::OPTIONAL ).
    setDescription( "Minimum volume fraction value for each phase" );

  registerWrapper( viewKeyStruct::phaseRelPermExponentString(), &m_phaseRelPermExponent␣
↪).
    setApplyDefaultValue( 1.0 ).
    setInputFlag( InputFlags::OPTIONAL ).
    setDescription( "Minimum relative permeability power law exponent for each phase" );


  registerWrapper( viewKeyStruct::phaseRelPermMaxValueString(), &m_phaseRelPermMaxValue␣
↪).
    setApplyDefaultValue( 0.0 ).
    setInputFlag( InputFlags::OPTIONAL ).
    setDescription( "Maximum relative permeability value for each phase" );

  registerWrapper( viewKeyStruct::volFracScaleString(), &m_volFracScale ).
    setApplyDefaultValue( 1.0 ).
    setDescription( "Factor used to scale the phase relative permeability, defined as:␣
↪one minus the sum of the phase minimum volume fractions." );
```

[source: src/coreComponents/constitutive/relativePermeability/BrooksCoreyRelativePermeability.cpp]

The XML block

We are ready to use the relative permeability model in GEOS. The corresponding XML block (child node of the "Constitutive" block) reads:

```
<Constitutive>
  <BrooksCoreyBakerRelativePermeability name="relperm"
                                        phaseNames="{oil, gas, water}"
                                        phaseMinVolumeFraction="{0.05, 0.05, 0.05}"
                                        waterOilRelPermExponent="{2.5, 1.5}"
                                        waterOilRelPermMaxValue="{0.8, 0.9}"
                                        gasOilRelPermExponent="{3, 3}"
                                        gasOilRelPermMaxValue="{0.4, 0.9}"/>
<Constitutive>
```

With this construct, we instruct the `ConstitutiveManager` class (whose `catalogName` is "Constitutive") to instantiate a subgroup of type `BrooksCoreyRelativePermeability`. We also fill the data members of the values that we want to use for the simulation. For a simulation with multiple regions, we could define multiple relative permeability models in the "Constitutive" XML block (yielding multiple relperm subgroups in GEOS), with a unique name attribute for each model.

For more examples on how to contribute to GEOS, please read Adding a new Physics Solver

Input Schema Generation

A schema file is a useful tool for validating input .xml files and constructing user-interfaces. Rather than manually maintaining the schema during development, GEOS is designed to automatically generate one by traversing the documentation structure.

To generate the schema, run GEOS with the input, schema, and the (optional) schema_level arguments, i.e.: `geosx -i input.xml -s schema.xsd`. There are two ways to limit the scope of the schema:

1. Setting the verbosity flag for an object in the documentation structure. If the schema-level argument is used, then only objects (and their children) and attributes with `(verbosity < schema-level)` will be output.
2. By supplying a limited input xml file. When GEOS builds its data structure, it will only include objects that are listed within the xml (or those that are explicitly appended when those objects are initialized). The code will add all available *attributes* for these objects to the schema.

To take advantage of this design it is necessary to use the automatic xml parsing approach that relies upon the documentation node. If values are read in manually, then the schema can not be used to validate xml those inputs.

Note: the lightweight xml parser that is used in GEOS cannot be used to validate inputs with the schema directly. As such, it is necessary to use an external tool for validation, such as the geosx_tools python module.

Working with data in GEOS

In `GEOS`, data is typically registered in the *Data Repository*. This allows for the writing/reading of data to/from restart and plot files. Any object that derives from *Group* may have data registered on it through the methods described in *Group*. Similarly, accessing data from outside the scope of an object is possible through one of the various `Group::get()`. Of course, for temporary data that does not need to persist between cycles, or across physics packages, you may simply define member or local variable which will not be registered with the *Data Repository*.

Working with data on the Mesh objects

The mesh objects in GEOS such as the FaceManager or NodeManager, are derived from ObjectManagerBase, which in turn derives from *Group*. The important distinction is that ObjectManagerBase contains various members that are useful when defining mesh object managers. When considering data that is attached to a mesh object, we group the data into two categories:

- Intrinsic data is data that is required to describe the object. For instance, to define a Node, the NodeManager contains an array of positions corresponding to each Node it contains. Thus the ReferencePosition is Intrinsic data. Intrinsic data is almost always a member of the mesh object, and is registered on the mesh object in the constructor of mesh object itself.
- Field data (or Extrinsic data) is data that is not required to define the object. For instance, a physics package may request that a Velocity value be stored on the nodes. Appropriately the data will be registered on the NodeManager. However, this data is not required to define a Node, and is viewed as Fields or Extrinsic. Field data is never a member of the mesh object, and is typically registered on the mesh object outside of the definition of the mesh object (i.e. from a physics solver).

Registering Intrinsic data on a Mesh Object

As mentioned above, Intrinsic data is typically a member of the mesh object, and is registered in the constructor of the mesh Object. Taking the NodeManager and the referencePosition as an example, we point out that the reference position is actually a member in the NodeManager.

```
  /**
   * @brief Get the mutable reference position array. This table will contain all the␣
↪node coordinates.
   * @return reference position array
   */
  array2d< real64, nodes::REFERENCE_POSITION_PERM > & referencePosition() { return m_
↪referencePosition; }

  /**
   * @brief Provide an immutable arrayView of the reference position. This table will␣
↪contain all the node coordinates.
   * @return an immutable arrayView of the reference position.
   */

  arrayView2d< real64 const, nodes::REFERENCE_POSITION_USD > referencePosition() const
  { return m_referencePosition; }
```

This member is registered in the constructor for the NodeManager.

```
NodeManager::NodeManager( string const & name,
                          Group * const parent ):
  ObjectManagerBase( name, parent ),
  m_referencePosition( 0, 3 )
{
  registerWrapper( viewKeyStruct::referencePositionString(), &m_referencePosition );
```

Finally in order to access this data, the NodeManager provides explicit accessors.

```
  /**
   * @brief Get the mutable reference position array. This table will contain all the␣
```

(continues on next page)

(continued from previous page)

```
↪node coordinates.
   * @return reference position array
   */
  array2d< real64, nodes::REFERENCE_POSITION_PERM > & referencePosition() { return m_
↪referencePosition; }

  /**
   * @brief Provide an immutable arrayView of the reference position. This table will␣
↪contain all the node coordinates.
   * @return an immutable arrayView of the reference position.
   */

  arrayView2d< real64 const, nodes::REFERENCE_POSITION_USD > referencePosition() const
  { return m_referencePosition; }
```

Thus the interface for `Intrinsic` data is set by the object that it is a part of, and the developer may only access the data through the accesssors from outside of the mesh object class scope.

Registering Field data on a Mesh Object

To register `Field` data, there are many ways a developer may proceed. We will use the example of registering a `totalDisplacement` on the `NodeManager` from the `SolidMechanics` solver. The most general approach is to define a string key and call one of the Group::registerWrapper() functions from `SolverBase::registerDataOnMesh()`. Then when you want to use the data, you can call `Group::getReference()`. For example this would look something like:

```
void SolidMechanicsLagrangianFEM::registerDataOnMesh( Group * const MeshBodies )
{
  for( auto & mesh : MeshBodies->GetSubGroups() )
  {
    NodeManager & nodes = mesh.second->groupCast< MeshBody * >()->getMeshLevel( 0 ).
↪getNodeManager();

    nodes.registerWrapper< array2d< real64, nodes::TOTAL_DISPLACEMENT_PERM > >(␣
↪keys::totalDisplacement ).
      setPlotLevel( PlotLevel::LEVEL_0 ).
      setRegisteringObjects( this->getName()).
      setDescription( "An array that holds the total displacements on the nodes." ).
      reference().resizeDimension< 1 >( 3 );
  }
}
```

and

```
arrayView2d< real64, nodes::TOTAL_DISPLACEMENT_USD > const & u = nodes.getReference<␣
↪array2d< real64, nodes::TOTAL_DISPLACEMENT_PERM > >( keys::totalDisplacement );
... do something with u
```

This approach is flexible and extendible, but is potentially error prone due to its verbosity and lack of information centralization. Therefore we also provide a more controlled/uniform method by which to register and extract commonly used data on the mesh. The `trait approach` requires the definition of a `traits struct` for each data object that will be supported. To apply the `trait approach` to the example use case shown above, there should be the following definition somewhere in a header file:

```
namespace fields
{
struct totalDisplacement
{
  static constexpr auto key = "totalDisplacement";
  using DataType = real64;
  using Type = array2d< DataType, nodes::TOTAL_DISPLACEMENT_PERM >;
  static constexpr DataType defaultValue = 0;
  static constexpr auto plotLevel = dataRepository::PlotLevel::LEVEL_0;

  /// Description of the data associated with this trait.
  static constexpr auto description = "An array that holds the total displacements on␣
↪the nodes.";
};
}
```

Also note that you should use the `DECLARE_FIELD` C++ macro that will perform this tedious task for you. Then the registration is simplified as follows:

```
void SolidMechanicsLagrangianFEM::registerDataOnMesh( Group * const MeshBodies )
{
  for( auto & mesh : MeshBodies->GetSubGroups() )
  {
    NodeManager & nodes = mesh.second->groupCast< MeshBody * >()->getMeshLevel( 0 ).
↪getNodeManager();
    nodes.registerField< fields::totalDisplacement >( this->getName() ).resizeDimension<␣
↪1 >( 3 );
  }
}
```

And to extract the data, the call would be:

```
arrayView2d< real64, nodes::TOTAL_DISPLACEMENT_USD > const & u = nodes.getField<␣
↪fields::totalDisplacement >();
... do something with u
```

The end result of the `trait approach` to this example is that the developer has defined a standard specification for `totalDisplacement`, which may be used uniformly across the code.

Mesh Hierarchy

In GEOS, the mesh structure consists of a hierarchy of classes intended to encapsulate data and functionality for each topological type. Each class in the mesh hierarchy represents a distinct topological object, such as a nodes, edges, faces, elements, etc. The mesh data structure is illustrated in an object instantiation hierarchy. The object instantiation hierarchy differs from a "class hierarchy" in that it shows how instantiations of each class relate to each other in the data hierarchy rather than how each class type relates to each other in an inheritance diagram.

To illustrate the mesh hierarchy, we propose to present it along with a model with two regions (Top and Bottom) (Fig. 1.82).

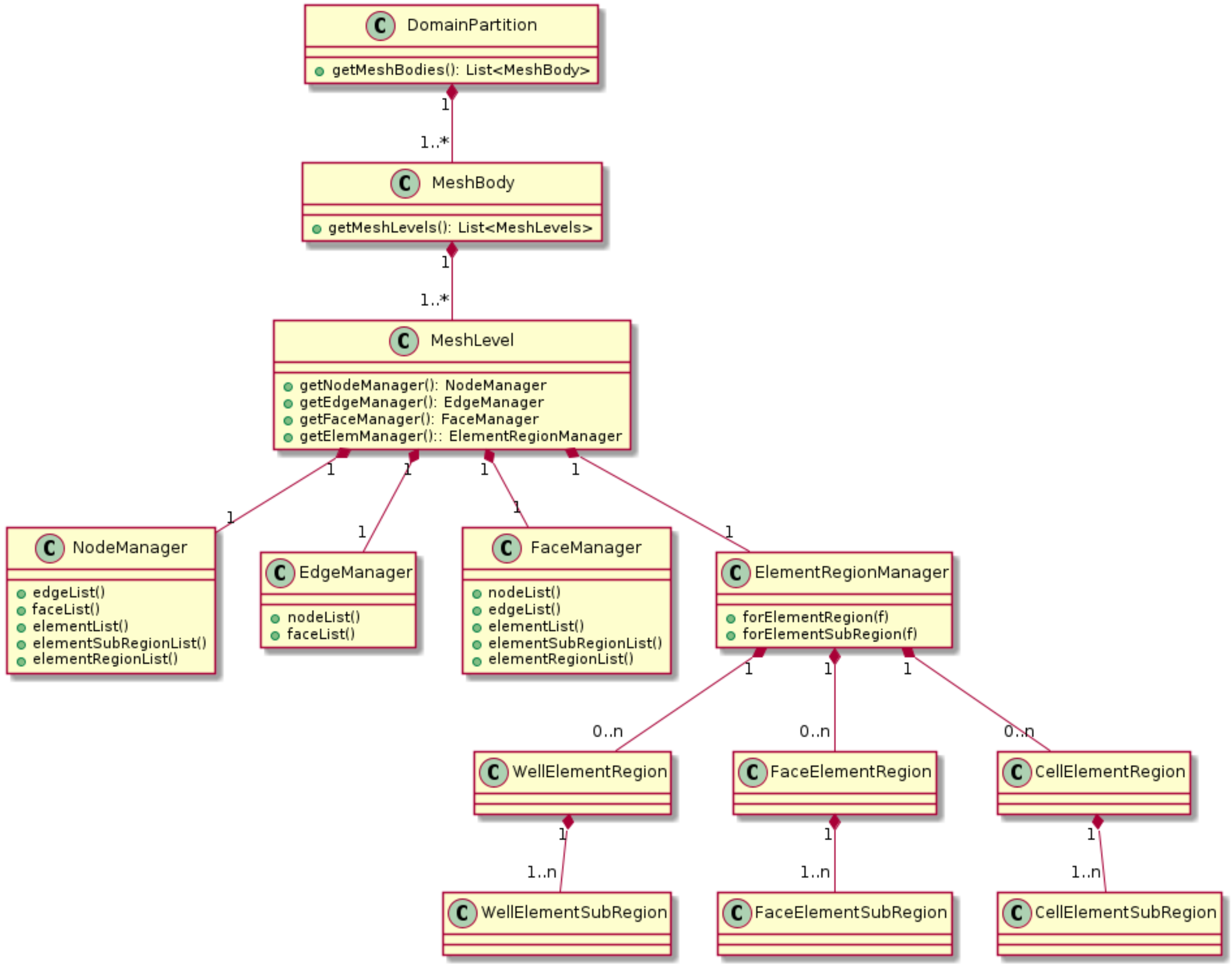


Fig. 1.81: Object instances describing the mesh domain. Cardinalities and relationships are indicated.

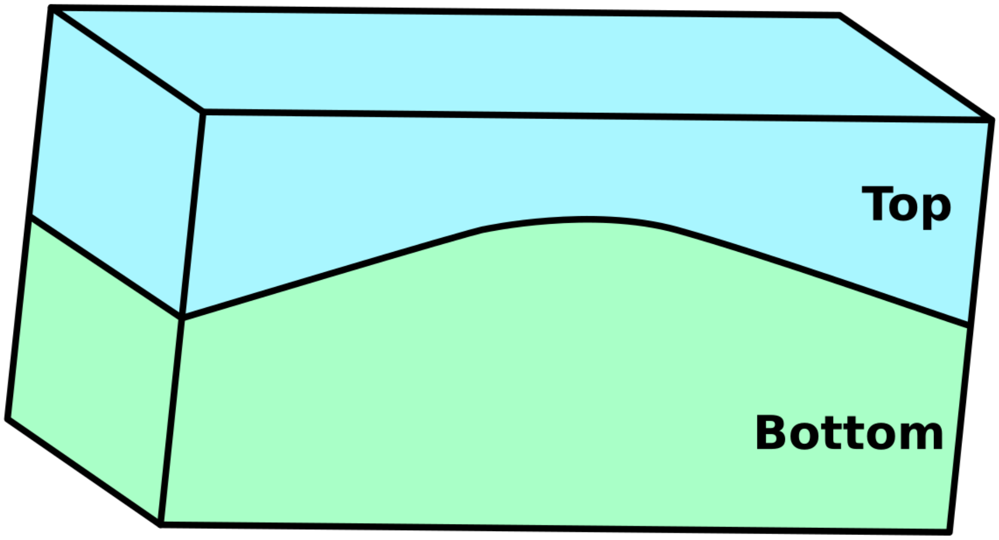


Fig. 1.82: Example of a model with two regions

DomainPartition

In MeshObjectInstantiationHierarchy the top level object DomainPartition represents a partition of the decomposed physical domain. At this time there is a unique DomainPartition for every MPI rank.

Note: Hypothetically, there may be more than one DomainPartition in cases where the ranks are overloaded. Currently GEOS does not support overloading multiple DomainPartition's onto a rank, although this may be a future option if its use is properly motivated.

For instance, the model presented as example can be split into two different domains (Fig. 1.83).

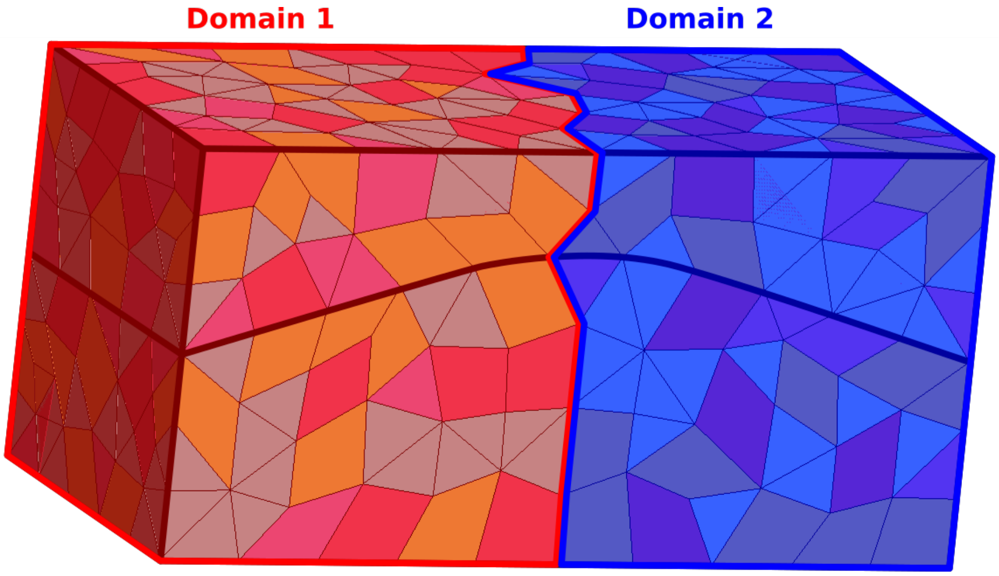


Fig. 1.83: Mesh partioned in two DomainPartition

MeshBody

The MeshBody represents a topologically distinct mesh body. For instance if a simulation of two separate spheres was required, then one option would be to have both spheres as part of a single mesh body, while another option would be to have each sphere be a individual body.

Note: While not currently utilized in GEOS, the intent is to have the ability to handle the bodies in a multi-body mesh on an individual basis. For instance, when conducting high resolution crush simulations of granular materials (i.e. sand), it may be advantagous to represent each particle as a MeshBody.

MeshLevel

The MeshLevel is intended to facilitate the representation of a multi-level discretization of a MeshBody.

Note: In current practice, the code utilizes a single MeshLevel until such time as we implement a proper multi-level mesh capability. The MeshLevel contains the main components that compose a discretized mesh in GEOS.

Topological Mesh Objects

Each of the "Manager" objects are responsible for holding child objects, data, and providing functionality specific to a single topological object. Each topological object that is used to define a discretized mesh has a "Manager" to allow for simple traversal over the hierarchy, and to provide modular access to data. As such, the NodeManager manages data for the "nodes", the EdgeManager manages data for the edges, the FaceManager holds data for the faces and the ElementRegionManager manages the physical groups within the MeshLevel (regions, fractures, wells etc…). Additionally each manager contains index maps to the other types objects that are connected to the objects in that manager. For instance, the FaceManager contains a downward pointing map that gives the nodes that comprise each face in the mesh. Similarly the FaceManager contains an upward pointing map that gives the elements that are connected to a face.

ElementRegionManager

The element data structure is significantly more complicated than the other Managers. While the other managers are "flat" across the MeshLevel, the element data structure seeks to provide a hierarchy in order to define groupings of the physical problem, as well as collecting discretization of similar topology. At the top of the element branch of the hierarchy is the ElementRegionManager. The ElementRegionManager holds a collection of instantiations of ElementRegionBase derived classes.

ElementRegion

Conceptually the ElementRegion are used to defined regions of the problem domain where a PhysicsSolver will be applied.

- The CellElementRegion is related to all the polyhedra
- The FaceElementRegion is related to all the faces that have physical meaning in the domain, such as fractures and faults. This object should not be mistaken with the FaceManager. The FaceManager handles all the faces of the mesh, not only the faces of interest.
- The WellElementRegion is related to the well geometry.

An ElementRegion also has a list of materials allocated at each quadrature point across the entire region. One example of the utility of the ElementRegion is the case of the simulation of the mechanics and flow within subsurface reservoir with an overburden. We could choose to have two ElementRegion, one being the reservoir, and one for the overburden. The mechanics solver would be applied to the entire problem, while the flow problem would be applied only to the reservoir region.

Each ElementRegion holds some number of ElementSubRegion. The ElementSubRegion is meant to hold all the element topologies present in an ElementSubRegion in their own groups. For instance, for a CellElementRegion, there may be one CellElementSubRegion for all tetrahedra, one for all hexahedra, one for all wedges and one for all the pyramids (Fig. 1.84).

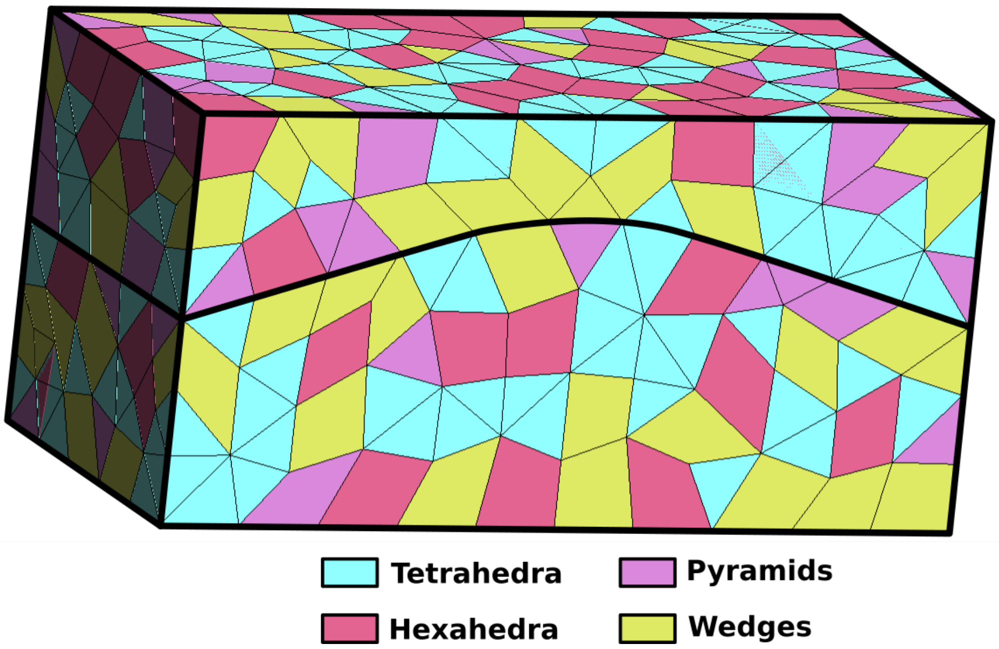


Fig. 1.84: Model meshed with different cell types

Now that all the classes of the mesh hierarchy has been described, we propose to adapt the diagram presented in Fig. 1.81 to match with the example presented in Fig. 1.82.

DoF Manager

This will contains a description of the DoF manager in GEOS.

Brief description

The main aim of the Degrees-of-Freedom (DoF) Manager class is to handle all degrees of freedom associated with fields that exist on mesh elements, faces, edges and nodes. It creates a map between local mesh objects and global DoF indices. Additionally, DofManager simplifies construction of system matrix sparsity patterns.

Key concepts are locations and connectors. Locations, that can be elements, faces, edges or nodes, represent where the DoF is assigned. For example, a DoF for pressure in a two-point flux approximation will be on a cell (i.e. element), while a displacement DoF for structural equations will be on a node. The counterparts of locations are connectors, that are the geometrical entities that link together different DoFs that create the sparsity pattern. Connectors can be elements, faces, edges, nodes or none. Using the same example as before, connectors will be faces and cells, respectively. The case of a mass matrix, where every element is linked only to itself, is an example when there are no connectors, i.e. these have to be set to none.

DoFs located on a mesh object are owned by the same rank that owns the object in parallel mesh partitioning. Two types of DoF numbering are supported, with the difference only showing in parallel runs of multi-field problems.

- Initially, each field is assigned an independent DoF numbering that starts at 0 and is contiguous across all MPI ranks. Within each rank, locally owned DoFs are numbered sequentially across mesh locations, and within each mesh location (e.g. node) - sequentially according to component number. With this numbering, sparsity patterns can be constructed for individual sub-matrices that represent diagonal/off-diagonal blocks of the global coupled system matrix.

- After all fields have been declared, the user can call `DofManager::reorderByRank()`, which constructs a globally contiguous DoF numbering across all fields. Specifically, all DoFs owned by rank 0 are numbered field-by-field starting from 0, then those on rank 1, etc. This makes global system sparsity pattern compatible with linear algebra packages that only support contiguous matrix rows on each rank. At this point, coupled system matrix sparsity pattern can be constructed.

Thus, each instance of `DofManager` only supports one type of numbering. If both types are required, the user is advised to maintain two separate instances of `DofManager`.

`DofManager` allocates a separate "DOF index" array for each field on the mesh. It is an array of global indices, where each value represents the first DoF index for that field and location (or equivalently, the row and column offset of that location's equations and variables for the field in the matrix). For example, if index array for a field with 3 components contains the value N, global DoF numbers for that location will be N, N+1, N+2. DoF on ghosted locations have the same indices as on the owning rank. The array is stored under a generated key, which can be queried from the DoF manager, and is typically used in system assembly.

Methods

The main methods of `DoF Manager` are:

- `setDomain`: sets the domain containing mesh bodies to operate on `domain` identifies the global domain

```
void setDomain( DomainPartition * const domain );
```

- `addField`: creates a new set of DoF, labeled `field`, with specific `location`. Default number of `components` is 1, like for pressure in flux. Default `regions` is the empty string, meaning all domain.

```
void addField( string const & fieldName,
               Location const location,
               localIndex const components,
               arrayView1d< string const > const & regions );
```

- `addCoupling`: creates a coupling between two fields (`rowField` and `colField`) according to a given `connectivity` in the regions defined by `regions`. Both fields (row and column) must have already been defined on the regions where is required the coupling among them. Default value for `regions` is the whole intersection between the regions where the first and the second fields are defined. This method also creates the coupling between `colField` and `rowField`, i.e. the transpose of the rectangular sparsity pattern. This default behaviour can be disabled by passing `symmetric = false`.

```
void addCoupling( string const & rowField,
                  string const & colField,
                  Connectivity const connectivity,
                  arrayView1d< string const > const & regions,
                  bool const symmetric );
```

- `reorderByRank`: finish populating field and coupling information and apply DoF re-numbering

```
void reorderByRank();
```

- `getKey`: returns the "key" associated with the field, that can be used to access the index array on the mesh object manager corresponding to field's location.

```
string const & getKey( string const & fieldName );
```

- `clear`: removes all fields, releases memory and re-opens the DofManager

```
void clear();
```

- `setSparsityPattern`: populates the sparsity for the given `rowField` and `colField` into `matrix`. Closes the matrix if `closePattern` is `true`.

```
void setSparsityPattern( MATRIX & matrix,
                         string const & rowField,
                         string const & colField,
                         bool closePattern = true) const;
```

- `setSparsityPattern`: populates the sparsity for the full system matrix into `matrix`. Closes the matrix if `closePattern` is `true`.

```
void setSparsityPattern( MATRIX & matrix,
                         bool closePattern = true ) const;
```

- `numGlobalDofs`: returns the total number of DoFs across all processors for the specified name `field` (if given) or all fields (if empty).

```
globalIndex numGlobalDofs( string const & field = "" ) const;
```

- `numLocalDofs`: returns the number of DoFs on this process for the specified name `field` (if given) or all fields (if empty).

```
localIndex numLocalDofs( string const & field = "" ) const;
```

- `printFieldInfo`: prints a short summary of declared fields and coupling to the output stream `os`.

```
void printFieldInfo( std::ostream & os = std::cout ) const;
```

Example

Here we show how the sparsity pattern is computed for a simple 2D quadrilateral mesh with 6 elements. Unknowns are pressure, located on the element center, and displacements (x and y components), located on the nodes. For fluxes, a two-point flux approximation (TPFA) is used. The representation of the sparsity pattern of the $\mathsf{C_L}$ matrix (connectors/locations) for the simple mesh, shown in Fig. 1.85, is reported in Fig. 1.86. It can be noticed that the two unknowns for the displacements x and y are grouped together. Elements are the connectivity for DoF on nodes (Finite Element Method for displacements) and on elements (pressures). Faces are the connectivity for DoF on elements (Finite Volume Method for pressure), being the flux computation based on the pressure on the two adjacent elements.

Fig. 1.85: Small 2D quadrilateral mesh used for this examples. Nodes are label with black numbers, elements with light gray numbers and faces with italic dark gray numbers.

Fig. 1.86: Sparsity pattern of the binary matrix connections/locations.

The global sparsity pattern, shown in Fig. 1.87, is obtained through the symbolic multiplication of the transpose of the matrix $\mathsf{C_L}$ and the matrix itself, i.e. $\mathsf{P} = \mathsf{C_L^T C_L}$.

Fig. 1.87: Sparsity pattern of the global matrix, where red and green entries are related to the displacement field and to the pressure field, respectively. Blue entries represent coupling blocks.

Real mesh and patterns

Now we build the pattern of the Jacobian matrix for a simple 3D mesh, shown in Fig. 1.88. Fields are:

- displacement (location: node, connectivity: element) defined on the blue, orange and red regions;
- pressure (location: element, connectivity: face) defined on the green, orange and red regions;
- mass matrix (location: element, connectivity: element) defined on the green region only.

Moreover, following coupling are imposed:

- displacement-pressure (connectivity: element) on the orange region only;
- pressure-mass matrix and transpose (connectivity: element) everywhere it is possibile.

Fig. 1.88: Real mesh used to compute the Jacobian pattern.

Fig. 1.89 shows the global pattern with the field-based ordering of unknowns. Different colors mean different fields. Red unkwnons are associated with displacement, yellow ones with pressure and blue ones with mass matrix. Orange means the coupling among displacement and pressure, while green is the symmetric coupling among pressure and mass matrix.

Fig. 1.89: Global pattern with field-based ordering. Red is associated with displacement unknowns, yellow with pressure ones and blue with those of mass matrix field. Orange means the coupling among displacement and pressure, while green is the symmetric coupling among pressure and mass matrix.

Fig. 1.90 shows the global pattern with the MPI rank-based ordering of unknowns. In this case, just two processes are used. Again, different colors indicate different ranks.

LvArray

Use in GEOS

LvArray containers are used in GEOS as primary storage mechanism for mesh topology, field data and any other type of "large" data sets (i.e. ones that scale with the size of the problem). When allocating a new field, using one of LvArray containers is mandatory if the data is meant to be used in any computational kernel. The file common/DataTypes.hpp provides shorthand aliases for commonly used containers:

```
/**
 * @name Aliases for LvArray::Array class family.
 */
///@{

/// Multidimensional array type. See LvArray:Array for details.
template< typename T,
          int NDIM,
          typename PERMUTATION=camp::make_idx_seq_t< NDIM > >
```

(continues on next page)

Fig. 1.90: Global pattern with MPI rank-based ordering. Red unkwnons are owned by rank 0 and green ones by rank 1. Blue indicates the coupling among the two processes.

(continued from previous page)

```
using Array = LvArray::Array< T, NDIM, PERMUTATION, localIndex, LvArray::ChaiBuffer >;

/// Multidimensional array view type. See LvArray:ArrayView for details.
template< typename T,
          int NDIM,
          int USD = NDIM - 1 >
using ArrayView = LvArray::ArrayView< T, NDIM, USD, localIndex, LvArray::ChaiBuffer >;

/// Multidimensional array slice type. See LvArray:ArraySlice for details.
template< typename T, int NDIM, int USD = NDIM - 1 >
using ArraySlice = LvArray::ArraySlice< T, NDIM, USD, localIndex >;

/// Multidimensional stack-based array type. See LvArray:StackArray for details.
template< typename T, int NDIM, int MAXSIZE, typename PERMUTATION=camp::make_idx_seq_t<␣
↪NDIM > >
using StackArray = LvArray::StackArray< T, NDIM, PERMUTATION, localIndex, MAXSIZE >;

///@}

/**
 * @name Short-hand aliases for commonly used array types.
 */
///@{

/// Alias for a local (stack-based) rank-1 tensor type
using R1Tensor = Tensor< real64, 3 >;
/// Alias for a local (stack-based) rank-1 tensor type using 32 bits integers
using R1Tensor32 = Tensor< real32, 3 >;

/// Alias for a local (stack-based) rank-2 Voigt tensor type
using R2SymTensor = Tensor< real64, 6 >;


/// Alias for 1D array.
template< typename T >
using array1d = Array< T, 1 >;

/// Alias for 1D array view.
template< typename T >
using arrayView1d = ArrayView< T, 1 >;

/// Alias for 1D array slice.
template< typename T, int USD = 0 >
using arraySlice1d = ArraySlice< T, 1, USD >;

/// Alias for 1D stack array.
template< typename T, int MAXSIZE >
```

(continues on next page)

(continued from previous page)

```
using stackArray1d = StackArray< T, 1, MAXSIZE >;

/// Alias for 2D array.
template< typename T, typename PERMUTATION=camp::make_idx_seq_t< 2 > >
using array2d = Array< T, 2, PERMUTATION >;

/// Alias for 2D array view.
template< typename T, int USD = 1 >
using arrayView2d = ArrayView< T, 2, USD >;

/// Alias for 2D array slice.
template< typename T, int USD = 1 >
using arraySlice2d = ArraySlice< T, 2, USD >;

/// Alias for 2D stack array.
template< typename T, int MAXSIZE >
using stackArray2d = StackArray< T, 2, MAXSIZE >;

/// Alias for 3D array.
template< typename T, typename PERMUTATION=camp::make_idx_seq_t< 3 > >
using array3d = Array< T, 3, PERMUTATION >;

/// Alias for 3D array view.
template< typename T, int USD=2 >
using arrayView3d = ArrayView< T, 3, USD >;

/// Alias for 3D array slice.
template< typename T, int USD=2 >
using arraySlice3d = ArraySlice< T, 3, USD >;

/// Alias for 3D stack array.
template< typename T, int MAXSIZE >
using stackArray3d = StackArray< T, 3, MAXSIZE >;

/// Alias for 4D array.
template< typename T, typename PERMUTATION=camp::make_idx_seq_t< 4 > >
using array4d = Array< T, 4, PERMUTATION >;

/// Alias for 4D array view.
template< typename T, int USD=3 >
using arrayView4d = ArrayView< T, 4, USD >;

/// Alias for 4D array slice.
template< typename T, int USD=3 >
using arraySlice4d = ArraySlice< T, 4, USD >;

/// Alias for 4D stack array.
template< typename T, int MAXSIZE >
using stackArray4d = StackArray< T, 4, MAXSIZE >;

/// Alias for 5D array.
template< typename T, typename PERMUTATION=camp::make_idx_seq_t< 5 > >
```

(continues on next page)

(continued from previous page)

```
using array5d = Array< T, 5, PERMUTATION >;

/// Alias for 5D array view.
template< typename T, int USD=4 >
using arrayView5d = ArrayView< T, 5, USD >;

/// Alias for 5D array slice.
template< typename T, int USD=4 >
using arraySlice5d = ArraySlice< T, 5, 4 >;

/// Alias for 5D stack array.
template< typename T, int MAXSIZE >
using stackArray5d = StackArray< T, 5, MAXSIZE >;

///@}

/**
 * @name Aliases for sorted arrays and set types.
 */
///@{

/// A set of local indices.
template< typename T >
using set = std::set< T >;

/// A sorted array of local indices.
template< typename T >
using SortedArray = LvArray::SortedArray< T, localIndex, LvArray::ChaiBuffer >;

/// A sorted array view of local indices.
template< typename T >
using SortedArrayView = LvArray::SortedArrayView< T, localIndex, LvArray::ChaiBuffer >;

///@}

/**
 * @name Aliases for LvArray::ArrayOfArrays class family.
 */
///@{

/// Array of variable-sized arrays. See LvArray::ArrayOfArrays for details.
template< typename T, typename INDEX_TYPE=localIndex >
using ArrayOfArrays = LvArray::ArrayOfArrays< T, INDEX_TYPE, LvArray::ChaiBuffer >;

/// View of array of variable-sized arrays. See LvArray::ArrayOfArraysView for details.
template< typename T, typename INDEX_TYPE=localIndex, bool CONST_SIZES=std::is_const< T >
↪::value >
using ArrayOfArraysView = LvArray::ArrayOfArraysView< T, INDEX_TYPE const, CONST_SIZES,␣
↪LvArray::ChaiBuffer >;

/// Array of variable-sized sets. See LvArray::ArrayOfSets for details.
template< typename T, typename INDEX_TYPE=localIndex >
```

(continues on next page)

(continued from previous page)

```
using ArrayOfSets = LvArray::ArrayOfSets< T, INDEX_TYPE, LvArray::ChaiBuffer >;

/// View of array of variable-sized sets. See LvArray::ArrayOfSetsView for details.
template< typename T, typename INDEX_TYPE=localIndex >
using ArrayOfSetsView = LvArray::ArrayOfSetsView< T, INDEX_TYPE const, LvArray::ChaiBuffer >;

/// Alias for Sparsity pattern class.
template< typename COL_INDEX, typename INDEX_TYPE=localIndex >
using SparsityPattern = LvArray::SparsityPattern< COL_INDEX, INDEX_TYPE, LvArray::ChaiBuffer >;

/// Alias for Sparsity pattern View.
template< typename COL_INDEX, typename INDEX_TYPE=localIndex >
using SparsityPatternView = LvArray::SparsityPatternView< COL_INDEX, INDEX_TYPE const, LvArray::ChaiBuffer >;

/// Alias for CRS Matrix class.
template< typename T, typename COL_INDEX=globalIndex >
using CRSMatrix = LvArray::CRSMatrix< T, COL_INDEX, localIndex, LvArray::ChaiBuffer >;

/// Alias for CRS Matrix View.
template< typename T, typename COL_INDEX=globalIndex >
using CRSMatrixView = LvArray::CRSMatrixView< T, COL_INDEX, localIndex const, LvArray::ChaiBuffer >;

///@}
```

LvArray documentation

Please refer to the full LvArray documentation for details on each of the classes.

Kernel interface

Finite Element Method Kernel Interface

The finite element method kernel interface (FEMKI) specifies an API for the launching of computational kernels for solving physics discretized using the finite element method. Using this approach, a set of generic element looping pattens and kernel launching functions may be implemented, and reused by various physics solvers that contain kernels conforming to the FEMKI.

There are several main components of the FEMKI:

1. A collection of element looping functions that provide various looping patterns, and call the `launch` function.
2. The kernel interface, which is specified by the finiteElement::KernelBase class. Each physics solver will define a class that contains its kernels functions, most likely deriving, or conforming to the API specified by the *KernelBase* class. Also part of this class will typically be a nested `StackVariables` class that defines a collection of stack variables for use in the various kernel interface functions.
3. A `launch` function, which launches the kernel, and calls the kernel interface functions conforming to the interface defined by `KernelBase`. This function is actually a member function of the `Kernel` class, so it may be overridden

by a specific physics kernel, allowing complete customization of the interface, while maintaining the usage of the looping patterns.

A Generic Element Looping Pattern

One example of a looping pattern is the regionBasedKernelApplication function.

The contents of the looping function are displayed here:

```
/**
 * @brief Performs a loop over specific regions (by type and name) and calls a kernel␣
↪launch on the subregions
 *   with compile time knowledge of sub-loop bounds such as number of nodes and␣
↪quadrature points per element.
 * @tparam POLICY The RAJA launch policy to pass to the kernel launch.
 * @tparam CONSTITUTIVE_BASE The common base class for constitutive pass-thru/dispatch␣
↪which gives the kernel
 *   launch compile time knowledge of the constitutive model. This is achieved through a␣
↪call to the
 *   ConstitutivePassThru function which should have a specialization for CONSTITUTIVE_
↪BASE implemented in
 *   order to perform the compile time dispatch.
 * @tparam SUBREGION_TYPE The type of subregion to loop over. TODO make this a parameter␣
↪pack?
 * @tparam KERNEL_FACTORY The type of @p kernelFactory, typically an instantiation of @c␣
↪KernelFactory, and
 *   must adhere to that interface.
 * @param mesh The MeshLevel object.
 * @param targetRegions The names of the target regions(of type @p SUBREGION_TYPE) to␣
↪apply the @p KERNEL_TEMPLATE.
 * @param finiteElementName The name of the finite element.
 * @param constitutiveStringName The key to the constitutive model name found on the␣
↪Region.
 * @param kernelFactory The object used to construct the kernel.
 * @return The maximum contribution to the residual, which may be used to scale the␣
↪residual.
 *
 * @details Loops over all regions Applies/Launches a kernel specified by the @p KERNEL_
↪TEMPLATE through
 * #::geos::finiteElement::KernelBase::kernelLaunch().
 */
template< typename POLICY,
          typename CONSTITUTIVE_BASE,
          typename SUBREGION_TYPE,
          typename KERNEL_FACTORY >
static
real64 regionBasedKernelApplication( MeshLevel & mesh,
                                     arrayView1d< string const > const & targetRegions,
                                     string const & finiteElementName,
                                     string const & constitutiveStringName,
                                     KERNEL_FACTORY & kernelFactory )
{
  GEOS_MARK_FUNCTION;
```

(continues on next page)

(continued from previous page)

```
  // save the maximum residual contribution for scaling residuals for convergence␣
↪criteria.
  real64 maxResidualContribution = 0;

  NodeManager & nodeManager = mesh.getNodeManager();
  EdgeManager & edgeManager = mesh.getEdgeManager();
  FaceManager & faceManager = mesh.getFaceManager();
  ElementRegionManager & elementRegionManager = mesh.getElemManager();

  // Loop over all sub-regions in regions of type SUBREGION_TYPE, that are listed in the␣
↪targetRegions array.
  elementRegionManager.forElementSubRegions< SUBREGION_TYPE >( targetRegions,
                                                               [&constitutiveStringName,
                                                                &maxResidualContribution,
                                                                &nodeManager,
                                                                &edgeManager,
                                                                &faceManager,
                                                                &kernelFactory,
                                                                &finiteElementName]
                                                                 ( localIndex const␣
↪targetRegionIndex, auto & elementSubRegion )
  {
    localIndex const numElems = elementSubRegion.size();

    // Get the constitutive model...and allocate a null constitutive model if required.

    constitutive::ConstitutiveBase * constitutiveRelation = nullptr;
    constitutive::NullModel * nullConstitutiveModel = nullptr;
    if( elementSubRegion.template hasWrapper< string >( constitutiveStringName ) )
    {
      string const & constitutiveName = elementSubRegion.template getReference< string >
↪( constitutiveStringName );
      constitutiveRelation = &elementSubRegion.template getConstitutiveModel(␣
↪constitutiveName );
    }
    else
    {
      nullConstitutiveModel = &elementSubRegion.template registerGroup<␣
↪constitutive::NullModel >( "nullModelGroup" );
      constitutiveRelation = nullConstitutiveModel;
    }

    // Call the constitutive dispatch which converts the type of constitutive model into␣
↪a compile time constant.
    constitutive::ConstitutivePassThru< CONSTITUTIVE_BASE >::execute(␣
↪*constitutiveRelation,
                                                                      [&
↪maxResidualContribution,
                                                                       &nodeManager,
                                                                       &edgeManager,
                                                                       &faceManager,
                                                                       targetRegionIndex,
```

(continues on next page)

(continued from previous page)

```
                                                                      &kernelFactory,
                                                                      &elementSubRegion,
                                                                      &
↪finiteElementName,
                                                                      numElems]
                                                                       ( auto &␣
↪castedConstitutiveRelation )
   {
     FiniteElementBase &
     subRegionFE = elementSubRegion.template getReference< FiniteElementBase >(␣
↪finiteElementName );

     finiteElement::FiniteElementDispatchHandler< SELECTED_FE_TYPES >::dispatch3D(␣
↪subRegionFE,
                                                                                   [&
↪maxResidualContribution,
                                                                                    &
↪nodeManager,
                                                                                    &
↪edgeManager,
                                                                                    &
↪faceManager,
                                                                                   ␣
↪targetRegionIndex,
                                                                                    &
↪kernelFactory,
                                                                                    &
↪elementSubRegion,
                                                                                   ␣
↪numElems,
                                                                                    &
↪castedConstitutiveRelation] ( auto const finiteElement )
     {
       auto kernel = kernelFactory.createKernel( nodeManager,
                                                 edgeManager,
                                                 faceManager,
                                                 targetRegionIndex,
                                                 elementSubRegion,
                                                 finiteElement,
                                                 castedConstitutiveRelation );

       using KERNEL_TYPE = decltype( kernel );

       // Call the kernelLaunch function, and store the maximum contribution to the␣
↪residual.
       maxResidualContribution =
         std::max( maxResidualContribution,
                   KERNEL_TYPE::template kernelLaunch< POLICY, KERNEL_TYPE >( numElems,␣
↪kernel ) );
     } );
   } );
```

(continues on next page)

(continued from previous page)

```
    // Remove the null constitutive model (not required, but cleaner)
    if( nullConstitutiveModel )
    {
      elementSubRegion.deregisterGroup( "nullModelGroup" );
    }

  } );

  return maxResidualContribution;
}
```

This pattern may be used with any kernel class that either:

1. Conforms to the `KernelBase` interface by defining each of the kernel functions in `KernelBase`.
2. Defines its own `kernelLaunch` function that conforms the the signature of `KernelBase::kernelLaunch`. This option essentially allows for a custom kernel that does not conform to the interface defined by `KernelBase` and `KernelBase::kernelLaunch`.

The KernelBase::kernelLaunch Interface

The `kernelLaunch` function is a member of the kernel class itself. As mentioned above, a physics implementation may use the existing `KernelBase` interface, or define its own. The `KernelBase::kernelLaunch` function defines a launching policy, and an internal looping pattern over the quadrautre points, and calls the functions defined by the `KernelBase` as shown here:

```
  template< typename POLICY,
            typename KERNEL_TYPE >
  static
  real64
  kernelLaunch( localIndex const numElems,
                KERNEL_TYPE const & kernelComponent )
  {
    GEOS_MARK_FUNCTION;

    // Define a RAJA reduction variable to get the maximum residual contribution.
    RAJA::ReduceMax< ReducePolicy< POLICY >, real64 > maxResidual( 0 );

    forAll< POLICY >( numElems,
                      [=] GEOS_HOST_DEVICE ( localIndex const k )
    {
      typename KERNEL_TYPE::StackVariables stack;

      kernelComponent.setup( k, stack );
      // #pragma unroll
      for( integer q=0; q<numQuadraturePointsPerElem; ++q )
      {
        kernelComponent.quadraturePointKernel( k, q, stack );
      }
      maxResidual.max( kernelComponent.complete( k, stack ) );
    } );
    return maxResidual.get();
```

(continues on next page)

(continued from previous page)

```
}
```

Each of the KernelBase functions called in the KernelBase::kernelLaunch function are intended to provide a certain amount of modularity and flexibility for the physics implementations. The general purpose of each function is described by the function name, but may be further descibed by the function documentation found here.

Constitutive models

In GEOS, all constitutive models defining fluid and rock properties are implemented in the namespace constitutive and derived from a common base class, ConstitutiveBase. All objects are owned and handled by the ConstitutiveManager.

Standalone models

Standalone constitutive models implement constitutive laws such as:

- mechanical material models (linear elasticity, plasticity, etc.),
- PVT fluid behaviors,
- relative permeability relationships,
- porosity and permeability dependencies on state variables,
- contact laws.

Storage, allocation, and update of properties

Each constitutive model owns, as member variables, LvArray::Array containers that hold the properties (or fields) and their derivatives with respect to the other fields needed to update each property. Each property is stored as an array with the first dimension representing the elementIndex and the second dimension storing the index of the integration point. These dimensions are determined by the number of elements of the subregion on which each constitutive model is registered, and by the chosen discretization method. Vector and tensor fields have an additional dimension to identify their components. Similarly, an additional dimension is necessary for multiphase fluid models with properties defined for each component in each phase. For example, a single-phase fluid model where density and viscosity are functions of the fluid pressure has the following members:

