Documentation

Version 1.1.0

September 2024

CONTENTS

.

.

The **SafeInCave** simulator is developed to study the mechanical behavior/stability of salt caverns for gas storage. This is a three-dimensional simulator developed in Python, in which the finite element implementation is based on FEniCS 2019.1.

Features:

- Fully open-source and documented.
- Comprehensive constitutive model able to capture transient creep, steady-state (dislocation) creep, and reverse transient creep.
- Robust numerical formulation for non-linear mechanics achieved by computing the consistent tangent matrix.
- Different choices of time integration schemes: explicit, Crank-Nicolson and fully-implicit.
- Interaction with the simulator happens through a single input file.
- No lines of code are necessary, except for building the input file (if necessary).
- Time-dependent and non-uniform boundary conditions can be assigned for the overburden, sideburden and gas pressure.
- Tests are added for the main classes and functions, thus enhancing robustness and facilitating external contributions.

Current members:

- [Hermínio Tasinafo Honório] (H.TasinafoHonorio@tudelft.nl), Maintainer, 2023-present
- [Hadi Hajibeygi] (h.hajibeygi@tudelft.nl), Principal Investigator

Acknowledgements:

This simulator was developed as a deliverable of a project also called **SafeInCave** and financially supported by Shell.

CHAPTER

ONE

GETTING STARTED

This chapter presents the installation steps for SafeInCave simulator and its dependencies. It also shows how to run a simple simulation of a triaxial test performed on a salt rock cubic sample. See *Tutorials* for more examples with detailed descriptions.

1.1 Installation

The SafeInCave simulator has been developed and tested only on Windows platform, but it should also work with other operational systems. In this section, only Windows installation is covered. The user can also access our Youtube channel and checkout our video tutorial on how to install SafeInCave simulator and its dependencies.

1.1.1 Windows Subsystem for Linux (WSL)

Because SafeInCave is based on FEniCS 2019.1, which can be installed on Ubuntu, the Windows installation requires the WSL. To install WSL, open the Power Shell in **administrator mode** and run the following commands:

```
PS C:\Users\user> Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Windows-
↪Subsystem-Linux
```

Type *yes* to restart the system. After restarting, open Power Shell once more in administrator mode and run:

```
PS C:\Users\user> wsl --update
```

1.1.2 Ubuntu

To install Ubuntu, run the following command on Power Shell:

```
>> wsl --install -d ubuntu
```

At this point, the Ubuntu terminal should be available at *Start button -> Ubuntu*.

1.1.3 Pip3

Most of the dependencies of the SafeInCave simulator are installed using *pip3*. To install it, run the following command on Ubuntu terminal:

```
user@computer:~$ sudo apt-get update
user@computer:~$ sudo apt install python3-pip
```

1.1.4 Matplotlib

Matplotlib should be installed using *pip3*. However, in order to allow for GUI interaction, it is advise to install Tkinter first:

```
user@computer:~$ sudo apt-get install python3-tk
```

To install Matplotlib, run the following command:

```
user@computer:~$ pip3 install matplotlib
```

1.1.5 Numpy

Install numpy version 1.23.5:

```
user@computer:~$ pip3 install numpy=1.23.5
```

1.1.6 Meshio

In this current version of SafeInCave, the meshio verion 3.3.1 is adopted. This version can be installed using *pip3* command, that is:

```
user@computer:~$ pip3 install meshio=3.3.1
```

1.1.7 Pytorch

Pytorch is used in SafeInCave to perform tensor operations in a efficient way. To install PyTorch, follow these instructions. For example, if you want to install it on you CPU, run:

```
user@computer:~$ conda install pytorch torchvision torchaudio cpuonly -c pytorch
```

Otherwise, if you have GPU in your machine, run:

```
user@computer:~$ conda install pytorch torchvision torchaudio pytorch-cuda=11.8 -c␣
↪pytorch -c nvidia
```

1.1.8 Pandas

Pandas is another useful package for manipulating data during post-processing. To install pandas, run the following commands on Ubuntu terminal:

```
user@computer:~$ pip3 install pandas==1.4.3
```

1.1.9 Gmsh

Gmsh is used for creating the grids in SafeInCave. First, install package *libgl1* and the install Gmsh from conda-forge, that is:

```
user@computer:~$ sudo apt install gmsh
```

1.1.10 FEniCS

There are different options for installing FEniCS 2019.1, as detailed here. To install it on Ubuntu, run the following commands:

```
user@computer:~$ sudo apt-get install software-properties-common
user@computer:~$ sudo add-apt-repository ppa:fenics-packages/fenics
user@computer:~$ sudo apt-get update
user@computer:~$ sudo apt-get install fenics
```

1.1.11 SafeInCave

To use SafeInCave simulator, clone the Gitlab repository to your machince:

```
user@computer:~$ git clone https://gitlab.tudelft.nl/ADMIRE_Public/safeincave.git
```

1.2 Running your first simulation

The fastest way to run a simulation is to execute the *main.py* file of one of the examples in the *examples* folder. The example in folder *safeincave/examples/triaxial* simulates a triaxial test performed on a salt rock sample of cubic shape. The salt sample is subjected to a constant confining pressure (horizontal stresses) and varying axial (vertical) load, which can be visualized by executing the *plot_bcs.py* file, that is

```
user@computer:~$ cd safeincave/examples/triaxial
user@computer:~/safeincave/examples/triaxial$ python plot_bcs.py
```

which produces the image shown in Fig. 1.1.

To run this example, simply do the following:

```
user@computer:~/safeincave/examples/triaxial$ python main.py
```

Once the simulation is finished, the results can be found in folder *triaxial/output/case_0*.

The results can be visualized on Paraview. Alternatively, use matplotlib to visualize results by doing the following:

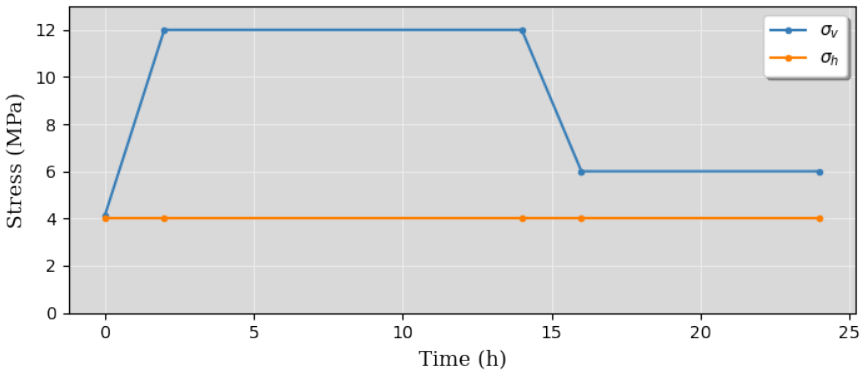


Fig. 1.1: Loading schedule for the triaxial test example.

```
user@computer:~/safeincave/examples/triaxial$ python plot_results.py
```

This will generate Fig. 1.2, which shows the vertical (ε_v) and horizontal (ε_h) deformation over time.

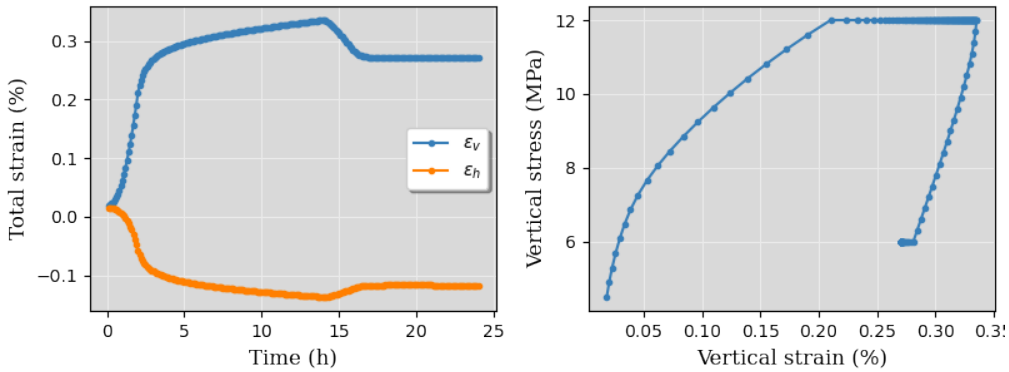


Fig. 1.2: Results obtained from the triaxial test simulation.

CHAPTER

TWO

THE INPUT FILE

The SafeInCave simulator runs entirely based on a single input file in JSON format. This chapter describes the structure of the input file and how it can be created automatically using class **InputFileAssistant**.

2.1 Main structure

The input file must be in JSON format, and it requires the following sections:

```
{
    "grid": {},
    "output": {},
    "solver_settings": {},
    "time_settings": {},
    "simulation_settings": {},
    "body_force": {},
    "boundary_conditions": {},
    "constitutive_model": {}
}
```

A detailed explanation of each section and the required content within each one is presented next.

2.1.1 Section *grid*

The section grid informs the grid to be used in the simulation. This section requires two keys: (1) *path* and (2) *name*. The key path specifies the relative path to the directory where the grid stored. The key *name* indicates the name of the grid files, which is usually *geom* (e.g. *geom.xml*, *geom_facet_region.msh*, *geom_physical_region.xml*). The snippet Listing 2.1 illustrates a typical example.

Listing 2.1: Input file section: *grid*.

```
{
    "grid": {
        "path": "../../grids/cube_0",
        "name": "geom"
    },
}
```

2.1.2 Section *output*

The section *output* only requires the key *path*, which is specifies the relative path to the directory where the output files will be saved. This is illustrated in Listing 2.2, where the results are saved in the directory *output/case_name*. This directory is automatically created in case it does not exist.

Listing 2.2: Input file section: *output*

```
{
    "output": {
        "path": "output/case_name"
     },
}
```

2.1.3 Section *solver_settings*

This section specifies which solver is used to solve the linear systems. The required keys for *solver_settings* are *type* and *method*. The *type* key can be either *LU* or *KrylovSolver*, for direct LU decomposition of a Krylov-based solver, respectively. If *LU* is chosen, the *method* key can be either *default*, *umfpack*, *mumps*, *pastix*, *superlu*, *superlu_dist*, or *petsc*, dependint on how PETSc has been installed. For example,

Listing 2.3: Input file section: *solver_settings* (LUSolver)

```
{
    "solver_settings": {
        "type": "LU",
        "method": "petsc"
     },
}
```

A Krylov-based solver can be chosen by specifying the keyword *KrylovSolver* to the *type* key. The specific Krylov solver is defined under the key *method*, and the main options are: *cg*, *bicg*, *bigcstab*, and *gmres*. In addition to *type* and *method*, the *KrylovSolver* requires keys *preconditioner* and *relative_tolerance*. The main options for key *preconditioner* are: *icc*, *ilu*, *petsc_amg*, *sor*, and *hypre*. For example,

Listing 2.4: Input file section: *solver_settings* (KrylovSolver)

```
{
    "solver_settings": {
        "type": "KrylovSolver",
        "method": "cg",
        "preconditioner": "petsc_amg",
        "relative_tolerance": 1e-12
     },
}
```

2.1.4 Section *simulation_settings*

This section specifies whether or not to compute the equilibrium condition before the actual simulation begins. It requires the *equilibrium* and *operation* keys, which specifies the settings for the equilibrium and operation simulation stages, respectively. In the equilibrium condition, the stresses specified at the initial time $t = 0$ are applied to the geometry and a simulation is run considering only the **elastic** and **viscoelastic** (if present) part of the constitutive model. This equilibrium simulation is run until the it reaches steady-state condition. The keyword *true* or *false* specify whether the equilibrium condition is computed or not. The key *dt_max* specifies the time step size adopted to reach steady-state condition, which is defined by the *time_tol* key.

The *operation* key requires the key *active*, which can be *true* or *false*. The *dt_max* key defines the time step size of the simulation during the operation stage. Finally the *n_skip* key specifies how many time steps to skip before saving the results. This is useful in simulations where a very small time step size is required, thus avoiding excessively large results files. An example is shown in Listing 2.5.

Listing 2.5: Input file section: *simulation_settings*

```
{
    "simulation_settings": {
       "equilibrium": {
           "active": true,
           "dt_max": 1800.0,
           "time_tol": 0.0001
       },
       "operation": {
           "active": true,
           "dt_max": 1800.0,
           "n_skip": 1
       }
    },
}
```

2.1.5 Section *body_forces*

This section defines the body forces associated to the rock mass. The gravity acceleration is specified under the key *gravity*; the rock density is defined under key *ensity*; and the direction along which the gravity acceleration is aligned is specified under the key *direction* (0 for x, 1 for y and 2 for z). For example, Listing 2.6.

Listing 2.6: Input file section: *body_force*

```
{
    "body_force": {
        "gravity": -9.81,
        "density": 2000,
        "direction": 2
    },
}
```

2.1.6 Section *time_settings*

In the *time_settings* section, the time integration method is defined by chosing the θ value under the key *theta* (0 for fully-implicit, 0.5 for Crank-Nicolson, and 1 for explicit). Next, the key *timeList* specifies the time schedule that defines the loading conditions (see *Section boundary_conditions*). For example,

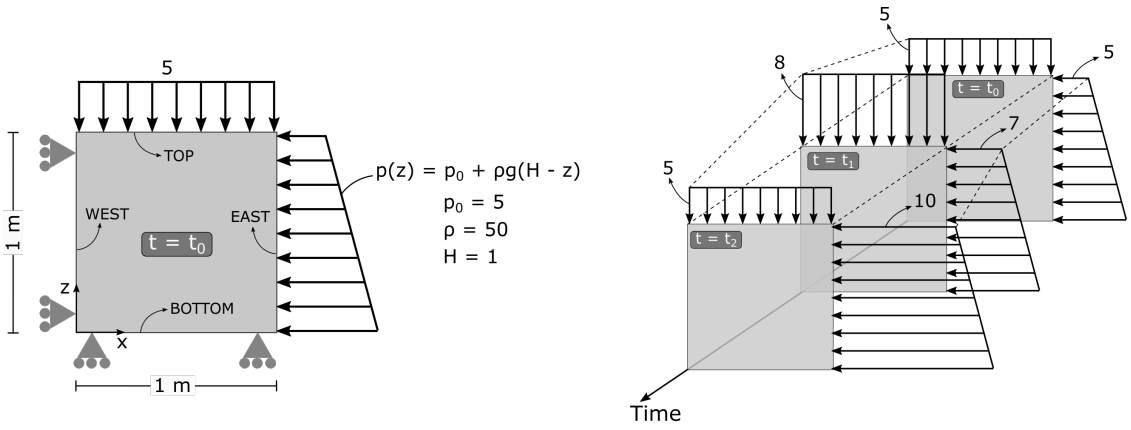
Listing 2.7: Input file section: *time_settings*

```
{
    "time_settings": {
        "theta": 0.0,
        "time_list": [0, 10, 20]
    },
}
```

2.1.7 Section *boundary_conditions*

This section allows for specifying the boundary conditions of the problem. For salt cavern simulations, it is often the case that the pressure inside the cavern varies with time. Additionally, for very tall caverns, there is a significant pressure difference between the top and the bottom of the cavern due to the gas specific weight. The sideburden, althought fixed in time, also varies significantly from top to bottom of the geometry. The section *boundary_conditions* was designed to allow for an easy spefication of such boundary conditions. To exemplify this process, consider the examples illustrated in Fig. 2.1, which shows a 2D view of a block with boundaries names *BOTTOM*, *TOP*, *WEST* and *EAST*. Fig. 2.1-a shows in details the boundary conditions applied at the initial time step t_0. As it can be verified, the *BOTTOM* and *WEST* boundaries are prevented from normal displacement (Dirichlet boundary condition), whereas the *TOP* boundary is subjected to a constant (in space) compressive load, and a z-dependent load is applied to boundary *EAST*. Moreover, Fig. 2.1-b shows that the applied loads actually vary with time.

The keys inside the *boundary_settings* section must be the boundary names. Inside each boundary name, there is a *type* key that can be either *dirichlet* or *neumann*. If *type* is *dirichlet, then the imposed displacement component must be specified under the key *component* (0 for x, 1 for y and 2 for z). Next, the key *values* receives a list of prescribed values for each time level according to the *time_list*, defined in section *time_settings* (**both lists must be the same size**). If *type* is *neumann*, then the keys *direction*, *density*, *reference_position* and *values* are required. The *direction* key defines the direction along which the boundary condition varies spacially; the *density* key specifies how much the load changes in that direction; the *reference_position* key defines the position H where the specified values p_0 are located (see Fig. 2.1-a); and the *values* key receives a list of prescribed loads corresponding to each time of *time_settings*.



(a) Boundary conditions and boundary names.

(b) Time variation of boundary conditions.

Fig. 2.1: Boundary conditions applied to block.

The boundary conditions illustrated in Fig. 2.1 are written in the JSON file as shown below (Listing 2.8). The *BOTTOM* and *WEST* boundaries are of type *dirichlet* with value 0 in the time interval between 0 and 20 s (see *Section time_settings*). The displacement component normal to boundary *BOTTOM* is in the z direction, that is why the key *component* receives the value 2. On the other hand, the normal displacement on boundary *WEST* is aligned to the x direction, thus the value 0 to the key *component*. The boundary *EAST* is subjected to a boundary condition of *type neumann*, and the spatial variation takes place in the z direction (*direction: 2*). The amount of variation ρ is specified as *density: 50* and the reference position H is *reference_position: 1.0*, according to Fig. 2.1-a. According to Fig. 2.1-a, the load imposed on the *TOP* boundary is uniform, so the *density* key should be zero. As a consequence, the value specified in the *direction* and *reference_position* keys and do not matter at all.

Note: The value of gravity g shown in Fig. 2.1-a is specified in *Input file section: body_force*.

Listing 2.8: Input file section: *boundary_conditions*

```
{
    "boundary_conditions": {
        "BOTTOM": {
            "type": "dirichlet",
            "component": 2,
            "values": [0.0, 0.0, 0.0]
        },
        "WEST": {
            "type": "dirichlet",
            "component": 0,
            "values": [0.0, 0.0, 0.0]
        },
        "EAST": {
            "type": "neumann",
            "density": 50.0,
            "direction": 2,
            "reference_position": 1.0,
            "values": [5.0, 7.0, 10.0]
        },
        "TOP": {
```

(continues on next page)

(continued from previous page)

```
            "type": "neumann",
            "density": 0.0,
            "direction": 0,
            "reference_position": 0.0,
            "values": [5.0, 8.0, 5.0]
        }
    }
}
```

2.1.8 Section *constitutive_model*

The SafeInCave simulator allows for very flexible choices of the constitutive model. As an example, we consider the constitutive model illustrated in Fig. 2.2, which is composed of a linear spring element, two Kelvin-Voigt elements, one viscoplastic element, and one dislocation creep element. Each one of these elements comprise its own set of material parameters, as indicated in the figure. Refer to *Constitutive models* for a detailed explanation of each element and the corresponding material properties.

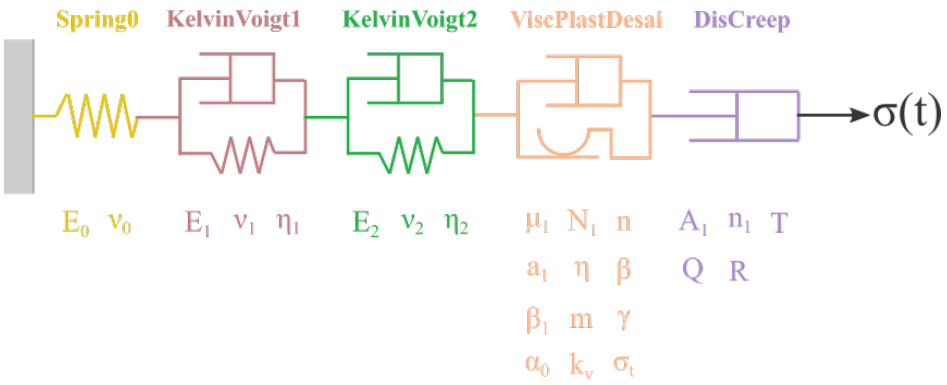


Fig. 2.2: Elements composing the constitutive model.

To be general, let us consider a simple mesh divided in two sub-domains with different material properties. This is illustrated in Fig. 2.3, where elements 0, 1, 4 and 5 belong to Ω_A, while elements 2, 3, 6 and 7 belong to Ω_B.

Note: A 2D grid is considered here only for simplicity. However, the SafeInCave simulator only handles 3D grids composed of tetrahedral elements.

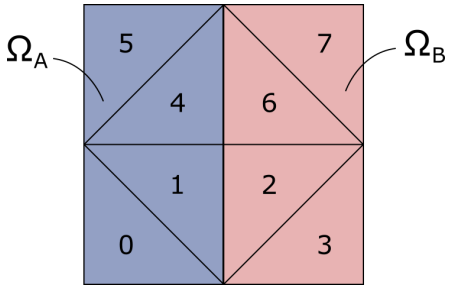


Fig. 2.3: Computational mesh divided in two sub-domains: Ω_A and Ω_B.

