gruner

Geotechnical Simulations with Moose

– DRAFT –

This whitepaper collects information and code
for geotechnical simulations with the software Moose.

Jörg Meier

Gruner AG, Geotechnical Engineering
Basel, Switzerland

16.10.2024

Contents

Chapter 1

Introduction

The open-source, parallel finite element framework Moose (mooseframework.inl.gov) can also be used to perform simulations that are used in the geotechnical design practice. This document aims to collect options and techniques that are suitable for creating such geotechnical simulations with Moose.

In the geotechnical planning practice, finite element models are often used for deformation prognosis. Furthermore, such FE models can help to identify complex failure mechanisms. The requirements for these models are accordingly a sufficiently good representation of the geotechnical structure including the subsoil, the groundwater conditions, pre-existing structures and similar. These FE models simulate the construction process over time. As this design process is generally intended to take place in a controlled and quasi-static form, the associated simulations are usually also transient and quasi-static. Therefore, scope and main focus of this document are transient quasi-static simulations.

This document is primarily intended as a collection of information rather than a textbook or step-by-step guide. As such, it is intended to be a 'living' document that will be continually edited and updated.

Chapter 2

Model Configuration

2.1 Strain calculation

In the 'solid_mechanics' module, Moose supports three different types of strain calculation:

- Small Linearized Total Strain
- Incremental Small Strains
- Finite Large Strains

Listing 2.1: Setting up incremental small strains within the Physics/SolidMechanics block

```
[Physics]
  [SolidMechanics]
    [QuasiStatic]
      [./all]
        strain = SMALL
        incremental = true
        ⋮
      []
    []
  []
[]
```

2.2 Stress calculation

Stresses.html

The stress calculation chosen by the user must be compatible with the strain calculation (section 2.1).

ComputeFiniteStrainElasticStress for incremental and finite strain formulations.

2.3 Fixation of the model boundaries

Fix the model boundaries (usually all boundaries are only fixed in the direction of the normal and the lower boundary can also be fixed horizontally).

(to be written)

Chapter 3

Geometry

3.1 General considerations on geometry

When modelling geotechnical problems, it is often desirable to make a meaningful geometric abstraction to avoid time-consuming modelling, but also to avoid inaccuracies due to oversimplification. However, experience shows that the effort involved in creating such a model geometry should not be underestimated and that the models consist of a large number of individual parts and components - often called 'geometrical objects'. Some of these objects have trivial shapes (e.g. rectangular surfaces for a sheet pile wall segment), others can have non-trivial curvilinear boundaries (e.g. geological bodies). All of these objects must be shaped to fit each other (without gaps or overlaps) and together form the FE model. Usually, geotechnical models are managed (e.g materials, staging, etc) on the level of these objects.

3.2 'Subdomains' resp. 'Blocks'

Geometrically, a Moose model consists of one or more 'subdomains'. Due to the use of libmesh by Moose, the term 'block' is also used as a synonym for 'subdomain' in many places. Element types and Moose objects such as materials, kernels, etc. are assigned to these subdomains in turn. Furthermore, each FE element belongs to exactly one subdomain.

It is therefore necessary to create at least one subdomain for each element type and each material. To simplify the handling of construction stages, it has also proved useful to further subdivide element sets into individual subdomains so they match the 'geometrical objects' mentioned before so they can to be used independently during the simulation (e.g. deactivated or activated at different times). This document follows this approach that Moose models are managed on the basis of subdomains/blocks.

As the shape of these individual parts and components directly influences the shape of the FE elements they are build of, further prerequisites must be considered during geometrisation: For instance, as few 'unfavourably shaped' (e.g. too acute-angled) FE elements as possible should result in order to avoid numerical instabilities or load increments that are too small. A common reason for acute-angled elements and large numbers of elements are corner or end points that are close together and unfavourable intersections of the objects. Another requirement for geometrisation is that the number of FE elements should remain within a range that enables acceptable calculation times.

Listing 3.1: Read mesh from a file

```
[Mesh]
    [file]
        type = FileMeshGenerator
        file = "source.msh"
    []
    second_order = true
[]
```

Listing 3.2: Fragment of an MSH file containing 'lower dimensional blocks'

```
$MeshFormat
4.1 0 8
$EndMeshFormat
$PhysicalNames
⋮
0 1 "PointA"                                    ← sideset (node)
1 2 "topleftedge"                               ← sideset (line)
1 3 "lineSignPole000" lower_dimensional_block   ← block with beam-elements
1 4 "lineSignPole001_lower_dimensional_block"   ← block with beam-elements
2 5 "BoundaryZMin"                              ← sideset (surface)
2 6 "Sign" lower_dimensional_block              ← block with shell-elements
⋮
```

3.3 Getting the mesh into Moose

Although Moose's 'Mesh System' provides a variety of tools for generating FE meshes on the basis of input file commands, the creation of geometrically complex models with these commands is often confusing and error-prone.