```
array2d< real64 > m_density;
array2d< real64 > m_dDensity_dPressure;
array2d< real64 > m_dDensity_dTemperature;

array2d< real64 > m_density_n;

array2d< real64 > m_viscosity;
array2d< real64 > m_dViscosity_dPressure;
array2d< real64 > m_dViscosity_dTemperature;

array2d< real64 > m_internalEnergy;
array2d< real64 > m_internalEnergy_n;
array2d< real64 > m_dInternalEnergy_dPressure;
array2d< real64 > m_dInternalEnergy_dTemperature;
```

(continues on next page)

(continued from previous page)

```
  array2d< real64 > m_enthalpy;
  array2d< real64 > m_dEnthalpy_dPressure;
  array2d< real64 > m_dEnthalpy_dTemperature;
```

Resizing all fields of the constitutive models happens during the initialization phase by the `ConstitutiveManger` through a call to `ConstitutiveManger::hangConstitutiveRelation`, which sets the appropriate subRegion as the parent Group of each constitutive model object. This function also resizes all fields based on the size of the subregion and the number of quadrature points on it, by calling `CONSTITUTIVE_MODEL::allocateConstitutiveData`. For the single phase fluid example used before, this call is:

```
void SingleFluidBase::allocateConstitutiveData( Group & parent,
                                                localIndex const␣
↪numConstitutivePointsPerParentIndex )
{
  ConstitutiveBase::allocateConstitutiveData( parent,␣
↪numConstitutivePointsPerParentIndex );

  resize( parent.size() );

  m_density.resize( parent.size(), numConstitutivePointsPerParentIndex );
  m_dDensity_dPressure.resize( parent.size(), numConstitutivePointsPerParentIndex );
  m_dDensity_dTemperature.resize( parent.size(), numConstitutivePointsPerParentIndex );
  m_density_n.resize( parent.size(), numConstitutivePointsPerParentIndex );

  m_viscosity.resize( parent.size(), numConstitutivePointsPerParentIndex );
  m_dViscosity_dPressure.resize( parent.size(), numConstitutivePointsPerParentIndex );
  m_dViscosity_dTemperature.resize( parent.size(), numConstitutivePointsPerParentIndex );

  m_internalEnergy.resize( parent.size(), numConstitutivePointsPerParentIndex );
  m_internalEnergy_n.resize( parent.size(), numConstitutivePointsPerParentIndex );
  m_dInternalEnergy_dPressure.resize( parent.size(), numConstitutivePointsPerParentIndex␣
↪);
  m_dInternalEnergy_dTemperature.resize( parent.size(),␣
↪numConstitutivePointsPerParentIndex );

  m_enthalpy.resize( parent.size(), numConstitutivePointsPerParentIndex );
  m_dEnthalpy_dPressure.resize( parent.size(), numConstitutivePointsPerParentIndex );
  m_dEnthalpy_dTemperature.resize( parent.size(), numConstitutivePointsPerParentIndex );
}
```

Any property or field stored on a constitutive model must be updated within a computational kernel to ensure that *host* and *device* memory in GPUs are properly synced, and that any updates are performed on *device*. Some properties are updated within finite element kernels of specific physics (such as stress in a mechanics kernel). Consequently, for each constitutive model class, a corresponding *nameOfTheModelUpdates*, which only contains `LvArray::arrayView` containers to the data, can be captured by value inside computational kernels. For example, for the single phase fluid model *Updates* are:

```
/**
 * @brief Base class for single-phase fluid model kernel wrappers.
 */
class SingleFluidBaseUpdate
{
```

(continues on next page)

(continued from previous page)

```
public:

  /**
   * @brief Get number of elements in this wrapper.
   * @return number of elements
   */
  GEOS_HOST_DEVICE
  localIndex numElems() const { return m_density.size( 0 ); }

  /**
   * @brief Get number of gauss points per element.
   * @return number of gauss points per element
   */
  GEOS_HOST_DEVICE
  localIndex numGauss() const { return m_density.size( 1 ); };

protected:

  /**
   * @brief Constructor.
   * @param density     fluid density
   * @param dDens_dPres derivative of density w.r.t. pressure
   * @param viscosity   fluid viscosity
   * @param dVisc_dPres derivative of viscosity w.r.t. pressure
   */
  SingleFluidBaseUpdate( arrayView2d< real64 > const & density,
                         arrayView2d< real64 > const & dDens_dPres,
                         arrayView2d< real64 > const & viscosity,
                         arrayView2d< real64 > const & dVisc_dPres )
    : m_density( density ),
    m_dDens_dPres( dDens_dPres ),
    m_viscosity( viscosity ),
    m_dVisc_dPres( dVisc_dPres )
  {}

  /**
   * @brief Copy constructor.
   */
  SingleFluidBaseUpdate( SingleFluidBaseUpdate const & ) = default;

  /**
   * @brief Move constructor.
   */
  SingleFluidBaseUpdate( SingleFluidBaseUpdate && ) = default;

  /**
   * @brief Deleted copy assignment operator
   * @return reference to this object
   */
  SingleFluidBaseUpdate & operator=( SingleFluidBaseUpdate const & ) = delete;

  /**
```

(continues on next page)

(continued from previous page)

```
   * @brief Deleted move assignment operator
   * @return reference to this object
   */
  SingleFluidBaseUpdate & operator=( SingleFluidBaseUpdate && ) = delete;


  /// Fluid density
  arrayView2d< real64 > m_density;

  /// Derivative of density w.r.t. pressure
  arrayView2d< real64 > m_dDens_dPres;

  /// Fluid viscosity
  arrayView2d< real64 > m_viscosity;

  /// Derivative of viscosity w.r.t. pressure
  arrayView2d< real64 > m_dVisc_dPres;
```

Because *Updates* classes are responsible for updating the fields owned by the constitutive models, they also implement all functions needed to perform property updates, such as:

```
private:

  /**
   * @brief Compute fluid properties at a single point.
   * @param[in]  pressure the target pressure value
   * @param[out] density fluid density
   * @param[out] viscosity fluid viscosity
   */
  GEOS_HOST_DEVICE
  virtual void compute( real64 const pressure,
                        real64 & density,
                        real64 & viscosity ) const = 0;

  /**
   * @brief Compute fluid properties and derivatives at a single point.
   * @param[in]  pressure the target pressure value
   * @param[out] density fluid density
   * @param[out] dDensity_dPressure fluid density derivative w.r.t. pressure
   * @param[out] viscosity fluid viscosity
   * @param[out] dViscosity_dPressure fluid viscosity derivative w.r.t. pressure
   */
  GEOS_HOST_DEVICE
  virtual void compute( real64 const pressure,
                        real64 & density,
                        real64 & dDensity_dPressure,
                        real64 & viscosity,
                        real64 & dViscosity_dPressure ) const = 0;

  /**
   * @brief Compute fluid properties and derivatives at a single point.
   * @param[in]  pressure the target pressure value
```

(continues on next page)

(continued from previous page)

```
 * @param[in]  temperature the target temperature value
 * @param[out] density fluid density
 * @param[out] dDensity_dPressure fluid density derivative w.r.t. pressure
 * @param[out] dDensity_dTemperature fluid density derivative w.r.t. temperature
 * @param[out] viscosity fluid viscosity
 * @param[out] dViscosity_dPressure fluid viscosity derivative w.r.t. pressure
 * @param[out] dViscosity_dTemperature fluid viscosity derivative w.r.t. temperature
 * @param[out] internalEnergy fluid internal energy
 * @param[out] dInternalEnergy_dPressure fluid internal energy derivative w.r.t. pressure
 * @param[out] dInternalEnergy_dTemperature fluid internal energy derivative w.r.t. temperature
 * @param[out] enthalpy fluid enthalpy
 * @param[out] dEnthalpy_dPressure fluid enthalpy derivative w.r.t. pressure
 * @param[out] dEnthalpy_dTemperature fluid enthalpy derivative w.r.t. temperature
 */
GEOS_HOST_DEVICE
virtual void compute( real64 const pressure,
                      real64 const temperature,
                      real64 & density,
                      real64 & dDensity_dPressure,
                      real64 & dDensity_dTemperature,
                      real64 & viscosity,
                      real64 & dViscosity_dPressure,
                      real64 & dViscosity_dTemperature,
                      real64 & internalEnergy,
                      real64 & dInternalEnergy_dPressure,
                      real64 & dInternalEnergy_dTemperature,
                      real64 & enthalpy,
                      real64 & dEnthalpy_dPressure,
                      real64 & dEnthalpy_dTemperature ) const = 0;

/**
 * @brief Update fluid state at a single point.
 * @param[in] k        element index
 * @param[in] q        gauss point index
 * @param[in] pressure the target pressure value
 */
GEOS_HOST_DEVICE
virtual void update( localIndex const k,
                     localIndex const q,
                     real64 const pressure ) const = 0;

/**
 * @brief Update fluid state at a single point.
 * @param[in] k           element index
 * @param[in] q           gauss point index
 * @param[in] pressure    the target pressure value
 * @param[in] temperature the target temperature value
 */
GEOS_HOST_DEVICE
virtual void update( localIndex const k,
```

(continues on next page)

(continued from previous page)

```
                        localIndex const q,
                        real64 const pressure,
                        real64 const temperature ) const = 0;

};
```

Compound models

Compound constitutive models are employed to mimic the behavior of a material that requires a combination of constitutive models linked together. These compound models do not hold any data. They serve only as an interface with the individual models that they couple.

Coupled Solids

`CoupledSolid` models are employed to represent porous materials that require both a mechanical behavior and constitutive laws that describe the dependency of porosity and permeability on the primary unknowns.

The base class `CoupledSolidBase` implements some basic behaviors and is used to access a generic `CoupledSolid` in a physics solver:

```
  CoupledSolidBase const & porousSolid =
    getConstitutiveModel< CoupledSolidBase >( subRegion, subRegion.template getReference
↪< string >( viewKeyStruct::solidNamesString() ) );
```

Additionally, a *template class* defines a base `CoupledSolid` model templated on the types of solid, porosity, and permeability models:

```
template< typename SOLID_TYPE,
          typename PORO_TYPE,
          typename PERM_TYPE >
class CoupledSolid : public CoupledSolidBase
```

While physics solvers that need a porous material only interface with a compound model, this one has access to the standalone models needed:

```
protected:
  SOLID_TYPE const & getSolidModel() const
  { return this->getParent().template getGroup< SOLID_TYPE >( m_solidModelName ); }

  PORO_TYPE const & getPorosityModel() const
  { return this->getParent().template getGroup< PORO_TYPE >( m_porosityModelName ); }

  PERM_TYPE const & getPermModel() const
  { return this->getParent().template getGroup< PERM_TYPE >( m_permeabilityModelName ); }
```

There are two specializations of a `CoupledSolid`:

- `CompressibleSolid`: this model is used whenever there is no need to define a full mechanical model, but only simple correlations that compute material properties (like porosity or permeability). This model assumes that the solid model is of type *NullModel* and is only templated on the types of porosity and permeability models.
- `PorousSolid`: this model is used to represent a full porous material where the porosity and permeability models need to be aware of the mechanical response of the material.

PVT Package Hierarchy

The architecture of the PVT package is as follows

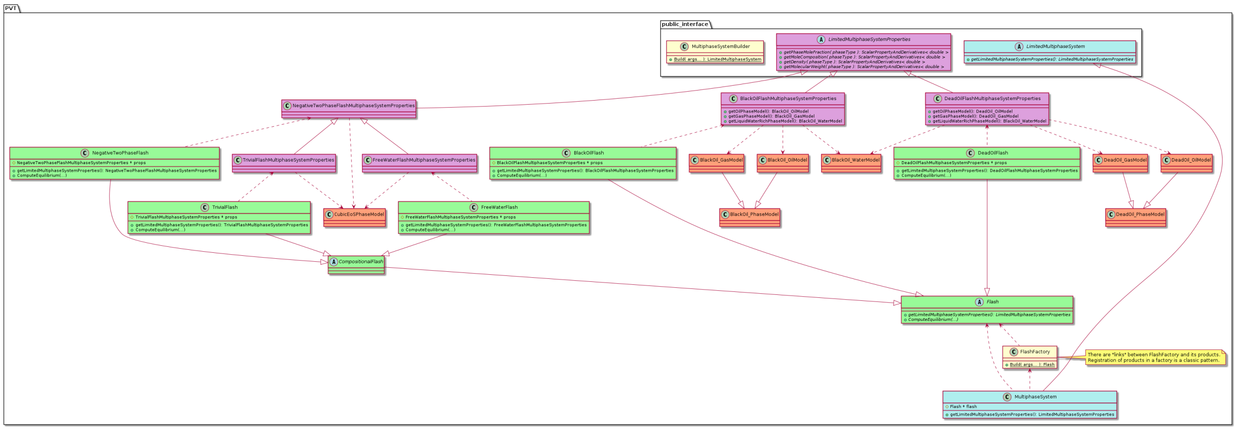

PVT
public_interface
MultiphaseSystemBuilder
+ Build(args...): LimitedMultiphaseSystem
LimitedMultiphaseSystemProperties
+ getPhaseMoleFraction(phaseType): ScalarPropertyAndDerivatives< double >
+ getMoleComposition(phaseType): ScalarPropertyAndDerivatives< double >
+ getDensity(phaseType): ScalarPropertyAndDerivatives< double >
+ getMolecularWeight(phaseType): ScalarPropertyAndDerivatives< double >
LimitedMultiphaseSystem
+ getLimitedMultiphaseSystemProperties(): LimitedMultiphaseSystemProperties
NegativeTwoPhaseFlashMultiphaseSystemProperties
BlackOilFlashMultiphaseSystemProperties
+ getOilPhaseModel(): BlackOil_OilModel
+ getGasPhaseModel(): BlackOil_GasModel
+ getLiquidWaterRichPhaseModel(): BlackOil_WaterModel
DeadOilFlashMultiphaseSystemProperties
+ getOilPhaseModel(): DeadOil_OilModel
+ getGasPhaseModel(): DeadOil_GasModel
+ getLiquidWaterRichPhaseModel(): BlackOil_WaterModel
NegativeTwoPhaseFlash
# NegativeTwoPhaseFlashMultiphaseSystemProperties * props
+ getLimitedMultiphaseSystemProperties(): NegativeTwoPhaseFlashMultiphaseSystemProperties
+ ComputeEquilibrium(...)
TrivialFlashMultiphaseSystemProperties
FreeWaterFlashMultiphaseSystemProperties
BlackOilFlash
# BlackOilFlashMultiphaseSystemProperties * props
+ getLimitedMultiphaseSystemProperties(): BlackOilFlashMultiphaseSystemProperties
+ ComputeEquilibrium(...)
BlackOil_GasModel
BlackOil_OilModel
BlackOil_WaterModel
DeadOilFlash
# DeadOilFlashMultiphaseSystemProperties * props
+ getLimitedMultiphaseSystemProperties(): DeadOilFlashMultiphaseSystemProperties
+ ComputeEquilibrium(...)
DeadOil_GasModel
DeadOil_OilModel
TrivialFlash
# TrivialFlashMultiphaseSystemProperties * props
+ getLimitedMultiphaseSystemProperties(): TrivialFlashMultiphaseSystemProperties
+ ComputeEquilibrium(...)
CubicEoSPhaseModel
FreeWaterFlash
# FreeWaterFlashMultiphaseSystemProperties * props
+ getLimitedMultiphaseSystemProperties(): FreeWaterFlashMultiphaseSystemProperties
+ ComputeEquilibrium(...)
BlackOil_PhaseModel
DeadOil_PhaseModel
CompositionalFlash
Flash
+ getLimitedMultiphaseSystemProperties(): LimitedMultiphaseSystemProperties
+ ComputeEquilibrium(...)
FlashFactory
+ Build(args...): Flash
There are "links" between FlashFactory and its products.
Registration of products in a factory is a classic pattern.
MultiphaseSystem
# Flash * flash
+ getLimitedMultiphaseSystemProperties(): LimitedMultiphaseSystemProperties

Fig. 1.91: Architecture of the PVT package

The color scheme is:

- Green is for computational flash classes
- Purple is for data classes
- Orange is for fluid models (black oil, free water…)
- Light blue are for computational system (algorithms and data combined)

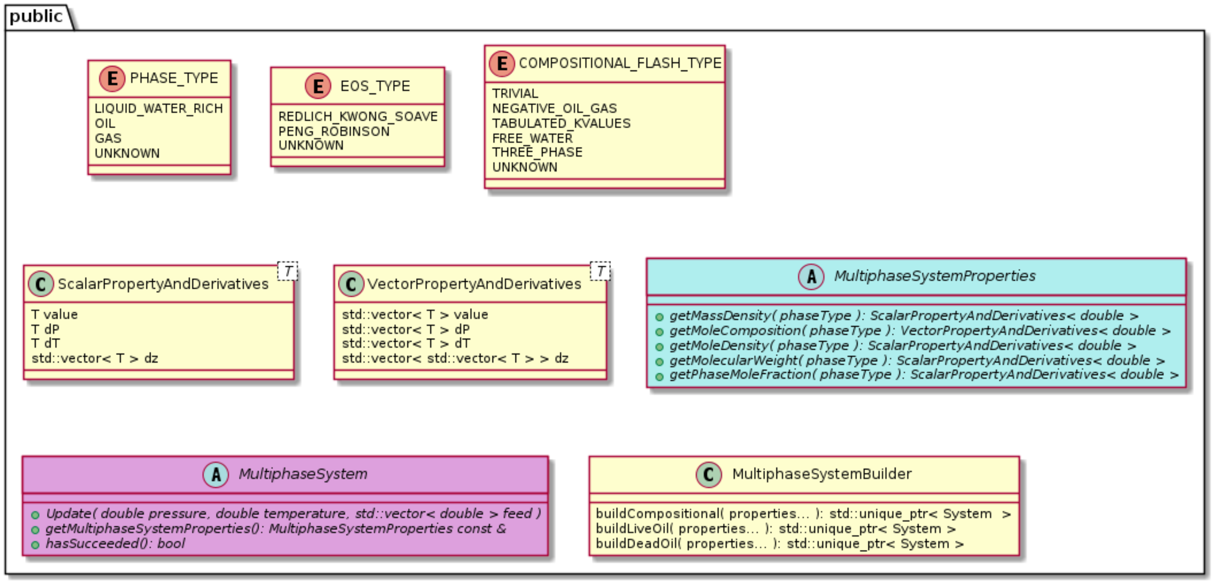

public
PHASE_TYPE
LIQUID_WATER_RICH
OIL
GAS
UNKNOWN
EOS_TYPE
REDLICH_KWONG_SOAVE
PENG_ROBINSON
UNKNOWN
COMPOSITIONAL_FLASH_TYPE
TRIVIAL
NEGATIVE_OIL_GAS
TABULATED_KVALUES
FREE_WATER
THREE_PHASE
UNKNOWN
ScalarPropertyAndDerivatives
T
T value
T dP
T dT
std::vector< T > dz
VectorPropertyAndDerivatives
T
std::vector< T > value
std::vector< T > dP
std::vector< T > dT
std::vector< std::vector< T > > dz
MultiphaseSystemProperties
getMassDensity(phaseType): ScalarPropertyAndDerivatives< double >
getMoleComposition(phaseType): VectorPropertyAndDerivatives< double >
getMoleDensity(phaseType): ScalarPropertyAndDerivatives< double >
getMolecularWeight(phaseType): ScalarPropertyAndDerivatives< double >
getPhaseMoleFraction(phaseType): ScalarPropertyAndDerivatives< double >
MultiphaseSystem
Update(double pressure, double temperature, std::vector< double > feed)
getMultiphaseSystemProperties(): MultiphaseSystemProperties const &
hasSucceeded(): bool
MultiphaseSystemBuilder
buildCompositional(properties...): std::unique_ptr< System >
buildLiveOil(properties...): std::unique_ptr< System >
buildDeadOil(properties...): std::unique_ptr< System >

Fig. 1.92: Public interface of the PVT package

Only classes exposed in the public interface of the PVT package is meant to be used outside the PVT package.

- `PHASE_TYPE`, `EOS_TYPE` and `COMPOSITIONAL_FLASH_TYPE` `enums` are meant to be used to select the models one wants to use.

- `ScalarPropertyAndDerivatives` and `VectorPropertyAndDerivatives` are utility classes used to return the results. Those classes should eventually be replaced by `LvArray`.
- `MultiphaseSystemProperties` agglomerates the result of the computation.
- `MultiphaseSystem` is responsible for performing the computation and serving the results.
- `MultiphaseSystemBuilder` builds the system.

Adding a new Physics Solver

In this tutorial, you will learn how to construct a new GEOS Physics Solver class. We will use *LaplaceFEM* solver, computing the solution of the Laplace problem in a specified material, as a starting point.

$$\begin{aligned} D^{*}\Delta X = f \quad & \text{in}\ \Omega \\ X = X^{g} \quad & \text{on}\ \Gamma_{g} \end{aligned}$$

It is advised to read *XML Input* preliminary to this tutorial. The goal of this document is to explain how to develop a new solver that solves Laplace's equation with a constant diffusion coefficient that is specified by users in the XML input.

For readability, member functions in the text will be referenced by their names but their arguments will be omitted.

LaplaceFEM overview

The *LaplaceFEM* solver can be found in `./src/coreComponents/physicsSolvers/simplePDE/`. Let us inspect declarations in `LaplaceFEM.hpp` and implementations in `LaplaceFEM.cpp` before diving into specifying a new solver class that meets our needs.

Declaration file (reference)

The included header is `physicsSolvers/simplePDE/LaplaceBaseH1.hpp` which declares the base class `LaplaceBaseH1`, shared by all Laplace solvers. Moreover, `physicsSolver/simplePDE/LaplaceBaseH1.hpp` includes the following headers:

- `common/EnumStrings.hpp` which includes facilities for enum-string conversion (useful for reading enum values from input);
- `physicsSolver/SolverBase.hpp` which declares the abstraction class shared by all physics solvers.
- `managers/FieldSpecification/FieldSpecificationManager.hpp` which declares a manager used to access and to set field on the discretized domain.

Let us jump forward to the class enum and variable as they contain the data used specifically in the implementation of *LaplaceFEM*.

class enums and variables (reference)

The class exhibits two member variables:

- m_fieldName which stores the name of the diffused variable (*e.g.* the temperature) as a *string*;
- m_timeIntegrationOption an *enum* value allowing to dispatch with respect to the transient treatment.

TimeIntegrationOption is an *enum* specifying the transient treatment which can be chosen respectively between *SteadyState* and *ImplicitTransient* depending on whether we are interested in the transient state.

```
  enum class TimeIntegrationOption : integer
  {
    SteadyState,
    ImplicitTransient
  };
```

In order to register an enumeration type with the Data Repository and have its value read from input, we must define stream insertion/extraction operators. This is a common task, so GEOS provides a facility for automating it. Upon including common/EnumStrings.hpp, we can call the following macro at the namespace scope (in this case, right after the LaplaceBaseH1 class definition is complete):

```
ENUM_STRINGS( LaplaceBaseH1::TimeIntegrationOption,
              "SteadyState",
              "ImplicitTransient" );
```

Once explained the main variables and enum, let us start reading through the different member functions:

```
class LaplaceFEM : public LaplaceBaseH1
{
public:
  /// The default nullary constructor is disabled to avoid compiler auto-generation:
  LaplaceFEM() = delete;

  /// The constructor needs a user-defined "name" and a parent Group (to place this␣
↪instance in the
  /// tree structure of classes)
  LaplaceFEM( const string & name,
              Group * const parent );

  /// Destructor
  virtual ~LaplaceFEM() override;

  /// "CatalogName()" return the string used as XML tag in the input file.  It ties the␣
↪XML tag with
  /// this C++ classes. This is important.
  static string catalogName() { return "LaplaceFEM"; }
```

Start looking at the class *LaplaceFEM* constructor and destructor declarations shows the usual *string* name and *Group** pointer to parent that are required to build the global file-system like structure of GEOS (see *Group : the base class of GEOS* for details). It can also be noted that the nullary constructor is deleted on purpose to avoid compiler automatic generation and user misuse.

The next method catalogName() is static and returns the key to be added to the *Catalog* for this type of solver (see *A few words about the ObjectCatalog* for details). It has to be paired with the following macro in the implementation file.

```
REGISTER_CATALOG_ENTRY( SolverBase, LaplaceFEM, string const &, Group * const )
```

Finally, the member function `registerDataOnMesh()` is declared in the `LaplaceBaseH1` class as

```
  /// This method ties properties with their supporting mesh
  virtual void registerDataOnMesh( Group & meshBodies ) override final;
```

It is used to assign fields onto the discretized mesh object and will be further discussed in the *Implementation File (reference)* section.

The next block consists in solver interface functions. These member functions set up and specialize every time step from the system matrix assembly to the solver stage.

```
  virtual void
  setupSystem( DomainPartition & domain,
               DofManager & dofManager,
               CRSMatrix< real64, globalIndex > & localMatrix,
               ParallelVector & rhs,
               ParallelVector & solution,
               bool const setSparsity = false ) override;

  virtual void
  assembleSystem( real64 const time,
                  real64 const dt,
                  DomainPartition & domain,
                  DofManager const & dofManager,
                  CRSMatrixView< real64, globalIndex const > const & localMatrix,
                  arrayView1d< real64 > const & localRhs ) override;
```

Furthermore, the following functions are inherited from the base class.

```
  virtual real64 solverStep( real64 const & time_n,
                             real64 const & dt,
                             integer const cycleNumber,
                             DomainPartition & domain ) override;

  virtual void
  implicitStepSetup( real64 const & time_n,
                     real64 const & dt,
                     DomainPartition & domain ) override;

  virtual void
  setupDofs( DomainPartition const & domain,
             DofManager & dofManager ) const override;

  virtual void
  applyBoundaryConditions( real64 const time,
                           real64 const dt,
                           DomainPartition & domain,
                           DofManager const & dofManager,
                           CRSMatrixView< real64, globalIndex const > const &␣
↪localMatrix,
                           arrayView1d< real64 > const & localRhs ) override;

```

(continues on next page)

(continued from previous page)

```
virtual void
applySystemSolution( DofManager const & dofManager,
                     arrayView1d< real64 const > const & localSolution,
                     real64 const scalingFactor,
                     real64 const dt,
                     DomainPartition & domain ) override;

virtual void updateState( DomainPartition & domain ) override final;

virtual void
  resetStateToBeginningOfStep( DomainPartition & GEOS_UNUSED_PARAM( domain ) )␣
↪override;

virtual void
implicitStepComplete( real64 const & time,
                      real64 const & dt,
                      DomainPartition & domain ) override;

/// This method is specific to this Laplace solver.
/// It is used to apply Dirichlet boundary condition
/// and called when the base class applyBoundaryConditions() is called.
virtual void applyDirichletBCImplicit( real64 const time,
                                       DofManager const & dofManager,
                                       DomainPartition & domain,
                                       CRSMatrixView< real64, globalIndex const >␣
↪const & localMatrix,
                                       arrayView1d< real64 > const & localRhs );
```

Eventually, `applyDirichletBCImplicit()` is the working specialized member functions called when `applyBoundaryConditions()` is called in this particular class override.

Browsing the base class `SolverBase`, it can be noted that most of the solver interface functions are called during either `SolverBase::linearImplicitStep()` or `SolverBase::nonlinearImplicitStep()` depending on the solver strategy chosen.

Switching to protected members, `postProcessInput()` is a central member function and will be called by `Group` object after input is read from XML entry file. It will set and dispatch solver variables from the base class `SolverBase` to the most derived class. For `LaplaceFEM`, it will allow us to set the right time integration scheme based on the XML value as will be further explored in the next *Implementation File (reference)* section.

Let us focus on a `struct` that plays an important role: the *viewKeyStruct* structure.

viewKeyStruct structure (reference)

This embedded instantiated structure is a common pattern shared by all solvers. It stores `dataRepository::ViewKey` type objects that are used as binding data between the input XML file and the source code.

```
struct viewKeyStruct : public SolverBase::viewKeyStruct
{
  static constexpr char const * timeIntegrationOption() { return "timeIntegrationOption
↪"; }
  static constexpr char const * fieldVarName() { return "fieldName"; }
};
```

We can check that in the *LaplaceFEM* companion integratedTest

```
    <LaplaceFEM
      name="laplace"
      discretization="FE1"
      timeIntegrationOption="SteadyState"
      fieldName="Temperature"
      targetRegions="{ Domain }">
      <LinearSolverParameters
        directParallel="0"/>
    </LaplaceFEM>
```

In the following section, we will see where this binding takes place.

Implementation File (reference)

Switching to implementation, we will focus on few implementations, leaving details to other tutorials. The `LaplaceFEM` constructor is implemented as follows.

```
LaplaceFEM::LaplaceFEM( const string & name,
                        Group * const parent ):
  LaplaceBaseH1( name, parent )
{}
```

As we see, it calls the `LaplaceBaseH1` constructor, that is implemented as follows.

```
LaplaceBaseH1::LaplaceBaseH1( const string & name,
                              Group * const parent ):
  SolverBase( name, parent ),
  m_fieldName( "primaryField" ),
  m_timeIntegrationOption( TimeIntegrationOption::ImplicitTransient )
{
  this->registerWrapper( viewKeyStruct::timeIntegrationOption(), &m_
↪timeIntegrationOption ).
    setInputFlag( InputFlags::REQUIRED ).
    setDescription( "Time integration method. Options are:\n* " + EnumStrings<␣
↪TimeIntegrationOption >::concat( "\n* " ) );

  this->registerWrapper( viewKeyStruct::fieldVarName(), &m_fieldName ).
    setInputFlag( InputFlags::REQUIRED ).
    setDescription( "Name of field variable" );

}
```

Checking out the constructor, we can see that the use of a `registerWrapper<T>(...)` allows us to register the key value from the *enum* `viewKeyStruct` defining them as:

- `InputFlags::OPTIONAL` if they are optional and can be provided;
- `InputFlags::REQUIRED` if they are required and will throw error if not;

and their associated descriptions for auto-generated docs.

```
void LaplaceBaseH1::registerDataOnMesh( Group & meshBodies )
{
```

(continues on next page)

(continued from previous page)

```
  meshBodies.forSubGroups< MeshBody >( [&] ( MeshBody & meshBody )
  {
    NodeManager & nodes = meshBody.getBaseDiscretization().getNodeManager();

    nodes.registerWrapper< real64_array >( m_fieldName ).
      setApplyDefaultValue( 0.0 ).
      setPlotLevel( PlotLevel::LEVEL_0 ).
      setDescription( "Primary field variable" );
  } );
}
```

`registerDataOnMesh()` is browsing all subgroups in the mesh `Group` object and for all nodes in the sub group:

- register the observed field under the chosen `m_fieldName` key;
- apply a default value;
- set the output verbosity level (here `PlotLevel::LEVEL_0`);
- set the field associated description for auto generated docs.

```
void LaplaceFEM::assembleSystem( real64 const GEOS_UNUSED_PARAM( time_n ),
                                 real64 const dt,
                                 DomainPartition & domain,
                                 DofManager const & dofManager,
                                 CRSMatrixView< real64, globalIndex const > const &␣
↪localMatrix,
                                 arrayView1d< real64 > const & localRhs )
{
  forDiscretizationOnMeshTargets( domain.getMeshBodies(), [&] ( string const &,
                                                                MeshLevel & mesh,
                                                                arrayView1d< string␣
↪const > const & regionNames )
  {
    NodeManager & nodeManager = mesh.getNodeManager();
    string const dofKey = dofManager.getKey( m_fieldName );
    arrayView1d< globalIndex const > const &
    dofIndex =  nodeManager.getReference< array1d< globalIndex > >( dofKey );

    LaplaceFEMKernelFactory kernelFactory( dofIndex, dofManager.rankOffset(),␣
↪localMatrix, localRhs, dt, m_fieldName );

    string const dummyString = "dummy";
    finiteElement::
      regionBasedKernelApplication< parallelDevicePolicy< >,
                                    constitutive::NullModel,
                                    CellElementSubRegion >( mesh,
                                                            regionNames,
                                                            this->
↪getDiscretizationName(),
                                                            dummyString,
                                                            kernelFactory );

  } );
```

(continues on next page)

(continued from previous page)

```
}
```

assembleSystem() will be our core focus as we want to change the diffusion coefficient from its hard coded value to a XML read user-defined value. One can see that this method is in charge of constructing in a parallel fashion the FEM system matrix. Bringing nodeManager and ElementRegionManager from domain local MeshLevel object together with FiniteElementDiscretizationManager from the NumericalMethodManager, it uses nodes embedded loops on degrees of freedom in a local index embedded loops to fill a matrix and a rhs container.

As we spotted the place to change in a code to get a user-defined diffusion coefficient into the game, let us jump to writing our new *LaplaceDiffFEM* solver.

Note: We might want to remove final keyword from postProcessInput() as it will prevent you from overriding it.

Start doing your own Physic solver

As we will extend *LaplaceFEM* capabilities, we will derive publicly from it.

Declaration File

As there is only few places where we have to change, the whole declaration file is reported below and commented afterwards.

```
#include "physicsSolvers/simplePDE/LaplaceFEM.hpp"

namespace geos
{

class LaplaceDiffFEM : public LaplaceFEM
{
public:

  LaplaceDiffFEM() = delete;

  LaplaceDiffFEM( const string& name,
                  Group * const parent );

  virtual ~LaplaceDiffFEM() override;

  static string catalogName() { return "LaplaceDiffFEM"; }

  virtual void
  assembleSystem( real64 const time,
                  real64 const dt,
                  DomainPartition * const domain,
                  DofManager const & dofManager,
                  ParallelMatrix & matrix,
                  ParallelVector & rhs ) override;
```

(continues on next page)

(continued from previous page)

```
  struct viewKeyStruct : public LaplaceFEM::viewKeyStruct
  {
    dataRepository::ViewKey diffusionCoeff = { "diffusionCoeff" };
  } laplaceDiffFEMViewKeys;

  protected:
  virtual void postProcessInput() override final;

private:
  real64 m_diffusion;

};
```

We intend to have a user-defined diffusion coefficient, we then need a *real64* class variable `m_diffusion` to store it.

Consistently with *LaplaceFEM*, we will also delete the nullary constructor and declare a constructor with the same arguments for forwarding to *Group* master class. Another mandatory step is to override the static `CatalogName()` method to properly register any data from the new solver class.

Then as mentioned in *Implementation File (reference)*, the diffusion coefficient is used when assembling the matrix coefficient. Hence we will have to override the `assembleSystem()` function as detailed below.

Moreover, if we want to introduce a new binding between the input XML and the code we will have to work on the three `struct viewKeyStruct` , `postProcessInput()` and the constructor.

Our new solver `viewKeyStruct` will have its own structure inheriting from the *LaplaceFEM* one to have the `timeIntegrationOption` and `fieldName` field. It will also create a `diffusionCoeff` field to be bound to the user defined homogeneous coefficient on one hand and to our `m_diffusion` class variable on the other.

Implementation File

As we have seen in *Implementation File (reference)*, the first place where to implement a new register from XML input is in the constructor. The `diffusionCoeff` entry we have defined in the `laplaceDiffFEMViewKeys` will then be asked as a required input. If not provided, the error thrown will ask for it described asked an “input uniform diffusion coefficient for the Laplace equation”.

```
LaplaceDiffFEM::LaplaceDiffFEM( const string& name,
                                Group * const parent ):
LaplaceFEM( name, parent ), m_diffusion(0.0)
{
  registerWrapper<string>(laplaceDiffFEMViewKeys.diffusionCoeff.Key()).
    setInputFlag(InputFlags::REQUIRED).
    setDescription("input uniform diffusion coeff for the laplace equation");
}
```

Another important spot for binding the value of the XML read parameter to our `m_diffusion` is in `postProcessInput()`.

```
void LaplaceDiffFEM::postProcessInput()
{
  LaplaceFEM::postProcessInput();
```

(continues on next page)

(continued from previous page)

```
  string sDiffCoeff = this->getReference<string>(laplaceDiffFEMViewKeys.diffusionCoeff);
  this->m_diffusion = std::stof(sDiffCoeff);
}
```

Now that we have required, read and bind the user-defined diffusion value to a variable, we can use it in the construction of our matrix into the overridden `assembleSystem()`.

```
// begin element loop, skipping ghost elements
for( localIndex k=0 ; k<elementSubRegion->size() ; ++k )
{
  if(elemGhostRank[k] < 0)
  {
    element_rhs = 0.0;
    element_matrix = 0.0;
    for( localIndex q=0 ; q<n_q_points ; ++q)
    {
      for( localIndex a=0 ; a<numNodesPerElement ; ++a)
      {
        elemDofIndex[a] = dofIndex[ elemNodes( k, a ) ];

        for( localIndex b=0 ; b<numNodesPerElement ; ++b)
        {
          element_matrix(a,b) += detJ[k][q] *
                                 m_diffusion *
                               + Dot( dNdX[k][q][a], dNdX[k][q][b] );
        }

      }
    }
    matrix.add( elemDofIndex, elemDofIndex, element_matrix );
    rhs.add( elemDofIndex, element_rhs );
  }
}
```

This completes the implementation of our new solver *LaplaceDiffFEM*.

Nonetheless, the compiler should complain that `m_fieldName` is privately as inherited from *LaplaceFEM*. One should then either promote `m_fieldName` to protected or add a getter in *LaplaceFEM* class to correct the error. The getter option has been chosen and the fix in our solver is then:

```
array1d<globalIndex> const & dofIndex =
  nodeManager->getReference< array1d<globalIndex> >( dofManager.getKey( getFieldName() )␣
↪);
```

Note: For consistency do not forget to change LaplaceFEM to LaplaceDiffFEM in the guards comments

Last steps

After assembling both declarations and implementations for our new solver, the final steps go as:

- add declarations to parent CMakeLists.txt (here add to `physicsSolvers_headers`);
- add implementations to parent CMakeLists.txt (here add to `physicsSolvers_sources`);
- check that Doxygen comments are properly set in our solver class;
- uncrustify it to match the code style;
- write unit tests for each new features in the solver class;
- write an integratedTests for the solver class.

1.7 Doxygen

The c++ source in GEOS is annotated using doxygen. Our doxygen pages are linked below.

Developers may find it helpful to review the key code components described in the Developer Guide before diving into the doxygen.

1.8 Python Tools

1.8.1 Python Tools Setup

The preferred method to setup the GEOSX python tools is to run the following command in the build directory:

```
make geosx_python_tools
```

This will attempt to install the required packages into the python distribution indicated via the *Python3_EXECUTABLE* cmake variable (also used by pygeosx).

If the user does not have write access for the target python distribution, the installation will attempt to create a new virtual python environment (Note: this requires that the virtualenv package be installed). If any package dependencies are missing, then the install script will attempt to fetch them from the internet using pip. After installation, these packages will be available for import within the associated python distribution, and a set of console scripts will be available within the GEOSX build bin directory.

Alternatively, these packages can be installed manually into a python environment using pip:

```
cd GEOSX/src/coreComponents/python/modules/geosx_mesh_tools_package
pip install --upgrade .

cd ../geosx_xml_tools_package
pip install --upgrade .

# Etc.
```

1.8.2 Packages

HDF5 Wrapper

The *hdf5_wrapper* python package adds a wrapper to *h5py* that greatly simplifies reading/writing to/from hdf5-format files.

Usage

Once loaded, the contents of a file can be navigated in the same way as a native python dictionary.

```
import hdf5_wrapper

data = hdf5_wrapper.hdf5_wrapper('data.hdf5')

test = data['test']
for k, v in data.items():
  print('key: %s, value: %s' % (k, str(v)))
```

If the user indicates that a file should be opened in write-mode (*w*) or read/write-mode (*a*), then the file can be created or modified. Note: for these changes to be written to the disk, the wrapper may need to be closed or deleted.

```
import hdf5_wrapper
import numpy as np

data = hdf5_wrapper.hdf5_wrapper('data.hdf5', mode='w')
data['string'] = 'string'
data['integer'] = 123
data['array'] = np.random.randn(3, 4, 5)
data['child'] = {'float': 1.234}
```

Existing dictionaries can be placed on the current level:

```
existing_dict = {'some': 'value'}
data.insert(existing_dict)
```

And external hdf5 format files can be linked together:

```
for k in ['child_a', 'child_b']:
  data.link(k, '%s.hdf5' % (k))
```

API

class hdf5_wrapper.wrapper.**hdf5_wrapper**(*fname: str = '', target: File | None = None, mode: str = 'r'*)

A class for reading/writing hdf5 files, which behaves similar to a native dict

close() → None

Closes the database

copy() → Dict[str, Any]

Copy the entire database into memory

Returns
a dictionary holding the database contents

Return type
dict

`get_copy()` → Dict[str, Any]

Copy the entire database into memory

Returns
a dictionary holding the database contents

Return type
dict

`insert`(*x: Dict[str, Any]* | hdf5_wrapper) → None

Insert the contents of the target object to the current location

Parameters
x (*dict, hdf5_wrapper*) – the dictionary to insert

`keys()` → Iterable[str]

Get a list of groups and arrays located at the current level

Returns
a list of key names pointing to objects at the current level

Return type
list

`link`(*k: str*, *target: str*) → None

Link an external hdf5 file to this location in the database

Parameters

- **k** (*str*) – the name of the new link in the database
- **target** (*str*) – the path to the external database

`values()` → Iterable[*hdf5_wrapper* | Any]

Get a list of values located on the current level

GEOS Mesh Tools

The *geosx_mesh_tools* python package includes tools for converting meshes from common formats (abaqus, etc.) to those that can be read by GEOS (gmsh, vtk). See *Python Tools Setup* for details on setup instructions, and *Using an External Mesh* for a detailed description of how to use external meshes in GEOS. The available console scripts for this package and its API are described below.

convert_abaqus

Compile an xml file with advanced features into a single file that can be read by GEOS.

```
usage: convert_abaqus [-h] [-v] input output
```

Positional Arguments

input Input abaqus mesh file name

output Output gmsh/vtu mesh file name

Named Arguments

-v, --verbose Increase verbosity level

Default: False

Note: For vtk format meshes, the user also needs to determine the region ID numbers and names of nodesets to import into GEOS. The following shows how these could look in an input XML file for a mesh with three regions (*REGIONA*, *REGIONB*, and *REGIONC*) and six nodesets (*xneg*, *xpos*, *yneg*, *ypos*, *zneg*, and *zpos*):

```
<Problem>
  <Mesh>
    <VTKMesh
      name="external_mesh"
      file="mesh.vtu"
      regionAttribute="REGIONA-REGIONB-REGIONC"
      nodesetNames="{ xneg, xpos, yneg, ypos, zneg, zpos }"/>
  </Mesh>

  <ElementRegions>
    <CellElementRegion
      name="ALL"
      cellBlocks="{ 0_tetrahedra, 1_tetrahedra, 2_tetrahedra }"
      materialList="{ water, porousRock }"
      meshBody="external_mesh"/>
  </ElementRegions>
</Problem>
```

API

geosx_mesh_tools.abaqus_converter.**convert_abaqus_to_gmsh**(*input_mesh: str*, *output_mesh: str*, *logger: Logger | None = None*) → int

Convert an abaqus mesh to gmsh 2 format, preserving nodeset information.

If the code encounters any issues with region/element indices, the conversion will attempt to continue, with errors indicated by -1 values in the output file.

Parameters

- **input_mesh** (*str*) – path of the input abaqus file
- **output_mesh** (*str*) – path of the output gmsh file
- **logger** (*logging.Logger*) – an instance of logging.Logger

Returns
Number of potential warnings encountered during conversion

Return type
int

geosx_mesh_tools.abaqus_converter.**convert_abaqus_to_vtu**(*input_mesh: str*, *output_mesh: str*, *logger: Logger | None = None*) → int

Convert an abaqus mesh to vtu format, preserving nodeset information.

If the code encounters any issues with region/element indices, the conversion will attempt to continue, with errors indicated by -1 values in the output file.

Parameters

- **input_mesh** (*str*) – path of the input abaqus file
- **output_mesh** (*str*) – path of the output vtu file
- **logger** (*logging.Logger*) – a logger instance

Returns
Number of potential warnings encountered during conversion

Return type
int

GEOS XML Tools

The *geosx_xml_tools* python package adds a set of advanced features to the GEOS xml format: units, parameters, and symbolic expressions. See *Python Tools Setup* for details on setup instructions, and *Advanced XML Features* for a detailed description of the input format. The available console scripts for this package and its API are described below.

convert_abaqus

Convert an abaqus format mesh file to gmsh or vtk format.

```
usage: preprocess_xml [-h] [-i INPUT] [-c COMPILED_NAME] [-s SCHEMA]
                      [-v VERBOSE] [-p PARAMETERS [PARAMETERS ...]]
```

Named Arguments

-i, --input Input file name (multiple allowed)

-c, --compiled-name Compiled xml file name (otherwise, it is randomly genrated)

Default: “”

-s, --schema GEOSX schema to use for validation

Default: “”

-v, --verbose Verbosity of outputs

Default: 0

-p, --parameters Parameter overrides (name value, multiple allowed)

Default: []

format_xml

Formats an xml file.

```
usage: format_xml [-h] [-i INDENT] [-s STYLE] [-d DEPTH] [-a ALPHEBITIZE]
                  [-c CLOSE] [-n NAMESPACE]
                  input
```

Positional Arguments

input Input file name

Named Arguments

-i, --indent Indent size

Default: 2

-s, --style Indent style

Default: 0

-d, --depth Block separation depth

Default: 2

-a, --alphebitize Alphebetize attributes

Default: 0

-c, --close Close tag style

Default: 0

-n, --namespace Include namespace

Default: 0

check_xml_attribute_coverage

Checks xml attribute coverage for files in the GEOS repository.

```
usage: check_xml_attribute_coverage [-h] [-r ROOT] [-o OUTPUT]
```

Named Arguments

-r, --root GEOSX root

Default: “”

-o, --output Output file name

Default: “attribute_test.xml”

check_xml_redundancy

Checks for redundant attribute definitions in an xml file, such as those that duplicate the default value.

