TOUGH3-FLAC: python package

Version 1.1.0

Keurfon Luu

Mar 25, 2020

Contents

Chapter 1

Introduction

This document describes how to run a coupled TOUGH3 and FLAC3D simulation for the analysis of coupled multiphase fluid flow and geomechanical processes. This document is intended for research collaborators, scientists and engineers who would like to apply coupled TOUGH3 and FLAC3D simulations in collaborative research projects. The TOUGH-FLAC simulator, which is a coupled code linking two existing simulators (TOUGH3 and FLAC3D), has been developed over the past several years to enable the analysis of coupled thermo-hydro-mechanical processes in geological media under multiphase flow conditions. A great benefit with this approach is that the two codes are well established in their respective fields; they have been applied and tested by many groups all over the world.

The earliest developments are presented in [Rutqvist et al., 2002] and [Rutqvist & Tsang, 2003a], and the approach have been applied to study coupled geomechanical aspects under multiphase flow conditions for a wide range of applications, including nuclear waste disposal ([Rutqvist & Tsang, 2003b], [Rutqvist et al., 2005], [Rutqvist et al., 2008b], [Rutqvist et al., 2009b]), CO_2 sequestration ([Rutqvist & Tsang, 2002], [Rutqvist & Tsang, 2005], [Rutqvist et al., 2007], [Rutqvist et al., 2008a], [Rutqvist et al., 2010], [Cappa & Rutqvist, 2011a], [Cappa & Rutqvist, 2011b]), geothermal energy extraction (e.g. [Rutqvist et al., 2006]), naturally occurring CO_2 upwelling with surface deformations ([Todesco et al., 2004], [Cappa et al., 2009]), gas production from hydrate bearing sediments ([Rutqvist & Moridis, 2007], [Rutqvist et al., 2009a], [Kim et al., 2011]), and underground compressed air energy storage ([Rutqvist et al., 2012], [Kim et al., 2012]).

First, the general outline for developing a coupled TOUGH-FLAC simulation is presented. Then detailed step-by-step hands-on instructions are given to describe development and execution of specific problems.

Chapter 2

Why Python instead of FISH?

In this version of TOUGH-FLAC, coupling has been done using Python. According to Itasca, the benefits of using Python instead of FISH are twofold:

1. Easier scripting using popular scientific numerical computing Python libraries (e.g. NumPy and SciPy included in FLAC3D),
2. Faster computation, especially thanks to vectorized array calculation with NumPy (see Table 2.1).

In addition, thanks to its large community (second most active on GitHub in 2019), users can easily find help online.

In previous versions of TOUGH-FLAC, data between TOUGH and FLAC3D were exchanged by the mean of external ASCII files written by FISH. In TOUGH3-FLAC3D, I/O has been improved by reading and writing TOUGH3 and FLAC3D data as binary files using Python (unformatted files that are faster to read and write).

`toughflac` has been designed to make coupled simulations more user-friendly by extensively using Python capabilities. The package is organized as submodules and functions consistent with FLAC3D own Python module `itasca` in terms of hierarchy and naming. It allows pre- and post-processing for TOUGH-FLAC to be (almost) fully done in FLAC3D.

Table 2.1: Speed up comparison between FISH and Python (according to Itasca).

	FISH	Python	NumPy
Zone loop	1	2.5	34
Gridpoint loop	1	2.5	19
Stress count	1	1.8	2.9
Set extra	1	1.3	6.4
Integer sum	1	4.8	27

Note: FLAC3D v6 embeds Python 2.7 which is no longer maintained and supported by most Python libraries. Some functionalities of `toughflac` (e.g. mesh import/export) depends on such external open-source libraries. Therefore, we recommend the user to upgrade to FLAC3D v7 (if possible) that embeds Python 3.6 to be able to use all of its capabilities.

Chapter 3

Installation

3.1 TOUGH-FLAC executables

To install TOUGH-FLAC, the user only needs to replace few TOUGH3 source files with the modified Fortran files provided. Then, the user can follow the instructions to install TOUGH3 for Cygwin or Windows Subsystem for Linux (WSL):

- Cygwin: when configuring Cygwin, GNU compilers version 7.3.0 (*gcc-core*, *gcc-fortran*, *gcc-g++*) must be selected and *openssh* package installed in addition to the remaining packages required by TOUGH3. Besides, when compiling TOUGH3 for Cygwin, *compile_T3_cygwin.sh* script should **not** have `--parallel=2` option enabled,
- WSL: we recommend to use MPICH instead of OpenMPI. TOUGH3-FLAC can be compiled by running the script *compile_T3_Linux.sh*.

Once installed, TOUGH-FLAC executables can be called anywhere from Cygwin or WSL (only if the file *.bashrc* has been correctly modified) by typing `tough3-eos filename` where `eos` is the equation-of-state identifier (e.g. `eco2n`) and `filename` the TOUGH3 input file (e.g. `INFILE`). TOUGH(-FLAC) can also be run in parallel using MPI (for instance with 4 cores) by executing the command:

```
mpiexec -n 4 tough3-eos filename
```

3.2 TOUGH-FLAC script

The location of FLAC3D executable is not hard coded in TOUGH3-FLAC to allow the user to modify it if FLAC3D is not installed in its default location. Instead, TOUGH3-FLAC uses the Bash script *flac3d.sh* that detects the platform on which it is run (Cygwin or WSL) and calls the appropriate command to open FLAC3D console executable. This script should be callable by TOUGH3-FLAC when running a coupled simulation, i.e. it should either be in the current working directory or set in the `PATH` so that it can be called from anywhere. We recommend the user to put the script in the installation directory of TOUGH3 if the file *bashrc* has been correctly modified.

Note: TOUGH3-FLAC will run the local script in priority if it is found in the current working directory.

3.3 TOUGH-FLAC Python module

TOUGH-FLAC Python module `toughflac` provides submodules and functions for pre- and post-processing for TOUGH-FLAC coupled simulations. The FISH routines in the previous versions of TOUGH-FLAC have been translated into Python and packaged as a submodule `coupling`. FLAC3D offers the possibility to install external Python packages through the package manager associated to its Python environment. By default, it is located in:

```
C:/Program Files/Itasca/Flac3d600/exe64/python27/Scripts/pip.exe
C:/Program Files/Itasca/FLAC3D700/exe64/python36/Scripts/pip.exe
```

and any package can be downloaded and installed from a Windows console:

```
cmd /C ""path/to/flac3d/pip"" install packagename --user
```

For convenience, a single Batch script *setup.bat* is provided to install and uninstall `toughflac`. Once installed, `toughflac` can be imported into FLAC3D embedded IPython console or within a script without having the Python source files in the current working directory:

```python
import toughflac as tf
```

Note: The user may have to edit the path to Python package installer *pip* if FLAC3D is not installed in its default location.

3.4 Update NumPy (FLAC3D v6)

Even though Python is usually faster than FISH (see Table 2.1), pure Python – as a high level interpreted language – is 1 to 2 order of magnitude slower when compared to lower level compiled languages such as C, C++ or Fortran. A simple trick is to compile computationally heavy functions and wrapping them in Python which then allows near optimal speed (minus inherent involved overheads). NumPy and most of optimized public scientific Python libraries rely on functions or libraries written in C or Fortran.

FLAC3D v6 embeds Python 2.7 with an outdated version of NumPy (1.9.3). To fully exploit NumPy vectorized array computation capabilities, it is recommended to update the package `numpy` embedded in FLAC3D:

```
cmd /C ""path/to/flac3d/pip"" install -U numpy --user
```

Note: *Microsoft Visual C++ Compiler for Python 2.7* (http://aka.ms/vcpython27) must be installed beforehand.

Chapter 4

Steps for developing and running a coupled TOUGH3-FLAC3D simulation

A coupled TOUGH3 and FLAC3D analysis for a particular problem is typically developed according to the steps described in this section. The user would start by constructing the numerical grid and input data for TOUGH3 and FLAC3D according to the standard procedures for each code.

4.1 Grid generation and basic input data preparation

The geometry and element numbering should be consistent in TOUGH3 and FLAC3D for a particular problem. This can be achieved by generating the meshes using the standard MESHMAKER module attached to the TOUGH3 code and by a special FISH routine in FLAC3D that is programmed such that it produces the same mesh consistent with the MESHMAKER. Another possibility is to use an external mesh generator (e.g. FEM mesh generator) and routines that can translate this FEM mesh into TOUGH3 and FLAC3D meshes. In TOUGH3, the numerical grid is defined by finite volume elements with one node located within the element boundaries that is connected to neighboring elements center nodes. A FLAC3D numerical grid, on the other hand, corner nodes define the element shapes. With the numerical grids defined, input files are prepared for TOUGH3 and FLAC3D including material properties, boundary and initial conditions.

`toughflac` provides a submodule `zone` to facilitate the generation of TOUGH3 input files consistent with FLAC3D grids. This submodule allows the user to:

- Define initial conditions for all the elements or a group of elements (i.e. with the same rock type),
- Define time-independent Dirichlet boundary conditions for TOUGH3,
- Export MESH and INCON files for TOUGH3 guaranteed to be consistent with current FLAC3D grid (if conformal),
- Import external meshes (e.g. Gmsh) into FLAC3D (requires external library `toughio`).

4.2 Test simulations and establishment of initial conditions

With the input files defined for both TOUGH3 and FLAC3D, analyses should first be conducted to assure that the problem can be solved and that the input data is correctly prepared. A TOUGH3 simulation may be conducted by running one steady state simulation to establish initial conditions, including pressure and thermal gradients, and perhaps saturation profiles. Similarly, a FLAC3D simulation may be conducted to establish initial mechanical stress profiles, if they cannot be exactly defined in the input data. Sometimes, it may be necessary to run a fully coupled TOUGH3 and FLAC3D simulation for establishing initial conditions.

4.3 Set-up of a coupled TOUGH-FLAC simulation

The FLAC3D interactive simulation is prepared by:

1. Starting FLAC3D,
2. Calling the FLAC3D mesh and property file (e.g. call *initialize_model.f3dat*).

The FLAC3D state containing the problem specific grid, material properties and initial and boundary conditions is then saved to the file *tf_in.f3sav*. This will be the initial FLAC3D state when invoking the FLAC3D code for the first time in a coupled TOUGH3-FLAC3D simulation.

4.4 Running a TOUGH-FLAC simulation

To run a coupled TOUGH-FLAC simulation, a new block `FLAC` has been introduced and must be present in the TOUGH input file. The block `FLAC` introduces material parameters and coupling options. It must be specified **after** the block `ROCKS` and contain two records for each rock type in the same order as in the block `ROCKS`, plus an additional record placed after the line containing the keyword `FLAC`.

Record FLAC.1: format (16I5), IE_THM(I), I = 1, 16

	Description	Options
IE_THM(1)	Select creep mode (not implemented yet)	• 0: default mode • 1: creep mode
IE_THM(2)	Select porosity evolution model	• 0: no mechanical-induced porosity change • 1: mechanical-induced porosity change • 2: porosity change as a function of volumetric strain
IE_THM(3)	Select FLAC3D version	• 6: call FLAC3D v6 from its default location • 7: call FLAC3D v7 from its default location • Any other integer: use version defined in coupling script *flac3d.sh*

Record FLAC.2: format (I10, 7E10.4), IHM, HM(I), I = 1, 7

	Description	Options
IHM	Select permeability model	• 0: permeability models defined in the Python script *flac3d.py* • 1: no mechanical-induced permeability change • 2: Minkoff et al. (2004)
HM(I)	Parameter values for selected permeability model	

In the current version, only one permeability model has been implemented in the Fortran source code. This model (IHM = 2) is only provided as a basis to implement additional case-specific models. Nevertheless, several permeability models have been implemented in `toughflac`, and beginners are advised to implement new models directly in Python (IHM = 0).

Record FLAC.3: format (I5, 5X, 7E10.4), ICP_THM, CP_THM(I)

	Description	Options
ICP	Select equivalent pore pressure model (Coussy, 2004)	• 3: Integral term is zero

An example of block `FLAC` for an input file with three different rock types is given below:

```
FLAC ----1----*----2----*----3----*----4----*----5----*----6----*----7----*----8
    0    1    7
         1       0.0       0.0       0.0       0.0       0.0       0.0       0.0
    3            0.0       0.0       0.0       0.0       0.0       0.0       0.0
         1       0.0       0.0       0.0       0.0       0.0       0.0       0.0
    3            0.0       0.0       0.0       0.0       0.0       0.0       0.0
         1       0.0       0.0       0.0       0.0       0.0       0.0       0.0
    3            0.0       0.0       0.0       0.0       0.0       0.0       0.0
```

If the block `FLAC` is not present in the TOUGH3 input file, TOUGH-FLAC will perform a TOUGH3 simulation only (for instance to calculate the steady state before running a coupled simulation as explained in Section 4.2).

In general, the coupled TOUGH3-FLAC3D analysis is executed by running the TOUGH3 code and at desired instances calling FLAC3D to perform a quasi-static mechanical analysis. When calling the FLAC3D code from TOUGH3, the FLAC3D code is activated and automatically looks for an initiation file called *flac3d.dat* that simply calls the Python script *flac3d.py*. The latter script import the function `run()` of the submodule `coupling` in which the user can define the main parameters for the coupling. Below is an example of coupling parameters in *flac3d.py*:

```
run(
    model_save="tf_in.f3sav",
    deterministic=False,
    damping="combined",
    mechanical_ratio=1.0e-7,
    n_threads=8,
)
```

In this case, the FLAC3D convergence criterion or mechanical ratio was set to `1.0e-7` down from the default value of `1.0e-5`. Using more stringent criterion leads to longer simulation runs, but is more accurate.

The FLAC3D code then conducts a few sequential commands, the first one is to restore the current geomechanical conditions stored in a file called *tf_in.f3sav*, thereafter a number of Python functions are invoked which have all been installed in the private submodule `coupling.io`. The final FLAC3D state is saved into *tf_fi.f3sav*.

The following files should reside in the current working directory:

- The TOUGH3 problem specific input file with block `FLAC` correctly defined,
- A file named *tf_in.f3sav* (consistent with the one associated to argument *model_save* in `run()`),
- A Python script *flac3d.py* that calls the function `run()` from the submodule `coupling` with some coupling parameters.

Optionally:

- *MESH* if the blocks `ELEME` and `CONNE` are not included in TOUGH3 input file,
- *INCON* if initial conditions are available (e.g. *SAVE* file from steady state calculation run),
- Additional files required by equation-of-state (e.g. *CO2TAB* for equations-of-state ECO2N, ECO2N v2 and ECO2M).

The TOUGH-FLAC simulation is then started by the usual command as explained in Section 3.1. For instance, for equation of state ECO2N:

```
tough3-eco2n INFILE
```

with `INFILE` the TOUGH input file name, or

```
mpiexec -n 4 tough3-eco2n INFILE
```

to run TOUGH(-FLAC) in parallel using 4 MPI processes.

4.5 TOUGH and FLAC outputs

Outputs from TOUGH3 and FLAC3D are printed and saved at desired times, or time step numbers, as defined in the TOUGH3 input file (in block `TIMES`). When TOUGH3 simulation outputs are printed to the TOUGH3 output file, the current FLAC3D state is saved at the same time to save file named *ftime.f3sav*, where *time* is the current simulation time. This file can later be restored in FLAC3D for plotting and printing. Moreover, a *SAVE* file is written which can be used to restart a TOUGH-FLAC simulation.

4.6 Restart of coupled TOUGH-FLAC simulation

A restart of a coupled TOUGH-FLAC simulation is conducted in the same way as a restart of a TOUGH3 simulation. That is, using the conditions from the *SAVE* file as initial conditions in *INCON*. The FLAC3D save state to restore is the one in *flac3d.py*.

Chapter 5

Examples

Following subsections provide instructions to develop and run two example problems. The first example provides a step-by-step description of the standard development of a problem, from grid generation to output visualization. The second example is less detailed since the main steps are similar to the first problem, and is mainly included to demonstrate the use of advanced features to run a coupled TOUGH-FLAC simulation (e.g. import grid generated by Gmsh, export results to a VTK file. . .).