The material properties assigned to each sub-domain is presented in Table 2.1. In this example, the values assigned to each material property are merely illustrative and **do not** correspond real physical values.

Table 2.1: Material properties for domains Ω_A and Ω_B.

Property name	Domain Ω_A	Domain Ω_B
E_0	100	250
ν_0	0.3	0.2
E_1	90	75
ν_1	0.15	0.42
η_1	7.0	8.2
E_2	120	165
ν_2	0.24	0.38
η_2	17.0	6.3
μ_1	5.3	2.1
N_1	3.1	2.9
n	3	3
a_1	1.9	2.3
η	0.82	0.97
β	0.99	0.76
β_1	0.38	0.75
m	-0.5	-0.5
γ	0.087	0.095
α_0	0.40	0.27
k_v	0.0	0.6
σ_t	5.0	4.2
A_1	1.9	2.3
n_1	3.1	4.2
T	298	298
Q	51600	51600
R	8.32	8.32

The *constitutive_model* section requires three mandatory keys: *Elastic*, *Viscoelastic* and *Inelastic*. A spring can be added to the *Elastic* key as shown in Listing 2.9. The name *Spring0* is an arbitrary name given to the spring; the *type* key must be *Spring*; the key *active* can be *true* or *false* depending on whether the user wants to include it or not to the constitutive model; finally, the key *parameters* contains the lists of the material parameters associated to the spring (i.e. Young's modulus, E, and Poisson's ratio, ν). The size of these lists must be the same as the number of grid elements (in this case, 8 elements, as shown in Fig. 2.3). Therefore, the values in these lists represent the material properties of each element of the grid.

Important: A constitutive model **must** include at least one spring. In other words, at least one spring must be **active**.

A Kelvin-Voigt element is a parallel arrangement between a linear spring and a linear dashpot. This type of element is added under the key *Viscoelastic*. In the example shown in Listing 2.9, two Kelvin-Voigt elements are added, namely, *KelvinVoigt1* and *KelvinVoigt2*. The key *type* must be *KelvinVoigt*. The material parameters associated to the Kelvin-Voigt element are the Poisson's ratio (ν) and Young's modulus (E) of the spring, and the viscosity (η) of the dashpot.

Note: A Kelvin-Voigt element with a nonlinear dashpot, if implemented, should be also added under the *Viscoelastic* key.

The viscoplastic and dislocation creep elements in Fig. 2.2 must be included under the *Inelastic* key. In this exam-

ple, the arbitrary names given to the viscoplastic and dislocation creep elements are *ViscPlastDesai* and *DisCreep*, respectively. The viscoplastic element must be of type *ViscoplasticDesai* and dislocation creep element must be of type *DislocationCreep*.

Listing 2.9: Input file section: *constitutive_model*

```
{
    "constitutive_model": {
        "Elastic": {
            "Spring0": {
                "type": "Spring",
                "active": true,
                "parameters": {
                    "E": [100, 100, 250, 250, 100, 100, 250, 250],
                    "nu": [0.3, 0.3, 0.2, 0.2, 0.3, 0.3, 0.2, 0.2]
                }
            }
        },
        "Viscoelastic": {
            "KelvinVoigt1": {
                "type": "KelvinVoigt",
                "active": true,
                "parameters": {
                    "E": [90.0, 90.0, 75.0, 75.0, 90.0, 90.0, 75.0, 75.0],
                    "nu": [0.15, 0.15, 0.42, 0.42, 0.15, 0.15, 0.42, 0.42],
                    "eta": [7.0, 7.0, 8.2, 8.2, 7.0, 7.0, 8.2, 8.2]
                }
            },
            "KelvinVoigt2": {
                "type": "KelvinVoigt",
                "active": true,
                "parameters": {
                    "E": [120.0, 120.0, 165.0, 165.0, 120.0, 120.0, 165.0, 165.0],
                    "nu": [0.24, 0.24, 0.38, 0.38, 0.24, 0.24, 0.38, 0.38],
                    "eta": [17.0, 17.0, 6.3, 6.3, 17.0, 17.0, 6.3, 6.3]
                }
            }
        },
        "Inelastic": {
            "ViscPlastDesai": {
                "type": "ViscoplasticDesai",
                "active": true,
                "parameters": {
                    "mu_1": [5.3, 5.3, 2.1, 2.1, 5.3, 5.3, 2.1, 2.1],
                    "N_1": [3.1, 3.1, 2.9, 2.9, 3.1, 3.1, 2.9, 2.9],
                    "n": [3, 3, 3, 3, 3, 3, 3, 3],
                    "a_1": [1.9, 1.9, 2.3, 2.3, 1.9, 1.9, 2.3, 2.3],
                    "eta": [0.82, 0.82, 0.97, 0.97, 0.82, 0.82, 0.97, 0.97],
                    "beta_1": [0.99, 0.99, 0.76, 0.76, 0.99, 0.99, 0.76, 0.76],
                    "beta": [0.38, 0.38, 0.75, 0.75, 0.38, 0.38, 0.75, 0.75],
                    "m": [-0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5],
                    "gamma": [0.087, 0.087, 0.095, 0.095, 0.087, 0.087, 0.095, 0.095],
                    "alpha_0": [0.40, 0.40, 0.27, 0.27, 0.40, 0.40, 0.27, 0.27],
                    "k_v": [0.0, 0.0, 0.6, 0.6, 0.0, 0.0, 0.6, 0.6],
```

(continues on next page)

(continued from previous page)

```
                    "sigma_t": [5.0, 5.0, 4.2, 4.2, 5.0, 5.0, 4.2, 4.2]
                }
            },
            "DisCreep": {
                "type": "DislocationCreep",
                "active": true,
                "parameters": {
                    "A": [1.9, 1.9, 2.3, 2.3, 1.9, 1.9, 2.3, 2.3],
                    "n": [3.1, 3.1, 4.2, 4.2, 3.1, 3.1, 4.2, 4.2],
                    "T": [298, 298, 298,298, 298, 298, 298,298],
                    "Q": [51600, 51600, 51600, 51600, 51600, 51600, 51600, 51600],
                    "R": [832, 832, 832, 832, 832, 832, 832, 832]
                }
            }
        }
    }
}
```

The elements available for composing the constitutive model are summarized in Table 2.2, where the correspoding material parameters are also shown. The parameters, as discussed above, must be informed as a list of values associated to each grid element. Currently, the linear elastic spring, the viscoelastic Kelvin-Voigt element, the viscoplastic model of Desai (1987), and the dislocation creep element are implemented in the SafeInCave simulator.

Table 2.2: Available elements for the constitutive model.

Category	Type	Material parameters
Elastic	Spring	E, nu
Viscoelastic	KelvinVoigt	E, nu, eta
Inelastic	DislocationCreep	A, n, T, Q, R
Inelastic	ViscoplasticDesai	mu_1, N_1, n, a_1, eta, beta_1, beta, m, gamma, alpha_0, k_v, sigma_t

2.1.9 Full input file

To conclude this section, the complete input file should look like as in Listing 2.10

Listing 2.10: Complete input file

```
{
    "grid": {
        "path": "../../grids/cube_0",
        "name": "geom"
    },
    "output": {
        "path": "output/case_name"
     },
    "solver_settings": {
        "type": "KrylovSolver",
        "method": "cg",
        "preconditioner": "petsc_amg",
        "relative_tolerance": 1e-12
     },
```

(continues on next page)

(continued from previous page)

```
    "simulation_settings": {
        "equilibrium": {
            "active": true,
            "dt_max": 1800.0,
            "time_tol": 0.0001
        },
        "operation": {
            "active": true,
            "dt_max": 1800.0,
            "n_skip": 1
        }
    },
    "body_force": {
        "gravity": -9.81,
        "density": 2000,
        "direction": 2
    },
    "time_settings": {
        "theta": 0.0,
        "time_list": [0, 10, 20]
    },
    "boundary_conditions": {
        "BOTTOM": {
            "type": "dirichlet",
            "component": 2,
            "values": [0.0, 0.0, 0.0]
        },
        "WEST": {
            "type": "dirichlet",
            "component": 0,
            "values": [0.0, 0.0, 0.0]
        },
        "EAST": {
            "type": "neumann",
            "density": 50.0,
            "direction": 2,
            "reference_position": 1.0,
            "values": [5.0, 7.0, 10.0]
        },
        "TOP": {
            "type": "neumann",
            "density": 0.0,
            "direction": 0,
            "reference_position": 0.0,
            "values": [5.0, 8.0, 5.0]
        }
    },
    "constitutive_model": {
        "Elastic": {
            "Spring0": {
                "type": "Spring",
                "active": true,
```

(continues on next page)

(continued from previous page)

```
                "parameters": {
                    "E": [100, 100, 250, 250, 100, 100, 250, 250],
                    "nu": [0.3, 0.3, 0.2, 0.2, 0.3, 0.3, 0.2, 0.2]
                }
            }
        },
        "Viscoelastic": {
            "KelvinVoigt1": {
                "type": "KelvinVoigt",
                "active": true,
                "parameters": {
                    "E": [90.0, 90.0, 75.0, 75.0, 90.0, 90.0, 75.0, 75.0],
                    "nu": [0.15, 0.15, 0.42, 0.42, 0.15, 0.15, 0.42, 0.42],
                    "eta": [7.0, 7.0, 8.2, 8.2, 7.0, 7.0, 8.2, 8.2]
                }
            },
            "KelvinVoigt2": {
                "type": "KelvinVoigt",
                "active": true,
                "parameters": {
                    "E": [120.0, 120.0, 165.0, 165.0, 120.0, 120.0, 165.0, 165.0],
                    "nu": [0.24, 0.24, 0.38, 0.38, 0.24, 0.24, 0.38, 0.38],
                    "eta": [17.0, 17.0, 6.3, 6.3, 17.0, 17.0, 6.3, 6.3]
                }
            }
        },
        "Inelastic": {
            "ViscPlastDesai": {
                "type": "ViscoplasticDesai",
                "active": true,
                "parameters": {
                    "mu_1": [5.3, 5.3, 2.1, 2.1, 5.3, 5.3, 2.1, 2.1],
                    "N_1": [3.1, 3.1, 2.9, 2.9, 3.1, 3.1, 2.9, 2.9],
                    "n": [3, 3, 3, 3, 3, 3, 3, 3],
                    "a_1": [1.9, 1.9, 2.3, 2.3, 1.9, 1.9, 2.3, 2.3],
                    "eta": [0.82, 0.82, 0.97, 0.97, 0.82, 0.82, 0.97, 0.97],
                    "beta_1": [0.99, 0.99, 0.76, 0.76, 0.99, 0.99, 0.76, 0.76],
                    "beta": [0.38, 0.38, 0.75, 0.75, 0.38, 0.38, 0.75, 0.75],
                    "m": [-0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5],
                    "gamma": [0.087, 0.087, 0.095, 0.095, 0.087, 0.087, 0.095, 0.095],
                    "alpha_0": [0.40, 0.40, 0.27, 0.27, 0.40, 0.40, 0.27, 0.27],
                    "k_v": [0.0, 0.0, 0.6, 0.6, 0.0, 0.0, 0.6, 0.6],
                    "sigma_t": [5.0, 5.0, 4.2, 4.2, 5.0, 5.0, 4.2, 4.2]
                }
            },
            "DisCreep": {
                "type": "DislocationCreep",
                "active": true,
                "parameters": {
                    "A": [1.9, 1.9, 2.3, 2.3, 1.9, 1.9, 2.3, 2.3],
                    "n": [3.1, 3.1, 4.2, 4.2, 3.1, 3.1, 4.2, 4.2],
                    "T": [298, 298, 298,298, 298, 298, 298,298],
```

(continues on next page)

(continued from previous page)

```
                    "Q": [51600, 51600, 51600, 51600, 51600, 51600, 51600, 51600],
                    "R": [832, 832, 832, 832, 832, 832, 832, 832]
                }
            }
        }
    }
}
```

2.2 Input file assistant

One of the main difficults to manually write the input file is to assign the material properties for meshes that depend on many elements (unlike the hypothetical example shown in Listing 2.10, where only 8 elements compose the mesh). For example, if we decide to replace the mesh in section *grid* by another mesh with different number of elements, the lists of material properties must be updated accordingly. Moreover, for complex loading schedules (sections *time_settings* and *boundary_conditions*), writing the time list and boundary conditions can easily become a very tedious task. Finally, manually writing the input file is always prone to errors.

Therefore, although the input file can be manually built, most of the time it will be more convenient to write the input file in an automatic manner. This can be achieved with class **InputFileAssistant** (check *InputFileAssistant module*), as presented below.

Import modules.

```
import os
import sys
import numpy as np
sys.path.append(os.path.join("..", "..", "safeincave"))
from Grid import GridHandlerGMSH
from InputFileAssistant import BuildInputFile
```

Define some useful units for convenience.

```
hour = 60*60
day = 24*hour
MPa = 1e6
```

Initialize the input file assistant object.

```
bif = BuildInputFile()
```

Create *input_grid* section.

```
path_to_grid = os.path.join("..", "..", "grids", "cube_2regions")
bif.section_input_grid(path_to_grid, "geom")
```

Create *output* section.

```
bif.section_output(os.path.join("output", "case_1"))
```

Create *solver_settings* section.

```
solver_settings = {
    "type": "KrylovSolver",
    "method": "cg",
    "preconditioner": "petsc_amg",
    "relative_tolerance": 1e-12,
}
bif.section_solver(solver_settings)
```

Create *simulation_settings* section.

```
bif.section_simulation(
    simulation_settings = {
        "equilibrium": {
            "active": True,
            "dt_max": 0.5*hour,
            "time_tol": 1e-4
        },
        "operation": {
            "active": True,
            "dt_max": 0.5*hour,
            "n_skip": 1
        }
    }
)
```

Create *body_forces* section.

```
salt_density = 2000
bif.section_body_forces(value=salt_density, direction=2)
```

Create *time_settings* section.

```
time_list = [0*hour,  2*hour,  10*hour, 12*hour, 14*hour, 16*hour, 20*hour, 22*hour,␣
↪24*hour]
bif.section_time(time_list, theta=0.0)
```

Create *boundary_conditions* section.

```
bif.section_boundary_conditions()

# Add Dirichlet boundary conditions
bif.add_boundary_condition(
    boundary_name = "WEST",
    bc_data = {
        "type": "dirichlet",
        "component": 0,
        "values": list(np.zeros(len(time_list)))
    }
)
bif.add_boundary_condition(
    boundary_name = "SOUTH",
    bc_data = {
        "type": "dirichlet",
```

(continues on next page)

(continued from previous page)

```
        "component": 1,
        "values": list(np.zeros(len(time_list)))
    }
)
bif.add_boundary_condition(
    boundary_name = "BOTTOM",
        bc_data = {
        "type": "dirichlet",
        "component": 2,
        "values": list(np.zeros(len(time_list)))
    }
)

# Add Neumann boundary condition
bif.add_boundary_condition(
    boundary_name = "EAST",
    bc_data = {
        "type": "neumann",
        "direction": 2,
        "density": 0.0,
        "reference_position": 1.0,
        "values": [5*MPa, 5*MPa, 5*MPa, 5*MPa, 5*MPa, 5*MPa, 5*MPa, 5*MPa, 5*MPa]
    }
)
bif.add_boundary_condition(
    boundary_name = "NORTH",
    bc_data = {
        "type": "neumann",
        "direction": 2,
        "density": 0.0,
        "reference_position": 1.0,
        "values": [5*MPa, 5*MPa, 5*MPa, 5*MPa, 5*MPa, 5*MPa, 5*MPa, 5*MPa, 5*MPa]
    }
)
bif.add_boundary_condition(
    boundary_name = "TOP",
    bc_data = {
        "type": "neumann",
        "direction": 2,
        "density": 0.0,
        "reference_position": 1.0,
        "values": [6*MPa, 10*MPa, 10*MPa, 6*MPa, 6*MPa, 12*MPa, 12*MPa, 6*MPa, 6*MPa]
    }
)
```

Create *constitutive_model* section.

```
bif.section_constitutive_model()

# Add elastic properties
bif.add_elastic_element(
    element_name = "Spring_0",
```

(continues on next page)

(continued from previous page)

```
    element_parameters = {
        "type": "Spring",
        "active": True,
        "parameters": {
            "E":  list(102e9*np.ones(bif.n_elems)),
            "nu": list(0.3*np.ones(bif.n_elems))
        }
    }
)

# Add viscoelastic properties
bif.add_viscoelastic_element(
    element_name = "KelvinVoigt_0",
    element_parameters = {
        "type": "KelvinVoigt",
        "active": True,
        "parameters": {
            "E":   list(10e9*np.ones(bif.n_elems)),
            "nu":  list(0.32*np.ones(bif.n_elems)),
            "eta": list(105e11*np.ones(bif.n_elems))
        }
    }
)

# Add viscoplastic parameters
bif.add_inelastic_element(
    element_name = "desai",
    element_parameters = {
        "type": "ViscoplasticDesai",
        "active": False,
        "parameters": {
            "mu_1":     list(5.3665857009859815e-11*np.ones(bif.n_elems)),
            "N_1":      list(3.1*np.ones(bif.n_elems)),
            "n":        list(3.0*np.ones(bif.n_elems)),
            "a_1":      list(1.965018496922832e-05*np.ones(bif.n_elems)),
            "eta":      list(0.8275682807874163*np.ones(bif.n_elems)),
            "beta_1":   list(0.0048*np.ones(bif.n_elems)),
            "beta":     list(0.995*np.ones(bif.n_elems)),
            "m":        list(-0.5*np.ones(bif.n_elems)),
            "gamma":    list(0.095*np.ones(bif.n_elems)),
            "alpha_0":  list(0.004071571404980058 6*np.ones(bif.n_elems)),
            "k_v":      list(0.0*np.ones(bif.n_elems)),
            "sigma_t":  list(5.0*np.ones(bif.n_elems))
        }
    }
)

# Add dislocation creep parameters
bif.add_inelastic_element(
    element_name = "creep",
    element_parameters = {
        "type": "DislocationCreep",
```

(continues on next page)

(continued from previous page)

```
        "active": True,
        "parameters": {
            "A":    list(1.9e-20*np.ones(bif.n_elems)),
            "n":    list(3.0*np.ones(bif.n_elems)),
            "T":    list(298*np.ones(bif.n_elems)),
            "Q":    list(51600*np.ones(bif.n_elems)),
            "R":    list(8.32*np.ones(bif.n_elems))
        }
    }
)
```

Save input_file.json.

```
bif.save_input_file("input_file.json")
```

Defining a simulation is just a matter of chosing a mesh for the problem (with the desired geometry, boundary and region names) and appropriately writing the input file.

CHAPTER

THREE

TUTORIALS

This chapter presents two tutorials that illustrate different capabilities of the SafeInCave simulator. The first tutorial handles a heterogeneous medium and shows how to assign different material properties for each region of the domain. The second tutorial addresses how to simulate a salt cavern with realistic boundary conditions.

3.1 Preliminary

Before proceeding with the tutorials, the notation used throughout this chapter is introduced below

Notation:

- Position vector: $\mathbf{r} = \begin{bmatrix} x & y & z \end{bmatrix}^T$.
- Displacement vector: $\mathbf{u} = \begin{bmatrix} u & v & w \end{bmatrix}^T$.
- Normal vector: $\mathbf{n} = \begin{bmatrix} n_x & n_y & n_z \end{bmatrix}^T$.
- Stress tensor: $\boldsymbol{\sigma} = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{xy} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{xz} & \sigma_{yz} & \sigma_{zz} \end{bmatrix}^T$ ..

3.2 Tutorial 1

In this example, we simulate a cubic material subjected to a triaxial stress condition. The domain Ω, its dimensions and boundary conditions are shown in Fig. 3.1-a. Fig. 3.1-b presents the corresponding names of the geometry boundaries and regions. Finally, Fig. 3.1-c shows the tetrahedral mesh used for this problem.