```
usage: check_xml_redundancy [-h] [-r ROOT]
```

Named Arguments

-r, --root GEOSX root

Default: “”

API

Command line tools for geosx_xml_tools

`geosx_xml_tools.main.`**`check_mpi_rank`**`()` → int

Check the MPI rank

Returns
MPI rank

Return type
int

`geosx_xml_tools.main.`**`format_geosx_arguments`**`(`*compiled_name: str*, *unknown_args: Iterable[str]*`)` → Iterable[str]

Format GEOSX arguments

Parameters

- **`compiled_name`** (*`str`*) – Name of the compiled xml file
- **`unknown_args`** (*`list`*) – List of unprocessed arguments

Returns
List of arguments to pass to GEOSX

Return type
list

`geosx_xml_tools.main.`**`preprocess_parallel`**`()` → Iterable[str]

MPI aware xml preprocesing

`geosx_xml_tools.main.`**`preprocess_serial`**`()` → None

Entry point for the geosx_xml_tools console script geosx_xml_tools.main.**wait_for_file_write_rank_0**(*target_file_argument: int | str = 0*, *max_wait_time: float = 100*, *max_startup_delay: float = 1*) → Callable[[Callable[[...], Any]], Callable[[...], Any]]

Constructor for a function decorator that waits for a target file to be written on rank 0

Parameters

- **target_file_argument** (*int, str*) – Index or keyword of the filename argument in the decorated function
- **max_wait_time** (*float*) – Maximum amount of time to wait (seconds)
- **max_startup_delay** (*float*) – Maximum delay allowed for thread startup (seconds)

Returns

Wrapped function

Tools for processing xml files in GEOSX

geosx_xml_tools.xml_processor.**apply_regex_to_node**(*node: lxml.etree.Element*) → None

Apply regexes that handle parameters, units, and symbolic math to each xml attribute in the structure.

Parameters

node (*lxml.etree.Element*) – The target node in the xml structure.

geosx_xml_tools.xml_processor.**generate_random_name**(*prefix: str = ''*, *suffix: str = '.xml'*) → str

If the target name is not specified, generate a random name for the compiled xml

Parameters

- **prefix** (*str*) – The file prefix (default = ‘’).
- **suffix** (*str*) – The file suffix (default = ‘.xml’)

Returns

Random file name

Return type

str

geosx_xml_tools.xml_processor.**merge_included_xml_files**(*root: lxml.etree.Element*, *fname: str*, *includeCount: int*, *maxInclude: int = 100*) → None

Recursively merge included files into the current structure.

Parameters

- **root** (*lxml.etree.Element*) – The root node of the base xml structure.
- **fname** (*str*) – The name of the target xml file to merge.
- **includeCount** (*int*) – The current recursion depth.
- **maxInclude** (*int*) – The maximum number of xml files to include (default = 100)

geosx_xml_tools.xml_processor.**merge_xml_nodes**(*existingNode: lxml.etree.Element*, *targetNode: lxml.etree.Element*, *level: int*) → None

Merge nodes in an included file into the current structure level by level.

Parameters

- **existingNode** (*lxml.etree.Element*) – The current node in the base xml structure.
- **targetNode** (*lxml.etree.Element*) – The node to insert.

- **level** (*int*) – The xml file depth.

geosx_xml_tools.xml_processor.**process**(*inputFiles: Iterable[str]*, *outputFile: str = ''*, *schema: str = ''*, *verbose: int = 0*, *parameter_override: List[Tuple[str, str]] = []*, *keep_parameters: bool = True*, *keep_includes: bool = True*) → str

Process an xml file

Parameters

- **inputFiles** (*list*) – Input file names.
- **outputFile** (*str*) – Output file name (if not specified, then generate randomly).
- **schema** (*str*) – Schema file name to validate the final xml (if not specified, then do not validate).
- **verbose** (*int*) – Verbosity level.
- **parameter_override** (*list*) – Parameter value overrides
- **keep_parameters** (*bool*) – If True, then keep parameters in the compiled file (default = True)
- **keep_includes** (*bool*) – If True, then keep includes in the compiled file (default = True)

Returns
Output file name

Return type
str

geosx_xml_tools.xml_processor.**validate_xml**(*fname: str*, *schema: str*, *verbose: int*) → None

Validate an xml file, and parse the warnings.

Parameters

- **fname** (*str*) – Target xml file name.
- **schema** (*str*) – Schema file name.
- **verbose** (*int*) – Verbosity level.

geosx_xml_tools.xml_formatter.**format_attribute**(*attribute_indent: str*, *ka: str*, *attribute_value: str*) → str

Format xml attribute strings

Parameters

- **attribute_indent** (*str*) – Attribute indent string
- **ka** (*str*) – Attribute name
- **attribute_value** (*str*) – Attribute value

Returns
Formatted attribute value

Return type
str

geosx_xml_tools.xml_formatter.**format_file**(*input_fname: str*, *indent_size: int = 2*, *indent_style: bool = False*, *block_separation_max_depth: int = 2*, *alphebitize_attributes: bool = False*, *close_style: bool = False*, *namespace: bool = False*) → None

Script to format xml files

Parameters

- **input_fname** (*str*) – Input file name
- **indent_size** (*int*) – Indent size
- **indent_style** (*bool*) – Style of indentation (0=fixed, 1=hanging)
- **block_separation_max_depth** (*int*) – Max depth to separate xml blocks
- **alphebitize_attributes** (*bool*) – Alphebitize attributes
- **close_style** (*bool*) – Style of close tag (0=same line, 1=new line)
- **namespace** (*bool*) – Insert this namespace in the xml description

geosx_xml_tools.xml_formatter.**format_xml_level**(*output: TextIO*, *node: lxml.etree.Element*, *level: int*, *indent: str = ' '*, *block_separation_max_depth: int = 2*, *modify_attribute_indent: bool = False*, *sort_attributes: bool = False*, *close_tag_newline: bool = False*, *include_namespace: bool = False*) → None

Iteratively format the xml file

Parameters

- **output** (*file*) – the output text file handle
- **node** (*lxml.etree.Element*) – the current xml element
- **level** (*int*) – the xml depth
- **indent** (*str*) – the xml indent style
- **block_separation_max_depth** (*int*) – the maximum depth to separate adjacent elements
- **modify_attribute_indent** (*bool*) – option to have flexible attribute indentation
- **sort_attributes** (*bool*) – option to sort attributes alphabetically
- **close_tag_newline** (*bool*) – option to place close tag on a separate line
- **include_namespace** (*bool*) – option to include the xml namespace in the output

geosx_xml_tools.xml_formatter.**main**() → None

Script to format xml files

Parameters

- **input** (*str*) – Input file name
- **-i/--indent** (*int*) – Indent size
- **-s/--style** (*int*) – Indent style
- **-d/--depth** (*int*) – Block separation depth
- **-a/--alphebitize** (*int*) – Alphebitize attributes
- **-c/--close** (*int*) – Close tag style
- **-n/--namespace** (*int*) – Include namespace

Tools for managing units in GEOSX

class geosx_xml_tools.unit_manager.**UnitManager**

This class is used to manage unit definitions.

`buildUnits()` → None

Build the unit definitions.

`regexHandler`(*match: Match*) → str

Split the matched string into a scale and unit definition.

Parameters
`match` (*re.match*) – The matching string from the regex.

Returns
The string with evaluated unit definitions

Return type
str

Tools for managing regular expressions in geosx_xml_tools

`class geosx_xml_tools.regex_tools.DictRegexHandler`

This class is used to substitute matched values with those stored in a dict.

`geosx_xml_tools.regex_tools.SymbolicMathRegexHandler`(*match: Match*) → str

Evaluate symbolic expressions that are identified using the regex_tools.patterns['symbolic'].

Parameters
`match` (*re.match*) – A matching string identified by the regex.

`geosx_xml_tools.xml_redundancy_check.check_redundancy_level`(*local_schema: Dict[str, Any]*, *node: lxml.etree.Element*, *whitelist: Iterable[str] = ['component']*) → int

Check xml redundancy at the current level

Parameters

- `local_schema` (*dict*) – Schema definitions
- `node` (*lxml.etree.Element*) – current xml node
- `whitelist` (*list*) – always match nodes containing these attributes

Returns
Number of required attributes in the node and its children

Return type
int

`geosx_xml_tools.xml_redundancy_check.check_xml_redundancy`(*schema: Dict[str, Any]*, *fname: str*) → None

Check redundancy in an xml file

Parameters

- `schema` (*dict*) – Schema definitions
- `fname` (*str*) – Name of the target file

`geosx_xml_tools.xml_redundancy_check.main()` → None

Entry point for the xml attribute usage test script

Parameters
`-r/--root` (*str*) – GEOSX root directory

`geosx_xml_tools.xml_redundancy_check.`**`process_xml_files`**(*geosx_root: str*) → None

Test for xml redundancy

Parameters

`geosx_root` (*`str`*) – GEOSX root directory

`geosx_xml_tools.attribute_coverage.`**`collect_xml_attributes`**(*xml_types: Dict[str, Dict[str, Any]]*, *fname: str*, *folder: str*) → None

Collect xml attribute usage in a file

Parameters

- **`xml_types`** (*`dict`*) – dictionary containing attribute usage
- **`fname`** (*`str`*) – name of the target file
- **`folder`** (*`str`*) – the source folder for the current file

`geosx_xml_tools.attribute_coverage.`**`collect_xml_attributes_level`**(*local_types: Dict[str, Dict[str, Any]]*, *node: lxml.etree.Element*, *folder: str*) → None

Collect xml attribute usage at the current level

Parameters

- **`local_types`** (*`dict`*) – dictionary containing attribute usage
- **`node`** (*`lxml.etree.Element`*) – current xml node
- **`folder`** (*`str`*) – the source folder for the current file

`geosx_xml_tools.attribute_coverage.`**`main`**() → None

Entry point for the xml attribute usage test script

Parameters

- **`-r/--root`** (*`str`*) – GEOSX root directory
- **`-o/--output`** (*`str`*) – output file name

`geosx_xml_tools.attribute_coverage.`**`parse_schema`**(*fname: str*) → Dict[str, Dict[str, Any]]

Parse the schema file into the xml attribute usage dict

Parameters

`fname` (*`str`*) – schema name

Returns

Dictionary of attributes and children for the entire schema

Return type

dict

`geosx_xml_tools.attribute_coverage.`**`parse_schema_element`**(*root: lxml.etree.Element*, *node: lxml.etree.Element*, *xsd: str = '{http://www.w3.org/2001/XMLSchema}'*, *recursive_types: Iterable[str] = ['PeriodicEvent', 'SoloEvent', 'HaltEvent']*, *folders: Iterable[str] = ['src', 'examples']*) → Dict[str, Dict[str, Any]]

Parse the xml schema at the current level

Parameters

- **root** (*lxml.etree.Element*) – the root schema node
- **node** (*lxml.etree.Element*) – current schema node
- **xsd** (*str*) – the file namespace
- **recursive_types** (*list*) – node tags that allow recursive nesting
- **folders** (*list*) – folders to sort xml attribute usage into

Returns
Dictionary of attributes and children for the current node

Return type
dict

geosx_xml_tools.attribute_coverage.**process_xml_files**(*geosx_root: str*, *output_name: str*) → None

Test for xml attribute usage

Parameters

- **geosx_root** (*str*) – GEOSX root directory
- **output_name** (*str*) – output file name

geosx_xml_tools.attribute_coverage.**write_attribute_usage_xml**(*xml_types: Dict[str, Dict[str, Any]]*, *fname: str*) → None

Write xml attribute usage file

Parameters

- **xml_types** (*dict*) – dictionary containing attribute usage by xml type
- **fname** (*str*) – output file name

geosx_xml_tools.attribute_coverage.**write_attribute_usage_xml_level**(*local_types: Dict[str, Dict[str, Any]]*, *node: lxml.etree.Element*, *folders: Iterable[str] = ['src', 'examples']*) → None

Write xml attribute usage file at a given level

Parameters

- **local_types** (*dict*) – dict containing attribute usage at the current level
- **node** (*lxml.etree.Element*) – current xml node

Tools for reading/writing GEOSX ascii tables

geosx_xml_tools.table_generator.**read_GEOS_table**(*axes_files: Iterable[str]*, *property_files: Iterable[str]*) → Tuple[Iterable[ndarray], Dict[str, ndarray]]

Read an GEOS-compatible ascii table.

Parameters

- **axes_files** (*list*) – List of the axes file names in order.
- **property_files** (*list*) – List of property file names

Returns
List of axis definitions, dict of property values

Return type
tuple

`geosx_xml_tools.table_generator.`**`write_GEOS_table`**`(`*axes_values: Iterable[ndarray]*, *properties: Dict[str, ndarray]*, *axes_names: Iterable[str] = ['x', 'y', 'z', 't']*, *string_format: str = '%1.5e'*`)` → None

Write an GEOS-compatible ascii table.

Parameters

- **`axes_values`** (*`list`*) – List of arrays containing the coordinates for each axis of the table.
- **`properties`** (*`dict`*) – Dict of arrays with dimensionality/size defined by the axes_values
- **`axes_names`** (*`list`*) – Names for each axis (default = ['x', 'y', 'z', 't'])
- **`string_format`** (*`str`*) – Format for output values (default = %1.5e)

`geosx_xml_tools.table_generator.`**`write_read_GEOS_table_example`**`()` → None

Table read / write example.

PyGEOSX Tools

The *pygeosx_tools* python package adds a variety of tools for working with pygeosx objects. These include common operations such as setting the value of geosx wrappers with python functions, parallel communication, and file IO. Examples using these tools can be found here: *pygeosx Examples*.

API

`pygeosx_tools.wrapper.`**`allgather_wrapper`**`(`*problem*, *key*, *ghost_key=''*`)`

Get a global copy of a wrapper as a numpy ndarray on all ranks

Parameters

- **`problem`** (`pygeosx.Group`) – GEOSX problem handle
- **`target_key`** (*`str`*) – Key for the target wrapper

Returns

The wrapper as a numpy ndarray

Return type

np.ndarray

`pygeosx_tools.wrapper.`**`gather_wrapper`**`(`*problem*, *key*, *ghost_key=''*`)`

Get a global copy of a wrapper as a numpy ndarray on rank 0

Parameters

- **`problem`** (`pygeosx.Group`) – GEOSX problem handle
- **`target_key`** (*`str`*) – Key for the target wrapper

Returns

The wrapper as a numpy ndarray

Return type

np.ndarray

`pygeosx_tools.wrapper.`**`get_global_value_range`**`(`*problem*, *key*`)`

Get the range of a target value across all processes

Parameters

- **problem** (pygeosx.Group) – GEOSX problem handle
- **target_key** (*str*) – Key for the target wrapper

Returns
The global min/max of the target

Return type
tuple

pygeosx_tools.wrapper.**get_matching_wrapper_path**(*problem*, *filters*)

Recursively search the group and its children for wrappers that match the filters A successful match is identified if the wrapper path contains all of the strings in the filter. For example, if filters=[‘a’, ‘b’, ‘c’], the following could match any of the following: ‘a/b/c’, ‘c/b/a’, ‘d/e/c/f/b/a/a’

Parameters

- **problem** (pygeosx.Group) – GEOSX problem handle
- **filters** (*list*) – a list of strings

Returns
Key of the matching wrapper

Return type
str

pygeosx_tools.wrapper.**get_wrapper**(*problem*, *target_key*, *write_flag=False*)

Get a local copy of a wrapper as a numpy ndarray

Parameters

- **filename** (*str*) – Catalog file name
- **problem** (pygeosx.Group) – GEOSX problem handle
- **target_key** (*str*) – Key for the target wrapper
- **write_flag** (*bool*) – Sets write mode (default=False)

Returns
The wrapper as a numpy ndarray

Return type
np.ndarray

pygeosx_tools.wrapper.**get_wrapper_par**(*problem*, *target_key*, *allgather=False*, *ghost_key=''*)

Get a global copy of a wrapper as a numpy ndarray. Note: if ghost_key is set, it will try to remove any ghost elements

Parameters

- **problem** (pygeosx.Group) – GEOSX problem handle
- **target_key** (*str*) – Key for the target wrapper
- **allgather** (*bool*) – Flag to trigger allgather across ranks (False)
- **ghost_key** (*str*) – Key for the corresponding ghost wrapper (default=”)

Returns
The wrapper as a numpy ndarray

Return type
np.ndarray

`pygeosx_tools.wrapper.`**`plot_history`**(*records*, *output_root='.'*, *save_figures=True*, *show_figures=True*)

Plot the time-histories for the records structure. Note: If figures are shown, the GEOSX process will be blocked until they are closed

Parameters

- **records** (*dict*) – A dict of dicts containing the queries
- **output_root** (*str*) – Path to save figures (default = './')
- **save_figures** (*bool*) – Flag to indicate whether figures should be saved (default = True)
- **show_figures** (*bool*) – Flag to indicate whether figures should be drawn (default = False)

`pygeosx_tools.wrapper.`**`print_global_value_range`**(*problem*, *key*, *header*, *scale=1.0*, *precision='%1.4f'*)

Print the range of a target value across all processes

Parameters

- **problem** (`pygeosx.Group`) – GEOSX problem handle
- **target_key** (*str*) – Key for the target wrapper
- **header** (*str*) – Header to print with the range
- **scale** (*float*) – Multiply the range with this value before printing (default = 1.0)
- **precision** (*str*) – Format for printing the range (default = %1.4f)

Returns
The global min/max of the target

Return type
tuple

`pygeosx_tools.wrapper.`**`run_queries`**(*problem*, *records*)

Query the current GEOSX datastructure Note: The expected record request format is as follows. For now, the only supported query is to find the min/max values of the target record = {'path/of/wrapper': {'label': 'aperture (m)', # A label to include with plots 'scale': 1.0, # Value to scale results by 'history: [], # A list to store values over time 'fhandle': plt.figure() # A figure handle }}

Parameters

- **problem** (`pygeosx.Group`) – GEOSX problem handle
- **records** (*dict*) – A dict of dicts that specifies the queries to run

`pygeosx_tools.wrapper.`**`search_datastructure_wrappers_recursive`**(*group*, *filters*, *matching_paths*, *level=0*, *group_path=[]*)

Recursively search the group and its children for wrappers that match the filters

Parameters

- **problem** (`pygeosx.Group`) – GEOSX problem handle
- **filters** (*list*) – a list of strings
- **matching_paths** (*list*) – a list of matching values

`pygeosx_tools.wrapper.`**`set_wrapper_to_value`**(*problem*, *key*, *value*)

Set the value of a wrapper

Parameters

- **problem** (`pygeosx.Group`) – GEOSX problem handle

- **target_key** (*str*) – Key for the target wrapper
- **value** (*float*) – Value to set the wrapper

pygeosx_tools.wrapper.**set_wrapper_with_function**(*problem*, *target_key*, *input_keys*, *fn*, *target_index=-1*)

Set the value of a wrapper using a function

Parameters

- **problem** (pygeosx.Group) – GEOSX problem handle
- **target_key** (*str*) – Key for the target wrapper
- **input_keys** (*str, list*) – The input key(s)
- **fn** (*function*) – Vectorized function used to calculate target values
- **target_index** (*int*) – Target index to write the output (default = all)

pygeosx_tools.file_io.**load_tables**(*axes_names: Iterable[str]*, *property_names: Iterable[str]*, *table_root: str = './tables'*, *extension: str = 'csv'*) → Tuple[Iterable[ndarray], Dict[str, ndarray]]

Load a set of tables in GEOSX format

Parameters

- **axes_names** (*list*) – Axis file names in the target directory (with no extension)
- **property_names** (*list*) – Property file names in the target directory (with not extension)
- **table_root** (*str*) – Root path for the table directory
- **extension** (*str*) – Table file extension (default = 'csv')

Returns

List of axes values, and dictionary of table values

Return type

tuple

pygeosx_tools.file_io.**save_tables**(*axes: Iterable[ndarray]*, *properties: Dict[str, ndarray]*, *table_root: str = './tables'*, *axes_names: List[str] = []*) → None

Saves a set of tables in GEOSX format

The shape of these arrays should match the length of each axis in the specified order. The output directory will be created if it does not exist yet. If axes_names are not supplied, then they will be selected based on the dimensionality of the grid: 1D=[t]; 3D=[x, y, z]; 4D=[x, y, z, t].

Parameters

- **axes** (*list*) – A list of numpy ndarrays defining the table axes
- **properties** (*dict*) – A dict of numpy ndarrays defning the table values
- **table_root** (*str*) – The root path for the output directory
- **axes_names** (*list*) – A list of names for each potential axis (optional)

pygeosx_tools.mesh_interpolation.**apply_to_bins**(*fn: Callable[[float | ndarray], float]*, *position: ndarray*, *value: ndarray*, *bins: ndarray*, *collapse_edges: bool = True*)

Apply a function to values that are located within a series of bins Note: if a bin is empty, this function will fill a nan value

Parameters

- **fn** (*function*) – Function that takes a single scalar or array input
- **position** (*np.ndarray*) – A 1D list/array describing the location of each sample
- **value** (*np.ndarray*) – A 1D list/array of values at each location
- **bins** (*np.ndarray*) – The bin edges for the position data
- **collapse_edges** (*bool*) – Controls the behavior of edge-data (default=True)

Returns
an array of function results for each bin

Return type
np.ndarray

`pygeosx_tools.mesh_interpolation.`**`extrapolate_nan_values`**`(`*x*, *y*, *slope_scale=0.0*`)`

Fill in any nan values in two 1D arrays by extrapolating

Parameters

- **x** (*np.ndarray*) – 1D list/array of positions
- **y** (*np.ndarray*) – 1D list/array of values
- **slope_scale** (*float*) – value to scale the extrapolation slope (default=0.0)

Returns
The input array with nan values replaced by extrapolated data

Return type
np.ndarray

`pygeosx_tools.mesh_interpolation.`**`get_random_realization`**`(`*x*, *bins*, *value*, *rand_fill=0*, *rand_scale=0*, *slope_scale=0*`)`

Get a random realization for a noisy signal with a set of bins

Parameters

- **x** (*np.ndarray*) – 1D list/array of positions
- **bins** (*np.ndarray*) – 1D list/array of bin edges
- **value** (*np.ndarray*) – 1D list/array of values
- **rand_fill** (*float*) – The standard deviation to use where data is not defined (default=0)
- **rand_scale** (*float*) – Value to scale the standard deviation for the realization (default=0)
- **slope_scale** (*float*) – Value to scale the extrapolation slope (default=0.0)

Returns
An array containing the random realization

Return type
np.ndarray

`pygeosx_tools.mesh_interpolation.`**`get_realizations`**`(`*x*, *bins*, *targets*`)`

Get random realizations for noisy signals on target bins

Parameters

- **x** (*np.ndarray*) – 1D list/array of positions
- **bins** (*np.ndarray*) – 1D list/array of bin edges
- **targets** (*dict*) – Dict of geosx target keys, inputs to get_random_realization

Returns
Dictionary of random realizations

Return type
dict

`pygeosx_tools.well_log.`**`convert_E_nu_to_K_G`**`(`*E*, *nu*`)`

Convert young’s modulus and poisson’s ratio to bulk and shear modulus

Parameters

- **E** (*float, np.ndarray*) – Young’s modulus
- **nu** (*float, np.ndarray*) – Poisson’s ratio

Returns
bulk modulus, shear modulus with same size as inputs

Return type
tuple

`pygeosx_tools.well_log.`**`estimate_shmin`**`(`*z*, *rho*, *nu*`)`

Estimate the minimum horizontal stress using the poisson’s ratio

Parameters

- **z** (*float, np.ndarray*) – Depth
- **rho** (*float, np.ndarray*) – Density
- **nu** (*float, np.ndarray*) – Poisson’s ratio

Returns
minimum horizontal stress

Return type
float

`pygeosx_tools.well_log.`**`parse_las`**`(`*fname*, *variable_start='~C'*, *body_start='~A'*`)`

Parse an las format log file

Parameters

- **fname** (*str*) – Path to the log file
- **variable_start** (*str*) – A string that indicates the start of variable header information (default = ‘~CURVE INFORMATION’)
- **body_start** (*str*) – a string that indicates the start of the log body (default = ‘~A’)

Returns
a dict containing the values and unit definitions for each variable in the log

Return type
np.ndarray

Time History Tools

`timehistory.plot_time_history.`**`getHistorySeries`**(*database*, *variable*, *setname*, *indices=None*, *components=None*)

Retrieve a series of time history structures suitable for plotting in addition to the specific set index and component for the time series

Parameters

- **`database`** (*`hdf5_wrapper.hdf5_wrapper`*) – database to retrieve time history data from
- **`variable`** (*`str`*) – the name of the time history variable for which to retrieve time-series data
- **`setname`** (*`str`*) – the name of the index set as specified in the geosx input xml for which to query time-series data
- **`indices`** (*`int, list`*) – the indices in the named set to query for, if None, defaults to all
- **`components`** (*`int, list`*) – the components in the flattened data types to retrieve, defaults to all

Returns

list of (time, data, idx, comp) timeseries tuples for each time history data component

Return type

list

`mesh_doctor`

`mesh_doctor` is a `python` executable that can be used through the command line to perform various checks, validations, and tiny fixes to the `vtk` mesh that are meant to be used in `geos`. `mesh_doctor` is organized as a collection of modules with their dedicated sets of options. The current page will introduce those modules, but the details and all the arguments can be retrieved by using the `--help` option for each module.

Modules

To list all the modules available through `mesh_doctor`, you can simply use the `--help` option, which will list all available modules as well as a quick summary.

```
$ python mesh_doctor.py --help
usage: mesh_doctor.py [-h] [-v] [-q] -i VTK_MESH_FILE
                      {collocated_nodes,element_volumes,fix_elements_orderings,generate_
↪cube,generate_fractures,generate_global_ids,non_conformal,self_intersecting_elements,
↪supported_elements}
                      ...

Inspects meshes for GEOSX.

positional arguments:
  {collocated_nodes,element_volumes,fix_elements_orderings,generate_cube,generate_
↪fractures,generate_global_ids,non_conformal,self_intersecting_elements,supported_
↪elements}
        Modules
    collocated_nodes
```

(continues on next page)

(continued from previous page)

```
        Checks if nodes are collocated.
    element_volumes
        Checks if the volumes of the elements are greater than "min".
    fix_elements_orderings
        Reorders the support nodes for the given cell types.
    generate_cube
        Generate a cube and its fields.
    generate_fractures
        Splits the mesh to generate the faults and fractures. [EXPERIMENTAL]
    generate_global_ids
        Adds globals ids for points and cells.
    non_conformal
        Detects non conformal elements. [EXPERIMENTAL]
    self_intersecting_elements
        Checks if the faces of the elements are self intersecting.
    supported_elements
        Check that all the elements of the mesh are supported by GEOSX.

options:
  -h, --help
        show this help message and exit
  -v    Use -v 'INFO', -vv for 'DEBUG'. Defaults to 'WARNING'.
  -q    Use -q to reduce the verbosity of the output.
  -i VTK_MESH_FILE, --vtk-input-file VTK_MESH_FILE

Note that checks are dynamically loaded.
An option may be missing because of an unloaded module.
Increase verbosity (-v, -vv) to get full information.
```

Then, if you are interested in a specific module, you can ask for its documentation using the `mesh_doctor module_name --help` pattern. For example

```
$ python mesh_doctor.py collocated_nodes --help
usage: mesh_doctor.py collocated_nodes [-h] --tolerance TOLERANCE

options:
  -h, --help            show this help message and exit
  --tolerance TOLERANCE
                        [float]: The absolute distance between two nodes for
                        them to be considered collocated.
```

`mesh_doctor` loads its module dynamically. If a module can't be loaded, `mesh_doctor` will proceed and try to load other modules. If you see a message like

```
[1970-04-14 03:07:15,625][WARNING] Could not load module "collocated_nodes": No module␣
↪named 'vtkmodules'
```

then most likely `mesh_doctor` could not load the `collocated_nodes` module, because the `vtk` python package was not found. Thereafter, the documentation for module `collocated_nodes` will not be displayed. You can solve this issue by installing the dependencies of `mesh_doctor` defined in its `requirements.txt` file (`python -m pip install -r requirements.txt`).

Here is a list and brief description of all the modules available.

collocated_nodes

Displays the neighboring nodes that are closer to each other than a prescribed threshold. It is not uncommon to define multiple nodes for the exact same position, which will typically be an issue for geos and should be fixed.

```
$ python mesh_doctor.py collocated_nodes --help
usage: mesh_doctor.py collocated_nodes [-h] --tolerance TOLERANCE

options:
  -h, --help            show this help message and exit
  --tolerance TOLERANCE
                        [float]: The absolute distance between two nodes for
                        them to be considered collocated.
```

element_volumes

Computes the volumes of all the cells and displays the ones that are below a prescribed threshold. Cells with negative volumes will typically be an issue for geos and should be fixed.

```
$ python mesh_doctor.py element_volumes --help
usage: mesh_doctor.py element_volumes [-h] --min 0.0

options:
  -h, --help  show this help message and exit
  --min 0.0   [float]: The minimum acceptable volume. Defaults to 0.0.
```

fix_elements_orderings

It sometimes happens that an exported mesh does not abide by the vtk orderings. The fix_elements_orderings module can rearrange the nodes of given types of elements. This can be convenient if you cannot regenerate the mesh.

```
$ python mesh_doctor.py fix_elements_orderings --help
usage: mesh_doctor.py fix_elements_orderings [-h]
                                             [--Hexahedron 1,6,5,4,7,0,2,3]
                                             [--Prism5 8,2,0,7,6,9,5,1,4,3]
                                             [--Prism6 11,2,8,10,5,0,9,7,6,1,4,3]
                                             [--Pyramid 3,4,0,2,1]
                                             [--Tetrahedron 2,0,3,1]
                                             [--Voxel 1,6,5,4,7,0,2,3]
                                             [--Wedge 3,5,4,0,2,1] --output
                                             OUTPUT
                                             [--data-mode binary, ascii]


options:
  -h, --help            show this help message and exit
  --Hexahedron 1,6,5,4,7,0,2,3
                        [list of integers]: node permutation for "Hexahedron".
  --Prism5 8,2,0,7,6,9,5,1,4,3
                        [list of integers]: node permutation for "Prism5".
  --Prism6 11,2,8,10,5,0,9,7,6,1,4,3
```

(continues on next page)

(continued from previous page)

```
                        [list of integers]: node permutation for "Prism6".
  --Pyramid 3,4,0,2,1   [list of integers]: node permutation for "Pyramid".
  --Tetrahedron 2,0,3,1
                        [list of integers]: node permutation for
                        "Tetrahedron".
  --Voxel 1,6,5,4,7,0,2,3
                        [list of integers]: node permutation for "Voxel".
  --Wedge 3,5,4,0,2,1   [list of integers]: node permutation for "Wedge".
  --output OUTPUT       [string]: The vtk output file destination.
  --data-mode binary, ascii
                        [string]: For ".vtu" output format, the data mode can
                        be binary or ascii. Defaults to binary.
```

generate_cube

This module conveniently generates cubic meshes in vtk. It can also generate fields with simple values. This tool can also be useful to generate a trial mesh that will later be refined or customized.

```
$ python mesh_doctor.py generate_cube --help
usage: mesh_doctor.py generate_cube [-h] [--x 0:1.5:3] [--y 0:5:10] [--z 0:1]
                                    [--nx 2:2] [--ny 1;1] [--nz 4]
                                    [--fields name:support:dim [name:support:dim ...]]
                                    [--cells] [--no-cells] [--points]
                                    [--no-points] --output OUTPUT
                                    [--data-mode binary, ascii]

options:
  -h, --help            show this help message and exit
  --x 0:1.5:3           [list of floats]: X coordinates of the points.
  --y 0:5:10            [list of floats]: Y coordinates of the points.
  --z 0:1               [list of floats]: Z coordinates of the points.
  --nx 2:2              [list of integers]: Number of elements in the X
                        direction.
  --ny 1;1              [list of integers]: Number of elements in the Y
                        direction.
  --nz 4                [list of integers]: Number of elements in the Z
                        direction.
  --fields name:support:dim [name:support:dim ...]
                        Create fields on CELLS or POINTS, with given dimension
                        (typically 1 or 3).
  --cells               [bool]: Generate global ids for cells. Defaults to
                        true.
  --no-cells            [bool]: Don't generate global ids for cells.
  --points              [bool]: Generate global ids for points. Defaults to
                        true.
  --no-points           [bool]: Don't generate global ids for points.
  --output OUTPUT       [string]: The vtk output file destination.
  --data-mode binary, ascii
                        [string]: For ".vtu" output format, the data mode can
                        be binary or ascii. Defaults to binary.
```

generate_fractures

For a conformal fracture to be defined in a mesh, geos requires the mesh to be split at the faces where the fracture gets across the mesh. The generate_fractures module will split the mesh and generate the multi-block vtk files.

```
$ python mesh_doctor.py generate_fractures --help
usage: mesh_doctor.py generate_fractures [-h] --policy field
                                         [--split_on_domain_boundary True]
                                         [--name NAME] [--values VALUES]
                                         --output OUTPUT
                                         [--data-mode binary, ascii]
                                         --fracture-output FRACTURE_OUTPUT
                                         [--fracture-data-mode binary, ascii]

options:
  -h, --help            show this help message and exit
  --policy field        [string]: The criterion to define the surfaces that
                        will be changed into fracture zones. Possible values
                        are "field"
  --split_on_domain_boundary True
                        [bool]: Split policy if the fracture touches the
                        boundary of the mesh. Defaults to true.
  --name NAME           [string]: If the "field" policy is selected, defines
                        which field will be considered to define the
                        fractures.
  --values VALUES       [list of comma separated integers]: If the "field"
                        policy is selected, which changes of the field will be
                        considered as a fracture.
  --output OUTPUT       [string]: The vtk output file destination.
  --data-mode binary, ascii
                        [string]: For ".vtu" output format, the data mode can
                        be binary or ascii. Defaults to binary.
  --fracture-output FRACTURE_OUTPUT
                        [string]: The vtk output file destination.
  --fracture-data-mode binary, ascii
                        [string]: For ".vtu" output format, the data mode can
                        be binary or ascii. Defaults to binary.
```

generate_global_ids

When running geos in parallel, *global ids* can be used to refer to data across multiple ranks. The generate_global_ids can generate *global ids* for the imported vtk mesh.

```
$ python mesh_doctor.py generate_global_ids --help
usage: mesh_doctor.py generate_global_ids [-h] [--cells] [--no-cells]
                                          [--points] [--no-points] --output
                                          OUTPUT [--data-mode binary, ascii]

options:
  -h, --help            show this help message and exit
  --cells               [bool]: Generate global ids for cells. Defaults to
                        true.
```

(continues on next page)

(continued from previous page)

```
  --no-cells            [bool]: Don't generate global ids for cells.
  --points              [bool]: Generate global ids for points. Defaults to
                        true.
  --no-points           [bool]: Don't generate global ids for points.
  --output OUTPUT       [string]: The vtk output file destination.
  --data-mode binary, ascii
                        [string]: For ".vtu" output format, the data mode can
                        be binary or ascii. Defaults to binary.
```

non_conformal

This module will detect elements which are close enough (there's a user defined threshold) but which are not in front of each other (another threshold can be defined). *Close enough* can be defined in terms or proximity of the nodes and faces of the elements. The angle between two faces can also be precribed. This module can be a bit time consuming.

```
$ python mesh_doctor.py non_conformal --help
usage: mesh_doctor.py non_conformal [-h] [--angle_tolerance 10.0]
                                    [--point_tolerance POINT_TOLERANCE]
                                    [--face_tolerance FACE_TOLERANCE]

options:
  -h, --help            show this help message and exit
  --angle_tolerance 10.0
                        [float]: angle tolerance in degrees. Defaults to 10.0
  --point_tolerance POINT_TOLERANCE
                        [float]: tolerance for two points to be considered
                        collocated.
  --face_tolerance FACE_TOLERANCE
                        [float]: tolerance for two faces to be considered
                        "touching".
```

self_intersecting_elements

Some meshes can have cells that auto-intersect. This module will display the elements that have faces intersecting.

```
$ python mesh_doctor.py self_intersecting_elements --help
usage: mesh_doctor.py self_intersecting_elements [-h]
                                                 [--min 2.220446049250313e-16]

options:
  -h, --help            show this help message and exit
  --min 2.220446049250313e-16
                        [float]: The tolerance in the computation. Defaults to
                        your machine precision 2.220446049250313e-16.
```

supported_elements

`geos` supports a specific set of elements. Let's cite the standard elements like *tetrahedra*, *wedges*, *pyramids* or *hexahedra*. But also prismes up to 11 faces. `geos` also supports the generic `VTK_POLYHEDRON/42` elements, which are converted on the fly into one of the elements just described.

The `supported_elements` check will validate that no unsupported element is included in the input mesh. It will also verify that the `VTK_POLYHEDRON` cells can effectively get converted into a supported type of element.

```
$ python mesh_doctor.py supported_elements --help
usage: mesh_doctor.py supported_elements [-h] [--chunck_size 1] [--nproc 2]

options:
  -h, --help        show this help message and exit
  --chunck_size 1   [int]: Defaults chunk size for parallel processing to 1
  --nproc 2         [int]: Number of threads used for parallel processing.
                    Defaults to your CPU count 2.
```

1.9 Build Guide

Welcome to the GEOS build guide.

1.9.1 System prerequisites

To configure and build GEOS you will need the following tools available on your system.

List of prerequisites

Minimal requirements:

- CMake build system generator (3.17+).
- build tools (GNU make or ninja on Linux, XCode on MacOS).
- a C++ compiler with full c++17 standard support (gcc 8.3+ or clang 10.0+ are recommended).
- python (2.7+ or 3.6+).
- `zlib`, `blas` and `lapack` libraries
- any compatible MPI runtime and compilers (if building with MPI)

If you want to build from a repository check out (instead of a release tarball):

- git (2.20+ is tested, but most versions should work fine)

If you plan on building bundled third-party library (TPLs) dependencies yourself:

- Compatible C and Fortran compilers

If you will be checking out and running integrated tests (a submodule of GEOS, currently not publicly available):

- git-lfs (Git Large File Storage extension)
- h5py and mpi4py python modules

If you are interested in building Doxygen documentation:

- GNU bison
- LaTeX
- ghostscript
- Graphviz

In order for XML validation to work (executed as an optional build step):

- xmllint

Installing prerequisites

On a local development machine with sudo/root privileges, most of these dependencies can be installed with a system package manager. For example, on a Debian-based system (check your package manager for specific package names):

```
sudo apt install build-essential git git-lfs gcc g++ gfortran cmake libopenmpi-dev␣
↪libblas-dev liblapack-dev zlib1g-dev python3 python3-h5py python3-mpi4py libxml2-utils
```

On HPC systems it is typical for these tools to be installed by system administrators and provided via modules. To list available modules, type:

```
module avail
```

Then load the appropriate modules using `module load` command. Please contact your system administrator if you need help choosing or installing appropriate modules.

1.9.2 Third-party dependencies

GEOS makes use of multiple third-party libraries (TPLs) and tools, some of which are mandatory and some optional. We only test against specific versions, and sometimes even require development snapshots (specific git commits). Not all of these guarantee backwards compatibility, so we strongly recommend building with these specific versions.

List of third-party libraries and tools

The two tables below lists the dependencies with their specific versions and relevant CMake variables. Some of these libraries may have their own system prerequisites.

Libraries

The following libraries are linked to by GEOS:

Name	Version	Enable option	Path variable	Description
Adiak	0.2.0	`ENABLE_CALI`	`ADIAK_DIR`	Library for collecting metadata from HPC application runs, and distributing that metadata to subscriber tools.
Caliper	2.4.0	`ENABLE_CALI`	`CALIPER_DI`	Instrumentation and performance profiling library.
conduit	0.5.0	*mandatory*	`CONDUIT_DI`	Simplified Data Exchange for HPC Simulations.
CHAI	2.2.2	*mandatory*	`CHAI_DIR`	Copy-hiding array abstraction to automatically migrate data between memory spaces.
RAJA	0.12.1	*mandatory*	`RAJA_DIR`	Collection of C++ software abstractions that enable architecture portability for HPC applications.
hdf5	1.10.5	*mandatory*	`HDF5_DIR`	High-performance data management and storage suite.
mathpresso	2015-12-15	`ENABLE_MATH`	`MATHPRESSC`	Mathematical Expression Parser and JIT Compiler.
pugixml	1.8.0	*mandatory*	`PUGIXML_DI`	Light-weight, simple and fast XML parser for C++ with XPath support.
parmeti	4.0.3	*mandatory* (with MPI)	`PARMETIS_D`	Parallel Graph Partitioning library. Should be built with 64-bit `idx_t` type.
suitesparse	5.8.1	`ENABLE_SUIT`	`SUITESPARS`	A suite of sparse matrix software.
superlu_di	0f6efc3	`ENABLE_SUPE`	`SUPERLU_DI`	General purpose library for the direct solution of large, sparse, nonsymmetric systems of linear equations.
hypre	2186a8	`ENABLE_HYPR`	`HYPRE_DIR`	Library of high performance preconditioners and solvers for large sparse linear systems on massively parallel computers.
PETSc	3.13.0	`ENABLE_PETS`	`PETSC_DIR`	Suite of data structures and routines for the scalable (parallel) solution of scientific applications.
Trilinos	12.18.1	`ENABLE_TRIL`	`TRILINOS_D`	Collection of reusable scientific software libraries, known in particular for linear solvers.
silo	4.10.3	*mandatory*	`SILO_DIR`	A Mesh and Field I/O Library and Scientific Database.
VTK	9.0.0-rc3	`ENABLE_VTK`	`VTK_DIR`	Open source software for manipulating and displaying scientific data.

Tools

The following tools are used as part of the build process to support GEOS development:

Name	Version	Enable option	Path variable	Description
doxygen	1.8.20	`ENABLE_DOXYG`	`DOXYGEN_EXECUTA`	De facto standard tool for generating documentation from annotated C++ sources.
sphinx	1.8.5	`ENABLE_SPHIN`	`SPHINX_EXECUTAB`	A tool that makes it easy to create intelligent and beautiful documentation.
uncrustify	401a40	`ENABLE_UNCRU`	`UNCRUSTIFY_EXEC`	A source code beautifier for C, C++, C#, ObjectiveC, D, Java, Pawn and VALA.

Some other dependencies (GoogleTest, GoogleBenchmark) are provided through BLT build system which is embedded in GEOS source. No actions are needed to build them.

If you would like to create a Docker image with all dependencies, take a look at Dockerfiles that are used in our CI process.

Building bundled dependencies

To simplify the process of building TPLs, we provide a git repository thirdPartyLibs. It contains source copies of exact TPL versions required and is updated periodically. It also contains a CMake script for building all TPLs in a single command.

The recommended steps to build TPLs are:

- Create a host-config file that sets all system-specific CMake variables (compiler and library paths, configuration flags, etc.) Take a look at host-config examples.
- Configure via `config-build.py` script:

  ```
  cd thirdPartyLibs
  python scripts/config-build.py --hostconfig=/path/to/host-config.cmake --
  ↪buildtype=Release --installpath=/path/to/install/dir -DNUM_PROC=8
  ```

  where

  - `--buildpath` or `-bp` is the build directory (by default, created under current).
  - `--installpath` or `-ip` is the installation directory(wraps `CMAKE_INSTALL_PREFIX`).
  - `--buildtype` or `-bt` is a wrapper to the `CMAKE_BUILD_TYPE` option.
  - `--hostconfig` or `-hc` is a path to host-config file.
  - all other command-line options are passed to CMake.

- Run the build:

  ```
  cd <buildpath>
  make
  ```

  Warning: Do not provide `-j` argument to `make` here, since the top-level make only launches sub-project builds. Instead use `-DNUM_PROC` option above, which is passed to each sub-project's `make` command.

You may also run the CMake configure step manually instead of relying on `config-build.py`. The full TPL build may take anywhere between 15 minutes and 2 hours, depending on your machine, number of threads and libraries enabled.

Note: An exception from the above pattern, `sphinx` is currently not a part of the TPL bundle and must be installed with your Python or package manager.

Note: PETSc build currently downloads pt-scotch from the internet. If you do not have access to internet, modify the *./configure* step of petsc in *CMakeLists.txt* and change the `--download-ptscotch` option accordingly. *pt-scotch* also relies on *bison* and *flex*.

Installing dependencies individually

You may also install each individual TPL separately, either manually or through a package manager. This is a more difficult route, since you are responsible for configuring dependencies in a compatible manner. Again, we strongly recommend using the exact versions listed above, to avoid possible build problems.

You may look at our TPL CMake script to see how we configure TPL builds.

1.9.3 Building GEOS

Build steps

- Create a host-config file that sets all system-specific CMake variables. Take a look at host-config examples. We recommend the same host-config is used for both TPL and GEOS builds. In particular, certain options (such as `ENABLE_MPI` or `ENABLE_CUDA`) need to match between the two.
- Provide paths to all enabled TPLs. This can be done in one of two ways:
  - Provide each path via a separate CMake variable (see *Third-party dependencies* for path variable names).
  - If you built TPLs from the `tplMirror` repository, you can set `GEOSX_TPL_DIR` variable in your host-config to point to the TPL installation path, and

    ```
    include("/path/to/GEOS/host-configs/tpls.cmake")
    ```

    which will set all the individual TPL paths for you.
- Configure via `config-build.py` script:

  ```
  cd GEOS
  python scripts/config-build.py --hostconfig=/path/to/host-config.cmake --
  ↪buildtype=Release --installpath=/path/to/install/dir
  ```

  where

  - `--buildpath` or `-bp` is the build directory (by default, created under current working dir).
  - `--installpath` or `-ip` is the installation directory(wraps `CMAKE_INSTALL_PREFIX`).
  - `--buildtype` or `-bt` is a wrapper to the `CMAKE_BUILD_TYPE` option.
  - `--hostconfig` or `-hc` is a path to host-config file.
  - all unrecognized options are passed to CMake.

  If `--buildpath` is not used, build directory is automatically named `build-<config-filename-without-extension>-<buildtype>`. It is possible to keep automatic naming and change the build root directory with `--buildrootdir`. In that case, build path will be set to `<buildrootdir>/<config-filename-without-extension>-<buildtype>`. Both `--buildpath` and `--buildrootdir` are incompatible and cannot be used in the same time. Same pattern is applicable to install path, with `--installpath` and `--installrootdir` options.
- Run the build:

  ```
  cd <buildpath>
  make -j $(nproc)
  ```

You may also run the CMake configure step manually instead of relying on `config-build.py`. A full build typically takes between 10 and 30 minutes, depending on chosen compilers, options and number of cores.

Configuration options

Below is a list of CMake configuration options, in addition to TPL options above. Some options, when enabled, require additional settings (e.g. ENABLE_CUDA). Please see host-config examples.

Option	Default	Explanation
ENABLE_MPI	ON	Build with MPI (also applies to TPLs)
ENABLE_OPENMP	OFF	Build with OpenMP (also applies to TPLs)
ENABLE_CUDA	OFF	Build with CUDA (also applies to TPLs)
ENABLE_CUDA_NVTOOLSE	OFF	Enable CUDA NVTX user instrumentation (via GEOS_MARK_SCOPE or GEOS_MARK_FUNCTION macros)
ENABLE_HIP	OFF	Build with HIP/ROCM (also applies to TPLs)
ENABLE_DOCS	ON	Build documentation (Sphinx and Doxygen)
ENABLE_WARNINGS_AS_E	ON	Treat all warnings as errors
ENABLE_PVTPackage	ON	Enable PVTPackage library (required for compositional flow runs)
ENABLE_TOTALVIEW_OUI	OFF	Enables TotalView debugger custom view of GEOS data structures
GEOS_ENABLE_TESTS	ON	Enables unit testing targets
GEOSX_ENABLE_FPE	ON	Enable floating point exception trapping
GEOSX_LA_INTERFACE	Hypre	Choie of Linear Algebra backend (Hypre/Petsc/Trilinos)
GEOSX_BUILD_OBJ_LIBS	ON	Use CMake Object Libraries build
GEOSX_BUILD_SHARED_L	OFF	Build `geosx_core` as a shared library instead of static
GEOSX_PARALLEL_COMPI		Max. number of compile jobs (when using Ninja), in addition to `-j` flag
GEOSX_PARALLEL_LINK_		Max. number of link jobs (when using Ninja), in addition to `-j` flag
GEOSX_INSTALL_SCHEMA	ON	Enables schema generation and installation

1.9.4 Spack and Uberenv

GEOS is transitioning to a new Spack and Uberenv system for building our dependencies. We refer the reader to the Spack documentation and Uberenv documentation, in particular the Spack documentation for specs and dependencies, manual compiler configuration and external packages are worth reading.

Building the dependencies can be as simple as running

```
./scripts/uberenv/uberenv.py
```

This will create a directory `uberenv_libs` in the current working directory, clone Spack into `uberenv_libs/spack` and install the dependencies into `uberenv_libs/system_dependent_path`. It will then spit out a host-config file in the current directory which you can use to build GEOS. While the above command **should** work on every system, it should never be used. Invoked as such, Spack will ignore any system libraries you have installed and will go down a rabbit hole building dependencies. Furthermore this does not allow you to choose the compiler to build. Both of these are easily solved by creating a directory with a `packages.yaml` and a `compilers.yaml`.

To prevent this from happening you'll need to create a directory with a `packages.yaml` file and a `compilers.yaml` file. You can find working examples for commonly used systems in scripts/uberenv/spack_configs. It is worth noting that each LC system type has two such directories, for example there is a `toss_3_x85_54_ib` and `toss_3_x85_54_ib_python` directory. This is because when building `pygeosx` Python needs to be built from scratch, and as such cannot be listed in `packages.yaml`. However, when not building `pygeosx` other dependencies depend on python, but an existing system version works just fine, so it can be put in `packages.yaml` to prevent Spack from building it.

Once you have these files setup you can run Uberenv again and instruct it to use them with. If for instance you added Clang 10.0.1 to the `compilers.yaml` file the your command would look something like this:

```
./scripts/uberenv/uberenv.py --spack-config-dir=/path/to/your/config/directory/ --spec="
↪%clang@10.0.1"
```

Note: When building `pygeosx`, Spack will build various python packages, however by default they are not installed in python. There are various ways of accomplishing this, but the recommended approach is to use `spack activate`. The command would look something like this `./uberenv_libs/spack/bin/spack activate py-numpy py-scipy py-pip py-mpi4py`

Build Configuration

The GEOS Spack package has a lot of options for controlling which dependencies you would like to build and how you'd like them built. The GEOS Spack package file is at `` `scripts/uberenv/packages/geosx/package.py <https://github.com/GEOS-DEV/GEOS/tree/develop/scripts/uberenv/packages/geosx/package.py>`_. `` The variants for the package are as follows