5.1 CO_2 injection in a reservoir-caprock system

This example describes a problem of injection of CO_2 into a reservoir overlaid by a caprock. The problem was first presented in [Rutqvist & Tsang, 2002]. All the necessary input files can be found in a directory named *2dldV6*. This directory contains the coupling Python script *flac3d.py* and five subdirectories:

- *1_TH_INI* to run a steady state calculation with TOUGH3 only,
- *2_TH* to test input files required for TOUGH3,
- *3_THM* to conduct a fully coupled TOUGH3 and FLAC3D v6 simulation,
- *Preprocessing* that contains scripts to generate TOUGH3 and FLAC3D grids (*generate_mesh.py*) and configure the mechanical model for FLAC3D (*initialize_model.f3grid*),
- *Postprocessing* that contains a sample script to plot history variables.

The three former subdirectories contain the TOUGH3 input file, a file *CO2TAB* needed for equation-of-state ECO2N, and two Bash scripts *clean.sh* and *run.sh*. These two scripts are not necessary and are only here for convenience (facilitate the workflow).

The user might want to start and save a new FLAC3D project in this directory (i.e. *2dldV6*).

5.1.1 Grid generation for TOUGH3 and FLAC3D

Note: The pre-processing steps presented in this section are independent of the version of TOUGH-FLAC. This also means that FLAC3D can be used as a simple pre-processor to conduct a TOUGH simulation.

The first step is to generate a grid consistent for both TOUGH3 and FLAC3D. This step is usually the most time-consuming task from the user end and requires manual edit of TOUGH3 *MESH* file. `toughflac` provides several functions to facilitate and automate several manual labors.

The geometry of the problem is presented in Figure 5.1 and is extended far enough in the lateral direction to be infinite acting.

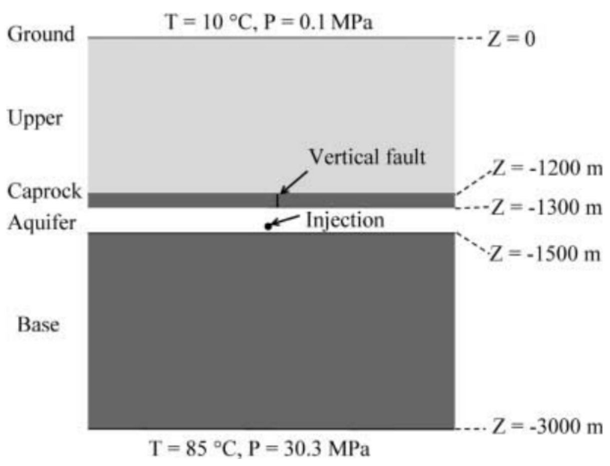


Figure 5.1: Vertical profile of the finite difference model with rock units ([Rutqvist & Tsang, 2002]). Note that the vertical fault is not modeled in this example.

The Python script *generate_mesh.py* is written to produce the grids for TOUGH3 and FLAC3D corresponding to this problem. The user can simply open and execute this script in FLAC3D to produce the desired grids. In the following, we will go through the content of this script line-by-line.

First, import both `itasca` and `toughflac` modules:

```
import itasca as it
import toughflac as tf
```

Generate grid

To generate the grid without any group, similarly to TOUGH3's MESHMAKER, we define the block sizes in X, Y and Z directions using a list (or anything array-like):

```
dx = [
    113255.0,   78894.0,    52596.0,    35064.0,    23376.0,    15584.0,    10389.
↪0,    6926.0,
    # ...
    78894.0,    113255.0,
]
dy = [1.0]
dz = [
    0.1,    250.0,  250.0,  194.0,  180.0,  120.0,  80.0,   53.0,
    # ...
    100.0,  200.0,  400.0,  700.0,  0.1,
][::-1]
```

Since the problem is 2D, `dy` is set to `[1.0]` which means that we want a single 1-meter thick block in Y direction. `dz` is defined from top to bottom. FLAC3D creates grid from bottom to top in a right-handed coordinate system, thus the list `dz` should be reversed (by using `[::-1]` for instance).

We also have to define the origin point of the grid to facilitate later coordinate queries:

```
point_0 = [-350000.0, -0.5, -3000.1]
```

Now, we can call the function *`toughflac.zone.create.tartan_brick()`* provided by the module to create an irregular cartesian grid given the block sizes `dx`, `dy` and `dz` and the origin point `point_0`:

```
tf.zone.create.tartan_brick(dx, dy, dz, point_0=point_0)
```

So far, only the grid points and zones have been created, but no group has been associated to the zones yet (the grid should look like Figure 5.2).

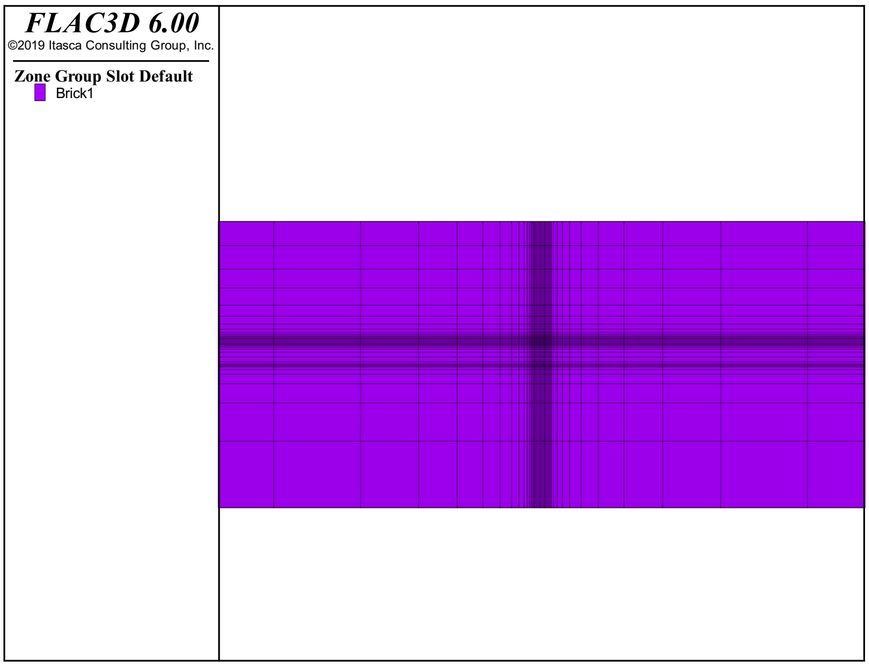


Figure 5.2: FLAC3D grid once created without any group assigned (all zones are assigned to default group *Brick1*.

Assign groups

Groups can be easily assigned using the function *toughflac.zone.group()* that selects zones that are within a specific rectangular region:

```
tf.zone.group("ATMOS", range_z=(0.0, 1.0))
tf.zone.group("UPPER", range_z=(-1200.0, 0.0))
tf.zone.group("CAPRO", range_z=(-1300.0, -1200.0))
tf.zone.group("AQFER", range_z=(-1500.0, -1300.0))
tf.zone.group("BASEM", range_z=(-3000.0, -1500.0))
tf.zone.group("BOTOM", range_z=(-3001.0, -3000.0))
```

The boundary groups `ATMOS` and `BOTOM` have been created to define the top and bottom boundary conditions, respectively. The grid should now look like Figure 5.3.

Define boundary and initial conditions

Usually, time-independent Dirichlet boundary conditions are defined by manually setting the volume of boundary elements to a large value (e.g. 10^{50}) and nodal distances to small value (e.g. 10^{-9}) in TOUGH3 *MESH* file. This task can be automated by using the function *toughflac.zone.set_dirichlet_bc()* and specifying to which zone groups Dirichlet boundary conditions should be applied. Initial conditions can be defined by initializing primary variables as constant or following a gradient with respect to zone depth using the function *toughflac.zone.initialize_pvariables()*.

In this example, the pressure and temperature are set to atmospheric conditions (P = 0.1 MPa and T = 10 deg). The pressure (X1) is set to hydrostatic pressure and the temperature gradient (X4) is 25 deg/km. Salt and CO_2 mass fractions (X2 and X3) are set to 0.05 and 0, respectively.

```
tf.zone.set_dirichlet_bc("ATMOS")
tf.zone.initialize_pvariables(
    x1=lambda z: 1.0e5 - 9810.0 * z,
    x2=lambda z: 0.05,
```

(continues on next page)

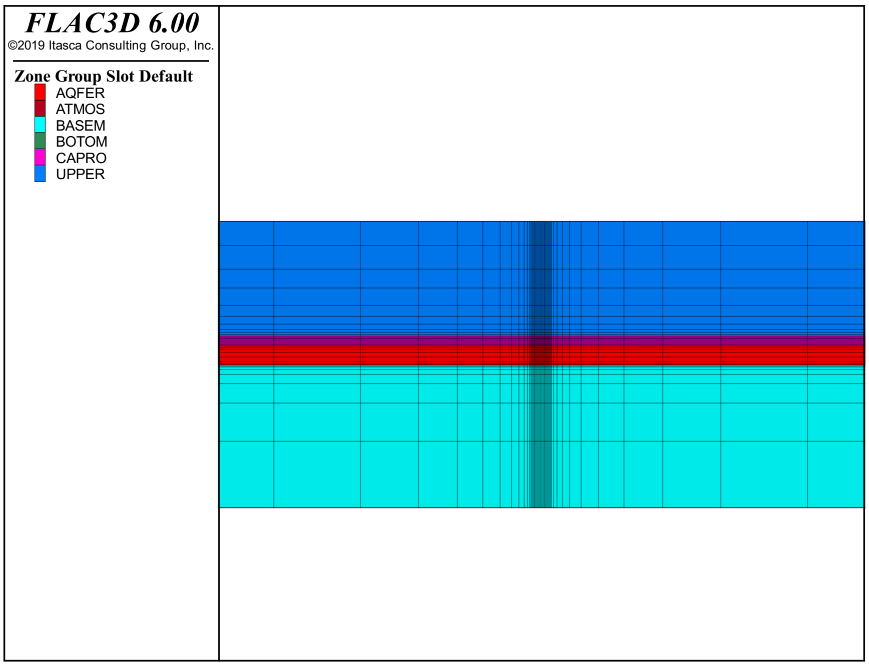


Figure 5.3: FLAC3D grid with all zone groups assigned. Group *Brick1* has disappeared from the left panel.

(continued from previous page)

```
    x3=lambda z: 0.0,
    x4=lambda z: 10.0 - 0.025 * z,
)
```

Tip: Anonymous functions `lambda` are generally used to define temporary functions to be passed to another function.

Export grids for TOUGH and FLAC3D

Finally, TOUGH3 *MESH* and *INCON* files can be written using the function *`toughflac.zone.export_tough()`*:

```
tf.zone.export_tough("MESH", incon=True)
```

The *MESH* file produced using this function does not need to be manually edited: element materials and Dirichlet boundary conditions have already been applied.

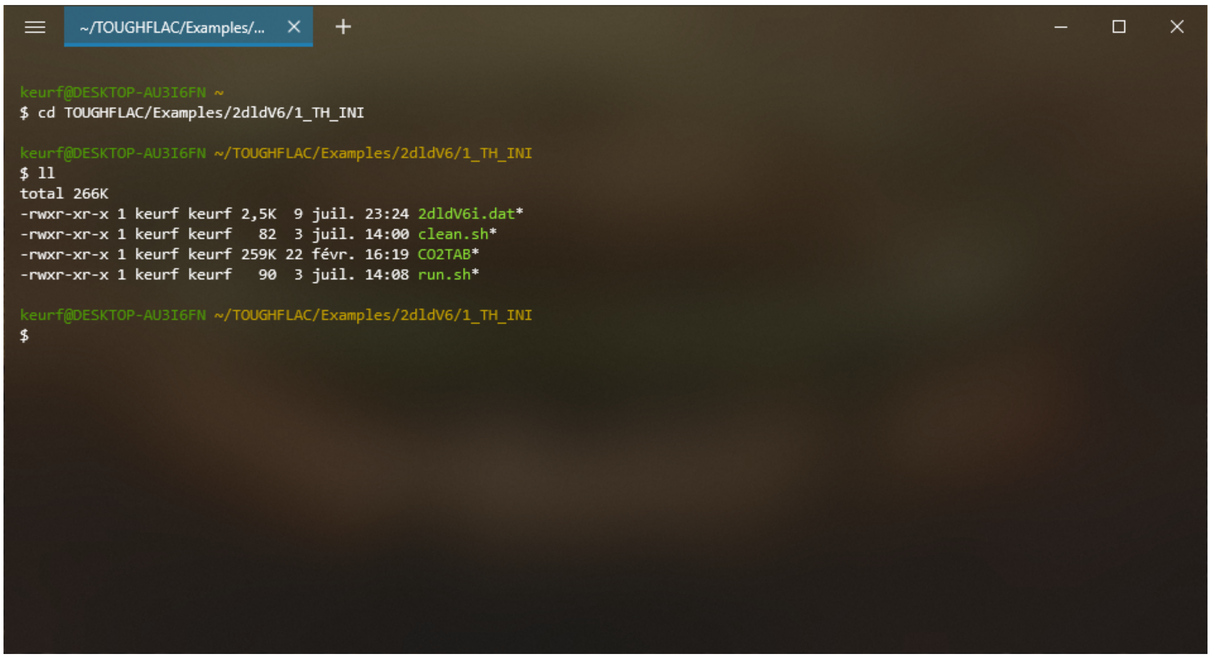
FLAC3D grid can be exported using the function *`toughflac.zone.export_flac()`*:

```
tf.zone.export_flac("mesh.f3grid", binary=True)
```

5.1.2 Calculation of steady state

It is common to run a steady state pre-injection calculation to obtain the correct reservoir temperature, pressure and stress fields at the start of the CO_2 injection.

We first run a TOUGH3 (only) simulation without injection to obtain the correct pressure and temperature conditions. Navigate to the folder *1_TH_INI* (see Figure 5.4).



```
keurf@DESKTOP-AU3I6FN ~
$ cd TOUGHFLAC/Examples/2dldV6/1_TH_INI

keurf@DESKTOP-AU3I6FN ~/TOUGHFLAC/Examples/2dldV6/1_TH_INI
$ ll
total 266K
-rwxr-xr-x 1 keurf keurf 2,5K  9 juil. 23:24 2dldV6i.dat*
-rwxr-xr-x 1 keurf keurf   82  3 juil. 14:00 clean.sh*
-rwxr-xr-x 1 keurf keurf 259K 22 févr. 16:19 CO2TAB*
-rwxr-xr-x 1 keurf keurf   90  3 juil. 14:08 run.sh*

keurf@DESKTOP-AU3I6FN ~/TOUGHFLAC/Examples/2dldV6/1_TH_INI
$
```

Figure 5.4: Contents of directory *1_TH_INI* before calculation of steady state.

To calculate the steady state, the subdirectory should contain the files:

- *2dldV6i.dat*: TOUGH3 input file without blocks `FLAC` and `GENER`, the blocks `ROCKS`, `MULTI`, `SELEC`, `SOLVR` and `PARAM` have already been filled in. The run is conducted during 300 time steps,
- *MESH* and *INCON*: previously created in FLAC3D in Section 5.1.1,
- *CO2TAB*: required by equation-of-state ECO2N.

To start the simulation (without MPI), simply type in Cygwin:

```
tough3-eco2n 2dldV6i.dat
```

This should produce a *SAVE* file that will be used as initial conditions for the subsequent runs.

Tip: The user can execute the Bash script *run.sh* that will automatically import *MESH* and *INCON* before running the simulation:

```
./run.sh
```

The relative path to files *MESH* and *INCON* might need to be modified if the FLAC3D project has not been created in *2dldV6*.

Then, in FLAC3D, open and execute the script *initialize_model.f3grid* (the model is already configured for this example problem). A save file * tf_in.f3sav* is created once the mechanical steady state is calculated. This save file contains the grid, model parameters and steady state conditions that will be imported when running the coupled simulation.

5.1.3 Run the coupled simulation

Note: Beforehand, the user can optionally run a TOUGH3 (only) simulation to check whether the TOUGH3 input files are correctly edited (in subdirectory *2_TH*).

Once steady states have been conducted for both codes, the coupled simulation is ready to be performed. The coupled TOUGH-FLAC simulation for this example can be run from subdirectory *3_THM*. Before running the simulation, this subdirectory should contain the files:

- *2dldV6.dat*: TOUGH3 input file with blocks `FLAC` and `GENER` (and additional blocks for outputs). The run is conducted for 10 years,

  Tip: Corresponding element label in `MESH` file for a specific point can be queried using the function *`toughflac.zone.near()`*. This can be useful to define the injection point or elements to save in the block `FOFT`. In this input file, the label for the injection point can be found following:

  ```
  tf.zone.near((0.0, 0.0, -1480.0))
  ```

- *flac3d.py*: main TOUGH-FLAC (Python) script for the coupling,
- *tf_in.f3sav*: FLAC3D save file from steady state run,
- *MESH*: previously created in FLAC3D in Section 5.1.1,
- *INCON*: *SAVE* file from steady state run renamed as *INCON*,

  Tip: Last lines of the *SAVE* file must be deleted starting from line *+++*. The Bash script *run.sh* automates this task and assures the file to be correctly edited.

  ```
  sed "/+++/Q" < ../1_TH_INI/SAVE > INCON
  echo "" >> INCON
  ```

- *CO2TAB*: required by equation-of-state ECO2N.