The domain Ω is bounded by a closed surface Γ. For mathematical convenience, let us split Γ into non-overlapping subsets comprising each boundary shown in Figure 3.1-b, such that

$$
\begin{aligned}
\Gamma_{\text{west}} &= \{\mathbf{r} \in \Omega | x = 0\} \\
\Gamma_{\text{east}} &= \{\mathbf{r} \in \Omega | x = 1 \text{ m}\} \\
\Gamma_{\text{south}} &= \{\mathbf{r} \in \Omega | y = 0\} \\
\Gamma_{\text{north}} &= \{\mathbf{r} \in \Omega | y = 1 \text{ m}\} \\
\Gamma_{\text{bottom}} &= \{\mathbf{r} \in \Omega | z = 0\} \\
\Gamma_{\text{top}} &= \{\mathbf{r} \in \Omega | z = 1 \text{ m}\},
\end{aligned}
$$

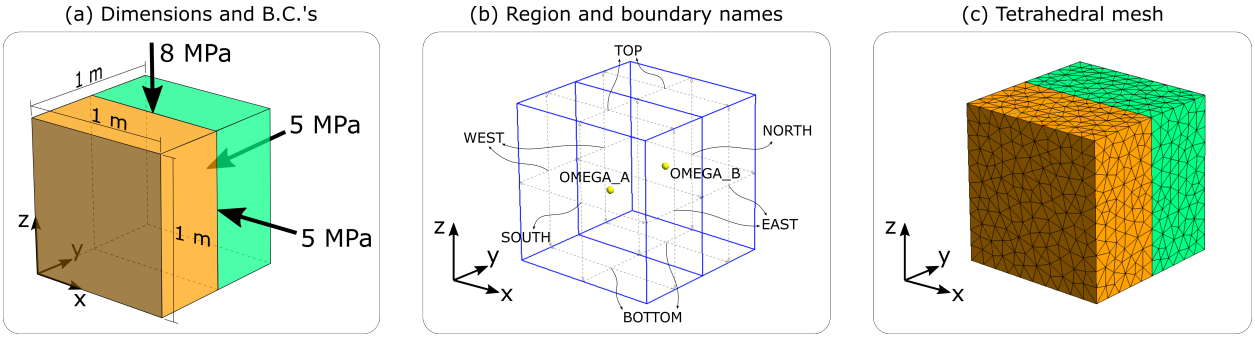


Fig. 3.1: Boundary names, region (subdomain) names and computational mesh.

and $\Gamma = \Gamma_{\text{west}} \cup \Gamma_{\text{east}} \cup \Gamma_{\text{south}} \cup \Gamma_{\text{north}} \cup \Gamma_{\text{bottom}} \cup \Gamma_{\text{top}}$. Finally, the boundary conditions applied for this problem can be written as,

$$
\begin{aligned}
u(\mathbf{r},t) &= 0, && \forall \quad \mathbf{r} \in \Gamma_{\text{west}}, \\
v(\mathbf{r},t) &= 0, && \forall \quad \mathbf{r} \in \Gamma_{\text{south}}, \\
w(\mathbf{r},t) &= 0, && \forall \quad \mathbf{r} \in \Gamma_{\text{bottom}}, \\
\boldsymbol{\sigma}(\mathbf{r},t) \cdot \mathbf{n} &= 5 \text{ MPa}, && \forall \quad \mathbf{r} \in \Gamma_{\text{east}}, \\
\boldsymbol{\sigma}(\mathbf{r},t) \cdot \mathbf{n} &= 5 \text{ MPa}, && \forall \quad \mathbf{r} \in \Gamma_{\text{north}}, \\
\boldsymbol{\sigma}(\mathbf{r},t) \cdot \mathbf{n} &= 8 \text{ MPa}, && \forall \quad \mathbf{r} \in \Gamma_{\text{top}}.
\end{aligned}
$$

The material behavior represented by a constitutive model composed of a linear spring and a Kelvin-Voigt element (i.e. viscoelastic model), as shown in Fig. 3.2.

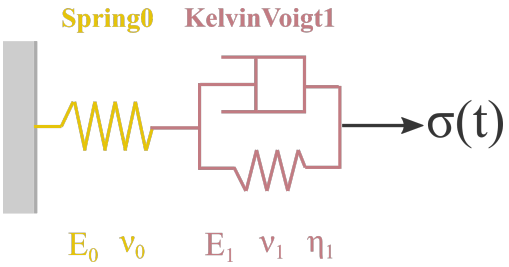


Fig. 3.2: Viscoelastic constitutive model considered for Tutorial 1.

Figure 3.1-b also shows that the cubic sample is divided in two regions (subdomains), called Ω_A and Ω_B. Different material properties are assigned to these two regions, and they are summarized in Table 3.1.

Table 3.1: Material properties for domains Ω_A and Ω_B.

Property name	Domain Ω_A	Domain Ω_B
E_0 (GPa)	8	10
ν_0 (-)	0.2	0.3
E_1 (GPa)	8	5
ν_1 (-)	0.35	0.28
η_1 (Pa.s)	105×10^{11}	38×10^{11}

The next subsection describes the creation of the input file for this particular problem.

3.2.1 Build input file

Listing 3.1 imports the necessary modules and defines some useful units to be used throughout this example. Notice the *dolfin* package is imported in line 6. The reason for this is because we want to loop over the mesh elements to identify to which region (Ω_A or Ω_B) each element belongs to. For this, we some tools from *dolfin* package.

Listing 3.1: Import modules.

```
1  import os
2  import sys
3  import numpy as np
4  sys.path.append(os.path.join("..", "..", "safeincave"))
5  from InputFileAssistant import BuildInputFile
6  import dolfin as do
7
8  # Useful units
9  hour = 60*60
10 day = 24*hour
11 MPa = 1e6
12 GPa = 1e9
```

Initialize input file object.

```
1  ifa = BuildInputFile()
```

Create *input_grid* section.

```
1  path_to_grid = os.path.join("..", "..", "grids", "cube")
2  ifa.section_input_grid(path_to_grid, "geom")
```

Create *output* section.

```
1  ifa.section_output(os.path.join("output", "case_0"))
```

Create *solver_settings* section and choose conjugate gradient method with algebraic multi-grid for solving the linear system.

```
1  solver_settings = {
2      "type": "KrylovSolver",
3      "method": "cg",
4      "preconditioner": "petsc_amg",
5      "relative_tolerance": 1e-12,
6  }
7  ifa.section_solver(solver_settings)
```

Create *simulation_settings* section. Note in line 3 that we set the *equilibrium* stage to **False**. Since the external loads applied to the cubic sample are constant in time and the constitutive model is viscoelastic, running only the *equilibrium* stage or only the *operational* stage will produce the same result (provided that the time step sizes are the same in these two stages). Setting both stages to **True**, however, would produce zero results for the *operational* stage.

Tip: The user is encouraged to play with the *equilibrium* and *operational* stages and checking the results in */output/case_0/equilibrium* and */output/case_0/operational*.

```
ifa.section_simulation(simulation_settings = {
                               "equilibrium": {
                                   "active": False,
                                   "dt_max": 0.5*hour,
                                   "time_tol": 1e-4
                               },
                               "operation": {
                                   "active": True,
                                   "dt_max": 0.005*hour,
                                   "n_skip": 1
                               }
                           })
```

Create *body_forces* section.

```
salt_density = 2000
ifa.section_body_forces(value=salt_density, direction=2)
```

Create *time_settings* section. The transient simulation is set to run from $t = 0$ to $t = 1.0$ hour, and the fully-implicit method is chosen for time integration.

```
time_list = [0*hour,  1*hour]
ifa.section_time(time_list, theta=0.0)
```

Create *boundary_conditions* section. Boundaries *WEST*, *SOUTH* and *BOTTOM* are prevented from normal displacement (Dirichlet boundary condition). A normal compressive load is applied to boundaries *EAST*, *NORTH* and *TOP*, and the corresponding loading values are respectively shown in lines 37, 47 and 57.

```
ifa.section_boundary_conditions()

# Add Dirichlet boundary conditions
ifa.add_boundary_condition(
    boundary_name = "WEST",
    bc_data = {
        "type": "dirichlet",
        "component": 0,
        "values": list(np.zeros(len(time_list)))
    }
)
ifa.add_boundary_condition(
    boundary_name = "SOUTH",
    bc_data = {
        "type": "dirichlet",
        "component": 1,
        "values": list(np.zeros(len(time_list)))
    }
)
ifa.add_boundary_condition(
    boundary_name = "BOTTOM",
    bc_data = {
        "type": "dirichlet",
        "component": 2,
        "values": list(np.zeros(len(time_list)))
```

(continues on next page)

(continued from previous page)

```
    }
)

# Add Neumann boundary condition
ifa.add_boundary_condition(
    boundary_name = "EAST",
    bc_data = {
        "type": "neumann",
        "direction": 2,
        "density": 0,
        "reference_position": 1.0,
        "values": [5*MPa, 5*MPa]
    }
)
ifa.add_boundary_condition(
    boundary_name = "NORTH",
    bc_data = {
        "type": "neumann",
        "direction": 2,
        "density": 0,
        "reference_position": 1.0,
        "values": [5*MPa, 5*MPa]
    }
)
ifa.add_boundary_condition(
    boundary_name = "TOP",
    bc_data = {
        "type": "neumann",
        "direction": 2,
        "density": 0.0,
        "reference_position": 1.0,
        "values": [8*MPa, 8*MPa]
    }
)
```

Before creating the *constitutive_model* section, we first mark the element of the grid that belong to regions Ω_A and Ω_B. The first step is to check the tags (integers) used by FEniCS to identify these two subdomains. This can be achieved the *get_subdomain_tag* function of the *grid* object belonging to object *ifa*. That is,

Listing 3.2: Subdomain tags.

```
>>> region_marker_A = ifa.grid.get_subdomain_tags("OMEGA_A")
>>> print(region_marker_A)
1
>>> region_marker_B = ifa.grid.get_subdomain_tags("OMEGA_B")
>>> print(region_marker_B)
2
```

As shown in Listing 3.2, the tags corresponding to subdomains Ω_A and Ω_B are 1 and 2, respectively. This information is used in Listing 3.3 to store the element indices belonging to regions Ω_A and Ω_B in lists *index_A* and *index_B*, respectively. In line 6, the attribute *subdomains* is a dolfin *MeshFunction* object that retrieves the subdomain tag associated to element *cell*. This can be compared to the corresponding tags of each region to decide whether the element index is stored in *index_A* or *index_B*.

Listing 3.3: Identifying element regions.

```
index_A = []
index_B = []

# Sweep over the grid regions and elements
for cell in do.cells(ifa.grid.mesh):
    region_marker = ifa.grid.subdomains[cell]
    if region_marker == ifa.grid.get_subdomain_tags("OMEGA_A"):
        index_A.append(cell.index())
    elif region_marker == ifa.grid.get_subdomain_tags("OMEGA_B"):
        index_B.append(cell.index())
    else:
        raise Exception("Subdomain tag not valid. Check your mesh file.")
```

Now that we have identified to which region each element belongs to, we can create the *constitutive_model* section with appropriate lists of material properties.

```
ifa.section_constitutive_model()
```

As summarized in Table 3.1, the Young's modulus of the linear spring for regions Ω_A and Ω_B are 8 GPa and 10 GPa, respectively. These two properties are assigned in lines 3 and 4 of Listing 3.4. Notice how *index_A* and *index_B* are used as indices of numpy array *E*, created in line 2. A similar procedure is done for assigning the Poisson's ratios in lines 6, 7 and 8.

Listing 3.4: Assign linear spring to constitutive model.

```
# Add elastic properties
E = np.zeros(ifa.n_elems)
E[index_A] = 8*GPa
E[index_B] = 10*GPa

nu = np.zeros(ifa.n_elems)
nu[index_A] = 0.2
nu[index_B] = 0.3

ifa.add_elastic_element(
    element_name = "Spring0",
    element_parameters = {
        "type": "Spring",
        "active": True,
        "parameters": {
            "E": list(E),
            "nu": list(nu)
        }
    }
)
```

The viscoelastic properties (E_1, ν_1 and η_1) are assigned in the same manner in lines 2, 3, 5, 6, 9 and 10 of Listing 3.5.

Listing 3.5: Assign viscoelastic properties.

```
# Add viscoelastic properties
```

(continues on next page)

(continued from previous page)

```
E[index_A] = 8*GPa
E[index_B] = 5*GPa

nu[index_A] = 0.35
nu[index_B] = 0.28

eta = np.zeros(ifa.n_elems)
eta[index_A] = 105e11
eta[index_B] = 38e11

# Add viscoelastic properties
ifa.add_viscoelastic_element(
    element_name = "KelvinVoigt1",
    element_parameters = {
        "type": "KelvinVoigt",
        "active": True,
        "parameters": {
            "E":    list(E),
            "nu":   list(nu),
            "eta":  list(eta)
        }
    }
)
```

Finally, the input_file.json is saved in the current directory.

```
ifa.save_input_file("input_file.json")
```

To run this example, execute the *main.py* file in *examples/tutorial_1* folder. That is,

```
user@computer:~/safeincave$ cd examples/tutorial_1
user@computer:~/safeincave/examples/tutorial_1$ python main.py
```

3.2.2 Visualize results

The results for equilibrium and operational stages are respectively stored in *output/case_0/equilibrium* and *output/case_0/operational* folders. Although these results can be readily visualized in Paraview, the code below shows how to plot the vertical displacements on boundary *TOP* over time using Python.

Listing 3.6 imports the necessary modules. Notice the function *read_vector_from_points* is imported from *ResultsHandler*, which is responsible for reading the vtk files, extracting the vector field defined on all nodes for all time steps, and saving them in pandas datasets. This facilitates the manipulation of results.

Listing 3.6: Results visualization for Tutorial 1.

```
import os
import sys
sys.path.append(os.path.join("..", "..", "safeincave"))
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from ResultsHandler import read_vector_from_points
```

The next code-block reads the displacement results from folder *operation*. Variable *df_coord* stores the coordinates of all grid nodes, whereas *u*, *v* and *w* stores the displacement components for all time steps of the simulation.

```
pvd_path = os.path.join("output", "case_0", "operation", "vtk", "displacement")
pvd_file = "displacement.pvd"
df_coord, u, v, w = read_vector_from_points(pvd_path, pvd_file)
```

Points A, B, C and D are shown in Fig. 3.3-a. To access the displacement at these points, it is necessary to identify their corresponding indexes. This is performed in the code-block below.

```
point_A = df_coord[(df_coord["z"]==1) & (df_coord["x"]==0) & (df_coord["y"]==0)].index[0]
point_B = df_coord[(df_coord["z"]==1) & (df_coord["x"]==0) & (df_coord["y"]==1)].index[0]
point_C = df_coord[(df_coord["z"]==1) & (df_coord["x"]==1) & (df_coord["y"]==1)].index[0]
point_D = df_coord[(df_coord["z"]==1) & (df_coord["x"]==1) & (df_coord["y"]==0)].index[0]
print(point_A, point_B, point_C, point_D)
```

Once the indices of the points of interest are identified, they can be used to access the vertical displacement *w* at these points. The list of time steps can also be retrieved from dataset *w*, as performed in line 5 of the code-block below.

```
w_A = w.iloc[point_A].values[1:]
w_B = w.iloc[point_B].values[1:]
w_C = w.iloc[point_C].values[1:]
w_D = w.iloc[point_D].values[1:]
t = w.iloc[point_A].index.values[1:]
```

Finally, plot the results using Matplotlib.

```
# Plot pressure schedule
fig, ax = plt.subplots(1, 1, figsize=(5, 3.5))
fig.subplots_adjust(
    top=0.970, bottom=0.135, left=0.140, right=0.980, hspace=0.35, wspace=0.225
)

ax.plot(t/60, w_A*1000, ".-", color="#377eb8", label="Point A")
ax.plot(t/60, w_B*1000, ".-", color="#ff7f00", label="Point B")
ax.plot(t/60, w_C*1000, ".-", color="#4daf4a", label="Point C")
ax.plot(t/60, w_D*1000, ".-", color="#f781bf", label="Point D")
ax.set_xlabel("Time (minutes)", size=12, fontname="serif")
ax.set_ylabel("Displacement (mm)", size=12, fontname="serif")
ax.grid(True)
ax.legend(loc=0, shadow=True, fancybox=True)

plt.show()
```

The results presented in Fig. 3.3-b reveal an interesting behavior. As shown in Table 3.1, the Young's modulus of the linear spring (E_0) for Ω_A is smaller than for Ω_B. In other words, Ω_B is instantaneously stiffer than Ω_A. For this reason, immediately after the load is applied, points A and D, which belong to Ω_A, present larger displacements than the other two points on Ω_B. However, the Kelvin-Voigt spring is stiffer for Ω_A than for Ω_B. Therefore, as time passes by, the Kelvin-Voigt element slowly starts to the deform and the displacements at points B and C (Ω_B) take over the other two points on Ω_A.

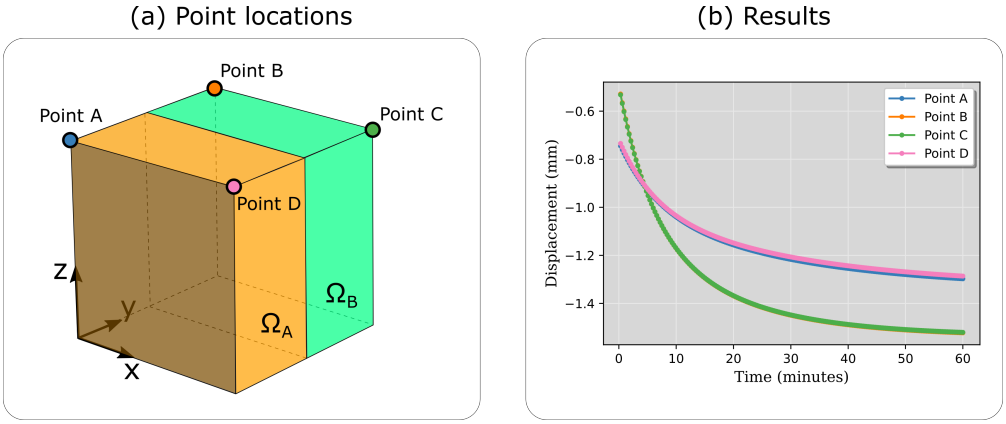


Fig. 3.3: Position of points of interest (a) and the corresponding vertical displacements over time (b).

3.3 Tutorial 2

This tutorial intends to setup a simulation for hydrogen storage in a salt cavern. The domain Ω and its dimensions are depicted in Fig. 3.4-a. This figure also shows the tetrahedral mesh employed, with mesh refinement close to the cavern walls. The names of each boundary for correctly applying the boundary conditions are shown in Figure 3.4-b.

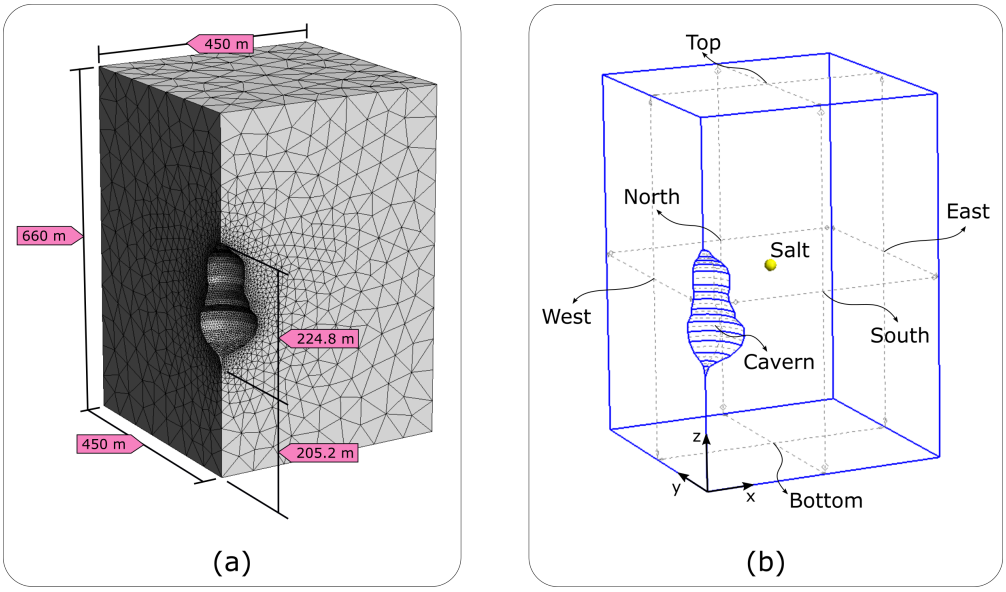


Fig. 3.4: Geometry dimensions and mesh (a); boundary and region names (b).

As illustrated in Fig. 3.5-a, a constant overburden of 10 MPa is applied to the geometry, whereas the a sideburden increases with depth according to salt density. This figure also presents a schematic representation of the gas pressure imposed on the cavern walls, showing the minimum (7 MPa) and maximum (10 MPa) gas pressure at the cavern roof.