```
    variant('shared', default=True, description='Build Shared Libs.')
    variant('caliper', default=True, description='Build Caliper support.')
    variant('mkl', default=False, description='Use the Intel MKL library.')
    variant('essl', default=False, description='Use the IBM ESSL library.')
    variant('suite-sparse', default=True, description='Build SuiteSparse support.')
    variant('trilinos', default=True, description='Build Trilinos support.')
    variant('hypre', default=True, description='Build HYPRE support.')
    variant('hypre-cuda', default=False, description='Build HYPRE with CUDA support.')
    variant('petsc', default=True, description='Build PETSc support.')
    variant('scotch', default=True, description='Build Scotch support.')
    variant('lai',
            default='trilinos',
            description='Linear algebra interface.',
            values=('trilinos', 'hypre', 'petsc'),
            multi=False)
    variant('pygeosx', default=False, description='Build the GEOSX python interface.')
```

For example if you wanted to build with GCC 8.3.1, without Caliper and with PETSC as the Linear Algebra Interface, your spec would be `%gcc@8.3.1 ~caliper lai=petsc`.

The GEOS Spack package lists out the libraries that GEOS depends ons. Currently these dependencies are

```
    depends_on('cmake@3.8:', type='build')
    depends_on('cmake@3.9:', when='+cuda', type='build')

    #
    # Virtual packages
    #
    depends_on('mpi')
    depends_on('blas')
    depends_on('lapack')

    #
```

(continues on next page)

(continued from previous page)

```
    # Performance portability
    #
    depends_on('raja@0.12.1 +openmp +shared ~examples ~exercises')
    depends_on('raja +cuda', when='+cuda')

    depends_on('umpire@4.1.2 ~c +shared +openmp ~examples')
    depends_on('umpire +cuda', when='+cuda')

    depends_on('chai@2.2.2 +shared +raja ~benchmarks ~examples')
    depends_on('chai@2.2.2 +cuda', when='+cuda')

    #
    # IO
    #
    depends_on('hdf5@1.10.5: +shared +pic +mpi', when='~vtk')

    depends_on('conduit@0.5.0 +shared ~test ~fortran +mpi +hdf5 ~hdf5_compat')

    depends_on('silo@4.10: ~fortran +shared ~silex +pic +mpi ~zlib')

    depends_on('adiak@0.2: +mpi +shared', when='+caliper')
    depends_on('caliper@2.4: +shared +adiak +mpi ~callpath ~libpfm ~gotcha ~sampler',␣
↪when='+caliper')

    depends_on('pugixml@1.8: +shared')

    depends_on('fmt@8.0: cxxstd=14 +pic')

    #
    # Math
    #
    depends_on('intel-mkl +shared ~ilp64', when='+mkl')

    # depends_on('essl ~ilp64 threads=openmp +lapack +cuda', when='+essl')

    depends_on('parmetis@4.0.3: +shared +int64')

    depends_on('scotch@6.0.9: +mpi +int64', when='+scotch')

    depends_on('superlu-dist +int64 +openmp +shared', when='~petsc')
    depends_on('superlu-dist@6.3.0 +int64 +openmp +shared', when='+petsc')

    depends_on('suite-sparse@5.8.1: +pic +openmp +amd +camd +colamd +ccolamd +cholmod␣
↪+umfpack', when='+suite-sparse')
    depends_on('suite-sparse +blas-no-underscore', when='%gcc +suite-sparse +essl')

    trilinos_build_options = '~fortran +openmp +shared'
    trilinos_tpls = '~boost ~glm ~gtest ~hdf5 ~hypre ~matio ~metis +mpi ~mumps ~netcdf ~
↪suite-sparse'
    trilinos_packages = '+amesos +aztec +epetra +epetraext +ifpack +kokkos +ml +stk␣
↪+stratimikos +teuchos +tpetra ~amesos2 ~anasazi ~belos ~exodus ~ifpack2 ~muelu ~sacado␣
↪~zoltan ~zoltan2'
```

(continues on next page)

(continued from previous page)

```
    depends_on('trilinos@12.18.1 ' + trilinos_build_options + trilinos_tpls + trilinos_
↪packages, when='+trilinos')
    depends_on('trilinos +blas_lowercase_no_underscore', when='+trilinos +essl')
    # depends_on('trilinos +force-new-lapack', when='+trilinos +essl')

    depends_on('hypre@2.20.300 +shared +superlu-dist +mixedint +mpi +openmp', when=
↪'+hypre')
    depends_on('hypre@2.20.300 +cuda +shared +superlu-dist +mpi +openmp +unified-memory␣
↪+cusparse', when='+hypre-cuda')

    petsc_build_options = '+shared +mpi'
    petsc_tpls = '+metis ~hdf5 ~hypre +superlu-dist +int64'
    depends_on('petsc@3.13.0: ' + petsc_build_options + petsc_tpls, when='+petsc')

    #
    # Python
    #
    depends_on('python +shared +pic', when='+pygeosx')
    depends_on('py-numpy@1.19: +blas +lapack +force-parallel-build', when='+pygeosx')
    depends_on('py-scipy@1.5.2: +force-parallel-build', when='+pygeosx')
    depends_on('py-mpi4py@3.0.3:', when='+pygeosx')
    depends_on('py-pip', when='+pygeosx')

    #
    # Dev tools
    #
    depends_on('uncrustify@0.71:')

    #
    # Documentation
    #
    depends_on('doxygen@1.8.13:', when='+docs', type='build')
    depends_on('py-sphinx@1.6.3:', when='+docs', type='build')
```

Using the Spack spec syntax you can inturn specify variants for each of the dependencies of GEOS. So for example if you could modify the spec above to build RAJA in debug by using `%gcc@8.3.1 ~caliper lai=petsc ^raja build_type=Debug`. When building with Uberenv Spack should print out a table containing the full spec for every dependency it will build. If you would like to look at the variants for say RAJA in more detail you can find the package file at `uberenv_libs/spack/var/spack/repos/builtin/packages/raja/package.py`.

Adding a Dependency (Advanced)

Adding a dependency to GEOS is straight forward if the dependency already builds with Spack. If that is the case then all you need to do is add a `depends_on('cool-new-library')` to the GEOS `package.py` file. If however the dependency doesn't have a Spack package, you will have to add one by creating a `cool-new-library/package.yaml` file in the `scripts/uberenv/packages` directory and adding the logic to build it there.

Oftentimes (unfortunately), even when a package already exists, it might not work out of the box for your system. In this case copy over the existing `package.py` file from the Spack repository into `scripts/uberenv/packages/cool-new-library/package.py`, as if you were adding a new package, and perform your modifications there. Once you have the package working, copy the package back into the Spack repository (running Uberenv should do this for you) and commit+push your changes to Spack.

1.9.5 Continuous Integration process

To save building time, the third party libraries (that do not change so often) and GEOS are build separately.

Everytime a pull is requested in the TPL repository, docker images are generated and deployed on dockerhub. The repository names (ubuntu18.04-gcc8, centos7.7.1908-clang9.0.0, centos7.6.1810-gcc8.3.1-cuda10.1.243 etc.) obviously reflect the OS and the compiler flavour used. For each image, the unique tag `${PULL_REQUEST_NUMBER}-${BUILD_NUMBER}` (defined as `${{ github.event.number }}-${{ github.run_number }}` in github actions) is used so we can connect the related code source in a rather convenient way. Each docker contains the `org.opencontainers.image.created` and `org.opencontainers.image.revision` labels to provide additional information.

It is necessary to define the global environment `GEOSX_TPL_TAG` (*e.g.* something like `235-52`) in the .github/workflows/ci_tests.yml file, to build against one selected version of the TPL.

It must be mentioned that one and only one version of the compiled TPL tarball is stored per pull request (older ones are removed automatically). Therefore, a client building against a work in progress PR may experience a 404 error sooner or later.

Building docker images

Our continuous integration process builds the TPL and GEOS against two operating systems (ubuntu and centos) and two compilers (clang and gcc). The docker files use multi-stage builds in order to minimise the sizes of the images.

- First stage installs and defines all the elements that are commons to both TPL and GEOS (for example, MPI and c++ compiler, BLAS, LAPACK, path to the installation directory. . .).
- As a second stage, we install everything needed to build (*not run*) the TPLs. We keep nothing from this second step for GEOS, except the compiled TPL themselves. For example, a fortran compiler is needed by the TPL but not by GEOS: it shall be installed during this step, so GEOS won't access a fortran compiler (it does not have to).
- Last stage copies the compiled TPL from second stage and installs the elements only required by GEOS (there are few).

Docker images contract

GEOS will find a compiled version of the third party libraries.

As part of the contract provided by the TPL, the docker images also defines several environment variables. The

```
GEOSX_TPL_DIR
```

variable contains the absolute path of the installation root directory of the third party libraries. GEOS must use it when building.

Other variables are classical absolute path compiler variables.

```
CC
CXX
MPICC
MPICXX
```

And the absolute path the mpirun (or equivalent) command.

```
MPIEXEC
```

The following `openmpi` environment variables allow it to work properly in the docker container. But there should be no reason to access or use them explicitly.