Therefore, the mesh including all subdomains and FE-elements is prepared outside of Moose and imported using the `FileMeshGenerator`. The MSH file format has proven itself in this context. One possible workflow is to create the geometry of the subdomains in Blender, then generate the mesh using gmsdh and save it as an MSH file. See listing Listing 3.1 for the code required in a Moose input file for the import of an MSH file.

3.4 Lower dimensional blocks in MSH files

A notable quirk of the MSH file format when used with Moose is the definition of lower dimensional blocks. The term 'lower dimensional blocks' is used here to describe subdomains with FE elements with a lower dimensionality (e.g. shells and beams in a model with 3D solid elements). The MSH file format makes no internal distinction as to whether blocks with lower dimensional elements are to be interpreted as a sideset or as a 'real' element block. To introduce this distiction Moose recognizes the `lower_dimensional_block` codeword (although this codeword is not an official part of the MSH file format). If this codeword is found in the line of an entry in the `$PhysicalNames`, Moose assumes that this block does not represent a sideset, but contains lower dimensional elements. As shown in Listing 3.2 the codeword may be just appended at the end of the line (without beeing part of the name of the physical name) or even be pat of the physical name itself.

Subdomain type	Element type	Comments
volume	TET10	tetrahedral element with 10 nodes and 4 integration points
surface (e.g. shell)	TRI6	triangular element with 6 nodes
line (e.g. beam)	EDGE3	line element with 3 nodes

Table 3.1: Commonly used compatible element types for tetrahedral/triangular discretisation

Listing 3.3: Moose input file fragment: variable holding several block names and fake-user

```
[Mesh]
    # Variable holding names of all blocks with volumes
    Volumes = 'Base Column Suzanne'
[]

[Functions]
    # Fake user for the variable above
    [FakeUser_Volumes]
        type = ParsedFunction
        expression = 'a'
        symbol_names = 'a'
        symbol_values = '1'
        control_tags = ${Mesh/Volumes}
    []
[]
```

3.5 Choice of element types

For obvious reasons, the element types of the individual subdomains must be compatible with each other (e.g. the number and distribution of nodes must match, etc). Given the often complex geometry of geotechnical models (see section 3.1), elements with triangular faces are often preferred. In this case, volume elements are correspondingly tetrahedral (TET) and surface elements are triangular (TRI). However, since linear TET and TRI elements (TET4 and TRI3) have very poor numerical properties (e.g. unrealistically high stiffness, locking), 'quadratic' elements (TET10 and TRI6) must be used. This leads to the commonly used compatible element types for tetrahedral/triangular discretisation show in Table 3.1.

Currently, Moose has full support for TET10 elements, but not for EDGE3 and TET6 elements. EDGE3 and TRI6 elements can currenty be used in a Moose-app by cloning the following Moose code-plugins as a 'contrib':

- beams: github.com/Kavan-Khaledi/moose-codeplugin-beam
- shells: github.com/Kavan-Khaledi/moose-codeplugin-shell

3.6 Groups of geometrical objects

Moose does not explicity support groups geometrical objects but allows to create variables in the input file holding names of several objects on one hand and to use these variables directly for block-restriction (e.g. assignment of materials).

Due to the fact that Moose throws an error if unused variables are found in the input file, but the user may want to define groups even if they are not (currently) used, it has proven useful to add 'fake users' using a ParsedFunction as shown in Listing 3.3.

Chapter 4

Modelling of geometric entities, structures, loads, etc.

4.1 Overview

This chapter collects information on how different types of geometric entities, structures, loads etc. can be modelled. Table 4.1 contains a list of common geotechnical components and associated options that can be used for modelling them.

4.2 Entities

4.2.1 Volumes

To model a volume, such as a soil volume, concrete volume, etc., the following is required:

- subdomain: A subdomain of FE-elements of an appropriate element type (e.g. TET10).
- materials: Assignment of Moose materials defining the constitutive behaviour.

4.2.2 FixedEndSprings

Fixed end springs are currently not directly supported by Moose. As a workaround one can define a spring between nodes (subsection 4.2.3) and fix the other node.