Define permeability models

Permeability models can be defined directly in Python for rock groups with identifier 0 in the block `FLAC`. In this example, porosity ϕ and permeability k are related to mean effective stress σ'_M following

$$\phi = (\phi_0 - \phi_r) \exp\left(5 \cdot 10^{-8} \cdot \sigma'_M\right) + \phi_r$$

$$k = k_0 \exp\left(22.2 \left(\frac{\phi}{\phi_0} - 1\right)\right)$$

where ϕ_0, ϕ_r and k_0 are respectively the stress-free and residual porosities, and stress-free permeability. This permeability model is already implemented and can be imported from `toughflac.coupling.permeability`:

```
from toughflac.coupling.permeability import rutqvist2002
```

To apply this permeability model to the desired rock groups (i.e. with identifier 0 in the block `FLAC`), a dictionary `permeability_func` can be passed to the main coupling function `run()` which keys are the group labels and values are the permeability functions. Permeability functions can be individually defined for each rock group using anonymous `lambda` functions, the only input argument of these functions must be a boolean array (internally generated using `itasca.zonearray.in_group()`) while the group hydromechanical parameters are directly passed to the permeability model. In this example, the dictionary `permeability_func` is defined as follow:

```
permeability_func = {
    "UPPER": lambda g: rutqvist2002(g, k0=1.0e-15, phi0=0.1, phir=0.09),
    "CAPRO": lambda g: rutqvist2002(g, k0=1.0e-17, phi0=0.01, phir=0.009),
    "AQFER": lambda g: rutqvist2002(g, k0=1.0e-13, phi0=0.1, phir=0.09),
    "BASEM": lambda g: rutqvist2002(g, k0=1.0e-17, phi0=0.01, phir=0.009),
}
```

Different permeability models can be associated to different rock groups. The user can also define custom permeability models that are not available in `toughflac.coupling.permeability` as long as it returns the permeability and porosity arrays for a given group (i.e. the size of the output arrays matches the number of elements that belong to the input group). Custom permeability models should basically be defined as follow:

```
from toughflac.coupling.permeability import permeability

@permeability
def my_permeability_model(group, k0, phi0, *args):
    # group is a boolean array returned by :func:`itasca.zonearray.in_group`
    # k0 is stress-free permeability
    # phi0 is stress-free porosity
    # Do something to calculate permeability k and porosity phi
    return k, phi
```

Note: *`toughflac.coupling.permeability.permeability()`* is a function decorator that performs simple checks to verify that the defined function is a valid permeability model that can be used with TOUGH3-FLAC.

Define history variables

History variables to save during the simulation can be defined using a dictionary with specific keywords to be passed to the function *`toughflac.coupling.run()`* in the main script *flac3d.py*. This dictionary (named `history` here) is hierarchized as follow:

```
history = {
    filename: {
        variable: coordinates,
    },
}
```

where `filename` is the output history file name (format *csv*), `variable` is the name of a variable attribute available in `toughflac` (see Table 8.1) and `coordinates` is a list with the coordinates or ID of the points or zones where to query the variable parameter `variable`. The dictionary `history` can contain more than one file, and each file can contain more than one variable. Array `coordinates` must always be 2D even though only one coordinate is needed.

In this example problem:

```
hist_variables = [
    "temp",
    "pp",
    "stress_prin_x",
    "stress_prin_y",
    "stress_prin_z",
    "stress_xx",
    "stress_yy",
    "stress_zz",
]
history = {
```

(continues on next page)

(continued from previous page)

```
    "hist1": {
        "disp_z": [
            [0.0, 0.0, 0.0],
            [0.0, 0.0, -1200.0],
            [0.0, 0.0, -1300.0],
            [0.0, 0.0, -1500.0],
        ],
    },
    "hist2": {k: [[0.0, 0.0, -1450.0]] for k in hist_variables},
    "hist3": {k: [[0.0, 0.0, -1290.0]] for k in hist_variables},
}
```

will save in files:

1. *hist1.csv* the vertical displacement of gridpoints near `[0.0, 0.0, 0.0]`, `[0.0, 0.0, -1200.0]`, `[0.0, 0.0, -1300.0]` and `[0.0, 0.0, -1500.0]`,
2. *hist2.csv* the temperature, pressure, principle stresses, and diagonal elements of the stress tensor for zone near `[0.0, 0.0, -1450.0]`,
3. *hist3.csv* the temperature, pressure, principle stresses, and diagonal elements of the stress tensor for zone near `[0.0, 0.0, -1290.0]`.

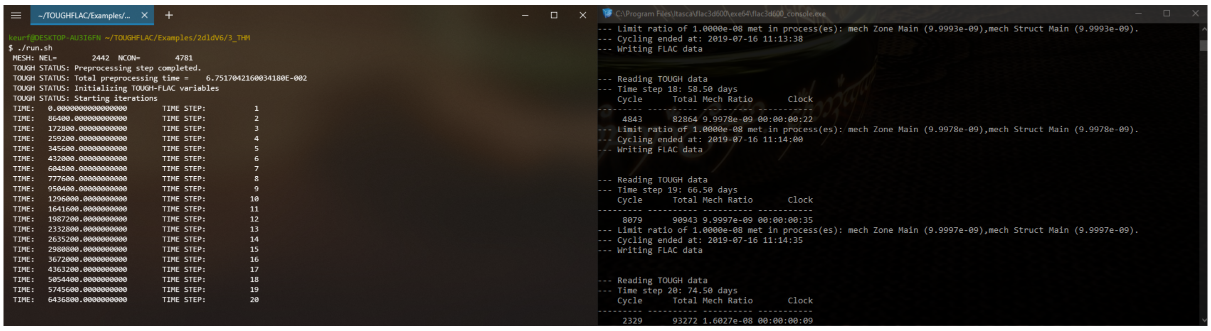
Run the simulation

The user can also *customize* the TOUGH-FLAC simulation by specifying several coupling parameters among the FLAC3D save file to restore, the output directory the damping, the mechanical ratio, the number of threads, additional Python and/or FISH functions to execute after each mechanical analysis…

Once every files are set up, the coupled simulation can be conducted by executing the command `tough3-eco2n 2dldV6.dat`.

Tip: The user can execute the Bash script *run.sh* that will automatically import the required files before running the simulation.

Figure 5.5 shows a screenshot of the two console windows when running a coupled TOUGH-FLAC simulation.


~/TOUGHFLAC/Examples/...
C:\Program Files\Itasca\flac3d600\exe64\flac3d600_console.exe

```
keurf@DESKTOP-AU3I6FN ~/TOUGHFLAC/Examples/2dldV6/3_THM
$ ./run.sh
 MESH: NEL=        2442  NCON=        4781
 TOUGH STATUS: Preprocessing step completed.
 TOUGH STATUS: Total preprocessing time =    6.7517042160034180E-002
 TOUGH STATUS: Initializing TOUGH-FLAC variables
 TOUGH STATUS: Starting iterations
 TIME:   0.0000000000000000        TIME STEP:           1
 TIME:   86400.000000000000        TIME STEP:           2
 TIME:   172800.00000000000        TIME STEP:           3
 TIME:   259200.00000000000        TIME STEP:           4
 TIME:   345600.00000000000        TIME STEP:           5
 TIME:   432000.00000000000        TIME STEP:           6
 TIME:   604800.00000000000        TIME STEP:           7
 TIME:   777600.00000000000        TIME STEP:           8
 TIME:   950400.00000000000        TIME STEP:           9
 TIME:   1296000.0000000000        TIME STEP:          10
 TIME:   1641600.0000000000        TIME STEP:          11
 TIME:   1987200.0000000000        TIME STEP:          12
 TIME:   2332800.0000000000        TIME STEP:          13
 TIME:   2635200.0000000000        TIME STEP:          14
 TIME:   2980800.0000000000        TIME STEP:          15
 TIME:   3672000.0000000000        TIME STEP:          16
 TIME:   4363200.0000000000        TIME STEP:          17
 TIME:   5054400.0000000000        TIME STEP:          18
 TIME:   5745600.0000000000        TIME STEP:          19
 TIME:   6436800.0000000000        TIME STEP:          20
```

```
--- Limit ratio of 1.0000e-08 met in process(es): mech Zone Main (9.9993e-09),mech Struct Main (9.9993e-09).
--- Cycling ended at: 2019-07-16 11:13:38
--- Writing FLAC data


--- Reading TOUGH data
--- Time step 18: 58.50 days
    Cycle      Total Mech Ratio       Clock
--------- ---------- ---------- -----------
     4843      82864 9.9978e-09 00:00:00:22
--- Limit ratio of 1.0000e-08 met in process(es): mech Zone Main (9.9978e-09),mech Struct Main (9.9978e-09).
--- Cycling ended at: 2019-07-16 11:14:00
--- Writing FLAC data


--- Reading TOUGH data
--- Time step 19: 66.50 days
    Cycle      Total Mech Ratio       Clock
--------- ---------- ---------- -----------
     8079      90943 9.9997e-09 00:00:00:35
--- Limit ratio of 1.0000e-08 met in process(es): mech Zone Main (9.9997e-09),mech Struct Main (9.9997e-09).
--- Cycling ended at: 2019-07-16 11:14:35
--- Writing FLAC data


--- Reading TOUGH data
--- Time step 20: 74.50 days
    Cycle      Total Mech Ratio       Clock
--------- ---------- ---------- -----------
     2329      93272 1.6027e-08 00:00:00:09
```

Figure 5.5: Screenshot of coupled TOUGH-FLAC running at time step 20.

5.1.4 Outputs and results

This example problem should produce the FLAC3D save files *f1d.f3sav*, *f7d.f3sav*, *f30d.f3sav*, *f1y.f3sav*, *f3y.f3sav*, *f6s.f3sav* and *f10y.f3sav*. These are save files at 1 day, 1 week, 1 month, etc… as predefined in the block `TIMES` of the TOUGH3 input file. These files can be restored in FLAC3D for plotting of results or used to restart a TOUGH-FLAC simulation at any of these times.

Contour plots can be made from conventional TOUGH3 output file as well as from FLAC3D save files. Time-dependent TOUGH3 data can be plotted from *FOFT*. Figures 5.6 and 5.7 show the pressure and vertical displacement contour plots at the last time step.

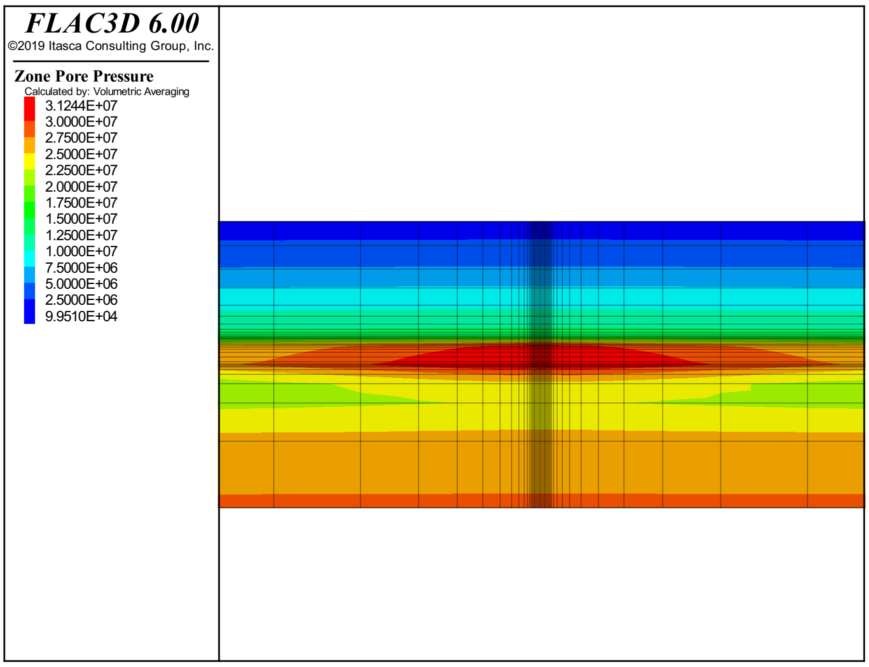


Figure 5.6: Contour plot of pressure at 10 years.

History variables can be plotted using the function *`toughflac.plot.history()`* that uses the `matplotlib` backend. This function provides many different options to customize history plots. For instance, the following code will produce Figure 5.8 using the style `ggplot` (sample script *Postprocessing/plot_history.py*).

```python
import matplotlib.pyplot as plt
plt.style.use("ggplot")

tf.plot.history(
    filename="3_THM/f3out/hist1.csv",
    time_unit="year",
    ylabel="Vertical displacement (m)",
    plt_kws={
        "linewidth": [2, 2, 2, 2],
        "label": ["0 m", "-1200 m", "-1300 m", "-1500 m"],
    },
    legend_kws={
        "title": "Depth",
        "fontsize": 12,
        "loc": "lower center",
        "bbox_to_anchor": (0.5, 0.98),
        "ncol": 4,
        "frameon": False,
    },
```

(continues on next page)

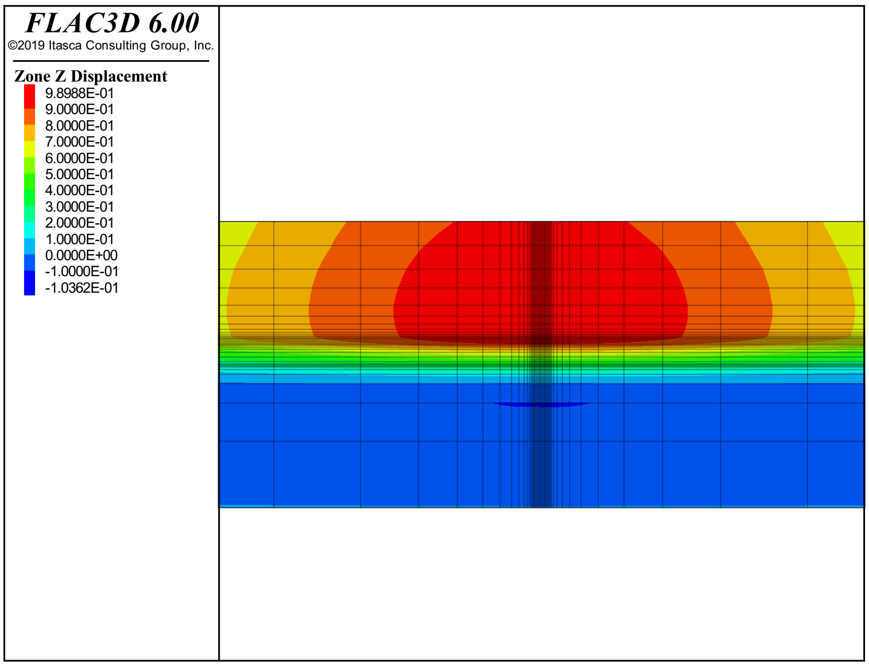


Figure 5.7: Contour plot of vertical displacement at 10 years.

(continued from previous page)

```
)
```

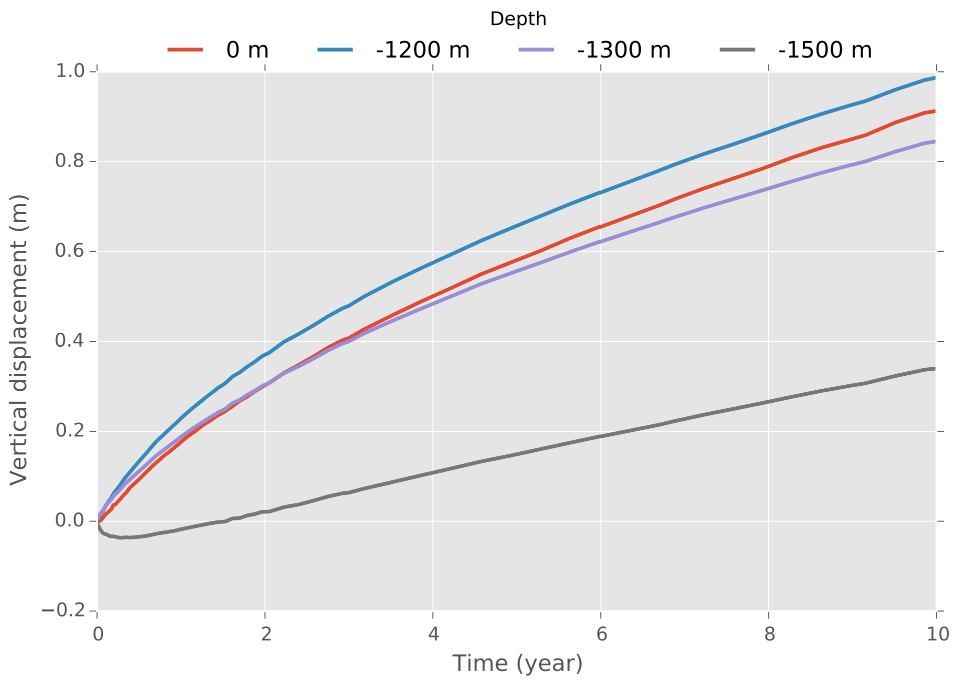


Figure 5.8: Evolution of the vertical displacement plotted from *hist1.csv*.

Tip: The function `toughflac.plot.history()` can also be used to plot the content of TOUGH *FOFT* files. For instance:

```
tf.plot.history("3_THM/FOFT_A1793.csv", columns=1)
```

will plot the evolution of the pressure at the injection point.

5.2 Upwelling of CO_2-rich fluid along a vertical fault

This problem involves upwelling of CO_2-rich fluid along a vertical fault. All the necessary input files can be found in a directory named *2DSEP5V6*. This directory contains the coupling Python script *flac3d.py* and five subdirectories:

- *1_TH_INI* to run a steady state calculation with TOUGH3 only,
- *2_TH* to test input files required for TOUGH3,
- *3_THM* to conduct a fully coupled TOUGH3 and FLAC3D v6 simulation,
- *Preprocessing* that contains scripts to generate TOUGH3 and FLAC3D grids (*generate_mesh.py* and *generate_mesh_gmsh.py*) and configure the mechanical model for FLAC3D (*initialize_model.f3grid*),
- *Postprocessing* that contains sample scripts to plot history variables and export results to a VTK file.

The three former subdirectories contain the TOUGH3 input file, a file *CO2TAB* needed for equation-of-state ECO2N, and two Bash scripts *clean.sh* and *run.sh*. These two scripts are not necessary and are only here for convenience (facilitate the workflow).

The user might want to start and save a new FLAC3D project in this directory (i.e. *2DSEP5V6*).

Note: More advanced pre- and post-processing (e.g. import external meshes, export results to a VTK file) require the Python library `toughio` to be installed.

5.2.1 Grid generation for TOUGH3 and FLAC3D

Although the grid files can be generated using the script *generate_mesh.py* in the same way as in the first example, this example will rather show how to import an external mesh file created by another software (here Gmsh, see Figure 5.9).

In the following, we will selectively go through the content of the script *generate_mesh_gmsh.py* and only detail mesh import and processing.

The mesh file can be imported using the function `toughio.read_mesh()`:

```
mesh = toughio.read_mesh("gmsh/mesh.msh")
```

This creates a `toughio.Mesh` object that contains the grid points coordinates, the cell connectivity (i.e. zones) and the data associated to these cells.

The input mesh is 2D and has been defined in the XY plane. Thus, grid points Y and Z coordinates must be swapped:

```
mesh.points[:, [1, 2]] = mesh.points[:, [2, 1]]
```

and a third dimension must be created along the Y axis using the method `toughio.Mesh.extrude_to_3d()`:

```
mesh.extrude_to_3d(height=1.0, axis=1)
```

Finally, the processed mesh can be imported using the function *`toughflac.zone.import_mesh()`*:

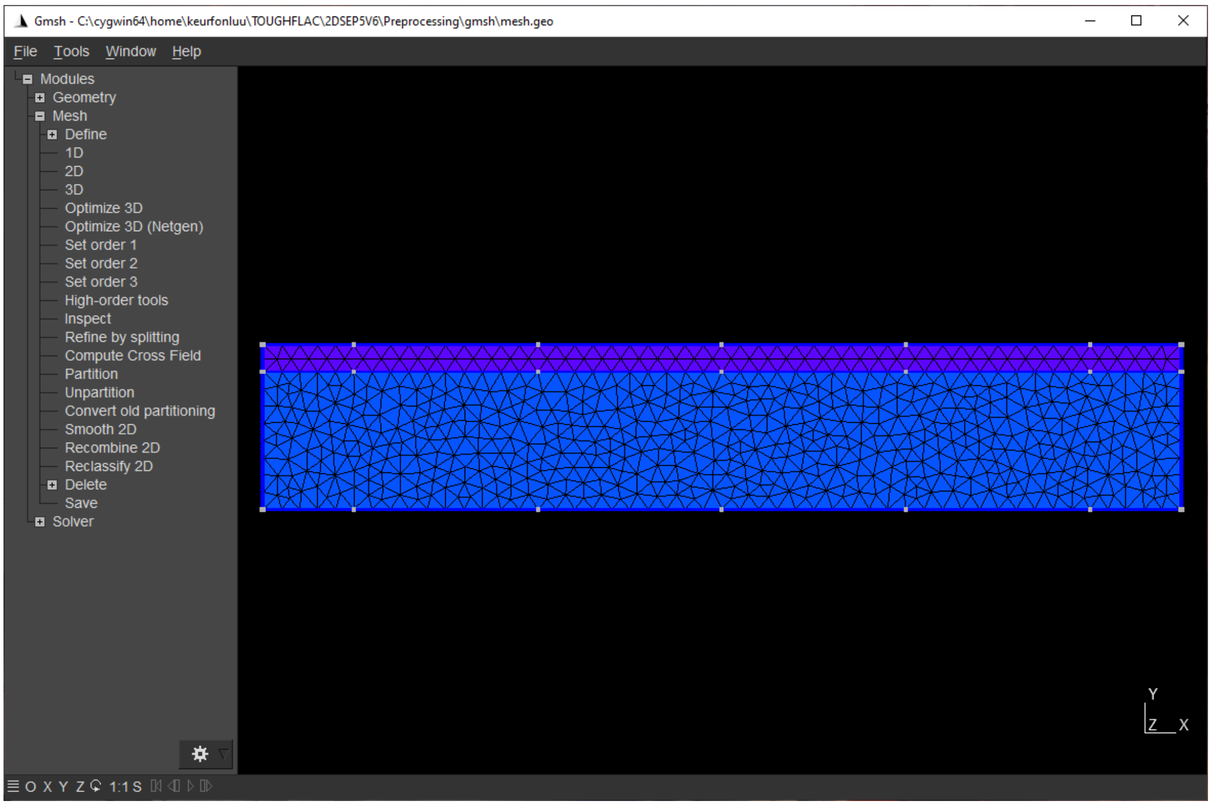


Figure 5.9: Mesh created in Gmsh. Physical groups (including boundary elements) have been defined in Gmsh.

```
tf.zone.import_mesh(mesh)
```

Tip: A mesh file that does not require additional processing (e.g. already 3D) can be directly imported using the function *toughflac.zone.import_()*.

The FLAC3D grid should look like Figure 5.10.

For this example, initial conditions are defined for the top and the bottom of the model and pressure and temperature are fixed to constant values at the top of the grid:

```
tf.zone.set_dirichlet_bc("TOPBC")
tf.zone.set_dirichlet_bc("TOPFC")

tf.zone.initialize_pvariables(
    x1=lambda z: 1.5e5,
    x2=lambda z: 0.3e-2,
    x3=lambda z: 0.0,
    x4=lambda z: 8.0,
    group = ["TOPBC", "TOPFC"],
)
tf.zone.initialize_pvariables(
    x1=lambda z: 1.77e6,
    x2=lambda z: 0.3e-2,
    x3=lambda z: 0.0,
    x4=lambda z: 98.0,
    group = ["BOTBC", "BOTFC"],
)
```

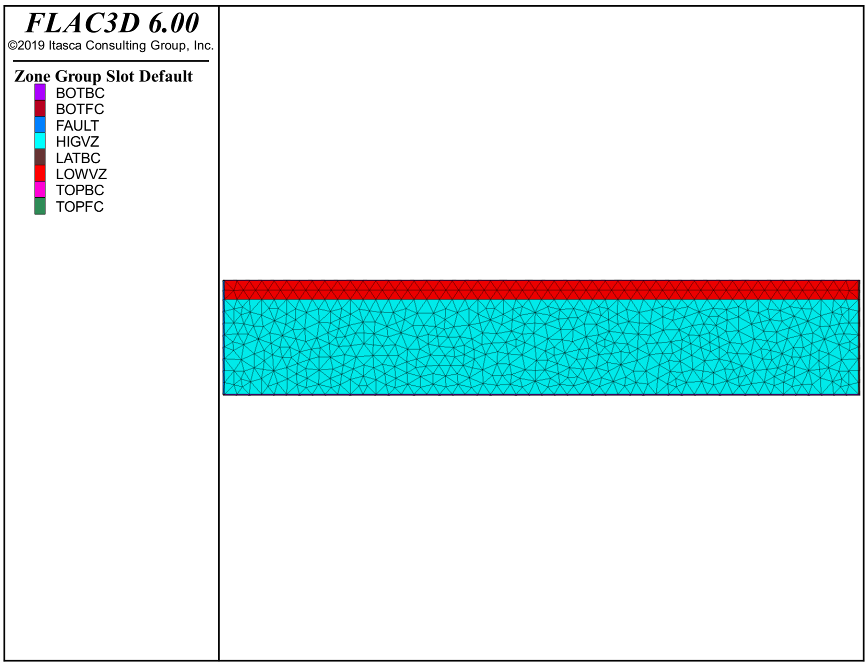


Figure 5.10: Gmsh grid imported into FLAC3D. Physical groups defined in Gmsh have been associated as zone groups.

Note:

1. Initializing primary variables as constant values for specific groups is no different from setting `INDOM` block in TOUGH3 input file,
2. Once the grid files are generated, the workflow is the same as in the first example.

5.2.2 Outputs and results

Contour plots of pressure and vertical displacement at the last time step are shown in Figures 5.11 and 5.12. The history of the displacement for three different grid points is shown in Figure 5.13 (sample script *Postprocessing/plot_history.py*).

5.2.3 Export to VTK file

The results for any FLAC3D save file can also be exported to a VTK file to be visualized with ParaView or `pyvista`, for instance, for advanced post-processing (sample script *Postprocessing/export_vtk.py*). The complete list of attribute variables to export can be found in Table 8.1.

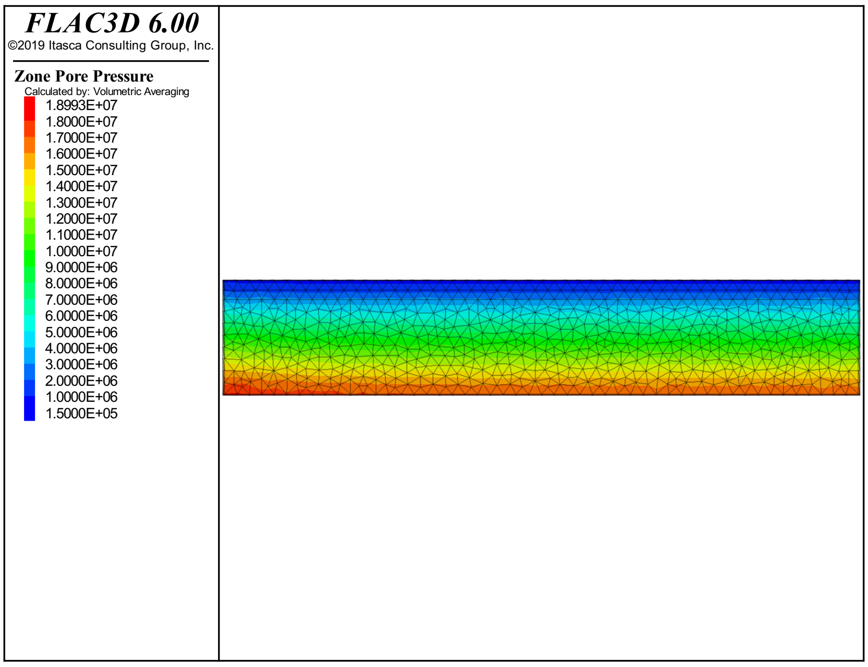


Figure 5.11: Contour plot of pressure at 4 years.

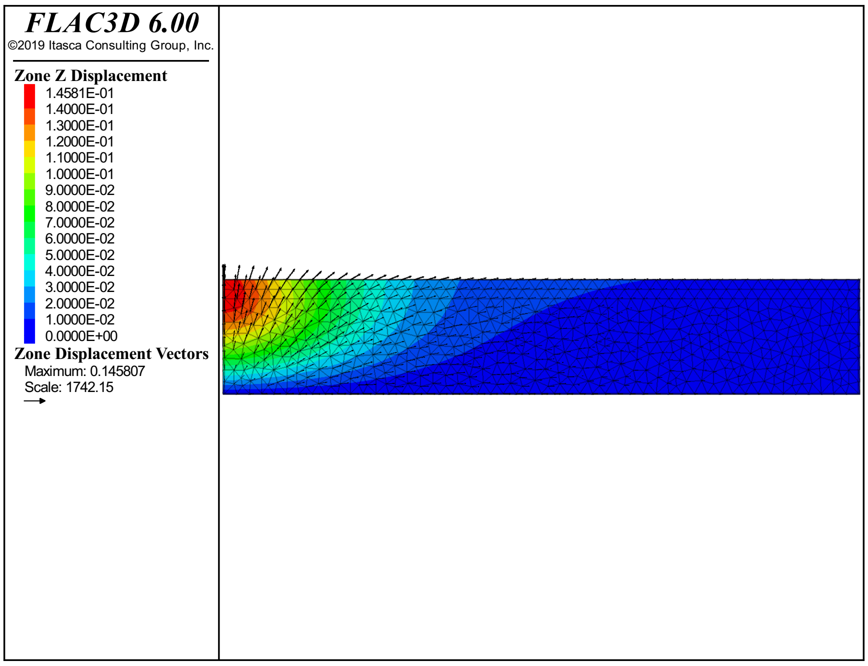


Figure 5.12: Contour plot of vertical displacement at 4 years.

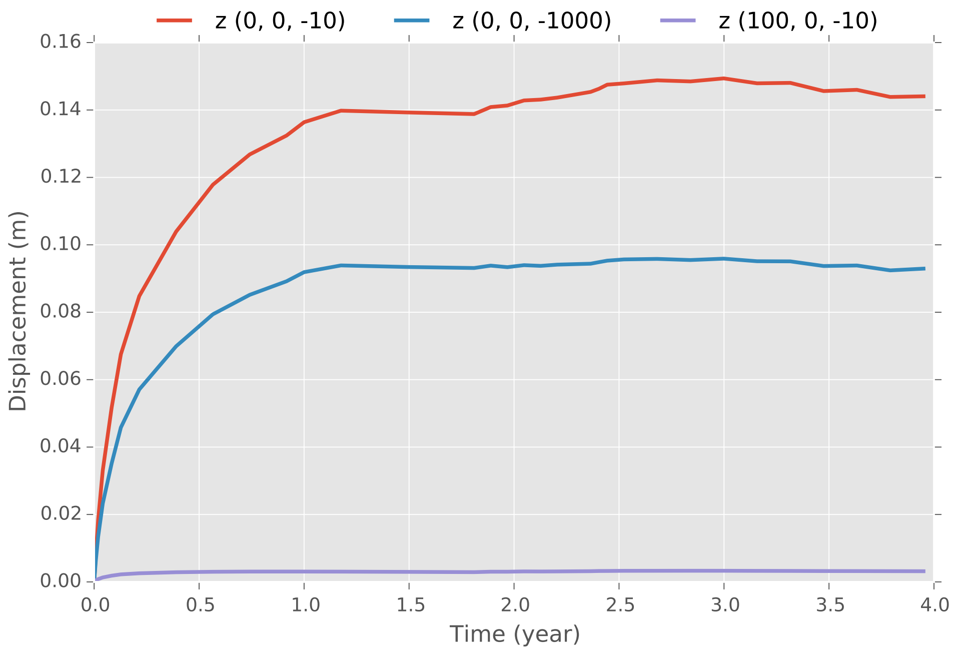


Figure 5.13: Evolution of the displacement plotted from *hist1.csv*.

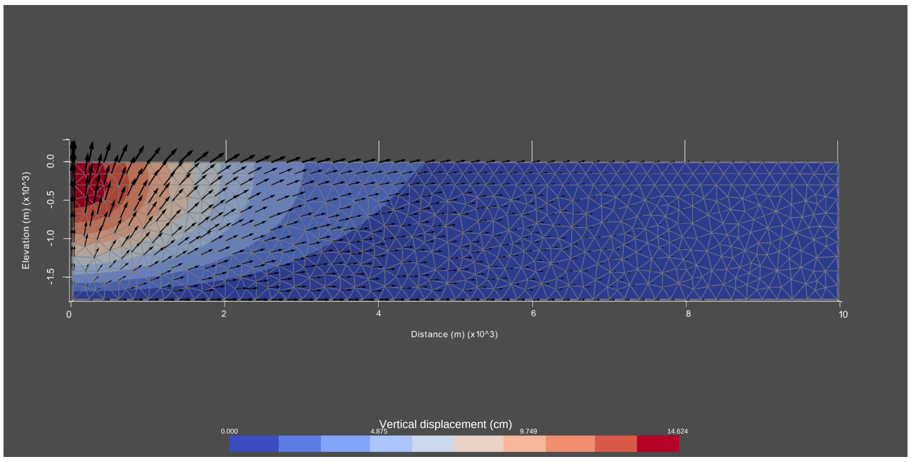


Figure 5.14: Contour plot of vertical displacement at 4 years (figure generated by `pyvista` using script *Postprocessing/plot_pyvista.py*).

5.3 Injection of CO_2 in an anisotropic reservoir rock

This example describes a problem of CO_2 injection in a reservoir rock with high horizontal to vertical permeability ratio. All the necessary input files can be found in a directory named *CR2011*. This directory contains four subdirectories:

- *1_TH_INI* to run a steady state calculation,
- *2_TH* to conduct a TOUGH3 simulation,
- *Preprocessing* that contains scripts to generate TOUGH3 grid (*generate_mesh.py* and *generate_mesh_gmsh.py*),
- *Postprocessing* that contains a sample script to import and visualize TOUGH3 results in FLAC3D.

The two former subdirectories contain the TOUGH3 input file, a file *CO2TAB* needed for equation-of-state ECO2N, and two Bash scripts *clean.sh* and *run.sh*. These two scripts are not necessary and are only here for convenience (facilitate the workflow).

The user might want to start and save a new FLAC3D project in this directory (i.e. *CR2011*).

Note: In this example, FLAC3D is only used as a pre- and post-processor to run a simple TOUGH3 simulation.

5.3.1 Grid generation for TOUGH3

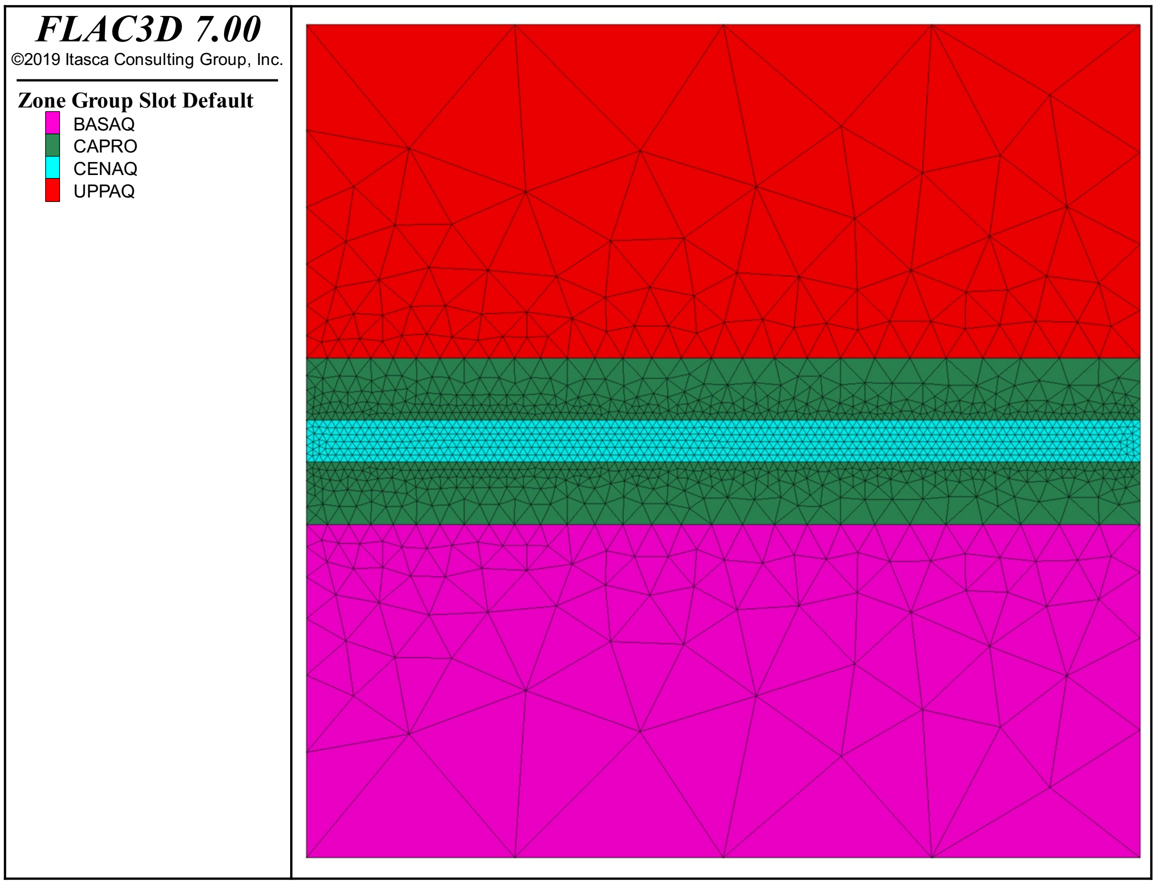


Figure 5.15: Gmsh grid imported into FLAC3D.

In this example, we will generate the mesh using Gmsh again (script *generate_mesh_gmsh.py*). Grid import and processing, initial and boundary conditions settings are the same as in the previous example (see Section 5.2). Only new arguments in function *toughflac.zone.export_tough()* will be described.

```
tf.zone.export_tough(
    "MESH",
    permeability=lambda zone: [1.0e-13, 1.0e-13, 1.0e-15] if zone.group() == "CENAQ
↪" else [-1.0, -1.0, -1.0],
    group_end="BOUND",
    incon=True,
)
```

In TOUGH3, isotropic and/or anisotropic permeability modifiers can be defined for each element of the mesh in either block `ELEME` (columns `81-110`) or block `INCON` (columns `31-60`). Individual permeability (and/or porosity) values can be exported by providing argument `permeability` (and/or `porosity`) as a function of `itasca.Zone`. In this example, vertical permeability of reservoir rock `CENAQ` is 100 times lower than its horizontal permeability (`[1.0e-13, 1.0e-13, 1.0e-15]`). The permeability in the other rocks is isotropic and set to `[-1.0, -1.0, -1.0]` since negative permeability modifiers are treated as multipliers in TOUGH3.

Note:

- In the current implementation, permeability modifiers are exported in columns `31-60` of block `INCON` so that custom porosity and permeability values for individual elements are at the same location.
- For this simple example, providing argument `permeability` is not necessary as permeability is homogeneous for each rock which could be done simply by providing anisotropic permeability values in TOUGH3 input data. Nevertheless, providing arguments `permeability` (and/or `porosity`) to function *toughflac.zone.export_tough()* can be used, for instance, to interpolate permeability logs to the model (and/or porosity logs).

Providing argument `group_end="BOUND"` moves all the elements in group `BOUND` to the bottom of block `ELEME` which can be useful if boundary elements are defined in a single rock type (and must be ignored by FLAC3D when running a coupled simulation).

5.3.2 TOUGH3 input

Although there is not much difference with the previous examples, the user still has to enable some options in TOUGH3 to correctly process anisotropic permeability values.

Since permeability modifiers are written in block `INCON` rather than `ELEME`, `MOP(13)` in block `PARAM` must be set to `1` in order to write these custom values in *SAVE* file after running a steady state calculation.

In addition, `MOP2(19)` in block `MOMOP` must be set to a non-zero value which generalizes how TOUGH3 handles anisotropic permeability values to any type of mesh (i.e. not only hexahedral meshes).

Note: Currently, this option is not available in base TOUGH3 and has only been implemented in TOUGH3-FLAC.

5.3.3 Outputs and results

TOUGH3 output files (*SAVE* and *OUTPUT_ELEME.csv*) can be imported into FLAC3D using functions `toughflac.zone.import_tough_save()` and `toughflac.zone.import_tough_output()`, respectively (see sample script *Postprocessing/import_tough.py*).

Contour plots of gas saturation at the last time step are shown in Figures 5.16 and 5.16 which shows the difference of CO_2 plume when `MOP2(19) = 0` and `MOP2(19) = 1`. Figure 5.18 is displayed to allow comparison as a reference figure (i.e. expected result).

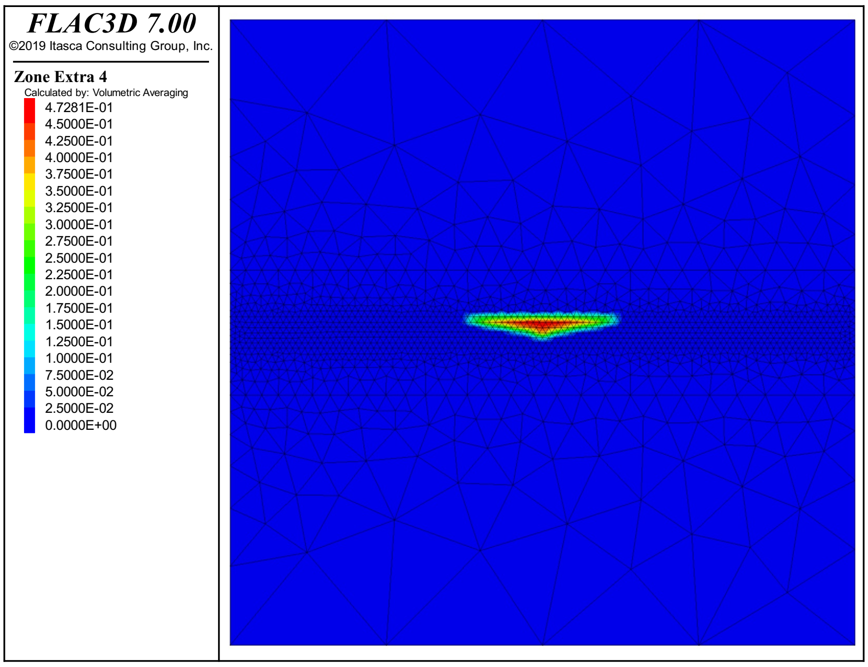


Figure 5.16: Contour plot of gas saturation with triangular mesh (`MOP2(19) = 0`).

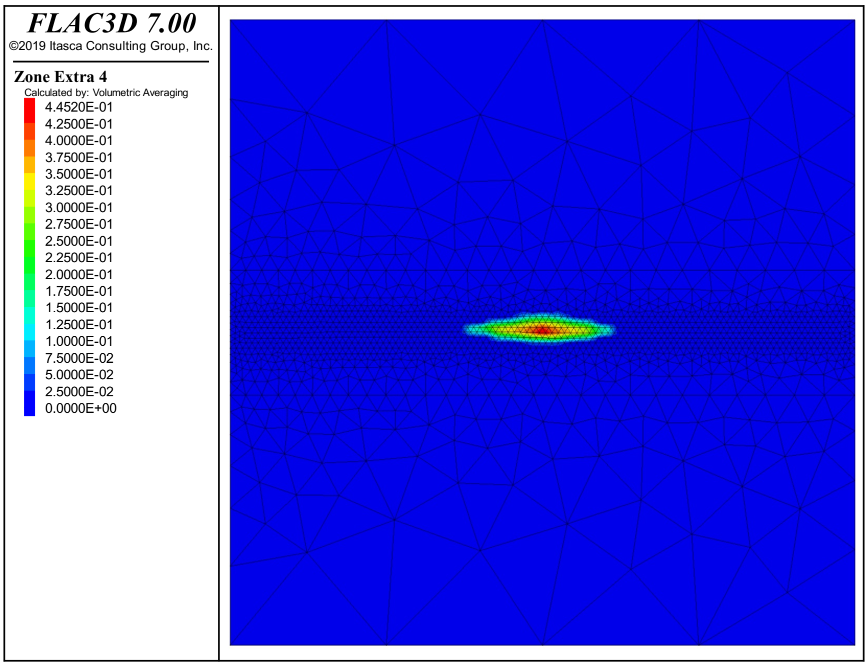


Figure 5.17: Contour plot of gas saturation with triangular mesh (`MOP2(19) = 1`).

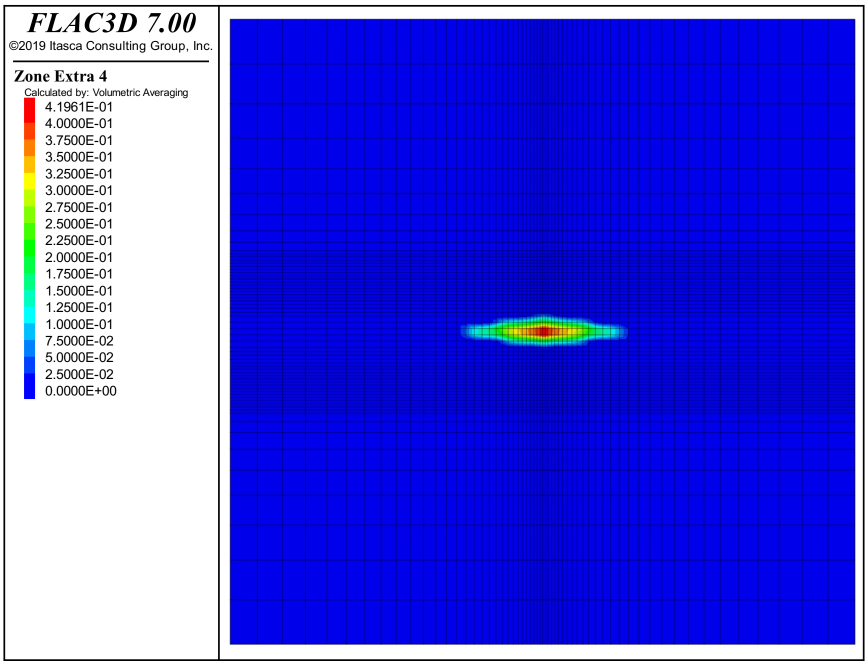


Figure 5.18: Contour plot of gas saturation with hexahedral mesh.

Appendix A

Welcome

This documentation details submodules and functions included in `toughflac`, describing what they are, what they want to do, and for some, how they achieve what they want to do. The module `toughflac` is organized as follow:

```
toughflac
  ├─coupling
  ├─model
  ├─plot
  ├─utils
  ├─zone
  └─zonearray
```

Appendix B

Coupling

Submodule `coupling` provides functions required to run a coupled TOUGH-FLAC simulation and is organized as follow:

```
coupling
 ├─extra
 ├─run
 └─permeability
     ├─permeability
     ├─constant
     ├─chin2000
     ├─rutqvist2002
     ├─minkoff2004
     └─hsiung2005
```

Note: The original FISH routines in the previous versions of TOUGH-FLAC have been translated into Python in a private file named *io.py*. The user does not need to have access to the content of this file.

B.1 coupling.run

`toughflac.coupling.`**`run`**`(`*model_save=None*, *deterministic=False*, *damping='combined'*, *mechanical_ratio=1e-05*, *n_threads=None*, *thermal=True*, *biot_func=None*, *permeability_func=None*, *python_func_tough=None*, *python_func_flac=None*, *fish_func_tough=None*, *fish_func_flac=None*, *history_func=None*, *history=None*, *savedir=None*, *save_python=False*`)`

Run TOUGH-FLAC coupled simulation.

Parameters

- **`model_save`** (*`str or None, optional, default None`*) – FLAC3D save file to restore before simulation. It should contain zone grid and model configuration.
- **`deterministic`** (*`bool, optional, default False`*) – If *True*, set deterministic mode.
- **`damping`** (*`str ('local' or 'combined'), optional, default 'combined'`*) – Set damping for static mechanical process.
- **`mechanical_ratio`** (*`scalar, optional, default 1.0e-5`*) – FLAC3D mechanical convergence criterion.

- **n_threads** (*int or None, optional, default None*) – Number of threads. If *None*, *n_threads* is imported from `itasca`.
- **thermal** (*bool, optional, default True*) – If *True*, activate FLAC3D thermal module.
- **biot_func** (*callable or None, optional, default None*) – Function to update Biot's coefficient for each zone.
- **permeability_func** (*dict or None, optional, default None*) – Custom permeability models for every keys in *permeability_func*.
- **python_func_tough** (*list of callable or None, optional, default None*) – List of additional Python functions to run (after reading TOUGH file, before mechanical analysis).
- **python_func_flac** (*list of callable or None, optional, default None*) – List of additional Python functions to run (after mechanical analysis).
- **fish_func_tough** (*list of str or None, optional, default None*) – List of additional FISH functions to run (after reading TOUGH file, before mechanical analysis).
- **fish_func_flac** (*list of str or None, optional, default None*) – List of additional FISH functions to run (after mechanical analysis).
- **history_func** (*dict or None, optional, default None*) – Functions for custom history variables in *history*.
- **history** (*dict or None, optional, default None*) – Save history variables in different files for every keys in *history*. Each subdictionaries should contain keywords describing variables to export and values defining which points to save. See examples for more details.
- **savedir** (*str or None, optional, default None*) – Output directory in which FLAC3D save and history files are exported.
- **save_python** (*bool, optional, default False*) – If *True*, Python variables are also exported when saving FLAC3D state.

Examples

History variables that have to be saved during a coupled simulation at each TOUGH time step can be defined using a dictionary, for instance:

```
>>> history = {
>>>     "hist1": {
>>>         "disp_z" = [
>>>             [0.0, 0.0, 0.0],
>>>             [0.0, 0.0, -10.0],
>>>         ],
>>>         "disp_x" = [
>>>             [10.0, 0.0, 0.0],
>>>         ],
>>>     },
>>>     "hist2": {
>>>         "pp" = [
>>>             [0.0, 0.0, 0.0],
>>>         ],
>>>         "temp" = [
>>>             [0.0, 0.0, 0.0],
>>>         ],
```

(continues on next page)

(continued from previous page)

```
>>>     },
>>> }
```

This will save for each TOUGH time step, the vertical displacement at points `[0.0, 0.0, 0.0]` and `[0.0, 0.0, -10.0]` and the horizontal displacement at point `[10.0, 0.0, 0.0]` in file *hist1.csv*, and the pore pressure and temperature at point `[0.0, 0.0, 0.0]` in file *hist2.csv*.

This dictionary can now be passed to the argument *history* of `run()`.

Warning: The user should never have to call this function. It should only be called and executed within the script *flac3d.py* provided for the coupling.

B.2 coupling.permeability

Several permeability models have been implemented in TOUGH-FLAC Python module. The user can also implement new permeability model taking these functions as examples: the first argument of such function **must** be a mask array `group` and return permeability and porosity arrays only for current group (if permeability is not affected by porosity change, the porosity array can be whatever array of same size e.g. zeros).

B.2.1 coupling.permeability.permeability

`toughflac.coupling.permeability.`**`permeability`**`(`*func*`)`

Decorate permeability model functions.

B.2.2 coupling.permeability.constant

`toughflac.coupling.permeability.`**`constant`**`(`*group*, *k0*, *phi0*`)`

No mechanical-induced permeability change.

Parameters

- **`group`** (*`array_like`*) – Mask array for queried group.
- **`k0`** (*`scalar`*) – Stress-free permeability.
- **`phi0`** (*`scalar`*) – Stress-free porosity.

Returns

- *array_like* – New permeability array for queried group.
- *array_like* – New porosity array for queried group.

Note: This function is only provided for completeness. It is recommended to directly use permeability model identifier 1 in block ‘FLAC’.

B.2.3 coupling.permeability.chin2000

`toughflac.coupling.permeability.`**`chin2000`**`(`*group*, *k0*, *phi0*, *ke=5.6*`)`

After Chin et al. (2000).

Parameters

- **`group`** (*`array_like`*) – Mask array for queried group.
- **`k0`** (*`scalar`*) – Stress-free permeability.
- **`phi0`** (*`scalar`*) – Stress-free porosity.
- **`ke`** (*`scalar, optional, default 5.6`*) – Permeability exponent.

Returns

- *array_like* – New permeability array for queried group.
- *array_like* – New porosity array for queried group.

B.2.4 coupling.permeability.rutqvist2002

`toughflac.coupling.permeability.`**`rutqvist2002`**`(`*group*, *k0*, *phi0*, *phir=0.0*, *ke=22.2*, *phie=5e-08*`)`

After Rutqvist and Tsang (2002).

Parameters

- **`group`** (*`array_like`*) – Mask array for queried group.
- **`k0`** (*`scalar`*) – Stress-free permeability.
- **`phi0`** (*`scalar`*) – Stress-free porosity.
- **`phir`** (*`scalar, optional, default 0.0`*) – Residual porosity.
- **`ke`** (*`scalar, optional, default 22.2`*) – Permeability exponent.
- **`phie`** (*`scalar, optional, default 5.0e-8`*) – Porosity exponent.

Returns

- *array_like* – New permeability array for queried group.
- *array_like* – New porosity array for queried group.

B.2.5 coupling.permeability.minkoff2004

`toughflac.coupling.permeability.`**`minkoff2004`**`(`*group*, *k0*, *phi0*, *ke*`)`

After Minkoff et al. (2004).

Parameters

- **`group`** (*`array_like`*) – Mask array for queried group.
- **`k0`** (*`scalar`*) – Stress-free permeability.
- **`phi0`** (*`scalar`*) – Stress-free porosity.
- **`ke`** (*`scalar`*) – Permeability exponent.

Returns

- *array_like* – New permeability array for queried group.
- *array_like* – New porosity array for queried group.

B.2.6 coupling.permeability.hsiung2005

`toughflac.coupling.permeability.`**`hsiung2005`**`(`*group*, *k0*, *phi0*, *n*, *psi*, *a*, *sig0*, *joint=False*`)`

After Hsiung et al. (2005).

Parameters

- **`group`** (*`array_like`*) – Mask array for queried group.
- **`k0`** (*`scalar`*) – Stress-free permeability.
- **`phi0`** (*`scalar`*) – Stress-free porosity.
- **`n`** (*`array_like`*) – Unit normal vector (3 components).
- **`psi`** (*`scalar`*) – Dilation angle.
- **`a`** (*`scalar`*) – Inverse of stiffness.
- **`sig0`** (*`array_like`*) – Initial normal effective stress. Compressions are positive.
- **`joint`** (*`bool, optional, default False`*) – If *True*, shear and tensile strains are read from joint values.

Returns

- *array_like* – New permeability array for queried group.
- *array_like* – New porosity array for queried group.

Appendix C

Model

Submodule `model` provides functions to reset, save and restore FLAC3D and Python states, it is currently organized as follow:

```
model
 ├─new
 ├─save
 ├─restore
 ├─save_python
 └─restore_python
```

C.1 model.new

`toughflac.model.`**`new`**`(`*reset_python=False*`)`

Reset model state.

Parameters **`reset_python`** (*`bool, optional, default False`*) – If *True*, Python state will be reset.

C.2 model.save

`toughflac.model.`**`save`**`(`*filename*, *python_variables=None*`)`

Save FLAC3D model and Python variables.

Parameters

- **`filename`** (*`str`*) – Output file name.
- **`python_variables`** (*`dict or None, optional, default None`*) – Python variables to dump to *filename.pysav*. If *None*, only save FLAC3D state.

C.3 model.restore

`toughflac.model.`**`restore`**(*filename*, *python=False*, *python_target=None*)

Restore FLAC3D model and associated Python variables.

Parameters

- **`filename`** (*`str`*) – Input file name.
- **`python`** (*`bool, optional, default False`*) – If *True*, also restore associated Python variables.
- **`python_target`** (*`dict or None, optional, default None`*) – Dictionary in which Python variables will be restored. If *locals()* or *globals()*, Python variables are directly loaded in the current Python session (existing variables with the same name will be overwritten without any notice). If not *None*, restore associated Python variables even if `python == False`.

Returns Python variables in *filename.pysav*.

Return type dict

C.4 model.save_python

`toughflac.model.`**`save_python`**(*filename*, *variables*)

Save Python variables.

Parameters

- **`filename`** (*`str`*) – Output file name.
- **`variables`** (*`dict`*) – Python variables to dump to *filename*.

C.5 model.restore_python

`toughflac.model.`**`restore_python`**(*filename*, *target=None*)

Restore Python variables.

Parameters

- **`filename`** (*`str`*) – Input file name.
- **`target`** (*`dict or None, optional, default None`*) – Dictionary in which Python variables will be restored. If *locals()* or *globals()*, Python variables are directly loaded in the current Python session (existing variables with the same name will be overwritten without any notice).

Returns Python variables in *filename*.

Return type dict

Appendix D

Plot

Submodule `plot` aims to facilitate plotting in a pythonic manner, it is currently organized as follow:

```
plot
 ├─history
 ├─mohr_diagram
 ├─stress_polygon
 └─gutenberg_richter
```

D.1 plot.history

`toughflac.plot.`**`history`**`(`*filename*, *columns=None*, *time_unit='second'*, *yscale=None*, *ylabel=None*, *ax=None*, *figsize=None*, *show_legend=True*, *plot_type='default'*, *plt_kws={}*, *legend_kws={}*`)`

Plot history variables.

Parameters

- **`filename`** (*`str`*) – History file name.
- **`columns`** (*`int, array_like or None, optional, default None`*) – Columns to display. If *None*, all the columns > 0 are displayed.
- **`time_unit`** (*`str ('second', 'hour', 'day', 'year'), optional, default 'second'`*) – Time axis unit.
- **`yscale`** (*`scalar or None, optional, default None`*) – Scaling factor for y axis.
- **`ylabel`** (*`str or None, optional, default None`*) – Y axis label.
- **`ax`** (*`matplotlib.axes.Axes or None, optional, default None`*) – Matplotlib axes. If *None*, a new figure and axe is created.
- **`figsize`** (*`array_like or None, optional, default None`*) – New figure size if *ax* is *None*.
- **`show_legend`** (*`bool, optional, default True`*) – If *True*, show legend.
- **`plot_type`** (*`str ('default', 'semilogx', 'semiglogy' or 'loglog'), optional,`*) – default 'default' Plot axis type.
- **`plt_kws`** (*`dict, optional, default {}`*) – Additional keywords passed to `plt.plot()`. Values associated to keys must be array-like consistent with the columns to display.

- **legend_kws** (*dict, optional*) – Additional keywords passed to `ax.legend()`.

Returns Matplotlib axes.

Return type matplotlib.axes.Axes

D.2 plot.mohr_diagram

`toughflac.plot.`**`mohr_diagram`**(*sig1*, *sig2=None*, *sig3=None*, *pp=0.0*, *friction=0.6*, *cohesion=0.0*, *unit=None*, *fill=True*, *ax=None*)

Mohr-Coulomb stress diagram.

Parameters

- **sig1** (*scalar*) – Maximum principal stress.
- **sig2** (*scalar or None, optional, default None*) – Intermediate principal stress.
- **sig3** (*scalar or None, optional, default None*) – Minimum principal stress. If *None*, the minimum horizontal stress is calculated using *sig1*.
- **pp** (*scalar, optional, default 0.*) – Pore pressure.
- **friction** (*scalar, optional, default 0.6*) – Coefficient of friction.
- **cohesion** (*scalar, optional, default 0.*) – Coefficient of cohesion.
- **unit** (*str or None, optional, default None*) – Axes unit. If *None*, unit is not displayed in axis labels.
- **fill** (*bool, optional, default True*) – If *True*, shade area of allowable stress components.
- **ax** (*matplotlib.axes.Axes or None, optional, default None*) – Matplotlib axes.

Returns Matplotlib axes.

Return type matplotlib.axes.Axes

Note: Compressive stresses are positive.

D.3 plot.stress_polygon

`toughflac.plot.`**`stress_polygon`**(*stress_vertical*, *pp=0.0*, *friction=0.6*, *unit=None*, *ax=None*)

Plot stress polygon (a.k.a. Zobackogram).

Parameters

- **stress_vertical** (*scalar*) – Vertical stress (overburden).
- **pp** (*scalar, optional, default 0.*) – Pore pressure.
- **friction** (*scalar, optional, default 0.6*) – Coefficient of friction.
- **unit** (*str or None, optional, default None*) – Axes unit. If *None*, unit is not displayed in axis labels.
- **ax** (*matplotlib.axes.Axes or None, optional, default None*) – Matplotlib axes.

Returns Matplotlib axes.

Return type matplotlib.axes.Axes

Note: Compressive stresses are positive.

D.4 plot.gutenberg_richter

`toughflac.plot.`**`gutenberg_richter`**`(`*magnitudes*, *n=50*, *cutoff=None*, *ax=None*`)`

Plot Gutenberg-Richter plot.

Parameters

- **`magnitudes`** (*`array_like`*) – Magnitudes array.
- **`n`** (*`int, optional, default 50`*) – Number of intervals of magnitudes.
- **`cutoff`** (*`scalar or None, optional, default None`*) – Cutoff magnitude of completeness for regression (to fit b-value). If *None*, it is estimated using the maximum curvature method.
- **`ax`** (*`matplotlib.axes.Axes or None, optional, default None`*) – Matplotlib axes.

Returns Matplotlib axes.

Return type matplotlib.axes.Axes

Appendix E

Utils

Submodule `utils` provides several functions to help developing a model and is currently organized as follow:

```
utils
 ├─fault_normal
 ├─normal_stress
 ├─rotation_matrix
 └─rotate_tensor
```

E.1 utils.fault_normal

`toughflac.utils.`**`fault_normal`**(*strike*, *dip*)

Return fault normal vector given fault strike and dip angles.

Parameters

- **`strike`** (*`scalar`*) – Strike angle (between 0 and 360 degrees).
- **`dip`** (*`scalar`*) – Dip angle (between 0 and 90 degrees).

Returns Fault normal vector.

Return type array_like

E.2 utils.normal_stress

`toughflac.utils.`**`normal_stress`**(*stress*, *normal*, *return_shear=False*)

Return normal stress(es) given stress tensor(s) and normal vector.

Parameters

- **`stress`** (*`array_like`*) – Stress tensor (3 by 3 matrix) or list of stress tensors.
- **`normal`** (*`array_like`*) – Normal vector.
- **`return_shear`** (*`bool, optional, default False`*) – If *True*, also return shear stress.

Returns

- *scalar or array_like* – Normal stress or list of normal stresses.
- *scalar or array_like* – Shear stress or list of shear stresses. Only if `return_shear == True`.

E.3 utils.rotation_matrix

`toughflac.utils.`**`rotation_matrix`**`(`*angle*, *axis=2*`)`

Return rotation matrix for input angle and axis.

Parameters

- **`angle`** (*`scalar`*) – Rotation angle.
- **`axis`** (*`int (0, 1 or 2), optional, default 2`*) –

  Rotation axis:

  - 0: X axis,
  - 1: Y axis,
  - 2: Z axis.

Returns Rotation matrix.

Return type array_like

E.4 utils.rotate_tensor

`toughflac.utils.`**`rotate_tensor`**`(`*tensor*, *angle*, *axis=2*`)`

Rotate a tensor along an axis.

Parameters

- **`tensor`** (*`array_like`*) – Tensor to rotate.
- **`angle`** (*`scalar`*) – Rotation angle.
- **`axis`** (*`int (0, 1 or 2), optional, default 2`*) –

  Rotation axis:

  - 0: X axis,
  - 1: Y axis,
  - 2: Z axis.

Returns Rotated tensor.

Return type array_like

Appendix F

Zone

Submodule `zone` provides functions to facilitate pre-processing for coupled TOUGH-FLAC simulation and is organized as follow:

```
zone
 ├─import_
 ├─import_mesh
 ├─import_flac
 ├─import_tough_output
 ├─import_tough_save
 ├─export
 ├─export_tough
 ├─export_flac
 ├─export_mesh
 ├─export_time_series
 ├─in_group
 ├─near
 ├─group
 ├─set_dirichlet_bc
 ├─initialize_pvariables
 └─create
    └─tartan_brick
```

F.1 zone.import_

`toughflac.zone.`**`import_`**`(`*filename=None*, *file_format=None*, ***kwargs*`)`

Import external mesh file in FLAC3D.

Parameters

- **`filename`** (*`str or None, default None`*) – Input file name. If *None*, a file dialog window will be opened.
- **`file_format`** (*`str or None, optional, default None`*) – Input file format. If *None*, it will be guessed from file's extension. Importing non-FLAC3D grids requires `toughio`.

Note:

1. The trailing underscore in the function name has been added because *import* is already a keyword in Python,
2. When importing non-FLAC3D grid, `toughio` writes a temporary f3grid file that is then imported using FLAC3D **`zone import`**.

F.2 zone.import_mesh

`toughflac.zone.`**`import_mesh`**`(`*mesh*, *meshname=None*, *prune_duplicates=False*`)`

Import meshio.Mesh object as FLAC3D grid.

This function requires `toughio` to be installed.

Parameters

- **`mesh`** (*`toughio.Mesh`*) – Mesh to import.
- **`meshname`** (*`str or None, optional, default None`*) – Mesh name displayed in FLAC3D project.
- **`prune_duplicates`** (*`bool, optional, default False`*) – Delete duplicate gridpoints and zones. *Requires numpy >= 1.13.0*.

F.3 zone.import_flac

`toughflac.zone.`**`import_flac`**`(`*filename=None*, *binary=False*`)`

Import f3grid file in FLAC3D.

Parameters

- **`filename`** (*`str or None, default None`*) – Input file name. If *None*, a file dialog window will be opened.
- **`binary`** (*`bool, optional, default False`*) – If *True*, read input file as binary.

F.4 zone.import_tough_output

`toughflac.zone.`**`import_tough_output`**`(`*filename=None*`)`

Import TOUGH output file for each time step.

Parameters **`filename`** (*`str or None, default None`*) – Input file name. If *None*, a file dialog window will be opened.

Returns

- *dict* – TOUGH output with one item per variable as an array of shape (nt, nzone).
- *array_like* – Time steps array.

F.5 zone.import_tough_save

`toughflac.zone.`**`import_tough_save`**`(`*filename=None*`)`

Import TOUGH SAVE file as FLAC3D extra variables.

Primary variables are saved in slots 2 to N+1. Porosity is saved in slot 1.

Parameters **`filename`** (*`str or None, default None`*) – Input file name. If *None*, a file dialog window will be opened.

F.6 zone.export

`toughflac.zone.`**`export`**`(`*filename*, *file_format='flac3d'*, ***kwargs*`)`

Export current FLAC3D grid.

Parameters

- **`filename`** (*`str`*) – Output file name.
- **`file_format`** (*`str, optional, default 'flac3d'`*) – Output file format.

Other Parameters

- **nodal_distance** (*str ('line' or 'orthogonal'), optional, default 'line'*) – Only if `file_format = "tough"`. Method to calculate connection nodal distances: - 'line': distance between node and common face along connecting line (distance is not normal), - 'orthogonal' : distance between node and its orthogonal projection onto common face (shortest distance).
- **porosity** (*array_like, callable or None, optional, default None*) – Only if `file_format = "tough"`. Porosity array (nzone,) or function to calculate porosity (with input `itasca.Zone`).
- **permeability** (*array_like, callable or None, optional, default None*) – Only if `file_format = "tough"`. Permeability modifiers array (nzone,) or (nzone, 3) or function to calculate permeability modifiers (with input `itasca.Zone`).
- **group_name** (*dict or None, default None*) – Only if `file_format = "tough"`. Rename zone group.
- **group_end** (*str, array_like or None, default None*) – Only if `file_format = "tough"`. Move elements to bottom of block 'ELEME' if they belong to a group in *group_end*.
- **slot** (*str or None, default None*) – Only if `file_format = "tough"`. Slot in which to get group name. If *None*, group names are retrieved from slot 'Default'.
- **incon** (*bool, optional, default False*) – Only if `file_format = "tough"`. If *True*, initial conditions will be written in file *INCON*.

F.7 zone.export_tough

`toughflac.zone.`**`export_tough`**`(`*filename='MESH'*, *nodal_distance='line'*, *porosity=None*, *permeability=None*, *group_name=None*, *group_end=None*, *slot=None*, *incon=False*`)`

Write TOUGH input MESH file consistent with current FLAC3D grid.

If *incon* is *True*, also write TOUGH input *INCON* file if initial conditions have been set using *`toughflac.zone.initialize_pvariables()`* with the correct number of primary variables (no check is performed). Porosity and permeability modifiers can also be exported in columns 16-60 of *INCON*.

Parameters

- **`filename`** (str, optional, default *MESH*) – Output file name.
- **`nodal_distance`** (*`str ('line' or 'orthogonal'), optional, default 'line'`*) – Method to calculate connection nodal distances: - 'line': distance between node and common face along connecting line (distance is not normal), - 'orthogonal' : distance between node and its orthogonal projection onto common face (shortest distance).

- **porosity** (*array_like, callable or None, optional, default None*) – Porosity array (nzone,) or function to calculate porosity (with input `itasca.Zone`).
- **permeability** (*array_like, callable or None, optional, default None*) – Permeability modifiers array (nzone,) or (nzone, 3) or function to calculate permeability modifiers (with input `itasca.Zone`).
- **group_name** (*dict or None, default None*) – Rename zone group.
- **group_end** (*str, array_like or None, default None*) – Move elements to bottom of block 'ELEME' if they belong to a group in *group_end*.
- **slot** (*str or None, default None*) – Slot in which to get group name. If *None*, group names are retrieved from slot 'Default'.
- **incon** (*bool, optional, default False*) – If *True*, initial conditions will be written in file *INCON*.

Note: This function is a shortcut for `toughflac.zone.export(filename, file_format = "tough", **kwargs)`.

TOUGH input *MESH* file is composed of two blocks (`ELEME` and `CONNE`). All the parameters required to write the block `ELEME` are imported from FLAC3D (i.e. element material, volume and center). Connections between elements are reconstructed using the function `itasca.zonearray.neighbors()` and areas of common interfaces imported using function `itasca.Zone.face_areas()`.

The line connecting the centers of two adjacent elements is usually not normal to their common interface as required by TOUGH3 and the finite-volume approach. This is only true with Voronoi tessellations and regular structured hexahedral meshes (e.g. TOUGH3 MeshMaker). Thus, the nodal distances can only be approximated for most types of meshes (e.g. tetrahedra). In the current version, two methods have been implemented which can be selected with the option `nodal_distance`:

1. Distance between node and common face along connecting line (default, Figure 6.1 (right)),
2. Distance between node and its orthogonal projection onto common face (default, Figure 6.1 (middle)).

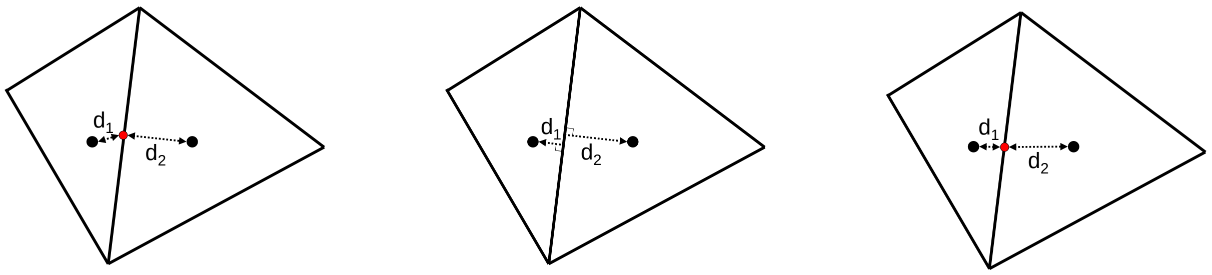


Figure 6.1: 2D view of calculation of nodal distances d_1 and d_2 as distances between nodes and (Left) center of common interface, (Middle) orthogonal projection onto common interface, and (Right) intersection point of line connecting the two nodes and common interface.

For Dirichlet boundary elements, nodal distances are fixed to `1.0e-9` as recommended in TOUGH3 user guide.

F.8 zone.export_flac

`toughflac.zone.`**`export_flac`**`(`*filename*, *binary=False*`)`

Export current FLAC3D grid to a f3grid file.

Parameters

- **`filename`** (*`str`*) – Output file name.
- **`binary`** (*`bool, optional, default False`*) – If *True*, export as binary file.

Note: This function calls FLAC3D **`zone export filename b`**.

F.9 zone.export_mesh

`toughflac.zone.`**`export_mesh`**`(`*filename*, *attribute=None*, *file_format=None*, *interpolate_cell_data=False*, *n_phase=None*, *extra_func=None*`)`

Export queried attributes to external mesh.

This function requires `toughio` to be installed.

Parameters

- **`filename`** (*`str`*) – Output file name.
- **`attribute`** (*`str, list of str or None, default None`*) – Attribute or list of attributes (available in `itasca`) to export.
- **`file_format`** (*`str or None, optional, default None`*) – Input file format. If *None*, it will be guessed from file's extension.
- **`interpolate_cell_data`** (*`bool, optional, default False`*) – If *True*, cell data are interpolated onto grid points using volumetric averaging.
- **`n_phase`** (*`int or None, optional, default None`*) – Number of phases (for attributes 'saturation' and 'pcap').
- **`extra_func`** (*`dict or None, optional, default None`*) – Extra variables to export with custom functions (zone only).

F.10 zone.export_time_series

`toughflac.zone.`**`export_time_series`**`(`*filename*, *savefiles*, *attribute*, *time_steps=None*, *interpolate_cell_data=False*, *n_phase=None*, *extra_func=None*`)`

Export FLAC3D save files as XDMF time series (can be visualized with ParaView).

This function requires `toughio` to be installed.

Parameters

- **`filename`** (*`str`*) – Output file name.
- **`savefiles`** (*`list of str`*) – List of FLAC3D save files to export for each time step.
- **`attribute`** (*`str, list of str or None, default None`*) – Attribute or list of attributes (available in `itasca`) to export.

- **`time_steps`** (*`array_like or None, optional, default None`*) – List of time steps (in seconds). The order of *time_steps* must be consistent with *savefiles*.
- **`interpolate_cell_data`** (*`bool, optional, default False`*) – If *True*, cell data are interpolated onto grid points using volumetric averaging.
- **`n_phase`** (*`int or None, optional, default None`*) – Number of phases (for attributes 'saturation' and 'pcap').
- **`extra_func`** (*`dict or None, optional, default None`*) – Extra variables to export with custom functions (zone only).

F.11 zone.in_group

`toughflac.zone.`**`in_group`**`(`*group*, *slot=None*`)`

Return zone group membership as a bool array. Support multiple groups.

Parameters

- **`group`** (*`str or array_like`*) – Group or list of groups to query.
- **`slot`** (*`str or None, default None`*) – Slot in which to get group name. If *None*, group names are retrieved from slot 'Default'.

Returns Group membership as a bool array.

Return type array_like

F.12 zone.near

`toughflac.zone.`**`near`**`(`*point*, *group=None*, *return_id=False*, *return_neighbors=False*, *slot=None*`)`

Return element name (in MESH file) nearest to query point.

Parameters

- **`point`** (*`array_like`*) – Coordinates of point to query.
- **`group`** (*`str, array_like or None, optional, default None`*) – Group or list of groups in which nearest zone is looked for.
- **`return_id`** (*`bool, optional, False`*) – If *True*, also return zone ID.
- **`return_neighbors`** (*`bool, optional, default False`*) – If *True*, also return labels of connected elements with their positions.
- **`slot`** (*`str or None, default None`*) – Slot in which to get group name. If *None*, group names are retrieved from slot 'Default'.

Returns

- **label** (*str*) – Queried point label.
- **id** (*int*) – Queried point ID. Only if `return_id == True`.
- **neighbors** (*dict, optional*) – Queried point neighbors labels and positions. Only if `return_neighbors == True`.

F.13 zone.group

`toughflac.zone.`**`group`**`(`*name*, *range_x=None*, *range_y=None*, *range_z=None*`)`

Assign zone within *range_x*, *range_y* and *range_z* to group *name*.

Parameters

- **`name`** (*`str`*) – Group name.
- **`range_x`** (*`array_like or None, optional, default None`*) – Minimum and maximum values in X direction.
- **`range_y`** (*`array_like or None, optional, default None`*) – Minimum and maximum values in Y direction.
- **`range_z`** (*`array_like or None, optional, default None`*) – Minimum and maximum values in Z direction.

Note: This function calls FLAC3D **`zone group s range`**.

F.14 zone.set_dirichlet_bc

`toughflac.zone.`**`set_dirichlet_bc`**`(`*group*, *slot=None*`)`

Set time-independent Dirichlet boundary conditions for TOUGH elements.

Parameters

- **`group`** (*`str`*) – Group name.
- **`slot`** (*`str or None, default None`*) – Slot in which to get group name. If *None*, group names are retrieved from slot 'Default'.

Note: This function creates an extra array in slot 1. This extra array is imported by *`toughflac.zone.export_tough()`*.

F.15 zone.initialize_pvariables

`toughflac.zone.`**`initialize_pvariables`**`(`*x1=None*, *x2=None*, *x3=None*, *x4=None*, *group=None*, *slot=None*`)`

Initialize primary variables for *INCON*.

Parameters

- **`x1`** (*`callable or None, optional, default None`*) – Function to initialize primary variable X1.
- **`x2`** (*`callable or None, optional, default None`*) – Function to initialize primary variable X2.
- **`x3`** (*`callable or None, optional, default None`*) – Function to initialize primary variable X3.
- **`x4`** (*`callable or None, optional, default None`*) – Function to initialize primary variable X4.
- **`group`** (*`str, tuple, list or None, optional, default None`*) – Group(s) to which initial conditions are applied. If *None*, initial conditions are applied to all zones.

- **slot** (*str or None, default None*) – Slot in which to get group name. If *None*, group names are retrieved from slot 'Default'.

Note:

1. This function creates extra arrays in slots 2, 3, 4 and 5, which means that initial conditions are calculated and applied at the time this function is called. These extra arrays are imported by *toughflac.zone.export_tough()* if *incon* is *True*,
2. Currently, only four primary variables are supported.

Examples

Input functions should have 1 argument corresponding to the depth of the zones (even for constant initial conditions). For instance, the user can initialize the four primary variables of equation-of-state ECO2N for the whole grid using anonymous functions *lambda*:

```
>>> tf.zone.initialize_pvariables(
>>>     x1=lambda z: 1.0e5 - 9810.0 * z,
>>>     x2=lambda z: 0.05,
>>>     x3=lambda z: 0.0,
>>>     x4=lambda z: 10.0 - 0.025 * z,
>>> )
```

This will initialize the pressure (X1) at hydrostatic pressure and the temperature (X4) with a thermal gradient of 25 deg/km (1e5 Pa pressure and 10 deg temperature at z = 0). The other two primary variables X2 and X3 are constant.

Initial conditions can also be applied for specific groups using the argument *group*:

```
>>> tf.zone.initialize_pvariables(
>>>     x1=lambda z: 1.5e5,
>>>     x2=lambda z: 0.3e-2,
>>>     x3=lambda z: 0.0,
>>>     x4=lambda z: 8.0,
>>>     group=["TOPBC", "TOPFC"],
>>> )
>>> tf.zone.initialize_pvariables(
>>>     x1=lambda z: 1.77e6,
>>>     x2=lambda z: 0.3e-2,
>>>     x3=lambda z: 0.0,
>>>     x4=lambda z: 98.0,
>>>     group="BOTBC",
>>> )
```

Note that this particular example is no different from setting 'INDOM' in TOUGH input file, but this approach can also be used to apply gradient initial conditions for a specific group.

F.16 zone.create

F.16.1 zone.create.tartan_brick

toughflac.zone.create.**tartan_brick**(*dx, dy, dz, point_0=[0.0, 0.0, 0.0], group=None*)
Create a 3D irregular cartesian grid.

Parameters

- **dx** (*array_like*) – Step (in meters) in X direction.

- **dy** (*array_like*) – Step (in meters) in Y direction.
- **dz** (*array_like*) – Step (in meters) in Z direction.
- **point_0** (*list, optional, default [0., 0., 0.]*) – Origin point coordinate.
- **group** (*str or None, optional, default None*) – Grid zone group name.

Note: This function creates a regular brick by calling FLAC3D **`zone create brick size nx ny nz`** where `nx`, `ny` and `nz` are the size of *dx*, *dy* and *dz*. Correct gridpoint positions are then calculated using `numpy` and then modified using `itasca.gridpointarray.set_pos()`.

Appendix G

Zonearray

Submodule `zonearray` provides functions to easily access extra variable arrays saved during the coupling and is organized as follow:

```
zonearray
  ├─permeability
  ├─porosity
  ├─porosity_initial
  ├─porosity_delta
  ├─pp
  ├─pp_equivalent
  ├─pp_delta
  ├─temperature
  ├─temperature_delta
  ├─strain_vol
  ├─stress_delta
  ├─stress_delta_flat
  ├─stress_delta_prin_x
  ├─stress_delta_prin_y
  ├─stress_delta_prin_z
  ├─stress_delta_min
  ├─stress_delta_max
  ├─stress_delta_mean
  ├─density
  ├─biot
  ├─therm_coeff
  ├─saturation
  └─pcap
```

G.1 zonearray.permeability

`toughflac.zonearray.`**`permeability`**`()`

Get an array of the zone permeabilities.

G.2 zonearray.porosity

`toughflac.zonearray.`**`porosity`**`()`

Get an array of the zone porosities.

G.3 zonearray.porosity_initial

`toughflac.zonearray.`**`porosity_initial`**`()`

Get an array of the zone initial porosity.

G.4 zonearray.porosity_delta

`toughflac.zonearray.`**`porosity_delta`**`()`

Get an array of the zone changes in porosity.

G.5 zonearray.pp

`toughflac.zonearray.`**`pp`**`()`

Get an array of the zone pore pressures.

Note: Zone pore pressures in `itasca` are equivalent pore pressures scaled by Biot coefficients.

G.6 zonearray.pp_equivalent

`toughflac.zonearray.`**`pp_equivalent`**`()`

Get an array of the zone equivalent pore pressures.

G.7 zonearray.pp_delta

`toughflac.zonearray.`**`pp_delta`**`()`

Get an array of the zone changes in pore pressure.

G.8 zonearray.temperature

`toughflac.zonearray.`**`temperature`**`()`
Get an array of the zone temperatures.

G.9 zonearray.temperature_delta

`toughflac.zonearray.`**`temperature_delta`**`()`
Get an array of the zone changes in temperature.

G.10 zonearray.strain_vol

`toughflac.zonearray.`**`strain_vol`**`()`
Get an array of the zone volumetric strains.

G.11 zonearray.stress_delta

`toughflac.zonearray.`**`stress_delta`**`()`
Get an array of the zone changes in stress.

G.12 zonearray.stress_delta_flat

`toughflac.zonearray.`**`stress_delta_flat`**`()`
Get an array of the zone changes in stress.

The return value is a 2D array where the component ordering is xx, yy, zz, xy, yz, xz.

G.13 zonearray.stress_delta_prin_x

`toughflac.zonearray.`**`stress_delta_prin_x`**`()`
Get an array of the x-component of the zone changes in principal stress.

G.14 zonearray.stress_delta_prin_y

`toughflac.zonearray.`**`stress_delta_prin_y`**`()`
Get an array of the y-component of the zone changes in principal stress.

G.15 zonearray.stress_delta_prin_z

`toughflac.zonearray.`**`stress_delta_prin_z`**`()`
Get an array of the z-component of the zone changes in principal stress.

G.16 zonearray.stress_delta_min

`toughflac.zonearray.`**`stress_delta_min`**`()`

Get an array of the zone minimum changes in principal stress.

G.17 zonearray.stress_delta_max

`toughflac.zonearray.`**`stress_delta_max`**`()`

Get an array of the zone maximum changes in principal stress.

G.18 zonearray.stress_delta_mean

`toughflac.zonearray.`**`stress_delta_mean`**`()`

Get an array of the zone average changes in stress.

G.19 zonearray.density

`toughflac.zonearray.`**`density`**`()`

Get an array of the zone densities.

G.20 zonearray.biot

`toughflac.zonearray.`**`biot`**`()`

Get an array of the zone Biot coefficients.

G.21 zonearray.therm_coeff

`toughflac.zonearray.`**`therm_coeff`**`()`

Get an array of the zone volumetric thermal expansion coefficients.

G.22 zonearray.saturation

`toughflac.zonearray.`**`saturation`**`(`*n_phase*`)`

Get an array of the zone saturations for each phase.

Parameters **`n_phase`** (*`int`*) – Number of phases.

G.23 zonearray.pcap

`toughflac.zonearray.`**`pcap`**`(`*n_phase*`)`

Get an array of the zone capillary pressures for each phase.

Parameters **`n_phase`** (*`int`*) – Number of phases.

Appendix H

Attributes

Table 8.1 summarizes all the attributes that can be exported during the coupling (i.e. history variables) or with the functions *toughflac.zone.export_mesh()* and *toughflac.zone.export_time_series()*.

Table 8.1: List of available attributes in toughflac.

Attribute	Description
"disp_mag"	Magnitude of gridpoint displacement vector.
"disp_x"	x-component of gridpoint displacement vector.
"disp_y"	y-component of gridpoint displacement vector.
"disp_z"	z-component of gridpoint displacement vector.
"temp"	Zone temperature.
"pp"	Zone pore pressure.
"stress_xx"	xx-component of the stress tensor.
"stress_yy"	yy-component of the stress tensor.
"stress_zz"	zz-component of the stress tensor.
"stress_xy"	xy-component of the stress tensor.
"stress_yz"	yz-component of the stress tensor.
"stress_xz"	xz-component of the stress tensor.
"stress_prin_x"	x-component of the principal stress.
"stress_prin_y"	y-component of the principal stress.
"stress_prin_z"	z-component of the principal stress.
"strain_xx"	xx-component of the strain tensor.
"strain_yy"	yy-component of the strain tensor.
"strain_zz"	zz-component of the strain tensor.
"strain_xy"	xy-component of the strain tensor.
"strain_yz"	yz-component of the strain tensor.
"strain_xz"	xz-component of the strain tensor.
"stress_delta_xx"	xx-component of the change in stress tensor.
"stress_delta_yy"	yy-component of the change in stress tensor.
"stress_delta_zz"	zz-component of the change in stress tensor.
"stress_delta_xy"	xy-component of the change in stress tensor.
"stress_delta_yz"	yz-component of the change in stress tensor.
"stress_delta_xz"	xz-component of the change in stress tensor.
"stress_delta_prin_x"	x-component of the change in principal stress.
"stress_delta_prin_y"	y-component of the change in principal stress.
"stress_delta_prin_z"	z-component of the change in principal stress.
"temp_delta"	Zone change in temperature.
"pp_delta"	Zone change in pore pressure.
"strain_vol"	Zone volumetric strain.

Continued on next page

Table 8.1 – continued from previous page

Attribute	Description
`"pp_equivalent"`	Zone equivalent pore pressure.
`"density"`	Zone density.
`"porosity"`	Zone porosity.
`"porosity_delta"`	Zone change in porosity.
`"porosity_initial"`	Zone initial porosity.
`"permeability"`	Zone permeability.
`"biot"`	Zone Biot's coefficient.
`"therm_coeff"`	Zone volumetric thermal expansion coefficient.
`"saturation_i"`	Zone saturation for phase `i` (integer).
`"pcap_i"`	Zone capillary pressure for phase `i` (integer).
`"prop_s"`	Property `s` (string corresponding to a property available in FLAC3D).
`"extra_i"`	Extra variable in slot `i` (integer from 1 to 128).

Bibliography

[Cappa & Rutqvist, 2011a] Cappa, F., & Rutqvist, J. (2011 , sep). Impact of CO2 geological sequestration on the nucleation of earthquakes. *Geophysical Research Letters*, *38*(17). URL: http://doi.wiley.com/10.1029/2011GL048487, doi:10.1029/2011GL048487

[Cappa & Rutqvist, 2011b] Cappa, F., & Rutqvist, J. (2011 , mar). Modeling of coupled deformation and permeability evolution during fault reactivation induced by deep underground injection of CO2. *International Journal of Greenhouse Gas Control*, *5*(2), 336–346. URL: https://linkinghub.elsevier.com/retrieve/pii/S1750583610001337, doi:10.1016/j.ijggc.2010.08.005

[Cappa et al., 2009] Cappa, F., Rutqvist, J., & Yamamoto, K. (2009 , oct). Modeling crustal deformation and rupture processes related to upwelling of deep CO2-rich fluids during the 1965-1967 Matsushiro earthquake swarm in Japan. *Journal of Geophysical Research*, *114*(B10), B10304. URL: http://doi.wiley.com/10.1029/2009JB006398, doi:10.1029/2009JB006398

[Kim et al., 2012] Kim, H.-M., Rutqvist, J., Ryu, D.-W., Choi, B.-H., Sunwoo, C., & Song, W.-K. (2012 , apr). Exploring the concept of compressed air energy storage (CAES) in lined rock caverns at shallow depth: A modeling study of air tightness and energy balance. *Applied Energy*, *92*, 653–667. URL: https://linkinghub.elsevier.com/retrieve/pii/S0306261911004582, doi:10.1016/j.apenergy.2011.07.013

[Kim et al., 2011] Kim, J., Yang, D., Moridis, G. J., & Rutqvist, J. (2011 , apr). Numerical Studies on Two-way Coupled Fluid Flow and Geomechanics in Hydrate Deposits. *SPE Reservoir Simulation Symposium*. Society of Petroleum Engineers. URL: http://www.onepetro.org/doi/10.2118/141304-MS, doi:10.2118/141304-MS

[Rutqvist et al., 2005] Rutqvist, J., Barr, D., Datta, R., Gens, A., Millard, A., Olivella, S., . . . Tsang, Y. (2005 , jul). Coupled thermal-hydrological-mechanical analyses of the Yucca Mountain Drift Scale Test - Comparison of field measurements to predictions of four different numerical models. *International Journal of Rock Mechanics and Mining Sciences*, *42*(5-6), 680–697. URL: https://linkinghub.elsevier.com/retrieve/pii/S1365160905000353, doi:10.1016/j.ijrmms.2005.03.008

[Rutqvist et al., 2007] Rutqvist, J., Birkholzer, J., Cappa, F., & Tsang, C.-F. (2007 , jun). Estimating maximum sustainable injection pressure during geological sequestration of CO2 using coupled fluid flow and geomechanical fault-slip analysis. *Energy Conversion and Management*, *48*(6), 1798–1807. URL: https://linkinghub.elsevier.com/retrieve/pii/S0196890407000337, doi:10.1016/j.enconman.2007.01.021

[Rutqvist et al., 2008a] Rutqvist, J., Birkholzer, J.T., & Tsang, C.-F. (2008 , feb). Coupled reservoir geomechanical analysis of the potential for tensile and shear failure associated with CO2 injection in multilayered reservoir caprock systems. *International Journal of Rock Mechanics and Mining Sciences*, *45*(2), 132–143. URL: https://linkinghub.elsevier.com/retrieve/pii/S1365160907000548, doi:10.1016/j.ijrmms.2007.04.006

[Rutqvist et al., 2008b] Rutqvist, J., Freifeld, B., Min, K.-B., Elsworth, D., & Tsang, Y. (2008 , dec). Analysis of thermally induced changes in fractured rock permeability during 8 years of heating and cooling at the Yucca Mountain Drift Scale Test. *International Journal of Rock Mechanics and Mining Sciences*, *45*(8), 1373–1389. URL: https://linkinghub.elsevier.com/retrieve/pii/S1365160908000208, doi:10.1016/j.ijrmms.2008.01.016

[Rutqvist et al., 2009a] Rutqvist, J., Moridis, G.J., Grover, T., & Collett, T. (2009 , jul). Geomechanical response of permafrost-associated hydrate deposits to depressurization-induced gas production. *Journal of Petroleum Science and Engineering*, *67*(1-2), 1–12. URL: https://linkinghub.elsevier.com/retrieve/pii/S0920410509000801, doi:10.1016/j.petrol.2009.02.013

[Rutqvist & Tsang, 2005] Rutqvist, J., & Tsang, C.-F. (2005). Coupled hydromechanical effects of CO2 injection. *Development in Water Sciences*, *52*, 649–679. URL: https://linkinghub.elsevier.com/retrieve/pii/S0167564805520501, doi:10.1016/S0167-5648(05)52050-1

[Rutqvist et al., 2002] Rutqvist, J., Wu, Y.-S., Tsang, C.-F., & Bodvarsson, G. (2002 , jun). A modeling approach for analysis of coupled multiphase fluid flow, heat transfer, and deformation in fractured porous rock. *International Journal of Rock Mechanics and Mining Sciences*, *39*(4), 429–442. URL: https://linkinghub.elsevier.com/retrieve/pii/S1365160902000229, doi:10.1016/S1365-1609(02)00022-9

[Rutqvist et al., 2009b] Rutqvist, J., Barr, D., Birkholzer, J. T., Fujisaki, K., Kolditz, O., Liu, Q.-S., . . . Zhang, C.-Y. (2009 , may). A comparative simulation study of coupled THM processes and their effect on fractured rock permeability around nuclear waste repositories. *Environmental Geology*, *57*(6), 1347–1360. URL: http://link.springer.com/10.1007/s00254-008-1552-1, doi:10.1007/s00254-008-1552-1

[Rutqvist et al., 2006] Rutqvist, J., Birkholzer, J., Cappa, F., Oldenburg, C., & Tsang, C.-F. (2006). Shear-Slip Analysis in Multiphase Fluid-Flow Reservoir Engineering Applications Using Tough-Flac. *TOUGH Symposium 2006*, pp. 1–9.

[Rutqvist et al., 2012] Rutqvist, J., Kim, H.-M., Ryu, D.-W., Synn, J.-H., & Song, W.-K. (2012 , jun). Modeling of coupled thermodynamic and geomechanical performance of underground compressed air energy storage in lined rock caverns. *International Journal of Rock Mechanics and Mining Sciences*, *52*, 71–81. URL: https://linkinghub.elsevier.com/retrieve/pii/S1365160912000408, doi:10.1016/j.ijrmms.2012.02.010

[Rutqvist & Moridis, 2007] Rutqvist, J., & Moridis, G. J. (2007 , apr). Numerical Studies on the Geomechanical Stability of Hydrate-Bearing Sediments. *Offshore Technology Conference*. Offshore Technology Conference. URL: http://www.onepetro.org/doi/10.4043/18860-MS, doi:10.4043/18860-MS

[Rutqvist & Tsang, 2003a] Rutqvist, J., & Tsang, C. (2003). TOUGH-FLAC: a numerical simulator for analysis of coupled thermal-hydrologic-mechanical processes in fractured and porous geological media under multi-phase flow conditions. *Proceedings of the TOUGH Symposium*, pp. 1–9.

[Rutqvist & Tsang, 2002] Rutqvist, J., & Tsang, C.-F. (2002 , jun). A study of caprock hydromechanical changes associated with CO2-injection into a brine formation. *Environmental Geology*, *42*(2-3), 296–305. URL: http://link.springer.com/10.1007/s00254-001-0499-2, doi:10.1007/s00254-001-0499-2

[Rutqvist & Tsang, 2003b] Rutqvist, J., & Tsang, C.-F. (2003 , apr). Analysis of thermal-hydrologic-mechanical behavior near an emplacement drift at Yucca Mountain. *Journal of Contaminant Hydrology*, *62-63*, 637–652. URL: https://linkinghub.elsevier.com/retrieve/pii/S0169772202001845, doi:10.1016/S0169-7722(02)00184-5

[Rutqvist et al., 2010] Rutqvist, J., Vasco, D. W., & Myer, L. (2010 , mar). Coupled reservoir-geomechanical analysis of CO2 injection and ground deformations at In Salah, Algeria. *International Journal of Greenhouse Gas Control*, *4*(2), 225–230. URL: https://linkinghub.elsevier.com/retrieve/pii/S1750583609001388, doi:10.1016/j.ijggc.2009.10.017

[Todesco et al., 2004] Todesco, M., Rutqvist, J., Chiodini, G., Pruess, K., & Oldenburg, C. M. (2004 , aug). Modeling of recent volcanic episodes at Phlegrean Fields (Italy): geochemical variations and ground deformation. *Geothermics*, *33*(4), 531–547. URL: https://linkinghub.elsevier.com/retrieve/pii/S0375650503001536, doi:10.1016/j.geothermics.2003.08.014