```
OMPI_CC=$CC
OMPI_CXX=$CXX
```

1.10 Datastructure Index

1.10.1 Input Schema Definitions

`XML Schema`

Element: AcousticFirstOrderSEM

Name	Type	Default	Description
cflFactor	real64	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
discretization	string	require	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
dtSeismoTrace	real64	0	Time step for output pressure at receivers
enableLifo	integer	0	Set to 1 to enable LIFO storage feature
forward	integer	1	Set to 1 to compute forward propagation
initialDt	real64	1e+9	Initial time-step value required by the solver to the event manager.
lifoOnDevice	integer	-80	Set the capacity of the lifo device storage (if negative, opposite of percentage of remaining memory)
lifoOnHost	integer	-80	Set the capacity of the lifo host storage (if negative, opposite of percentage of remaining memory)
lifoSize	integer	2147	Set the capacity of the lifo storage (should be the total number of buffers to store in the LIFO)
linearDASGeometry	real64	{{0}	Geometry parameters for a linear DAS fiber (dip, azimuth, gauge length)
logLevel	integer	0	Log level
name	string	require	A name is required for any non-unique nodes
outputSeismoTrace	integer	0	Flag that indicates if we write the seismo trace in a file .txt, 0 no output, 1 otherwise
receiverCoordinates	real64	require	Coordinates (x,y,z) of the receivers
rickerOrd	integer	2	Flag that indicates the order of the Ricker to be used: {0:4}. Order 2 by default
saveFields	integer	0	Set to 1 to save fields during forward and restore them during backward
shotIndex	integer	0	Set the current shot for temporary files
sourceCoordinates	real64	require	Coordinates (x,y,z) of the sources

Element: AcousticSEM

Name	Type	Default	Description
cflFactor	real64	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
discretization	string	require	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
dtSeismoTrace	real64	0	Time step for output pressure at receivers
enableLifo	integer	0	Set to 1 to enable LIFO storage feature
forward	integer	1	Set to 1 to compute forward propagation
initialDt	real64	1e+9	Initial time-step value required by the solver to the event manager.
lifoOnDevice	integer	-80	Set the capacity of the lifo device storage (if negative, opposite of percentage of remaining memory)
lifoOnHost	integer	-80	Set the capacity of the lifo host storage (if negative, opposite of percentage of remaining memory)
lifoSize	integer	2147	Set the capacity of the lifo storage (should be the total number of buffers to store in the LIFO)
linearDASGeometry	real64	{{0}	Geometry parameters for a linear DAS fiber (dip, azimuth, gauge length)
logLevel	integer	0	Log level
name	string	require	A name is required for any non-unique nodes
outputSeismoTrace	integer	0	Flag that indicates if we write the seismo trace in a file .txt, 0 no output, 1 otherwise
receiverCoordinates	real64	require	Coordinates (x,y,z) of the receivers
rickerOrd	integer	2	Flag that indicates the order of the Ricker to be used: {0:4}. Order 2 by default
saveFields	integer	0	Set to 1 to save fields during forward and restore them during backward
shotIndex	integer	0	Set the current shot for temporary files
sourceCoordinates	real64	require	Coordinates (x,y,z) of the sources

Element: AcousticVTISEM

Name	Type	Default	Description
cflFactor	real64	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
discretization	string	require	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
dtSeismoTrace	real64	0	Time step for output pressure at receivers
forward	integer	1	Set to 1 to compute forward propagation
initialDt	real64	1e+9	Initial time-step value required by the solver to the event manager.
lifoOnDevice	integer	0	Set the capacity of the lifo device storage
lifoOnHost	integer	0	Set the capacity of the lifo host storage
lifoSize	integer	0	Set the capacity of the lifo storage
linearDASGeometry	real64	{{0}	Geometry parameters for a linear DAS fiber (dip, azimuth, gauge length)
logLevel	integer	0	Log level
name	string	require	A name is required for any non-unique nodes
outputSeismoTrace	integer	0	Flag that indicates if we write the seismo trace in a file .txt, 0 no output, 1 otherwise
receiverCoordinates	real64	require	Coordinates (x,y,z) of the receivers
rickerOrd	integer	2	Flag that indicates the order of the Ricker to be used: {0:4}. Order 2 by default
saveFields	integer	0	Set to 1 to save fields during forward and restore them during backward
shotIndex	integer	0	Set the current shot for temporary files
sourceCoordinates	real64	require	Coordinates (x,y,z) of the sources
targetRegions	string	require	Allowable regions that the solver may be applied to. Note that this does not indicate that the solver will be applied to these regions, only that allocation will occur such that the solver may be applied to these regions. The decision about what regions this solver will

Element: Aquifer

Name	Type	Default	Description
allowAllPhasesIntoAquifer	integer	0	Flag to allow all phases to flow into the aquifer. This flag only matters for the configuration in which flow is from reservoir to aquifer. - If the flag is equal to 1, then all phases, including non-aqueous phases, are allowed to flow into the aquifer. - If the flag is equal to 0, then only the water phase is allowed to flow into the aquifer. If you are in a configuration in which flow is from reservoir to aquifer and you expect non-aqueous phases to saturate the reservoir cells next to the aquifer, set this flag to 1. This keyword is ignored for single-phase flow simulations
aquiferAngle	real64	required	Angle subtended by the aquifer boundary from the center of the reservoir [degress]
aquiferElevation	real64	required	Aquifer elevation (positive going upward) [m]
aquiferInitialPressure	real64	required	Aquifer initial pressure [Pa]
aquiferInnerRadius	real64	required	Aquifer inner radius [m]
aquiferPermeability	real64	required	Aquifer permeability [m^2]
aquiferPorosity	real64	required	Aquifer porosity
aquiferThickness	real64	required	Aquifer thickness [m]
aquiferTotalCompressibility	real64	required	Aquifer total compressibility (rock and fluid) [Pa^-1]
aquiferWaterDensity	real64	required	Aquifer water density [kg.m^-3]
aquiferWaterPhaseComponentFraction	real64_array	{0}	Aquifer water phase component fraction. This keyword is ignored for single-phase flow simulations

Element: Benchmarks

Name	Type	Default	Description
crusher	node	unique	*Element: crusher*
lassen	node	unique	*Element: lassen*
quartz	node	unique	*Element: quartz*

Element: BiotPorosity

Name	Type	Default	Description
defaultReferencePorosity	real64	required	Default value of the reference porosity
defaultThermalExpansionCoefficient	real64	0	Default thermal expansion coefficient
grainBulkModulus	real64	required	Grain bulk modulus
name	string	required	A name is required for any non-unique nodes

Element: BlackOilFluid

Name	Type	Default	Description
componentMolarWeight	real64_array	required	Component molar weights
componentNames	string_array	{}	List of component names
hydrocarbonFormationVolFactorTableNames	string_array	{}	List of formation volume factor TableFunction names from the Functions block. The user must provide one TableFunction per hydrocarbon phase, in the order provided in “phaseNames”. For instance, if “oil” is before “gas” in “phaseNames”, the table order should be: oilTableName, gasTableName
hydrocarbonViscosityTableNames	string_array	{}	List of viscosity TableFunction names from the Functions block. The user must provide one TableFunction per hydrocarbon phase, in the order provided in “phaseNames”. For instance, if “oil” is before “gas” in “phaseNames”, the table order should be: oilTableName, gasTableName
name	string	required	A name is required for any non-unique nodes
phaseNames	string_array	required	List of fluid phases
surfaceDensities	real64_array	required	List of surface mass densities for each phase
tableFiles	path_array	{}	List of filenames with input PVT tables (one per phase)
waterCompressibility	real64	0	Water compressibility
waterFormationVolumeFactor	real64	0	Water formation volume factor
waterReferencePressure	real64	0	Water reference pressure
waterViscosity	real64	0	Water viscosity

Element: Blueprint

Name	Type	Default	Description
childDirectory	string		Child directory path
name	string	required	A name is required for any non-unique nodes
outputFullQuadratureData	integer	0	If true writes out data associated with every quadrature point.
parallelThreads	integer	1	Number of plot files.
plotLevel	geos_dataRepository_PlotL	1	Determines which fields to write.

Element: Box

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes
strike	real64	-90	The strike angle of the box
xMax	R1Tensor	required	Maximum (x,y,z) coordinates of the box
xMin	R1Tensor	required	Minimum (x,y,z) coordinates of the box

Element: BrooksCoreyBakerRelativePermeability

Name	Type	Default	Description
gasOilRelPermExponent	real64_array	{1}	Rel perm power law exponent for the pair (gas phase, oil phase) at residual water saturation The expected format is "{ gasExp, oilExp }", in that order
gasOilRelPermMaxValue	real64_array	{0}	Maximum rel perm value for the pair (gas phase, oil phase) at residual water saturation The expected format is "{ gasMax, oilMax }", in that order
name	string	required	A name is required for any non-unique nodes
phaseMinVolumeFraction	real64_array	{0}	Minimum volume fraction value for each phase
phaseNames	string_array	required	List of fluid phases
waterOilRelPermExponent	real64_array	{1}	Rel perm power law exponent for the pair (water phase, oil phase) at residual gas saturation The expected format is "{ waterExp, oilExp }", in that order
waterOilRelPermMaxValue	real64_array	{0}	Maximum rel perm value for the pair (water phase, oil phase) at residual gas saturation The expected format is "{ waterMax, oilMax }", in that order

Element: BrooksCoreyCapillaryPressure

Name	Type	Default	Description
capPressureEpsilon	real64	1e-06	Wetting-phase saturation at which the max cap. pressure is attained; used to avoid infinite cap. pressure values for saturations close to zero
name	string	required	A name is required for any non-unique nodes
phaseCapPressureExponentInv	real64_a	{2}	Inverse of capillary power law exponent for each phase
phaseEntryPressure	real64_a	{1}	Entry pressure value for each phase
phaseMinVolumeFraction	real64_a	{0}	Minimum volume fraction value for each phase
phaseNames	string_ar	required	List of fluid phases

Element: BrooksCoreyRelativePermeability

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes
phaseMinVolumeFraction	real64_array	{0}	Minimum volume fraction value for each phase
phaseNames	string_array	required	List of fluid phases
phaseRelPermExponent	real64_array	{1}	Minimum relative permeability power law exponent for each phase
phaseRelPermMaxValue	real64_array	{0}	Maximum relative permeability value for each phase

Element: CO2BrineEzrokhiFluid

Name	Type	Default	Description
componentMolarWeight	real64_array	{0}	Component molar weights
componentNames	string_array	{}	List of component names
flashModelParaFile	path	required	Name of the file defining the parameters of the flash model
name	string	required	A name is required for any non-unique nodes
phaseNames	string_array	{}	List of fluid phases
phasePVTParaFiles	path_array	required	Names of the files defining the parameters of the viscosity and density models

Element: CO2BrineEzrokhiThermalFluid

Name	Type	Default	Description
componentMolarWeight	real64_array	{0}	Component molar weights
componentNames	string_array	{}	List of component names
flashModelParaFile	path	required	Name of the file defining the parameters of the flash model
name	string	required	A name is required for any non-unique nodes
phaseNames	string_array	{}	List of fluid phases
phasePVTParaFiles	path_array	required	Names of the files defining the parameters of the viscosity and density models

Element: CO2BrinePhillipsFluid

Name	Type	Default	Description
componentMolarWeight	real64_array	{0}	Component molar weights
componentNames	string_array	{}	List of component names
flashModelParaFile	path	required	Name of the file defining the parameters of the flash model
name	string	required	A name is required for any non-unique nodes
phaseNames	string_array	{}	List of fluid phases
phasePVTParaFiles	path_array	required	Names of the files defining the parameters of the viscosity and density models

Element: CO2BrinePhillipsThermalFluid

Name	Type	Default	Description
componentMolarWeight	real64_array	{0}	Component molar weights
componentNames	string_array	{}	List of component names
flashModelParaFile	path	required	Name of the file defining the parameters of the flash model
name	string	required	A name is required for any non-unique nodes
phaseNames	string_array	{}	List of fluid phases
phasePVTParaFiles	path_array	required	Names of the files defining the parameters of the viscosity and density models

Element: CarmanKozenyPermeability

Name	Type	Default	Description
anisotropy	R1Tensor	{1,1,1}	Anisotropy factors for three permeability components.
name	string	required	A name is required for any non-unique nodes
particleDiameter	real64	required	Diameter of the spherical particles.
sphericity	real64	required	Sphericity of the particles.

Element: CellElementRegion

Name	Type	Default	Description
cellBlocks	string_array	{}	(no description available)
coarseningRatio	real64	0	(no description available)
materialList	string_array	required	List of materials present in this region
meshBody	string		Mesh body that contains this region
name	string	required	A name is required for any non-unique nodes

Element: ChomboIO

Name	Type	Default	Description
beginCycle	real64	required	Cycle at which the coupling will commence.
childDirectory	string		Child directory path
inputPath	string	/INVALID_INPUT_PA'	Path at which the chombo to geosx file will be written.
name	string	required	A name is required for any non-unique nodes
outputPath	string	required	Path at which the geosx to chombo file will be written.
parallelThreads	integer	1	Number of plot files.
useChomboPressures	integer	0	True iff geosx should use the pressures chombo writes out.
waitForInput	integer	required	True iff geosx should wait for chombo to write out a file. When true the inputPath must be set.

Element: CompositeFunction

Name	Type	Default	Description
expression	string		Composite math expression
functionNames	string_array	{}	List of source functions. The order must match the variableNames argument.
inputVarNames	string_array	{}	Name of fields are input to function.
name	string	required	A name is required for any non-unique nodes
variableNames	string_array	{}	List of variables in expression

Element: CompositionalMultiphaseFVM

Name	Type	Default	Description
allowLocalCompDensityChopping	integer	1	Flag indicating whether local (cell-wise) chopping of negative compositions is allowed
cflFactor	real64	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
discretization	string	required	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
initialDt	real64	1e+99	Initial time-step value required by the solver to the event manager.
isThermal	integer	0	Flag indicating whether the problem is thermal or not.
logLevel	integer	0	Log level
maxCompFractionChange	real64	0.5	Maximum (absolute) change in a component fraction in a Newton iteration
maxRelativePressureChange	real64	0.5	Maximum (relative) change in pressure in a Newton iteration (expected value between 0 and 1)
maxRelativeTemperatureChange	real64	0.5	Maximum (relative) change in temperature in a Newton iteration (expected value between 0 and 1)
name	string	required	A name is required for any non-unique nodes
scalingType	geos_CompositionalMultipl	Global	Solution scaling type.Valid options: * Global * Local
solutionChangeScalingFactor	real64	0.5	Damping factor for solution change targets
targetPhaseVolFraction-	real64	0.2	Target (absolute) change

Element: CompositionalMultiphaseFluid

Name	Type	Default	Description
componentAcentricFactor	real64_array	required	Component acentric factors
componentBinaryCoeff	real64_array2d	{{0}}	Table of binary interaction coefficients
componentCriticalPressure	real64_array	required	Component critical pressures
componentCriticalTemperature	real64_array	required	Component critical temperatures
componentMolarWeight	real64_array	required	Component molar weights
componentNames	string_array	required	List of component names
componentVolumeShift	real64_array	{0}	Component volume shifts
equationsOfState	string_array	required	List of equation of state types for each phase
name	string	required	A name is required for any non-unique nodes
phaseNames	string_array	required	List of fluid phases

Element: CompositionalMultiphaseHybridFVM

Name	Type	Default	Description
allowLocalCompDensityChopping	integer	1	Flag indicating whether local (cell-wise) chopping of negative compositions is allowed
cflFactor	real6	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
discretization	string	require	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
initialDt	real6	1e+9	Initial time-step value required by the solver to the event manager.
isThermal	integer	0	Flag indicating whether the problem is thermal or not.
logLevel	integer	0	Log level
maxCompFractionChange	real6	0.5	Maximum (absolute) change in a component fraction in a Newton iteration
maxRelativePressureChange	real6	0.5	Maximum (relative) change in pressure in a Newton iteration (expected value between 0 and 1)
maxRelativeTemperatureChange	real6	0.5	Maximum (relative) change in temperature in a Newton iteration (expected value between 0 and 1)
name	string	require	A name is required for any non-unique nodes
solutionChangeScalingFactor	real6	0.5	Damping factor for solution change targets
targetPhaseVolFractionChangeInTimeStep	real6	0.2	Target (absolute) change in phase volume fraction in a time step
targetRegions	string	require	Allowable regions that the solver may be applied to. Note that this does not indicate that the solver will be applied to these regions, only that allocation will occur such that the solver may be applied to these regions. The decision about what regions this solver will beapplied to rests in the EventManager.
targetRelativePressureChangeInTimeStep	real6	0.2	Target (relative) change in pressure in a time step (expected value between 0 and 1)
targetRelativeTemperatureChangeInTimeStep	real6	0.2	Target (relative) change in temperature in a time step (expected value between 0 and 1)
temperature	real6	require	Temperature
useMass	integer	0	Use mass formulation instead of molar

Element: CompositionalMultiphaseReservoir

Name	Type	Default	Description
cflFactor	real64	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
flowSolverName	string	require	Name of the flow solver used by the coupled solver
initialDt	real64	1e+99	Initial time-step value required by the solver to the event manager.
logLevel	integer	0	Log level
name	string	require	A name is required for any non-unique nodes
targetRegions	string	require	Allowable regions that the solver may be applied to. Note that this does not indicate that the solver will be applied to these regions, only that allocation will occur such that the solver may be applied to these regions. The decision about what regions this solver will beapplied to rests in the EventManager.
wellSolverName	string	require	Name of the well solver used by the coupled solver
LinearSolverParameters	node	uniqu	*Element: LinearSolverParameters*
NonlinearSolverParameters	node	uniqu	*Element: NonlinearSolverParameters*

Element: CompositionalMultiphaseStatistics

Name	Type	Default	Description
computeCFLNumbers	integer	0	Flag to decide whether CFL numbers are computed or not
computeRegionStatistics	integer	1	Flag to decide whether region statistics are computed or not
flowSolverName	string	required	Name of the flow solver
logLevel	integer	0	Log level
name	string	required	A name is required for any non-unique nodes
relpermThreshold	real6	1e-06	Flag to decide whether a phase is considered mobile (when the relperm is above the threshold) or immobile (when the relperm is below the threshold) in metric 2

Element: CompositionalMultiphaseWell

Name	Type	Default	Description
allowLocalCompDensityChopping	integer	1	Flag indicating whether local (cell-wise) chopping of negative compositions is allowed
cflFactor	real6	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
initialDt	real6	1e+9	Initial time-step value required by the solver to the event manager.
logLevel	integer	0	Log level
maxCompFractionChange	real6	1	Maximum (absolute) change in a component fraction between two Newton iterations
maxRelativePressureChange	real6	1	Maximum (relative) change in pressure between two Newton iterations (recommended with rate control)
name	string	require	A name is required for any non-unique nodes
targetRegions	string	require	Allowable regions that the solver may be applied to. Note that this does not indicate that the solver will be applied to these regions, only that allocation will occur such that the solver may be applied to these regions. The decision about what regions this solver will beapplied to rests in the EventManager.
useMass	integer	0	Use mass formulation instead of molar
LinearSolverParameters	node	uniqu	*Element: LinearSolverParameters*
NonlinearSolverParameters	node	uniqu	*Element: NonlinearSolverParameters*
WellControls	node		*Element: WellControls*

Element: CompressibleSinglePhaseFluid

Name	Type	Default	Description
compressibility	real64	0	Fluid compressibility
defaultDensity	real64	required	Default value for density.
defaultViscosity	real64	required	Default value for viscosity.
densityModelType	geos_constitutive_Exponent	linear	Type of density model. Valid options: * exponential * linear * quadratic
name	string	required	A name is required for any non-unique nodes
referenceDensity	real64	1000	Reference fluid density
referencePressure	real64	0	Reference pressure
referenceViscosity	real64	0.001	Reference fluid viscosity
viscosibility	real64	0	Fluid viscosity exponential coefficient
viscosityModelType	geos_constitutive_Exponent	linear	Type of viscosity model. Valid options: * exponential * linear * quadratic

Element: CompressibleSolidCarmanKozenyPermeability

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes
permeabilityModelName	string	required	Name of the permeability model.
porosityModelName	string	required	Name of the porosity model.
solidInternalEnergyModelName	string		Name of the solid internal energy model.
solidModelName	string	required	Name of the solid model.

Element: CompressibleSolidConstantPermeability

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes
permeabilityModelName	string	required	Name of the permeability model.
porosityModelName	string	required	Name of the porosity model.
solidInternalEnergyModelName	string		Name of the solid internal energy model.
solidModelName	string	required	Name of the solid model.

Element: CompressibleSolidExponentialDecayPermeability

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes
permeabilityModelName	string	required	Name of the permeability model.
porosityModelName	string	required	Name of the porosity model.
solidInternalEnergyModelName	string		Name of the solid internal energy model.
solidModelName	string	required	Name of the solid model.

Element: CompressibleSolidParallelPlatesPermeability

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes
permeabilityModelName	string	required	Name of the permeability model.
porosityModelName	string	required	Name of the porosity model.
solidInternalEnergyModelName	string		Name of the solid internal energy model.
solidModelName	string	required	Name of the solid model.

Element: CompressibleSolidSlipDependentPermeability

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes
permeabilityModelName	string	required	Name of the permeability model.
porosityModelName	string	required	Name of the porosity model.
solidInternalEnergyModelName	string		Name of the solid internal energy model.
solidModelName	string	required	Name of the solid model.

Element: CompressibleSolidWillisRichardsPermeability

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes
permeabilityModelName	string	required	Name of the permeability model.
porosityModelName	string	required	Name of the porosity model.
solidInternalEnergyModelName	string		Name of the solid internal energy model.
solidModelName	string	required	Name of the solid model.

Element: ConstantPermeability

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes
permeabilityComponents	R1Tensor	required	xx, yy and zz components of a diagonal permeability tensor.

Element: Constitutive

Name	Type	Default	Description
BiotPorosity	node		*Element: BiotPorosity*
BlackOilFluid	node		*Element: BlackOilFluid*
BrooksCoreyBakerRelativePermeability	node		*Element: BrooksCoreyBakerRelativePermeability*
BrooksCoreyCapillaryPressure	node		*Element: BrooksCoreyCapillaryPressure*
BrooksCoreyRelativePermeability	node		*Element: BrooksCoreyRelativePermeability*
CO2BrineEzrokhiFluid	node		*Element: CO2BrineEzrokhiFluid*
CO2BrineEzrokhiThermalFluid	node		*Element: CO2BrineEzrokhiThermalFluid*
CO2BrinePhillipsFluid	node		*Element: CO2BrinePhillipsFluid*
CO2BrinePhillipsThermalFluid	node		*Element: CO2BrinePhillipsThermalFluid*
CarmanKozenyPermeability	node		*Element: CarmanKozenyPermeability*
CompositionalMultiphaseFluid	node		*Element: CompositionalMultiphaseFluid*
CompressibleSinglePhaseFluid	node		*Element: CompressibleSinglePhaseFluid*
CompressibleSolidCarmanKozenyPermeability	node		*Element: CompressibleSolidCarmanKozenyPermeability*
CompressibleSolidConstantPermeability	node		*Element: CompressibleSolidConstantPermeability*
CompressibleSolidExponentialDecayPermeability	node		*Element: CompressibleSolidExponentialDecayPermeability*
CompressibleSolidParallelPlatesPermeability	node		*Element: CompressibleSolidParallelPlatesPermeability*
CompressibleSolidSlipDependentPermeability	node		*Element: CompressibleSolidSlipDependentPermeability*
CompressibleSolidWillisRichardsPermeability	node		*Element: CompressibleSolidWillisRichardsPermeability*
ConstantPermeability	node		*Element: ConstantPermeability*
Coulomb	node		*Element: Coulomb*
DamageElasticIsotropic	node		*Element: DamageElasticIsotropic*
DamageSpectralElasticIsotropic	node		*Element: DamageSpectralElasticIsotropic*
DamageVolDevElasticIsotropic	node		*Element: DamageVolDevElasticIsotropic*
DeadOilFluid	node		*Element: DeadOilFluid*
DelftEgg	node		*Element: DelftEgg*
DruckerPrager	node		*Element: DruckerPrager*
ElasticIsotropic	node		*Element: ElasticIsotropic*
ElasticIsotropicPressureDependent	node		*Element: ElasticIsotropicPressureDependent*

continues on next page

Table 1.2 – continued from previous page

Name	Type	Default	Description
ElasticOrthotropic	node		*Element: ElasticOrthotropic*
ElasticTransverseIsotropic	node		*Element: ElasticTransverseIsotropic*
ExponentialDecayPermeability	node		*Element: ExponentialDecayPermeability*
ExtendedDruckerPrager	node		*Element: ExtendedDruckerPrager*
FrictionlessContact	node		*Element: FrictionlessContact*
JFunctionCapillaryPressure	node		*Element: JFunctionCapillaryPressure*
ModifiedCamClay	node		*Element: ModifiedCamClay*
MultiPhaseConstantThermalConductivity	node		*Element: MultiPhaseConstantThermalConductivity*
MultiPhaseVolumeWeightedThermalConductivity	node		*Element: MultiPhaseVolumeWeightedThermalConductivity*
NullModel	node		*Element: NullModel*
ParallelPlatesPermeability	node		*Element: ParallelPlatesPermeability*
ParticleFluid	node		*Element: ParticleFluid*
PermeabilityBase	node		*Element: PermeabilityBase*
PorousDelftEgg	node		*Element: PorousDelftEgg*
PorousDruckerPrager	node		*Element: PorousDruckerPrager*
PorousElasticIsotropic	node		*Element: PorousElasticIsotropic*
PorousElasticOrthotropic	node		*Element: PorousElasticOrthotropic*
PorousElasticTransverseIsotropic	node		*Element: PorousElasticTransverseIsotropic*
PorousExtendedDruckerPrager	node		*Element: PorousExtendedDruckerPrager*
PorousModifiedCamClay	node		*Element: PorousModifiedCamClay*
PressurePorosity	node		*Element: PressurePorosity*
ProppantPermeability	node		*Element: ProppantPermeability*
ProppantPorosity	node		*Element: ProppantPorosity*
ProppantSlurryFluid	node		*Element: ProppantSlurryFluid*
ProppantSolidProppantPermeability	node		*Element: ProppantSolidProppantPermeability*
ReactiveBrine	node		*Element: ReactiveBrine*
ReactiveBrineThermal	node		*Element: ReactiveBrineThermal*
SinglePhaseConstantThermalConductivity	node		*Element: SinglePhaseConstantThermalConductivity*
SlipDependentPermeability	node		*Element: SlipDependentPermeability*
SolidInternalEnergy	node		*Element: SolidInternalEnergy*
TableCapillaryPressure	node		*Element: TableCapillaryPressure*
TableRelativePermeability	node		*Element: TableRelativePermeability*
TableRelativePermeabilityHysteresis	node		*Element: TableRelativePermeabilityHysteresis*
ThermalCompressibleSinglePhaseFluid	node		*Element: ThermalCompressibleSinglePhaseFluid*
VanGenuchtenBakerRelativePermeability	node		*Element: VanGenuchtenBakerRelativePermeability*
VanGenuchtenCapillaryPressure	node		*Element: VanGenuchtenCapillaryPressure*
ViscoDruckerPrager	node		*Element: ViscoDruckerPrager*
ViscoExtendedDruckerPrager	node		*Element: ViscoExtendedDruckerPrager*
ViscoModifiedCamClay	node		*Element: ViscoModifiedCamClay*
WillisRichardsPermeability	node		*Element: WillisRichardsPermeability*

Element: Coulomb

Name	Type	Default	Description
apertureTableName	string	required	Name of the aperture table
apertureTolerance	real64	1e-09	Value to be used to avoid floating point errors in expressions involving aperture. For example in the case of dividing by the actual aperture (not the effective aperture that results from the aperture function) this value may be used to avoid the 1/0 error. Note that this value may have some physical significance in its usage, as it may be used to smooth out highly nonlinear behavior associated with 1/0 in addition to avoiding the 1/0 error.
cohesion	real64	required	Cohesion
displacementJumpThreshold	real64	2.220 16	A threshold valued to determine whether a fracture is open or not.
frictionCoefficient	real64	required	Friction coefficient
name	string	required	A name is required for any non-unique nodes
penaltyStiffness	real64	0	Value of the penetration penalty stiffness. Units of Pressure/length
shearStiffness	real64	0	Value of the shear elastic stiffness. Units of Pressure/length

Element: CustomPolarObject

Name	Type	Default	Description
center	R1Tensoı	required	(x,y,z) coordinates of the center of the CustomPolarObject
coefficients	real64_aı	required	Coefficients of the CustomPolarObject function relating the localradius to the angle theta.
lengthVector	R1Tensoı	required	Tangent vector defining the orthonormal basis along with the normal.
name	string	required	A name is required for any non-unique nodes
normal	R1Tensoı	required	Normal (n_x,n_y,n_z) to the plane (will be normalized automatically)
tolerance	real64	1e-05	Tolerance to determine if a point sits on the CustomPolarObject or not. It is relative to the maximum dimension of the CustomPolarObject.
widthVector	R1Tensoı	required	Tangent vector defining the orthonormal basis along with the normal.

Element: Cylinder

Name	Type	Default	Description
firstFaceCenter	R1Tensor	required	Center point of the first face of the cylinder
innerRadius	real64	-1	Inner radius of the annulus
name	string	required	A name is required for any non-unique nodes
outerRadius	real64	required	Outer radius of the cylinder
secondFaceCenter	R1Tensor	required	Center point of the second face of the cylinder

Element: DamageElasticIsotropic

Name	Type	Default	Description
compressiveStrength	real64	0	Compressive strength from the uniaxial compression test
criticalFractureEnergy	real64	required	Critical fracture energy
criticalStrainEnergy	real64	required	Critical stress in a 1d tension test
defaultBulkModulus	real64	-1	Default Bulk Modulus Parameter
defaultDensity	real64	required	Default Material Density
defaultPoissonRatio	real64	-1	Default Poisson's Ratio
defaultShearModulus	real64	-1	Default Shear Modulus Parameter
defaultThermalExpansionCoefficient	real64	0	Default Linear Thermal Expansion Coefficient of the Solid Rock Frame
defaultYoungModulus	real64	-1	Default Young's Modulus
degradationLowerLimit	real64	0	The lower limit of the degradation function
deltaCoefficient	real64	-1	Coefficient in the calculation of the external driving force
extDrivingForceFlag	integer	0	Whether to have external driving force. Can be 0 or 1
lengthScale	real64	required	Length scale l in the phase-field equation
name	string	required	A name is required for any non-unique nodes
tensileStrength	real64	0	Tensile strength from the uniaxial tension test

Element: DamageSpectralElasticIsotropic

Name	Type	Default	Description
compressiveStrength	real64	0	Compressive strength from the uniaxial compression test
criticalFractureEnergy	real64	required	Critical fracture energy
criticalStrainEnergy	real64	required	Critical stress in a 1d tension test
defaultBulkModulus	real64	-1	Default Bulk Modulus Parameter
defaultDensity	real64	required	Default Material Density
defaultPoissonRatio	real64	-1	Default Poisson's Ratio
defaultShearModulus	real64	-1	Default Shear Modulus Parameter
defaultThermalExpansionCoefficient	real64	0	Default Linear Thermal Expansion Coefficient of the Solid Rock Frame
defaultYoungModulus	real64	-1	Default Young's Modulus
degradationLowerLimit	real64	0	The lower limit of the degradation function
deltaCoefficient	real64	-1	Coefficient in the calculation of the external driving force
extDrivingForceFlag	integer	0	Whether to have external driving force. Can be 0 or 1
lengthScale	real64	required	Length scale l in the phase-field equation
name	string	required	A name is required for any non-unique nodes
tensileStrength	real64	0	Tensile strength from the uniaxial tension test

Element: DamageVolDevElasticIsotropic

Name	Type	Default	Description
compressiveStrength	real64	0	Compressive strength from the uniaxial compression test
criticalFractureEnergy	real64	required	Critical fracture energy
criticalStrainEnergy	real64	required	Critical stress in a 1d tension test
defaultBulkModulus	real64	-1	Default Bulk Modulus Parameter
defaultDensity	real64	required	Default Material Density
defaultPoissonRatio	real64	-1	Default Poisson's Ratio
defaultShearModulus	real64	-1	Default Shear Modulus Parameter
defaultThermalExpansionCoefficient	real64	0	Default Linear Thermal Expansion Coefficient of the Solid Rock Frame
defaultYoungModulus	real64	-1	Default Young's Modulus
degradationLowerLimit	real64	0	The lower limit of the degradation function
deltaCoefficient	real64	-1	Coefficient in the calculation of the external driving force
extDrivingForceFlag	integer	0	Whether to have external driving force. Can be 0 or 1
lengthScale	real64	required	Length scale l in the phase-field equation
name	string	required	A name is required for any non-unique nodes
tensileStrength	real64	0	Tensile strength from the uniaxial tension test

Element: DeadOilFluid

Name	Type	Default	Description
componentMolarWeight	real64_array	required	Component molar weights
componentNames	string_array	{}	List of component names
hydrocarbonFormationVolFactorTableNames	string_array	{}	List of formation volume factor TableFunction names from the Functions block. The user must provide one TableFunction per hydrocarbon phase, in the order provided in "phaseNames". For instance, if "oil" is before "gas" in "phaseNames", the table order should be: oilTableName, gasTableName
hydrocarbonViscosityTableNames	string_array	{}	List of viscosity TableFunction names from the Functions block. The user must provide one TableFunction per hydrocarbon phase, in the order provided in "phaseNames". For instance, if "oil" is before "gas" in "phaseNames", the table order should be: oilTableName, gasTableName
name	string	required	A name is required for any non-unique nodes
phaseNames	string_array	required	List of fluid phases
surfaceDensities	real64_array	required	List of surface mass densities for each phase
tableFiles	path_array	{}	List of filenames with input PVT tables (one per phase)
waterCompressibility	real64	0	Water compressibility
waterFormationVolumeFactor	real64	0	Water formation volume factor
waterReferencePressure	real64	0	Water reference pressure
waterViscosity	real64	0	Water viscosity

Element: DelftEgg

Name	Type	Default	Description
defaultBulkModulus	real64	-1	Default Bulk Modulus Parameter
defaultCslSlope	real64	1	Slope of the critical state line
defaultDensity	real64	required	Default Material Density
defaultPoissonRatio	real64	-1	Default Poisson's Ratio
defaultPreConsolidationPressure	real64	-1.5	Initial preconsolidation pressure
defaultRecompressionIndex	real64	0.002	Recompresion Index
defaultShapeParameter	real64	1	Shape parameter for the yield surface
defaultShearModulus	real64	-1	Default Shear Modulus Parameter
defaultThermalExpansionCoefficient	real64	0	Default Linear Thermal Expansion Coefficient of the Solid Rock Frame
defaultVirginCompressionIndex	real64	0.005	Virgin compression index
defaultYoungModulus	real64	-1	Default Young's Modulus
name	string	required	A name is required for any non-unique nodes

Element: Dirichlet

Name	Type	Default	Description
bcApplicationTableName	string		Name of table that specifies the on/off application of the boundary condition.
beginTime	real64	-1e+99	Time at which the boundary condition will start being applied.
component	integer	-1	Component of field (if tensor) to apply boundary condition to.
direction	R1Tensor	{0,0,0}	Direction to apply boundary condition to.
endTime	real64	1e+99	Time at which the boundary condition will stop being applied.
fieldName	string		Name of field that boundary condition is applied to.
functionName	string		Name of function that specifies variation of the boundary condition.
initialCondition	integer	0	Boundary condition is applied as an initial condition.
logLevel	integer	0	Log level
name	string	required	A name is required for any non-unique nodes
objectPath	string		Path to the target field
scale	real64	0	Scale factor for value of the boundary condition.
setNames	string_array	required	Name of sets that boundary condition is applied to.

Element: Disc

Name	Type	Default	Description
center	R1Tenso	required	(x,y,z) coordinates of the center of the disc
lengthVector	R1Tenso	required	Tangent vector defining the orthonormal basis along with the normal.
name	string	required	A name is required for any non-unique nodes
normal	R1Tenso	required	Normal (n_x,n_y,n_z) to the plane (will be normalized automatically)
radius	real64	required	Radius of the disc.
tolerance	real64	1e-05	Tolerance to determine if a point sits on the disc or not. It is relative to the maximum dimension of the disc.
widthVector	R1Tenso	required	Tangent vector defining the orthonormal basis along with the normal.

Element: DruckerPrager

Name	Type	Default	Description
defaultBulkModulus	real64	-1	Default Bulk Modulus Parameter
defaultCohesion	real64	0	Initial cohesion
defaultDensity	real64	required	Default Material Density
defaultDilationAngle	real64	30	Dilation angle (degrees)
defaultFrictionAngle	real64	30	Friction angle (degrees)
defaultHardeningRate	real64	0	Cohesion hardening/softening rate
defaultPoissonRatio	real64	-1	Default Poisson's Ratio
defaultShearModulus	real64	-1	Default Shear Modulus Parameter
defaultThermalExpansionCoefficient	real64	0	Default Linear Thermal Expansion Coefficient of the Solid Rock Frame
defaultYoungModulus	real64	-1	Default Young's Modulus
name	string	required	A name is required for any non-unique nodes

Element: ElasticFirstOrderSEM

Name	Type	Default	Description
cflFactor	real64	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
discretization	string	require	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
dtSeismoTrace	real64	0	Time step for output pressure at receivers
enableLifo	integer	0	Set to 1 to enable LIFO storage feature
forward	integer	1	Set to 1 to compute forward propagation
initialDt	real64	1e+9	Initial time-step value required by the solver to the event manager.
lifoOnDevice	integer	-80	Set the capacity of the lifo device storage (if negative, opposite of percentage of remaining memory)
lifoOnHost	integer	-80	Set the capacity of the lifo host storage (if negative, opposite of percentage of remaining memory)
lifoSize	integer	2147	Set the capacity of the lifo storage (should be the total number of buffers to store in the LIFO)
linearDASGeometry	real64	{{0}	Geometry parameters for a linear DAS fiber (dip, azimuth, gauge length)
logLevel	integer	0	Log level
name	string	require	A name is required for any non-unique nodes
outputSeismoTrace	integer	0	Flag that indicates if we write the seismo trace in a file .txt, 0 no output, 1 otherwise
receiverCoordinates	real64	require	Coordinates (x,y,z) of the receivers
rickerOrd	integer	2	Flag that indicates the order of the Ricker to be used: {0:4}. Order 2 by default
saveFields	integer	0	Set to 1 to save fields during forward and restore them during backward
shotIndex	integer	0	Set the current shot for temporary files
sourceCoordinates	real64	require	Coordinates (x,y,z) of the sources

Element: ElasticIsotropic

Name	Type	Default	Description
defaultBulkModulus	real64	-1	Default Bulk Modulus Parameter
defaultDensity	real64	required	Default Material Density
defaultPoissonRatio	real64	-1	Default Poisson's Ratio
defaultShearModulus	real64	-1	Default Shear Modulus Parameter
defaultThermalExpansionCoefficient	real64	0	Default Linear Thermal Expansion Coefficient of the Solid Rock Frame
defaultYoungModulus	real64	-1	Default Young's Modulus
name	string	required	A name is required for any non-unique nodes

Element: ElasticIsotropicPressureDependent

Name	Type	Default	Description
defaultDensity	real64	required	Default Material Density
defaultRecompressionIndex	real64	0.002	Recompresion Index
defaultRefPressure	real64	-1	Reference Pressure
defaultRefStrainVol	real64	0	Reference Volumetric Strain
defaultShearModulus	real64	-1	Elastic Shear Modulus Parameter
defaultThermalExpansionCoefficient	real64	0	Default Linear Thermal Expansion Coefficient of the Solid Rock Frame
name	string	required	A name is required for any non-unique nodes

Element: ElasticOrthotropic

Name	Type	Default	Description
defaultC11	real64	-1	Default C11 Component of Voigt Stiffness Tensor
defaultC12	real64	-1	Default C12 Component of Voigt Stiffness Tensor
defaultC13	real64	-1	Default C13 Component of Voigt Stiffness Tensor
defaultC22	real64	-1	Default C22 Component of Voigt Stiffness Tensor
defaultC23	real64	-1	Default C23 Component of Voigt Stiffness Tensor
defaultC33	real64	-1	Default C33 Component of Voigt Stiffness Tensor
defaultC44	real64	-1	Default C44 Component of Voigt Stiffness Tensor
defaultC55	real64	-1	Default C55 Component of Voigt Stiffness Tensor
defaultC66	real64	-1	Default C66 Component of Voigt Stiffness Tensor
defaultDensity	real64	required	Default Material Density
defaultE1	real64	-1	Default Young’s Modulus E1
defaultE2	real64	-1	Default Young’s Modulus E2
defaultE3	real64	-1	Default Young’s Modulus E3
defaultG12	real64	-1	Default Shear Modulus G12
defaultG13	real64	-1	Default Shear Modulus G13
defaultG23	real64	-1	Default Shear Modulus G23
defaultNu12	real64	-1	Default Poission’s Ratio Nu12
defaultNu13	real64	-1	Default Poission’s Ratio Nu13
defaultNu23	real64	-1	Default Poission’s Ratio Nu23
defaultThermalExpansionCoefficient	real64	0	Default Linear Thermal Expansion Coefficient of the Solid Rock Frame
name	string	required	A name is required for any non-unique nodes

Element: ElasticSEM

Name	Type	Default	Description
cflFactor	real64	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
discretization	string	require	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
dtSeismoTrace	real64	0	Time step for output pressure at receivers
enableLifo	integer	0	Set to 1 to enable LIFO storage feature
forward	integer	1	Set to 1 to compute forward propagation
initialDt	real64	1e+9	Initial time-step value required by the solver to the event manager.
lifoOnDevice	integer	-80	Set the capacity of the lifo device storage (if negative, opposite of percentage of remaining memory)
lifoOnHost	integer	-80	Set the capacity of the lifo host storage (if negative, opposite of percentage of remaining memory)
lifoSize	integer	2147	Set the capacity of the lifo storage (should be the total number of buffers to store in the LIFO)
linearDASGeometry	real64	{{0}	Geometry parameters for a linear DAS fiber (dip, azimuth, gauge length)
logLevel	integer	0	Log level
name	string	require	A name is required for any non-unique nodes
outputSeismoTrace	integer	0	Flag that indicates if we write the seismo trace in a file .txt, 0 no output, 1 otherwise
receiverCoordinates	real64	require	Coordinates (x,y,z) of the receivers
rickerOrd	integer	2	Flag that indicates the order of the Ricker to be used: {0:4}. Order 2 by default
saveFields	integer	0	Set to 1 to save fields during forward and restore them during backward
shotIndex	integer	0	Set the current shot for temporary files
sourceCoordinates	real64	require	Coordinates (x,y,z) of the sources

Element: ElasticTransverseIsotropic

Name	Type	Default	Description
defaultC11	real64	-1	Default Stiffness Parameter C11
defaultC13	real64	-1	Default Stiffness Parameter C13
defaultC33	real64	-1	Default Stiffness Parameter C33
defaultC44	real64	-1	Default Stiffness Parameter C44
defaultC66	real64	-1	Default Stiffness Parameter C66
defaultDensity	real64	required	Default Material Density
defaultPoissonRatioAxialTransverse	real64	-1	Default Axial-Transverse Poisson's Ratio
defaultPoissonRatioTransverse	real64	-1	Default Transverse Poisson's Ratio
defaultShearModulusAxialTransverse	real64	-1	Default Axial-Transverse Shear Modulus
defaultThermalExpansionCoefficient	real64	0	Default Linear Thermal Expansion Coefficient of the Solid Rock Frame
defaultYoungModulusAxial	real64	-1	Default Axial Young's Modulus
defaultYoungModulusTransverse	real64	-1	Default Transverse Young's Modulus
name	string	required	A name is required for any non-unique nodes

Element: ElementRegions

Name	Type	Default	Description
CellElementRegion	node		*Element: CellElementRegion*
SurfaceElementRegion	node		*Element: SurfaceElementRegion*
WellElementRegion	node		*Element: WellElementRegion*

Element: EmbeddedSurfaceGenerator

Name	Type	Default	Description
cflFactor	real64	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
discretization	string	required	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
fractureRegion	string	FractureRegion	(no description available)
initialDt	real64	1e+99	Initial time-step value required by the solver to the event manager.
logLevel	integer	0	Log level
mpiCommOrder	integer	0	Flag to enable MPI consistent communication ordering
name	string	required	A name is required for any non-unique nodes
targetObjects	string	required	List of geometric objects that will be used to initialized the embedded surfaces/fractures.
targetRegions	string	required	Allowable regions that the solver may be applied to. Note that this does not indicate that the solver will be applied to these regions, only that allocation will occur such that the solver may be applied to these regions. The decision about what regions this solver will beapplied to rests in the EventManager.
LinearSolverParameters	node	unique	*Element: LinearSolverParameters*
NonlinearSolverParameters	node	unique	*Element: NonlinearSolverParameters*

Element: Events

Name	Type	Default	Description
logLevel	integer	0	Log level
maxCycle	integer	2147483647	Maximum simulation cycle for the global event loop.
maxTime	real64	1.79769e+30	Maximum simulation time for the global event loop.
minTime	real64	0	Start simulation time for the global event loop.
timeOutputFormat	geos_EventManager_TimeOutputF	seconds	Format of the time in the GEOS log.
HaltEvent	node		*Element: HaltEvent*
PeriodicEvent	node		*Element: PeriodicEvent*
SoloEvent	node		*Element: SoloEvent*

Element: ExponentialDecayPermeability

Name	Type	Default	Description
empiricalConstant	real64	required	an empirical constant.
initialPermeability	R1Tensor	required	initial permeability of the fracture.
name	string	required	A name is required for any non-unique nodes

Element: ExtendedDruckerPrager

Name	Type	Default	Description
defaultBulkModulus	real64	-1	Default Bulk Modulus Parameter
defaultCohesion	real64	0	Initial cohesion
defaultDensity	real64	required	Default Material Density
defaultDilationRatio	real64	1	Dilation ratio [0,1] (ratio = tan dilationAngle / tan frictionAngle)
defaultHardening	real64	0	Hardening parameter (hardening rate is faster for smaller values)
defaultInitialFrictionAngle	real64	30	Initial friction angle (degrees)
defaultPoissonRatio	real64	-1	Default Poisson’s Ratio
defaultResidualFrictionAngle	real64	30	Residual friction angle (degrees)
defaultShearModulus	real64	-1	Default Shear Modulus Parameter
defaultThermalExpansionCoefficient	real64	0	Default Linear Thermal Expansion Coefficient of the Solid Rock Frame
defaultYoungModulus	real64	-1	Default Young’s Modulus
name	string	required	A name is required for any non-unique nodes

Element: FieldSpecification

Name	Type	Default	Description
bcApplicationTableName	string		Name of table that specifies the on/off application of the boundary condition.
beginTime	real64	-1e+99	Time at which the boundary condition will start being applied.
component	integer	-1	Component of field (if tensor) to apply boundary condition to.
direction	R1Tensor	{0,0,0}	Direction to apply boundary condition to.
endTime	real64	1e+99	Time at which the boundary condition will stop being applied.
fieldName	string		Name of field that boundary condition is applied to.
functionName	string		Name of function that specifies variation of the boundary condition.
initialCondition	integer	0	Boundary condition is applied as an initial condition.
logLevel	integer	0	Log level
name	string	required	A name is required for any non-unique nodes
objectPath	string		Path to the target field
scale	real64	0	Scale factor for value of the boundary condition.
setNames	string_array	required	Name of sets that boundary condition is applied to.

Element: FieldSpecifications

Name	Type	Default	Description
Aquifer	node		*Element: Aquifer*
Dirichlet	node		*Element: Dirichlet*
FieldSpecification	node		*Element: FieldSpecification*
HydrostaticEquilibrium	node		*Element: HydrostaticEquilibrium*
PML	node		*Element: PML*
SourceFlux	node		*Element: SourceFlux*
Traction	node		*Element: Traction*

Element: File

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes

Element: FiniteElementSpace

Name	Type	Default	Description
formulation	string	default	Specifier to indicate any specialized formuations. For instance, one of the many enhanced assumed strain methods of the Hexahedron parent shape would be indicated here
name	string	required	A name is required for any non-unique nodes
order	integer	required	The order of the finite element basis.
useVirtualElements	integer	0	Specifier to indicate whether to force the use of VEM

Element: FiniteElements

Name	Type	Default	Description
FiniteElementSpace	node		*Element: FiniteElementSpace*
LinearSolverParameters	node	unique	*Element: LinearSolverParameters*
NonlinearSolverParameters	node	unique	*Element: NonlinearSolverParameters*

Element: FiniteVolume

Name	Type	Default	Description
HybridMimeticDiscretization	node		*Element: HybridMimeticDiscretization*
TwoPointFluxApproximation	node		*Element: TwoPointFluxApproximation*

Element: FlowProppantTransport

Name	Type	Default	Description
cflFactor	real64	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
flowSolverName	string	required	Name of the flow solver used by the coupled solver
initialDt	real64	1e+99	Initial time-step value required by the solver to the event manager.
logLevel	integer	0	Log level
name	string	required	A name is required for any non-unique nodes
proppantSolverName	string	required	Name of the proppant solver used by the coupled solver
targetRegions	string	required	Allowable regions that the solver may be applied to. Note that this does not indicate that the solver will be applied to these regions, only that allocation will occur such that the solver may be applied to these regions. The decision about what regions this solver will beapplied to rests in the EventManager.
LinearSolverParameters	node	unique	*Element: LinearSolverParameters*
NonlinearSolverParameters	node	unique	*Element: NonlinearSolverParameters*

Element: FrictionlessContact

Name	Type	Default	Description
apertureTab Name	string	require	Name of the aperture table
apertureTolerance	real6	1e-09	Value to be used to avoid floating point errors in expressions involving aperture. For example in the case of dividing by the actual aperture (not the effective aperture that results from the aperture function) this value may be used to avoid the 1/0 error. Note that this value may have some physical significance in its usage, as it may be used to smooth out highly nonlinear behavior associated with 1/0 in addition to avoiding the 1/0 error.
displacementJumpT old	real6	2.220 16	A threshold valued to determine whether a fracture is open or not.
name	string	require	A name is required for any non-unique nodes
penaltyStiffness	real6	0	Value of the penetration penalty stiffness. Units of Pressure/length
shearStiffness	real6	0	Value of the shear elastic stiffness. Units of Pressure/length

Element: Functions

Name	Type	Default	Description
CompositeFunction	node		*Element: CompositeFunction*
MultivariableTableFunction	node		*Element: MultivariableTableFunction*
SymbolicFunction	node		*Element: SymbolicFunction*
TableFunction	node		*Element: TableFunction*

Element: Geometry

Name	Type	Default	Description
Box	node		*Element: Box*
CustomPolarObject	node		*Element: CustomPolarObject*
Cylinder	node		*Element: Cylinder*
Disc	node		*Element: Disc*
Rectangle	node		*Element: Rectangle*
ThickPlane	node		*Element: ThickPlane*

Element: HaltEvent

Name	Type	Default	Description
beginTime	real64	0	Start time of this event.
endTime	real64	1e+100	End time of this event.
finalDtStretch	real64	0.001	Allow the final dt request for this event to grow by this percentage to match the endTime exactly.
forceDt	real64	-1	While active, this event will request this timestep value (ignoring any children/targets requests).
logLevel	integer	0	Log level
maxEventDt	real64	-1	While active, this event will request a timestep <= this value (depending upon any child/target requests).
maxRuntime	real64	required	The maximum allowable runtime for the job.
name	string	required	A name is required for any non-unique nodes
target	string		Name of the object to be executed when the event criteria are met.
targetExactStartStop	integer	1	If this option is set, the event will reduce its timestep requests to match any specified beginTime/endTimes exactly.
HaltEvent	node		*Element: HaltEvent*
PeriodicEvent	node		*Element: PeriodicEvent*
SoloEvent	node		*Element: SoloEvent*

Element: HybridMimeticDiscretization

Name	Type	Default	Description
innerProductType	string	required	Type of inner product used in the hybrid FVM solver
name	string	required	A name is required for any non-unique nodes

Element: Hydrofracture

Name	Type	Default	Description
cflFactor	real64	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
contactRelationName	string	required	Name of contact relation to enforce constraints on fracture boundary.
flowSolverName	string	required	Name of the flow solver used by the coupled solver
initialDt	real64	1e+99	Initial time-step value required by the solver to the event manager.
isMatrixPoroelastic	integer	0	(no description available)
isThermal	integer	0	Flag indicating whether the problem is thermal or not. Set isThermal="1" to enable the thermal coupling
logLevel	integer	0	Log level
maxNumResolves	integer	10	Value to indicate how many resolves may be executed to perform surface generation after the execution of flow and mechanics solver.
name	string	required	A name is required for any non-unique nodes
solidSolverName	string	required	Name of the solid solver used by the coupled solver
surfaceGeneratorName	string	required	Name of the surface generator to use in the hydrofracture solver
targetRegions	string	required	Allowable regions that the solver may be applied to. Note that this does not indicate that the solver will be applied to these regions, only that allocation will occur such that the solver may be applied to these regions. The decision about what regions this solver will beapplied to rests in the EventManager.
LinearSolverParameters	node	unique	*Element: LinearSolverParameters*
NonlinearSolverParameters	node	unique	*Element: NonlinearSolverParameters*

Element: HydrostaticEquilibrium

Name	Type	Default	Description
bcApplicationTableName	string		Name of table that specifies the on/off application of the boundary condition.
beginTime	real64	-1e+99	Time at which the boundary condition will start being applied.
componentFractionVsElevationTableNames	string_ar	{}	Names of the tables specifying the (component fraction vs elevation) relationship for each component
componentNames	string_ar	{}	Names of the fluid components
datumElevation	real64	required	Datum elevation [m]
datumPressure	real64	required	Datum pressure [Pa]
direction	R1Tenso	{0,0,0	Direction to apply boundary condition to.
elevationIncrementInHydrostaticPressureTable	real64	0.6096	Elevation increment [m] in the hydrostatic pressure table constructed internally
endTime	real64	1e+99	Time at which the boundary condition will stop being applied.
equilibrationTolerance	real64	0.001	Tolerance in the fixed-point iteration scheme used for hydrostatic initialization
functionName	string		Name of function that specifies variation of the boundary condition.
initialPhaseName	string		Name of the phase initially saturating the reservoir
logLevel	integer	0	Log level
maxNumberOfEquilibrationIterations	integer	5	Maximum number of equilibration iterations
name	string	required	A name is required for any non-unique nodes
objectPath	string		Path to the target field
scale	real64	0	Scale factor for value of the boundary condition.
temperatureVsElevationTableName	string		Name of the table specifying the (temperature [K] vs elevation) relationship

Element: Included

Name	Type	Default	Description
File	node		*Element: File*

Element: InternalMesh

Name	Type	Default	Description
cellBlockNames	string_arra	required	Names of each mesh block
elementTypes	string_arra	required	Element types of each mesh block
name	string	required	A name is required for any non-unique nodes
nx	integer_array	required	Number of elements in the x-direction within each mesh block
ny	integer_array	required	Number of elements in the y-direction within each mesh block
nz	integer_array	required	Number of elements in the z-direction within each mesh block
positionTolerance	real64	1e-10	A position tolerance to verify if a node belong to a nodeset
trianglePattern	integer	0	Pattern by which to decompose the hex mesh into wedges
xBias	real64_arra	{1}	Bias of element sizes in the x-direction within each mesh block (dx_left=(1+b)*L/N, dx_right=(1-b)*L/N)
xCoords	real64_arra	required	x-coordinates of each mesh block vertex
yBias	real64_arra	{1}	Bias of element sizes in the y-direction within each mesh block (dy_left=(1+b)*L/N, dx_right=(1-b)*L/N)
yCoords	real64_arra	required	y-coordinates of each mesh block vertex
zBias	real64_arra	{1}	Bias of element sizes in the z-direction within each mesh block (dz_left=(1+b)*L/N, dz_right=(1-b)*L/N)
zCoords	real64_arra	required	z-coordinates of each mesh block vertex
InternalWell	node		*Element: InternalWell*

Element: InternalWell

Name	Type	Default	Description
logLevel	integer	0	Log level
minElementLength	real64	0.001	Minimum length of a well element, computed as (segment length / number of elements per segment) [m]
minSegmentLength	real64	0.01	Minimum length of a well segment [m]
name	string	required	A name is required for any non-unique nodes
numElementsPerSegment	integer	required	Number of well elements per polyline segment
polylineNodeCoords	real64_array2d	required	Physical coordinates of the well polyline nodes
polylineSegmentConn	globalIndex_array2d	required	Connectivity of the polyline segments
radius	real64	required	Radius of the well [m]
wellControlsName	string	required	Name of the set of constraints associated with this well
wellRegionName	string	required	Name of the well element region
Perforation	node		*Element: Perforation*

Element: InternalWellbore

Name	Type	Default	Description
autoSpaceRadialElems	real64_	{-1}	Automatically set number and spacing of elements in the radial direction. This overrides the values of nr!Value in each block indicates factor to scale the radial increment.Larger numbers indicate larger radial elements.
cartesianMappingInnerRadius	real64	1e+9	If using a Cartesian aligned outer boundary, this is inner radius at which to start the mapping.
cellBlockNames	string_	require	Names of each mesh block
elementTypes	string_	require	Element types of each mesh block
hardRadialCoords	real64_	{0}	Sets the radial spacing to specified values
name	string	require	A name is required for any non-unique nodes
nr	integer_arr	require	Number of elements in the radial direction
nt	integer_arr	require	Number of elements in the tangent direction
nz	integer_arr	require	Number of elements in the z-direction within each mesh block
positionTolerance	real64	1e-10	A position tolerance to verify if a node belong to a nodeset
rBias	real64_	{-0.8}	Bias of element sizes in the radial direction
radius	real64_	require	Wellbore radius
theta	real64_	require	Tangent angle defining geometry size: 90 for quarter, 180 for half and 360 for full wellbore geometry