CoupledForceNodalKernel

pamm.202200045

4.2.3 Springs between nodes (NodeToNodeAnchors)

(to be written)

mastodon: LinearSpring

4.2.4 Beams (line element)

To model beams, the following is required:

- subdomain: A subdomain of FE-elements of an appropriate element type (e.g. EDGE3).
- materials: Assignment of Moose materials defining the constitutive behaviour of the beam.

Entity	Options for Modelling	Reference
soil volume	• cluster of volume elements	sec. 4.2.1
excavation pit wall	• shell elements	sec. 4.2.5
	• cluster of volume elements in case of a 'thick' wall (e.g. slurry wall, bored pile wall)	sec. 4.2.1
concrete volume	• cluster of volume elements	sec. 4.2.1
	• shell elements in case of a 'thin' walls	sec. 4.2.5
sprayed concrete	• shell elements	sec. 4.2.5
pile, soil nail	• cluster of volume elements in case of a pile with large diameter (e.g. bored pile)	sec. 4.2.1
prop	• spring between nodes	sec. 4.2.3
	• fixed end spring	sec. 4.2.2
	• beam	sec. 4.2.4
ground anchor	• spring between nodes + (embedded) beam	sec. 4.2.3

Table 4.1: Selected geotechnical entities and their respective modelling

Currently, Moose has no support for appropriate beam elements and materials for EDGE3. In your MooseApp, the beam material for EDGE may be included as an 'contrib' by cloning into github.com/Kavan-Khaledi/moose-codeplugin-beam

4.2.5 Shell (surface element)

To model shells (or "plates"), the following is required:

• subdomain: A subdomain of FE-elements of an appropriate element type (e.g. TRI6).

• materials: Assignment of Moose materials defining the constitutive behaviour of the shell.

Currently, Moose has no support for appropriate shell elements and materials for TRI6. In your MooseApp, the shell material for TRI6 may be included as an 'contrib' by cloning into github.com/Kavan-Khaledi/moose-codeplugin-shell

4.2.6 Interfaces (surface element)

(to be written)

4.2.7 Contact

(to be written)

4.2.8 Pore Pressure Boundaries

(to be written)

Chapter 5

Initial Conditions

5.1 Initial conditions for Moose objects

For definition of the initial (starting) conditions for the variables a Moose simulation the ICs system is to be used.

(to be written)

5.2 Gravitation and initial stress state

The initial stress state of geotechnical simulations is often non-trivial. This is because the section of geosphere to be modelled often has a history of millions of years under different loading conditions, including gravity and tectonical processes. In addition, the Earth's surface is often non-horizontal, so that stress trajectories near the surface are oriented accordingly. Furthermore, the initial stress state may be anisotropic with a specific orientation. Gravity and the initial stress state are normally taken into account by means of:

- Definition of appropriate displacement boundary conditions for the model (see section 2.3).
- Activation of gravitational body force (see subsection 5.2.1).
- Definition of eigenstrains from the inital stress field (e.g. using ComputeEigenstrainFromInitialStress, see subsection 5.2.1 and subsection 5.2.3).

5.2.1 Gravity

To consider gravity, the following aspects must be included in the model:

- When compiling the MooseApp: As the effect of gravity is taken into account in interaction with the material density, the Moose module `MISC` must be active in addition to the Moose module `SOLID_MECHANICS`.
- Gravity must be activated in the input file. This is done by adding a special kernel of the type `Gravity` as shown in Listing 5.1. The direction and strength of the gravitational field is to be defined here.
- The materials must be assigned a material density (Listing 5.2).

Listing 5.1: Gravity kernel in a Moose inut file

```
[Kernels]
    [gravity]
        type = Gravity
        use_displaced_mesh = false
        variable = disp_z
        value = -9.81
    []
[]
```

Listing 5.2: Assignment of a density to subdomain 'block1'

```
[Materials]
    [undrained_density]
      type = GenericConstantMaterial
      block = 'Block1'
      prop_names = density
      prop_values = 0.0025
    []
[]
```

5.2.2 Initial stress state using ComputeEigenstrainFromInitialStress

This initial stress state corresponds to the 'geostatic stress state' in the uninfluenced homogeneous half-space (or a section thereof) under the effect of gravity, which can be represented analytically in a stress point as follows:

$$\sigma_{ini,vert} = \sigma_{ini,vert,h=0} + \gamma h \tag{5.1}$$

$$\sigma_{ini,horiz} = \sigma_{ini,vert} K_0 \quad \text{with} \quad K_0 = 1 - \sin\varphi \tag{5.2}$$

In these equations, $\gamma = \rho g$ corresponds to the weight of the soil, h to the overburden height, $\sigma_{ini,vert,h=0}$ to the vertical stress at $h = 0$ and K_0 to the earth pressure coefficient. This earth pressure coefficient is usually estimated as a function of the angle of internal friction φ (simplified equation according to Jaky).