The domain Ω is bounded by a closed surface Γ. For mathematical convenience, let us split Γ into non-overlapping subsets comprising each boundary shown in Figure 3.4-b, such that

$$\Gamma = \Gamma_{\text{bottom}} \cup \Gamma_{\text{top}} \cup \Gamma_{\text{south}} \cup \Gamma_{\text{north}} \cup \Gamma_{\text{west}} \cup \Gamma_{\text{east}} \cup \Gamma_{\text{cavern}}.$$

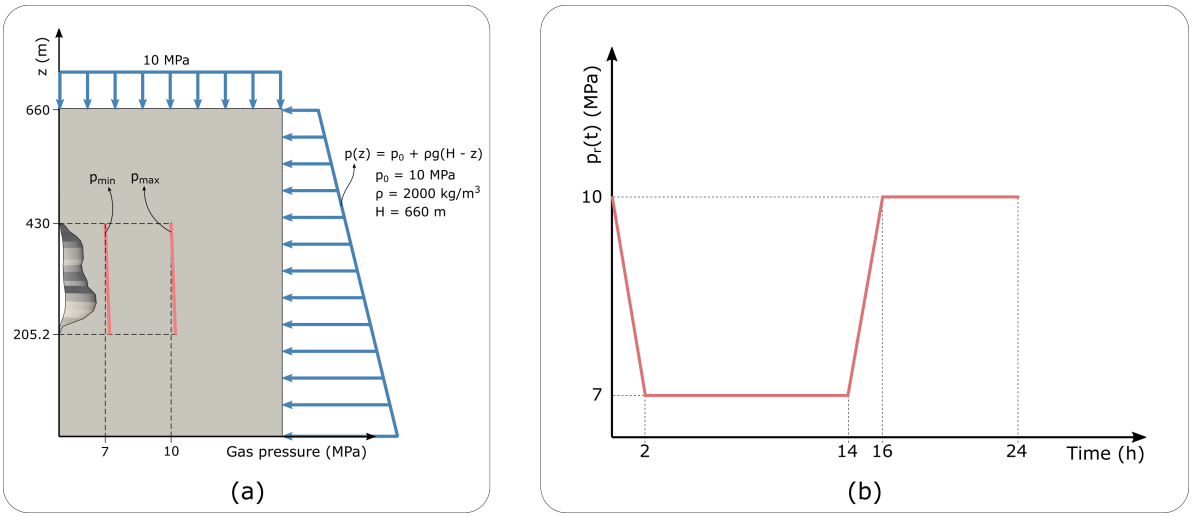


Fig. 3.5: Planar view of overburden and sideburden (a); time history of gas pressure at cavern roof (b).

In this manner, the boundary conditions applied to this problem can be written as,

$$
\begin{aligned}
u(\mathbf{r},t) &= 0, \quad \forall \quad \mathbf{r} \in \Gamma_{\text{west}}, \\
v(\mathbf{r},t) &= 0, \quad \forall \quad \mathbf{r} \in \Gamma_{\text{south}}, \\
w(\mathbf{r},t) &= 0, \quad \forall \quad \mathbf{r} \in \Gamma_{\text{bottom}}, \\
\boldsymbol{\sigma}(\mathbf{r},t)\cdot\mathbf{n} &= p_1(z), \quad \forall \quad \mathbf{r} \in \Gamma_{\text{east}}, \\
\boldsymbol{\sigma}(\mathbf{r},t)\cdot\mathbf{n} &= p_1(z), \quad \forall \quad \mathbf{r} \in \Gamma_{\text{north}}, \\
\boldsymbol{\sigma}(\mathbf{r},t)\cdot\mathbf{n} &= p_1(z), \quad \forall \quad \mathbf{r} \in \Gamma_{\text{top}}, \\
\boldsymbol{\sigma}(\mathbf{r},t)\cdot\mathbf{n} &= p_2(z,t), \quad \forall \quad \mathbf{r} \in \Gamma_{\text{cavern}},
\end{aligned}
$$

in which

$$
\begin{aligned}
p_1(z) &= p_h + \rho_{\text{salt}} g (H - z), \\
p_2(z,t) &= p_r(t) + \rho_{\text{H}_2} g (H_r - z),
\end{aligned}
$$

with $\rho_{\text{salt}} = 2000\ \text{kg/m}^3$, $\rho_{\text{H}_2} = 10\ \text{kg/m}^3$, $H_r = 430$ m, $p_h = 10$ MPa and $p_r(t)$ is shown in Fig. 3.5-b.

The constitutive model employed in this example is shown in Fig. 3.6. It comprises a linear spring for instantaneous elastic response, a Kelvin-Voigt element for time dependent elasticity (i.e. viscoelasticity), a viscoplastic element following Desai's model, and a power law model for steady-state dislocation creep.

3.3.1 Build input file

The following code-blocks show how to build the input file for the problem described above.

In Listing 3.7 the relevant modules are imported, useful units are defined and the *BuildInputFile* object (*bif*) is created. Notice in lines 20 and 21 how the boundary and subdomain (region) names can be retrieved

Listing 3.7: Initial steps.

```
1 import os
2 import sys
3 import numpy as np
4 sys.path.append(os.path.join("..", "..", "safeincave"))
```

(continues on next page)

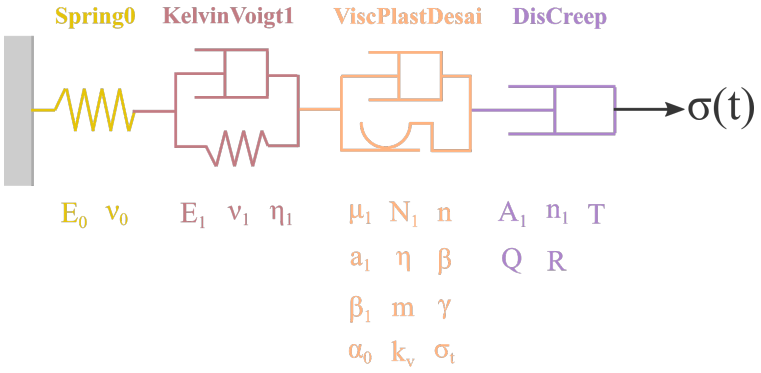


Fig. 3.6: Constitutive model employed in Tutorial 2.

(continued from previous page)

```
from Grid import GridHandlerGMSH
from InputFileAssistant import BuildInputFile

# Useful units
hour = 60*60
day = 24*hour
MPa = 1e6

# Initialize input file object
bif = BuildInputFile()

# Create input_grid section
path_to_grid = os.path.join("..", "..", "grids", "cavern_irregular")
bif.section_input_grid(path_to_grid, "geom")

print(bif.grid.get_boundary_names())
print(bif.grid.get_subdomain_names())
```

In Listing 3.8, some useful dimensions are extracted from the grid, except for the cavern roof position, which is hard coded for convenience.

Listing 3.8: Extract geometry dimensions.

```
Lx = bif.grid.Lx
Ly = bif.grid.Ly
Lz = bif.grid.Lz
cavern_roof = 430
```

The following code block is pretty much the same as in Tutorial 1.

```
# Create output section
bif.section_output(os.path.join("output", "case_1"))

# Create solver settings section
solver_settings = {
    "type": "KrylovSolver",
    "method": "cg",
    "preconditioner": "petsc_amg",
```

(continues on next page)

(continued from previous page)

```
    "relative_tolerance": 1e-12,
}
bif.section_solver(solver_settings)

# Create simulation_settings section
bif.section_simulation(
    simulation_settings = {
        "equilibrium": {
            "active": True,
            "dt_max": 0.5*hour,
            "time_tol": 1e-4
        },
        "operation": {
            "active": True,
            "dt_max": 0.1*hour,
            "n_skip": 2
        }
    }
)

# Create body_forces section
salt_density = 2000
bif.section_body_forces(value=salt_density, direction=2)
```

As shown in Fig. 3.5-b, the times when the gas pressure change are 0, 2h, 14h, 16h and 24h. Therefore, the time list should be defined as in line 2 of Listing 3.9

Listing 3.9: Defining time list according to Fig. 3.5-b.

```
time_list = [0*hour,  2*hour,  14*hour, 16*hour, 24*hour]
bif.section_time(time_list, theta=0.0)
```

The boundary conditions are defined in Listing 3.10. The Dirichlet boundary conditions are defined in lines 5, 13 and 21. The sideburden is applied to boundaries *East* and *North* in lines 31 and 42, respectively. Notice the salt density is assigned to the *density* key, and height of the geometry (Lz) is assigned to *reference_position* key. The sideburden and overburden are time independent, which is why lists with constant values are used in lines 38, 49 and 60. Also note that it would make no difference if the density value in line 57 would be zero, as this boundary is perpendicular to the z direction (direction 2). For the cavern wall (boundary *Cavern*), the *density* value is the one of hydrogen (line 69), the *reference_position* corresponds to the cavern roof (line 70), and a time dependent list is informed in line 71 according to Fig. 3.5-b.

Listing 3.10: Create *boundary_conditions* section.

```
bif.section_boundary_conditions()

# Add Dirichlet boundary conditions
bif.add_boundary_condition(
    boundary_name = "West",
    bc_data = {
        "type": "dirichlet",
        "component": 0,
        "values": list(np.zeros(len(time_list)))
    }
```

(continues on next page)

(continued from previous page)

```
)
bif.add_boundary_condition(
    boundary_name = "South",
    bc_data = {
        "type": "dirichlet",
        "component": 1,
        "values": list(np.zeros(len(time_list)))
    }
)
bif.add_boundary_condition(
    boundary_name = "Bottom",
    bc_data = {
        "type": "dirichlet",
        "component": 2,
        "values": list(np.zeros(len(time_list)))
    }
)

# Add Neumann boundary condition
bif.add_boundary_condition(
    boundary_name = "East",
    bc_data = {
        "type": "neumann",
        "direction": 2,
        "density": salt_density,
        "reference_position": Lz,
        "values": [10*MPa, 10*MPa, 10*MPa, 10*MPa, 10*MPa]
    }
)

bif.add_boundary_condition(
    boundary_name = "North",
    bc_data = {
        "type": "neumann",
        "direction": 2,
        "density": salt_density,
        "reference_position": Lz,
        "values": [10*MPa, 10*MPa, 10*MPa, 10*MPa, 10*MPa]
    }
)

bif.add_boundary_condition(
    boundary_name = "Top",
    bc_data = {
        "type": "neumann",
        "direction": 2,
        "density": salt_density,
        "reference_position": Lz,
        "values": [10*MPa, 10*MPa, 10*MPa, 10*MPa, 10*MPa]
    }
)

```

(continues on next page)

(continued from previous page)

```
h2_density = 10
bif.add_boundary_condition(
    boundary_name = "Cavern",
    bc_data = {
        "type": "neumann",
        "direction": 2,
        "density": h2_density,
        "reference_position": cavern_roof,
        "values": [10*MPa, 7*MPa, 7*MPa, 10*MPa, 10*MPa]
    }
)
```

The code block Listing 3.11 defines the constitutive model shown in Fig. 3.6. The material properties, as discussed before, are all homogeneous (i.e. no spatial variation).

Listing 3.11: Defining *constitutive_model* section.

```
# Assign material properties
bif.section_constitutive_model()

# Add elastic properties
bif.add_elastic_element(
    element_name = "Spring0",
    element_parameters = {
        "type": "Spring",
        "active": True,
        "parameters": {
            "E":  list(102e9*np.ones(bif.n_elems)),
            "nu": list(0.3*np.ones(bif.n_elems))
        }
    }
)

# Add viscoelastic properties
bif.add_viscoelastic_element(
    element_name = "KelvinVoigt1",
    element_parameters = {
        "type": "KelvinVoigt",
        "active": True,
        "parameters": {
            "E":   list(10e9*np.ones(bif.n_elems)),
            "nu":  list(0.32*np.ones(bif.n_elems)),
            "eta": list(105e11*np.ones(bif.n_elems))
        }
    }
)

# Add viscoplastic parameters
bif.add_inelastic_element(
    element_name = "ViscPlastDesai",
    element_parameters = {
        "type": "ViscoplasticDesai",
```

(continues on next page)

(continued from previous page)

```
        "active": False,
        "parameters": {
            "mu_1":     list(5.3665857009859815e-11*np.ones(bif.n_elems)),
            "N_1":      list(3.1*np.ones(bif.n_elems)),
            "n":        list(3.0*np.ones(bif.n_elems)),
            "a_1":      list(1.965018496922832e-05*np.ones(bif.n_elems)),
            "eta":      list(0.8275682807874163*np.ones(bif.n_elems)),
            "beta_1":   list(0.0048*np.ones(bif.n_elems)),
            "beta":     list(0.995*np.ones(bif.n_elems)),
            "m":        list(-0.5*np.ones(bif.n_elems)),
            "gamma":    list(0.095*np.ones(bif.n_elems)),
            "alpha_0":  list(0.0022*np.ones(bif.n_elems)),
            "k_v":      list(0.0*np.ones(bif.n_elems)),
            "sigma_t":  list(5.0*np.ones(bif.n_elems))
        }
    }
)

# Add dislocation creep parameters
bif.add_inelastic_element(
    element_name = "DisCreep",
    element_parameters = {
        "type": "DislocationCreep",
        "active": True,
        "parameters": {
            "A": list(1.9e-20*np.ones(bif.n_elems)),
            "n": list(3.0*np.ones(bif.n_elems)),
            "T": list(298*np.ones(bif.n_elems)),
            "Q": list(51600*np.ones(bif.n_elems)),
            "R": list(8.32*np.ones(bif.n_elems))
        }
    }
)

# Save input_file.json
bif.save_input_file("input_file.json")
```

To run this example, build the input file and execute the *main.py* file in *examples/tutorial_2* folder. That is,

```
user@computer:~/safeincave$ cd examples/tutorial_2
user@computer:~/safeincave/examples/tutorial_2$ python build_input_file.py
user@computer:~/safeincave/examples/tutorial_2$ python main.py
```

3.3.2 Visualize results

For the purpose of demonstration, in this section we intend to plot the initial and final cavern shape, as well as the volumetric closure of the cavern over time.

```
import os
import sys
sys.path.append(os.path.join("..", "..", "safeincave"))
from ResultsHandler import convert_vtk_to_pandas
import matplotlib.pyplot as plt
import numpy as np
import meshio
```

```
minute = 60
hour = 60*minute
day = 24*hour
MPa = 1e6
```

```
def trapezoidal_volume(x, y):
    volume = 0.0
    area = 0.0
    n = len(x)
    for i in range(1, n):
        R = 0.5*(y[i] + y[i-1])
        A = np.pi*R**2
        d = x[i] - x[i-1]
        area += R*d
        volume += A*d
    return volume
```

```
def reorder_data(df_coord, u, v, w, wall_ind):
    # Initial cavern shape
    x0 = df_coord.iloc[wall_ind]["x"]
    y0 = df_coord.iloc[wall_ind]["y"]
    z0 = df_coord.iloc[wall_ind]["z"]
    # Reorder all coordinates according to coordinate z
    sorted_z0_ind = z0.sort_values().index
    x0 = x0[sorted_z0_ind]
    y0 = y0[sorted_z0_ind]
    z0 = z0[sorted_z0_ind]
    # Reorder all displacements according to coordinate z
    u = u.iloc[wall_ind].loc[sorted_z0_ind]
    v = v.iloc[wall_ind].loc[sorted_z0_ind]
    w = w.iloc[wall_ind].loc[sorted_z0_ind]
    return x0, y0, z0, u, v, w
```

```
# Define folders
results_folder = os.path.join("output", "case_0", "operation", "vtk")
mesh_folder = os.path.join("..", "..", "grids", "cavern_regular")
```

```
# Read displacement results
pvd_path = os.path.join(results_folder, "displacement")
```

(continues on next page)

(continued from previous page)

```
pvd_file = "displacement.pvd"
df_coord, u, v, w = convert_vtk_to_pandas(pvd_path, pvd_file)
```

During the geometry construction in Gmsh, we have selected the border of the cavern wall (a line) and gave it a name. In this manner, this information is saved in the mesh file so that we can retrieve it at our convenience. This is done in line 2 of Listing 3.12.

Listing 3.12: Get indices of wall profile.

```
mesh = meshio.read(os.path.join(mesh_folder, "geom.msh"))
wall_ind = np.unique(mesh.cells["line"].flatten())
```

```
# Get reordered data over cavern wall
x0, y0, z0, u, v, w = reorder_data(df_coord, u, v, w, wall_ind)
```

```
# Get times
times = u.columns.values
t_final = times[-1]
```

```
# Compute cavern volumes over time
vol_0 = trapezoidal_volume(z0.values, x0.values)
volumes = []
for t in times[1:]:
    z = z0.values + w[t].values
    x = x0.values + u[t].values
    vol = trapezoidal_volume(z, x)
    volumes.append(100*abs(vol_0 - vol)/vol_0)
```

```
# Create figure
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(9, 3))
fig.subplots_adjust(
    top=0.985, bottom=0.145, left=0.070, right=0.990, hspace=0.35, wspace=0.260
)
```

```
# Plot cavern shape
expansion_factor = 50
xf = x0 + expansion_factor*u[t_final]
yf = y0 + expansion_factor*v[t_final]
zf = z0 + expansion_factor*w[t_final]
ax1.plot(x0, z0, "-", color="black", linewidth=2.0, label="Initial shape")
ax1.plot(-x0, z0, "-", color="black", linewidth=2.0)
ax1.plot(xf, zf, "-", color="#377eb8", linewidth=2.0, label=f"Final shape")
ax1.plot(-xf, zf, "-", color="#377eb8", linewidth=2.0)
ax1.set_xlabel("x (m)", size=12, fontname="serif")
ax1.set_ylabel("z (m)", size=12, fontname="serif")
ax1.legend(loc=1, shadow=True, fancybox=True)
ax1.axis("equal")
ax1.grid(True, color='0.92')
ax1.set_facecolor("0.85")
```

```
# Plot cavern volumetric closure
ax2.plot(times[2:]/hour, volumes[1:] - 0*volumes[1], ".-", color="#377eb8", linewidth="2.0")
ax2.set_xlabel("Time (h)", size=12, fontname="serif")
ax2.set_ylabel("Cavern closure (%)", size=12, fontname="serif")
ax2.grid(True, color='0.92')
ax2.set_facecolor("0.85")
```

```
plt.show()
```

The results are shown in Fig. 3.7.

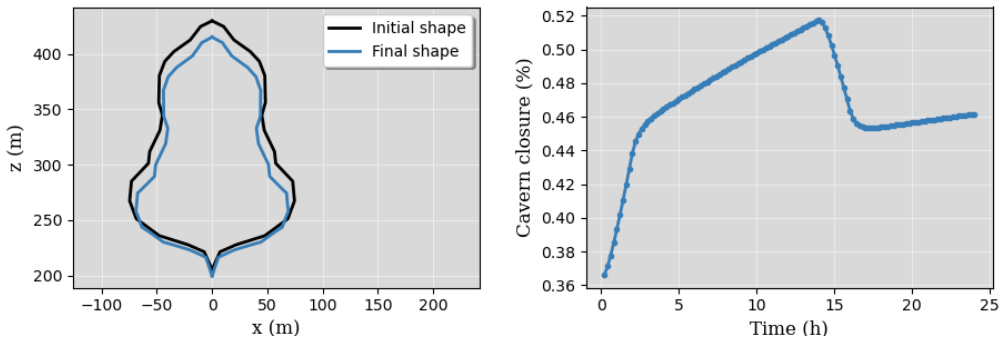


Fig. 3.7: Initial and final cavern shape (left) and cavern volumetric closure (right).

CHAPTER

FOUR

FUNDAMENTAL THEORY

This chapter is intended to provide the basic concepts of computational solid mechanics necessary to understand the SafeInCave implementation.

4.1 Initial considerations

Definitions used throughout this documentation:

- A **spring element** describes an instantaneous (i.e. time-independent) elastic response. When a load is applied, this element instantaneously deforms to its final configuration. When the load is removed, the material instantaneously recovers to its initial configuration.
- A **Kelvin-Voigt element** describes a viscoelastic (i.e. time-dependent elastic) response. When a load is applied, it takes a finite amount of time for this element to reach its final configuration (equilibrium condition). When the load is removed, the deformation is fully recovered within a finite amount of time.
- A **plastic element** describes an instantaneous (i.e. time-indepentent) inelastic response. If the stresses applied exceeds a certain threshold (yield surface), this element will instantaneously deform to its final configuration. If the load is removed, the material does not recover at all. In case the applied load does not exceed the threshold, this element does not deform.
- A **viscoplastic element** describes a time-dependent inelastic response. It behaves exactly as the plastic element, except that the inelastic deformations take place within a finite amount of time. For example, if the applied stresses exceed the yield surface, the material will not instantaneously reach its final configuration, but within some time.

- Stress tensor:

$$\boldsymbol{\sigma} = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{xy} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{xz} & \sigma_{yz} & \sigma_{zz} \end{bmatrix}$$

- Identity (rank-2) tensor:

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Deviatoric stress:

$$\mathbf{s} = \boldsymbol{\sigma} - \frac{1}{3}\text{tr}(\boldsymbol{\sigma})\mathbf{I}$$

- Von Mises stress:

$$q = \sqrt{\frac{1}{2}\left((s_{xx} - s_{yy})^2 + (s_{xx} - s_{zz})^2 + (s_{yy} - s_{zz})^2 + 6(s_{xy}^2 + s_{xz}^2 + s_{yz}^2)\right)}$$

or

$$q = \sqrt{\frac{3}{2}\mathbf{s} : \mathbf{s}} = \sqrt{\frac{3}{2}s_{ij}s_{ij}}$$

or

$$q = \sqrt{3J_2}$$

- Stress invariants of $\boldsymbol{\sigma}$:

$$I_1 = s_{xx} + s_{yy} + s_{zz} = \text{tr}(\boldsymbol{\sigma})$$
$$I_2 = s_{xx}s_{yy} + s_{yy}s_{zz} + s_{xx}s_{zz} - s_{xy}^2 - s_{yz}^2 - s_{xz}^2$$
$$I_3 = s_{xx}s_{yy}s_{zz} + 2s_{xy}s_{yz}s_{xz} - s_{zz}s_{xy}^2 - s_{xx}s_{yz}^2 - s_{yy}s_{xz}^2 = \det(\boldsymbol{\sigma})$$

- Stress invariants of $\mathbf{s}$:

$$J_1 = \frac{1}{3}I_1^2 - I_2$$
$$J_2 = \frac{2}{27}I_1^3 - \frac{1}{3}I_1I_2 + I_3$$
$$J_3 = 0$$

- Lode's angle, ψ:

$$\cos(3\psi) = -\frac{\sqrt{27}}{2}\frac{J_3}{J_2^{3/2}} \quad \rightarrow \quad \psi = \frac{1}{3}\cos^{-1}\left(-\frac{\sqrt{27}}{2}\frac{J_3}{J_2^{3/2}}\right)$$

- Von Mises stress:

$$q = \sqrt{\frac{1}{2}\left((\sigma_{xx} - \sigma_{yy})^2 + (\sigma_{xx} - \sigma_{zz})^2 + (\sigma_{yy} - \sigma_{zz})^2 + 6(\sigma_{xy}^2 + \sigma_{xz}^2 + \sigma_{yz}^2)\right)}$$

or

$$q = \sqrt{\frac{3}{2}\mathbf{s} : \mathbf{s}} = \sqrt{\frac{3}{2}s_{ij}s_{ij}}$$

or

$$q = \sqrt{3J_2}$$

4.2 Constitutive models

In general, a constitutive model can be represented as illustrated in Fig. 4.1, which shows a serial arrangement of different types of elements (springs, dashpots, etc). The total deformation $\boldsymbol{\varepsilon}$ is given by the sum of the individual deformation of all elements composing the constitutive model. In this text, we make a distinction between **elastic** and **non-elastic** deformations. Elastic deformations $\boldsymbol{\varepsilon}_e$ refer exclusively to time-independent (instantaneous) elastic deformations – in other words, in only includes the deformation of the yellow spring in Fig. 4.1. The non-elastic deformations comprise the viscoelastic ($\boldsymbol{\varepsilon}_{ve}$) and inelastic ($\boldsymbol{\varepsilon}_{ie}$) deformations. In the SafeInCave simulator, the only viscoelastic element implemented is the Kelvin-Voigt element, which consists of parallel arrangement between a spring and a dashpot. More than one Kelvin-Voigt element can be arranged in series. For inelastic elements, the SafeInCave simulator provides two options: a viscoplastic element and a dislocation creep element. The viscoplastic element refers to the model proposed by Desai and Varadarajan (1987) [1] and used in Khaledi *et al* (2016) [2] for salt caverns. This element can be represented by a parallel arrangement between a dashpot, which represents the time dependency, and a friction element, which indicates that the dashpot will only move if the stresses exceed a certain threshold (the yield surface). As shown below, this dashpot also includes a hardening rule that expands the yield surface. Finally, the dislocation creep element is represented by a single dashpot, that starts to deform as soon as a non-zero deviatoric stress is applied. Moreover, this element has a non-linear dependency on stress.

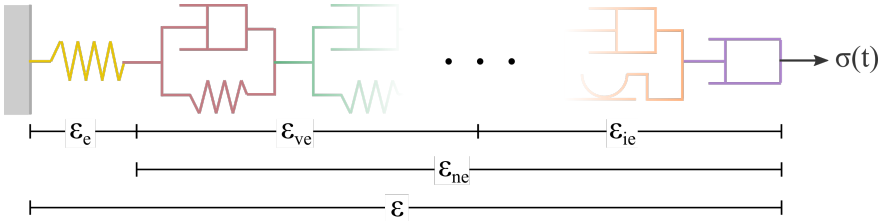


Fig. 4.1: Constitutive model composed of elastic and non-elastic (viscoelastic and inelastic) deformations.

From the discussion above and from Fig. 4.1, it follows that total deformation can be written as

$$\boldsymbol{\varepsilon} = \boldsymbol{\varepsilon}_e + \underbrace{\boldsymbol{\varepsilon}_{ve} + \boldsymbol{\varepsilon}_{ie}}_{\boldsymbol{\varepsilon}_{ne}}.$$

The mathematical formulations of these different types of elements are described in the following subsections.

Note: Technically, the dislocation creep element is also a viscoplastic element, as it describes a time-dependent inelastic deformation. However, it differs from Desai's model in the sense that it does not present a yield surface. In better terms, its yield surface is a point, hence any applied deviatoric stress exceeds the yield surface.

Note: In salt rocks, plastic deformations are always time-dependent, we do not address plastic deformations in this documentation.

4.2.1 Kelvin-Voigt element

The Kelvin-Voigt element consists of a parallel arrangement between a spring and a dashpot. The stress $\boldsymbol{\sigma}$ applied this type of element is balanced by the stresses on the spring and dashpot. That is,

$$\boldsymbol{\sigma} = \underbrace{\mathbb{C}_1 : \boldsymbol{\varepsilon}_{ve}}_{\text{spring}} + \underbrace{\eta_1 \dot{\boldsymbol{\varepsilon}}_{ve}}_{\text{dashpot}} \tag{4.1}$$

where $\boldsymbol{\varepsilon}_{ve}$ represents the deformation of both spring and dashpot. Solving Eq. (4.1) for $\dot{\boldsymbol{\varepsilon}}_{ve}$,

$$\dot{\boldsymbol{\varepsilon}}_{ve} = \frac{1}{\eta_1} \left(\boldsymbol{\sigma} - \mathbb{C}_1 : \boldsymbol{\varepsilon}_{ve} \right) \tag{4.2}$$

4.2.2 Dislocation creep element

The dislocation creep mechanism is commonly described by a power-law function together with Arrhenius law. The expression for the dislocation creep strain rate can be written as,

$$\dot{\boldsymbol{\varepsilon}}_{cr} = A \exp\left(-\frac{Q}{RT} \right) q^{n-1} \mathbf{s} \tag{4.3}$$

where A and n are material parameters, Q is the activation energy (in J/mol), R is the universal gas constant ($R = 8.32\ \text{JK}^{-1}\text{mol}^{-1}$), and T is the temperature in Kelvin. Additionally, q and s represent the Von Mises stress and the deviatoric stress, respectively.

4.2.3 Viscoplastic element

The viscoplastic element follows the formulation proposed in [1], that is,

$$\dot{\boldsymbol{\varepsilon}}_{vp} = \mu_1 \left\langle \frac{F_{vp}}{F_0} \right\rangle^{N_1} \frac{\partial Q_{vp}}{\partial \boldsymbol{\sigma}} \tag{4.4}$$

where μ_1 and N_1 are material parameters, and F_0 is reference value equal to 1 MPa. The terms F_{vp} and Q_{vp} represent the yield and potential functions, respectively. In this work, only the associative formulation is implemented, that is, $F_{vp} = Q_{vp}$. The yield function is given by

$$F_{vp}(\boldsymbol{\sigma}, \alpha) = J_2 - (-\alpha I_1^n + \gamma I_1^2) \left[\exp\left(\beta_1 I_1\right) - \beta \cos(3\psi)\right]^m \tag{4.5}$$

where γ, n, β_1, β and m are material parameters. The terms I_1, J_2 and ψ are stress invariants (see *Initial considerations*). Finally, α represents the internal hardening parameter. It's function is to enlarge the yield surface as the inelastic deformation (ξ) accumulates in the material. The evolution equation for the hardening parameter adopted in this work has the following form,

$$\alpha = a_1 \left[\left(\frac{a_1}{\alpha_0} \right)^{1/\eta} + \xi \right]^{-\eta}, \tag{4.6}$$

where a_1 and η are material parameters, α_0 is the initial hardening parameter, and the accumulated inelastic strain is given by

$$\xi = \int_{t_0}^{t} \sqrt{\dot{\boldsymbol{\varepsilon}}_{vp} : \dot{\boldsymbol{\varepsilon}}_{vp}} \mathrm{dt}. \tag{4.7}$$

The initial hardening parameter can be chosen arbitrarily or based on a specific value of F_{vp}. For a certain value F_{vp}^*, for example, the initial hardening parameter can be computed as

$$\alpha_0 = \gamma I_1^{2-n} + \frac{F_{vp}^* - J_2}{I_1^n} \left[\exp(\beta_1 I_1) + \beta \cos(3\psi)\right].$$

Evidently, placing the stress state at the onset of viscoplasticity is achieved by setting $F_{vp}^* = 0$.

4.3 Mathematical Formulation

This section presents the linear momentum balance equation for a general consitutive model considering small strains assumption, such that the additive decomposition can be applied to the total strain tensor.

4.3.1 Linear momentum balance equation

The linear momentum balance equation considering quasi-static loads can be written as

$$\nabla \cdot \boldsymbol{\sigma} = \mathbf{f} \tag{4.8}$$

with $\mathbf{f}$ representing the body forces. In Eq. (4.8), the stress is calculated as,

$$\boldsymbol{\sigma} = \mathbb{C}_0 : \boldsymbol{\varepsilon}_e \tag{4.9}$$

where $\boldsymbol{\varepsilon}_e$ is the elastic strain tensor and $\mathbb{C}_0$ is the 4th-order tensor associated to the linear elastic response of the material (yellow spring of Fig. 4.1). However, most constitutive models for geomaterials, especially salt rocks, comprise elastic, viscoelastic (i.e. time-dependent elastic), and viscoplastic (i.e. time-dependent inelastic) deformations.

Note: In the present work, non-elastic deformation includes all types of deformation that are not instantaneously elastic, that is, viscoelastic (time dependent elastic) and inelastic (viscoplastic, plastic, creep, etc) deformations.

The elastic strain tensor can be represented as

$$\boldsymbol{\varepsilon}_e = \boldsymbol{\varepsilon} - \boldsymbol{\varepsilon}_{ne} \tag{4.10}$$

where the non-elastic strains are given by

$$\boldsymbol{\varepsilon}_{ne} = \sum_{i=1}^{N_{ne}} \boldsymbol{\varepsilon}_i, \tag{4.11}$$

with N_{ne} denoting the number of non-elastic elements included in the constitutive model. In this manner, the stress tensor can be expressed as

$$\boldsymbol{\sigma} = \mathbb{C}_0^{-1} : (\boldsymbol{\varepsilon} - \boldsymbol{\varepsilon}_{ne}) . \tag{4.12}$$

In general, the non-elastic strain rates have a (non-)linear dependency on the stress tensor $\boldsymbol{\sigma}$ and, possibly, on internal parameters α_i. For example, for a non-elastic element *i*,

$$\dot{\boldsymbol{\varepsilon}}_i = \dot{\boldsymbol{\varepsilon}}_i (\boldsymbol{\sigma}, \alpha_i) . \tag{4.13}$$

The circular dependency of the non-elastic strains on the stress tensor $\boldsymbol{\sigma}$ makes of Eq. (4.8) a non-linear equation. The numerical procedure for treating this non-linearity and solving Eq. (4.8) is described below.

4.4 Numerical formulation

This section discusses possible linearization strategies for the momentum balance equation presented above. First, the time discretization method (θ-method) is presented, which is followed by Picard's and Newton's linearization methods. The weak form of the governning equations is also presented, but the finite element implementation is supressed as it is handled by FEniCS. Finally, the algorithm for the Newton's linearization method implemented in SafeInCave simulator is discussed.

4.4.1 Time integration

The strain tensor at time $t + \Delta t$ of a given non-elastic element i can be approximated by

$$\boldsymbol{\varepsilon}_i^{t+\Delta t} = \boldsymbol{\varepsilon}_i^t + \Delta t \dot{\boldsymbol{\varepsilon}}_i^\theta$$

where $\dot{\boldsymbol{\varepsilon}}_i^\theta = \theta \dot{\boldsymbol{\varepsilon}}_i^t + (1-\theta)\dot{\boldsymbol{\varepsilon}}_i^{t+\Delta t}$, and θ can be chosen among 0.0, 0.5 and 1.0 for fully implicit, Crank-Nicolson and explicit time integration, respectively. However, the strain rate $\dot{\boldsymbol{\varepsilon}}_i^{t+\Delta t}$ is unknown and it will be determined in a iterative process, so we drop the superscript $t + \Delta t$ and replace it by $k + 1$, where k denotes the iterative level. In this manner, the strain of element i at iteration $k + 1$ is

$$\boldsymbol{\varepsilon}_i^{k+1} = \boldsymbol{\varepsilon}_i^t + \Delta t \theta \dot{\boldsymbol{\varepsilon}}_i^t + \Delta t (1-\theta) \dot{\boldsymbol{\varepsilon}}_i^{k+1}. \tag{4.14}$$

For conciseness, let us consider $\phi_1 = \Delta t \theta$ and $\phi_2 = \Delta t (1 - \theta)$. Recalling Eq. (4.11) and substituting Eq. (4.14) into Eq. (4.12), the stress tensor at iteration $k + 1$ is expressed as

$$\boldsymbol{\sigma}^{k+1} = \mathbb{C}_0 : \left(\boldsymbol{\varepsilon}^{k+1} - \boldsymbol{\varepsilon}_{ne}^t - \phi_1 \dot{\boldsymbol{\varepsilon}}_{ne}^t - \phi_2 \dot{\boldsymbol{\varepsilon}}_{ne}^{k+1} \right) \tag{4.15}$$

where $\dot{\boldsymbol{\varepsilon}}_{ne}^{k+1}$ is obviously unknown, which requires a linearization method for its evaluation.

Note: Keep in mind that both $\boldsymbol{\varepsilon}_i^t$ and $\dot{\boldsymbol{\varepsilon}}_i^t$ are known quantities.

4.4.2 Picard's method

One alternative to linearize Eq. (4.15) is to simply consider

$$\dot{\boldsymbol{\varepsilon}}_i^{k+1} = \dot{\boldsymbol{\varepsilon}}_i^k$$

in which $\dot{\boldsymbol{\varepsilon}}_i^k = \dot{\boldsymbol{\varepsilon}}_i\left(\boldsymbol{\sigma}^k, \alpha_i^k\right)$. As a consequence, the stress tensor is linearized as

$$\boldsymbol{\sigma}^{k+1} = \mathbb{C}_0 : \left(\boldsymbol{\varepsilon}^{k+1} - \boldsymbol{\varepsilon}_{ne}^t - \phi_1 \dot{\boldsymbol{\varepsilon}}_{ne}^t - \phi_2 \dot{\boldsymbol{\varepsilon}}_{ne}^k\right)$$

and the momentum balance equation becomes

$$\nabla \cdot \mathbb{C}_0 : \boldsymbol{\varepsilon}^{k+1} = \mathbf{f} + \nabla \cdot \mathbb{C}_0 : \left(\boldsymbol{\varepsilon}_{ne}^t + \phi_1 \dot{\boldsymbol{\varepsilon}}_{ne}^t + \phi_2 \dot{\boldsymbol{\varepsilon}}_{ne}^k\right). \tag{4.16}$$

Although very simple, Eq. (4.16) requires many iterations to converge and it is often unstable, especially when highly non-linear deformations are present.

4.4.3 Newton's method

Alternatively, the strain rate can be expanded from iteration k to iteration $k+1$ by using Taylor series, that is,

$$\dot{\boldsymbol{\varepsilon}}_i^{k+1} = \dot{\boldsymbol{\varepsilon}}_i^k + \frac{\partial \dot{\boldsymbol{\varepsilon}}_i}{\partial \boldsymbol{\sigma}} : \delta\boldsymbol{\sigma} + \frac{\partial \dot{\boldsymbol{\varepsilon}}_i}{\partial \alpha_i}\delta\alpha_i \tag{4.17}$$

where $\delta\boldsymbol{\sigma} = \boldsymbol{\sigma}^{k+1} - \boldsymbol{\sigma}^k$ and $\delta\alpha_i = \alpha_i^{k+1} - \alpha_i^k$.

Note: The term $\frac{\partial \dot{\boldsymbol{\varepsilon}}_i}{\partial \boldsymbol{\sigma}}$ is a rank-4 tensor, whereas $\delta\boldsymbol{\sigma}$ is a rank-2 tensor, hence the double dot product between them, which results a rank-2 tensor.

The increment of internal variable $\delta\alpha_i$ can be obtained by defining a residual equation of the evolution equation of α_i and using Newton-Raphson to drive the residue to zero. Considering the residual equation is of the form $r_i = r_i(\boldsymbol{\sigma}, \alpha_i)$, it follows that

$$r_i^{k+1} = r^k + \frac{\partial r_i}{\partial \boldsymbol{\sigma}} : \delta\boldsymbol{\sigma} + \underbrace{\frac{\partial r_i}{\partial \alpha_i}}_{h_i}\delta\alpha_i = 0 \quad \rightarrow \quad \delta\alpha_i = -\frac{1}{h_i}\left(r_i^k + \frac{\partial r_i}{\partial \boldsymbol{\sigma}} : \delta\boldsymbol{\sigma}\right). \tag{4.18}$$

Substituting Eq. (4.18) into Eq. (4.17) yields

$$\dot{\boldsymbol{\varepsilon}}_i^{k+1} = \dot{\boldsymbol{\varepsilon}}_i^k + \underbrace{\left(\frac{\partial \dot{\boldsymbol{\varepsilon}}_i}{\partial \boldsymbol{\sigma}} - \frac{1}{h_i}\frac{\partial \dot{\boldsymbol{\varepsilon}}_i}{\partial \alpha_i}\frac{\partial r_i}{\partial \boldsymbol{\sigma}}\right)}_{\mathbb{G}_i} : \delta\boldsymbol{\sigma} - \underbrace{\frac{r_i^k}{h_i}\frac{\partial \dot{\boldsymbol{\varepsilon}}_i}{\partial \alpha_i}}_{\mathbf{B}_i} \tag{4.19}$$

Considering all non-elastic elements,

$$\dot{\boldsymbol{\varepsilon}}_{ne}^{k+1} = \dot{\boldsymbol{\varepsilon}}_{ne}^k + \mathbb{G}_{ne} : \delta\boldsymbol{\sigma} - \mathbf{B}_{ne} \tag{4.20}$$

where $\mathbb{G}_{ne} = \sum_{i=1}^{N_{ne}} \mathbb{G}_i$ and $\mathbf{B}_{ne} = \sum_{i=1}^{N_{ne}} \mathbf{B}_i$.

Finally, substituting Eq. (4.20) into Eq. (4.15) leads to

$$\boldsymbol{\sigma}^{k+1} = \mathbb{C}_T : \left[\boldsymbol{\varepsilon}^{k+1} - \bar{\boldsymbol{\varepsilon}}_{ne}^k + \phi_2\left(\mathbb{G}_{ne} : \boldsymbol{\sigma}^k + \mathbf{B}_{ne}\right)\right] \tag{4.21}$$

where $\bar{\boldsymbol{\varepsilon}}_{ne}^k = \boldsymbol{\varepsilon}_{ne}^t + \phi_1 \dot{\boldsymbol{\varepsilon}}_{ne}^t + \phi_2 \dot{\boldsymbol{\varepsilon}}_{ne}^k$ and the consistent tangent matrix $\mathbb{C}_T$ is given by

$$\mathbb{C}_T = \left(\mathbb{C}_0^{-1} + \phi_1 \mathbb{G}_{ne}\right)^{-1}. \tag{4.22}$$

We can further simplify Eq. (4.21) by defining

$$\boldsymbol{\varepsilon}_{\text{rhs}}^{k} = \bar{\boldsymbol{\varepsilon}}_{ne}^{k} - \phi_2 \left(\mathbb{G}_{ne} : \boldsymbol{\sigma}^{k} + \mathbf{B}_{ne} \right) \tag{4.23}$$

In this manner, the stress tensor can be expressed as

$$\boldsymbol{\sigma}^{k+1} = \mathbb{C}_T : \left(\boldsymbol{\varepsilon}^{k+1} - \boldsymbol{\varepsilon}_{\text{rhs}}^{k} \right) \tag{4.24}$$

and the linearized momentum balance equation becomes

$$\nabla \cdot \mathbb{C}_T : \boldsymbol{\varepsilon}^{k+1} = \mathbf{f} + \nabla \cdot \mathbb{C}_T : \boldsymbol{\varepsilon}_{\text{rhs}}^{k} \tag{4.25}$$

Note: It is important to note that $\mathbb{G}_{ne}$ is a rank-4 tensor, hence the double dot product : between $\mathbb{G}_{ne}$ and $\boldsymbol{\sigma}^{k}$. On the other hand, $\mathbf{B}_{ne}$ is a rank-2 tensor.

To close the formulation, the tensors $\mathbb{G}_i$ and $\mathbf{B}_i$ must be computed for each type of element included in the constitutive model. The procedure to compute these tensors for each element is presented below.

Viscoelastic element

An expression for the viscoelastic strain rate can be obtained by combining Equations (4.2) and (4.14), with $i = ve$, and solving for $\dot{\boldsymbol{\varepsilon}}_{ve}$. This results in the following equations

$$\dot{\boldsymbol{\varepsilon}}_{ve} = \left(\eta_1 \mathbb{I} + \phi_2 \mathbb{C}_1 \right)^{-1} : \left[\boldsymbol{\sigma} - \mathbb{C}_1 : \left(\boldsymbol{\varepsilon}_{ve}^{t} + \phi_1 \dot{\boldsymbol{\varepsilon}}_{ve}^{t} \right) \right], \tag{4.26}$$

where $\mathbb{I}$ represents the 4th-order identity tensor. The derivative of Eq. (4.26) with respect to $\boldsymbol{\sigma}$ gives,

$$\mathbb{G}_{ve} = \frac{\partial \dot{\boldsymbol{\varepsilon}}_{ve}}{\partial \boldsymbol{\sigma}} = \left(\eta_1 \mathbb{I} + \phi_2 \mathbb{C}_1 \right)^{-1}. \tag{4.27}$$

Furthermore, the viscoelastic model has no internal variables, implying that $\mathbf{B}_{ve} = 0$. Finally, the viscoelastic strain at iteration $k+1$ can be computed as,

$$\boldsymbol{\varepsilon}_{ve}^{k+1} = \boldsymbol{\varepsilon}_{ve}^{t} + \phi_1 \dot{\boldsymbol{\varepsilon}}_{ve}^{t} + \phi_2 \left(\dot{\boldsymbol{\varepsilon}}_{ve}^{k} + \mathbb{G}_{ve} : \delta\boldsymbol{\sigma} \right).$$

Dislocation creep element

Equation (4.3) gives the strain rate for the dislocation creep element. The tangent matrix is given by

$$\mathbb{G}_{cr} = \frac{\partial \dot{\boldsymbol{\varepsilon}}_{cr}}{\partial \boldsymbol{\sigma}},$$

where the partial derivatives are computed by finite differences. Since the dislocation creep model does not depend on internal variables ($\mathbf{B}_{cr} = 0$), the creep strain at iteration $k+1$ is computed as,

$$\boldsymbol{\varepsilon}_{cr}^{k+1} = \boldsymbol{\varepsilon}_{cr}^{t} + \phi_1 \dot{\boldsymbol{\varepsilon}}_{cr}^{t} + \phi_2 \left(\dot{\boldsymbol{\varepsilon}}_{cr}^{k} + \mathbb{G}_{cr} : \delta\boldsymbol{\sigma} \right).$$

Viscoplastic element

The viscoplastic strain rate is computed by Eq. (4.4), where α is an internal variable. For this reason, a residual function is defined based on the hardening rule as

$$r_{vp}^k = \alpha^k - a_1 \left[\left(\frac{a_1}{\alpha_0} \right)^{1/\eta} + \xi^k \right]^{-\eta}. \tag{4.28}$$

From Equations (4.28) and (4.4), tensors $\mathbb{G}_{vp}$ and $\mathbf{B}_{vp}$ are calculated as

$$\mathbb{G}_{vp} = \frac{\partial \dot{\boldsymbol{\varepsilon}}_{vp}}{\partial \boldsymbol{\sigma}} - \frac{1}{h_{vp}} \frac{\partial \dot{\boldsymbol{\varepsilon}}_{vp}}{\partial \alpha} \frac{\partial r_{vp}}{\partial \boldsymbol{\sigma}},$$

$$\mathbf{B}_{vp} = \frac{r_i^k}{h_i} \frac{\partial \dot{\boldsymbol{\varepsilon}}_i}{\partial \alpha_i},$$

where $h_{vp}^k = \frac{\partial r_{vp}^k}{\partial \alpha}$, according to Eq. (4.18). All derivatives, in this case, are computed by finite differences. The viscoplastic strain at iteration $k+1$ is

$$\boldsymbol{\varepsilon}_{vp}^{k+1} = \boldsymbol{\varepsilon}_{vp}^t + \phi_1 \dot{\boldsymbol{\varepsilon}}_{vp}^t + \phi_2 \left(\dot{\boldsymbol{\varepsilon}}_{vp}^k + \mathbb{G}_{vp} : \delta\boldsymbol{\sigma} \right) - \phi_2 \mathbf{B}_{vp}. \tag{4.29}$$

Note: To compute the derivative of r_{vp}, shown in Eq. (4.29), with respect to α, one must remember that the accumulated vicoplastic strain ξ also depends on α (see Eq. (4.7)).

4.4.4 Weak formulation

Consider a domain Ω bounded by a surface Γ outward oriented by a normal vector $\mathbf{n}$. Additionally, consider a vector **test** function $\mathbf{v} \in \mathcal{V}$ and a vector **trial** function $\mathbf{u} \in \mathcal{V}$, where $\mathcal{V}$ is the test function space generated by continuous piecewise linear polynomials. In this manner, the weak formulation of the linearized momentum balance equation can be expressed as,

$$\underbrace{\int_\Omega \mathbb{C}_T : \boldsymbol{\varepsilon}\left(\mathbf{u}^{k+1}\right) : \boldsymbol{\varepsilon}\left(\mathbf{v}\right) \mathrm{d}\Omega}_{a(\mathbf{u},\mathbf{v})} = \underbrace{\int_\Omega \mathbf{f} \cdot \mathbf{v} \mathrm{d}\Omega + \int_\Gamma \mathbf{t} \cdot \mathbf{v} \mathrm{d}\Gamma + \int_\Omega \mathbb{C}_T : \boldsymbol{\varepsilon}_{\text{rhs}}^k : \boldsymbol{\varepsilon}\left(\mathbf{v}\right) \mathrm{d}\Omega}_{L(\mathbf{v})} \tag{4.30}$$

where $a\left(\mathbf{u}, \mathbf{v}\right)$ and $L\left(\mathbf{v}\right)$ represent the well-known bilinear a linear operators, respectively. The term $\mathbf{t}$ is the traction vector applied at the portion of Γ where Neumann boundary conditions are imposed. Additionally, due to small strain assumption,

$$\boldsymbol{\varepsilon}(\mathbf{w}) = \frac{1}{2}\left(\nabla \mathbf{w} + \nabla \mathbf{w}^T\right),$$

in which $\mathbf{w} \in \mathcal{V}$.

4.4.5 Algorithm

The linear momentum balance equation is solved iteratively for each time step of the simulation due to the nonlinearities contained in the strain rates. The time marching and iterative procedures are illustrated in Fig. 4.2. At the beginning of the time marching loop, the total strain, non-elastic strain rate and stress tensor fields are updated with the values of the previous time level. To make it general, line 5 loops over all non-elastic elements to update the corresponding internal parameters α_i. It should be emphasized, however, that the only element in our constitutive model that depends on an internal parameter is the viscoplastic element.

Note: Remember that superscript *t* denotes a variable evaluated at the previous time level, whereas the superscripts *k* and *k+1* denote the variables evaluated at previous and current iteration, respectively, of the current time level.

1: **while** $t < t_{\text{final}}$ **do**
2: $\quad \boldsymbol{\varepsilon}^k \leftarrow \boldsymbol{\varepsilon}^t$
3: $\quad \dot{\boldsymbol{\varepsilon}}_{\text{ne}}^k \leftarrow \dot{\boldsymbol{\varepsilon}}_{\text{ne}}^t$
4: $\quad \boldsymbol{\sigma}^k \leftarrow \boldsymbol{\sigma}^t$
5: $\quad$ **for** $i \leftarrow 1$ **to** N_{ne} **do**
6: $\quad\quad \alpha_i^k \leftarrow \alpha_i^t$
7: $\quad$ **end for**
8: $\quad$ **while** error $>$ tolerance **do**
9: $\quad\quad \mathbb{G}_{\text{ne}} \leftarrow \sum_i \mathbb{G}_i$
10: $\quad\quad \mathbf{B}_{\text{ne}} \leftarrow \sum_i \mathbf{B}_i$
11: $\quad\quad \bar{\boldsymbol{\varepsilon}}_{\text{ne}}^k \leftarrow \boldsymbol{\varepsilon}_{\text{ne}}^t + \phi_1 \dot{\boldsymbol{\varepsilon}}_{\text{ne}}^t + \phi_2 \dot{\boldsymbol{\varepsilon}}_{\text{ne}}^k$
12: $\quad\quad \mathbb{C}_T \leftarrow \left(\mathbb{C}_0^{-1} + \phi_1 \mathbb{G}_{\text{ne}}\right)^{-1}$
13: $\quad\quad \boldsymbol{\varepsilon}_{\text{rhs}}^k \leftarrow \bar{\boldsymbol{\varepsilon}}_{\text{ne}}^k - \phi_2 \left(\mathbb{G}_{\text{ne}} : \boldsymbol{\sigma}^k + \mathbf{B}_{\text{ne}}\right)$
14: $\quad\quad$ Solve $\nabla \cdot \mathbb{C}_T : \boldsymbol{\varepsilon}^{k+1} \leftarrow \mathbf{f} + \nabla \cdot \mathbb{C}_T : \boldsymbol{\varepsilon}_{\text{rhs}}^k$
15: $\quad\quad \boldsymbol{\sigma}^{k+1} \leftarrow \mathbb{C}_T : \left(\boldsymbol{\varepsilon}^{k+1} - \boldsymbol{\varepsilon}_{\text{rhs}}^k\right)$
16: $\quad\quad$ **for** $i \leftarrow 1$ **to** N_{ne} **do**
17: $\quad\quad\quad \alpha_i^{k+1} \leftarrow \alpha_i^k - \frac{1}{h_i^k}\left(r_i^k + \frac{\partial r_i^k}{\partial \boldsymbol{\sigma}} : \delta\boldsymbol{\sigma}\right)$
18: $\quad\quad$ **end for**
19: $\quad\quad \dot{\boldsymbol{\varepsilon}}_{\text{ne}}^{k+1} \leftarrow \sum_i \dot{\boldsymbol{\varepsilon}}_i \left(\boldsymbol{\sigma}^{k+1}, \alpha_i^{k+1}\right)$
20: $\quad\quad \text{error} \leftarrow \dfrac{||\boldsymbol{\varepsilon}^{k+1} - \boldsymbol{\varepsilon}^k||}{||\boldsymbol{\varepsilon}^{k+1}||}$
21: $\quad\quad k \leftarrow k + 1$
22: $\quad$ **end while**
23: $\quad \boldsymbol{\varepsilon}_{\text{ne}}^t \leftarrow \bar{\boldsymbol{\varepsilon}}_{\text{ne}}^k + \phi_2 \left(\mathbb{G}_{\text{ne}} : \delta\boldsymbol{\sigma} - \mathbf{B}_{\text{ne}}\right)$
24: $\quad \dot{\boldsymbol{\varepsilon}}_{\text{ne}}^t \leftarrow \dot{\boldsymbol{\varepsilon}}_{\text{ne}}^k$
25: **end while**

Fig. 4.2: Pseudocode for solving the nonlinear mechanical model using Newton's method.

The iterative loop starts in line 8. In lines 9 and 10, the computation of matrices $\mathbb{G}_i$ and $\mathbf{B}_i$ follows Eq. (4.19). The consistent tangent matrix is computed in line 12, according to Eq. (4.22). In line 13, the linear momentum balance equation is solved for total strain $\boldsymbol{\varepsilon}^{k+1}$. This is, of course, a simplification of the process. In reallity, the discretized formulation of the weak form presented in Eq. (4.30) is solved for $\mathbf{u}^{k+1}$, and the total strain is computed as,

$$\boldsymbol{\varepsilon}^{k+1} = \frac{1}{2}\left[\nabla \mathbf{u}^{k+1} + \left(\nabla \mathbf{u}^{k+1}\right)^T\right] = \nabla_s \mathbf{u}^{k+1}.$$

As soon as the new total strain is computed, the stress tensor field of the current iteration is calculated in line 15, which then allows for incrementing the internal variables in line 17. With updated stress and internal variables, the nonelastic strain rate is of the current iteration $\dot{\boldsymbol{\varepsilon}}_{\text{ne}}$ is computed in line 19.

Note: Note that variables r_i^k, h_i^k, and $\frac{\partial r_i^k}{\partial \boldsymbol{\sigma}}$ required in line 17 have already been calculated in line 9, when matrix $\mathbb{G}_{\text{ne}}$

had to be calculated.

CHAPTER

FIVE

API DOCUMENTATION

API reference

5.1 safeincave

5.1.1 ConstitutiveModel module

It builds the constitutive model and contains all the information about it.

class ConstitutiveModel.**ConstitutiveModel**(*n_elems*, *input_constitutive_model*)

Bases: `object`

This class is responsible for constructing the constitutive model.

Parameters

- **n_elems** (*int*) – Number of grid elements. This is used to create arrays of C0 and C0_inv.
- **input_constitutive_model** (*dict*) – This is a dictionary that defines the constitutive model.

property C0

This is a (n_elems, 6, 6) tensor storing the stiffness matrix associated to the linear spring for each element of the grid.

Type
torch.Tensor

property C0_inv

This is a (n_elems, 6, 6) tensor storing the inverse of C0 for each element of the grid.

Type
torch.Tensor

property elems_e

This is a Python list storing elastic elements (i.e., objects of class *Elements.Spring*).

Type
list

property elems_ie

This is a Python list storing ineelastic elements (e.g., objects of class *Elements.DislocationCreep*).

Type
list

property elems_ve

This is a Python list storing viscoelastic elements (i.e., objects of class *Elements.Viscoelastic*).

Type
list

get_list_of_elements(*element_class='Elastic'*)

This is an internal function responsible to build a list of elements based on the input_constitutive_model dictionary.

Parameters
element_class (*str, default: "Elastic"*) – Possible values are "Elastic", "Viscoelastic" and "Inelastic".

Returns
Python list containing the elements of a particular class (element_class).

Return type
list

property n_elems

Number of grid elements.

Type
int

5.1.2 Elements module

It defines classes for all types for elements that can be included into the constitutive model, such as Spring, DislocationCreep, etc.

class Elements.**DislocationCreep**(*props*)

Bases: object

This class implements the necessary data and methods for the dislocation creep element (non-linear dashpot).

Parameters
props (*dict*) – Dictionary containing the material properties of a dislocation creep element. It includes the following parameters:

- A: [$\text{Pa}^{-1}\text{s}^{-1}$].
- n: [$-$].
- Q: Activation energy [J/mol].
- R: Universal gas constant [8.32 $\text{JK}^{-1}\text{mol}^{-1}$].
- T: Temperature [K].

property Aexp

List containing the term $A_{\text{exp}} = \exp\left(-Q/RT\right)$ for all grid elements.

Type
list

property B

A (nelems, 3, 3) tensor storing matrix $\mathbf{B}_{cr}$ for all grid elements.

Type
torch.Tensor

property G

A (nelems, 6, 6) tensor storing $\mathbb{G}_{cr}$ for all grid elements.

Type
torch.Tensor

compute_E(*stress_vec*)

This function computes $\frac{\partial \dot{\boldsymbol{\varepsilon}}_{ie}}{\partial \boldsymbol{\sigma}}$ by finite differences.

Parameters
stress_vec (*torch.Tensor*) – A (nelems, 3, 3) pytorch tensor storing the stress tensor for all grid elements.

compute_G_B(*stress_vec*, **args*)

This function computes matrices $\mathbb{G}_{ie}$ and $\mathbf{B}_{ie}$.

Note: In this case, $\mathbf{B}_{ie} = 0$ since the dislocation creep element does not depend on internal parameters.

Parameters
stress_vec (*torch.Tensor*) – A (nelems, 3, 3) pytorch tensor storing the stress tensor for all grid elements.

compute_eps_bar(*phi1*, *phi2*)

Computes $\bar{\boldsymbol{\varepsilon}}_{cr}^{k} = \boldsymbol{\varepsilon}_{cr}^{t} + \phi_1 \dot{\boldsymbol{\varepsilon}}_{cr}^{t} + \phi_2 \dot{\boldsymbol{\varepsilon}}_{cr}^{k}$.

Parameters

- **phi1** (*float*) – Quantity associated to time integration, $\phi_1 = \Delta t \theta$.
- **phi2** (*float*) – Quantity associated to time integration, $\phi_2 = \Delta t (1 - \theta)$.

compute_eps_ie(*stress*, *stress_k*, *phi2*)

Computes the dislocation creep strain as:

$$\boldsymbol{\varepsilon}_{cr} = \bar{\boldsymbol{\varepsilon}}_{cr}^{k} + \phi_2 \left[\mathbb{G}_{cr} : \left(\boldsymbol{\sigma}^{k+1} - \boldsymbol{\sigma}^{k} \right) - \mathbf{B}_{cr} \right]$$

Parameters

- **stress** (*torch.Tensor*) – A (nelems, 3, 3) pytorch tensor storing the stress of the **current** iteration level (i.e. $\boldsymbol{\sigma}^{k+1}$).
- **stress_k** (*torch.Tensor*) – A (nelems, 3, 3) pytorch tensor storing the stress of the **previous** iteration level (i.e. $\boldsymbol{\sigma}^{k}$).
- **phi2** (*float*) – Quantity associated to time integration, $\phi_2 = \Delta t (1 - \theta)$.

compute_eps_ie_rate(*stress_vec*, *return_eps_ie=False*, **args*)

Computes the dislocation creep strain rate as

$$\dot{\boldsymbol{\varepsilon}}_{cr} = A \exp\left(-\frac{Q}{RT} \right) q^{n-1} \mathbf{s}$$

where $\mathbf{s} = \boldsymbol{\sigma} - \frac{1}{3}\text{tr}(\boldsymbol{\sigma})\mathbf{I}$ is the deviatoric stress and $q = \sqrt{\frac{3}{2}\mathbf{s} : \mathbf{s}}$ denotes the von Mises stress.

Parameters

- **stress_vec** (*torch.Tensor*) – A (nelems, 3, 3) pytorch tensor storing the stress tensor for all grid elements.

- **return_eps_ie** (*bool*) – If *False* it stores $\dot{\boldsymbol{\varepsilon}}_{ie}$ to *eps_ie_rate*, otherwise the function returns as a value. This is used when computing $\frac{\partial \dot{\boldsymbol{\varepsilon}}_{ie}}{\partial \boldsymbol{\sigma}}$ by finite differences.

property eps_bar

A (nelems, 3, 3) tensor storing $\bar{\boldsymbol{\varepsilon}}_{cr}$ for all grid elements.

Type
torch.Tensor

property eps_ie

A (nelems, 3, 3) tensor storing $\boldsymbol{\varepsilon}_{ie}$ (i.e. at the **current** time level) for all grid elements.

Type
torch.Tensor

property eps_ie_old

A (n_elems, 3, 3) tensor storing $\boldsymbol{\varepsilon}_{cr}^{t}$ (i.e. at the **previous** time level) for all grid elements.

Type
torch.Tensor

property eps_ie_rate

A (n_elems, 3, 3) tensor storing $\dot{\boldsymbol{\varepsilon}}_{cr}$ for all grid elements.

Type
torch.Tensor

property eps_ie_rate_old

A (n_elems, 3, 3) tensor storing $\dot{\boldsymbol{\varepsilon}}_{cr}^{t}$ (i.e. at the **previous** time level) for all grid elements.

Type
torch.Tensor

increment_internal_variables(**args*)

The disclocation creep element does not depend on internal variables, so this function does nothing.

property n

List containing the material property n $(-)$ for all grid elements.

Type
list

update_eps_ie_old()

Updates dislocation creep strain at time t with the value of time $t + \Delta t$, that is, $\boldsymbol{\varepsilon}_{cr}^{t} \leftarrow \boldsymbol{\varepsilon}_{cr}^{t+\Delta t}$

update_eps_ie_rate_old()

Updates dislocation creep strain rate at time t with the value of time $t + \Delta t$, that is, $\dot{\boldsymbol{\varepsilon}}_{cr}^{t} \leftarrow \dot{\boldsymbol{\varepsilon}}_{cr}^{t+\Delta t}$

update_internal_variables(**args*)

The disclocation creep element does not depend on internal variables, so this function does nothing.

class Elements.**Spring**(*props*)

Bases: object

This class implements the necessary data and methods for a linear spring element.

Parameters
props (*dict*) – Dictionary containing the material properties of a linear elastic spring.

property C0

This is a (nelems, 6, 6) tensor storing the $\mathbb{C}_0$ matrix for each element of the grid.

Type
torch.Tensor

property C0_inv

This is a (nelems, 6, 6) tensor storing $\mathbb{C}_0^{-1}$ matrix for each element of the grid.

Type
torch.Tensor

property E

This list contains the Young's modulus for each element of the grid.

Type
list

compute_eps_e(*stress*)

This function computes the elastic strain as $\varepsilon_e = \mathbb{C}_0^{-1} : \boldsymbol{\sigma}$.

property eps_e

This is a (nelems, 3, 3) tensor storing the elastic strain tensor for all grid elements.

Type
torch.Tensor

initialize()

Initialize spring data associated to the linear spring, i.e., $\mathbb{C}_0$ and $\mathbb{C}_0^{-1}$ for each element of the grid.

Note: The quantity $\mathbb{C}_0$ is a 4th-order tensor, but due to Voigt notation it is represented as 6x6 matrix for each grid element.

---

property nu

This list contains the Poisson's ratio for each element of the grid.

Type
list

class Elements.Viscoelastic(*props*)

Bases: object

This class implements the necessary data and methods for a linear Kelvin-Voigt element (viscoelastic).

Parameters
props (*dict*) – Dictionary containing the material properties of a Kelvin-Voigt element. It includes Young's modulus and Poisson's ratio for the spring and a viscosity for the dashpot.

property B

A (nelems, 3, 3) tensor storing matrix $\mathbf{B}_{ve}$ for all grid elements.

Type
torch.Tensor

property C1

A (nelems, 6, 6) tensor storing $\mathbb{C}_1$ for all grid elements.

Type
torch.Tensor

property E

This list contains the Young's modulus for each element of the grid.

Type
list

property G

A (nelems, 6, 6) tensor storing $\mathbb{G}_{ve}$ for all grid elements.

Type
torch.Tensor

compute_G_B(*stress_vec*, *phi2*)

Compute matrices $\mathbb{G}_{ve}$ and $\mathbf{B}_{ve}$ for the Kelvin-Voigt element.

Note: Since the viscoelastic element is linear, the value of $\mathbb{G}_{ve}$ is constant, and it is given by $\mathbb{G}_{ve} = \left(\eta\mathbf{I} + \phi_2\mathbb{C}_1\right)^{-1}$.

Note: For a viscoelastic element, since it does not depend on internal parameters, we must have $\mathbf{B}_{ve} = 0$.

Parameters

- **stress_vec** (*torch.Tensor*) – This is a (nelems, 3, 3) storing the stress tensor for each grid element.
- **phi2** (*float*) – Quantity associated to the time integration, $\phi_2 = \Delta t(1-\theta)$.

compute_eps_bar(*phi1*, *phi2*)

Computes $\bar{\varepsilon}^k_{ve} = \varepsilon^t_{ve} + \phi_1\dot{\varepsilon}^t_{ve} + \phi_2\dot{\varepsilon}^k_{ve}$.

Parameters

- **phi1** (*float*) – Quantity associated to time integration, $\phi_1 = \Delta t\theta$.
- **phi2** (*float*) – Quantity associated to time integration, $\phi_2 = \Delta t(1-\theta)$.

compute_eps_ve(*stress*, *stress_k*, *phi2*)

Computes the viscoelastic strain as:

$$\varepsilon_{ve} = \bar{\varepsilon}^k_{ve} + \phi_2\left[\mathbb{G}_{ve} : \left(\boldsymbol{\sigma}^{k+1} - \boldsymbol{\sigma}^k\right) - \mathbf{B}_{ve}\right]$$

Parameters

- **stress** (*torch.Tensor*) – A (nelems, 3, 3) pytorch tensor storing the stress of the **current** iteration level (i.e. $\boldsymbol{\sigma}^{k+1}$).
- **stress_k** (*torch.Tensor*) – A (nelems, 3, 3) pytorch tensor storing the stress of the **previous** iteration level (i.e. $\boldsymbol{\sigma}^k$).
- **phi2** (*float*) – Quantity associated to time integration, $\phi_2 = \Delta t(1-\theta)$.

compute_eps_ve_rate(*stress_vec*, *phi1*, *return_eps_ve=False*)

Computes the viscoelastic strain rate as

$$\dot{\varepsilon}_{ve} = \mathbb{G}_{ve} : \left[\boldsymbol{\sigma} - \mathbb{C}_1 : \left(\varepsilon^t_{ve} + \phi_1\dot{\varepsilon}^t_{ve}\right)\right]$$

property eps_bar

A (nelems, 3, 3) tensor storing $\bar{\varepsilon}_{ve}$ for all grid elements.

Type
torch.Tensor

property eps_ve

A (nelems, 3, 3) tensor storing $\boldsymbol{\varepsilon}_{ve}$ (i.e. at the **current** time level) for all grid elements.

Type
torch.Tensor

property eps_ve_old

A (n_elems, 3, 3) tensor storing $\boldsymbol{\varepsilon}_{ve}^{t}$ (i.e. at the **previous** time level) for all grid elements.

Type
torch.Tensor

property eps_ve_rate

A (n_elems, 3, 3) tensor storing $\dot{\boldsymbol{\varepsilon}}_{ve}$ for all grid elements.

Type
torch.Tensor

property eps_ve_rate_old

A (n_elems, 3, 3) tensor storing $\dot{\boldsymbol{\varepsilon}}_{ve}^{t}$ (i.e. at the **previous** time level) for all grid elements.

Type
torch.Tensor

property eta

This list contains the viscosities associated to the Kelvin-Voigt dashpot for each element of the grid.

Type
list

increment_internal_variables(**args*)

The viscoelastic element does not depend on internal variables, so this function does nothing.

property nu

This list contains the Poisson's ratio for each element of the grid.

Type
list

update_eps_ve_old()

Updates viscoelastic strain at time t with the value of time $t + \Delta t$, that is, $\boldsymbol{\varepsilon}_{ve}^{t} \leftarrow \boldsymbol{\varepsilon}_{ve}^{t+\Delta t}$

update_eps_ve_rate_old()

Updates viscoelastic strain rate at time t with the value of time $t + \Delta t$, that is, $\dot{\boldsymbol{\varepsilon}}_{ve}^{t} \leftarrow \dot{\boldsymbol{\varepsilon}}_{ve}^{t+\Delta t}$

update_internal_variables(**args*)

The viscoelastic element does not depend on internal variables, so this function does nothing.

class Elements.**ViscoplasticDesai**(*props*)

Bases: object

This class implements the necessary data and methods for the viscoplastic model proposed by Desai (1987).

Parameters

props (*dict*) – Dictionary containing the material properties of viscoplastic model. It includes the following parameters:

- μ_1: [s^{-1}].
- N_1: [$-$].
- a_1: [MPa^{2-n}].
- η: [$-$].
- n: [$-$].
- β_1: [MPa^{-1}].
- β: [$-$].
- m: [$-$].
- γ: [$-$].
- σ_t: Tensile strength [MPa].
- α_0: Initial hardening parameter [$-$].

property B

property Fvp

property G

property H

property N_1

property P

property Q

property a_1

property alpha

property alpha_0

property beta

property beta_1

compute_E(*stress_vec*)

This function computes $\frac{\partial \dot{\boldsymbol{\varepsilon}}_{vp}}{\partial \boldsymbol{\sigma}}$ by finite differences.

Parameters

stress_vec (*torch.Tensor*) – A (nelems, 3, 3) pytorch tensor storing the stress tensor for all grid elements.

compute_Fvp(*alpha*, *I1*, *J2*, *Sr*)

Computes the yield function value according to

$$F_{vp}(\boldsymbol{\sigma}, \alpha) = J_2 - (-\alpha I_1^n + \gamma I_1^2) \left[\exp\left(\beta_1 I_1\right) - \beta S_r\right]^m$$

Parameters

- **alpha** (*torch.Tensor*) – A (nelems,) storing the hardening parameters for all grid elements.
- **I1** (*torch.Tensor*) – A (nelems,) storing I_1 for all grid elements.
- **J2** (*torch.Tensor*) – A (nelems,) storing J_2 for all grid elements.
- **Sr** (*torch.Tensor*) – A (nelems,) storing S_r for all grid elements.

Returns
Fvp – A (nelems,) storing the yield function values F_{vp} for all grid elements.

Return type
torch.Tensor

compute_G_B(*stress_vec*, *dt*)

This function computes matrices $\mathbb{G}_{vp}$ and $\mathbf{B}_{vp}$. Since the viscoplastic model depends on a internal parameter (α), these matrices are given by

$$\mathbb{G}_i = \frac{\partial \dot{\boldsymbol{\varepsilon}}_i}{\partial \boldsymbol{\sigma}} - \frac{1}{h_i}\frac{\partial \dot{\boldsymbol{\varepsilon}}_i}{\partial \alpha_i}\frac{\partial r_i}{\partial \boldsymbol{\sigma}} \quad \text{and} \quad \mathbf{B}_i = \frac{r_i}{h_i}\frac{\partial \dot{\boldsymbol{\varepsilon}}_i}{\partial \alpha_i}$$

where $i = vp$.

Note: Remember that $\mathbb{G}_{ie}$ is a 4th-order tensor represented as matrix by using Voigt notation.

Parameters

- **stress_vec** (*torch.Tensor*) – A (nelems, 3, 3) pytorch tensor storing the stress tensor for all grid elements.
- **dt** (*float*) – Time step size.

compute_H(*Q*, *P*)

This function computes the tensor product between $\mathbf{Q}$ and $\mathbf{P}$, both rank-2 tensors, which implies that it results in a rank-4 tensor. That is,

$$\mathbb{H} = \mathbf{Q} \otimes \mathbf{P} \quad \rightarrow \quad H_{ijkl} = Q_{ij}P_{kl}$$

This operation, however, is performed considering the symmetry of $\mathbf{Q}$ and $\mathbf{P}$, which allows us to represent $\mathbb{H}$ using Voigt notation.

Parameters

- **Q** (*torch.Tensor*) – A (nelems, 3, 3) pytorch tensor storing $\mathbf{Q}$ for all grid elements.
- **P** (*torch.Tensor*) – A (nelems, 3, 3) pytorch tensor storing $\mathbf{P}$ for all grid elements.

Returns
H – A (nelems, 6, 6) pytorch tensor storing $\mathbb{H}$ for all grid elements. Note that Voigt notation is employed here.

Return type
torch.Tensor

compute_eps_bar(*phi1*, *phi2*)

Computes $\bar{\boldsymbol{\varepsilon}}_{cr}^k = \boldsymbol{\varepsilon}_{cr}^t + \phi_1 \dot{\boldsymbol{\varepsilon}}_{cr}^t + \phi_2 \dot{\boldsymbol{\varepsilon}}_{cr}^k$.

Parameters

- **phi1** (*float*) – Quantity associated to time integration, $\phi_1 = \Delta t \theta$.

- **phi2** (*float*) – Quantity associated to time integration, $\phi_2 = \Delta t(1-\theta)$.

compute_eps_ie(*stress*, *stress_k*, *phi2*)

Computes the dislocation creep strain as:

$$\boldsymbol{\varepsilon}_{vp} = \bar{\boldsymbol{\varepsilon}}_{vp}^{k} + \phi_2 \left[\mathbb{G}_{vp} : \left(\boldsymbol{\sigma}^{k+1} - \boldsymbol{\sigma}^{k} \right) - \mathbf{B}_{vp} \right]$$

Parameters

- **stress** (*torch.Tensor*) – A (nelems, 3, 3) pytorch tensor storing the stress of the **current** iteration level (i.e. $\boldsymbol{\sigma}^{k+1}$).
- **stress_k** (*torch.Tensor*) – A (nelems, 3, 3) pytorch tensor storing the stress of the **previous** iteration level (i.e. $\boldsymbol{\sigma}^{k}$).
- **phi2** (*float*) – Quantity associated to time integration, $\phi_2 = \Delta t(1-\theta)$.

compute_eps_ie_rate(*stress*, *alpha=None*, *return_eps_ie=False*)

Computes the viscoplastic strain rate as

$$\dot{\boldsymbol{\varepsilon}}_{vp} = \mu_1 \left\langle \frac{F_{vp}}{F_0} \right\rangle^{N_1} \frac{\partial F_{vp}}{\partial \boldsymbol{\sigma}}$$

where $\langle \cdot \rangle$ is the ramp function.

Parameters

- **stress_vec** (*torch.Tensor*) – A (nelems, 3, 3) pytorch tensor storing the stress tensor for all grid elements.
- **alpha** (*torch.Tensor*) – A (nelems,) pytorch tensor storing the hardening parameters for all grid elements.
- **return_eps_ie** (*bool*) – If *False* it stores $\dot{\boldsymbol{\varepsilon}}_{ie}$ to *eps_ie_rate*, otherwise the function returns as a value. This is used when computing $\frac{\partial \dot{\boldsymbol{\varepsilon}}_{ie}}{\partial \boldsymbol{\sigma}}$ by finite differences.

compute_initial_hardening(*stress*, *Fvp_0=0.0*)

Computes the initial hardening parameter values such that $F_{vp} = F_{vp,0}$. This is done through the following expression:

$$\alpha_0 = \gamma I_1^{2-n} - I_1^{-n} J_2 \left[\exp\left(\beta_1 I_1\right) - \beta S_r \right]^{-m}$$

Note: Specifying $F_{vp} = 0$ means that all the elements are on the onset of viscoplastic deformation.

Parameters

- **stress** (*torch.Tensor*) – A (nelems, 3, 3) storing the stress tensor for all grid elements.
- **Fvp_0** (*float*) – Specified value for the yield function value in the entire domain.

compute_residue(*eps_rate*, *alpha*, *dt*)

Computes the residue of the hardening rule. In this case,

$$r_{vp}^{k} = \alpha^{k} - a_1 \left[\left(\frac{a_1}{\alpha_0} \right)^{1/\eta} + \xi^{k} \right]^{-\eta}, \quad \text{where} \quad \xi^{k} = \int_{t_0}^{t} \sqrt{\dot{\boldsymbol{\varepsilon}}_{vp}^{k} : \dot{\boldsymbol{\varepsilon}}_{vp}^{k}} \mathrm{dt}$$

Parameters

- **eps_rate** (*torch.Tensor*) – A (nelems, 3, 3) tensor storing $\dot{\varepsilon}_{vp}$ for all grid elements.
- **alpha** (*list*) – List containing the hardening parameter values for all grid elements.
- **dt** (*float*) – Time step size.

compute_stress_invariants(*s_xx*, *s_yy*, *s_zz*, *s_xy*, *s_xz*, *s_yz*)

This function compute the following stress invariants

$$I_1 = \sigma_{xx} + \sigma_{yy} + \sigma_{zz}$$

$$I_2 = \sigma_{xx}\sigma_{yy} + \sigma_{yy}\sigma_{zz} + \sigma_{xx}\sigma_{zz} - \sigma_{xy}^2 - \sigma_{yz}^2 - \sigma_{xz}^2$$

$$I_3 = \sigma_{xx}\sigma_{yy}\sigma_{zz} + 2\sigma_{xy}\sigma_{yz}\sigma_{xz} - \sigma_{zz}\sigma_{xy}^2 - \sigma_{xx}\sigma_{yz}^2 - \sigma_{yy}\sigma_{xz}^2$$

$$J_2 = \frac{1}{3} * I_1^2 - I_2$$

$$J_3 = \frac{2}{27} I_1^3 - \frac{1}{3} I_1 I_2 + I_3$$

$$S_r = -\frac{J_3\sqrt{27}}{2J_2^{1.5}}$$

$$I_1^* = I_1 + \sigma_t$$

Parameters

- **s_xx** (*torch.Tensor*) – A (nelems,) array storing σ_{xx} for all grid elements.
- **s_yy** (*torch.Tensor*) – A (nelems,) array storing σ_{yy} for all grid elements.
- **s_zz** (*torch.Tensor*) – A (nelems,) array storing σ_{zz} for all grid elements.
- **s_xy** (*torch.Tensor*) – A (nelems,) array storing σ_{xy} for all grid elements.
- **s_xz** (*torch.Tensor*) – A (nelems,) array storing σ_{xz} for all grid elements.
- **s_yz** (*torch.Tensor*) – A (nelems,) array storing σ_{yz} for all grid elements.

Returns

- **I1** (*torch.Tensor*) – A (nelems,) array storing I_1 for all grid elements.
- **I2** (*torch.Tensor*) – A (nelems,) array storing I_2 for all grid elements.
- **I3** (*torch.Tensor*) – A (nelems,) array storing I_3 for all grid elements.
- **J2** (*torch.Tensor*) – A (nelems,) array storing J_2 for all grid elements.
- **J3** (*torch.Tensor*) – A (nelems,) array storing J_3 for all grid elements.
- **Sr** (*torch.Tensor*) – A (nelems,) array storing S_r for all grid elements.
- **I1_star** (*torch.Tensor*) – A (nelems,) array storing I_1^* for all grid elements.

property eps_bar

property eps_ie

property eps_ie_old

property eps_ie_rate

property eps_ie_rate_old

property eta

extract_stress_components(*stress*)

This is a convenient function to extract the stress components.

Parameters

stress (*torch.Tensor*) – A (nelems, 3, 3) storing the stress tensor for all grid elements.

Returns

- **s_xx** (*torch.Tensor*) – A (nelems,) array storing σ_{xx} for all grid elements.
- **s_yy** (*torch.Tensor*) – A (nelems,) array storing σ_{yy} for all grid elements.
- **s_zz** (*torch.Tensor*) – A (nelems,) array storing σ_{zz} for all grid elements.
- **s_xy** (*torch.Tensor*) – A (nelems,) array storing σ_{xy} for all grid elements.
- **s_xz** (*torch.Tensor*) – A (nelems,) array storing σ_{xz} for all grid elements.
- **s_yz** (*torch.Tensor*) – A (nelems,) array storing σ_{yz} for all grid elements.

property gamma

property h

increment_internal_variables(*stress*, *stress_k*, *dt*)

Increment the hardening parameter by the following expression

$$\alpha_i^{k+1} \leftarrow \alpha_i^k + \delta\alpha_i \quad \text{where} \quad \delta\alpha_i = -\frac{1}{h_i}\left(r_i^k + \frac{\partial r_i^k}{\partial \boldsymbol{\sigma}} : \delta\boldsymbol{\sigma}\right).$$

where $i = vp$.

property m

property mu_1

property n

property qsi

property qsi_old

property r

property sigma_t

update_eps_ie_old()

Updates viscoplastic strain at time t with the value of time $t + \Delta t$, that is, $\boldsymbol{\varepsilon}_{vp}^t \leftarrow \boldsymbol{\varepsilon}_{vp}^{t+\Delta t}$

update_eps_ie_rate_old()

Updates viscoplastic strain rate at time t with the value of time $t + \Delta t$, that is, $\dot{\boldsymbol{\varepsilon}}_{vp}^t \leftarrow \dot{\boldsymbol{\varepsilon}}_{vp}^{t+\Delta t}$

update_internal_variables()

Updates the previous value of the accumulated viscoplastic strain with the current one, that is $\xi^t \leftarrow \xi^{t+\Delta t}$.

5.1.3 Equations module

Everything related to the solution of the linear momentum balance equation.

class Equations.**LinearMomentum**(*grid*, *theta*, *input_file*)

Bases: object

This class is intended to solve the linear momemum balance equation considering a general non-linear constitutive models.

Parameters

- **m** (safeincave.ConstitutiveModel.ConstitutiveModel) – Object containing the constitutive model data.
- **theta** (*int*) – Choice of the time integration method: explicit (theta=1.0), Crank-Nicolson (theta=0.5) or fully-implicit (theta=0.0).

apply_neumann_bc(*t*)

It reads all Neumann boundary conditions (external loads) applied to the geometry and builds the right-hand side vector.

Parameters
t (*float*) – Time level.

compute_CT(*dt*)

Computes the consistent tangent matrix $\mathbb{C}_T$, which is given by

$$\mathbb{C}_T = \left(\mathbb{C}_0^{-1} + \Delta t(1-\theta)\mathbb{G}_T\right)^{-1}$$

where $\mathbb{C}_0$ is the stiffness matrix associated to the linear spring, and $\mathbb{G}_T = \mathbb{G}_{ve,T} + \mathbb{G}_{ie,T}$.

Parameters
dt (*float*) – Time step size.

Return type
None

compute_GT_BT_ie(*dt*)

Computes matrices $\mathbb{G}_{ie,T}$ and $\mathbf{B}_{ie,T}$ for the inelastic elements. That is

$$\mathbb{G}_{ie,T} = \sum_{i=1}^{N_{ie}} \mathbb{G}_i \quad \text{and} \quad \mathbf{B}_{ie,T} = \sum_{i=1}^{N_{ie}} \mathbf{B}_i$$

where N_{ie} denotes the number of inelastic elements present in the constitutive model.

Parameters
dt (*float*) – Time step size.

Return type
None

compute_GT_BT_ve(*dt*)

Computes matrices $\mathbb{G}_T$ and $\mathbf{B}_T$ for the viscoelastic elements. That is

$$\mathbb{G}_T = \sum_{i=1}^{N_{ve}} \mathbb{G}_i \quad \text{and} \quad \mathbf{B}_T = \sum_{i=1}^{N_{ve}} \mathbf{B}_i$$

where N_{ve} denotes the number of viscoelastic (Kelvin-Voigt) elements present in the constitutive model.

Parameters
dt (*float*) – Time step size.

Return type
None

`compute_eps_bar`(*dt*)

Computes the eps_bar of **all** elements of the constitutive model.

Parameters
dt (*float*) – Time step size.

Return type
None

`compute_eps_bar_ie`(*dt*)

Computes the eps_bar of all **inelastic** elements of the constitutive model.

Parameters
dt (*float*) – Time step size.

Return type
None

`compute_eps_bar_ve`(*dt*)

Computes the eps_bar of all **viscoelastic** elements of the constitutive model.

Parameters
dt (*float*) – Time step size.

Return type
None

`compute_eps_e`()

Computes the strains of all **elastic** elements of the constitutive model.

Parameters
dt (*float*) – Time step size.

Return type
None

`compute_eps_ie`(*dt*)

Computes the strains of all **inelastic** elements of the constitutive model.

Parameters
dt (*float*) – Time step size.

Return type
None

`compute_eps_ie_rate`()

Computes strain rates of all inelastic elements of the constitutive model.

`compute_eps_rhs`(*dt*, **args*)

Computes the term $\boldsymbol{\varepsilon}_{rhs}^{k}$ of the linearized momentum equation, that is,

$$\boldsymbol{\varepsilon}_{\text{rhs}}^{k} = \bar{\boldsymbol{\varepsilon}}_{ne}^{k} - \Delta t\,(1-\theta)\left(\mathbb{G}_{ne} : \boldsymbol{\sigma}^{k} + \mathbf{B}_{ne}\right)$$

Parameters
dt (*float*) – Time step size.

Return type
None

`compute_eps_ve`(*dt*)

Computes the strains of all **viscoelastic** elements of the constitutive model.

Parameters
dt (*float*) – Time step size.

Return type
None

`compute_eps_ve_rate`(*dt*)

Computes the strain rates of the viscoelastic elements of the constitutive model.

Parameters
dt (*float*) – Time step size.

`compute_initial_hardening`(*verbose*)

`compute_stress`(*eps_tot*, *dt*)

Compute stress tensor as

$$\boldsymbol{\sigma}^{k+1} = \mathbb{C}_T : \left[\boldsymbol{\varepsilon}^{k+1} - \bar{\boldsymbol{\varepsilon}}_{ne}^{k} + \Delta t(1-\theta)\left(\mathbf{B}_{ne} + \mathbb{G}_{ne} : \boldsymbol{\sigma}^{k}\right)\right]$$

Parameters
eps_tot (*torch.Tensor*) – A (nelems, 3, 3) storing the total strain for all grid elements.

`compute_stress_C0`(*eps_e*)

Compute stress tensor as

$$\boldsymbol{\sigma} = \mathbb{C}_0 : \boldsymbol{\varepsilon}_e$$

Note: This operation considers that $\mathbb{C}_0$ is represented by Voigt notation.

Parameters
eps_e (*torch.Tensor*) – A (nelems, 3, 3) storing the elastic strain for all grid elements.

`define_dirichlet_bc`(*t*)

Defines a list of Dirichlet boundary conditions to be applied to the linear system.

Parameters
t (*float*) – Time level of the simulation.

Returns
bcs – List containing the Dirichlet boundary conditions.

Return type
list[dolfin.fem.dirichletbc.DirichletBC]

`define_solver`()

Defines the solver for solving the linear system according to the specifications in the input_file.json.

Returns
solver – The solver can be either an iterative solver (KrylovSolver) or a direct solver (LUSolver).

Return type
dolfin.cpp.la.KrylovSolver or dolfin.cpp.la.LUSolver

increment_internal_variables(*dt*)

Increment internal variables.

Parameters
dt (*float*) – Time step size.

initialize(*solve_equilibrium=False*, *verbose=True*, *save_results=False*)

save_solution(*t*)

solve(*t*, *dt*)

solve_equilibrium(*verbose=True*, *save_results=False*)

update_eps_ie_old()

Updates inelastic strains of the previous time level, that is $\boldsymbol{\varepsilon}_{ie}^{t} \leftarrow \boldsymbol{\varepsilon}_{ie}^{t+\Delta t}$.

update_eps_ie_rate_old()

Updates inelastic strain rates of the previous time level, that is $\dot{\boldsymbol{\varepsilon}}_{ie}^{t} \leftarrow \dot{\boldsymbol{\varepsilon}}_{ie}^{t+\Delta t}$.

update_eps_ve_old()

Updates viscoelastic strains of the previous time level, that is $\boldsymbol{\varepsilon}_{ve}^{t} \leftarrow \boldsymbol{\varepsilon}_{ve}^{t+\Delta t}$.

update_eps_ve_rate_old()

Updates viscoelastic strain rates of the previous time level, that is $\dot{\boldsymbol{\varepsilon}}_{ve}^{t} \leftarrow \dot{\boldsymbol{\varepsilon}}_{ve}^{t+\Delta t}$.

update_internal_variables()

Update internal variables with values of previous iteration, that is $\alpha_i^k \leftarrow \alpha_i^{k+1}$.

update_stress()

Updates stress of the previous interation, that is $\boldsymbol{\sigma}^k \leftarrow \boldsymbol{\sigma}^{k+1}$.

5.1.4 Grid module

It reads gmsh grids and gives access to relevant mesh information.

class Grid.**GridHandlerFEniCS**(*mesh*)

Bases: object

This class is responsible to read a FEniCS mesh of a brick shape (not necessarily a cube) and build the internal data structure in a convenient way.

Parameters

- **geometry_name** (*str*) – Name of the .geo file (usually *geom.geo*).
- **grid_folder** (*str*) – Path to where the geometry and grid are located.

build_boundaries()

Builds list of boundaries for faces EAST, WEST, NORTH, SOUTH, BOTTOM and TOP.

build_box_dimensions()

Reads box dimensions, that is, maximum *x*, *y* and *z* coordinates.

`build_dolfin_tags()`

Defines tags for boundaries (2D) and subdomains (3D).

`build_grid_dimensions()`

Reads the grid dimensions.

`build_subdomains()`

Builds subdomains, which in this case is just one.

`get_boundaries()`

Get mesh boundaries. It is used when applying Dirichlet boundary conditions.

Returns
boundaries – Mesh boundaries.

Return type
dolfin.cpp.mesh.MeshFunctionSizet

`get_boundary_names()`

Provides the names of all the boundaries.

Returns
boundary_names – List of strings containing the boundary names.

Return type
dict_keys

`get_boundary_tags(`*BOUNDARY_NAME*`)`

Get boundary tag (integer) corresponding to *BOUNDARY_NAME*.

Parameters
BOUNDARY_NAME (*str*) – Name of the boundary.

Returns
tag_number – Integer representing BOUNDARY_NAME

Return type
int

`get_subdomain_names()`

Provides the names of all the subdomains.

Returns
subdomain_names – List of strings containing the subdomain names.

Return type
dict_keys

`get_subdomain_tags(`*DOMAIN_NAME*`)`

Get subdomain tag (integer) corresponding to *DOMAIN_NAME*.

Parameters
DOMAIN_NAME (*str*) – Name of the subdomain.

Returns
tag_number – Integer representing DOMAIN_NAME

Return type
int

get_subdomains()

Get mesh subdomains. It can be used for solving different models in different subdomains.

Returns
subdomains – Mesh subdomains.

Return type
dolfin.cpp.mesh.MeshFunctionSizet

class Grid.GridHandlerGMSH(*geometry_name*, *grid_folder*)

Bases: object

This class is responsible to read a Gmsh grid and build the internal data structure in a convenient way.

Parameters

- **geometry_name** (*str*) – Name of the .geo file (usually *geom.geo*).
- **grid_folder** (*str*) – Path to where the geometry and grid are located.

build_box_dimensions()

Reads box dimensions, that is, maximum x, y and z coordinates.

build_tags()

Reads the mesh tags for lines (1D), surfaces(2D) and volumes (3D) and creates a dictionary associating the entity name to an unique integer.

get_boundaries()

Get mesh boundaries. It is used when applying Dirichlet boundary conditions.

Returns
boundaries – Mesh boundaries.

Return type
dolfin.cpp.mesh.MeshFunctionSizet

get_boundary_names()

Provides the names of all the boundaries.

Returns
boundary_names – List of strings containing the boundary names.

Return type
dict_keys

get_boundary_tags(*BOUNDARY_NAME*)

Get boundary tag (integer) corresponding to *BOUNDARY_NAME*.

Parameters
BOUNDARY_NAME (*str*) – Name of the boundary.

Returns
tag_number – Integer representing BOUNDARY_NAME

Return type
int

get_subdomain_names()

Provides the names of all the subdomains.

Returns
subdomain_names – List of strings containing the subdomain names.

Return type
dict_keys

`get_subdomain_tags`(*DOMAIN_NAME*)

Get subdomain tag (integer) corresponding to *DOMAIN_NAME*.

Parameters
DOMAIN_NAME (*str*) – Name of the subdomain.

Returns
tag_number – Integer representing DOMAIN_NAME

Return type
int

`get_subdomains`()

Get mesh subdomains. It can be used for solving different models in different subdomains.

Returns
subdomains – Mesh subdomains.

Return type
dolfin.cpp.mesh.MeshFunctionSizet

`load_boundaries`()

Renumbers boundaries (2D) tag numbers.

`load_mesh`()

Reads the .xml file containing the mesh.

`load_subdomains`()

Renumbers subdomain (3D) tag numbers.

`read_grid_dimensions`()

Reads the grid dimensions.

5.1.5 InputFileAssistant module

Useful class to assist building the input_file.json.

`class` InputFileAssistant.**BuildInputFile**

Bases: `object`

`add_boundary_condition`(*boundary_name*, *bc_data*)

`add_elastic_element`(*element_name*, *element_parameters*)

`add_element`(*element_name*, *element_parameters*, *element_type='Elastic'*)

`add_inelastic_element`(*element_name*, *element_parameters*)

`add_viscoelastic_element`(*element_name*, *element_parameters*)

`save_input_file`(*input_file_name*)

`section_body_forces`(*value*, *direction*)

`section_boundary_conditions`()

section_constitutive_model()

section_input_grid(*path_to_grid*, *grid_name='geom'*)

section_output(*output_folder*)

section_simulation(*simulation_settings*)

section_solver(*solver_settings*)

section_time(*time_list*, *theta=0.5*)

5.1.6 MaterialPointSimulator module

Material point model. This class is experimental and needs to be tested.

class MaterialPointSimulator.**MaterialPoint**(*input_bc*, *input_model*)

Bases: object

build_model(*input_model*)

compute_strains()

class MaterialPointSimulator.**MaterialPoint2**(*input_bc*, *input_model*)

Bases: object

build_model(*input_model*)

compute_strains()

5.1.7 ResultsHandler module

Useful to read vtk files and post-process results.

ResultsHandler.**convert_vtk_to_pandas**(*pvd_path*, *pvd_file*)

This function reads vtk files containing the time dependent *displacement* solution and convert it to pandas dataframes

Parameters

- **pvd_path** (*str*) – Path to the .pvd file, which must be at the same directory as the .vtk files.
- **pvd_file** (*str*) – Name of the .pvd file (usually *displacement.pvd*).

Returns

- **df_coord** (*pandas.core.frame.DataFrame*) – Spatial coordinates (x, y, z) of the grid nodes.
- **df_ux** (*pandas.core.frame.DataFrame*) – Component x of the displacement vector solution.
- **df_uy** (*pandas.core.frame.DataFrame*) – Component y of the displacement vector solution.
- **df_uz** (*pandas.core.frame.DataFrame*) – Component z of the displacement vector solution.

ResultsHandler.**read_scalar_from_cells**(*pvd_path*, *pvd_file*)

This function reads vtk files containing the time dependent solution of a scalar function defined on elements and convert it to pandas dataframes.

Parameters

- **pvd_path** (*str*) – Path to the .pvd file, which must be at the same directory as the .vtk files.
- **pvd_file** (*str*) – Name of the .pvd file (usually *displacement.pvd*).

Returns

- **df_coord** (*pandas.core.frame.DataFrame*) – Spatial coordinates (*x*, *y*, *z*) of the centroids of all grid elements.
- **df_scalar** (*pandas.core.frame.DataFrame*) – Scalar values at each grid element.

ResultsHandler.**read_tensor_from_cells**(*pvd_path*, *pvd_file*)

This function reads vtk files containing the time dependent solution of a rank-2 tensor function **A** defined on elements and convert it to pandas dataframes.

Parameters

- **pvd_path** (*str*) – Path to the .pvd file, which must be at the same directory as the .vtk files.
- **pvd_file** (*str*) – Name of the .pvd file (usually *displacement.pvd*).

Returns

- **df_coord** (*pandas.core.frame.DataFrame*) – Spatial coordinates (*x*, *y*, *z*) of the centroids of all grid elements.
- **df_sx** (*pandas.core.frame.DataFrame*) – Values of component A_{xx}.
- **df_sy** (*pandas.core.frame.DataFrame*) – Values of component A_{yy}.
- **df_sz** (*pandas.core.frame.DataFrame*) – Values of component A_{zz}.
- **df_sxy** (*pandas.core.frame.DataFrame*) – Values of component A_{xy}.
- **df_sxz** (*pandas.core.frame.DataFrame*) – Values of component A_{xz}.
- **df_syz** (*pandas.core.frame.DataFrame*) – Values of component A_{yz}.

ResultsHandler.**read_tensor_from_cells_old**(*pvd_path*, *pvd_file*)

This function reads vtk files containing the time dependent solution of a rank-2 tensor function **A** defined on elements and convert it to pandas dataframes.

Parameters

- **pvd_path** (*str*) – Path to the .pvd file, which must be at the same directory as the .vtk files.
- **pvd_file** (*str*) – Name of the .pvd file (usually *displacement.pvd*).

Returns

- **df_coord** (*pandas.core.frame.DataFrame*) – Spatial coordinates (*x*, *y*, *z*) of the centroids of all grid elements.
- **df_sx** (*pandas.core.frame.DataFrame*) – Values of component A_{xx}.
- **df_sy** (*pandas.core.frame.DataFrame*) – Values of component A_{yy}.
- **df_sz** (*pandas.core.frame.DataFrame*) – Values of component A_{zz}.
- **df_sxy** (*pandas.core.frame.DataFrame*) – Values of component A_{xy}.
- **df_sxz** (*pandas.core.frame.DataFrame*) – Values of component A_{xz}.
- **df_syz** (*pandas.core.frame.DataFrame*) – Values of component A_{yz}.

ResultsHandler.**read_vector_from_points**(*pvd_path*, *pvd_file*)

This function reads vtk files containing the time dependent *displacement* solution at grid nodes and convert it to pandas dataframes.

Parameters

- **pvd_path** (*str*) – Path to the .pvd file, which must be at the same directory as the .vtk files.
- **pvd_file** (*str*) – Name of the .pvd file (usually *displacement.pvd*).

Returns

- **df_coord** (*pandas.core.frame.DataFrame*) – Spatial coordinates (x, y, z) of the grid nodes.
- **df_ux** (*pandas.core.frame.DataFrame*) – Component x of the displacement vector solution.
- **df_uy** (*pandas.core.frame.DataFrame*) – Component y of the displacement vector solution.
- **df_uz** (*pandas.core.frame.DataFrame*) – Component z of the displacement vector solution.

5.1.8 Simulator module

The class implements the iterative process to solve the non-linear equilibrium equations.

class Simulator.**Simulator**(*input_file*)

Bases: object

This class is responsible to carry out the simulation according to the input_file.json. It loads the mesh, builds the constitutive model, builds the linear momenum equation, defines the solver, defines the weak formulation the run the simulation.

Parameters

input_file (*dict*) – Dictionary extracted from the JSON file.

run()

Runs transient simulation.

run_simulation(*solve_equilibrium=False*, *verbose=True*)

Runs simulation **without** solving the equilibrium condition.

Parameters

- **solve_equilibrium** (*bool*) – If **True**, it calculates the equilibrium conditions considering the elastic and viscoelastic (if present) elements only. If **False**, then it skips the equilibrium condition.
- **verbose** (*bool*) – Shows real time simulation info on screen.

5.1.9 Utils module

Useful functions used in safeincave.

Utils.**compute_C**(*n_elems*, *nu*, *E*)

Assemble the $\mathbb{C}$ matrices for each element of the grid.

Parameters

- **n_elems** (*int*) – Number of grid elements.
- **nu** (*list*) – List containing Poisson's ratio values for each grid element.
- **E** (*list*) – List containing the value of Young's modulus for each grid element.

Returns
C – Returns a (n_elems, 6, 6) containing matrices $\mathbb{C}$ for all grid element.

Return type
torch.Tensor

Utils.**dotdot**(*C*, *eps*)

Computes the double dot product between **C** (4th-order tensor in Voigt notation) and **eps** (2nd-order tensor in tensor notation). It first converts **eps** to Voigt notation, then it performs the *dot* product with **C**, and finally converts the resulting quantity to tensor notation.

Parameters

- **C** (*dolfin.function.function.Function*) – 4th-order tensor in Voigt notation.
- **eps** (*dolfin.function.function.Function*) – 2nd-order tensor in tensor notation.

Returns
tensor – Double dot product between **C** and **eps**.

Return type
ufl.tensors.ListTensor

Utils.**dotdot2**(*C_voigt*, *eps_tensor*)

This function performs the double dot product between a 4th-order tensor represented in Voigt notation and a 2nd-order tensor (without Voigt notation). The operation is performed by first applying the Voigt notation to the 2nd-order tensor, perform matrix-vector products to obtain the 2nd-order tensor in Voigt notation, and finally transform the resulting 2nd-order tensor to tensor notation.

Parameters

- **C_voigt** (*torch.Tensor*) – This is a pytorch tensor storing the 4th-order tensors for all grid elements. Since Voigt notation is used, C_voigt has dimension (n_elems, 6, 6).
- **eps_tensor** (*torch.Tensor*) – This is a pytorch tensor storing 2nd-order tensors for all grid elements using tensor notation. Therefore, its dimensions are (n_elems, 3, 3).

Returns
stress_torch – A tensor of dimensions (n_elems, 3, 3) resulting from the double dot product.

Return type
torch.Tensor

Utils.**epsilon**(*u*)

It computes the strain tensor based on the displacement field **u**, that is,

$$\boldsymbol{\varepsilon}(\mathbf{u}) = \frac{1}{2}\left(\nabla\mathbf{u} + \nabla\mathbf{u}^T\right)$$

Parameters
u (*dolfin.function.function.Function*) – Displacement field.

Returns
grad_u – Symmetric gradient of **u**.

Return type
ufl.tensoralgebra.Sym

Utils.**local_projection**(*tensor*, *V*)

Utils.**numpy2torch**(*numpy_array*)

It properly converts a numpy array into a pytorch tensor.

Parameters
numpy_array (*numpy.ndarray*) – Numpy array to be converted.

Returns
torch_array – A pytorch tensor with the same dimension as *numpy_array*.

Return type
torch.tensor

Utils.**read_json**(*file_name*)

This is just a convenient function to read json files.

Parameters
file_name (*str*) – Full path to json file, including file name, e.g. to/folder/file.json.

Returns
data – A dictionary containing the json data.

Return type
dict

Utils.**save_json**(*data*, *file_name*)

This is just a convenient function to save dictionaries into json files.

Parameters

- **data** (*dict*) – Dictionary containing the data to be saved in the json file.
- **file_name** (*str*) – Full path to where the json file is supposed to be saved,including file name, e.g. to/folder/file.json.

Utils.**tensor2voigt**(*e*)

Converts tensor notation to Voigt notation.

Parameters
e (*dolfin.function.function.Function*) – A 2nd-order tensor in tensor notation.

Returns
e_voigt – A 2nd-order tensor in Voigt notation.

Return type
ufl.tensors.ListTensor

Utils.**voigt2tensor**(*s*)

Converts a tensor from Voigt notation to tensor notation.

Parameters
s (*ufl.tensors.ListTensor*) – A 2nd-order tensor in Voigt notation.

Returns
s_tensor – A 2nd-order tensor in tensor notation.

Return type
ufl.tensors.ListTensor

BIBLIOGRAPHY

[1] CS Desai and A Varadarajan. A constitutive model for quasi-static behavior of rock salt. *Journal of Geophysical Research: Solid Earth*, 92(B11):11445–11456, 1987.

[2] Kavan Khaledi, Elham Mahmoudi, Maria Datcheva, and Tom Schanz. Stability and serviceability of underground energy storage caverns in rock salt subjected to mechanical cyclic loading. *International journal of rock mechanics and mining sciences*, 86:115–131, 2016.

PYTHON MODULE INDEX