trajectory	real64_	{{0}	Coordinates defining the wellbore trajectory
trianglePattern	integer	0	Pattern by which to decompose the hex mesh into wedges
useCartesianOuterBoundary	integer	1000	Enforce a Cartesian aligned outer boundary on the outer block starting with the radial block specified in this value
xBias	real64_	{1}	Bias of element sizes in the x-direction within each mesh block (dx_left=(1+b)*L/N, dx_right=(1-b)*L/N)
yBias	real64_	{1}	Bias of element sizes in the y-direction within each mesh block (dy_left=(1+b)*L/N, dx_right=(1-b)*L/N)
zBias	real64_	{1}	Bias of element sizes in the z-direction within each mesh block (dz_left=(1+b)*L/N, dz_right=(1-b)*L/N)
zCoords	real64_	require	z-coordinates of each mesh block vertex
InternalWell	node		*Element: InternalWell*

Element: JFunctionCapillaryPressure

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes
nonWettingIntermediateJFunctionTableName	string		J-function table (dimensionless) for the pair (non-wetting phase, intermediate phase) Note that this input is only used for three-phase flow. If you want to do a two-phase simulation, please use instead wettingNonWettingJFunctionTableName to specify the table names.
nonWettingIntermediateSurfaceTension	real64	0	Surface tension [N/m] for the pair (non-wetting phase, intermediate phase) If you have a value in [dyne/cm], divide it by 1000 to obtain the value in [N/m] Note that this input is only used for three-phase flow. If you want to do a two-phase simulation, please use instead wettingNonWettingSurfaceTension to specify the surface tensions.
permeabilityDirection	geos_constitutive_JFunctior	required	Permeability direction. Options are: XY - use the average of the permeabilities in the x and y directions, X - only use the permeability in the x direction, Y - only use the permeability in the y direction, Z - only use the permeability in the z direction.
permeabilityExponent	real64	0.5	Permeability exponent
phaseNames	string_array	required	List of fluid phases
porosityExponent	real64	0.5	Porosity exponent

Element: LagrangianContact

Name	Type	Default	Description
cflFactor	real6·	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
contactRelationName	string	required	Name of contact relation to enforce constraints on fracture boundary.
discretization	string	required	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
fractureRegionName	string	required	Name of the fracture region.
initialDt	real6·	1e+9·	Initial time-step value required by the solver to the event manager.
logLevel	integer	0	Log level
name	string	required	A name is required for any non-unique nodes
solidSolverName	string	required	Name of the solid mechanics solver in the rock matrix
stabilizationName	string	required	Name of the stabilization to use in the lagrangian contact solver
targetRegions	string	required	Allowable regions that the solver may be applied to. Note that this does not indicate that the solver will be applied to these regions, only that allocation will occur such that the solver may be applied to these regions. The decision about what regions this solver will beapplied to rests in the EventManager.
LinearSolverParameters	node	uniqu	*Element: LinearSolverParameters*
NonlinearSolverParameters	node	uniqu	*Element: NonlinearSolverParameters*

Element: LaplaceFEM

Name	Type	Default	Description
cflFactor	real64	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
discretization	string	required	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
fieldName	string	required	Name of field variable
initialDt	real64	1e+99	Initial time-step value required by the solver to the event manager.
logLevel	integer	0	Log level
name	string	required	A name is required for any non-unique nodes
targetRegions	string_array	required	Allowable regions that the solver may be applied to. Note that this does not indicate that the solver will be applied to these regions, only that allocation will occur such that the solver may be applied to these regions. The decision about what regions this solver will beapplied to rests in the EventManager.
timeIntegrationOption	geos_LaplaceBaseH1_Time	required	Time integration method. Options are: * SteadyState * ImplicitTransient
LinearSolverParameters	node	unique	*Element: LinearSolverParameters*
NonlinearSolverParameters	node	unique	*Element: NonlinearSolverParameters*

Element: LinearSolverParameters

Name	Type	Default	Description
amgAggressiveCoarseningLevels	integer	0	AMG number of levels for aggressive coarsening
amgAggressiveCoarseningPaths	integer	1	AMG number of paths for aggressive coarsening
amgAggressiveInterpType	geos_LinearSolverParamete	multipass	AMG aggressive interpolation algorithm. Available options are: `default|extendedIStage2|standardStage2|extendedStage2|multipass|modifiedExtended|modifiedExtendedI|modifiedExtendedE|modifiedMultipass`
amgCoarseSolver	geos_LinearSolverParamete	direct	AMG coarsest level solver/smoother type. Available options are: `default|jacobi|l1jacobi|fgs|sgs|l1sgs|chebyshev|direct|bgs`
amgCoarseningType	geos_LinearSolverParamete	HMIS	AMG coarsening algorithm. Available options are: `default|CLJP|RugeStueben|Falgout|PMIS|HMIS`
amgInterpolationMaxNonZeros	integer	4	AMG interpolation maximum number of nonzeros per row
amgInterpolationType	geos_LinearSolverParamete	extendedI	AMG interpolation algorithm. Available options are: `default|modifiedClassical|direct|multipass|extendedI|standard|extended|directBAMG|modifiedExtended|modifiedExtendedI|modifiedExtendedE`
amgNullSpaceType	geos_LinearSolverParamete	constantModes	AMG near null space approximation. Available options are:`constantModes|rigidBodyModes`
amgNumFunctions	integer	1	AMG number of functions
amgNumSweeps	integer	1	AMG smoother sweeps

continues on next page

Table 1.3 – continued from previous page

Name	Type	Default	Description
amgRelaxWeight	real64	1	AMG relaxation factor for the smoother
amgSeparateComponents	integer	0	AMG apply separate component filter for multi-variable problems
amgSmootherType	geos_LinearSolverParamete	l1sgs	AMG smoother type. Available options are: `default|jacobi|l1jacobi|fgs|bgs|sgs|l1sgs|chebyshev|ilu0|ilut|ic0|ict`
amgThreshold	real64	0	AMG strength-of-connection threshold
directCheckResidual	integer	0	Whether to check the linear system solution residual
directColPerm	geos_LinearSolverParamete	metis	How to permute the columns. Available options are: `none|MMD_AtplusA|MMD_AtA|colAMD|metis|parmetis`
directEquil	integer	1	Whether to scale the rows and columns of the matrix
directIterRef	integer	1	Whether to perform iterative refinement
directParallel	integer	1	Whether to use a parallel solver (instead of a serial one)
directReplTinyPivot	integer	1	Whether to replace tiny pivots by sqrt(epsilon)*norm(A)
directRowPerm	geos_LinearSolverParamete	mc64	How to permute the rows. Available options are: `none|mc64`
iluFill	integer	0	ILU(K) fill factor
iluThreshold	real64	0	ILU(T) threshold factor
krylovAdaptiveTol	integer	0	Use Eisenstat-Walker adaptive linear tolerance
krylovMaxIter	integer	200	Maximum iterations allowed for an iterative solver
krylovMaxRestart	integer	200	Maximum iterations before restart (GMRES only)

continues on next page

Table 1.3 – continued from previous page

Name	Type	Default	Description
krylovTol	real64	1e-06	Relative convergence tolerance of the iterative method If the method converges, the iterative solution x_k is such that the relative residual norm satisfies: $\|b - Ax_k\|_2 <$ `krylovTol` * $\|b\|_2$
krylovWeakestTol	real64	0.001	Weakest-allowed tolerance for adaptive method
logLevel	integer	0	Log level
preconditionerType	geos_LinearSolverParamete	iluk	Preconditioner type. Available options are: `none|jacobi|l1jacobi|fgs|sgs|l1sgs|chebyshev|iluk|ilut|icc|ict|amg|mgr|block|direct|bgs`
solverType	geos_LinearSolverParamete	direct	Linear solver type. Available options are: `direct|cg|gmres|fgmres|bicgstab|preconditioner`
stopIfError	integer	1	Whether to stop the simulation if the linear solver reports an error

Element: Mesh

Name	Type	Default	Description
InternalMesh	node		*Element: InternalMesh*
InternalWellbore	node		*Element: InternalWellbore*
VTKMesh	node		*Element: VTKMesh*

Element: ModifiedCamClay

Name	Type	Default	Description
defaultCslSlope	real64	1	Slope of the critical state line
defaultDensity	real64	required	Default Material Density
defaultPreConsolidationPressure	real64	-1.5	Initial preconsolidation pressure
defaultRecompressionIndex	real64	0.002	Recompresion Index
defaultRefPressure	real64	-1	Reference Pressure
defaultRefStrainVol	real64	0	Reference Volumetric Strain
defaultShearModulus	real64	-1	Elastic Shear Modulus Parameter
defaultThermalExpansionCoefficient	real64	0	Default Linear Thermal Expansion Coefficient of the Solid Rock Frame
defaultVirginCompressionIndex	real64	0.005	Virgin compression index
name	string	required	A name is required for any non-unique nodes

Element: MultiPhaseConstantThermalConductivity

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes
phaseNames	string_arra	required	List of fluid phases
thermalConductivityComponents	R1Tensor	required	xx, yy, and zz components of a diagonal thermal conductivity tensor [J/(s.m.K)]

Element: MultiPhaseVolumeWeightedThermalConductivity

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes
phaseNames	string_arra	required	List of fluid phases
phaseThermalConductivity	real64_arr	required	Phase thermal conductivity [W/(m.K)]
rockThermalConductivityComponents	R1Tensor	required	xx, yy, and zz components of a diagonal rock thermal conductivity tensor [W/(m.K)]

Element: MultiphasePoromechanics

Name	Type	Default	Description
cflFactor	real64	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
flowSolverName	string	required	Name of the flow solver used by the coupled solver
initialDt	real64	1e+99	Initial time-step value required by the solver to the event manager.
isThermal	integer	0	Flag indicating whether the problem is thermal or not. Set isThermal=”1” to enable the thermal coupling
logLevel	integer	0	Log level
name	string	required	A name is required for any non-unique nodes
solidSolverName	string	required	Name of the solid solver used by the coupled solver
stabilizationMultiplier	real64	1	Constant multiplier of stabilization strength.
stabilizationRegionNames	string_array	{}	Regions where stabilization is applied.
stabilizationType	geos_MultiphasePoromecha	None	Stabilization type. Options are: None - Add no stabilization to mass equation, Global - Add stabilization to all faces, Local - Add stabilization only to interiors of macro elements.
targetRegions	string_array	required	Allowable regions that the solver may be applied to. Note that this does not indicate that the solver will be applied to these regions, only that allocation will occur such that the solver may be applied to these regions. The decision about what regions this solver will beapplied to rests in the EventManager.
LinearSolverParameters	node	unique	*Element: LinearSolverParameters*
NonlinearSolverParameters	node	unique	*Element: NonlinearSolverParameters*

Element: MultiphasePoromechanicsInitialization

Name	Type	Default	Description
logLevel	integer	0	Log level
name	string	required	A name is required for any non-unique nodes
performStressInitialization	integer	required	Flag to indicate that the solver is going to perform stress initialization
poromechanicsSolverName	string	required	Name of the poromechanics solver

Element: MultiphasePoromechanicsReservoir

Name	Type	Default	Description
cflFactor	real6	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
initialDt	real6	1e+9	Initial time-step value required by the solver to the event manager.
logLevel	integer	0	Log level
name	string	require	A name is required for any non-unique nodes
poromechanicsSolverName	string	require	Name of the poromechanics solver used by the coupled solver
targetRegions	string	require	Allowable regions that the solver may be applied to. Note that this does not indicate that the solver will be applied to these regions, only that allocation will occur such that the solver may be applied to these regions. The decision about what regions this solver will beapplied to rests in the EventManager.
wellSolverName	string	require	Name of the well solver used by the coupled solver
LinearSolverParameters	node	uniqu	*Element: LinearSolverParameters*
NonlinearSolverParameters	node	uniqu	*Element: NonlinearSolverParameters*

Element: MultivariableTableFunction

Name	Type	Default	Description
inputVarNames	string_array	{}	Name of fields are input to function.
name	string	required	A name is required for any non-unique nodes

Element: NonlinearSolverParameters

Name	Type	Default	Description
allowNonConverged	integer	0	Allow non-converged solution to be accepted. (i.e. exit from the Newton loop without achieving the desired tolerance)
couplingType	geos_NonlinearSolverParam	FullyImplicit	Type of coupling. Valid options: * FullyImplicit * Sequential
lineSearchAction	geos_NonlinearSolverParam	Attempt	How the line search is to be used. Options are: * None - Do not use line search. * Attempt - Use line search. Allow exit from line search without achieving smaller residual than starting residual. * Require - Use line search. If smaller residual than starting resdual is not achieved, cut time step.
lineSearchCutFactor	real64	0.5	Line search cut factor. For instance, a value of 0.5 will result in the effective application of the last solution by a factor of (0.5, 0.25, 0.125, …)
lineSearchInterpolationType	geos_NonlinearSolverParam	Linear	Strategy to cut the solution update during the line search. Options are: * Linear * Parabolic
lineSearchMaxCuts	integer	4	Maximum number of line search cuts.
logLevel	integer	0	Log level
maxAllowedResidualNorm	real64	1e+09	Maximum value of residual norm that is allowed in a Newton loop
maxNumConfigurationAttempts	integer	10	Max number of times that the configuration can be changed
maxSubSteps	integer	10	Maximum number of time sub-steps allowed for the solver
maxTimeStepCuts	integer	2	Max number of time step cuts

Element: NullModel

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes

Element: NumericalMethods

Name	Type	Default	Description
FiniteElements	node	unique	*Element: FiniteElements*
FiniteVolume	node	unique	*Element: FiniteVolume*

Element: Outputs

Name	Type	Default	Description
Blueprint	node		*Element: Blueprint*
ChomboIO	node		*Element: ChomboIO*
Python	node		*Element: Python*
Restart	node		*Element: Restart*
Silo	node		*Element: Silo*
TimeHistory	node		*Element: TimeHistory*
VTK	node		*Element: VTK*

Element: PML

Name	Type	Default	Description
bcApplicationTableName	string		Name of table that specifies the on/off application of the boundary condition.
beginTime	real64	-1e+99	Time at which the boundary condition will start being applied.
component	integer	-1	Component of field (if tensor) to apply boundary condition to.
direction	R1Tensc	{0,0,0}	Direction to apply boundary condition to.
endTime	real64	1e+99	Time at which the boundary condition will stop being applied.
functionName	string		Name of function that specifies variation of the boundary condition.
logLevel	integer	0	Log level
name	string	required	A name is required for any non-unique nodes
objectPath	string		Path to the target field
reflectivity	real32	0.001	Desired reflectivity of the PML region, used to compute the damping profile
scale	real64	0	Scale factor for value of the boundary condition.
setNames	string_ar	required	Name of sets that boundary condition is applied to.
thicknessMaxXYZ	R1Tensc	{-1,-1,-1}	Thickness of the PML region, at right, back, and bottom sides, used to compute the damping profile
thicknessMinXYZ	R1Tensc	{-1,-1,-1}	Thickness of the PML region, at left, front, and top sides, used to compute the damping profile
waveSpeedMaxXYZ	R1Tensc	{-1,-1,-1}	Wave speed in the PML, at right, back, and bottom sides, used to compute the damping profile
waveSpeedMinXYZ	R1Tensc	{-1,-1,-1}	Wave speed in the PML, at left, front, and top sides, used to compute the damping profile
xMax	R1Tensc	{3.40282e+38,3.40282e+38	Maximum (x,y,z) coordinates of the inner PML boundaries
xMin	R1Tensc	{-3.40282e+38,-3.40282e+38,-3.40282e+38}	Minimum (x,y,z) coordinates of the inner PML boundaries

Element: PVTDriver

Name	Type	Default	Description
baseline	path	none	Baseline file
feedComposition	real64_array	required	Feed composition array [mol fraction]
fluid	string	required	Fluid to test
logLevel	integer	0	Log level
name	string	required	A name is required for any non-unique nodes
output	string	none	Output file
pressureControl	string	required	Function controlling pressure time history
steps	integer	required	Number of load steps to take
temperatureControl	string	required	Function controlling temperature time history

Element: PackCollection

Name	Type	Default	Description
disableCoordCollection	integer	0	Whether or not to create coordinate meta-collectors if collected objects are mesh objects.
fieldName	string	required	The name of the (packable) field associated with the specified object to retrieve data from
name	string	required	A name is required for any non-unique nodes
objectPath	string	required	The name of the object from which to retrieve field values.
onlyOnSetChange	integer	0	Whether or not to only collect when the collected sets of indices change in any way.
setNames	string_arra	{}	The set(s) for which to retrieve data.

Element: ParallelPlatesPermeability

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes

Element: Parameter

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes
value	string	required	Input parameter definition for the preprocessor

Element: Parameters

Name	Type	Default	Description
Parameter	node		*Element: Parameter*

Element: ParticleFluid

Name	Type	Default	Description
collisionAlpha	real64	1.27	Collision alpha coefficient
collisionBeta	real64	1.5	Collision beta coefficient
fluidViscosity	real64	0.001	Fluid viscosity
hinderedSettlingCoefficient	real64	5.9	Hindered settling coefficient
isCollisionalSlip	integer	0	Whether the collisional component of the slip velocity is considered
maxProppantConcentration	real64	0.6	Max proppant concentration
name	string	required	A name is required for any non-unique nodes
particleSettlingModel	geos_constitutive_ParticleS	required	Particle settling velocity model. Valid options: * Stokes * Intermediate * Turbulence
proppantDensity	real64	1400	Proppant density
proppantDiameter	real64	0.0002	Proppant diameter
slipConcentration	real64	0.1	Slip concentration
sphericity	real64	1	Sphericity

Element: Perforation

Name	Type	Default	Description
distanceFromHead	real64	required	Linear distance from well head to the perforation
name	string	required	A name is required for any non-unique nodes
transmissibility	real64	-1	Perforation transmissibility

Element: PeriodicEvent

Name	Type	Default	Description
beginTime	real6	0	Start time of this event.
cycleFrequency	integer	1	Event application frequency (cycle, default)
endTime	real6	1e+1(	End time of this event.
finalDtStretch	real6	0.001	Allow the final dt request for this event to grow by this percentage to match the endTime exactly.
forceDt	real6	-1	While active, this event will request this timestep value (ignoring any children/targets requests).
function	string		Name of an optional function to evaluate when the time/cycle criteria are met.If the result is greater than the specified eventThreshold, the function will continue to execute.
logLevel	integer	0	Log level
maxEventDt	real6	-1	While active, this event will request a timestep <= this value (depending upon any child/target requests).
name	string	requirec	A name is required for any non-unique nodes
object	string		If the optional function requires an object as an input, specify its path here.
set	string		If the optional function is applied to an object, specify the setname to evaluate (default = everything).
stat	integer	0	If the optional function is applied to an object, specify the statistic to compare to the eventThreshold.The current options include: min, avg, and max.
target	string		Name of the object to be executed when the event criteria are met.
targetExactStartStop	integer	1	If this option is set, the event will reduce its timestep requests to match any specified beginTime/endTimes exactly.
targetExactTimestep	integer	1	If this option is set, the event will reduce its timestep requests to match the specified timeFrequency perfectly: dt_request = min(dt_request, t_last + time_frequency - time)).
threshold	real6	0	If the optional function is used, the event will execute if the value returned by the function exceeds this threshold.
timeFrequency	real6	-1	Event application frequency (time). Note: if this value is specified, it will override any cycle-based behavior.
HaltEvent	node		*Element: HaltEvent*
PeriodicEvent	node		*Element: PeriodicEvent*
SoloEvent	node		*Element: SoloEvent*

Element: PermeabilityBase

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes

Element: PhaseFieldDamageFEM

Name	Type	Default	Description
cflFactor	real6	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
damageUpperBound	real6	1.5	The upper bound of the damage
discretization	string	require	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
fieldName	string	require	name of field variable
initialDt	real6	1e+9	Initial time-step value required by the solver to the event manager.
irreversibilityFlag	integer	0	The flag to indicate whether to apply the irreversibility constraint
localDissipation	string	require	Type of local dissipation function. Can be Linear or Quadratic
logLevel	integer	0	Log level
name	string	require	A name is required for any non-unique nodes
targetRegions	string	require	Allowable regions that the solver may be applied to. Note that this does not indicate that the solver will be applied to these regions, only that allocation will occur such that the solver may be applied to these regions. The decision about what regions this solver will beapplied to rests in the EventManager.
timeIntegrationOption	string	require	option for default time integration method
LinearSolverParameters	node	uniqu	*Element: LinearSolverParameters*
NonlinearSolverParameters	node	uniqu	*Element: NonlinearSolverParameters*

Element: PhaseFieldFracture

Name	Type	Default	Description
cflFactor	real6	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
damageSolverName	string	required	Name of the damage solver used by the coupled solver
initialDt	real6	1e+9	Initial time-step value required by the solver to the event manager.
logLevel	integer	0	Log level
name	string	required	A name is required for any non-unique nodes
solidSolverName	string	required	Name of the solid solver used by the coupled solver
targetRegions	string	required	Allowable regions that the solver may be applied to. Note that this does not indicate that the solver will be applied to these regions, only that allocation will occur such that the solver may be applied to these regions. The decision about what regions this solver will beapplied to rests in the EventManager.
LinearSolverParameters	node	uniqu	*Element: LinearSolverParameters*
NonlinearSolverParameters	node	uniqu	*Element: NonlinearSolverParameters*

Element: PorousDelftEgg

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes
permeabilityModelName	string	required	Name of the permeability model.
porosityModelName	string	required	Name of the porosity model.
solidInternalEnergyModelName	string		Name of the solid internal energy model.
solidModelName	string	required	Name of the solid model.

Element: PorousDruckerPrager

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes
permeabilityModelName	string	required	Name of the permeability model.
porosityModelName	string	required	Name of the porosity model.
solidInternalEnergyModelName	string		Name of the solid internal energy model.
solidModelName	string	required	Name of the solid model.

Element: PorousElasticIsotropic

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes
permeabilityModelName	string	required	Name of the permeability model.
porosityModelName	string	required	Name of the porosity model.
solidInternalEnergyModelName	string		Name of the solid internal energy model.
solidModelName	string	required	Name of the solid model.

Element: PorousElasticOrthotropic

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes
permeabilityModelName	string	required	Name of the permeability model.
porosityModelName	string	required	Name of the porosity model.
solidInternalEnergyModelName	string		Name of the solid internal energy model.
solidModelName	string	required	Name of the solid model.

Element: PorousElasticTransverseIsotropic

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes
permeabilityModelName	string	required	Name of the permeability model.
porosityModelName	string	required	Name of the porosity model.
solidInternalEnergyModelName	string		Name of the solid internal energy model.
solidModelName	string	required	Name of the solid model.

Element: PorousExtendedDruckerPrager

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes
permeabilityModelName	string	required	Name of the permeability model.
porosityModelName	string	required	Name of the porosity model.
solidInternalEnergyModelName	string		Name of the solid internal energy model.
solidModelName	string	required	Name of the solid model.

Element: PorousModifiedCamClay

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes
permeabilityModelName	string	required	Name of the permeability model.
porosityModelName	string	required	Name of the porosity model.
solidInternalEnergyModelName	string		Name of the solid internal energy model.
solidModelName	string	required	Name of the solid model.

Element: PressurePorosity

Name	Type	Default	Description
compressibility	real64	required	Solid compressibility
defaultReferencePorosity	real64	required	Default value of the reference porosity
name	string	required	A name is required for any non-unique nodes
referencePressure	real64	required	Reference pressure for solid compressibility

Element: Problem

Name	Type	Default	Description
Benchmarks	node	unique	*Element: Benchmarks*
Constitutive	node	unique	*Element: Constitutive*
ElementRegions	node	unique	*Element: ElementRegions*
Events	node	unique, required	*Element: Events*
FieldSpecifications	node	unique	*Element: FieldSpecifications*
Functions	node	unique	*Element: Functions*
Geometry	node	unique	*Element: Geometry*
Included	node	unique	*Element: Included*
Mesh	node	unique, required	*Element: Mesh*
NumericalMethods	node	unique	*Element: NumericalMethods*
Outputs	node	unique, required	*Element: Outputs*
Parameters	node	unique	*Element: Parameters*
Solvers	node	unique, required	*Element: Solvers*
Tasks	node	unique	*Element: Tasks*

Element: ProppantPermeability

Name	Type	Default	Description
maxProppantConcentration	real64	required	Maximum proppant concentration.
name	string	required	A name is required for any non-unique nodes
proppantDiameter	real64	required	Proppant diameter.

Element: ProppantPorosity

Name	Type	Default	Description
defaultReferencePorosity	real64	required	Default value of the reference porosity
maxProppantConcentration	real64	required	Maximum proppant concentration
name	string	required	A name is required for any non-unique nodes

Element: ProppantSlurryFluid

Name	Type	Default	Description
componentNames	string_array	{}	List of fluid component names
compressibility	real64	0	Fluid compressibility
defaultComponentDensity	real64_array	{0}	Default value for the component density.
defaultComponentViscosity	real64_array	{0}	Default value for the component viscosity.
defaultCompressibility	real64_array	{0}	Default value for the component compressibility.
flowBehaviorIndex	real64_array	{0}	Flow behavior index
flowConsistencyIndex	real64_array	{0}	Flow consistency index
maxProppantConcentration	real64	0.6	Maximum proppant concentration
name	string	required	A name is required for any non-unique nodes
referenceDensity	real64	1000	Reference fluid density
referencePressure	real64	100000	Reference pressure
referenceProppantDensity	real64	1400	Reference proppant density
referenceViscosity	real64	0.001	Reference fluid viscosity

Element: ProppantSolidProppantPermeability

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes
permeabilityModelName	string	required	Name of the permeability model.
porosityModelName	string	required	Name of the porosity model.
solidInternalEnergyModelName	string		Name of the solid internal energy model.
solidModelName	string	required	Name of the solid model.

Element: ProppantTransport

Name	Type	Default	Description
bridgingFactor	real64	0	Bridging factor used for bridging/screen-out calculation
cflFactor	real64	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
criticalShieldsNumber	real64	0	Critical Shields number
discretization	string	required	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
frictionCoefficient	real64	0.03	Friction coefficient
initialDt	real64	1e+99	Initial time-step value required by the solver to the event manager.
isThermal	integer	0	Flag indicating whether the problem is thermal or not.
logLevel	integer	0	Log level
maxProppantConcentration	real64	0.6	Maximum proppant concentration
name	string	required	A name is required for any non-unique nodes
proppantDensity	real64	2500	Proppant density
proppantDiameter	real64	0.000	Proppant diameter
targetRegions	string	required	Allowable regions that the solver may be applied to. Note that this does not indicate that the solver will be applied to these regions, only that allocation will occur such that the solver may be applied to these regions. The decision about what regions this solver will beapplied to rests in the EventManager.
updateProppantPacking	integer	0	Flag that enables/disables proppant-packing update
LinearSolverParameters	node	uniqu	*Element: LinearSolverParameters*
NonlinearSolverParam-	node	uniqu	*Element: NonlinearSolverParameters*

Element: Python

Name	Type	Default	Description
childDirectory	string		Child directory path
name	string	required	A name is required for any non-unique nodes
parallelThreads	integer	1	Number of plot files.

Element: ReactiveBrine

Name	Type	Default	Description
componentMolarWeight	real64_array	{0}	Component molar weights
componentNames	string_array	{}	List of component names
name	string	required	A name is required for any non-unique nodes
phaseNames	string_array	{}	List of fluid phases
phasePVTParaFiles	path_array	required	Names of the files defining the parameters of the viscosity and density models

Element: ReactiveBrineThermal

Name	Type	Default	Description
componentMolarWeight	real64_array	{0}	Component molar weights
componentNames	string_array	{}	List of component names
name	string	required	A name is required for any non-unique nodes
phaseNames	string_array	{}	List of fluid phases
phasePVTParaFiles	path_array	required	Names of the files defining the parameters of the viscosity and density models

Element: ReactiveCompositionalMultiphaseOBL

Name	Type	Default	Description
OBLOperatorsTableFile	path	require	File containing OBL operator values
allowLocalChopping	integer	1	Allow keeping solution within OBL limits
cflFactor	real6	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
componentNames	string	{}	List of component names
discretization	string	require	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
enableEnergyBalance	integer	require	Enable energy balance calculation and temperature degree of freedom
initialDt	real6	1e+9	Initial time-step value required by the solver to the event manager.
isThermal	integer	0	Flag indicating whether the problem is thermal or not.
logLevel	integer	0	Log level
maxCompFrationChange	real6	1	Maximum (absolute) change in a component fraction between two Newton iterations
name	string	require	A name is required for any non-unique nodes
numComponents	integer	require	Number of components
numPhas	integer	require	Number of phases
phaseNames	string	{}	List of fluid phases
targetRegions	string	require	Allowable regions that the solver may be applied to. Note that this does not indicate that the solver will be applied to these regions, only that allocation will occur such that the solver may be applied to these regions. The decision about what regions this solver will beapplied to rests in the EventManager.
transMultExp	real6	1	Exponent of dynamic transmissibility multiplier
useDART	integer	1	Use L2 norm calculation similar to one used DARTS
Linear-	node	uniqu	*Element: LinearSolverParameters*

Element: ReactiveFluidDriver

Name	Type	Default	Description
baseline	path	none	Baseline file
feedComposition	real64_array	required	Feed composition array: total concentration of the primary species
fluid	string	required	Fluid to test
logLevel	integer	0	Log level
name	string	required	A name is required for any non-unique nodes
output	string	none	Output file
pressureControl	string	required	Function controlling pressure time history
steps	integer	required	Number of load steps to take
temperatureControl	string	required	Function controlling temperature time history

Element: Rectangle

Name	Type	Default	Description
dimensions	real64_arra	required	Length and width of the bounded plane
lengthVector	R1Tensor	required	Tangent vector defining the orthonormal basis along with the normal.
name	string	required	A name is required for any non-unique nodes
normal	R1Tensor	required	Normal (n_x,n_y,n_z) to the plane (will be normalized automatically)
origin	R1Tensor	required	Origin point (x,y,z) of the plane (basically, any point on the plane)
tolerance	real64	1e-05	Tolerance to determine if a point sits on the plane or not. It is relative to the maximum dimension of the plane.
widthVector	R1Tensor	required	Tangent vector defining the orthonormal basis along with the normal.

Element: RelpermDriver

Name	Type	Default	Description
baseline	path	none	Baseline file
logLevel	integer	0	Log level
name	string	required	A name is required for any non-unique nodes
output	string	none	Output file
relperm	string	required	Relperm model to test
steps	integer	required	Number of saturation steps to take

Element: Restart

Name	Type	Default	Description
childDirectory	string		Child directory path
name	string	required	A name is required for any non-unique nodes
parallelThreads	integer	1	Number of plot files.

Element: Run

Name	Type	Default	Description
args	string		Any extra command line arguments to pass to GEOSX.
autoPartition	string		May be ‘Off’ or ‘On’, if ‘On’ partitioning arguments are created automatically. Default is Off.
meshSizes	integer_array	{0}	The target number of elements in the internal mesh (per-process for weak scaling, globally for strong scaling) default doesn’t modify the internalMesh.
name	string	required	The name of this benchmark.
nodes	integer	0	The number of nodes needed to run the base benchmark, default is 1.
scaleList	integer_array	{0}	The scales at which to run the problem (scale * nodes * tasksPerNode).
scaling	string		Whether to run a scaling, and which type of scaling to run.
tasksPerNode	integer	required	The number of tasks per node to run the benchmark with.
threadsPerTask	integer	0	The number of threads per task to run the benchmark with.
timeLimit	integer	0	The time limit of the benchmark.

Element: Silo

Name	Type	Default	Description
childDirectory	string		Child directory path
fieldNames	string_a	{}	Names of the fields to output. If this attribute is specified, GEOSX outputs all (and only) the fields specified by the user, regardless of their plotLevel
name	string	required	A name is required for any non-unique nodes
onlyPlotSpecifiedFieldNames	integer	0	If this flag is equal to 1, then we only plot the fields listed in *fieldNames*. Otherwise, we plot all the fields with the required *plotLevel*, plus the fields listed in *fieldNames*
parallelThreads	integer	1	Number of plot files.
plotFileRoot	string	plot	(no description available)
plotLevel	integer	1	(no description available)
writeCellElementMesh	integer	1	(no description available)
writeEdgeMesh	integer	0	(no description available)
writeFEMFaces	integer	0	(no description available)
writeFaceElementMesh	integer	1	(no description available)

Element: SinglePhaseConstantThermalConductivity

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes
thermalConductivityComponents	R1Tensor	required	xx, yy, and zz components of a diagonal thermal conductivity tensor [J/(s.m.K)]

Element: SinglePhaseFVM

Name	Type	Default	Description
cflFactor	real6	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
discretization	string	require	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
initialDt	real6	1e+9	Initial time-step value required by the solver to the event manager.
isThermal	integer	0	Flag indicating whether the problem is thermal or not.
logLevel	integer	0	Log level
name	string	require	A name is required for any non-unique nodes
targetRegions	string	require	Allowable regions that the solver may be applied to. Note that this does not indicate that the solver will be applied to these regions, only that allocation will occur such that the solver may be applied to these regions. The decision about what regions this solver will beapplied to rests in the EventManager.
temperature	real6	0	Temperature
LinearSolverParameters	node	uniqu	*Element: LinearSolverParameters*
NonlinearSolverParameters	node	uniqu	*Element: NonlinearSolverParameters*

Element: SinglePhaseHybridFVM

Name	Type	Default	Description
cflFactor	real6	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
discretization	string	require	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
initialDt	real6	1e+9	Initial time-step value required by the solver to the event manager.
isThermal	integer	0	Flag indicating whether the problem is thermal or not.
logLevel	integer	0	Log level
name	string	require	A name is required for any non-unique nodes
targetRegions	string	require	Allowable regions that the solver may be applied to. Note that this does not indicate that the solver will be applied to these regions, only that allocation will occur such that the solver may be applied to these regions. The decision about what regions this solver will beapplied to rests in the EventManager.
temperature	real6	0	Temperature
LinearSolverParameters	node	uniqu	*Element: LinearSolverParameters*
NonlinearSolverParameters	node	uniqu	*Element: NonlinearSolverParameters*

Element: SinglePhasePoromechanics

Name	Type	Default	Description
cflFactor	real6	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
flowSolverName	string	required	Name of the flow solver used by the coupled solver
initialDt	real6	1e+9	Initial time-step value required by the solver to the event manager.
isThermal	integer	0	Flag indicating whether the problem is thermal or not. Set isThermal=”1” to enable the thermal coupling
logLevel	integer	0	Log level
name	string	required	A name is required for any non-unique nodes
solidSolverName	string	required	Name of the solid solver used by the coupled solver
targetRegions	string	required	Allowable regions that the solver may be applied to. Note that this does not indicate that the solver will be applied to these regions, only that allocation will occur such that the solver may be applied to these regions. The decision about what regions this solver will beapplied to rests in the EventManager.
LinearSolverParameters	node	uniqu	*Element: LinearSolverParameters*
NonlinearSolverParameters	node	uniqu	*Element: NonlinearSolverParameters*

Element: SinglePhasePoromechanicsConformingFractures

Name	Type	Default	Description
LagrangianContactSolverName	string	required	Name of the LagrangianContact solver used by the coupled solver
cflFactor	real64	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
initialDt	real64	1e+99	Initial time-step value required by the solver to the event manager.
isThermal	integer	0	Flag indicating whether the problem is thermal or not. Set isThermal="1" to enable the thermal coupling
logLevel	integer	0	Log level
name	string	required	A name is required for any non-unique nodes
poromechanicsSolverName	string	required	Name of the poromechanics solver used by the coupled solver
targetRegions	string	required	Allowable regions that the solver may be applied to. Note that this does not indicate that the solver will be applied to these regions, only that allocation will occur such that the solver may be applied to these regions. The decision about what regions this solver will beapplied to rests in the EventManager.
LinearSolverParameters	node	unique	*Element: LinearSolverParameters*
NonlinearSolverParameters	node	unique	*Element: NonlinearSolverParameters*

Element: SinglePhasePoromechanicsEmbeddedFractures

Name	Type	Default	Description
cflFactor	real6	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
flowSolverName	string	require	Name of the flow solver used by the coupled solver
fracturesSolverName	string	require	Name of the fractures solver to use in the fractured poroelastic solver
initialDt	real6	1e+9	Initial time-step value required by the solver to the event manager.
isThermal	integer	0	Flag indicating whether the problem is thermal or not. Set isThermal="1" to enable the thermal coupling
logLevel	integer	0	Log level
name	string	require	A name is required for any non-unique nodes
solidSolverName	string	require	Name of the solid solver used by the coupled solver
targetRegions	string	require	Allowable regions that the solver may be applied to. Note that this does not indicate that the solver will be applied to these regions, only that allocation will occur such that the solver may be applied to these regions. The decision about what regions this solver will beapplied to rests in the EventManager.
LinearSolverParameters	node	uniqu	*Element: LinearSolverParameters*
NonlinearSolverParameters	node	uniqu	*Element: NonlinearSolverParameters*

Element: SinglePhasePoromechanicsInitialization

Name	Type	Default	Description
logLevel	integer	0	Log level
name	string	required	A name is required for any non-unique nodes
performStressInitialization	integer	required	Flag to indicate that the solver is going to perform stress initialization
poromechanicsSolverName	string	required	Name of the poromechanics solver

Element: SinglePhasePoromechanicsReservoir

Name	Type	Default	Description
cflFactor	real6	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
initialDt	real6	1e+9	Initial time-step value required by the solver to the event manager.
logLevel	integer	0	Log level
name	string	require	A name is required for any non-unique nodes
poromechanicsSolverName	string	require	Name of the poromechanics solver used by the coupled solver
targetRegions	string	require	Allowable regions that the solver may be applied to. Note that this does not indicate that the solver will be applied to these regions, only that allocation will occur such that the solver may be applied to these regions. The decision about what regions this solver will beapplied to rests in the EventManager.
wellSolverName	string	require	Name of the well solver used by the coupled solver
LinearSolverParameters	node	uniqu	*Element: LinearSolverParameters*
NonlinearSolverParameters	node	uniqu	*Element: NonlinearSolverParameters*

Element: SinglePhaseProppantFVM

Name	Type	Default	Description
cflFactor	real6	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
discretization	string	require	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
initialDt	real6	1e+9	Initial time-step value required by the solver to the event manager.
isThermal	integer	0	Flag indicating whether the problem is thermal or not.
logLevel	integer	0	Log level
name	string	require	A name is required for any non-unique nodes
targetRegions	string	require	Allowable regions that the solver may be applied to. Note that this does not indicate that the solver will be applied to these regions, only that allocation will occur such that the solver may be applied to these regions. The decision about what regions this solver will beapplied to rests in the EventManager.
temperature	real6	0	Temperature
LinearSolverParameters	node	uniqu	*Element: LinearSolverParameters*
NonlinearSolverParameters	node	uniqu	*Element: NonlinearSolverParameters*

Element: SinglePhaseReservoir

Name	Type	Default	Description
cflFactor	real64	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
flowSolverName	string	required	Name of the flow solver used by the coupled solver
initialDt	real64	1e+99	Initial time-step value required by the solver to the event manager.
logLevel	integer	0	Log level
name	string	required	A name is required for any non-unique nodes
targetRegions	string	required	Allowable regions that the solver may be applied to. Note that this does not indicate that the solver will be applied to these regions, only that allocation will occur such that the solver may be applied to these regions. The decision about what regions this solver will beapplied to rests in the EventManager.
wellSolverName	string	required	Name of the well solver used by the coupled solver
LinearSolverParameters	node	unique	*Element: LinearSolverParameters*
NonlinearSolverParameters	node	unique	*Element: NonlinearSolverParameters*

Element: SinglePhaseStatistics

Name	Type	Default	Description
flowSolverName	string	required	Name of the flow solver
logLevel	integer	0	Log level
name	string	required	A name is required for any non-unique nodes

Element: SinglePhaseWell

Name	Type	Default	Description
cflFactor	real6	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
initialDt	real6	1e+9	Initial time-step value required by the solver to the event manager.
logLevel	integer	0	Log level
name	string	require	A name is required for any non-unique nodes
targetRegions	string	require	Allowable regions that the solver may be applied to. Note that this does not indicate that the solver will be applied to these regions, only that allocation will occur such that the solver may be applied to these regions. The decision about what regions this solver will beapplied to rests in the EventManager.
LinearSolverParameters	node	uniqu	*Element: LinearSolverParameters*
NonlinearSolverParameters	node	uniqu	*Element: NonlinearSolverParameters*
WellControls	node		*Element: WellControls*

Element: SlipDependentPermeability

Name	Type	Default	Description
initialPermeability	R1Tensor	required	initial permeability of the fracture.
maxPermMultiplier	real64	required	Maximum permeability multiplier.
name	string	required	A name is required for any non-unique nodes
shearDispThreshold	real64	required	Threshold of shear displacement.

Element: SolidInternalEnergy

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes
referenceInternalEnergy	real64	required	Internal energy at the reference temperature [J/kg]
referenceTemperature	real64	required	Reference temperature [K]
volumetricHeatCapacity	real64	required	Solid volumetric heat capacity [J/(kg.K)]

Element: SolidMechanicsEmbeddedFractures

Name	Type	Default	Description
cflFactor	real6	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
contactRelationName	string	require	Name of contact relation to enforce constraints on fracture boundary.
fractureRegionName	string	require	Name of the fracture region.
initialDt	real6	1e+9	Initial time-step value required by the solver to the event manager.
logLevel	integer	0	Log level
name	string	require	A name is required for any non-unique nodes
solidSolverName	string	require	Name of the solid mechanics solver in the rock matrix
targetRegions	string	require	Allowable regions that the solver may be applied to. Note that this does not indicate that the solver will be applied to these regions, only that allocation will occur such that the solver may be applied to these regions. The decision about what regions this solver will beapplied to rests in the EventManager.
useStaticCondensation	integer	0	Defines whether to use static condensation or not.
LinearSolverParameters	node	uniqu	*Element: LinearSolverParameters*
NonlinearSolverParameters	node	uniqu	*Element: NonlinearSolverParameters*

Element: SolidMechanicsLagrangianSSLE

Name	Type	Default	Description
cflFactor	real64	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
contactRelationName	string	NOCONTACT	Name of contact relation to enforce constraints on fracture boundary.
discretization	string	required	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
initialDt	real64	1e+99	Initial time-step value required by the solver to the event manager.
logLevel	integer	0	Log level
massDamping	real64	0	Value of mass based damping coefficient.
maxNumResolves	integer	10	Value to indicate how many resolves may be executed after some other event is executed. For example, if a SurfaceGenerator is specified, it will be executed after the mechanics solve. However if a new surface is generated, then the mechanics solve must be executed again due to the change in topology.
name	string	required	A name is required for any non-unique nodes
newmarkBeta	real64	0.25	Value of β in the Newmark Method for Implicit Dynamic time integration option. This should be pow(newmarkGamma+0.5,2.0)/4.0 unless you know what you are doing.
newmarkGamma	real64	0.5	Value of γ in the Newmark Method for Implicit Dynamic time integration option
stiffnessDamping	real64	0	Value of stiffness based damping coefficient.

Element: SolidMechanicsStateReset

Name	Type	Default	Description
disableInelasticity	integer	0	Flag to enable/disable inelastic behavior
logLevel	integer	0	Log level
name	string	required	A name is required for any non-unique nodes
resetDisplacements	integer	1	Flag to reset displacements (and velocities)
solidSolverName	string	required	Name of the solid mechanics solver

Element: SolidMechanicsStatistics

Name	Type	Default	Description
logLevel	integer	0	Log level
name	string	required	A name is required for any non-unique nodes
solidSolverName	string	required	Name of the solid solver

Element: SolidMechanics_LagrangianFEM

Name	Type	Default	Description
cflFactor	real64	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
contactRelationName	string	NOCONTACT	Name of contact relation to enforce constraints on fracture boundary.
discretization	string	required	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
initialDt	real64	1e+99	Initial time-step value required by the solver to the event manager.
logLevel	integer	0	Log level
massDamping	real64	0	Value of mass based damping coefficient.
maxNumResolves	integer	10	Value to indicate how many resolves may be executed after some other event is executed. For example, if a SurfaceGenerator is specified, it will be executed after the mechanics solve. However if a new surface is generated, then the mechanics solve must be executed again due to the change in topology.
name	string	required	A name is required for any non-unique nodes
newmarkBeta	real64	0.25	Value of β in the Newmark Method for Implicit Dynamic time integration option. This should be pow(newmarkGamma+0.5,2.0)/4.0 unless you know what you are doing.
newmarkGamma	real64	0.5	Value of γ in the Newmark Method for Implicit Dynamic time integration option
stiffnessDamping	real64	0	Value of stiffness based damping coefficient.

Element: SoloEvent

Name	Type	Default	Description
beginTime	real64	0	Start time of this event.
endTime	real64	1e+10	End time of this event.
finalDtStretch	real64	0.001	Allow the final dt request for this event to grow by this percentage to match the endTime exactly.
forceDt	real64	-1	While active, this event will request this timestep value (ignoring any children/targets requests).
logLevel	integer	0	Log level
maxEventDt	real64	-1	While active, this event will request a timestep <= this value (depending upon any child/target requests).
name	string	required	A name is required for any non-unique nodes
target	string		Name of the object to be executed when the event criteria are met.
targetCycle	integer	-1	Targeted cycle to execute the event.
targetExactStartStop	integer	1	If this option is set, the event will reduce its timestep requests to match any specified beginTime/endTimes exactly.
targetExactTimestep	integer	1	If this option is set, the event will reduce its timestep requests to match the specified execution time exactly: dt_request = min(dt_request, t_target - time)).
targetTime	real64	-1	Targeted time to execute the event.
HaltEvent	node		*Element: HaltEvent*
PeriodicEvent	node		*Element: PeriodicEvent*
SoloEvent	node		*Element: SoloEvent*

Element: Solvers

Name	Type	Default	Description
gravityVector	R1Tensor	{0,0,-9.81}	Gravity vector used in the physics solvers
AcousticFirstOrderSEM	node		*Element: AcousticFirstOrderSEM*
AcousticSEM	node		*Element: AcousticSEM*
AcousticVTISEM	node		*Element: AcousticVTISEM*
CompositionalMultiphaseFVM	node		*Element: CompositionalMultiphaseFVM*
CompositionalMultiphaseHybridFVM	node		*Element: CompositionalMultiphaseHybridFVM*
CompositionalMultiphaseReservoir	node		*Element: CompositionalMultiphaseReservoir*
CompositionalMultiphaseWell	node		*Element: CompositionalMultiphaseWell*
ElasticFirstOrderSEM	node		*Element: ElasticFirstOrderSEM*
ElasticSEM	node		*Element: ElasticSEM*
EmbeddedSurfaceGenerator	node		*Element: EmbeddedSurfaceGenerator*
FlowProppantTransport	node		*Element: FlowProppantTransport*
Hydrofracture	node		*Element: Hydrofracture*

continues on next pa

Table 1.4 – continued from previous page

Name	Type	Default	Description
LagrangianContact	node		*Element: LagrangianContact*
LaplaceFEM	node		*Element: LaplaceFEM*
MultiphasePoromechanics	node		*Element: MultiphasePoromechanics*
MultiphasePoromechanicsReservoir	node		*Element: MultiphasePoromechanicsReservoir*
PhaseFieldDamageFEM	node		*Element: PhaseFieldDamageFEM*
PhaseFieldFracture	node		*Element: PhaseFieldFracture*
ProppantTransport	node		*Element: ProppantTransport*
ReactiveCompositionalMultiphaseOBL	node		*Element: ReactiveCompositionalMultiphaseOBL*
SinglePhaseFVM	node		*Element: SinglePhaseFVM*
SinglePhaseHybridFVM	node		*Element: SinglePhaseHybridFVM*
SinglePhasePoromechanics	node		*Element: SinglePhasePoromechanics*
SinglePhasePoromechanicsConformingFractures	node		*Element: SinglePhasePoromechanicsConformingFractu*
SinglePhasePoromechanicsEmbeddedFractures	node		*Element: SinglePhasePoromechanicsEmbeddedFractur*
SinglePhasePoromechanicsReservoir	node		*Element: SinglePhasePoromechanicsReservoir*
SinglePhaseProppantFVM	node		*Element: SinglePhaseProppantFVM*
SinglePhaseReservoir	node		*Element: SinglePhaseReservoir*
SinglePhaseWell	node		*Element: SinglePhaseWell*
SolidMechanicsEmbeddedFractures	node		*Element: SolidMechanicsEmbeddedFractures*
SolidMechanicsLagrangianSSLE	node		*Element: SolidMechanicsLagrangianSSLE*
SolidMechanics_LagrangianFEM	node		*Element: SolidMechanics_LagrangianFEM*
SurfaceGenerator	node		*Element: SurfaceGenerator*

Element: SourceFlux

Name	Type	Default	Description
bcApplicationTableName	string		Name of table that specifies the on/off application of the boundary condition.
beginTime	real64	-1e+99	Time at which the boundary condition will start being applied.
component	integer	-1	Component of field (if tensor) to apply boundary condition to.
direction	R1Tensor	{0,0,0}	Direction to apply boundary condition to.
endTime	real64	1e+99	Time at which the boundary condition will stop being applied.
functionName	string		Name of function that specifies variation of the boundary condition.
initialCondition	integer	0	Boundary condition is applied as an initial condition.
logLevel	integer	0	Log level
name	string	required	A name is required for any non-unique nodes
objectPath	string		Path to the target field
scale	real64	0	Scale factor for value of the boundary condition.
setNames	string_array	required	Name of sets that boundary condition is applied to.

Element: SurfaceElementRegion

Name	Type	Default	Description
defaultAperture	real64	required	The default aperture of newly formed surface elements.
faceBlock	string	FractureSubRegion	The name of the face block in the mesh, or the embedded surface.
materialList	string_array	required	List of materials present in this region
meshBody	string		Mesh body that contains this region
name	string	required	A name is required for any non-unique nodes
subRegionType	geos_SurfaceElementRegion_Sur	faceElement	Type of surface element subregion. Valid options: {faceElement, embeddedElement}.

Element: SurfaceGenerator

Name	Type	Default	Description
cflFactor	real6	0.5	Factor to apply to the CFL condition when calculating the maximum allowable time step. Values should be in the interval (0,1]
fractureRegion	string	Fracture	(no description available)
initialDt	real6	1e+9	Initial time-step value required by the solver to the event manager.
logLevel	integer	0	Log level
mpiCommOrder	integer	0	Flag to enable MPI consistent communication ordering
name	string	require	A name is required for any non-unique nodes
nodeBasedSIF	integer	0	Flag for choosing between node or edge based criteria: 1 for node based criterion
rockToughness	real6	require	Rock toughness of the solid material
targetRegions	string	require	Allowable regions that the solver may be applied to. Note that this does not indicate that the solver will be applied to these regions, only that allocation will occur such that the solver may be applied to these regions. The decision about what regions this solver will beapplied to rests in the EventManager.
LinearSolverParameters	node	uniqu	*Element: LinearSolverParameters*
NonlinearSolverParameters	node	uniqu	*Element: NonlinearSolverParameters*

Element: SymbolicFunction

Name	Type	Default	Description
expression	string	required	Symbolic math expression
inputVarNames	string_array	{}	Name of fields are input to function.
name	string	required	A name is required for any non-unique nodes
variableNames	string_array	required	List of variables in expression. The order must match the evaluate argument

Element: TableCapillaryPressure

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes
nonWettingIntermediateCapPressureTableName	string		Capillary pressure table [Pa] for the pair (non-wetting phase, intermediate phase) Note that this input is only used for three-phase flow. If you want to do a two-phase simulation, please use instead wettingNonWettingCapPressureTableName to specify the table names
phaseNames	string_array	required	List of fluid phases
wettingIntermediateCapPressureTableName	string		Capillary pressure table [Pa] for the pair (wetting phase, intermediate phase) Note that this input is only used for three-phase flow. If you want to do a two-phase simulation, please use instead wettingNonWettingCapPressureTableName to specify the table names
wettingNonWettingCapPressureTableName	string		Capillary pressure table [Pa] for the pair (wetting phase, non-wetting phase) Note that this input is only used for two-phase flow. If you want to do a three-phase simulation, please use instead wettingIntermediateCapPressureTableName and nonWettingIntermediateCapPressureTableName to specify the table names

Element: TableFunction

Name	Type	Default	Description
coordinateFiles	path_array	{}	List of coordinate file names for ND Table
coordinates	real64_array	{0}	Coordinates inputs for 1D tables
inputVarNames	string_array	{}	Name of fields are input to function.
interpolation	geos_TableFunction_Interpo	linear	Interpolation method. Valid options: * linear * nearest * upper * lower
name	string	required	A name is required for any non-unique nodes
values	real64_array	{0}	Values for 1D tables
voxelFile	path		Voxel file name for ND Table

Element: TableRelativePermeability

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes
nonWettingIntermediateRelPermTableNames	string_array	{}	List of relative permeability tables for the pair (non-wetting phase, intermediate phase) The expected format is "{ nonWettingPhaseRelPermTableName, intermediatePhaseRelPermTableName }", in that order Note that this input is only used for three-phase flow. If you want to do a two-phase simulation, please use instead wettingNonWettingRelPermTableNames to specify the table names
phaseNames	string_array	required	List of fluid phases
wettingIntermediateRelPermTableNames	string_array	{}	List of relative permeability tables for the pair (wetting phase, intermediate phase) The expected format is "{ wettingPhaseRelPermTableName, intermediatePhaseRelPermTableName }", in that order Note that this input is only used for three-phase flow. If you want to do a two-phase simulation, please use instead wettingNonWettingRelPermTableNames to specify the table names
wettingNonWettingRelPermTableNames	string_array	{}	List of relative permeability tables for the pair (wetting phase, non-wetting phase) The expected format is "{

Element: TableRelativePermeabilityHysteresis

Name	Type	Default	Description
drainageNonWettingIntermediateRelPermTableNames	string_array	{}	List of drainage relative permeability tables for the pair (non-wetting phase, intermediate phase) The expected format is "{ nonWettingPhaseRelPermTableName, intermediatePhaseRelPermTableName }", in that order Note that this input is only used for three-phase flow. If you want to do a two-phase simulation, please use instead drainageWettingNonWettingRelPermTableNames to specify the table names
drainageWettingIntermediateRelPermTableNames	string_array	{}	List of drainage relative permeability tables for the pair (wetting phase, intermediate phase) The expected format is "{ wettingPhaseRelPermTableName, intermediatePhaseRelPermTableName }", in that order Note that this input is only used for three-phase flow. If you want to do a two-phase simulation, please use instead drainageWettingNonWettingRelPermTableNames to specify the table names
drainageWettingNonWettingRelPermTableNames	string_array	{}	List of drainage relative permeability tables for the pair (wetting phase, non-wetting phase) The expected format is "{ wettingPhaseRelPermTableName,

Element: Tasks

Name	Type	Default	Description