Listing 5.3 shows the definition of a geostatic initial stress state using two functions `ini_xx_yy` and `ini_zz` and their subsequent use in a `ComputeEigenstrainFromInitialStress`.

5.2.3 Initial stress state for anisotropic stresses

Align the model to the orientation of the initial stresses

(to be written)

Listing 5.3: Definition of a geostatic initial stress state using 'ComputeEigenstrainFromInitialStress'

```
[Functions]
    [ini_xx_yy]
      type = ParsedFunction
      vars = 'sig_top z_top  rho     g    k0 '
      vals = '-1.5    0      0.0025  10   0.3'
      value = '(sig_top - rho * g * (z_top - z)) * k0'
    []
    [ini_zz]
      type = ParsedFunction
      vars = 'sig_top z_top  rho     g'
      vals = '-1.5    0      0.0025  10'
      value = '(sig_top - rho * g * (z_top - z))'
    []
[]

[Materials]
    [strain]
      type = ComputeIncrementalSmallStrain
      volumetric_locking_correction=false
      eigenstrain_names = ini_stress
    []
    [ini_stress]
      type = ComputeEigenstrainFromInitialStress
      eigenstrain_name = ini_stress
      initial_stress = 'ini_xx_yy 0 0 0 ini_xx_yy 0 0 0 ini_zz'
    []
[]
```

Chapter 6

Construction Stages

6.1 General comments on construction stages

The type of geotechnical simulation primarily discussed in this document aims to model a construction process. Accordingly, such a simulation must reproduce the various stages of construction over time. Since this document assumes that Moose models are organised on the basis of subdomains (section 3.2), the state changes also refer primarily to adjustments that affect entire subdomains.

The most common state changes in this context are:

- Activating subdomains (e.g. installation of sheet piles is usually modelled by activating the corresponding subdomain)
- Deactivating subdomains (e.g. excavation of a soil volume is usually modelled by deactivating the corresponding subdomain)
- Replacing subdomains (e.g. installation of a slurry wall modelled via volume elements leads to the need to replace the soil volume with the volume representing the slurry wall)
- Adjusting external loads (e.g. loads are used to model various influences. As these influences change over time, the corresponding loads have to be adjusted)
- Adjusting prestress (e.g. for props and stand anchors the prestress is to be adjusted)

As a consequence of the Moose input file concept, these status adjustments must be triggered at various points in an input file if Moose is used 'out of the box'. To provide a more centralised way of defining construction stages, the `[Stages]` MooseApp code plugin linked below can be used:

github.com/jmeier/moose-codeplugin-stages

This document primarily describes the procedure for the definition of construction stages, with the use of the `[Stages]` code plugin where applicable. The procedure without this plugin is only mentioned in selected places.

6.2 Activating and deactivating subdomains

ToDo: Ramp up / ramp down material properties

(to be written)

6.3 Replacing subdomains

(to be written)

6.4 Adjusting external loads

(to be written)

6.5 Adjusting prestress

(to be written)

Chapter 7

Material Models for Soil and Rock

7.1 Preliminary remarks on the material models

The built-in library of material models suitable for modelling soil and rock is currently very limited.

7.2 NEML2

Moose supports the use of materials defined by means of the external material modeling library NEML2 (Messner and Hu). Details on the integration of NEML2 into Moose are not scope of this document and can be found in the corresponding corresponding section of the Moose documentation.

7.3 Mohr-Coulomb (linear-elastic ideal-plastic)

(to be written)

Chapter 8

Proven Patterns

8.1 Partial input files

When a Moose model is defined as an 'input file' (usually with the file extension '*.i'), the size of such an input file can quickly grow to several hundred lines, even without the model geometry. Editing and troubleshooting becomes correspondingly confusing. Moose therefore also supports 'partial input files', where other input files can be called up from a 'central' input file. For example, the material parameters or variables with groups of geometrical objects could be maintained in a separate file.

The recommented option is to use the `!include` keyword within the input file. This option is shown in Listing 8.1. Please note, that the neither path not filename are allowed to contain a newline character, `#` character, or `[` character.

Listing 8.1: Include anpther input file

```
!include "path/to/file.i"
```

The alternative and not recommented option is to call a Moose app with several input files as arguments. These input files are interpreted by simply concatenating them. This functionality was included in Moose for test purposes, but is not suitable for productive models, as this call may not be not documented and may not be reproducible:

```
./moose-opt -i input1.i -i input2.i -i input3.i
```

8.2 Physical units

In Moose, all numerical values must be entered in a system of units which is consistent in itself - but which is selected by the user. Without additional options, the user must enter a number in the input file without a unit being explicitly labelled.

In addition, Moose - like other FE programmes - calculates internally with a finite precision. To avoid rounding errors and inaccuracies in the numerical calculation, it is therefore important that the numerical values entered are not too large. With a careful choice of the system of physical units, the user can avoid such numerical problems.

From the user's point of view, the challenge is to enter the numerical values in the 'correct' system of physical units everywhere in the input file. This conversion and the non-trivial verifiability is a source of error that should not be underestimated, particularly in the case of physical units with a time reference. Experience shows, that this is a frequent source of problems!

To mitigate this source of error, Moose offers the 'MooseUnits' unitily. In the input file this utility allows to convert the unit 'on the fly'. The input file fragment shown in Listing 8.2 converts $1000 \times 10^3\,\mathrm{kN/m^2}$ into $\mathrm{MN/m^2}$ and assigns the result to the variable `E`.

Listing 8.2: Converting $1000 \times 10^3\,\mathrm{kN/m^2}$ into $\mathrm{MN/m^2}$

```
E = ${units 1000E3 kN/m^2 -> MN/m^2}
```

In connection with variables holding the desired physical units for the model this utility can be used to enforce a consistent system of physical units in the input file and the user can simultaneously use the physical units of his source data (for better verifiability). Listing 8.3 shows the use of a variable used to hold the desired model unit for lengths.

Listing 8.3: Using a variable to hold the physical unit for lengths

```
# variable holding the pyhsical unit for lengths
modelunit_length = 'm'

# conversion of 10 mm into the desired model unit (m)
a = ${units 10 mm -> ${modelunit_length}}
```

A full set of pysical units can be setup as shown in Listing 8.4. If not all variables are used in the input file, Moose will issue an error. This can be avoided by adding fake-users to the variables.

Listing 8.4: Set of consistent physical units in a Moose inut file

```
# model units
modelunit_length = 'm'
modelunit_time = 's'
modelunit_mass = 'kg' # Mg = tons; Gg = kilotons

# derived units (this may be moved into a !include)
modelunit_force = ${raw ${modelunit_mass} * ${modelunit_length} / ${modelunit_time} ^ 2}
modelunit_pressure = ${raw ${modelunit_force} / ${modelunit_length} ^ 2}
modelunit_acceleration = ${raw ${modelunit_length} / ${modelunit_time} ^ 2}
modelunit_density = ${raw ${modelunit_mass} / ${modelunit_length} ^ 3}

# Fake userss for the variables above
[Functions]
    [FakeUser_modelunit_force]
        type = ParsedFunction
        expression = 'a'
        symbol_names = 'a'
        symbol_values = '1'
        control_tags = ${modelunit_force}
    []
    ⋮
[]
```

8.3 Moose code plugins

If MooseApps are extended with new MooseObjects that should be shared with other users, e.g. in workgroups, the need arises to be able to transfer this functionality to other MooseApps as a plug-in. Preferably without having to manually copy and adapt many code files. Furthermore, it is often good if there is independent source code management or versioning. The idea is to realise such code plugins for MooseApps as git submodules.

To address this need and provide versioning via a git submodule, Moose code plugins to be found in the following link can be used: github.com/jmeier/mooseapp-code-plugin

Chapter 9

Postprocessing

9.1 Paraview

Directly use the ExodusII-files produced by Moose.

(to be written)

9.2 Blender using the BVtkNodes addon

The BVtkNodes addon for Blender that wraps the Visualization Toolkit (VTK) library for scientific visualization in Blender.

(to be written)

9.3 Directly accessing ExodusII files

The primary output format of Moose is ExodusII. Accessing the data directly in this format is therefore a logical step. To work with ExodusII-files under Windows, one might want to compile the corresponding code in the seacas-repo under Windows. The step by step instructions on how to compile ExodusII for windows can be found here: seacas#375.

(to be written)