CompositionalMultiphaseStatistics	node		*Element: CompositionalMultiphaseStatistics*
MultiphasePoromechanicsInitialization	node		*Element: MultiphasePoromechanicsInitialization*
PVTDriver	node		*Element: PVTDriver*
PackCollection	node		*Element: PackCollection*
ReactiveFluidDriver	node		*Element: ReactiveFluidDriver*
RelpermDriver	node		*Element: RelpermDriver*
SinglePhasePoromechanicsInitialization	node		*Element: SinglePhasePoromechanicsInitialization*
SinglePhaseStatistics	node		*Element: SinglePhaseStatistics*
SolidMechanicsStateReset	node		*Element: SolidMechanicsStateReset*
SolidMechanicsStatistics	node		*Element: SolidMechanicsStatistics*
TriaxialDriver	node		*Element: TriaxialDriver*

Element: ThermalCompressibleSinglePhaseFluid

Name	Type	Default	Description
compressibility	real64	0	Fluid compressibility
defaultDensity	real64	required	Default value for density.
defaultViscosity	real64	required	Default value for viscosity.
densityModelType	geos_constitutive_Exponent	linear	Type of density model. Valid options: * exponential * linear * quadratic
internalEnergyModelType	geos_constitutive_Exponent	linear	Type of internal energy model. Valid options: * exponential * linear * quadratic
name	string	required	A name is required for any non-unique nodes
referenceDensity	real64	1000	Reference fluid density
referenceInternalEnergy	real64	0.001	Reference fluid internal energy
referencePressure	real64	0	Reference pressure
referenceTemperature	real64	0	Reference temperature
referenceViscosity	real64	0.001	Reference fluid viscosity
thermalExpansionCoeff	real64	0	Fluid thermal expansion coefficient. Unit: 1/K
viscosibility	real64	0	Fluid viscosity exponential coefficient
viscosityModelType	geos_constitutive_Exponent	linear	Type of viscosity model. Valid options: * exponential * linear * quadratic
volumetricHeatCapacity	real64	0	Fluid volumetric heat capacity. Unit: J/kg/K

Element: ThickPlane

Name	Type	Default	Description
name	string	required	A name is required for any non-unique nodes
normal	R1Tensor	required	Normal (n_x,n_y,n_z) to the plane (will be normalized automatically)
origin	R1Tensor	required	Origin point (x,y,z) of the plane (basically, any point on the plane)
thickness	real64	required	The total thickness of the plane (with half to each side)

Element: TimeHistory

Name	Type	Default	Description
childDirectory	string		Child directory path
filename	string	TimeHistory	The filename to which to write time history output.
format	string	hdf	The output file format for time history output.
name	string	required	A name is required for any non-unique nodes
parallelThreads	integer	1	Number of plot files.
sources	string_array	required	A list of collectors from which to collect and output time history information.

Element: Traction

Name	Type	Default	Description
bcApplicationTableName	string		Name of table that specifies the on/off application of the boundary condition.
beginTime	real64	-1e+99	Time at which the boundary condition will start being applied.
direction	R1Tensor	{0,0,0}	Direction to apply boundary condition to.
endTime	real64	1e+99	Time at which the boundary condition will stop being applied.
functionName	string		Name of function that specifies variation of the boundary condition.
initialCondition	integer	0	Boundary condition is applied as an initial condition.
inputStress	R2SymTensor	{0,0,0,0,0,0}	Input stress for tractionType = stress
logLevel	integer	0	Log level
name	string	required	A name is required for any non-unique nodes
objectPath	string		Path to the target field
scale	real64	0	Scale factor for value of the boundary condition.
setNames	string_array	required	Name of sets that boundary condition is applied to.
tractionType	geos_TractionBoundaryCon	vector	Type of traction boundary condition. Options are: vector - traction is applied to the faces as specified from the scale and direction, normal - traction is applied to the faces as a pressure specified from the product of scale and the outward face normal, stress - traction is applied to the faces as specified by the inner product of input stress and face normal.

Element: TriaxialDriver

Name	Type	Default	Description
axialControl	string	required	Function controlling axial stress or strain (depending on test mode)
baseline	path	none	Baseline file
initialStress	real64	required	Initial stress (scalar used to set an isotropic stress state)
logLevel	integer	0	Log level
material	string	required	Solid material to test
mode	string	required	Test mode [stressControl, strainControl, mixedControl]
name	string	required	A name is required for any non-unique nodes
output	string	none	Output file
radialControl	string	required	Function controlling radial stress or strain (depending on test mode)
steps	integer	required	Number of load steps to take

Element: TwoPointFluxApproximation

Name	Type	Default	Description
areaRelTol	real64	1e-08	Relative tolerance for area calculations.
meanPermCoefficient	real64	1	(no description available)
name	string	required	A name is required for any non-unique nodes
upwindingScheme	geos_UpwindingScheme	PPU	Type of upwinding scheme. Valid options: * PPU * C1PPU
usePEDFM	integer	0	(no description available)

Element: VTK

Name	Type	Default	Description
childDirectory	string		Child directory path
fieldNames	string_array	{}	Names of the fields to output. If this attribute is specified, GEOSX outputs all the fields specified by the user, regardless of their *plotLevel*
format	geos_vtk_VTI	binary	Output data format. Valid options: `binary`, `ascii`
name	string	require	A name is required for any non-unique nodes
onlyPlotSpecifiedFieldNames	integer	0	If this flag is equal to 1, then we only plot the fields listed in *fieldNames*. Otherwise, we plot all the fields with the required *plotLevel*, plus the fields listed in *fieldNames*
outputRegionType	geos_vtk_VTI	all	Output region types. Valid options: `cell`, `well`, `surface`, `all`
parallelThreads	integer	1	Number of plot files.
plotFileRoot	string	VTK	Name of the root file for this output.
plotLevel	integer	1	Level detail plot. Only fields with lower of equal plot level will be output.
writeFEMFaces	integer	0	(no description available)
writeGhostCells	integer	0	Should the vtk files contain the ghost cells or not.

Element: VTKMesh

Name	Type	Default	Description
faceBlocks	string_	{}	For multi-block files, names of the face mesh block.
fieldNames GEOS	string_	{}	Names of the volumic fields in GEOSX to import into
fieldsToImport	string_	{}	Volumic fields to be imported from the external mesh file
file	path	require	Path to the mesh file
logLev	integer	0	Log level
mainBlockName	string	main	For multi-block files, name of the 3d mesh block.
name	string	require	A name is required for any non-unique nodes
nodesetNames	string_	{}	Names of the VTK nodesets to import
partitionMetho	geos_v	parm	Method (library) used to partition the mesh
partitionRefinement	integer	1	Number of partitioning refinement iterations (defaults to 1, recommended value).A value of 0 disables graph partitioning and keeps simple kd-tree partitions (not recommended). Values higher than 1 may lead to slightly improved partitioning, but yield diminishing returns.
regionAttribute	string	attribut	Name of the VTK cell attribute to use as region marker
scale	R1Tens	{1,1,	Scale the coordinates of the vertices by given scale factors (after translation)
surfacicFieldsIn GEOS	string_	{}	Names of the surfacic fields in GEOSX to import into
surfacicFieldsToImport	string_	{}	Surfacic fields to be imported from the external mesh file
translate	R1Tens	{0,0,	Translate the coordinates of the vertices by a given vector (prior to scaling)
useGlobalIds	integer	0	Controls the use of global IDs in the input file for cells and points. If set to 0 (default value), the GlobalId arrays in the input mesh are used if available, and generated otherwise. If set to a negative value, the GlobalId arrays in the input mesh are not used, and generated global Ids are automatically generated. If set to a positive value, the GlobalId arrays in the input mesh are used and required, and the simulation aborts if they are not available
In-	node	*Element: InternalWell*

Element: VanGenuchtenBakerRelativePermeability

Name	Type	Default	Description
gasOilRelPermExponentInv	real64_array	{0.5}	Rel perm power law exponent inverse for the pair (gas phase, oil phase) at residual water saturation The expected format is "{ gasExp, oilExp }", in that order
gasOilRelPermMaxValue	real64_array	{0}	Maximum rel perm value for the pair (gas phase, oil phase) at residual water saturation The expected format is "{ gasMax, oilMax }", in that order
name	string	required	A name is required for any non-unique nodes
phaseMinVolumeFraction	real64_array	{0}	Minimum volume fraction value for each phase
phaseNames	string_array	required	List of fluid phases
waterOilRelPermExponentInv	real64_array	{0.5}	Rel perm power law exponent inverse for the pair (water phase, oil phase) at residual gas saturation The expected format is "{ waterExp, oilExp }", in that order
waterOilRelPermMaxValue	real64_array	{0}	Maximum rel perm value for the pair (water phase, oil phase) at residual gas saturation The expected format is "{ waterMax, oilMax }", in that order

Element: VanGenuchtenCapillaryPressure

Name	Type	Default	Description
capPressureEpsilon	real64	1e-06	Saturation at which the extremum capillary pressure is attained; used to avoid infinite capillary pressure values for saturations close to 0 and 1
name	string	required	A name is required for any non-unique nodes
phaseCapPressureExponentInv	real64_a	{0.5}	Inverse of capillary power law exponent for each phase
phaseCapPressureMultiplier	real64_a	{1}	Entry pressure value for each phase
phaseMinVolumeFraction	real64_a	{0}	Minimum volume fraction value for each phase
phaseNames	string_a	required	List of fluid phases

Element: ViscoDruckerPrager

Name	Type	Default	Description
defaultBulkModulus	real64	-1	Default Bulk Modulus Parameter
defaultCohesion	real64	0	Initial cohesion
defaultDensity	real64	required	Default Material Density
defaultDilationAngle	real64	30	Dilation angle (degrees)
defaultFrictionAngle	real64	30	Friction angle (degrees)
defaultHardeningRate	real64	0	Cohesion hardening/softening rate
defaultPoissonRatio	real64	-1	Default Poisson’s Ratio
defaultShearModulus	real64	-1	Default Shear Modulus Parameter
defaultThermalExpansionCoefficient	real64	0	Default Linear Thermal Expansion Coefficient of the Solid Rock Frame
defaultYoungModulus	real64	-1	Default Young’s Modulus
name	string	required	A name is required for any non-unique nodes
relaxationTime	real64	required	Relaxation time

Element: ViscoExtendedDruckerPrager

Name	Type	Default	Description
defaultBulkModulus	real64	-1	Default Bulk Modulus Parameter
defaultCohesion	real64	0	Initial cohesion
defaultDensity	real64	required	Default Material Density
defaultDilationRatio	real64	1	Dilation ratio [0,1] (ratio = tan dilationAngle / tan frictionAngle)
defaultHardening	real64	0	Hardening parameter (hardening rate is faster for smaller values)
defaultInitialFrictionAngle	real64	30	Initial friction angle (degrees)
defaultPoissonRatio	real64	-1	Default Poisson's Ratio
defaultResidualFrictionAngle	real64	30	Residual friction angle (degrees)
defaultShearModulus	real64	-1	Default Shear Modulus Parameter
defaultThermalExpansionCoefficient	real64	0	Default Linear Thermal Expansion Coefficient of the Solid Rock Frame
defaultYoungModulus	real64	-1	Default Young's Modulus
name	string	required	A name is required for any non-unique nodes
relaxationTime	real64	required	Relaxation time

Element: ViscoModifiedCamClay

Name	Type	Default	Description
defaultCslSlope	real64	1	Slope of the critical state line
defaultDensity	real64	required	Default Material Density
defaultPreConsolidationPressure	real64	-1.5	Initial preconsolidation pressure
defaultRecompressionIndex	real64	0.002	Recompresion Index
defaultRefPressure	real64	-1	Reference Pressure
defaultRefStrainVol	real64	0	Reference Volumetric Strain
defaultShearModulus	real64	-1	Elastic Shear Modulus Parameter
defaultThermalExpansionCoefficient	real64	0	Default Linear Thermal Expansion Coefficient of the Solid Rock Frame
defaultVirginCompressionIndex	real64	0.005	Virgin compression index
name	string	required	A name is required for any non-unique nodes
relaxationTime	real64	required	Relaxation time

Element: WellControls

Name	Type	Default	Description
control	geos_WellControls_Control	required	Well control. Valid options: * BHP * phaseVolRate * totalVolRate * uninitialized
enableCrossflow	integer	1	Flag to enable crossflow. Currently only supported for injectors: - If the flag is set to 1, both reservoir-to-well flow and well-to-reservoir flow are allowed at the perforations. - If the flag is set to 0, we only allow well-to-reservoir flow at the perforations.
initialPressureCoefficient	real64	0.1	Tuning coefficient for the initial well pressure of rate-controlled wells: - Injector pressure at reference depth initialized as: (1+initialPressureCoefficient)*reservoirPressureAtClosestPerfor + density*g*(zRef - zPerf) - Producer pressure at reference depth initialized as: (1-initialPressureCoefficient)*reservoirPres + density*g*(zRef - zPerf)
injectionStream	real64_array	{-1}	Global component densities of the injection stream [moles/m^3 or kg/m^3]
injectionTemperature	real64	-1	Temperature of the injection stream [K]
logLevel	integer	0	Log level
name	string	required	A name is required for any non-unique nodes
referenceElevation	real64	required	Reference elevation where

Element: WellElementRegion

Name	Type	Default	Description
materialList	string_array	required	List of materials present in this region
meshBody	string		Mesh body that contains this region
name	string	required	A name is required for any non-unique nodes

Element: WillisRichardsPermeability

Name	Type	Default	Description
dilationCoefficient	real64	required	Dilation coefficient (tan of dilation angle).
maxFracAperture	real64	required	Maximum fracture aperture at zero contact stress.
name	string	required	A name is required for any non-unique nodes
refClosureStress	real64	required	Effective normal stress causes 90% reduction in aperture.

Element: crusher

Name	Type	Default	Description
Run	node	unique	*Element: Run*

Element: lassen

Name	Type	Default	Description
Run	node	unique	*Element: Run*

Element: quartz

Name	Type	Default	Description
Run	node	unique	*Element: Run*

Datastructure Definitions

Datastructure: AcousticFirstOrderSEM

Name	Type	Description
indexSeismoTrace	integer	Count for output pressure at receivers
maxStableDt	real64	Value of the Maximum Stable Timestep for this solver.
meshTargets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >	MeshBody/Region combinations that the solver will be applied to.
pressureNp1AtR	real32_array2d	Pressure value at each receiver for each timestep
rcvElem	integer_array	Element containing the receivers
receiverIsLocal	integer_array	Flag that indicates whether the receiver is local to this MPI rank
receiverNodeIds	integer_array2d	Indices of the nodes (in the right order) for each receiver point
receiverRegion	integer_array	Region containing the receivers
sourceConstants	real64_array2d	Constant part of the receiver for the nodes listed in m_receiverNodeIds
sourceElem	integer_array	Element containing the sources
sourceIsAccessible	integer_array	Flag that indicates whether the source is local to this MPI rank
sourceNodeI	integer_array2d	Indices of the nodes (in the right order) for each source point
sourceRegion	integer_array	Region containing the sources
sourceValue	real32_array2d	Source Value of the sources
useDAS	integer	Flag to indicate if DAS type of data will be modeled
usePML	integer	Flag to apply PML
uxNp1AtRec	real32_array2d	Ux value at each receiver for each timestep
uyNp1AtRec	real32_array2d	Uy value at each receiver for each timestep
uzNp1AtRec	real32_array2d	Uz value at each receiver for each timestep
LinearSolverParameters	node	*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node	*Datastructure: NonlinearSolverParameters*
SolverStatistics	node	*Datastructure: SolverStatistics*

Datastructure: AcousticSEM

Name	Type	Description
indexSeismoTrace	integer	Count for output pressure at receivers
maxStableDt	real64	Value of the Maximum Stable Timestep for this solver.
meshTargets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >	MeshBody/Region combinations that the solver will be applied to.
pressureNp1AtR	real32_array2d	Pressure value at each receiver for each timestep
receiverIsLocal	integer_array	Flag that indicates whether the receiver is local to this MPI rank
receiverNodeIds	integer_array2d	Indices of the nodes (in the right order) for each receiver point
sourceConstants	real64_array2d	Constant part of the receiver for the nodes listed in m_receiverNodeIds
sourceIsAccessible	integer_array	Flag that indicates whether the source is local to this MPI rank
sourceNodeI	integer_array2d	Indices of the nodes (in the right order) for each source point
sourceValue	real32_array2d	Source Value of the sources
useDAS	integer	Flag to indicate if DAS type of data will be modeled
usePML	integer	Flag to apply PML
LinearSolverParameters	node	*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node	*Datastructure: NonlinearSolverParameters*
SolverStatistics	node	*Datastructure: SolverStatistics*

Datastructure: AcousticVTISEM

Name	Type	Description
indexSeismoTrace	integer	Count for output pressure at receivers
maxStableDt	real64	Value of the Maximum Stable Timestep for this solver.
meshTargets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >	MeshBody/Region combinations that the solver will be applied to.
pressureNp1AtR	real32_array2d	Pressure value at each receiver for each timestep
receiverIsLocal	integer_array	Flag that indicates whether the receiver is local to this MPI rank
receiverNodeIds	integer_array2d	Indices of the nodes (in the right order) for each receiver point
sourceConstants	real64_array2d	Constant part of the receiver for the nodes listed in m_receiverNodeIds
sourceIsAccessible	integer_array	Flag that indicates whether the source is local to this MPI rank
sourceNodeI	integer_array2d	Indices of the nodes (in the right order) for each source point
sourceValue	real32_array2d	Source Value of the sources
useDAS	integer	Flag to indicate if DAS type of data will be modeled
usePML	integer	Flag to apply PML
LinearSolverParameters	node	*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node	*Datastructure: NonlinearSolverParameters*
SolverStatistics	node	*Datastructure: SolverStatistics*

Datastructure: Aquifer

Name	Type	Description
component	integer	Component of field (if tensor) to apply boundary condition to.
cumulativeFlux	real64	(no description available)
fieldName	string	Name of field that boundary condition is applied to.
objectPath	string	Path to the target field

Datastructure: Benchmarks

Name	Type	Description
crusher	node	*Datastructure: crusher*
lassen	node	*Datastructure: lassen*
quartz	node	*Datastructure: quartz*

Datastructure: BiotPorosity

Name	Type	Description
averageMeanEffectiveStressIncrement_k	real64_array	Mean effective stress increment averaged over quadrature points at the previous sequential iteration
biotCoefficient	real64_array	Biot coefficient
dPorosity_dPressure	real64_array	Derivative of rock porosity with respect to pressure
dPorosity_dTemperature	real64_array	Derivative of rock porosity with respect to temperature
initialPorosity	real64_array	Initial porosity
meanEffectiveStressIncrement_k	real64_array	Mean effective stress increment at quadrature points at the previous sequential iteration
porosity	real64_array	Rock porosity
porosity_n	real64_array	Rock porosity at the previous converged time step
referencePorosity	real64_array	Reference porosity
solidBulkModulus	real64_array	Solid bulk modulus
thermalExpansionCoefficient	real64_array	Thermal expansion coefficient

Datastructure: BlackOilFluid

Name	Type	Description
PVTO	geos_constitutive_PVTOData	(no description available)
dPhaseCompFraction	LvArray_Array< double, 5, camp_int_seq< long, 0l, 1l, 2l, 3l, 4l >, int, LvArray_ChaiBuffer >	Derivative of phase component fraction with respect to pressure, temperature, and global component fractions
dPhaseDensity	real64_array4d	Derivative of phase density with respect to pressure, temperature, and global component fractions
dPhaseEnthalpy	real64_array4d	Derivative of phase enthalpy with respect to pressure, temperature, and global component fractions
dPhaseFraction	real64_array4d	Derivative of phase fraction with respect to pressure, temperature, and global component fractions
dPhaseInternalEnergy	real64_array4d	Derivative of phase internal energy with respect to pressure, temperature, and global component fractions
dPhaseMassDensity	real64_array4d	Derivative of phase mass density with respect to pressure, temperature, and global component fractions
dPhaseViscosity	real64_array4d	Derivative of phase viscosity with respect to pressure, temperature, and global component fractions
dTotalDensity	real64_array3d	Derivative of total density with respect to pressure, temperature, and global component fractions
formationVolFactorTableWrappers	LvArray_Array< geos_TableFunction_KernelWrapper, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >	(no description available)
hydrocarbonPhaseOrder	integer_array	(no description available)
phaseCompFraction	real64_array4d	Phase component fraction
phaseCompFraction_n	real64_array4d	Phase component fraction at the previous converged time step
phaseDensity	real64_array3d	Phase density
phaseDensity_n	real64_array3d	Phase density at the previous converged time step
phaseEnthalpy	real64_array3d	Phase enthalpy
phaseEnthalpy_n	real64_array3d	Phase enthalpy at the previous converged time step
phaseFraction	real64_array3d	Phase fraction
phaseInternalEnergy	real64_array3d	Phase internal energy
phaseInternalEnergy_n	real64_array3d	Phase internal energy at the previous converged time step
phaseMassDensity	real64_array3d	Phase mass density
phaseOrder	integer_array	(no description available)
phaseTypes	integer_array	(no description available)
phaseViscosity	real64_array3d	Phase viscosity
totalDensity	real64_array2d	Total density
totalDensity_n	real64_array2d	Total density at the previous converged time step

Datastructure: Blueprint

Name	Type	Description

Datastructure: Box

Name	Type	Description
center	R1Tensor	(no description available)
cosStrike	real64	(no description available)
sinStrike	real64	(no description available)

Datastructure: BrooksCoreyBakerRelativePermeability

Name	Type	Description
dPhaseRelPerm_dPhase	real64_arra	Derivative of phase relative permeability with respect to phase volume fraction
phaseOrder	integer_array	(no description available)
phaseRelPerm	real64_arra	Phase relative permeability
phaseRelPerm_n	real64_arra	Phase relative permeability at previous time
phaseTrappedVolFraction	real64_arra	Phase trapped volume fraction
phaseTypes	integer_array	(no description available)
volFracScale	real64	Factor used to scale the phase capillary pressure, defined as: one minus the sum of the phase minimum volume fractions.

Datastructure: BrooksCoreyCapillaryPressure

Name	Type	Description
dPhaseCapPressure_dPhaseVolFraction	real64_arra	Derivative of phase capillary pressure with respect to phase volume fraction
phaseCapPressure	real64_arra	Phase capillary pressure
phaseOrder	integer_array	(no description available)
phaseTypes	integer_array	(no description available)
volFracScale	real64	Factor used to scale the phase capillary pressure, defined as: one minus the sum of the phase minimum volume fractions.

Datastructure: BrooksCoreyRelativePermeability

Name	Type	Description
dPhaseRelPerm_dPhase	real64_arra	Derivative of phase relative permeability with respect to phase volume fraction
phaseOrder	integer_array	(no description available)
phaseRelPerm	real64_arra	Phase relative permeability
phaseRelPerm_n	real64_arra	Phase relative permeability at previous time
phaseTrappedVolFraction	real64_arra	Phase trapped volume fraction
phaseTypes	integer_array	(no description available)
volFracScale	real64	Factor used to scale the phase relative permeability, defined as: one minus the sum of the phase minimum volume fractions.

Datastructure: CO2BrineEzrokhiFluid

Name	Type	Description
dPhaseCompFraction	LvArray_Array< double, 5, camp_int_seq< long, 0l, 1l, 2l, 3l, 4l >, int, LvArray_ChaiBuffer >	Derivative of phase component fraction with respect to pressure, temperature, and global component fractions
dPhaseDensity	real64_array4d	Derivative of phase density with respect to pressure, temperature, and global component fractions
dPhaseEnthalpy	real64_array4d	Derivative of phase enthalpy with respect to pressure, temperature, and global component fractions
dPhaseFraction	real64_array4d	Derivative of phase fraction with respect to pressure, temperature, and global component fractions
dPhaseInternalEnergy	real64_array4d	Derivative of phase internal energy with respect to pressure, temperature, and global component fractions
dPhaseMassDensity	real64_array4d	Derivative of phase mass density with respect to pressure, temperature, and global component fractions
dPhaseViscosity	real64_array4d	Derivative of phase viscosity with respect to pressure, temperature, and global component fractions
dTotalDensity	real64_array3d	Derivative of total density with respect to pressure, temperature, and global component fractions
phaseCompFraction	real64_array4d	Phase component fraction
phaseCompFraction_n	real64_array4d	Phase component fraction at the previous converged time step
phaseDensity	real64_array3d	Phase density
phaseDensity_n	real64_array3d	Phase density at the previous converged time step
phaseEnthalpy	real64_array3d	Phase enthalpy
phaseEnthalpy_n	real64_array3d	Phase enthalpy at the previous converged time step
phaseFraction	real64_array3d	Phase fraction
phaseInternalEnergy	real64_array3d	Phase internal energy
phaseInternalEnergy_n	real64_array3d	Phase internal energy at the previous converged time step
phaseMassDensity	real64_array3d	Phase mass density
phaseViscosity	real64_array3d	Phase viscosity
totalDensity	real64_array2d	Total density
totalDensity_n	real64_array2d	Total density at the previous converged time step
useMass	integer	(no description available)

Datastructure: CO2BrineEzrokhiThermalFluid

Name	Type	Description
dPhaseCompFraction	LvArray_Array< double, 5, camp_int_seq< long, 0l, 1l, 2l, 3l, 4l >, int, LvArray_ChaiBuffer >	Derivative of phase component fraction with respect to pressure, temperature, and global component fractions
dPhaseDensity	real64_array4d	Derivative of phase density with respect to pressure, temperature, and global component fractions
dPhaseEnthalpy	real64_array4d	Derivative of phase enthalpy with respect to pressure, temperature, and global component fractions
dPhaseFraction	real64_array4d	Derivative of phase fraction with respect to pressure, temperature, and global component fractions
dPhaseInternalEnergy	real64_array4d	Derivative of phase internal energy with respect to pressure, temperature, and global component fractions
dPhaseMassDensity	real64_array4d	Derivative of phase mass density with respect to pressure, temperature, and global component fractions
dPhaseViscosity	real64_array4d	Derivative of phase viscosity with respect to pressure, temperature, and global component fractions
dTotalDensity	real64_array3d	Derivative of total density with respect to pressure, temperature, and global component fractions
phaseCompFraction	real64_array4d	Phase component fraction
phaseCompFraction_n	real64_array4d	Phase component fraction at the previous converged time step
phaseDensity	real64_array3d	Phase density
phaseDensity_n	real64_array3d	Phase density at the previous converged time step
phaseEnthalpy	real64_array3d	Phase enthalpy
phaseEnthalpy_n	real64_array3d	Phase enthalpy at the previous converged time step
phaseFraction	real64_array3d	Phase fraction
phaseInternalEnergy	real64_array3d	Phase internal energy
phaseInternalEnergy_n	real64_array3d	Phase internal energy at the previous converged time step
phaseMassDensity	real64_array3d	Phase mass density
phaseViscosity	real64_array3d	Phase viscosity
totalDensity	real64_array2d	Total density
totalDensity_n	real64_array2d	Total density at the previous converged time step
useMass	integer	(no description available)

Datastructure: CO2BrinePhillipsFluid

Name	Type	Description
dPhaseCompFraction	LvArray_Array< double, 5, camp_int_seq< long, 0l, 1l, 2l, 3l, 4l >, int, LvArray_ChaiBuffer >	Derivative of phase component fraction with respect to pressure, temperature, and global component fractions
dPhaseDensity	real64_array4d	Derivative of phase density with respect to pressure, temperature, and global component fractions
dPhaseEnthalpy	real64_array4d	Derivative of phase enthalpy with respect to pressure, temperature, and global component fractions
dPhaseFraction	real64_array4d	Derivative of phase fraction with respect to pressure, temperature, and global component fractions
dPhaseInternalEnergy	real64_array4d	Derivative of phase internal energy with respect to pressure, temperature, and global component fractions
dPhaseMassDensity	real64_array4d	Derivative of phase mass density with respect to pressure, temperature, and global component fractions
dPhaseViscosity	real64_array4d	Derivative of phase viscosity with respect to pressure, temperature, and global component fractions
dTotalDensity	real64_array3d	Derivative of total density with respect to pressure, temperature, and global component fractions
phaseCompFraction	real64_array4d	Phase component fraction
phaseCompFraction_n	real64_array4d	Phase component fraction at the previous converged time step
phaseDensity	real64_array3d	Phase density
phaseDensity_n	real64_array3d	Phase density at the previous converged time step
phaseEnthalpy	real64_array3d	Phase enthalpy
phaseEnthalpy_n	real64_array3d	Phase enthalpy at the previous converged time step
phaseFraction	real64_array3d	Phase fraction
phaseInternalEnergy	real64_array3d	Phase internal energy
phaseInternalEnergy_n	real64_array3d	Phase internal energy at the previous converged time step
phaseMassDensity	real64_array3d	Phase mass density
phaseViscosity	real64_array3d	Phase viscosity
totalDensity	real64_array2d	Total density
totalDensity_n	real64_array2d	Total density at the previous converged time step
useMass	integer	(no description available)

Datastructure: CO2BrinePhillipsThermalFluid

Name	Type	Description
dPhaseCompFraction	LvArray_Array< double, 5, camp_int_seq< long, 0l, 1l, 2l, 3l, 4l >, int, LvArray_ChaiBuffer >	Derivative of phase component fraction with respect to pressure, temperature, and global component fractions
dPhaseDensity	real64_array4d	Derivative of phase density with respect to pressure, temperature, and global component fractions
dPhaseEnthalpy	real64_array4d	Derivative of phase enthalpy with respect to pressure, temperature, and global component fractions
dPhaseFraction	real64_array4d	Derivative of phase fraction with respect to pressure, temperature, and global component fractions
dPhaseInternalEnergy	real64_array4d	Derivative of phase internal energy with respect to pressure, temperature, and global component fractions
dPhaseMassDensity	real64_array4d	Derivative of phase mass density with respect to pressure, temperature, and global component fractions
dPhaseViscosity	real64_array4d	Derivative of phase viscosity with respect to pressure, temperature, and global component fractions
dTotalDensity	real64_array3d	Derivative of total density with respect to pressure, temperature, and global component fractions
phaseCompFraction	real64_array4d	Phase component fraction
phaseCompFraction_n	real64_array4d	Phase component fraction at the previous converged time step
phaseDensity	real64_array3d	Phase density
phaseDensity_n	real64_array3d	Phase density at the previous converged time step
phaseEnthalpy	real64_array3d	Phase enthalpy
phaseEnthalpy_n	real64_array3d	Phase enthalpy at the previous converged time step
phaseFraction	real64_array3d	Phase fraction
phaseInternalEnergy	real64_array3d	Phase internal energy
phaseInternalEnergy_n	real64_array3d	Phase internal energy at the previous converged time step
phaseMassDensity	real64_array3d	Phase mass density
phaseViscosity	real64_array3d	Phase viscosity
totalDensity	real64_array2d	Total density
totalDensity_n	real64_array2d	Total density at the previous converged time step
useMass	integer	(no description available)

Datastructure: CarmanKozenyPermeability

Name	Type	Description
dPerm_dPorosity	real64_array3d	(no description available)
dPerm_dPressure	real64_array3d	Derivative of rock permeability with respect to pressure
permeability	real64_array3d	Rock permeability

Datastructure: CellElementRegion

Name	Type	Description
domainBoundaryIndicator	integer_array	(no description available)
ghostRank	integer_array	(no description available)
globalToLocalMap	geos_mapBase< long long, int, std_integral_constant< bool, false > >	(no description available)
isExternal	integer_array	(no description available)
localToGlobalMap	globalIndex_array	Array that contains a map from localIndex to globalIndex.
elementSubRegions	node	*Datastructure: elementSubRegions*
neighborData	node	*Datastructure: neighborData*
sets	node	*Datastructure: sets*

Datastructure: ChomboIO

Name	Type	Description

Datastructure: CompositeFunction

Name	Type	Description

Datastructure: CompositionalMultiphaseFVM

Name	Type	Description
maxStableDt	real64	Value of the Maximum Stable Timestep for this solver.
meshTargets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >	MeshBody/Region combinations that the solver will be applied to.
LinearSolverParameters	node	*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node	*Datastructure: NonlinearSolverParameters*
SolverStatistics	node	*Datastructure: SolverStatistics*

Datastructure: CompositionalMultiphaseFluid

Name	Type	Description
dPhaseCompFraction	LvArray_Array< double, 5, camp_int_seq< long, 0l, 1l, 2l, 3l, 4l >, int, LvArray_ChaiBuffer >	Derivative of phase component fraction with respect to pressure, temperature, and global component fractions
dPhaseDensity	real64_array4d	Derivative of phase density with respect to pressure, temperature, and global component fractions
dPhaseEnthalpy	real64_array4d	Derivative of phase enthalpy with respect to pressure, temperature, and global component fractions
dPhaseFraction	real64_array4d	Derivative of phase fraction with respect to pressure, temperature, and global component fractions
dPhaseInternalEnergy	real64_array4d	Derivative of phase internal energy with respect to pressure, temperature, and global component fractions
dPhaseMassDensity	real64_array4d	Derivative of phase mass density with respect to pressure, temperature, and global component fractions
dPhaseViscosity	real64_array4d	Derivative of phase viscosity with respect to pressure, temperature, and global component fractions
dTotalDensity	real64_array3d	Derivative of total density with respect to pressure, temperature, and global component fractions
phaseCompFraction	real64_array4d	Phase component fraction
phaseCompFraction_n	real64_array4d	Phase component fraction at the previous converged time step
phaseDensity	real64_array3d	Phase density
phaseDensity_n	real64_array3d	Phase density at the previous converged time step
phaseEnthalpy	real64_array3d	Phase enthalpy
phaseEnthalpy_n	real64_array3d	Phase enthalpy at the previous converged time step
phaseFraction	real64_array3d	Phase fraction
phaseInternalEnergy	real64_array3d	Phase internal energy
phaseInternalEnergy_n	real64_array3d	Phase internal energy at the previous converged time step
phaseMassDensity	real64_array3d	Phase mass density
phaseViscosity	real64_array3d	Phase viscosity
totalDensity	real64_array2d	Total density
totalDensity_n	real64_array2d	Total density at the previous converged time step
useMass	integer	(no description available)

Datastructure: CompositionalMultiphaseHybridFVM

Name	Type	Registered On	Description
maxStableD	real64		Value of the Maximum Stable Timestep for this solver.
meshTargets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >		MeshBody/Region combinations that the solver will be applied to.
facePressure_n	real64_array	*Datastructure: faceManager*	Face pressure at the previous converged time step
mimGravityCoefficient	real64_array	*Datastructure: faceManager*	Mimetic gravity coefficient
LinearSolverParameters	node		*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node		*Datastructure: NonlinearSolverParameters*
SolverStatistics	node		*Datastructure: SolverStatistics*

Datastructure: CompositionalMultiphaseReservoir

Name	Type	Description
discretization	string	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
maxSta	real64	Value of the Maximum Stable Timestep for this solver.
meshTa gets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >	MeshBody/Region combinations that the solver will be applied to.
LinearSolverParameters	node	*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node	*Datastructure: NonlinearSolverParameters*
SolverStatistics	node	*Datastructure: SolverStatistics*

Datastructure: CompositionalMultiphaseStatistics

Name	Type	Description

Datastructure: CompositionalMultiphaseWell

Name	Type	Description
discretization	string	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
maxSta	real64	Value of the Maximum Stable Timestep for this solver.
meshTargets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >	MeshBody/Region combinations that the solver will be applied to.
LinearSolverParameters	node	*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node	*Datastructure: NonlinearSolverParameters*
SolverStatistics	node	*Datastructure: SolverStatistics*
WellControls	node	*Datastructure: WellControls*

Datastructure: CompressibleSinglePhaseFluid

Name	Type	Description
dDensity_dPressure	real64_array2d	Derivative of density with respect to pressure
dDensity_dTemperature	real64_array2d	Derivative of density with respect to temperature
dEnthalpy_dPressure	real64_array2d	Derivative of enthalpy with respect to pressure
dEnthalpy_dTemperature	real64_array2d	Derivative of enthalpy with respect to temperature
dInternalEnergy_dPressure	real64_array2d	Derivative of internal energy with respect to pressure
dInternalEnergy_dTemperature	real64_array2d	Derivative of internal energy with respect to temperature
dViscosity_dPressure	real64_array2d	Derivative of viscosity with respect to pressure
dViscosity_dTemperature	real64_array2d	Derivative of viscosity with respect to temperature
density	real64_array2d	Density
density_n	real64_array2d	Density at the previous converged time step
enthalpy	real64_array2d	Enthalpy
internalEnergy	real64_array2d	Internal energy
internalEnergy_n	real64_array2d	Fluid internal energy at the previous converged step
viscosity	real64_array2d	Viscosity

Datastructure: CompressibleSolidCarmanKozenyPermeability

Name	Type	Description

Datastructure: CompressibleSolidConstantPermeability

Name	Type	Description

Datastructure: CompressibleSolidExponentialDecayPermeability

Name	Type	Description

Datastructure: CompressibleSolidParallelPlatesPermeability

Name	Type	Description

Datastructure: CompressibleSolidSlipDependentPermeability

Name	Type	Description

Datastructure: CompressibleSolidWillisRichardsPermeability

Name	Type	Description

Datastructure: ConstantPermeability

Name	Type	Description
dPerm_dPressure	real64_array3d	Derivative of rock permeability with respect to pressure
permeability	real64_array3d	Rock permeability

Datastructure: Constitutive

Name	Type	Description
BiotPorosity	node	*Datastructure: BiotPorosity*
BlackOilFluid	node	*Datastructure: BlackOilFluid*
BrooksCoreyBakerRelativePermeability	node	*Datastructure: BrooksCoreyBakerRelativePermeability*
BrooksCoreyCapillaryPressure	node	*Datastructure: BrooksCoreyCapillaryPressure*
BrooksCoreyRelativePermeability	node	*Datastructure: BrooksCoreyRelativePermeability*
CO2BrineEzrokhiFluid	node	*Datastructure: CO2BrineEzrokhiFluid*
CO2BrineEzrokhiThermalFluid	node	*Datastructure: CO2BrineEzrokhiThermalFluid*
CO2BrinePhillipsFluid	node	*Datastructure: CO2BrinePhillipsFluid*
CO2BrinePhillipsThermalFluid	node	*Datastructure: CO2BrinePhillipsThermalFluid*
CarmanKozenyPermeability	node	*Datastructure: CarmanKozenyPermeability*
CompositionalMultiphaseFluid	node	*Datastructure: CompositionalMultiphaseFluid*
CompressibleSinglePhaseFluid	node	*Datastructure: CompressibleSinglePhaseFluid*
CompressibleSolidCarmanKozenyPermeability	node	*Datastructure: CompressibleSolidCarmanKozenyPermeability*
CompressibleSolidConstantPermeability	node	*Datastructure: CompressibleSolidConstantPermeability*
CompressibleSolidExponentialDecayPermeability	node	*Datastructure: CompressibleSolidExponentialDecayPermeability*
CompressibleSolidParallelPlatesPermeability	node	*Datastructure: CompressibleSolidParallelPlatesPermeability*
CompressibleSolidSlipDependentPermeability	node	*Datastructure: CompressibleSolidSlipDependentPermeability*
CompressibleSolidWillisRichardsPermeability	node	*Datastructure: CompressibleSolidWillisRichardsPermeability*
ConstantPermeability	node	*Datastructure: ConstantPermeability*
Coulomb	node	*Datastructure: Coulomb*
DamageElasticIsotropic	node	*Datastructure: DamageElasticIsotropic*
DamageSpectralElasticIsotropic	node	*Datastructure: DamageSpectralElasticIsotropic*
DamageVolDevElasticIsotropic	node	*Datastructure: DamageVolDevElasticIsotropic*
DeadOilFluid	node	*Datastructure: DeadOilFluid*
DelftEgg	node	*Datastructure: DelftEgg*

continues on next page

Table 1.5 – continued from previous page

Name	Type	Description
DruckerPrager	node	*Datastructure: DruckerPrager*
ElasticIsotropic	node	*Datastructure: ElasticIsotropic*
ElasticIsotropicPressureDependent	node	*Datastructure: ElasticIsotropicPressureDependent*
ElasticOrthotropic	node	*Datastructure: ElasticOrthotropic*
ElasticTransverseIsotropic	node	*Datastructure: ElasticTransverseIsotropic*
ExponentialDecayPermeability	node	*Datastructure: ExponentialDecayPermeability*
ExtendedDruckerPrager	node	*Datastructure: ExtendedDruckerPrager*
FrictionlessContact	node	*Datastructure: FrictionlessContact*
JFunctionCapillaryPressure	node	*Datastructure: JFunctionCapillaryPressure*
ModifiedCamClay	node	*Datastructure: ModifiedCamClay*
MultiPhaseConstantThermalConductivity	node	*Datastructure: MultiPhaseConstantThermalConductivity*
MultiPhaseVolumeWeightedThermalConductivity	node	*Datastructure: MultiPhaseVolumeWeightedThermalConductivity*
NullModel	node	*Datastructure: NullModel*
ParallelPlatesPermeability	node	*Datastructure: ParallelPlatesPermeability*
ParticleFluid	node	*Datastructure: ParticleFluid*
PermeabilityBase	node	*Datastructure: PermeabilityBase*
PorousDelftEgg	node	*Datastructure: PorousDelftEgg*
PorousDruckerPrager	node	*Datastructure: PorousDruckerPrager*
PorousElasticIsotropic	node	*Datastructure: PorousElasticIsotropic*
PorousElasticOrthotropic	node	*Datastructure: PorousElasticOrthotropic*
PorousElasticTransverseIsotropic	node	*Datastructure: PorousElasticTransverseIsotropic*
PorousExtendedDruckerPrager	node	*Datastructure: PorousExtendedDruckerPrager*
PorousModifiedCamClay	node	*Datastructure: PorousModifiedCamClay*
PressurePorosity	node	*Datastructure: PressurePorosity*
ProppantPermeability	node	*Datastructure: ProppantPermeability*
ProppantPorosity	node	*Datastructure: ProppantPorosity*
ProppantSlurryFluid	node	*Datastructure: ProppantSlurryFluid*
ProppantSolidProppantPermeability	node	*Datastructure: ProppantSolidProppantPermeability*
ReactiveBrine	node	*Datastructure: ReactiveBrine*
ReactiveBrineThermal	node	*Datastructure: ReactiveBrineThermal*
SinglePhaseConstantThermalConductivity	node	*Datastructure: SinglePhaseConstantThermalConductivity*
SlipDependentPermeability	node	*Datastructure: SlipDependentPermeability*
SolidInternalEnergy	node	*Datastructure: SolidInternalEnergy*
TableCapillaryPressure	node	*Datastructure: TableCapillaryPressure*
TableRelativePermeability	node	*Datastructure: TableRelativePermeability*
TableRelativePermeabilityHysteresis	node	*Datastructure: TableRelativePermeabilityHysteresis*
ThermalCompressibleSinglePhaseFluid	node	*Datastructure: ThermalCompressibleSinglePhaseFluid*
VanGenuchtenBakerRelativePermeability	node	*Datastructure: VanGenuchtenBakerRelativePermeability*
VanGenuchtenCapillaryPressure	node	*Datastructure: VanGenuchtenCapillaryPressure*
ViscoDruckerPrager	node	*Datastructure: ViscoDruckerPrager*
ViscoExtendedDruckerPrager	node	*Datastructure: ViscoExtendedDruckerPrager*
ViscoModifiedCamClay	node	*Datastructure: ViscoModifiedCamClay*
WillisRichardsPermeability	node	*Datastructure: WillisRichardsPermeability*

Datastructure: ConstitutiveModels

Name	Type	Description

Datastructure: Coulomb

Name	Type	Description
elasticSlip	real64_array2d	Elastic Slip

Datastructure: CustomPolarObject

Name	Type	Description

Datastructure: Cylinder

Name	Type	Description

Datastructure: DamageElasticIsotropic

Name	Type	Description
bulkModulus	real64_array	Elastic Bulk Modulus Field
damage	real64_array2d	Material Damage Variable
density	real64_array2d	Material Density
extDrivingForce	real64_array2d	External Driving Force
oldStress	real64_array3d	Previous Material Stress
shearModulus	real64_array	Elastic Shear Modulus Field
strainEnergyDensity	real64_array2d	Strain Energy Density
stress	real64_array3d	Current Material Stress
thermalExpansionCoefficient	real64_array	Linear Thermal Expansion Coefficient Field

Datastructure: DamageSpectralElasticIsotropic

Name	Type	Description
bulkModulus	real64_array	Elastic Bulk Modulus Field
damage	real64_array2d	Material Damage Variable
density	real64_array2d	Material Density
extDrivingForce	real64_array2d	External Driving Force
oldStress	real64_array3d	Previous Material Stress
shearModulus	real64_array	Elastic Shear Modulus Field
strainEnergyDensity	real64_array2d	Strain Energy Density
stress	real64_array3d	Current Material Stress
thermalExpansionCoefficient	real64_array	Linear Thermal Expansion Coefficient Field

Datastructure: DamageVolDevElasticIsotropic

Name	Type	Description
bulkModulus	real64_array	Elastic Bulk Modulus Field
damage	real64_array2d	Material Damage Variable
density	real64_array2d	Material Density
extDrivingForce	real64_array2d	External Driving Force
oldStress	real64_array3d	Previous Material Stress
shearModulus	real64_array	Elastic Shear Modulus Field
strainEnergyDensity	real64_array2d	Strain Energy Density
stress	real64_array3d	Current Material Stress
thermalExpansionCoefficient	real64_array	Linear Thermal Expansion Coefficient Field

Datastructure: DeadOilFluid

Name	Type	Description
dPhaseCompFraction	LvArray_Array< double, 5, camp_int_seq< long, 0l, 1l, 2l, 3l, 4l >, int, LvArray_ChaiBuffer >	Derivative of phase component fraction with respect to pressure, temperature, and global component fractions
dPhaseDensity	real64_array4d	Derivative of phase density with respect to pressure, temperature, and global component fractions
dPhaseEnthalpy	real64_array4d	Derivative of phase enthalpy with respect to pressure, temperature, and global component fractions
dPhaseFraction	real64_array4d	Derivative of phase fraction with respect to pressure, temperature, and global component fractions
dPhaseInternalEnergy	real64_array4d	Derivative of phase internal energy with respect to pressure, temperature, and global component fractions
dPhaseMassDensity	real64_array4d	Derivative of phase mass density with respect to pressure, temperature, and global component fractions
dPhaseViscosity	real64_array4d	Derivative of phase viscosity with respect to pressure, temperature, and global component fractions
dTotalDensity	real64_array3d	Derivative of total density with respect to pressure, temperature, and global component fractions
formationVolFactorTableWrappers	LvArray_Array< geos_TableFunction_KernelWrapper, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >	(no description available)
hydrocarbonPhaseOrder	integer_array	(no description available)
phaseCompFraction	real64_array4d	Phase component fraction
phaseCompFraction_n	real64_array4d	Phase component fraction at the previous converged time step
phaseDensity	real64_array3d	Phase density
phaseDensity_n	real64_array3d	Phase density at the previous converged time step
phaseEnthalpy	real64_array3d	Phase enthalpy
phaseEnthalpy_n	real64_array3d	Phase enthalpy at the previous converged time step
phaseFraction	real64_array3d	Phase fraction
phaseInternalEnergy	real64_array3d	Phase internal energy
phaseInternalEnergy_n	real64_array3d	Phase internal energy at the previous converged time step
phaseMassDensity	real64_array3d	Phase mass density
phaseOrder	integer_array	(no description available)
phaseTypes	integer_array	(no description available)
phaseViscosity	real64_array3d	Phase viscosity
totalDensity	real64_array2d	Total density
totalDensity_n	real64_array2d	Total density at the previous converged time step
useMass	integer	(no description available)

Datastructure: DelftEgg

Name	Type	Description
bulkModulus	real64_array	Elastic Bulk Modulus Field
cslSlope	real64_array	Slope of the critical state line
density	real64_array2d	Material Density
oldPreConsolidationPressure	real64_array2d	Old preconsolidation pressure
oldStress	real64_array3d	Previous Material Stress
preConsolidationPressure	real64_array2d	New preconsolidation pressure
recompressionIndex	real64_array	Recompression index
shapeParameter	real64_array	Shape parameter for the yield surface
shearModulus	real64_array	Elastic Shear Modulus Field
stress	real64_array3d	Current Material Stress
thermalExpansionCoefficient	real64_array	Linear Thermal Expansion Coefficient Field
virginCompressionIndex	real64_array	Virgin compression index

Datastructure: Dirichlet

Name	Type	Description

Datastructure: Disc

Name	Type	Description

Datastructure: DruckerPrager

Name	Type	Description
bulkModulus	real64_array	Elastic Bulk Modulus Field
cohesion	real64_array2d	New cohesion state
density	real64_array2d	Material Density
dilation	real64_array	Plastic potential slope
friction	real64_array	Yield surface slope
hardening	real64_array	Hardening rate
oldCohesion	real64_array2d	Old cohesion state
oldStress	real64_array3d	Previous Material Stress
shearModulus	real64_array	Elastic Shear Modulus Field
stress	real64_array3d	Current Material Stress
thermalExpansionCoefficient	real64_array	Linear Thermal Expansion Coefficient Field

Datastructure: ElasticFirstOrderSEM

Name	Type	Description
displacementxNp1AtRecei	real32_array2d	Displacement value at each receiver for each timestep (x-components)
displacementyNp1AtRecei	real32_array2d	Displacement value at each receiver for each timestep (y-components)
displacementzNp1AtR	real32_array2d	Displacement value at each receiver for each timestep (z-components)
indexSeismoTrace	integer	Count for output pressure at receivers
maxStableDt	real64	Value of the Maximum Stable Timestep for this solver.
meshTargets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >	MeshBody/Region combinations that the solver will be applied to.
rcvElem	integer_array	Element containing the receivers
receiverIsLocal	integer_array	Flag that indicates whether the receiver is local to this MPI rank
receiverNodeIds	integer_array2d	Indices of the nodes (in the right order) for each receiver point
receiverRegion	integer_array	Region containing the receivers
sourceConstants	real64_array2d	Constant part of the receiver for the nodes listed in m_receiverNodeIds
sourceElem	integer_array	Element containing the sources
sourceIsAccessible	integer_array	Flag that indicates whether the source is local to this MPI rank
sourceNodeId	integer_array2d	Indices of the nodes (in the right order) for each source point
sourceRegion	integer_array	Region containing the sources
sourceValue	real32_array2d	Source Value of the sources
useDAS	integer	Flag to indicate if DAS type of data will be modeled
usePML	integer	Flag to apply PML
LinearSolverParameters	node	*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node	*Datastructure: NonlinearSolverParameters*
SolverStatistics	node	*Datastructure: SolverStatistics*

Datastructure: ElasticIsotropic

Name	Type	Description
bulkModulus	real64_array	Elastic Bulk Modulus Field
density	real64_array2d	Material Density
oldStress	real64_array3d	Previous Material Stress
shearModulus	real64_array	Elastic Shear Modulus Field
stress	real64_array3d	Current Material Stress
thermalExpansionCoefficient	real64_array	Linear Thermal Expansion Coefficient Field

Datastructure: ElasticIsotropicPressureDependent

Name	Type	Description
density	real64_array2d	Material Density
oldStress	real64_array3d	Previous Material Stress
recompressionIndex	real64_array	Recompression Index Field
refPressure	real64	Reference Pressure Field
refStrainVol	real64	Reference Volumetric Strain
shearModulus	real64_array	Elastic Shear Modulus
stress	real64_array3d	Current Material Stress
thermalExpansionCoefficient	real64_array	Linear Thermal Expansion Coefficient Field

Datastructure: ElasticOrthotropic

Name	Type	Description
c11	real64_array	Elastic Stiffness Field C11
c12	real64_array	Elastic Stiffness Field C12
c13	real64_array	Elastic Stiffness Field C13
c22	real64_array	Elastic Stiffness Field C22
c23	real64_array	Elastic Stiffness Field C23
c33	real64_array	Elastic Stiffness Field C33
c44	real64_array	Elastic Stiffness Field C44
c55	real64_array	Elastic Stiffness Field C55
c66	real64_array	Elastic Stiffness Field C66
density	real64_array2d	Material Density
oldStress	real64_array3d	Previous Material Stress
stress	real64_array3d	Current Material Stress
thermalExpansionCoefficient	real64_array	Linear Thermal Expansion Coefficient Field

Datastructure: ElasticSEM

Name	Type	Description
displacementXNp1AtRece	real32_array2d	Displacement value at each receiver for each timestep (x-component)
displacementYNp1AtRece	real32_array2d	Displacement value at each receiver for each timestep (y-component)
displacementZNp1AtR	real32_array2d	Displacement value at each receiver for each timestep (z-component)
indexSeismoTrace	integer	Count for output pressure at receivers
maxStableDt	real64	Value of the Maximum Stable Timestep for this solver.
meshTargets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >	MeshBody/Region combinations that the solver will be applied to.
receiverIsLocal	integer_array	Flag that indicates whether the receiver is local to this MPI rank
receiverNodeIds	integer_array2d	Indices of the nodes (in the right order) for each receiver point
sourceConstants	real64_array2d	Constant part of the receiver for the nodes listed in m_receiverNodeIds
sourceIsAccessible	integer_array	Flag that indicates whether the source is accessible to this MPI rank
sourceNodeIds	integer_array2d	Indices of the nodes (in the right order) for each source point
sourceValue	real32_array2d	Source Value of the sources
useDAS	integer	Flag to indicate if DAS type of data will be modeled
usePML	integer	Flag to apply PML
LinearSolverParameters	node	*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node	*Datastructure: NonlinearSolverParameters*
SolverStatistics	node	*Datastructure: SolverStatistics*

Datastructure: ElasticTransverseIsotropic

Name	Type	Description
c11	real64_array	Elastic Stiffness Field C11
c13	real64_array	Elastic Stiffness Field C13
c33	real64_array	Elastic Stiffness Field C33
c44	real64_array	Elastic Stiffness Field C44
c66	real64_array	Elastic Stiffness Field C66
density	real64_array2d	Material Density
oldStress	real64_array3d	Previous Material Stress
stress	real64_array3d	Current Material Stress
thermalExpansionCoefficient	real64_array	Linear Thermal Expansion Coefficient Field

Datastructure: ElementRegions

Name	Type	Description
domainBoundaryIndicator	integer_array	(no description available)
ghostRank	integer_array	(no description available)
globalToLocalMap	geos_mapBase< long long, int, std_integral_constant< bool, false > >	(no description available)
isExternal	integer_array	(no description available)
localToGlobalMap	globalIndex_array	Array that contains a map from localIndex to globalIndex.
CellElementRegion	node	*Datastructure: CellElementRegion*
SurfaceElementRegion	node	*Datastructure: SurfaceElementRegion*
WellElementRegion	node	*Datastructure: WellElementRegion*
elementRegionsGroup	node	*Datastructure: elementRegionsGroup*
neighborData	node	*Datastructure: neighborData*
sets	node	*Datastructure: sets*

Datastructure: EmbeddedSurfaceGenerator

Name	Type	Registered On	Description
maxStableI	real64		Value of the Maximum Stable Timestep for this solver.
meshTargets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >		MeshBody/Region combinations that the solver will be applied to.
parentEdgeIndex	integer_array	*Datastructure: embeddedSurfacesNodeManager*	Index of parent edge within the mesh object it is registered on.
LinearSolverParameters	node		*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node		*Datastructure: NonlinearSolverParameters*
SolverStatistics	node		*Datastructure: SolverStatistics*

Datastructure: Events

Name	Type	Description
currentSubEvent	integer	Index of the current subevent.
cycle	integer	Current simulation cycle number.
dt	real64	Current simulation timestep.
time	real64	Current simulation time.
HaltEvent	node	*Datastructure: HaltEvent*
PeriodicEvent	node	*Datastructure: PeriodicEvent*
SoloEvent	node	*Datastructure: SoloEvent*

Datastructure: ExponentialDecayPermeability

Name	Type	Description
dPerm_dDispJump	real64_array4d	Derivative of rock permeability with respect to displacement jump
dPerm_dPressure	real64_array3d	Derivative of rock permeability with respect to pressure
dPerm_dTraction	real64_array4d	Derivative of rock permeability with respect to the traction vector
permeability	real64_array3d	Rock permeability

Datastructure: ExtendedDruckerPrager

Name	Type	Description
bulkModulus	real64_array	Elastic Bulk Modulus Field
density	real64_array2d	Material Density
dilationRatio	real64_array	Plastic potential slope ratio
hardening	real64_array	Hardening parameter
initialFriction	real64_array	Initial yield surface slope
oldStateVariable	real64_array2d	Old equivalent plastic shear strain
oldStress	real64_array3d	Previous Material Stress
pressureIntercept	real64_array	Pressure point at cone vertex
residualFriction	real64_array	Residual yield surface slope
shearModulus	real64_array	Elastic Shear Modulus Field
stateVariable	real64_array2d	New equivalent plastic shear strain
stress	real64_array3d	Current Material Stress
thermalExpansionCoefficient	real64_array	Linear Thermal Expansion Coefficient Field

Datastructure: FieldSpecification

Name	Type	Description

Datastructure: FieldSpecifications

Name	Type	Description
Aquifer	node	*Datastructure: Aquifer*
Dirichlet	node	*Datastructure: Dirichlet*
FieldSpecification	node	*Datastructure: FieldSpecification*
HydrostaticEquilibrium	node	*Datastructure: HydrostaticEquilibrium*
PML	node	*Datastructure: PML*
SourceFlux	node	*Datastructure: SourceFlux*
Traction	node	*Datastructure: Traction*

Datastructure: File

Name	Type	Description

Datastructure: FiniteElementSpace

Name	Type	Description

Datastructure: FiniteElements

Name	Type	Description
FiniteElementSpace	node	*Datastructure: FiniteElementSpace*
LinearSolverParameters	node	*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node	*Datastructure: NonlinearSolverParameters*

Datastructure: FiniteVolume

Name	Type	Description
HybridMimeticDiscretization	node	*Datastructure: HybridMimeticDiscretization*
TwoPointFluxApproximation	node	*Datastructure: TwoPointFluxApproximation*

Datastructure: FlowProppantTransport

Name	Type	Description
discretization	string	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
maxSta	real64	Value of the Maximum Stable Timestep for this solver.
meshTa gets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >	MeshBody/Region combinations that the solver will be applied to.
LinearSolverParameters	node	*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node	*Datastructure: NonlinearSolverParameters*
SolverStatistics	node	*Datastructure: SolverStatistics*

Datastructure: FrictionlessContact

Name	Type	Description

Datastructure: Functions

Name	Type	Description
CompositeFunction	node	*Datastructure: CompositeFunction*
MultivariableTableFunction	node	*Datastructure: MultivariableTableFunction*
SymbolicFunction	node	*Datastructure: SymbolicFunction*
TableFunction	node	*Datastructure: TableFunction*

Datastructure: Geometry

Name	Type	Description
Box	node	*Datastructure: Box*
CustomPolarObject	node	*Datastructure: CustomPolarObject*
Cylinder	node	*Datastructure: Cylinder*
Disc	node	*Datastructure: Disc*
Rectangle	node	*Datastructure: Rectangle*
ThickPlane	node	*Datastructure: ThickPlane*

Datastructure: HaltEvent

Name	Type	Description
currentSubEvent	integer	Index of the current subevent
eventForecast	integer	Indicates when the event is expected to execute
isTargetExecuting	integer	Index of the current subevent
lastCycle	integer	Last event occurrence (cycle)
lastTime	real64	Last event occurrence (time)
HaltEvent	node	*Datastructure: HaltEvent*
PeriodicEvent	node	*Datastructure: PeriodicEvent*
SoloEvent	node	*Datastructure: SoloEvent*

Datastructure: HybridMimeticDiscretization

Name	Type	Description

Datastructure: Hydrofracture

Name	Type	Description
discretization	string	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
maxSta	real64	Value of the Maximum Stable Timestep for this solver.
meshTa gets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >	MeshBody/Region combinations that the solver will be applied to.
performStressI tialization	integer	Flag to indicate that the solver is going to perform stress initialization
LinearSolverParameters	node	*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node	*Datastructure: NonlinearSolverParameters*
SolverStatistics	node	*Datastructure: SolverStatistics*

Datastructure: HydrostaticEquilibrium

Name	Type	Description
component	integer	Component of field (if tensor) to apply boundary condition to.
fieldName	string	Name of field that boundary condition is applied to.
initialCondition	integer	Boundary condition is applied as an initial condition.
setNames	string_array	Name of sets that boundary condition is applied to.

Datastructure: Included

Name	Type	Description
File	node	*Datastructure: File*

Datastructure: InternalMesh

Name	Type	Description
InternalWell	node	*Datastructure: InternalWell*
meshLevels	node	*Datastructure: meshLevels*

Datastructure: InternalWell

Name	Type	Description
Perforation	node	*Datastructure: Perforation*

Datastructure: InternalWellbore

Name	Type	Description
nx	integer_array	Number of elements in the x-direction within each mesh block
ny	integer_array	Number of elements in the y-direction within each mesh block
xCoords	real64_array	x-coordinates of each mesh block vertex
yCoords	real64_array	y-coordinates of each mesh block vertex
InternalWell	node	*Datastructure: InternalWell*
meshLevels	node	*Datastructure: meshLevels*

Datastructure: JFunctionCapillaryPressure

Name	Type	Description
dPhaseCapPressure_dPhaseVolFrac	real64_array4d	Derivative of phase capillary pressure with respect to phase volume fraction
jFuncMultiplier	real64_array2d	Multiplier for the Leverett J-function
jFunctionWrappers	LvArray_Array< geos_TableFunction_KernelWrapper, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >	(no description available)
phaseCapPressure	real64_array3d	Phase capillary pressure
phaseOrder	integer_array	(no description available)
phaseTypes	integer_array	(no description available)

Datastructure: LagrangianContact

Name	Type	Description
maxStableDt	real64	Value of the Maximum Stable Timestep for this solver.
meshTargets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >	MeshBody/Region combinations that the solver will be applied to.
LinearSolverParameters	node	*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node	*Datastructure: NonlinearSolverParameters*
SolverStatistics	node	*Datastructure: SolverStatistics*

Datastructure: LaplaceFEM

Name	Type	Description
maxStableDt	real64	Value of the Maximum Stable Timestep for this solver.
meshTargets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >	MeshBody/Region combinations that the solver will be applied to.
LinearSolverParameters	node	*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node	*Datastructure: NonlinearSolverParameters*
SolverStatistics	node	*Datastructure: SolverStatistics*

Datastructure: Level0

Name	Type	Description
meshLevel	integer	(no description available)
ElementRegions	node	*Datastructure: ElementRegions*
edgeManager	node	*Datastructure: edgeManager*
embeddedSurfacesEdgeManager	node	*Datastructure: embeddedSurfacesEdgeManager*
embeddedSurfacesNodeManager	node	*Datastructure: embeddedSurfacesNodeManager*
faceManager	node	*Datastructure: faceManager*
nodeManager	node	*Datastructure: nodeManager*

Datastructure: LinearSolverParameters

Name	Type	Description

Datastructure: Mesh

Name	Type	Description
InternalMesh	node	*Datastructure: InternalMesh*
InternalWellbore	node	*Datastructure: InternalWellbore*
VTKMesh	node	*Datastructure: VTKMesh*

Datastructure: MeshBodies

Name	Type	Description
InternalMesh	node	*Datastructure: InternalMesh*
InternalWellbore	node	*Datastructure: InternalWellbore*
VTKMesh	node	*Datastructure: VTKMesh*

Datastructure: ModifiedCamClay

Name	Type	Description
cslSlope	real64_array	Slope of the critical state line
density	real64_array2d	Material Density
oldPreConsolidationPressure	real64_array2d	Old preconsolidation pressure
oldStress	real64_array3d	Previous Material Stress
preConsolidationPressure	real64_array2d	New preconsolidation pressure
recompressionIndex	real64_array	Recompression Index Field
refPressure	real64	Reference Pressure Field
refStrainVol	real64	Reference Volumetric Strain
shearModulus	real64_array	Elastic Shear Modulus
stress	real64_array3d	Current Material Stress
thermalExpansionCoefficient	real64_array	Linear Thermal Expansion Coefficient Field
virginCompressionIndex	real64_array	Virgin compression index

Datastructure: MultiPhaseConstantThermalConductivity

Name	Type	Description
dEffectiveConductivity_dPhaseVolFraction	real64_array4	Derivative of effective conductivity with respect to phase volume fraction
effectiveConductivity	real64_array3	Effective conductivity

Datastructure: MultiPhaseVolumeWeightedThermalConductivity

Name	Type	Description
dEffectiveConductivity_dPhaseVolFraction	real64_array4	Derivative of effective conductivity with respect to phase volume fraction
effectiveConductivity	real64_array3	Effective conductivity
rockThermalConductivity	real64_array3	Rock thermal conductivity

Datastructure: MultiphasePoromechanics

Name	Type	Description
discretization	string	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
maxSta	real64	Value of the Maximum Stable Timestep for this solver.
meshTa gets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >	MeshBody/Region combinations that the solver will be applied to.
performStressIn tialization	integer	Flag to indicate that the solver is going to perform stress initialization
LinearSolverParameters	node	*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node	*Datastructure: NonlinearSolverParameters*
SolverStatistics	node	*Datastructure: SolverStatistics*

Datastructure: MultiphasePoromechanicsInitialization

Name	Type	Description

Datastructure: MultiphasePoromechanicsReservoir

Name	Type	Description
discretization	string	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
maxSta	real64	Value of the Maximum Stable Timestep for this solver.
meshTargets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >	MeshBody/Region combinations that the solver will be applied to.
LinearSolverParameters	node	*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node	*Datastructure: NonlinearSolverParameters*
SolverStatistics	node	*Datastructure: SolverStatistics*

Datastructure: MultivariableTableFunction

Name	Type	Description

Datastructure: NonlinearSolverParameters

Name	Type	Description
newtonNumberOfIterations	integer	Number of Newton's iterations.
normType	geos_solverBaseKernels_NormType	Norm used by the flow solver to check nonlinear convergence. Valid options: * Linfinity * L2

Datastructure: NullModel

Name	Type	Description

Datastructure: NumericalMethods

Name	Type	Description
FiniteElements	node	*Datastructure: FiniteElements*
FiniteVolume	node	*Datastructure: FiniteVolume*

Datastructure: Outputs

Name	Type	Description
Blueprint	node	*Datastructure: Blueprint*
ChomboIO	node	*Datastructure: ChomboIO*
Python	node	*Datastructure: Python*
Restart	node	*Datastructure: Restart*
Silo	node	*Datastructure: Silo*
TimeHistory	node	*Datastructure: TimeHistory*
VTK	node	*Datastructure: VTK*

Datastructure: PML

Name	Type	Description
fieldName	string	Name of field that boundary condition is applied to.
initialCondition	integer	Boundary condition is applied as an initial condition.

Datastructure: PVTDriver

Name	Type	Description

Datastructure: PackCollection

Name	Type	Description

Datastructure: ParallelPlatesPermeability

Name	Type	Description
dPerm_dDispJump	real64_array4d	Derivative of rock permeability with respect to displacement jump
dPerm_dPressure	real64_array3d	Derivative of rock permeability with respect to pressure
permeability	real64_array3d	Rock permeability

Datastructure: Parameter

Name	Type	Description

Datastructure: Parameters

Name	Type	Description
Parameter	node	*Datastructure: Parameter*

Datastructure: ParticleFluid

Name	Type	Description
collisionFactor	real64_array	Collision factor
dCollisionFactor_dProppantConcentration	real64_array	Derivative of collision factor with respect to proppant concentration
dSettlingFactor_dComponentConcentration	real64_array2d	Derivative of settling factor with respect to component concentration
dSettlingFactor_dPressure	real64_array	Derivative of settling factor with respect to pressure
dSettlingFactor_dProppantConcentration	real64_array	Derivative of settling factor with respect to proppant concentration
proppantPackPermeability	real64_array	Proppant pack permeability
settlingFactor	real64_array	Settling factor

Datastructure: Perforation

Name	Type	Description

Datastructure: PeriodicEvent

Name	Type	Description
currentSubEvent	integer	Index of the current subevent
eventForecast	integer	Indicates when the event is expected to execute
isTargetExecuting	integer	Index of the current subevent
lastCycle	integer	Last event occurrence (cycle)
lastTime	real64	Last event occurrence (time)
HaltEvent	node	*Datastructure: HaltEvent*
PeriodicEvent	node	*Datastructure: PeriodicEvent*
SoloEvent	node	*Datastructure: SoloEvent*

Datastructure: PermeabilityBase

Name	Type	Description
dPerm_dPressure	real64_array3d	Derivative of rock permeability with respect to pressure
permeability	real64_array3d	Rock permeability

Datastructure: PhaseFieldDamageFEM

Name	Type	Description
maxStableDt	real64	Value of the Maximum Stable Timestep for this solver.
meshTargets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >	MeshBody/Region combinations that the solver will be applied to.
LinearSolverParameters	node	*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node	*Datastructure: NonlinearSolverParameters*
SolverStatistics	node	*Datastructure: SolverStatistics*

Datastructure: PhaseFieldFracture

Name	Type	Description
discretization	string	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
maxSta	real64	Value of the Maximum Stable Timestep for this solver.
meshTa gets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >	MeshBody/Region combinations that the solver will be applied to.
LinearSolverParameters	node	*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node	*Datastructure: NonlinearSolverParameters*
SolverStatistics	node	*Datastructure: SolverStatistics*

Datastructure: PorousDelftEgg

Name	Type	Description

Datastructure: PorousDruckerPrager

Name	Type	Description

Datastructure: PorousElasticIsotropic

Name	Type	Description

Datastructure: PorousElasticOrthotropic

Name	Type	Description

Datastructure: PorousElasticTransverseIsotropic

Name	Type	Description

Datastructure: PorousExtendedDruckerPrager

Name	Type	Description

Datastructure: PorousModifiedCamClay

Name	Type	Description

Datastructure: PressurePorosity

Name	Type	Description
dPorosity_dPressure	real64_array2d	Derivative of rock porosity with respect to pressure
dPorosity_dTemperature	real64_array2d	Derivative of rock porosity with respect to temperature
initialPorosity	real64_array2d	Initial porosity
porosity	real64_array2d	Rock porosity
porosity_n	real64_array2d	Rock porosity at the previous converged time step
referencePorosity	real64_array	Reference porosity

Datastructure: Problem

Name	Type	Description
Benchmarks	node	*Datastructure: Benchmarks*
Constitutive	node	*Datastructure: Constitutive*
ElementRegions	node	*Datastructure: ElementRegions*
Events	node	*Datastructure: Events*
FieldSpecifications	node	*Datastructure: FieldSpecifications*
Functions	node	*Datastructure: Functions*
Geometry	node	*Datastructure: Geometry*
Included	node	*Datastructure: Included*
Mesh	node	*Datastructure: Mesh*
NumericalMethods	node	*Datastructure: NumericalMethods*
Outputs	node	*Datastructure: Outputs*
Parameters	node	*Datastructure: Parameters*
Solvers	node	*Datastructure: Solvers*
Tasks	node	*Datastructure: Tasks*
commandLine	node	*Datastructure: commandLine*
domain	node	*Datastructure: domain*

Datastructure: ProppantPermeability

Name	Type	Description
dPerm_dDispJump	real64_array4d	Derivative of rock permeability with respect to displacement jump
dPerm_dPressure	real64_array3d	Derivative of rock permeability with respect to pressure
permeability	real64_array3d	Rock permeability
permeabilityMultiplier	real64_array3d	Rock permeability multiplier
proppantPackPermeability	real64	(no description available)

Datastructure: ProppantPorosity

Name	Type	Description
dPorosity_dPressure	real64_array2d	Derivative of rock porosity with respect to pressure
dPorosity_dTemperature	real64_array2d	Derivative of rock porosity with respect to temperature
initialPorosity	real64_array2d	Initial porosity
porosity	real64_array2d	Rock porosity
porosity_n	real64_array2d	Rock porosity at the previous converged time step
referencePorosity	real64_array	Reference porosity

Datastructure: ProppantSlurryFluid

Name	Type	Description
FluidDensity	real64_array2d	Fluid density
FluidViscosity	real64_array2d	Fluid viscosity
componentDensity	real64_array3d	Component density
dCompDens_dCompConc	real64_array4d	Derivative of component density with respect to component concentration
dCompDens_dPres	real64_array3d	Derivative of component density with respect to pressure
dDens_dCompConc	real64_array3d	Derivative of density with respect to component concentration
dDens_dProppantConc	real64_array2d	Derivative of density with respect to proppant concentration
dDensity_dPressure	real64_array2d	Derivative of density with respect to pressure
dDensity_dTemperature	real64_array2d	Derivative of density with respect to temperature
dEnthalpy_dPressure	real64_array2d	Derivative of enthalpy with respect to pressure
dEnthalpy_dTemperature	real64_array2d	Derivative of enthalpy with respect to temperature
dFluidDens_dCompConc	real64_array3d	Derivative of fluid density with respect to component concentration
dFluidDens_dPres	real64_array2d	Derivative of fluid density with respect to pressure
dFluidVisc_dCompConc	real64_array3d	Derivative of fluid viscosity with respect to component concentration
dFluidVisc_dPres	real64_array2d	Derivative of fluid viscosity with respect to pressure
dInternalEnergy_dPressure	real64_array2d	Derivative of internal energy with respect to pressure
dInternalEnergy_dTemperature	real64_array2d	Derivative of internal energy with respect to temperature
dVisc_dCompConc	real64_array3d	Derivative of viscosity with respect to component concentration
dVisc_dProppantConc	real64_array2d	Derivative of viscosity with respect to proppant concentration
dViscosity_dPressure	real64_array2d	Derivative of viscosity with respect to pressure
dViscosity_dTemperature	real64_array2d	Derivative of viscosity with respect to temperature
density	real64_array2d	Density
density_n	real64_array2d	Density at the previous converged time step
enthalpy	real64_array2d	Enthalpy
internalEnergy	real64_array2d	Internal energy
internalEnergy_n	real64_array2d	Fluid internal energy at the previous converged step
viscosity	real64_array2d	Viscosity

Datastructure: ProppantSolidProppantPermeability

Name	Type	Description

Datastructure: ProppantTransport

Name	Type	Description
maxStableDt	real64	Value of the Maximum Stable Timestep for this solver.
meshTargets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >	MeshBody/Region combinations that the solver will be applied to.
LinearSolverParameters	node	*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node	*Datastructure: NonlinearSolverParameters*
SolverStatistics	node	*Datastructure: SolverStatistics*

Datastructure: Python

Name	Type	Description

Datastructure: ReactiveBrine

Name	Type	Description
dPhaseCompFraction	LvArray_Array< double, 5, camp_int_seq< long, 0l, 1l, 2l, 3l, 4l >, int, LvArray_ChaiBuffer >	Derivative of phase component fraction with respect to pressure, temperature, and global component fractions
dPhaseDensity	real64_array4d	Derivative of phase density with respect to pressure, temperature, and global component fractions
dPhaseEnthalpy	real64_array4d	Derivative of phase enthalpy with respect to pressure, temperature, and global component fractions
dPhaseFraction	real64_array4d	Derivative of phase fraction with respect to pressure, temperature, and global component fractions
dPhaseInternalEnergy	real64_array4d	Derivative of phase internal energy with respect to pressure, temperature, and global component fractions
dPhaseMassDensity	real64_array4d	Derivative of phase mass density with respect to pressure, temperature, and global component fractions
dPhaseViscosity	real64_array4d	Derivative of phase viscosity with respect to pressure, temperature, and global component fractions
dTotalDensity	real64_array3d	Derivative of total density with respect to pressure, temperature, and global component fractions
kineticReactionRates	real64_array2d	kineticReactionRates
phaseCompFraction	real64_array4d	Phase component fraction
phaseCompFraction_n	real64_array4d	Phase component fraction at the previous converged time step
phaseDensity	real64_array3d	Phase density
phaseDensity_n	real64_array3d	Phase density at the previous converged time step
phaseEnthalpy	real64_array3d	Phase enthalpy
phaseEnthalpy_n	real64_array3d	Phase enthalpy at the previous converged time step
phaseFraction	real64_array3d	Phase fraction
phaseInternalEnergy	real64_array3d	Phase internal energy
phaseInternalEnergy_n	real64_array3d	Phase internal energy at the previous converged time step
phaseMassDensity	real64_array3d	Phase mass density
phaseViscosity	real64_array3d	Phase viscosity
primarySpeciesConcentration	real64_array2d	primarySpeciesConcentration
primarySpeciesTotalConcentration	real64_array2d	primarySpeciesTotalConcentration
secondarySpeciesConcentration	real64_array2d	secondarySpeciesConcentration
totalDensity	real64_array2d	Total density
totalDensity_n	real64_array2d	Total density at the previous converged time step
useMass	integer	(no description available)

Datastructure: ReactiveBrineThermal

Name	Type	Description
dPhaseCompFraction	LvArray_Array< double, 5, camp_int_seq< long, 0l, 1l, 2l, 3l, 4l >, int, LvArray_ChaiBuffer >	Derivative of phase component fraction with respect to pressure, temperature, and global component fractions
dPhaseDensity	real64_array4d	Derivative of phase density with respect to pressure, temperature, and global component fractions
dPhaseEnthalpy	real64_array4d	Derivative of phase enthalpy with respect to pressure, temperature, and global component fractions
dPhaseFraction	real64_array4d	Derivative of phase fraction with respect to pressure, temperature, and global component fractions
dPhaseInternalEnergy	real64_array4d	Derivative of phase internal energy with respect to pressure, temperature, and global component fractions
dPhaseMassDensity	real64_array4d	Derivative of phase mass density with respect to pressure, temperature, and global component fractions
dPhaseViscosity	real64_array4d	Derivative of phase viscosity with respect to pressure, temperature, and global component fractions
dTotalDensity	real64_array3d	Derivative of total density with respect to pressure, temperature, and global component fractions
kineticReactionRates	real64_array2d	kineticReactionRates
phaseCompFraction	real64_array4d	Phase component fraction
phaseCompFraction_n	real64_array4d	Phase component fraction at the previous converged time step
phaseDensity	real64_array3d	Phase density
phaseDensity_n	real64_array3d	Phase density at the previous converged time step
phaseEnthalpy	real64_array3d	Phase enthalpy
phaseEnthalpy_n	real64_array3d	Phase enthalpy at the previous converged time step
phaseFraction	real64_array3d	Phase fraction
phaseInternalEnergy	real64_array3d	Phase internal energy
phaseInternalEnergy_n	real64_array3d	Phase internal energy at the previous converged time step
phaseMassDensity	real64_array3d	Phase mass density
phaseViscosity	real64_array3d	Phase viscosity
primarySpeciesConcentration	real64_array2d	primarySpeciesConcentration
primarySpeciesTotalConcentration	real64_array2d	primarySpeciesTotalConcentration
secondarySpeciesConcentration	real64_array2d	secondarySpeciesConcentration
totalDensity	real64_array2d	Total density
totalDensity_n	real64_array2d	Total density at the previous converged time step
useMass	integer	(no description available)

Datastructure: ReactiveCompositionalMultiphaseOBL

Name	Type	Description
maxStableDt	real64	Value of the Maximum Stable Timestep for this solver.
meshTargets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >	MeshBody/Region combinations that the solver will be applied to.
LinearSolverParameters	node	*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node	*Datastructure: NonlinearSolverParameters*
SolverStatistics	node	*Datastructure: SolverStatistics*

Datastructure: ReactiveFluidDriver

Name	Type	Description

Datastructure: Rectangle

Name	Type	Description

Datastructure: RelpermDriver

Name	Type	Description

Datastructure: Restart

Name	Type	Description

Datastructure: Run

Name	Type	Description

Datastructure: Silo

Name	Type	Description

Datastructure: SinglePhaseConstantThermalConductivity

Name	Type	Description
effectiveConductivity	real64_array3d	Effective conductivity

Datastructure: SinglePhaseFVM

Name	Type	Description
maxStableDt	real64	Value of the Maximum Stable Timestep for this solver.
meshTargets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >	MeshBody/Region combinations that the solver will be applied to.
LinearSolverParameters	node	*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node	*Datastructure: NonlinearSolverParameters*
SolverStatistics	node	*Datastructure: SolverStatistics*

Datastructure: SinglePhaseHybridFVM

Name	Type	Registered On	Description
maxStableI	real64		Value of the Maximum Stable Timestep for this solver.
meshTargets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >		MeshBody/Region combinations that the solver will be applied to.
facePressure_n	real64_array	*Datastructure: faceManager*	Face pressure at the previous converged time step
LinearSolverParameters	node		*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node		*Datastructure: NonlinearSolverParameters*
SolverStatistics	node		*Datastructure: SolverStatistics*

Datastructure: SinglePhasePoromechanics

Name	Type	Description
discretization	string	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
maxSta	real64	Value of the Maximum Stable Timestep for this solver.
meshTa gets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >	MeshBody/Region combinations that the solver will be applied to.
performStressI tialization	integer	Flag to indicate that the solver is going to perform stress initialization
LinearSolverParameters	node	*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node	*Datastructure: NonlinearSolverParameters*
SolverStatistics	node	*Datastructure: SolverStatistics*

Datastructure: SinglePhasePoromechanicsConformingFractures

Name	Type	Description
discretization	string	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
maxSta	real64	Value of the Maximum Stable Timestep for this solver.
meshTa gets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >	MeshBody/Region combinations that the solver will be applied to.
LinearSolverParameters	node	*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node	*Datastructure: NonlinearSolverParameters*
SolverStatistics	node	*Datastructure: SolverStatistics*

Datastructure: SinglePhasePoromechanicsEmbeddedFractures

Name	Type	Description
discretization	string	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
maxSta	real64	Value of the Maximum Stable Timestep for this solver.
meshTa gets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >	MeshBody/Region combinations that the solver will be applied to.
performStressI tialization	integer	Flag to indicate that the solver is going to perform stress initialization
LinearSolverParameters	node	*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node	*Datastructure: NonlinearSolverParameters*
SolverStatistics	node	*Datastructure: SolverStatistics*

Datastructure: SinglePhasePoromechanicsInitialization

Name	Type	Description

Datastructure: SinglePhasePoromechanicsReservoir

Name	Type	Description
discretization	string	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
maxSta	real64	Value of the Maximum Stable Timestep for this solver.
meshTa gets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >	MeshBody/Region combinations that the solver will be applied to.
LinearSolverParameters	node	*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node	*Datastructure: NonlinearSolverParameters*
SolverStatistics	node	*Datastructure: SolverStatistics*

Datastructure: SinglePhaseProppantFVM

Name	Type	Description
maxStableDt	real64	Value of the Maximum Stable Timestep for this solver.
meshTargets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >	MeshBody/Region combinations that the solver will be applied to.
LinearSolverParameters	node	*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node	*Datastructure: NonlinearSolverParameters*
SolverStatistics	node	*Datastructure: SolverStatistics*

Datastructure: SinglePhaseReservoir

Name	Type	Description
discretization	string	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
maxSta	real64	Value of the Maximum Stable Timestep for this solver.
meshTa gets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >	MeshBody/Region combinations that the solver will be applied to.
LinearSolverParameters	node	*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node	*Datastructure: NonlinearSolverParameters*
SolverStatistics	node	*Datastructure: SolverStatistics*

Datastructure: SinglePhaseStatistics

Name	Type	Description

Datastructure: SinglePhaseWell

Name	Type	Description
discretization	string	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
maxSta	real64	Value of the Maximum Stable Timestep for this solver.
meshTa gets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >	MeshBody/Region combinations that the solver will be applied to.
LinearSolverParameters	node	*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node	*Datastructure: NonlinearSolverParameters*
SolverStatistics	node	*Datastructure: SolverStatistics*
WellControls	node	*Datastructure: WellControls*

Datastructure: SlipDependentPermeability

Name	Type	Description
dPerm_dDispJump	real64_array4d	Derivative of rock permeability with respect to displacement jump
dPerm_dPressure	real64_array3d	Derivative of rock permeability with respect to pressure
permeability	real64_array3d	Rock permeability

Datastructure: SolidInternalEnergy

Name	Type	Description
dInternalEnergy_dTemperature	real64_array2d	Derivative of the solid internal energy w.r.t. temperature [J/(m^3.K)]
internalEnergy	real64_array2d	Internal energy of the solid per unit volume [J/m^3]
internalEnergy_n	real64_array2d	Internal energy of the solid per unit volume at the previous time-step [J/m^3]

Datastructure: SolidMechanicsEmbeddedFractures

Name	Type	Description
discretization	string	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
maxSta	real64	Value of the Maximum Stable Timestep for this solver.
meshTargets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >	MeshBody/Region combinations that the solver will be applied to.
LinearSolverParameters	node	*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node	*Datastructure: NonlinearSolverParameters*
SolverStatistics	node	*Datastructure: SolverStatistics*

Datastructure: SolidMechanicsLagrangianSSLE

Name	Type	Description
maxForce	real64	The maximum force contribution in the problem domain.
maxStableDt	real64	Value of the Maximum Stable Timestep for this solver.
meshTargets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >	MeshBody/Region combinations that the solver will be applied to.
LinearSolverParameters	node	*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node	*Datastructure: NonlinearSolverParameters*
SolverStatistics	node	*Datastructure: SolverStatistics*

Datastructure: SolidMechanicsStateReset

Name	Type	Description

Datastructure: SolidMechanicsStatistics

Name	Type	Description

Datastructure: SolidMechanics_LagrangianFEM

Name	Type	Description
maxForce	real64	The maximum force contribution in the problem domain.
maxStableDt	real64	Value of the Maximum Stable Timestep for this solver.
meshTargets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >	MeshBody/Region combinations that the solver will be applied to.
LinearSolverParameters	node	*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node	*Datastructure: NonlinearSolverParameters*
SolverStatistics	node	*Datastructure: SolverStatistics*

Datastructure: SoloEvent

Name	Type	Description
currentSubEvent	integer	Index of the current subevent
eventForecast	integer	Indicates when the event is expected to execute
isTargetExecuting	integer	Index of the current subevent
lastCycle	integer	Last event occurrence (cycle)
lastTime	real64	Last event occurrence (time)
HaltEvent	node	*Datastructure: HaltEvent*
PeriodicEvent	node	*Datastructure: PeriodicEvent*
SoloEvent	node	*Datastructure: SoloEvent*

Datastructure: SolverStatistics

Name	Type	Description
numDiscardedLinearIterations	integer	Cumulative number of discarded linear iterations
numDiscardedNonlinearIterations	integer	Cumulative number of discarded nonlinear iterations
numDiscardedOuterLoopIterations	integer	Cumulative number of discarded outer loop iterations
numSuccessfulLinearIterations	integer	Cumulative number of successful linear iterations
numSuccessfulNonlinearIterations	integer	Cumulative number of successful nonlinear iterations
numSuccessfulOuterLoopIterations	integer	Cumulative number of successful outer loop iterations
numTimeStepCuts	integer	Number of time step cuts
numTimeSteps	integer	Number of time steps

Datastructure: Solvers

Name	Type	Description
AcousticFirstOrderSEM	node	*Datastructure: AcousticFirstOrderSEM*
AcousticSEM	node	*Datastructure: AcousticSEM*
AcousticVTISEM	node	*Datastructure: AcousticVTISEM*
CompositionalMultiphaseFVM	node	*Datastructure: CompositionalMultiphaseFVM*
CompositionalMultiphaseHybridFVM	node	*Datastructure: CompositionalMultiphaseHybridFVM*
CompositionalMultiphaseReservoir	node	*Datastructure: CompositionalMultiphaseReservoir*
CompositionalMultiphaseWell	node	*Datastructure: CompositionalMultiphaseWell*
ElasticFirstOrderSEM	node	*Datastructure: ElasticFirstOrderSEM*
ElasticSEM	node	*Datastructure: ElasticSEM*
EmbeddedSurfaceGenerator	node	*Datastructure: EmbeddedSurfaceGenerator*
FlowProppantTransport	node	*Datastructure: FlowProppantTransport*
Hydrofracture	node	*Datastructure: Hydrofracture*
LagrangianContact	node	*Datastructure: LagrangianContact*
LaplaceFEM	node	*Datastructure: LaplaceFEM*
MultiphasePoromechanics	node	*Datastructure: MultiphasePoromechanics*
MultiphasePoromechanicsReservoir	node	*Datastructure: MultiphasePoromechanicsReservoir*
PhaseFieldDamageFEM	node	*Datastructure: PhaseFieldDamageFEM*
PhaseFieldFracture	node	*Datastructure: PhaseFieldFracture*
ProppantTransport	node	*Datastructure: ProppantTransport*
ReactiveCompositionalMultiphaseOBL	node	*Datastructure: ReactiveCompositionalMultiphaseOBL*
SinglePhaseFVM	node	*Datastructure: SinglePhaseFVM*
SinglePhaseHybridFVM	node	*Datastructure: SinglePhaseHybridFVM*
SinglePhasePoromechanics	node	*Datastructure: SinglePhasePoromechanics*
SinglePhasePoromechanicsConformingFractures	node	*Datastructure: SinglePhasePoromechanicsConformingFractures*
SinglePhasePoromechanicsEmbeddedFractures	node	*Datastructure: SinglePhasePoromechanicsEmbeddedFractures*
SinglePhasePoromechanicsReservoir	node	*Datastructure: SinglePhasePoromechanicsReservoir*
SinglePhaseProppantFVM	node	*Datastructure: SinglePhaseProppantFVM*
SinglePhaseReservoir	node	*Datastructure: SinglePhaseReservoir*
SinglePhaseWell	node	*Datastructure: SinglePhaseWell*
SolidMechanicsEmbeddedFractures	node	*Datastructure: SolidMechanicsEmbeddedFractures*
SolidMechanicsLagrangianSSLE	node	*Datastructure: SolidMechanicsLagrangianSSLE*
SolidMechanics_LagrangianFEM	node	*Datastructure: SolidMechanics_LagrangianFEM*
SurfaceGenerator	node	*Datastructure: SurfaceGenerator*

Datastructure: SourceFlux

Name	Type	Description
fieldName	string	Name of field that boundary condition is applied to.

Datastructure: SurfaceElementRegion

Name	Type	Description
domainBoundaryIndicator	integer_array	(no description available)
ghostRank	integer_array	(no description available)
globalToLocalMap	geos_mapBase< long long, int, std_integral_constant< bool, false > >	(no description available)
isExternal	integer_array	(no description available)
localToGlobalMap	globalIndex_array	Array that contains a map from localIndex to globalIndex.
elementSubRegions	node	*Datastructure: elementSubRegions*
neighborData	node	*Datastructure: neighborData*
sets	node	*Datastructure: sets*

Datastructure: SurfaceGenerator

Name	Type	Description
discretization	string	Name of discretization object (defined in the *Numerical Methods*) to use for this solver. For instance, if this is a Finite Element Solver, the name of a *Finite Element Discretization* should be specified. If this is a Finite Volume Method, the name of a *Finite Volume Discretization* discretization should be specified.
failCriterion	integer	(no description available)
maxSta	real64	Value of the Maximum Stable Timestep for this solver.
meshTa gets	geos_mapBase< std_pair< string, string >, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >	MeshBody/Region combinations that the solver will be applied to.
tipEdge	LvArray_SortedArray< int, int, LvArray_ChaiBuffer >	Set containing all the tip edges
tipFaces	LvArray_SortedArray< int, int, LvArray_ChaiBuffer >	Set containing all the tip faces
tipNodes	LvArray_SortedArray< int, int, LvArray_ChaiBuffer >	Set containing all the nodes at the fracture tip
trailingFaces	LvArray_SortedArray< int, int, LvArray_ChaiBuffer >	Set containing all the trailing faces
LinearSolverParameters	node	*Datastructure: LinearSolverParameters*
NonlinearSolverParameters	node	*Datastructure: NonlinearSolverParameters*
SolverStatistics	node	*Datastructure: SolverStatistics*

Datastructure: SymbolicFunction

Name	Type	Description

Datastructure: TableCapillaryPressure

Name	Type	Description
capPresWrappers	LvArray_Array< geos_TableFunction_KernelWrapper, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >	(no description available)
dPhaseCapPressure_dPhaseVolFrac	real64_array4d	Derivative of phase capillary pressure with respect to phase volume fraction
phaseCapPressure	real64_array3d	Phase capillary pressure
phaseOrder	integer_array	(no description available)
phaseTypes	integer_array	(no description available)

Datastructure: TableFunction

Name	Type	Description

Datastructure: TableRelativePermeability

Name	Type	Description
dPhaseRelPerm_dl	real64_array4d	Derivative of phase relative permeability with respect to phase volume fraction
phaseMinVolumeFraction	real64_array	(no description available)
phaseOrder	integer_array	(no description available)
phaseRelPerm	real64_array3d	Phase relative permeability
phaseRelPerm_n	real64_array3d	Phase relative permeability at previous time
phaseTrappedVolFraction	real64_array3d	Phase trapped volume fraction
phaseTypes	integer_array	(no description available)
relPermWrappers	LvArray_Array< geos_TableFunction_KernelWrapper, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >	(no description available)

Datastructure: TableRelativePermeabilityHysteresis

Name	Type	Description
dPhaseRelPerm_dI	real64_array4d	Derivative of phase relative permeability with respect to phase volume fraction
drainagePhaseMaxVolumeFraction	real64_array	(no description available)
drainagePhaseMinVolumeFraction	real64_array	(no description available)
drainagePhaseRelP mEndPoint	real64_array	(no description available)
drainageRelPermW pers	LvArray_Array< geos_TableFunction_KernelWrapper, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >	(no description available)
imbibitionPhaseMaxVolumeFraction	real64_array	(no description available)
imbibitionPhaseMinVolumeFraction	real64_array	(no description available)
imbibitionPhaseRelPermEndPoint	real64_array	(no description available)
imbibitionRelPermWrappers	LvArray_Array< geos_TableFunction_KernelWrapper, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >	(no description available)
landParameter	real64_array	(no description available)
phaseHasHysteresis	integer_array	(no description available)
phaseMaxHistoricalVolFraction	real64_array2d	Phase max historical phase volume fraction
phaseMinHistoricalVolFraction	real64_array2d	Phase min historical phase volume fraction
phaseOrder	integer_array	(no description available)
phaseRelPerm	real64_array3d	Phase relative permeability
phaseRelPerm_n	real64_array3d	Phase relative permeability at previous time
phaseTrappedVolFraction	real64_array3d	Phase trapped volume fraction
phaseTypes	integer_array	(no description available)

Datastructure: Tasks

Name	Type	Description
CompositionalMultiphaseStatistics	node	*Datastructure: CompositionalMultiphaseStatistics*
MultiphasePoromechanicsInitialization	node	*Datastructure: MultiphasePoromechanicsInitialization*
PVTDriver	node	*Datastructure: PVTDriver*
PackCollection	node	*Datastructure: PackCollection*
ReactiveFluidDriver	node	*Datastructure: ReactiveFluidDriver*
RelpermDriver	node	*Datastructure: RelpermDriver*
SinglePhasePoromechanicsInitialization	node	*Datastructure: SinglePhasePoromechanicsInitialization*
SinglePhaseStatistics	node	*Datastructure: SinglePhaseStatistics*
SolidMechanicsStateReset	node	*Datastructure: SolidMechanicsStateReset*
SolidMechanicsStatistics	node	*Datastructure: SolidMechanicsStatistics*
TriaxialDriver	node	*Datastructure: TriaxialDriver*

Datastructure: ThermalCompressibleSinglePhaseFluid

Name	Type	Description
dDensity_dPressure	real64_array2d	Derivative of density with respect to pressure
dDensity_dTemperature	real64_array2d	Derivative of density with respect to temperature
dEnthalpy_dPressure	real64_array2d	Derivative of enthalpy with respect to pressure
dEnthalpy_dTemperature	real64_array2d	Derivative of enthalpy with respect to temperature
dInternalEnergy_dPressure	real64_array2d	Derivative of internal energy with respect to pressure
dInternalEnergy_dTemperature	real64_array2d	Derivative of internal energy with respect to temperature
dViscosity_dPressure	real64_array2d	Derivative of viscosity with respect to pressure
dViscosity_dTemperature	real64_array2d	Derivative of viscosity with respect to temperature
density	real64_array2d	Density
density_n	real64_array2d	Density at the previous converged time step
enthalpy	real64_array2d	Enthalpy
internalEnergy	real64_array2d	Internal energy
internalEnergy_n	real64_array2d	Fluid internal energy at the previous converged step
viscosity	real64_array2d	Viscosity

Datastructure: ThickPlane

Name	Type	Description

Datastructure: TimeHistory

Name	Type	Description
restart	integer	The current history record to be written, on restart from an earlier time allows use to remove invalid future history.

Datastructure: Traction

Name	Type	Description
component	integer	Component of field (if tensor) to apply boundary condition to.
fieldName	string	Name of field that boundary condition is applied to.

Datastructure: TriaxialDriver

Name	Type	Description

Datastructure: TwoPointFluxApproximation

Name	Type	Description
cellStencil	geos_CellElementStencilTPFA	(no description available)
coefficientName	string	Name of coefficient field
edfmStencil	geos_EmbeddedSurfaceToCellStencil	(no description available)
faceElementToCellStencil	geos_FaceElementToCellStencil	(no description available)
fieldName	string_array	Name of primary solution field
fractureStencil	geos_SurfaceElementStencil	(no description available)
targetRegions	geos_mapBase< string, LvArray_Array< string, 1, camp_int_seq< long, 0l >, int, LvArray_ChaiBuffer >, std_integral_constant< bool, true > >	List of regions to build the stencil for

Datastructure: VTK

Name	Type	Description

Datastructure: VTKMesh

Name	Type	Description
InternalWell	node	*Datastructure: InternalWell*
meshLevels	node	*Datastructure: meshLevels*

Datastructure: VanGenuchtenBakerRelativePermeability

Name	Type	Description
dPhaseRelPerm_dPhase	real64_arra	Derivative of phase relative permeability with respect to phase volume fraction
phaseOrder	integer_array	(no description available)
phaseRelPerm	real64_arra	Phase relative permeability
phaseRelPerm_n	real64_arra	Phase relative permeability at previous time
phaseTrappedVolFraction	real64_arra	Phase trapped volume fraction
phaseTypes	integer_array	(no description available)
volFracScale	real64	Factor used to scale the phase capillary pressure, defined as: one minus the sum of the phase minimum volume fractions.

Datastructure: VanGenuchtenCapillaryPressure

Name	Type	Description
dPhaseCapPressure_dPhaseVolFraction	real64_arra	Derivative of phase capillary pressure with respect to phase volume fraction
phaseCapPressure	real64_arra	Phase capillary pressure
phaseOrder	integer_array	(no description available)
phaseTypes	integer_array	(no description available)
volFracScale	real64	Factor used to scale the phase capillary pressure, defined as: one minus the sum of the phase minimum volume fractions.

Datastructure: ViscoDruckerPrager

Name	Type	Description
bulkModulus	real64_array	Elastic Bulk Modulus Field
cohesion	real64_array2d	New cohesion state
density	real64_array2d	Material Density
dilation	real64_array	Plastic potential slope
friction	real64_array	Yield surface slope
hardening	real64_array	Hardening rate
oldCohesion	real64_array2d	Old cohesion state
oldStress	real64_array3d	Previous Material Stress
shearModulus	real64_array	Elastic Shear Modulus Field
stress	real64_array3d	Current Material Stress
thermalExpansionCoefficient	real64_array	Linear Thermal Expansion Coefficient Field

Datastructure: ViscoExtendedDruckerPrager

Name	Type	Description
bulkModulus	real64_array	Elastic Bulk Modulus Field
density	real64_array2d	Material Density
dilationRatio	real64_array	Plastic potential slope ratio
hardening	real64_array	Hardening parameter
initialFriction	real64_array	Initial yield surface slope
oldStateVariable	real64_array2d	Old equivalent plastic shear strain
oldStress	real64_array3d	Previous Material Stress
pressureIntercept	real64_array	Pressure point at cone vertex
residualFriction	real64_array	Residual yield surface slope
shearModulus	real64_array	Elastic Shear Modulus Field
stateVariable	real64_array2d	New equivalent plastic shear strain
stress	real64_array3d	Current Material Stress
thermalExpansionCoefficient	real64_array	Linear Thermal Expansion Coefficient Field

Datastructure: ViscoModifiedCamClay

Name	Type	Description
cslSlope	real64_array	Slope of the critical state line
density	real64_array2d	Material Density
oldPreConsolidationPressure	real64_array2d	Old preconsolidation pressure
oldStress	real64_array3d	Previous Material Stress
preConsolidationPressure	real64_array2d	New preconsolidation pressure
recompressionIndex	real64_array	Recompression Index Field
refPressure	real64	Reference Pressure Field
refStrainVol	real64	Reference Volumetric Strain
shearModulus	real64_array	Elastic Shear Modulus
stress	real64_array3d	Current Material Stress
thermalExpansionCoefficient	real64_array	Linear Thermal Expansion Coefficient Field
virginCompressionIndex	real64_array	Virgin compression index

Datastructure: WellControls

Name	Type	Description
currentControl	geos_WellControls_Control	Current well control

Datastructure: WellElementRegion

Name	Type	Description
domainBoundaryIndicator	integer_array	(no description available)
ghostRank	integer_array	(no description available)
globalToLocalMap	geos_mapBase< long long, int, std_integral_constant< bool, false > >	(no description available)
isExternal	integer_array	(no description available)
localToGlobalMap	globalIndex_array	Array that contains a map from localIndex to globalIndex.
wellControlsName	string	(no description available)
wellGeneratorName	string	(no description available)
elementSubRegions	node	*Datastructure: elementSubRegions*
neighborData	node	*Datastructure: neighborData*
sets	node	*Datastructure: sets*

Datastructure: WellElementRegionUniqueSubRegion

Name	Type	Description
domainBoundaryIndicator	integer_array	(no description available)
elementCenter	real64_array2d	(no description available)
elementVolume	real64_array	(no description available)
ghostRank	integer_array	(no description available)
globalToLocalMap	geos_mapBase< long long, int, std_integral_constant< bool, false > >	(no description available)
isExternal	integer_array	(no description available)
localToGlobalMap	globalIndex_array	Array that contains a map from localIndex to globalIndex.
nextWellElementIndex	integer_array	(no description available)
nextWellElementIndexGlobal	integer_array	(no description available)
nodeList	geos_InterObjectRelation< LvArray_Array< int, 2, camp_int_seq< long, 0l, 1l >, int, LvArray_ChaiBuffer > >	(no description available)
numEdgesPerElement	integer	(no description available)
numFacesPerElement	integer	(no description available)
numNodesPerElement	integer	(no description available)
radius	real64_array	(no description available)
topRank	integer	(no description available)
topWellElementIndex	integer	(no description available)
wellControlsName	string	(no description available)
ConstitutiveModels	node	*Datastructure: ConstitutiveModels*
neighborData	node	*Datastructure: neighborData*
sets	node	*Datastructure: sets*
wellElementSubRegion	node	*Datastructure: wellElementSubRegion*

Datastructure: WillisRichardsPermeability

Name	Type	Description
dPerm_dDispJump	real64_array4d	Derivative of rock permeability with respect to displacement jump
dPerm_dPressure	real64_array3d	Derivative of rock permeability with respect to pressure
dPerm_dTraction	real64_array4d	Derivative of rock permeability with respect to the traction vector
permeability	real64_array3d	Rock permeability

Datastructure: commandLine

Name	Type	Description
beginFromRestart	integer	Flag to indicate restart run.
inputFileName	string	Name of the input xml file.
outputDirectory	string	Directory in which to put the output files, if not specified defaults to the current directory.
overridePartitionNumbers	integer	Flag to indicate partition number override
problemName	string	Used in writing the output files, if not specified defaults to the name of the input file.
restartFileName	string	Name of the restart file.
schemaFileName	string	Name of the output schema
suppressPinned	integer	Whether to disallow using pinned memory allocations for MPI communication buffers.
useNonblockingMPI	integer	Whether to prefer using non-blocking MPI communication where implemented (results in non-deterministic DOF numbering).
xPartitionsOverride	integer	Number of partitions in the x-direction
yPartitionsOverride	integer	Number of partitions in the y-direction
zPartitionsOverride	integer	Number of partitions in the z-direction

Datastructure: crusher

Name	Type	Description
Run	node	*Datastructure: Run*

Datastructure: domain

Name	Type	Description
Neighbors	std_vector< geos_NeighborCommunicator, std_allocator< geos_NeighborCommunicator > >	(no description available)
partitionManager	geos_PartitionBase	(no description available)
Constitutive	node	*Datastructure: Constitutive*
MeshBodies	node	*Datastructure: MeshBodies*

Datastructure: edgeManager

Name	Type	Description
domainBoundaryIndicator	integer_array	(no description available)
faceList	geos_InterObjectRelation< LvArray_ArrayOfSets< int, int, LvArray_ChaiBuffer > >	(no description available)
ghostRank	integer_array	(no description available)
globalToLocalMap	geos_mapBase< long long, int, std_integral_constant< bool, false > >	(no description available)
isExternal	integer_array	(no description available)
localToGlobalMap	globalIndex_array	Array that contains a map from localIndex to globalIndex.
nodeList	geos_InterObjectRelation< LvArray_Array< int, 2, camp_int_seq< long, 0l, 1l >, int, LvArray_ChaiBuffer > >	(no description available)
neighborData	node	*Datastructure: neighborData*
sets	node	*Datastructure: sets*

Datastructure: elementRegionsGroup

Name	Type	Description

Datastructure: elementSubRegions

Name	Type	Description
WellElementRegionUniqueSubRegion	node	*Datastructure: WellElementRegionUniqueSubRegion*

Datastructure: embeddedSurfacesEdgeManager

Name	Type	Description
domainBoundaryIndicator	integer_array	(no description available)
faceList	geos_InterObjectRelation< LvArray_ArrayOfSets< int, int, LvArray_ChaiBuffer > >	(no description available)
ghostRank	integer_array	(no description available)
globalToLocalMap	geos_mapBase< long long, int, std_integral_constant< bool, false > >	(no description available)
isExternal	integer_array	(no description available)
localToGlobalMap	globalIndex_array	Array that contains a map from localIndex to globalIndex.
nodeList	geos_InterObjectRelation< LvArray_Array< int, 2, camp_int_seq< long, 0l, 1l >, int, LvArray_ChaiBuffer > >	(no description available)
neighborData	node	*Datastructure: neighborData*
sets	node	*Datastructure: sets*

Datastructure: embeddedSurfacesNodeManager

Name	Type	Registered By	Description
domainBoundaryIndicator	integer_array		(no description available)
edgeList	geos_InterObjectRelation< LvArray_ArrayOfSets< int, int, LvArray_ChaiBuffer > >		(no description available)
elemList	LvArray_ArrayOfArrays< int, int, LvArray_ChaiBuffer >		(no description available)
elemRegionList	LvArray_ArrayOfArrays< int, int, LvArray_ChaiBuffer >		(no description available)
elemSubRegionList	LvArray_ArrayOfArrays< int, int, LvArray_ChaiBuffer >		(no description available)
ghostRank	integer_array		(no description available)
globalToLocalMap	geos_mapBase< long long, int, std_integral_constant< bool, false > >		(no description available)
isExternal	integer_array		(no description available)
localToGlobalMap	globalIndex_array		Array that contains a map from localIndex to globalIndex.
parentEdgeGlobalIndex	globalIndex_array		(no description available)
referencePosition	real64_array2d		(no description available)
parentEdgeIndex	integer_array	*Datastructure: EmbeddedSurfaceGenerator*	Index of parent edge within the mesh object it is registered on.
neighborData	node		*Datastructure: neighborData*
sets	node		*Datastructure: sets*

Datastructure: faceManager

Name	Type	Registered By	Description
domainBoundaryIndicator	integer_array		(no description available)
edgeList	geos_InterObjectRelation< LvArray_ArrayOfArrays< int, int, LvArray_ChaiBuffer > >		(no description available)
elemList	integer_array2d		(no description available)
elemRegionList	integer_array2d		(no description available)
elemSubRegionList	integer_array2d		(no description available)
faceArea	real64_array		(no description available)
faceCenter	real64_array2d		(no description available)
faceNormal	real64_array2d		(no description available)
ghostRank	integer_array		(no description available)
globalToLocalMap	geos_mapBase< long long, int, std_integral_constant< bool, false > >		(no description available)
isExternal	integer_array		(no description available)
localToGlobalMap	globalIndex_array		Array that contains a map from localIndex to globalIndex.
nodeList	geos_InterObjectRelation< LvArray_ArrayOfArrays< int, int, LvArray_ChaiBuffer > >		(no description available)
facePressure_n	real64_array	*Datastructure: CompositionalMultiphaseHybridFVM*, *Datastructure: SinglePhaseHybridFVM*	Face pressure at the previous converged time step
mimGravityCoefficient	real64_array	*Datastructure: CompositionalMultiphaseHybridFVM*	Mimetic gravity coefficient
neighborData	node		*Datastructure: neighborData*
sets	node		*Datastructure: sets*

Datastructure: lassen

Name	Type	Description
Run	node	*Datastructure: Run*

Datastructure: meshLevels

Name	Type	Description
Level0	node	*Datastructure: Level0*

Datastructure: neighborData

Name	Type	Description

Datastructure: nodeManager

Name	Type	Description
ReferencePosition	real64_array2d	(no description available)
domainBoundaryIndicator	integer_array	(no description available)
edgeList	geos_InterObjectRelation< LvArray_ArrayOfSets< int, int, LvArray_ChaiBuffer > >	(no description available)
elemList	LvArray_ArrayOfArrays< int, int, LvArray_ChaiBuffer >	(no description available)
elemRegionList	LvArray_ArrayOfArrays< int, int, LvArray_ChaiBuffer >	(no description available)
elemSubRegionList	LvArray_ArrayOfArrays< int, int, LvArray_ChaiBuffer >	(no description available)
faceList	geos_InterObjectRelation< LvArray_ArrayOfSets< int, int, LvArray_ChaiBuffer > >	(no description available)
ghostRank	integer_array	(no description available)
globalToLocalMap	geos_mapBase< long long, int, std_integral_constant< bool, false > >	(no description available)
isExternal	integer_array	(no description available)
localToGlobalMap	globalIndex_array	Array that contains a map from localIndex to globalIndex.
primaryField	real64_array	Primary field variable
neighborData	node	*Datastructure: neighborData*
sets	node	*Datastructure: sets*

Datastructure: quartz

Name	Type	Description
Run	node	*Datastructure: Run*

Datastructure: sets

Name	Type	Description
externalSet	LvArray_SortedArray< int, int, LvArray_ChaiBuffer >	(no description available)

Datastructure: wellElementSubRegion

Name	Type	Description
domainBoundaryIndicator	integer_array	(no description available)
ghostRank	integer_array	(no description available)
globalToLocalMap	geos_mapBase< long long, int, std_integral_constant< bool, false > >	(no description available)
isExternal	integer_array	(no description available)
localToGlobalMap	globalIndex_array	Array that contains a map from localIndex to globalIndex.
location	real64_array2d	For each perforation, physical location (x,y,z coordinates)
numPerforationsGlobal	globalIndex	(no description available)
reservoirElementIndex	integer_array	For each perforation, element index of the perforated element
reservoirElementRegion	integer_array	For each perforation, elementRegion index of the perforated element
reservoirElementSubregion	integer_array	For each perforation, elementSubRegion index of the perforated element
wellElementIndex	integer_array	For each perforation, index of the well element
wellTransmissibility	real64_array	For each perforation, well transmissibility
neighborData	node	*Datastructure: neighborData*
sets	node	*Datastructure: sets*

1.11 Contributors

An up-to-date list of all GEOS contributors can be found on our Github page:

GEOS Contributors

The following is the list of GEOS contributors as of January 2019:

Author Name	Affiliation
Nicola Castelletto	*Atmospheric, Earth, and Energy Division, Lawrence Livermore National Laboratory*
Benjamin Corbett	*Applications, Simulations, and Quality Division, Lawrence Livermore National Laboratory*
Matthias Cremon	*Department of Energy Resources Engineering, Stanford University*
Pengcheng Fu	*Atmospheric, Earth, and Energy Division, Lawrence Livermore National Laboratory*
Hervé Gross	*Total*
François Hamon	*Total*
Jixiang Huang	*Atmospheric, Earth, and Energy Division, Lawrence Livermore National Laboratory*
Sergey Klevtsov	*Department of Energy Resources Engineering, Stanford University*
Alexandre Lapene	*Total*
Antoine Mazuyer	*Department of Energy Resources Engineering, Stanford University*
Shabnam Semnani	*Atmospheric, Earth, and Energy Division, Lawrence Livermore National Laboratory*
Randolph Settgast	*Atmospheric, Earth, and Energy Division, Lawrence Livermore National Laboratory*
Christopher Sherman	*Atmospheric, Earth, and Energy Division, Lawrence Livermore National Laboratory*
Arturo Vargas	*Applications, Simulations, and Quality Division, Lawrence Livermore National Laboratory*
Joshua A. White	*Atmospheric, Earth, and Energy Division, Lawrence Livermore National Laboratory*
Christopher White	*Applications, Simulations, and Quality Division, Lawrence Livermore National Laboratory*

1.12 Publications

Last updated 5-September-2023

1.12.1 Preprints and Early-Views

1.12.2 2023

- - **A phase-field model for hydraulic fracture nucleation and propagation in porous media**
    F Fei, A Costa, JE Dolbow, R Settgast, M Cusini
    International Journal for Numerical and Analytical Methods in Geomechanics
    doi.org/10.1002/nag.3612

1.12.3 2022

Smooth implicit hybrid upwinding for compositional multiphase flow in porous media
SBM Bosma, FP Hamon, BT Mallison, HA Tchelepi
Computer Methods in Applied Mechanics and Engineering
doi:10.1016/j.cma.2021.114288

A Multi-resolution approach to hydraulic fracture simulation
A Costa, M Cusini, T Jin, R Settgast, JE Dolbow
International Journal of Fracture
doi:10.1007/s10704-022-00662-y

Phase-field modeling of rock fractures with roughness
F Fei, J Choo, C Liu, JA White
International Journal for Numerical and Analytical Methods in Geomechanics
doi:10.1002/nag.3317

Scalable preconditioning for the stabilized contact mechanics problem
A Franceschini, N Castelletto, JA White, HA Tchelepi
Journal of Computational Physics
doi:10.1016/j.jcp.2022.111150

A scalable preconditioning framework for stabilized contact mechanics with hydraulically active fractures
A Franceschini, L Gazzola, M Ferronato
Journal of Computational Physics
doi:10.1016/j.jcp.2022.111276

Enhanced Relaxed Physical Factorization preconditioner for coupled poromechanics
M Frigo, N Castelletto, M Ferronato
Computers & Mathematics with Applications
doi:10.1016/j.camwa.2021.11.015

Validation and Application of a Three-Dimensional Model for Simulating Proppant Transport and Fracture Conductivity
J Huang, Y Hao, RR Settgast, JA White, K Mateen, H Gross
Rock Mechanics and Rock Engineering
doi:10.1007/s00603-022-03092-3

An aggregation-based nonlinear multigrid solver for two-phase flow and transport in porous media
CS Lee, F Hamon, N Castelletto, PS Vassilevski, JA White
Computers & Mathematics with Applications
doi:10.1016/j.camwa.2022.03.026

1.12.4 2021

Hybrid mimetic finite-difference and virtual element formulation for coupled poromechanics
A Borio, FP Hamon, N Castelletto, JA White, RR Settgast
Computer Methods in Applied Mechanics and Engineering
doi:10.1016/j.cma.2021.113917

Enhanced multiscale restriction-smoothed basis (MsRSB) preconditioning with applications to porous media flow and geomechanics
SBM Bosma, S Klevtsov, O Møyner, N Castelletto
Journal of Computational Physics
doi:10.1016/j.jcp.2020.109934

Multigrid reduction preconditioning framework for coupled processes in porous and fractured media
QM Bui, FP Hamon, N Castelletto, D Osei-Kuffuor, RR Settgast, JA White
Computer Methods in Applied Mechanics and Engineering
doi:10.1016/j.cma.2021.114111

A macroelement stabilization for mixed finite element/finite volume discretizations of multiphase poromechanics
JT Camargo, JA White, RI Borja
Computational Geosciences
doi:10.1007/s10596-020-09964-3

Preconditioners for multiphase poromechanics with strong capillarity
JT Camargo, JA White, N Castelletto, RI Borja
International Journal for Numerical and Analytical Methods in Geomechanics
doi:10.1002/nag.3192

An anisotropic viscoplasticity model for shale based on layered microstructure homogenization
J Choo, SJ Semnani, JA White
International Journal for Numerical and Analytical Methods in Geomechanics
doi:10.1002/nag.3167

Simulation of coupled multiphase flow and geomechanics in porous media with embedded discrete fractures
M Cusini, JA White, N Castelletto, RR Settgast
International Journal for Numerical and Analytical Methods in Geomechanics
doi:10.1002/nag.3168

Approximate inverse-based block preconditioners in poroelasticity
A Franceschini, N Castelletto, M Ferronato
Computational Geosciences
doi:10.1007/s10596-020-09981-2

1.12.5 2020

Scalable multigrid reduction framework for multiphase poromechanics of heterogeneous media
QM Bui, D Osei-Kuffuor, N Castelletto, JA White
SIAM Journal on Scientific Computing
doi:10.1137/19M1256117

Multi-stage preconditioners for thermal–compositional–reactive flow in porous media
MA Cremon, N Castelletto, JA White
Journal of Computational Physics
doi:10.1016/j.jcp.2020.109607

Algebraically stabilized Lagrange multiplier method for frictional contact mechanics with hydraulically active fractures
A Franceschini, N Castelletto, JA White, HA Tchelepi
Computer Methods in Applied Mechanics and Engineering
doi:10.1016/j.cma.2020.113161

Fully implicit multidimensional hybrid upwind scheme for coupled flow and transport
F Hamon, B Mallison
Computer Methods in Applied Mechanics and Engineering
doi:10.1016/j.cma.2019.112606

Nonlinear multigrid based on local spectral coarsening for heterogeneous diffusion problems
CS Lee, F Hamon, N Castelletto, PS Vassilevski, JA White
Computer Methods in Applied Mechanics and Engineering
doi:10.1016/j.cma.2020.113432

An inelastic homogenization framework for layered materials with planes of weakness
SJ Semnani, JA White
Computer Methods in Applied Mechanics and Engineering
doi:10.1016/j.cma.2020.113221

1.12.6 2019

Multiscale two-stage solver for Biot's poroelasticity equations in subsurface media
N Castelletto, S Klevtsov, H Hajibeygi, HA Tchelepi
Computational Geosciences
doi:10.1007/s10596-018-9791-z

Block preconditioning for fault/fracture mechanics saddle-point problems
A Franceschini, N Castelletto, M Ferronato
Computer Methods in Applied Mechanics and Engineering
doi:10.1016/j.cma.2018.09.039

A relaxed physical factorization preconditioner for mixed finite element coupled poromechanics
M Frigo, N Castelletto, M Ferronato
SIAM Journal on Scientific Computing
doi:10.1137/18M120645X

A two-stage preconditioner for multiphase poromechanics in reservoir simulation
JA White, N Castelletto, S Klevtsov, QM Bui, D Osei-Kuffuor, HA Tchelepi
Computer Methods in Applied Mechanics and Engineering
doi:10.1016/j.cma.2019.112575

1.13 Acknowledgements

GEOS was developed with supporting funds from a number of organizations, including

- Lawrence Livermore National Laboratory
- U.S. Department of Energy, Office of Science
- TotalEnergies.

This support is gratefully acknowledged.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX