Fracture Model for Fluid Saturated Geomaterials Implemented Via a Poro-Elasto-Plastic Cohesive Surface Finite Element

John D. Sweetser
University of Colorado at Boulder, john.sweetser@colorado.edu

Follow this and additional works at: https://scholar.colorado.edu/mcen_gradetds
Part of the Civil Engineering Commons, and the Mechanical Engineering Commons

Recommended Citation
https://scholar.colorado.edu/mcen_gradetds/54

This Thesis is brought to you for free and open access by Mechanical Engineering at CU Scholar. It has been accepted for inclusion in Mechanical Engineering Graduate Theses & Dissertations by an authorized administrator of CU Scholar. For more information, please contact cus scholaradmin@colorado.edu.
Fracture Model for Fluid Saturated Geomaterials
Implemented Via a Poro-Elasto-Plastic Cohesive Surface
Finite Element

by

John D. Sweetser

A.S., Colorado Mountain College, 2008
B.S., University of Colorado, Boulder, 2012

A thesis submitted to the
Faculty of the Graduate School of the
University of Colorado in partial fulfillment
of the requirements for the degree of
Masters of Science
Department of Mechanical Engineering

2012
This thesis entitled:
Fracture Model for Fluid Saturated Geomaterials Implemented Via a Poro-Elasto-Plastic Cohesive Surface Finite Element
written by John D. Sweetser
has been approved for the Department of Mechanical Engineering

Assoc. Prof. Richard Regueiro

Assist. Prof. Jianliang Xiao

Date __________________

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.
Fractures involving poromechanical fluid flow are known to exhibit a strong coupling between the behavior of the solid skeleton and the pore fluid. Though there are other ways to model this poromechanical coupling, in this thesis, fractures are modeled discretely using a quadratic, 6-noded, poromechanical cohesive surface element (CSE). The CSE has specialized behavior in both the way that fluid flow and solid mechanics are modeled. The mechanical behavior of the CSE, when activated, will go through various stages as the fracture occurs. Further, the CSE models fluid flow along the fracture (longitudinal flow) as well as fluid flow exchange between the bulk porous material and cohesive surface element (transverse flow). Specialized tools are developed using MATLAB’s object oriented functionality to allow for rapid generation of meshes and the associated initial conditions/boundary conditions. These tools also allow the mesh to be plotted and updated as the solution progresses. Example problems are presented to demonstrate the behavior of the poromechanical CSE as well as the close hydromechanical coupling.
Dedication

To my parents, Dave and Vicky, for all their encouragement, support, wisdom, and guidance, which has been instrumental in getting me to where I am today.
Acknowledgements

This thesis would not have been possible without the support of many people and institutions. I would like to acknowledge and thank Sandia National Laboratories for funding my masters and for all the help, support, and guidance I’ve received throughout the last 2 years. I am immensely indebted to many of my colleagues.

Further, I would like to express my deepest gratitude and appreciation to my advisor, Assoc. Prof. Rich Regueiro, for the countless hours of mentoring and guidance over the last two years. It has truly been an honor working with you.

I would also like to thank my committee members, Dr. Joe Bishop and Assist. Prof. Jianliang Xiao, for their essential feedback on my thesis. I am grateful for all the support I’ve received from the mechanical engineering department and the college of engineering.
Contents

Chapter

1 Introduction

1.1 Overview ................................................................. 1
1.2 Background ............................................................... 2

2 Model

2.1 Mechanics ............................................................... 7
2.2 Poromechanics ........................................................... 7
2.3 Bulk Poromechanical Finite Element ................................. 7
   2.3.1 Formulation and Poromechanics ............................... 8
2.4 Cohesive Surface Finite Element .................................. 10
   2.4.1 Behavior of Cohesive Surface Finite Element ............... 10
   2.4.2 Cohesive Surface Element Variables and Parameters .......... 14
   2.4.3 Formulation ......................................................... 17
   2.4.4 Balance of linear momentum: elastostatics, quasistatic ........ 20
   2.4.5 Balance of Mass ................................................... 26
   2.4.6 Components of Consistent Tangent ............................ 29

3 Numerical Implementation ............................................. 31

3.1 Solution Methods ....................................................... 31
   3.1.1 Fully Coupled (Monolithic) ..................................... 31
### 3.1.2 Staggered (partitioned) .................................................. 33

### 3.2 Finite Element Utilities Design in MATLAB ............................. 33

#### 3.2.1 Object Oriented Programming ......................................... 33

#### 3.2.2 Performance ................................................................. 36

### 4 Examples .............................................................................. 38

#### 4.1 Comparison of a 64 bulk element mesh with and without cohesive surface elements .................................................. 38

##### 4.1.1 Example with applied traction and no cohesive surface elements .................................................. 38

##### 4.1.2 Example with applied traction and 8 vertically oriented cohesive surface elements .................................................. 42

##### 4.1.3 Example - Comparison of a mesh with cohesive surface elements against the same mesh without cohesive surface elements .................................................. 45

#### 4.2 Example - 64 bulk elements, 8 vertically oriented cohesive surface elements (Mesh 16) with a specified volume flux of fluid injected .................................................. 49

### 5 Discussion ........................................................................... 55

#### 5.1 Meshing ........................................................................... 55

##### 5.1.1 Other Mesh Considerations ............................................ 55

##### 5.1.2 Dynamic Insertion ......................................................... 56

##### 5.1.3 Mesh Dependence ........................................................ 56

##### 5.1.4 Random Voronoi Mesh .................................................. 56

### 6 Conclusions and Future Work .................................................. 59

### Bibliography ............................................................................ 60

### Appendix .................................................................................. 63

#### A Problem Definitions ............................................................ 63
B  MATLAB Implementation and Source Code 67

B.1 Important Variable Definitions .............................................. 68
B.2 Main Script ........................................................................ 71
B.3 General Data Handling, Manipulating, and Plotting ................. 87
  B.3.1 Node Definition Class ....................................................... 87
  B.3.2 Element Manager ............................................................ 98
  B.3.3 Element Class to Manage Each Individual Element ............ 104
  B.3.4 All Info Class to Allow Access and Storage of Important Parameters 113
B.4 Problem Definition Format and Problem Definitions ............... 119
  B.4.1 Problem Definition Manager .............................................. 119
  B.4.2 Problem Initialization and Operations ................................ 121
  B.4.3 Problem Definition Standard Format and Operations .......... 125
  B.4.4 Problem Definition 14 ...................................................... 128
  B.4.5 Problem Definition 140 .................................................... 131
  B.4.6 Problem Definition 16 ...................................................... 133
B.5 Bulk Element Functions ....................................................... 136
B.6 Cohesive Surface Element Functions ..................................... 161
Tables

Table

4.1 Parameters .................................................. 39
Figures

Figure

2.1 Comparison of linear vs. quadratic meshes ........................................ 6
2.2 Bulk Element Diagram ................................................................. 9
2.3 Diagram of cohesive surface element ................................................. 11
2.4 Plot of the plastic yield function. The inequality holds for all values within the above curve. ................................................................. 12
2.5 A 3 element mesh with the CSE inactive. ........................................... 13
2.6 Cohesive surface element behavior under a prescribed displacement ......... 15
2.7 Cohesive surface element confining pressure behavior .......................... 16
2.8 Diagram of continuum domain for formulation ..................................... 19
2.9 Cohesive surface element diagram for derivation ................................. 24
3.1 MATLAB Implementation Structure ................................................. 34
4.1 Diagram of Mesh 14 - 64 bulk and no vertically oriented cohesive surface elements .............................................................. 39
4.2 Mesh 140 - Displacement vs. Time .................................................... 40
4.3 Mesh 140 - Pore Fluid Pressure vs. Time ........................................... 41
4.4 Diagram of Mesh 14 - 64 bulk and 8 vertically oriented cohesive surface elements .............................................................. 43
4.5 Mesh 14 - Pressure vs. Time ............................................................ 43
4.6 Mesh 14 - Displacement vs. Time ...................................................... 44
4.7 Displacement - Mesh 14 vs. Mesh 140 .............................................. 45
4.8 Pressure at Node 137 - Mesh 14 vs. Mesh 140 ................................. 46
4.9 Pressure at Node 1 - Mesh 14 vs. Mesh 140 ................................. 48
4.10 Mesh 16 Diagram .................................................................... 50
4.11 Pressure at Various Depths From Injection ............................... 51
4.12 Aperture with Time .................................................................. 52
4.13 Pressure Distribution Within Mesh ............................................. 53
4.14 72 Element Horizontal Mesh Results ........................................... 54

5.1 Diagram of hypothetical mesh to demonstrate cases of mesh dependence .. 56
5.2 Illustration of Random Voronoi Mesh ............................................. 58

A.1 Diagram of Mesh 14 - 64 bulk and 8 vertically oriented cohesive surface elements .. 64
A.2 Diagram of Mesh 140 - 64 bulk and 0 vertically oriented cohesive surface elements .. 65
A.3 Mesh 16 Diagram .................................................................... 66
Chapter 1

Introduction

1.1 Overview

Due to their vast range of applications, there is significant interest in the use of cohesive surface elements (CSEs) in the field of fracture modeling using finite element analysis. Properly implemented, CSE’s are applicable for determining reservoir behavior for carbon sequestration problems, as shown in much of the literature [Iding and Ringrose, 2010, Segura and Carol, 2010]. CSE’s have also been used to model groundwater flow [Woodbury and Zhang, 2001] and have even been used to determine the maximum reservoir height a dam can support [Carol, 2011]. While there exists other methods for modeling fracture (e.g. smeared crack approach [Carol et al., 1997]), CSEs are used to model fracture discretely [Regueiro and Yu, 2010, Segura and Carol, 2008a, Segura and Carol, 2008b]. As a result, they must be present for fracture to occur. These elements are placed where a fracture/joint is, or where a fracture/joint is likely to occur. If the fracture does not exist at the start of the simulation, the CSE has a thickness of zero. Thus, the joint is double noded. Once the fracture forms at a specific junction, the CSE’s function is to model the behavior of the fracture. This is further discussed in Section 2.4.

Fluid flow coupled to solid skeleton deformation through porous media and fractures is a situation commonly encountered in many areas of geomechanics, such as hydraulic fracturing and water retention [Segura and Carol, 2004]. As a result, there has already been much research in adapting CSEs to also account for fluid flow [Segura and Carol, 2004]. In these situations, there exists a close poro-mechanical coupling between the pore fluid pressure and bulk solid skeleton
element displacement, as noted by [Segura and Carol, 2008a]. That is, as cracks nucleate and propagate, fluid flow into the fractures (which typically have a higher hydraulic conductivity than the surrounding porous material) can influence the nucleation and propagation of further fracture behavior. Simply stated, the flow through a fracture can affect the properties of the fracture which, in turn, can affect the fluid flow.

Many models already exist to simulate the creation and propagation of cracks containing fluid flow in a porous medium as a continuum, i.e. the smeared approach. In this thesis, however we will focus on a discrete approach because it has several important advantages, such as the ability to model fracture between different materials, as shown in [Regueiro and Yu, 2010]. The model developed also has the capabilities to model deformation in both the bulk porous material as well as the pressure sensitive CSE. Finally, the model can also simulate fully saturated poromechanical fluid flow in both the bulk element and CSE.

1.2 Background

Over the past decade, significant research has been conducted in the area of modeling fracture in geomaterials using cohesive zone models with poromechanical fluid flow. To accomplish this, various approaches for both developing and solving the governing equations have been used. Extra complication is added to solve for both solid skeleton deformation and poromechanical fluid flow due to the previously-mentioned hydromechanical coupling.

In fracture modeling, two main approaches are used - discrete fracture modeling and the smeared crack approach (macroscopic modeling). The smeared crack approach approximates the crack at the macro scale level. That is, this method approximates the nature of the crack or cracks across the entire mesh by modifying the bulk element’s behavior to account for the fractures. A drawback of the smeared approach is that only one bulk material can be used, and significant hydro-mechanical coupling is difficult to accurately model in the fracture or joint. As a result, only discrete fracture modeling will be considered here.

Though discrete fracture modeling is advantageous to other methods in many ways, it is
important to point out that there are some drawbacks. In order for a fracture to occur, a CSE must be present. For this reason, the solution can be considered mesh-dependent. For example, if a solution involves a fracture at a specific location and if no CSE is present, no fracture is able to occur. For this reason, it is important to make sure that the meshing is carefully considered. This is discussed further in Section 5.1.3. One solution for this is dynamic insertion. In this method, the mesh is allowed to be altered after the start of the simulation. Once a specified criterion is met at a specific facet, a CSE is dynamically inserted, as demonstrated in [Camacho and Ortiz, 1996]. This is discussed further in Section 5.1.

Due to the issues arising from a lack of CSEs where they are required, it may be tempting to place CSEs at every facet in the mesh. Unfortunately, this method also has drawbacks. CSEs can be computationally expensive to implement. Accordingly, placing these elements at every bulk element junction can lead to significantly increased simulation time. As mentioned previously, the CSE is said to not be activated when the fracture has not yet occurred. Due to the mechanical properties of the CSE, the bulk behavior may still be affected. With higher densities of CSEs, another problem is encountered. An abnormal behavior consisting of oscillations can occur when the parameters for the CSE are within a certain range. This is due to the elastic compliance of the element and can be reduced by decreasing the amount of CSEs, dynamically inserting the CSEs as needed, or by modifying the parameters [Rene Kregting, 2005].

Within this area of research, there are different ways to model the solid skeleton deformation as well as the pore fluid flow. Though all of the following models solve similar problems, they can be distinguished by three main factors:

1. Behavior of solid skeleton (Section 2.3)
2. Behavior of pore fluid (and coupling to solid skeleton, Section 2.3)
3. Solution algorithm and method of coupling solid skeleton mechanics and pore fluid flow (e.g. monolithically, staggered, coupled, Section 3.1).

**Solid Skeleton** As noted by [Anandarajah, 2010], nearly all materials are porous at some
scale. However, it needs to be determined if the effects from the porosity need to be considered in order to accurately model the material’s behavior. That is, will the porosity of a material effect its macroscopic behavior.

**Fluid Flow** In previous work, fluid flow modeling within a fracture has been implemented primarily using two different methods - the finite element method and the finite difference method. In a 2004 paper, [Segura and Carol, 2004] demonstrated the behavior of a CSE that allows both longitudinal fluid flow as well as transverse fluid flow. Solving for conservation of mass resulted in Equation 1.1 [Segura and Carol, 2004].

\[
\frac{dq_1}{dx} + s + q^- + q^+ = \frac{1}{\rho} \frac{\partial (\rho w)}{\partial t}
\] (1.1)

- \(q_1\): Fluid flow flow along element (longitudinal)
- \(q^+\): Fluid flow into fracture from top bulk element
- \(q^-\): Fluid flow into fracture from bottom bulk element
- \(w\): Aperture of the fracture
- \(\rho\): fluid mass density

In the majority of cases, the governing fluid flow equations are influenced by Darcy’s law (such as in [Segura and Carol, 2008b]). However, others exists using models such as lubrication theory (such as in [Sarris and Papanastasiou, 2011]).

**Solution Algorithm and Coupling** In the literature, two methods are employed to solve the hydromechanical coupled problem - the staggered and monolithic approaches.

As the above aspects are derived for the purposes of this thesis in subsequent sections, previous work in each area is discussed as well as the reasoning behind choosing the current model.
Chapter 2

Model

Modeling of the poro-elastic continuum is accomplished using a 9-noded quadratic bulk element while the fracture is modeled using a 6-noded quadratic pressure sensitive cohesive surface element (CSE). The element is formulated to account for both transverse fluid flow (flow into or out of the bulk element) and longitudinal fluid flow (flow along the fracture). When no fracture is present, the cohesive surface element has a thickness of zero and is said to be inactive (from a fluid flow perspective). Theoretically, when the cohesive surface element is not activated, its presence should not alter the mesh’s behavior. However, as discussed below, this is not always true. Once the fracture forms, the CSE may gain some finite thickness, as discussed in section 2.4. Further, as an initial condition, a preexisting aperture may be defined with a specified thickness.

The current quadratic model consisting of a 9 node bulk element and 6 node CSE is built upon an existing single phase (solid) model [Yu, 2010] which utilizes 4 node bulk elements and 4 node interface elements. This model does not take into account poromechanical fluid flow. In adding poromechanics, it became necessary to increase the order of the solid skeleton displacement shape functions beyond those of the pore fluid pressure shape functions. When an undrained condition is encountered, this keeps the solution from locking up. Figure 2.1 compares the linear vs quadratic implementations.

Before discussing the model’s equations in further detail, it is important to understand the applicability and assumptions of the model. For the purposes of this thesis, it will be assumed that two phases are present, a solid skeleton (bulk) material and a pore fluid that exists within the bulk
Figure 2.1: Linear vs. Quadratic. The CSE has been artificially expanded for visualization

**Linear** (left) - Existing model with 4 node bulk elements (elements 1 & 3) and a 4 node CSE in the center (element 3)

**Quadratic** (right) - Implemented model with 9 node bulk elements and a 6 node CSE
porous material and within any preexisting/newly-formed fractures. Further, it will be assumed that the domain is fully saturated with fluid; that is, only one fluid phase is present (no additional fluid/gas phase).

Based on the above assumptions, the applicability of the model can now be discussed. There exist cases with a saturated porous material that can be modeled neglecting the effects of the pore fluid, i.e., incorporating solid mechanics only. This primarily occurs in two situations [Anandarajah, 2010].

(1) When loading is applied slowly allowing the pore fluid sufficient time to seep out (drained)

(2) When the pore fluid does not have a path to escape (undrained), the pore fluid’s behavior may be incorporated into the bulk element’s properties.

In Section 2.3, the existing bulk element formulations are discussed and expanded to quadratic elements that incorporate poromechanics. In Section 2.4, a similar process is applied to the interface element. Section 3 discusses the numerical solution method and implementation in object oriented MATLAB. Finally, Section 4 demonstrates examples of the presented model. Appendices are included with further detail on the variables, MATLAB code, and more detail on each problem being solved.

2.1 Mechanics

2.2 Poromechanics

2.3 Bulk Poromechanical Finite Element

To model the poro-elastic bulk material, a mixed Q9P4 element is implemented (see figure 2.2). The solid skeleton displacement \( \mathbf{u}^h \) is interpolated quadratically (see Equations 2.1 and 2.2) while the pore fluid pressure uses bilinear shape functions.

Where,

\[
\mathbf{u}^h(\xi, \eta, t) = \sum_{a=1}^{9} N_a^u(\xi, \eta) \mathbf{d}_a(t)
\]  

(2.1)
Where $N_a^u(\xi, \eta)$ are the nodal shape functions and $d_a(t)$ is the nodal solid skeleton displacement vector (see Equation 2.3).

$$d_a(t) = \begin{bmatrix} d_x^{(a)}(t) \\ d_y^{(a)}(t) \end{bmatrix}$$  \hspace{2cm} (2.3)

### 2.3.1 Formulation and Poromechanics

As mentioned previously, the bulk elements were extended from a previously implemented ([Yu, 2010]) linear (4-noded) element to a quadratic (9-noded) element (see Figure 2.1). Accordingly, the bulk element solid skeleton displacement shape functions are given in Equation 2.2.

Though the solid skeleton displacement shape functions are quadratic utilizing all 9 nodes, the pore fluid pressure is found at the four corner nodes. The bilinear pore fluid pressure shape functions are derived in Equation 2.4.

$$p^h_w(\xi, \eta, t) = \sum_{b=1}^{4} N_b^f(\xi, \eta)\theta_b(t)$$  \hspace{2cm} (2.4)

Where $\theta_b(t)$ is the nodal fluid pressure, and $N_b^f$ is the bilinear shape function of local coordinates $(\xi, \eta)$ for pore fluid pressure.
Figure 2.2: A diagram of the local node numbering and local coordinate system $(\xi, \eta)$ for the 9 node bulk element. Pore fluid pressure is interpolated using bilinear shape functions at nodes 1, 2, 3, and 4.
2.4 Cohesive Surface Finite Element

The presented CSE has a few significant mechanical characteristics, including pressure sensitivity (friction) and softening. These are now illustrated with equations and examples applied to the mesh in Figure 2.5. Further, porous fluid flow is ignored for the purpose of this discussion.

The cohesive zone model is defined by equation 2.6 (the plastic yield function) to be pressure sensitive through the normal component of traction to the surface $T_n$. Pressure sensitivity, and in particular, an increase in fracture strength in the presence of confining pressure is supported by previous work ([Yu, 2010, Sarris and Papanastasiou, 2011]). First, the normal traction ($T_n$) and tangential traction ($T_t$) are calculated so that the plastic yield function can be related to the local coordinate system of the CSE.

The yield function along the CSE is written as:

$$F = \sqrt{T_t^2 + (c + \chi\tan(\phi))^2} - (c - T_n\tan(\phi)) \leq 0$$

2.5 Pressure shape functions

\[
\begin{align*}
N_1^p(\xi, \eta) &= \frac{1}{4}(1 - \xi)(1 - \eta) \\
N_2^p(\xi, \eta) &= \frac{1}{4}(\xi + 1)(1 - \eta) \\
N_3^p(\xi, \eta) &= \frac{1}{4}(\xi + 1)(\eta + 1) \\
N_4^p(\xi, \eta) &= \frac{1}{4}(1 - \xi)(\eta + 1)
\end{align*}
\]
Figure 2.3: An expanded view of the local node numbering and local coordinate system for the 6 node cohesive surface element. Note: The nodes (I, II, III) in the center are fictitious.
Where $F$: the plastic yield function

$\chi$: tensile strength

c: cohesion

$\phi$: friction angle of the bulk material

A plot of this yield function with parameters given in Table 4.1, is shown in Figure 2.4.

![Plot of the plastic yield function](image)

Figure 2.4: Plot of the plastic yield function. The inequality holds for all values within the above curve.

**Force-Displacement**

The cohesive surface element’s traction-displacement behavior is now demonstrated. The setup consists of two bulk elements and one CSE with a horizontal confining pressure (discussed later, parameters given in table 4.1). A downward vertical displacement is prescribed until a maximum deflection is reached. At this point, the displacement is moved back to its starting position. The reaction vertical force across the top three nodes is shown in Figure 2.6. The stages the CSE goes through are:
Figure 2.5: A 3 element mesh with the CSE inactive.
(1) **Phase 1** A linear elastic increase in the reaction force (as compressive vertical displacement is increased) up to the yield strength of the cohesive surface element.

(2) **Phase 2** An exponential decrease in the reaction force as the cohesive surface element yields and softens [Yu, 2010].

(3) **Phase 3** At this point, the prescribed displacement begins to travel back to the starting position. As a result, the system unloads elastically.

(4) **Phase 4** A constant force opposes the movement back to the initial position.

**Pressure Sensitivity**

To demonstrate the influence of the cohesive surface element’s pressure sensitivity, two test cases are presented, each utilizing one bulk element on the top and one bulk element on the bottom with a CSE inserted on the bulk element’s interface (Figure 2.5). In one case, a confining pressure of 10kPa was prescribed in the horizontal direction while in the other case, no confining pressure was present. A downward displacement across the top three nodes is prescribed (x-axis) while the vertical force across the top nodes is measured (y-axis). As shown in Figure 2.7, the case with the confining pressure has a significantly higher yield strength. This is formulated into the cohesive surface element’s behavior in Equation 2.6.

The plastic potential function (Equation 2.7) for the cohesive zone model [Yu, 2010] is:

\[
G = \sqrt{T_l^2 + (c + \chi \tan(\psi))^2} - (c - T_n \tan(\psi))
\] (2.7)

Where  
- \( G \) : the plastic potential function  
- \( \chi \) : tensile strength  
- \( \psi \) : the dilation angle  
- \( c \) : cohesion  

**2.4.2 Cohesive Surface Element Variables and Parameters**

The CSE is considered to have an elastic stiffness in both the normal \( k_n \) and tangential \( k_t \) directions. This is shown in the diagonal \( k \) matrix (see Equation 2.12).
Figure 2.6: The reaction force over the top three nodes (y-axis) is plotted against the prescribed vertical displacement of the top nodes (x-axis). The path traveled is clockwise, starting with linear elastic loading, an exponential decrease in force, elastic unloading, and a return back to the initial position under constant force. The displacement is a triangular function applied in the downward direction at the top nodes (see mesh in Figure 2.5)
Figure 2.7: A 3 element mesh subject to confining pressure in the horizontal direction is compared to the same mesh without confining pressure (mesh shown in Figure 2.5). The yield strength decreases dramatically when no confining pressure is present. This is due to the plastic yield function (Equation 2.6) being pressure sensitive through $T_n$ and friction $\tan(\phi)$.
\[
\mathbf{k} = \begin{bmatrix}
k_t & 0 \\
0 & k_n
\end{bmatrix}
\]  

(2.12)

The material parameters, which control the rate of softening, are shown below in Equation 2.13 ([Yu, 2010]).

\[
\begin{aligned}
\alpha\chi \\
\alpha_c \\
\alpha\phi \\
\alpha\psi
\end{aligned}
\]

(2.13)

2.4.3 Formulation

The cohesive surface element has six nodes. Each node has two solid skeleton displacement degrees of freedom (2D). The solid skeleton displacement varies quadratically while the pore fluid pressure varies linearly between each of the four corner nodes, as shown in 2.19.

\[
\begin{aligned}
N_1^p(\xi) &= \frac{1}{2}\xi(\xi - 1) \\
N_2^p(\xi) &= \frac{1}{2}\xi(\xi + 1) \\
N_3^p(\xi) &= \frac{1}{2}(1 - \xi^2) \\
N_4^p(\xi) &= \frac{1}{2}\xi(\xi + 1) \\
N_5^p(\xi) &= \frac{1}{2}\xi(\xi - 1) \\
N_6^p(\xi) &= \frac{1}{2}(1 - \xi^2)
\end{aligned}
\]

(2.14)

\[
[[\mathbf{u}^h(\xi)]] = -\mathbf{u}^{h+}(\xi) + \mathbf{u}^{h-}(\xi) = -\sum_{a=1}^{3} N_a(\xi) \mathbf{d}_a^e + \sum_{a=4}^{6} N_a(\xi) \mathbf{d}_a^e
\]

(2.15)


\[
[u^h(\xi, t)] = \begin{bmatrix}
-N_1 & -N_2 & -N_3 & N_4 & N_5 & N_6
\end{bmatrix}
\]

(2.16)

\[
N_a(\xi) = \begin{bmatrix}
N_a(\xi) & 0 \\
0 & N_a(\xi)
\end{bmatrix}
\]

(2.17)

\[
d^e_a(t) = \begin{bmatrix}
d^e_{x(a)}(t) \\
d^e_{y(a)}(t)
\end{bmatrix}
\]

(2.18)

The pore fluid pressure shape functions are shown below in equation 2.19

\[
\begin{align*}
N_1^p(\xi) &= \frac{1}{2}(1 - \xi) \\
N_2^p(\xi) &= \frac{1}{2}(\xi + 1) \\
N_4^p(\xi) &= \frac{1}{2}(\xi + 1) \\
N_5^p(\xi) &= \frac{1}{2}(1 - \xi)
\end{align*}
\]

(2.19)

The virtual mid-plane can be interpolated as follows

\[
x^h(\xi) = N_I(\xi)x_I + N_{II}(\xi)x_{II} + N_{III}(\xi)x_{III}
\]

(2.20)

where

\[
x_I = \frac{1}{2}(x_1 + x_5)
\]

(2.21)

\[
x_{II} = \frac{1}{2}(x_2 + x_4)
\]

\[
x_{III} = \frac{1}{2}(x_3 + x_6)
\]

(2.22)
Figure 2.8: Diagram of a solid with a discontinuity $S$. An example fixed displacement boundary condition ($g$) is shown on the left, applied over $\Gamma_g$. A traction of $t^\sigma$ is applied on the right over the area $\Gamma_t$. The area is divided by a fracture $S$. 
Where,

\[ x_1 = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} \]  \hspace{1cm} (2.23)

\[ \frac{\partial x^h}{\partial \xi} = \begin{bmatrix} \frac{\partial x^h}{\partial \xi} \\ \frac{\partial y^h}{\partial \xi} \end{bmatrix} \]  \hspace{1cm} (2.24)

### 2.4.4 Balance of linear momentum: elastostatics, quasistatic

To derive the governing equations, balance of linear momentum is first stated in Equation 2.25.

\[
(S) \begin{cases}
\frac{\partial \sigma_{ij}}{\partial x_j} + \rho b_i = 0 & x_i \in \Omega \\
u_i = g_i & x_i \in \Gamma_g \\
_{ij} \nu_j = t_i' & x_i \in \Gamma_t \\
^+_{ij} n_j^+ = T_i'^+ & x_i \in S^+ \\
^-_{ij} n_j^- = T_i'^- & x_i \in S^-
\end{cases}
\]  \hspace{1cm} (2.25)

Assume effective stress principle [Anandarajah, 2010, Terzaghi, 1943]

\[ \sigma_{ij} = \sigma'_{ij} - p_f \delta_{ij} \]  \hspace{1cm} (2.26)

Where \( \sigma_{ij} \) is the total stress, \( \sigma'_{ij} \) is the effective stress, and \( p_f \) is the pore fluid pressure. Further, \( p_f = -\frac{1}{3n} tr(\sigma') \) (positive in compression) and the mean effective stress is positive in tension [Anandarajah, 2010]. We assume that the fluid is isotropic and can be approximated as nearly inviscid.

Let \( n_i = n_i^+ = -n_i^- \) (for small rotations) or \( n_i = \frac{1}{2}(n_i^+ - n_i^-) \) for large rotations with \( S \) (the mid-plane), thus the effective traction is

\[ T_i' = T_i'^+ = -T_i'^+ \]  \hspace{1cm} (2.27)
Assume the discontinuity pore fluid pressure $p^S_f = \frac{1}{2}(p_f^+ + p_f^-)$ on $S$ is averaged from the values on the facets of the adjacent bulk elements. Next, applying the method of weighted residuals, let $w_i = \delta u_i$ be the weighting function or variation of $u_i$, where we have $w_i = \delta u_i = \delta g_i = \delta g_i^u = 0$ on $\Gamma_g$

$$\int_\Omega w_i \left( \frac{\partial \sigma_{ij}}{\partial x_j} + \rho b_i \right) dv = 0$$

(2.28)

Where $w_i$ is the test function, $b_i$ is the gravity acceleration vector. Applying the chain rule to Equation 2.28 results in Equation 2.29 as

$$\frac{\partial (w_i \sigma_{ij})}{\partial x_j} = \frac{\partial w_i}{\partial x_j} \sigma_{ij} + w_i \frac{\partial \sigma_{ij}}{\partial x_j}$$

(2.29)

$$w_i \frac{\partial \sigma_{ij}}{\partial x_j} = \frac{\partial (w_i \sigma_{ij})}{\partial x_j} - \frac{\partial w_i}{\partial x_j} \sigma_{ij} + \sigma_{ij}' - p_f \delta_{ij}$$

(2.30)

Where by the Divergence Theorem,

$$\int_\Omega \frac{\partial (w_i \sigma_{ij})}{\partial x_j} dv = \int_{\Gamma} w_i \sigma_{ij} \tilde{n}_j da$$

(2.31)

To relate the functions within the domain to the (known or applied) values at the surface, the divergence theorem is applied in Equation 2.31. (See Figure 2.8 for a definition of the surfaces)

$$\int_{\Gamma} w_i \sigma_{ij} \tilde{n}_j da = \int_{\Gamma_t} w_2 \sigma_{ij} \nu_j da + \int_{\Gamma_g} w_i \sigma_{ij} \nu_j da + \int_{S^+} w_i^+ \sigma_{ij}^+ \nu_j^+ da + \int_{S^-} w_i^- \sigma_{ij}^- \nu_j^- da$$

(2.32)

Where $\tilde{n}_j$ is the generic unit normal vector. The second term in Equation 2.32 goes to zero for a fixed displacement on the boundary $\Gamma_g$.

Apply Effective Stress Principle on traction vector

\[\begin{align*}
\sigma_{ij} &= \sigma_{ij}' - p_f \delta_{ij} \\
\sigma_{ij}^+ &= \sigma_{ij}^{+'} - p_f^+ \delta_{ij} \\
\sigma_{ij}^- &= \sigma_{ij}^{-'} - p_f^- \delta_{ij}
\end{align*}\]

(2.33)

Substituting for $\sigma_{ij}$ using equation 2.33 results in:
\[
\int_{\Gamma_t} w_i \sigma_{ij} \nu_j da = \int_{\Gamma_t} w_i (\sigma_{ij}' - p f \delta_{ij}) \nu_j da = \int_{\Gamma_t} w_i \sigma_{ij}' \nu_j da - \int_{\Gamma_t} w_i p f \nu_i da
\]
(2.34)

and,
\[
\int_{S^+} w_i^+ \sigma_{ij}^+ n_j^+ da = \int_{S^+} w_i^+ (\sigma_{ij}'^+ - p_f^+ \delta_{ij}) n_j^+ da = \int_{S^+} w_i^+ \sigma_{ij}'^+ n_j^+ da - \int_{S^+} w_i^+ p_f^+ n_i^+ da
\]
\[
T_i'^+ = -T_i'^- = T_i'
\]
(2.35)
\[
= \int_{S} w_i^+ T_i'da - \int_{S} w_i^+ n_i p_f^+ da
\]
(2.36)

Where \(p_f^+\) is calculated from nodal values on \(S^+\), and the values are interpolated along the virtual surface \(S\). Assuming \(T_i'^- = -T_i'^+\) on \(S^-\) (traction is equal and opposite), we have
\[
\int_{S^-} w_i^- \sigma_{ij}^- n_j^- da = \int_{S^-} w_i^- (\sigma_{ij}'^- - p_f^- \delta_{ij}) n_j^- da
\]

Combining all surface traction terms,
\[
\int_{\Gamma_t} w_i \sigma_{ij} \tilde{n}_j da = \int_{\Gamma_t} w_i \sigma_{ij}' \tilde{n}_j da - \int_{\Gamma_t} w_i \nu_i p_f \tilde{n}_i da
\]
\[
+ \int_{S} w_i^+ T_i'da - \int_{S} w_i^+ n_i p_f^+ da
\]
\[
+ \int_{S} (-w_i^-) T_i'da - \int_{S} (-w_i^-) n_i p_f^- da
\]
(2.37)

On virtual mid-plane \(S\), we let \(p_f^+ = p_f^- = p_f^S\), and then,
\[
\int_{\Gamma_t} w_i \sigma_{ij}' da - \int_{\Gamma_t} w_i \nu_i p_f da - \int_{S} (w_i^- - w_i^+) T_i'da + \int_{S} (w_i^- - w_i^+) n_i p_f^S da
\]

Where \(T_i'\) is the effective traction on the discontinuity \(S\), for which we define the constitutive equations [Yu, 2010], where the symbol \([[\cdot]]\) denotes a jump. The variational equation is
\[
\int_{\Gamma_1} w_i t''_i da - \int_{\Gamma_1} w_i \nu_i p_f da - \int_S [[w_i]] T'_i da + \int_S [[w_i]] n_i p_f^S da + \int_\Omega w_i b_i \rho dv - \int_\Omega \frac{\partial w_i}{\partial x_j} \sigma_{ij}' \delta_{ij} \ dv = 0
\]

(2.38)

Then, in summary,

\[
\int_\Omega \frac{\partial w_i}{\partial x_j} \sigma_{ij}' \ dv - \int_\Omega \frac{\partial w_i}{\partial x_i} p_f \ dv = \int_\Omega \rho w_i b_i \ dv + \int_{\Gamma_1} w_i t_i' da
\]

\[
- \int_{\Gamma_1} w_i \nu_i p_f da - \int_S [[w_i]] T'_i da + \int_S [[w_i]] n_i p_f^S da
\]

(2.39)

Where \( t_i' \) is the applied effective traction, \( p_f \) is the pore fluid pressure degree of freedom, and \( T'_i \) is the effective traction on \( S \).

The interpolation of pore fluid pressure on each facet of the discontinuity of \( S^h \):

\[
p_f^{+h} = \sum_{a=1}^{2} N^p_a(\xi) \theta_a^e
\]

\[
p_f^{-h} = \sum_{a=4}^{5} N^p_a(\xi) \theta_a^e
\]

(2.40)

Equation 2.19 shows the shape functions used for the nodal pore fluid pressures at local nodes 1, 2, 4, and 5. Then, the pore fluid pressure on the virtual mid-plane of the fracture \( S^h \) is:

\[
p_f^{S^h} = \frac{1}{2} (p_f^{+h} + p_f^{-h})
\]

(2.41)

\[
= \frac{1}{2} \left( N^p_1 \theta_1 + N^p_2 \theta_2 + N^p_4 \theta_4 + N^p_5 \theta_5 \right)
\]

(2.42)

\[
= \frac{1}{2} \begin{bmatrix} N^p_1 & N^p_2 & N^p_4 & N^p_5 \end{bmatrix} \begin{bmatrix} \theta_1 \\ \theta_2 \\ \theta_4 \\ \theta_5 \end{bmatrix}
\]

(2.43)

\[
= \frac{\text{N}_{\text{cse}}}{<1\times4>} \cdot \theta_{\text{cse}}^{<4\times1>}
\]

(2.44)
Figure 2.9: Diagram of a cohesive surface element. The top surface is referred to as $S^-$, the center surface is $S$, and the bottom surface is $S^+$. Pressure is found at the four corner nodes (local number 1, 2, 4, and 5). A fictitious plane $S^h$ is shown in the center (dotted line).
$D = \begin{pmatrix}
(\lambda + 2\mu) & \lambda & 0 \\
\lambda & (\lambda + 2\mu) & 0 \\
0 & 0 & \mu
\end{pmatrix}$ \hspace{1cm} (2.45)

\[ \mathbf{A}^{n_{e}}(C^{e})^{T} \cdot \left[ \int_{\Omega^{e}} \mathbf{B}^{e,uT} \cdot \mathbf{\sigma}^{th} d\alpha - \left( \int_{\Omega^{e}} \tilde{\mathbf{B}}^{e,uT} \mathbf{N}^{e,p} d\alpha \right) \cdot \mathbf{\theta}^{e} \right. \]

\[ = \int_{\Omega^{e}} \rho \mathbf{N}^{e} \mathbf{g} d\alpha + \int_{\Gamma^{e}} \mathbf{N}^{e} \mathbf{t} d\alpha - \left( \int_{\Gamma^{e}} (\mathbf{N}^{e} \mathbf{\nu}) \mathbf{N}^{e,p} dS \right) \mathbf{\theta}^{e} \]

\[ - \int_{S^{e}} \mathbf{N}^{cse,uT} \mathbf{T}^{th}(d^{cse}) dS \]

\[ + \left( \int_{S^{e}} \mathbf{N}^{cse,uT} \cdot \mathbf{n} \cdot \mathbf{N}^{cse,p} dS \right) \cdot \mathbf{\theta}^{cse} \] \hspace{1cm} (2.46)

Where $\mathbf{A}$ is the element assembly operator. We map to integrate in the natural coordinates $\xi$ and $\eta$

\[ \text{area:} \quad d\alpha \rightarrow f^{\xi} d\xi d\gamma \] \hspace{1cm} (2.47)

\[ \text{surface:} \quad d\gamma \rightarrow f(\xi) d\xi \] \hspace{1cm} (2.48)

\[ \mathbf{A}^{c} C^{eT} \cdot \left[ \mathbf{f}^{e,d,\text{INT}}(d^{e}) - \mathbf{k}^{e,d\theta\theta} \mathbf{\theta}^{e} = \mathbf{f}^{e,d,\text{EXT}}(d^{e}) - \mathbf{k}^{e,d\theta} \cdot \mathbf{\theta}^{e} \right. \]

\[ \left. - \mathbf{f}^{cse,d,\text{INT}}(d^{cse}) + \mathbf{k}^{cse,d\theta} \cdot \mathbf{\theta}^{cse} \right] \] \hspace{1cm} (2.49)

Assemble:

\[ \mathbf{F}^{d,\text{INT}}(d) - \mathbf{K}^{d\theta\theta} : \mathbf{\theta} = \mathbf{F}^{d,\text{EXT}}(d) - \mathbf{K}^{d\theta2} : \mathbf{\theta} - \mathbf{f}^{cse,d,\text{INT}}(d) + \mathbf{K}^{cse,d\theta} : \mathbf{\theta} \] \hspace{1cm} (2.50)

\[ \mathbf{R}^{d}(d, \mathbf{\theta}) = \mathbf{F}^{d,\text{INT}}(d) + \mathbf{F}^{cse,d,\text{INT}}(d) \]

\[ + \left( - \mathbf{K}^{d\theta\theta} + \mathbf{K}^{d\theta2} - \mathbf{K}^{cse,d\theta} \right) : \mathbf{\theta} - \mathbf{F}^{d,\text{EXT}}(d) = 0 \] \hspace{1cm} (2.51)
2.4.5 Balance of Mass

Before a model that incorporates both pore fluid flow and solid skeleton deformation behavior with fracture can be developed, governing equations for the flow of the fluid within the fracture must be developed. Following examples in previous literature [Segura and Carol, 2004], the fluid is considered to flow in two primary ways, transversal flow (across the fracture, out of the porous bulk material) and longitudinal flow (parallel within the crack, neither entering nor leaving the bulk material). In order to develop the model, it will be assumed that the fluid is incompressible, e.g. water. Further, to simplify the analysis, the porous media will be assumed to be fully fluid saturated, i.e. no separate gas phase in the void space.

While the equation modeling the longitudinal flow within an element can be derived from “Conservation of Mass” (i.e. Fluid in + Fluid out = Change in fluid of system), the transverse flow must take into account the porous nature of the bulk element. Due to the porous nature, Darcy’s law will be used to derive the transverse flow [Anandarajah, 2010]. It will be assumed that there exists a fluid flux into the fracture from the neighboring bulk elements.

Next, the balance of mass for a mixture involving a solid, a (single phase) fluid, and a discontinuity S are derived ([Regueiro, 2011, Anandarajah, 2010])

\[
\begin{align*}
\frac{\partial u_i}{\partial x_i} + \frac{\partial \bar{v}_i^s}{\partial x_i} &= 0 \quad x_i \in \Omega \text{ and } x_i \in S \\
p_f &= r \quad x_i \in \Gamma_r \\
-\nu_i (\bar{v}_i^s)_f &= S_f \quad x_i \in \Gamma_s \\
-n^+_i (\bar{v}_i^s)_f^+ &= \bar{v}_n^+ \quad x_i \in S^+ \\
-n^-_i (\bar{v}_i^s)_f^- &= \bar{v}_n^- \quad x_i \in S^-
\end{align*}
\]

(2.53)

\[
\hat{\nu}_f = \nu_f - \nu = \text{relative velocity vector of pore fluid with respect to solid skeleton motion,}
\]

\[
\hat{v}_f = n_f \hat{\nu}_f \quad \text{is the seepage velocity. Within the local coordinate system of the CSE, the jump displacement vector is}
\]
\[ \begin{bmatrix} \hat{\mathbf{u}} \end{bmatrix} = \begin{bmatrix} u_t \\ u_n \end{bmatrix} \] (tangential) 

Assume the pore fluid pressure along the fracture is an average of the values on the adjacent facets of the bulk elements:

\[ p_f^S = \frac{1}{2} \left( p_f^+ + p_f^- \right) \]

Now, introduce a jump discontinuity in solid skeleton displacement into the balance of mass of the mixture:

\[ \frac{\partial \hat{u}_i}{\partial x_i} + \frac{\partial (\hat{v}_i^S)}{\partial x_i} = 0 \quad \mathbf{x} \in S^l \]

Where,

\[ \hat{u}_i = \begin{bmatrix} \hat{u}_i \end{bmatrix} H_S(\mathbf{x}) \quad \mathbf{x} \in S^l \]

Where \( H_S(\mathbf{x}) \) is the Heaviside function centered at discontinuity \( S \), and \( l \) is the discontinuity aperture. Next, take the derivatives

\[ \frac{\partial \hat{u}_i}{\partial x_i} = \frac{\partial \begin{bmatrix} \hat{u}_i \end{bmatrix}}{\partial x_i} H_S + \begin{bmatrix} \hat{u}_i \end{bmatrix} \frac{\partial}{\partial x_i} \frac{\partial \hat{v}_i^S}{\partial x_i} \delta_S \]

Where \( \delta_S \) is the Delta-function on \( S \). The seepage divergence term can be related to the components in the local coordinate system \( x_i \).

\[ \frac{\partial (\hat{v}_i^S)_f}{\partial x_i} = \frac{\partial \hat{v}_i^S}{\partial x_n} + \frac{\partial \hat{v}_i^S}{\partial x_t} \]

For the weighted residual: \( \eta = \delta p_f \)

\[ \int_{S^l} \eta \left( \hat{u}_n \delta_S + \frac{\partial (\hat{v}_i^S)_f}{\partial x_i} \right) dv = 0 \]

\[ \int_{S^l} \eta \hat{u}_n \delta_S dv + \int_{S^l} \eta \frac{\partial (\hat{v}_i^S)_f}{\partial x_i} dv = 0 \]
The Delta function $\delta_S$ localizes the integral onto $S$ as

$$\int_{S'} \eta \delta_S dv = \int_S \eta^S \delta_S da$$

Where $\eta^S = \frac{1}{2}(\eta^{S-} + \eta^{S+})$. Using the chain rule,

$$\frac{\partial}{\partial x_i} \left( \eta^S \right)_f = \frac{\partial \eta}{\partial x_i} \left( \tilde{v}^S \right)_f + \eta \frac{\partial (\tilde{v}^S)_f}{\partial x_i}$$

(2.54)

$$\int_{S'} \eta \frac{\partial (\tilde{v}^S)_f}{\partial x_i} dv = \int_{S'} \frac{\partial}{\partial x_i} \left( \eta^S \right)_f dv - \int_{S'} \frac{\partial \eta}{\partial x_i} (\tilde{v}^S)_f dv$$

(2.55)

and the Divergence theorem,

$$\int_{S'} \frac{\partial}{\partial x_i} \left( \eta^S \right)_f dv = \int_S \left( \eta^S \right)_f n^+_i da + \int_{S'} \eta^- (\tilde{v}^S)_f n^-_i da + \int_{\Gamma_S} \eta^S f \nu_i da$$

The flux is positive into the crack, such that

$$\int_{S'} \frac{\partial}{\partial x_i} \left( \eta^S \right)_f dv = \int_S (\eta^+ - \eta^-) \tilde{v}^S_n da - \int_{\Gamma_S} \eta S f da$$

Along the crack,

$$\frac{\partial \eta}{\partial x_i} (\tilde{v}^S)_f = \frac{\partial \eta^S}{\partial x_i} \tilde{v}^S_i + \frac{\partial \eta^S}{\partial x_n} \tilde{v}^n_i$$

Where $\tilde{v}^S_i$ is the tangential flow in the crack and $\tilde{v}^n_i$ is the normal flow in the crack. Putting the terms together,

$$\int_{S'} \frac{\partial \eta}{\partial x_t} \tilde{v}^S_i dv + \int_{S'} [[\eta]] \delta_S \tilde{v}^S_n dv = \int_S \int_0^l \frac{\partial \eta}{\partial x_t} \tilde{v}^S_i dxn da + \int_S [[\eta]] \tilde{v}^S_n da = \int_S \frac{\partial \eta^S}{\partial s} \tilde{v}^S l da + \int_S [[\eta]] \tilde{v}^S_n da$$

Where $l$ is the crack aperture, and

$$\int_S \eta^S \delta_n da - \int_{S'} [[\eta]] \tilde{v}^S_n da - \int_{\Gamma_S} \eta S f da - \int_S \frac{\partial \eta^S}{\partial s} (\tilde{v}^S l) da - \int_S [[\eta]] \tilde{v}^S_n da = 0$$

Finally, in 2D plane strain, assuming unit thickness into the page,
\[ \int_S \eta^S \hat{u}_n ds - \int_S \left[ \left[ \eta \right] \right](2 \hat{v}_n^S) ds - \int_S \frac{\partial \eta^S}{\partial s} (\hat{v}_i^S l) ds = (\eta_0 S f_n + \eta_L S f_L) l \]

Merging the bulk and discontinuity balance of mass equations, we have in 2D

\[ \int \eta \frac{\partial \hat{u}_i}{\partial x_i} da - \int \eta \frac{\partial (\hat{v}_i^n)}{\partial x_i} da - \int \eta S_f ds - \int \left[ \left[ \eta \right] \right](2 \hat{v}_n^S) ds - \int \frac{\partial \eta^S}{\partial s} (\hat{v}_i^S l) ds - \int (\hat{v}_i^S \hat{u}_n) l ds = 0 \]

Where \( \eta^S = \frac{1}{2}(\eta^+ + \eta^-) \) and \( \left[ \left[ \eta \right] \right] = (\eta^- - \eta^+) \).

The nonlinear finite element matrix equations to solve are

\[ C \cdot \dot{D} + F^{INT}(D) = F^{EXT}(D) \]

\[ D = \begin{bmatrix} d \\ \theta \end{bmatrix}, \dot{D} = \begin{bmatrix} \dot{d} \\ \dot{\theta} \end{bmatrix} \]

\[ C = \begin{bmatrix} 0 & 0 \\ (K^{d\theta 1})^T + K^{\text{cse},d\theta} & 0 \end{bmatrix} \]

\[ dF^{INT}(d) = \begin{bmatrix} \frac{\partial F^{d,INT}}{\partial d}(d) + \frac{\partial F^{\text{cse},d,INT}}{\partial d}(d) + (K^{d\theta 1} + K^{d\theta 2}) \cdot \theta - K^{\text{cse},d\theta} \\ -F^{\theta,INT}(d, \theta) - F^{\text{cse}2,p,INT}(d, \theta) - F^{\text{cse}2,p,INT}(d, \theta) \end{bmatrix} \]

\[ F^{EXT}(D) = \begin{bmatrix} \frac{\partial F^{d,EXT}}{\partial d}(d) \\ \frac{\partial F^{\theta,S,EXT}}{\partial d} + \frac{\partial F^{\text{cse},\theta,S,EXT}}{\partial d} + \frac{\partial F^{\text{cse}L,\theta,1,EXT}}{\partial d} \end{bmatrix} \]

### 2.4.6 Components of Consistent Tangent

To implement the Newton-Raphson solution method for the nonlinear problem, the consistent tangent is now found (shown below).

\[ \frac{\partial F^{INT}}{\partial d} = \begin{bmatrix} \frac{\partial F^{d,INT}}{\partial d} + \frac{\partial F^{\text{cse},d,INT}}{\partial d} \\ \frac{\partial F^{\theta,INT}}{\partial d} - \frac{\partial F^{\text{cse}1,p,INT}}{\partial d} - \frac{\partial F^{\text{cse}2,p,INT}}{\partial d} \end{bmatrix} \]
\[
\frac{\partial F^{INT}}{\partial d} = \begin{bmatrix}
-K^{d\theta_1} + K^{d\theta_2} - K^{cse,d\theta} \\
\frac{\partial F^{e,INT}}{\partial \theta} - \frac{\partial F^{cse,1,p,INT}}{\partial \theta} - \frac{\partial F^{cse,2,p,INT}}{\partial \theta}
\end{bmatrix}
\] (2.57)

\[
\frac{\partial F^{EXT}}{\partial d} = \begin{bmatrix}
\frac{\partial F^{d,EXT}}{\partial d} \\
0
\end{bmatrix}
\] (2.58)

\[
\frac{\partial F^{INT}}{\partial d} = \begin{bmatrix}
\frac{\partial F^{e,INT}}{\partial d} \\
\frac{\partial F^{INT}}{\partial \theta}
\end{bmatrix}
\] (2.59)

\[
\frac{\partial F^{EXT}}{\partial d} = \begin{bmatrix}
\frac{\partial F^{e,EXT}}{\partial d} \\
\frac{\partial F^{EXT}}{\partial \theta}
\end{bmatrix}
\] (2.60)

\[
\frac{\partial F^{EXT}}{\partial \theta} = \begin{bmatrix}
0 \\
0
\end{bmatrix}
\] (2.61)
3.1 Solution Methods

In a 2008 paper [Segura and Carol, 2008a] compares the staggered and monolithic solution methods to solve a model for coupled solid-skeleton deformation and pore fluid diffusion analysis. The model uses double-noded, zero-thickness interface elements. The nonlinear equations are solved using both the staggered (partitioned) approach and the fully coupled (monolithic) approach (via the Newton-Raphson method). There are a few important differences in these solution techniques. The more straightforward and robust fully coupled (monolithic) solution directly solves the coupled equations for the unknowns (in this case, solid skeleton displacements and pore fluid pressures). In comparison, the more modular, iteratively coupled (staggered) approach uses two distinct models, one for pore fluid flow and one for the solid skeleton deformation. During the solution process, both models are solved many times at each time step (alternating back and forth) until the solution converges. However, in cases where the hydro-mechanical coupling is significant, convergence may not be achieved.

3.1.1 Fully Coupled (Monolithic)

In the present thesis, the fully coupled method is used to solve the equations for displacement and pressure. At each time step, the Newton-Raphson method is used with the exit criteria being the residual.

General process for the solution:
The solid skeleton volumetric strain ($\epsilon_v$) is found as a function of change in solid skeleton displacement:

$$f(\delta_d) \rightarrow \epsilon_v$$

The porosity ($n_f$) is found as a function of $\text{del} \_\text{epsv} \_\text{vol}$:

$$f(\text{del} \_\text{epsv} \_\text{vol}) \rightarrow n_f$$

From the densities of both the solid ($\rho_s$) and fluid ($\rho_f$), as well as the porosity found above, the total density ($\rho$) is found:

$$f(n_f, \rho_f, \rho_s) \rightarrow \rho$$

The effective stress ($\sigma'$) and strain ($\epsilon$) are found from the elastic modulus and element solid skeleton displacements:

$$f(\sigma) \rightarrow f_{int}$$

With the effective stress above, the internal force (for each element) is now integrated (3 x 3 gaussian) resulting in ($f_{int}$)

$$f(d) = p_f$$

The pore fluid pressure is found as a function of the current pore fluid pressure utilizing bilinear 2D shape functions.

$$f(d, \text{perm}) \rightarrow f_{thint}$$

The element fluid internal force is found as a function of the solid skeleton displacement and the permeability.

$$f(g, \rho) \rightarrow f_e$$

The body force is found (with 3x3 Gaussian integration)

assemble($f_{thint}, f_{int}, kdtheta$) $\rightarrow$ $F_{thint}$

Assemble variables to get $F_{thint}$

assemble($f^d_{f}, f^d_{t}$) $\rightarrow$ $F_{dthext}$

Assemble the traction ($f^d_{t}$) and body force ($f^d_{f}$)

$$f(\text{Dep}) \rightarrow dfdintdd$$

Integrate consistent tangent matrix to get $dfdintdd$

$$f(\theta, \text{perm}, \text{del} \_\text{epsv}, n_f, \rho_f, g) \rightarrow dfthindt$$

Integrate consistent tangent matrix to get $dfthindt$

$$f(\text{perm}) \rightarrow dfthindtth$$

Calculate $dfthindtth$ as a function of the permeability

$$f(n_f, \text{del} \_\text{epsv}, g, \rho_f, \rho_s) \rightarrow df$$

Integrate body force tangent to get $df$

$$f(f_{int}, f_{ext}, f_{thint}) \rightarrow f_{int}$$

Assemble internal force vector $f_{int}$

$$f(f_{ext}, F_{dthext}) \rightarrow f_{ext}$$

Assemble external force vector $f_{ext}$
3.1.2 Staggered (partitioned)

Implementation of the staggered method involves two solution algorithms and is contrasted with the coupled approach in [Segura and Carol, 2008a]. One solves for the solid skeleton displacement and the other solves for the pore fluid pressure. A third algorithm links these two together iteratively. That is, one solution is found (e.g. solid skeleton displacement) and the results from this solution are plugged into the other one (e.g. pore fluid pressure) as initial conditions.

Convergence is considered to be achieved when the change in pore fluid pressure between iteration loops is less than a pore fluid pressure tolerance, and the change in solid skeleton displacement between iteration loops is less than a displacement tolerance. However, as discussed by [Segura and Carol, 2008a], convergence with the staggered method is not always easily achieved and other methods may be required to achieve a solution.

3.2 Finite Element Utilities Design in MATLAB

3.2.1 Object Oriented Programming

To allow for rapid problem generation (i.e. mesh generation, BC’s, IC’s, parameters), many class definitions in MATLAB were designed. These rely on object oriented programming and were able to streamline the problem setup. Developing the code in this way will allow it to be utilized in future projects and research due to the inherent modular properties. As shown in Figure 3.1, a format for a problem definition is specified. The definition file includes the parameters required to define the bulk element, the boundary conditions (e.g. fixed nodes, fixed pressure), initial conditions (e.g. specified pressures), and where the CSEs are to be inserted. This allows various problems to be rapidly switched out and further allows any definition file that adheres to the format to be used.

The programs are designed with many object oriented principles in mind. Each finite element in the mesh is represented by a specially designed class that encapsulates all information relevant
Figure 3.1: A schematic of the programming structure implemented in MATLAB. ProbDef handles all of the problem specific settings while the rest is the framework for setting up the problem.
to that element, e.g. the element’s ID, nodes it consists of, and past operations carried out on it. Encapsulating all functions relevant to the finite element allows for further simplification. For example, all elements are able to plot themselves and sort their nodes to the correct local node numbering.

As meshes grow larger, an area of concern is how much time it takes to accurately generate and manipulate the mesh. To handle this, many of the preexisting programs were converted to utilize logical vectors.

All of the information pertaining to the nodes is managed by an individual class instance (NodeDefClass.m). This was chosen over managing each node as an instance of the class due to previous experience indicating that MATLAB is slow when handling large amounts of handles (such as a handle for each node). The node class holds all the information relative to all the nodes and functions to create and manipulate them.

The meshes presented required the insertion of cohesive surface elements throughout the mesh. In order to handle these reliably, methods were designed into the element class and node definition class.

A cohesive surface element can be inserted between two neighboring bulk elements by specifying a variety of different parameters, e.g. between two elements, between two rows, etc. For example, an element can be inserted by specifying the interface nodes. Because these are quadratic, this will consist of three nodes. The nodes are duplicated, resulting in 6 nodes. These new copied nodes keep all the same parameters that were present in the original nodes. Thus, if the original node was fixed in the y direction, the copied node will be as well. However, issues can arise when another CSE is inserted adjacent to a prior one. The node class handles this as well by setting rules for how many times a single node can duplicate itself.

When all 6 of the nodes are handed over to a new instance of the element class to define the new cohesive surface element, the class updates all references to reflect this. At this point, the class then sorts the nodes to ensure that the local node numbering is consistent.

Inserting CSEs in this manner has a major advantage over creating the CSEs during bulk
element mesh generation. When the CSEs are inserted in this way, the only information required is related to connectivity. This allows the CSEs to be inserted at any time, including after the simulation has started. As a result, extending the model to dynamically insert CSEs will be very straightforward.

A very useful feature of this object oriented design were *get functions*. Get functions are assigned to be called in place of an actual variable. Utilizing this aspect, a method can replace a variable to ensure that the outputted value is always up to date. For example, if the properties of a class consists of the modulus of elasticity and strain (both of which could be changing in time), and an output of the stress is desired, a get function could be used in place of a stress variable to calculate the stress in real time. This is extremely important to ensure that old (non-updated) values are not used when new values are calculated. Within the program, these functions are applied to filter the data to a consistent format. As a result, the data can be stored and manipulated in a more intuitive format while maintaining compatibility with other functions and programs.

### 3.2.2 Performance

The computational time for a solution to be found can range from moderate to substantial. In an effort to make the model and solution method more feasible, many methods were attempted to increase the performance within MATLAB. These methods included:

1. Implementing a variable time step size
2. Allowing a solution to be retried when a previous solution fails to converge
3. Vectorizing all calculations to reduce the use of loops
4. Converting the code to run in parallel
5. Compiling the code to a standalone application

A variable time step was implemented based on a set of parameters. These parameters govern the behavior of the time step size based on user specified constants (max time step size, max change...
in time step size from step to step). The input to the function is the number of iterations that the previous step required. However, it was observed that the number of iterations required in the previous step did not correlate well with the tolerable time step size for the subsequent step. As a result, this was disabled in the model.

Originally, the code would simply stop running when the first time step that did not converge was reached. To improve this, a function was implemented to calculate the solution at a specific time step. This allowed the solution to be evaluated and, if necessary, repeated with different input parameters at the same time step.

From past experience with improving MATLAB’s performance, it was attempted to decrease simulation time by reducing the degree of nested loops by vectorizing the operations and defining three-dimensional matrix multiplication. Surprisingly, this lead to minimal decrease in simulation time. It was speculated that this was because MATLAB’s compiler is now advanced enough to convert loops to their equivalent vectorized operations.

Finally, the operations were converted to run in parallel (within a time step). This method is possible because each elemental operation is only dependent upon the previous time step (as opposed to the neighboring elements). This approach was tested on some of the initial meshes consisting of six total elements. Unexpectedly, this drastically increased the solution time (by about 4x). On closer inspection, it is believed that this is caused by the nature of the code. There were a large number of time steps (up to 700,000), while the number of nested loops within this was relatively small (around 6). Further, the time to complete the calculations within each time step was relatively small (less than a tenth of a second). Threading out each calculation to a different processor has some associated inherit overhead. It is believed that this overhead simply outweighed any decrease in actual computation time. However, as the meshes grow larger, the ability to solve them in parallel may become more beneficial. For a mesh with over two thousand elements, the solution time at a given time step was around 30 minutes (with a predicted total time of approximately 14 days). Depending on the number of processors and speed at which information can be handed off, this mesh may benefit from a parallelized algorithm.
Chapter 4

Examples

To provide further insight into the newly developed model, many different problems are solved in this chapter. These include problems with various types of initial conditions and boundary conditions such as an applied vertical traction (Section 4.1) or a specified input flux into the fracture (Section 4.2). All of the following were solved via the fully coupled, backward Euler method implicitly in conjunction with the Newton-Raphson method for nonlinear solutions.

4.1 Comparison of a 64 bulk element mesh with and without cohesive surface elements

A problem was set up similar to that in [Segura and Carol, 2008b] consisting of 64 bulk elements and no CSEs (Section 4.1.1). The bottom nodes are fixed in the vertical direction, while the left and right sides are fixed in the horizontal direction. To understand the CSE’s influence on the large scale behavior of the material, a second mesh was created exactly the same as the mesh mentioned above except it included 8 vertically-oriented CSEs dividing the mesh into a left half and a right half (Section 4.1.2). In both cases, a downward traction of 10kPa is applied across the top nodes. The flux across the bottom, left, and right boundaries is assumed to be zero.

4.1.1 Example with applied traction and no cohesive surface elements

This example consists of 64 bulk elements with no CSEs (see Figure 4.1). This allows the mesh to be characterized based solely on the bulk element’s behavior. The vertical displacements
Table 4.1: Parameters used for examples

<table>
<thead>
<tr>
<th>variable</th>
<th>Value</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>9.81</td>
<td>m/s²</td>
</tr>
<tr>
<td>ρs</td>
<td>2700</td>
<td>kg/m³</td>
</tr>
<tr>
<td>ρf</td>
<td>1000</td>
<td>kg/m³</td>
</tr>
<tr>
<td>n₀ₛ</td>
<td>0.58</td>
<td></td>
</tr>
<tr>
<td>n₀ᶠ</td>
<td>0.42</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>1 × 10⁶</td>
<td>Pa</td>
</tr>
<tr>
<td>ν</td>
<td>0.30</td>
<td></td>
</tr>
<tr>
<td>αₓ</td>
<td>2e2</td>
<td></td>
</tr>
<tr>
<td>αᵧ</td>
<td>2e2</td>
<td></td>
</tr>
<tr>
<td>ϕ</td>
<td>900</td>
<td></td>
</tr>
<tr>
<td>ψ</td>
<td>900</td>
<td></td>
</tr>
<tr>
<td>perm0</td>
<td>1 × 10⁻⁹</td>
<td>m²</td>
</tr>
<tr>
<td>kcrack</td>
<td>1 × 10⁻¹¹</td>
<td>m²</td>
</tr>
<tr>
<td>kₜ</td>
<td>1 × 10¹⁰</td>
<td>Pa</td>
</tr>
<tr>
<td>kₙ</td>
<td>1 × 10¹⁰</td>
<td>Pa</td>
</tr>
</tbody>
</table>

Figure 4.1: 64 bulk elements and no cohesive surface elements. Note: duplicated node ID’s are right of the node. (See Appendix A for more information on the problem setup)
of the top nodes vs time (one on the edge of the mesh and one in the center) is shown in Figure 4.2. This is a consolidation process as the solid skeleton displacement approaches steady state.

![Vertical Displacement at Top Nodes](image)

**Figure 4.2:** (Mesh140) The displacement of the top nodes as a function of time

Figure 4.3 shows the pore fluid pressure at two bottom nodes, one near the center and one at the edge. The node near the center maintains a slightly lower pressure throughout the simulation.
Figure 4.3: (Mesh 140) The pore fluid pressure at the bottom nodes as a function of time. All nodes with a pore fluid pressure degree of freedom are initialized to hydrostatic pressure.
4.1.2 Example with applied traction and 8 vertically oriented cohesive surface elements

This example consists of 64 bulk elements with 8 vertically-oriented CSEs inserted that divide the problem into a left half and right half (see Figure 4.4).
Figure 4.4: 64 bulk elements and 8 cohesive surface elements dividing the mesh into a left and right half. This is the same as Figure 4.1 except that CSEs are inserted. Note: Duplicated node IDs are right of the node. (See Appendix A for more information on the problem setup)

Figure 4.5: The pore fluid pressure at the bottom nodes as a function of time
Figure 4.6: The displacement of the top nodes as a function of time. As expected, the solid skeleton near the CSE is able to displace faster than the solid skeleton further away from the CSE.
4.1.3 Example - Comparison of a mesh with cohesive surface elements against the same mesh without cohesive surface elements

Meshes both with and without CSEs are compared to demonstrate the CSE's impact of the bulk material. Both meshes have 64 bulk elements. However, one has no CSEs (Mesh 140), while the other has 8 vertically oriented CSEs subdividing it into a left and right half.

Figure 4.7 compares the displacements of a top node in each mesh. As expected, the mesh containing cohesive surface elements more rapidly displaces. This is due to the cohesive surface element (with a finite aperture) providing a pathway for the fluid to escape through and as a result, allowing a more rapid consolidation of the solid skeleton.

Figure 4.7: The displacement of the top nodes as a function of time compares a mesh without CSEs (Mesh 140) to a mesh with CSEs (Mesh 14)

Figure 4.8 compares the pore fluid pressure at a node near the bottom-center of the mesh for both the case with (mesh 14) and without (mesh140) CSEs. Similar to the displacement case, the mesh containing a CSE (with finite aperture) allows the pore fluid pressure to be more rapidly dissipated.
Figure 4.8: The pore fluid pressure of node 137 (bottom center of mesh) as a function of time compares a mesh without CSEs (Mesh 140) to a mesh with CSEs (Mesh 14)
For a reference point further away from the CSE, Figure 4.9 compares the pore fluid pressure at the bottom left node for both the case with and without a CSE present. Contrasting Figures 4.9 and 4.8, it is visible that the case with a node further away from any CSEs takes longer to dissipate pore fluid pressure.

The above figures display the previously mentioned hydromechanical coupling. It is visible that the pore fluid flow into the fracture effects the fracture’s aperture which in turn, effects the fluid flow.
Figure 4.9: The pore fluid pressure of node 1 (bottom left of mesh) as a function of time compares a mesh without CSEs (Mesh 140) to a mesh with CSEs (Mesh 14). The mesh containing CSEs allows the fluid to escape faster and thus maintains a lower pressure.
4.2 Example - 64 bulk elements, 8 vertically oriented cohesive surface elements (Mesh 16) with a specified volume flux of fluid injected

A mesh similar to the previous sections was setup consisting of 64 bulk elements and 8 horizontally-oriented CSEs. However, rather than specifying an external traction, a fixed pore fluid flux is specified at the left edge of the fracture. (see Figure 4.10).

Figure 4.11 displays the pore fluid pressure as a function of the distance from the injection location (located at x = 0). Though the pore fluid pressure does decrease with increasing distance from the injection location, the amount is minute. This indicates that the pore fluid pressure can quickly reach equilibrium in the longitudinal direction. As time increases however, the pressure continues to build somewhat asymptotically. During this time, the variation in pore fluid pressure along the crack remains small. This may indicate a higher resistance to transverse flow.

Figure 4.12 displays the effect of the input fluid flux on the fracture aperture. As fluid is injected, the fracture continues to increase in width. Even so, the rate of increase in aperture deceases with time (similar to how the rate of change of pore fluid pressure decreased with time in Figure 4.11).
64 bulk elements, 8 CSE's
306 nodes, 559 displ. DOF's 81 pressure DOF's

Figure 4.10: Mesh 16 Diagram with first cohesive surface element on left artificially expanded for visualization. A fixed velocity of fluid flux is injected on the left as indicated by the arrow. The CSE IDs start at 65 on the left where fluid is being injected, and increase to 72 on the right (between bulk elements 32 and 40)
Figure 4.11: Pore fluid pressure as a function of distance from the injection site.
Figure 4.12: Aperture near the injection site as a function of time. See Figure 4.10 for element locations
Figure 4.13: Pore fluid pressure distribution within the mesh near t=0
Figure 4.14: The results of a 64 element mesh subdivided horizontally by 8 CSEs resulting in an upper and lower half. A input fluid flux is prescribed on the left side.
Chapter 5

Discussion

5.1 Meshing

5.1.1 Other Mesh Considerations

Careful attention must be given to the meshing of the domain. Because the discrete method is used, fracture can only occur where a cohesive surface element is placed. As a result, the mesh will define where fracture can or will occur. To compensate for this, it may be tempting to place a cohesive surface element at every segment separating two elements. However, doing so presents two disadvantages:

(1) The computational time required to solve the problem will increase due to the increased number of degrees of freedom.

(2) Under some conditions, the mesh may exhibit oscillations due to the elasticity of the cohesive surface elements. However, adjustment of the parameters of the cohesive surface element may be able to reduce this effect.

Because the solution is mesh dependent, even in the best case scenario, other options must be explored to gain better insight into the solution. One way of solving this is to use a random Voronoi mesh (Figure 5.2). Voronoi meshes use randomly placed points which allow the calculation for randomly shaped polygonal elements (in 2D). Even in this case, the fractures are still only capable of being along the edge between two elements and, as a result, is still mesh dependent. However, because the elements are much more random, this may result in more realistic solutions.
5.1.2 Dynamic Insertion

Another solution is to dynamically insert cohesive surface elements based on a specified criteria. This eliminates any oscillations that may occur before the CSE is activated, but the solution will still be mesh dependent.

5.1.3 Mesh Dependence

As mentioned previously, the solution can be dependent upon the characteristics of the mesh. Figure 5.1 is a hypothetical initial mesh (left) and a hypothetical solution under certain loading conditions (right). This is in comparison to the theoretical experimental solution. Since fracture can only occur with a CSE present, the solution shown is only correct if the experimental solution did not have a crack at the upper-left vertical facet.

Furthermore, in a regular rectangular mesh, after a fracture occurs at one point, it is more likely to continue along the line (or plane in 3D) resulting in a completely linear (or planar in 3D) fracture path. A possible remedy is discussed in Section 5.1.4.

![Figure 5.1: A hypothetical mesh and solution](image)

5.1.4 Random Voronoi Mesh

A possible solution to the mesh dependence was shown by Bishop, 2009, Bishop, 2012. In this method, a mesh consisting of polyhedral elementals is created by randomly seeding in points
with a limit on proximity to the nearest neighbor. Each point can be considered to represent an element where each element’s domain is all the area in 2D (or volume in 3D) whose closest point is the element’s point. This results in a mesh of random polygonal (2D) or polyhedral (3D) elements, referred to as a **randomly close packed Voronoi tessellation** (see Figure 5.2). With this method, there are rarely a straight line (2D) or plane (3D) throughout the mesh, which more accurately mimics fracture found experimentally.
Figure 5.2: An example of a random Voronoi tessellation
In this thesis, a model was derived and implemented to discretely model hydromechanical fracture applicable to geomaterials. Though the model has numerous strengths, there is still room for improvement. While some shortcomings are related to the implementation of the model, others are related to the concept. That is, regardless of the model, fracture can only occur at facets between bulk elements and therefore, will always be mesh dependent (unless representing the interface between a grain and the matrix). Though methods have been proposed to solve this (Section 5.1.4), it needs to be investigated further to increase the capabilities of the model presented.

Further, the implementation could likely benefit from an improved algorithm in MATLAB. As mentioned previously, under certain conditions, the solution could be slightly computationally expensive. Reducing this would increase the range and size of solvable problems. MATLAB’s parallel processing tool kit was briefly implemented with negative results. However, with better data handling, positive results may be easily achievable.

To improve the robustness of the model presented, it may be worthwhile to investigate methods for reducing the effect of the CSEs when inactive. As mentioned above, the CSEs can increase computational expense and cause oscillation issues.

The next step for the model will be to add in a third (gas) phase in addition to the solid and liquid phases. This adds significant complexity because of the new physics that must be accurately modeled (e.g. phase changes).
Bibliography


Appendix A

Problem Definitions
Figure A.1: 64 bulk elements and 8 cohesive surface elements dividing the mesh into a left and right half. (Note: duplicated node ID's are right of the node)
Figure A.2: 64 bulk elements and no cohesive surface elements. This is the same as Figure 4.4 except for no cohesive surface elements are present. (Note: duplicated node ID’s are right of the node)
Figure A.3: Mesh 16 Diagram with first cohesive surface element on left expanded for visualization. A fixed volume of fluid is injected in the direction of the arrow.
Appendix B

MATLAB Implementation and Source Code
B.1 Important Variable Definitions

Equation B.1 shows the internal state variables (ISVs) for each cohesive surface element.

\[
\begin{align*}
T_t & \quad \text{tangential traction} \\
T_n & \quad \text{normal traction} \\
u_t & \quad \text{tangential jump displacement} \\
u_n & \quad \text{normal jump displacement} \\
u_t^p & \quad \text{tangential plastic jump displacement} \\
u_n^p & \quad \text{normal plastic jump displacement} \\
\epsilon^p_s & \\
\epsilon^p_n & \\
\chi & \quad \text{tensile strength} \\
c & \quad \text{cohesion} \\
\tan \phi & \quad \text{friction angle} \\
\tan \psi & \quad \text{dilation angle} \\
F & \quad \text{plastic yield function} \\
dlam & \quad d\lambda \\
\text{iplastic} & \quad iplastic \\
\text{normr} & \quad |r| \\
kk & \quad kk
\end{align*}
\]
<table>
<thead>
<tr>
<th>MATLAB Variable</th>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>del_epsv_el</td>
<td>δε_v</td>
<td>Volumetric strain at each ip</td>
</tr>
<tr>
<td>del_epsv_el</td>
<td>ε_v</td>
<td>Volumetric strain of solid skeleton</td>
</tr>
<tr>
<td>del_epsv_el</td>
<td>δε_v</td>
<td>Volumetric strain at each ip</td>
</tr>
<tr>
<td>porosity</td>
<td>n_f</td>
<td>porosity of the element</td>
</tr>
<tr>
<td>perm</td>
<td>k</td>
<td>permeability of the element</td>
</tr>
<tr>
<td></td>
<td>p_f</td>
<td>real pore fluid pressure (positive in compression)</td>
</tr>
<tr>
<td></td>
<td>Ω_e</td>
<td>Element Domain</td>
</tr>
<tr>
<td></td>
<td>Ω_h</td>
<td>Discrete Domain</td>
</tr>
<tr>
<td>density</td>
<td>ρ</td>
<td>Density of the element at each ip</td>
</tr>
<tr>
<td>stress</td>
<td>σ</td>
<td>biquadratic quadrilateral element stress</td>
</tr>
<tr>
<td>strain</td>
<td>ε</td>
<td>element strain</td>
</tr>
<tr>
<td>T_i^+</td>
<td></td>
<td>Traction on S^+</td>
</tr>
<tr>
<td>T_i^-</td>
<td></td>
<td>Traction on S^-</td>
</tr>
<tr>
<td>n_j^+</td>
<td></td>
<td>Unit normal to S^+</td>
</tr>
<tr>
<td>n_j^-</td>
<td></td>
<td>Unit normal to S^-</td>
</tr>
<tr>
<td>N^e</td>
<td></td>
<td>Element Shape Function Matrix</td>
</tr>
<tr>
<td>d^e</td>
<td></td>
<td>Vector of nodal displacements</td>
</tr>
<tr>
<td>B^e</td>
<td></td>
<td>Element strain-displacement matrix</td>
</tr>
<tr>
<td>d^e_a</td>
<td></td>
<td>Global element displacement vector at node a</td>
</tr>
<tr>
<td>d^e_a</td>
<td></td>
<td>Local element displacement vector at node a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>within the cse tangent normal frame</td>
</tr>
<tr>
<td>α</td>
<td></td>
<td>angle between x-axis and average cse orientation</td>
</tr>
<tr>
<td>α</td>
<td></td>
<td>time integrator parameter</td>
</tr>
<tr>
<td>v</td>
<td></td>
<td>velocity of the solid skeleton</td>
</tr>
<tr>
<td>v_f</td>
<td></td>
<td>velocity of pore fluid</td>
</tr>
<tr>
<td>v_f</td>
<td></td>
<td>relative velocity of fluid WRT solid</td>
</tr>
<tr>
<td>f^{e,θ}</td>
<td></td>
<td>Fluid forcing vector</td>
</tr>
<tr>
<td>f_t</td>
<td></td>
<td>Element traction force vector</td>
</tr>
<tr>
<td>f_f</td>
<td></td>
<td>Element body force vector</td>
</tr>
<tr>
<td>f_e,θ</td>
<td></td>
<td>Element stiffness for pore pressure DOF</td>
</tr>
<tr>
<td>Fdthint</td>
<td>F^{d,INT}(d)</td>
<td>Topic 5 Page 42-121</td>
</tr>
</tbody>
</table>
\[
\text{perm_cse_params} \begin{cases} 
\text{kcrack} \\
\text{perm_crack} \\
\text{fluid_visc} \\
\text{small_num} \\
\text{fchi_r} \\
\text{rhoF} \\
\text{grav} \\
\text{small_num0}
\end{cases} \quad (B.3)
\]
B.2 Main Script

```
1  % Main Script
2  % clear all values
3      format long e
4  clear;
5  clc;
6  close all;
7  %
8  addpath('MeshingClasses/')
9  addpath('AllFunctions/')
10  if exist('DetailsClass') ~ 2
11     addpath('MATLABTools/');
12  end
13
14  %--------------------------------------------
15  % CREATE MESH---------------------------------
16  % MeshType = 11; % 4 bulk + 2 cse's (horizontal)
17  % MeshType = 13; % 4 bulk + 2 cse's (vertical)
18  MeshType = 14; % 64 bulk + 8 cse's (vertical) + vertical pressure
19  % MeshType = 140; % 64 bulk + 0 cse's (vertical) + vertical pressure
20  % MeshType = 15; % 2268 bulk + 162 cse's
21  % MeshType = 16; % 64 bulk + 8 cse's (horizontal)
22  % MeshType = 17; % 64 bulk + 8 cse's (horizontal) + fluid flux at left
23  % MeshType = 18; % 64 bulk + 8 cse's (vertical) + vertical pressure + 0 CSE ...
24     pressure
25  PDef = ProbDefManager(MeshType);
26  V = AI(PDef);  % hold all important variables
27
28  % Flux into element, specified inside of ProbDef14/16
29  Sflux = V.Sflux;
30
31  % V.NDCh.PlotNodeIDsG = false;
32  % PDef.PlotAll; % uncomment to plot the mesh
33  % CREATE MESH----------------------------------
34
35  %-----------------------------------------------
36  % TIME STEPS------------------------------------
37  %
38  dt = 1e-5;
39  time = 7e-3;
40  % time_steps = round(time/dt);
41  % time_steps = 200; % for debugging
42  %
43  %-----------------------------------------------
44
45  % setup everything for plotting deformed structure-----------------
46  plotlive = false;
47  % plotlive = true;
48  % setup everything for plotting deformed structure-----------------
```
d = V.d; % pull out d from all info class handle
V.NDCh.d = d;

D = V.D; % isotropic plane strain elasticity matrix for solid skeleton

time_factor = V.time_factor;

% necessary parameters of cse
Kcse = V.Kcse;
cse_para = V.cse_para;
perm_cse_params = V.perm_cse_params;

% element and mesh info
nel_bulk = V.nel_bulk; % bulk element numbers
nel_cse = V.nel_cse; % cse numbers
nel = V.nel; % element numbers

% Degrees of freedom
neld_bulk_dof = 18; % displ dofs per bulk element
nelpf_bulk_dof = 4; % num el bulk poro DOF's (pore pressure)
nelpf_cse_dof = neld_bulk_dof + nelpf_bulk_dof; % num bulk el DOF's

neld_cse_dof = 12;
nelpf_cse_dof = 4; % num el cse poro DOF's (pore pressure)
nel_cse_dof = neld_cse_dof + nelpf_cse_dof; % num el cse DOF's

nporoDOF = V.nporoDOF; % number of poro pressure DOF's

%press dof along CSE
ndof = V.ndof; % total dofs for mesh

zero4x18=zeros(nelpf_bulk_dof,neld_bulk_dof);
zero18x18=zeros(neld_bulk_dof,neld_bulk_dof);
zero18x4=zeros(neld_bulk_dof,nelpf_bulk_dof);
zero18x1=zeros(neld_bulk_dof,1);
zero4x4=zeros(nelpf_bulk_dof,nelpf_bulk_dof);
zero4=zeros(nelpf_bulk_dof,1);
zero4x6=zeros(nelpf_cse_dof,neld_bulk_dof-neld_cse_dof);

Fext = V.Fext; % pressure on top nodes
for i = 1:nel Bulk
    kdd = kdd_qdr_quad(coords(i,:),D);
ktheta(:,i) = ktheta_qdr_quad(coords(i,:));
kthth(:,i) = kthth_quad(coords(i,:),V.kp);
Kk(:,i) = [ kdd kthth(:,i) ; zero4x18 kthth(:,i) ];
Cc(:,i) = [ zero18x18 zero18x4 ; kthth(:,i)' zero4x4 ];
fdt(:,i) = V.EMCh.Elems(i).Fextxy;
end

% initialize necessary vectors and matrices
% d = zeros(ndof, 1); % total nodal displacements and pore pressures
d_el = zeros(nel, nel_bulk_dof); % element displacements and pore pressures at...
% current time step
state_n = zeros(V.n_cse_ip, V.n_cse_isv, nel_cse);
% state(i,1) = T_t; % state(i,2) = T_n;
% state(i,3) = u_t; % state(i,4) = u_n;
% state(i,5) = up_t; % state(i,6) = up_n;
% state(i,7) = epsp_s; % state(i,8) = epsp_n;
% state(i,9) = chi; % state(i,10) = c;
% state(i,11) = tan(phi); % state(i,12) = tan(psi);
% state(i,13) = F; % state(i,14) = dlam;
% state(i,15) = iplastic; % state(i,16) = normr;
% state(i,17) = kk;

% calculate initial internal state variables
flag = 0; % 0: calculate the initial internal state variables
begin cse element loop
    % calculate "damping" matrix
    Cc(:,1+nel_el) = zeros(nel_bulk_dof,nel_bulk_dof);
ktheta_cse_el(:,1+nel_el) = cse_el_ktheta(coords(1+nel_el,:), n_cse_ip);
Cc(:,1+nel_el) = [ zero18x18 zero18x4 ; ktheta_cse_el(:,1+nel_el)' zero4x4 ];

% calculate the coordinate transformation Q
[Q_cse_el(:,1+el), normal_cse_el(:,1+el)] = Q_f(coords(1+nel_el,:));

% calculate local jump displacements of cse at each ip
d_cse_local_ip = d_cse_local_f(d_el(1+nel_bulk,1:12), Q_cse_el(:,1+el), ... n_cse_ip);

% calculate state variables
state_n(:,1+el) = cse2D(Kcse, d_cse_local_ip, state_n(:,1+el), cse_para, ... flag, n_cse_ip, n_cse_isv);

% calculate initial stiffness of cse
Kk(:,1+nel_el) = zeros(nel_bulk_dof,nel Bulk_dof);
Kk(1:12,1:12,1+nel_el) = cse_element_stiffness(...
coords(el+nel_bulk,1:12),
Kcse,...
del1+nel_bulk,1:12),...
state_n(:,el),...
cse_para,...
flag,...
n_cse_ip,...
end %end cse element loop

flag = 1;
% 1: after the initial, internal state variables are ...
calculated

force_flag = 0;
% control how to determine the force of cse
% 0: use state_n to calculate the cse force

% initialize
n_bulk_stress=3;
n_bulk_ip=9;
for el=1:nel_bulk,
  stress el(:,el)=zeros(n_bulk_ip,n_bulk_stress);
  strain el(:,el)=zeros(n_bulk_ip,n_bulk_stress);
  pf el(:,1,el)=zeros(n_bulk_ip,1);
end

% Preallocate
D = zeros(ndof,ndof);
K = zeros(ndof,ndof);
C = zeros(ndof,ndof);
Ft = zeros(ndof,1);
Fint = zeros(ndof,1);
% Fext = zeros(ndof,1);
R0 = zeros(ndof, 1);
R = zeros(ndof, 1);
% initialize bulk elements
for el=1:nel_bulk,
  del epsilon el,el)=zeros(n_bulk_ip,1);
  porosity el,el)=ones(9,1)*V.nF0;
  porosity el last,el)=porosity el,el);
  perm el,el)=ones(9,1)*V.perm;
  density el,el)=ones(9,1)*V.init dens;
  fdint (:,el)=fdint_quad(coords1,el,:),stress el(:,el));
  fthint(:,el)=element fluid int force(coords1,el,:),del el (neld bulk dof+1):nel bulk dof),perm el,el);
  fdf(:,el)=ddf qdr_quad(coords1,el,:),density el,el,:),V.grav); % passes in ...
  gravity (in y direction)
  Fdthint (:,el)= [ fdint (:,el) - ...
  ktheta(:,el)*d el (neld bulk dof+1):nel bulk dof)'; -fthint(:,el) ];
  Fdthext (:,el)= [ fdf(:,el)+fdt (:,el); zero4 ]; % gravity
  %Fdthext (:,el)= [ fdf (:,el); zero4 ];
  temp=LM(:,el);
  for i=nel bulk dof, I=temp(i);
  if I>0
    Fint(I)=Fint(I)+Fdthint(i,el);
Fext(I)=Fext(I)+Fdthext(i,el);
for j=1:nel_bulk_dof,
    J=temp(j);
    if J>0
        C(I,J)=C(I,J)+Cc(i,j,el);
        K(I,J)=K(I,J)+Kk(i,j,el);
    end
end
for i=1:nel_bulk_ip
    Dep_el(:,:,i,el)=D;
end
% INITIALIZE COHESIVE SURFACE ELEMENTS----------------------------------------
for el=(nel_bulk+1):nel,
    del_epsv_el(el,:)=zeros(nel_bulk_ip,1);
    porosity_el_last(el,:)=porosity_el(el,:);
    perm_el(el,:)=ones(9,1)*V.perm;
    density_el(el,:)=ones(9,1)*V.init_dens;
    %forces
    Fcse_el(:,el) = cse_element_force(coords(el,:), Kcse, d_el(1,:), ...,  
        state_n(:,el-nel_bulk), cse_para, ...,  
        flag, force_flag, n_cse_ip, n_cse_isv);
    kdth_cse_el(:,el) = cse_el_kdth(coords(el,:), n_cse_ip);
    fth1_cse_el(:,el) = fth1_mult*cse_el_fth1(coords(el,:), d_el(1,:), ...,  
        n_cse_ip, perm_cse_params);
    fth2_cse_el(:,el) = fth2_mult*cse_el_fth2(coords(el,:), d_el(1,:), ...,  
        n_cse_ip, perm_cse_params);
    Fcsedthint(:,el)= [ ...  
        Fcse_el(:,el)-kdth_cse_el(:,el)*d_el(1,(neld_bulk_dof+1):nel_bulk_dof)' ...  
        zeros(6,1); -fth1_cse_el(:,el)-fth2_cse_el(:,el) ];
    temp=LM(:,el);
for i=1:nel_bulk_dof,
    I=temp(i);
    if I>0
        Fint(I)=Fint(I)+Fcsedthint(i,el);
        for j=1:nel_bulk_dof,
            J=temp(j);
            if J>0
                C(I,J)=C(I,J)+Cc(i,j,el);
                K(I,J)=K(I,J)+Kk(i,j,el);
            end
        end
    end
end
%initialize time solution parameters
%all time parameters in seconds or days
    t = 0;
    time_tot = 7e-3; %days
% drained:
% dt1=1e-4
% fprintf('
 dt1: %2.3e\n',dt1);
% timel=7e-3;
% time_steps1=round(timel/dt1)
% %
% consolidating:
% NOW TIME STEPS ARE DEFINED AT START OF PROGRAM
% dt = 1e-5;
% time = 7e-3;
% time_steps = round(time/dt);
% time_steps = 50; %for debugging
% %
% flow only:
% dt1=1e-6
% fprintf('
 dt1: %2.3e\n',dt1);
% timel=5e-5;
% time_steps1=round(timel/dt1)
% time_steps1=3 %for debugging

iter_break = 20; % limiting number of iterations
increment_force_displ = 200;

% dimensionalize arrays for saving for output and plotting
% time_output_increments = 100;
% time_output_increments = time_steps;
% time_output_increment = round(time_steps/time_output_increments);
Rtol_save=zeros(time_output_increments,iter_break);
Atol_save=zeros(time_output_increments,iter_break);
dg = zeros(time_output_increments+1, 1);
Fsolve = zeros(time_output_increments+1, nnodes * 2 - ndispDOF);
state_solve = zeros(n_cse_ip, n_cse_isv, nel_cse, time_output_increments+1);
Tsolve = zeros(time_output_increments+1, 1);
Tsolvev = zeros(time_output_increments+1, 1);
dsolve=zeros(time_output_increments+1, ndof);
dsolve(1,:)=d; %set to initial
tsolve=zeros(time_output_increments+1,1);
stress_solve= zeros(n_bulk_ip,n_bulk_stress,nel_bulk,time_output_increments+1);
strain_solve= zeros(n_bulk_ip,n_bulk_stress,nel_bulk,time_output_increments+1);
pf_solve=zeros(n_bulk_ip,1,nel_bulk,time_output_increments+1);

t_ramp=1/time_factor; %s, d, ...
tolr = 1e-10;
tola = 1e-8;

alpha = 1.0; %Backward Euler
dt = dt; % time step size
dR = C + alpha*dt*K;

% solve for nodal displacements and pore pressures
% initial condition
 d_last=d;
 Delta_d=zeros(ndof,1);
v=zeros(ndof,1);
% applied displacement; none here
  displ = 0;
% G matrix of prescribed displacement
G_n = zeros(nel_bulk_dof, nel);
G_final = g*displ;

% more zeros matrices
zero12x6 = zeros(12, 6);
zero6x18 = zeros(6, 18);
zero6x4 = zeros(6, 4);
zero4x6 = zeros(4, 6);

if plotlive % if plotting live
  mov(1:(time_steps+1)) = struct('cdata', [],
  'colormap', []);
  V.NDCh.exag = 15; % amount to exaggerate displ. by
  PDef.PlotLive;
  Stitle = sprintf('Time Step %u', 0);
  ht = title(Stitle);
  set(ht, 'FontSize', 18);
  colorbar;
  mov(1) = getframe;
else % if false, show a status bar
  hwb = waitbar(0, 'Finding Solution');
end

% Setup details class to log parameters
DC = DetailsClass;
  DC.LivePrint = true; % true to print details as entered
  DC.FName = 'RunSummary'; %
  DC.NameInt('Mesh Type', MeshType);
  DC.NameInt('time_factor', time_factor);
  DC.NameVal('stress_factor', V.stress_factor);
  DC.NameInt('# of bulk elements', nel_bulk);
  DC.NameInt('# of CS elements', nel_cse);
  DC.NameInt('Degrees of Freedom', ndof);
  DC.NameInt('Total Number of nodes', nnodes);
  DC.NameVal('perm0sat', V.perm0sat);
  DC.NameInt('Time Steps', time_steps);
  DC.NameVal('Time Steps Size', dt);
  DC.NameInt('t_ramp', t_ramp);
  DC.NameInt('Displ. Exaggeration', V.NDCh.exag);
  DC.NameVal('Max specified flux', max(Sflux(:)));
  DC.NameVal('Aperture', AI.aperture);

% Setup details class to log parameters

Delta_del = zeros(nel, nel_bulk_dof);
%temp - just until compatibility added with AI
  perm = V.perm;
  grav = V.grav;
rhoS = V.rhoS;
rhoF = V.rhoF;

% time_output.increment_counter = 1; %initialize to 1
tic;
% Delta_d_el = d_el - d_el_last;
% d_el_last = V.EMCh.d_el;

% Newton Raphson Loop
for n = 1:time_steps

n;

t = t + dt;
dtilde = d + (1 - alpha)*dt*v;
if t<t_ramp
    ratio=t/t_ramp;
else
    ratio=1.0;
end

CSE6Aperture(n) = V.EMCh.Elems(72).aperture;
CSE6Time(n) = t;
fprintf('\n Aperture of CSE 6: %2.3e',CSE6Aperture(n));

%ratio_grav=ratio;
ratio_grav=1.0;
G = G_final*ratio_grav;
A = displ*ratio;
%
% dR = zeros(ndof,ndof);
Fint = zeros(ndof,1);
Fext = zeros(ndof,1);

for el = 1:nel
    temp=LM(:,el);
    for i=1:nel bulk dof,
        I=temp(i);
        if I>0 % if DOF exists
            d_el(el,i) = d(I);
            Delta_d_el(el,i) = Delta_d(I);
        else
            d_el(el,i) = G(i,el);
            Delta_d_el(el,i) = G(i,el)-G_n(i,el);
        end
    end
    coords(el,:)=coords0(el,:)+d_el(el,1:18);
end

%-----------------------------
% BULK ELEMENTS

% MOVED OUTSIDE OF BULK ELEMENT LOOP
del_eps_v_el(1:nel bulk, :) = BKLP.element_del_vol_quad(coords, Delta_d_el, ...
    neld bulk dof, nel bulk);
porosity_el(1:nel_bulk, :) = BKL.element_porosity_quad(del_epsv_el, ...
    porosity_el_last, nel_bulk);
perm_el(1:nel_bulk, :) = BKL.element_perm_quad(porosity_el, perm, nel_bulk);
density_el(1:nel_bulk, :) = BKL.element_density_quad(porosity_el, rhoS, ...
    rhoF, nel_bulk);
[stress_el, strain_el] = BKL.elementstress_quad(coords, D, d_el, nel_bulk);
fdf = BKL.fdf_quad(coords, stress_el, nel_bulk);

% assemble components of Fint and dR for bulk elements
for el=1:nel_bulk
    temp=LM(:,el);
    for i=1:nel_bulk_dof,
        I=temp(i);
        if I>0
            Fint(I)=Fint(I)+Fdthint(i,el);
            Fext(I)=Fext(I)+Fdthext(i,el);
            for j=1:nel_bulk_dof,
                J=temp(j);
                if J>0
                    dR(I,J)=dR(I,J)+dFintdD(i,j)+dFextdD(i,j);
                end
            end
        end
    end
end

% BULK ELEMENTS

% CS Elements
F_cse_el = CSLP.cse_element_force(coords, Kcse, d_el, state_n, cse_para, ...
flag, force_flag, n_cse_ip, ...
    n_cse_isv, ...
    nel_bulk, nel_cse);
kdth_cse_el = CSLP.cse_el_kdth(coords, n_cse_ip, nel_bulk, nel_cse);
% tangents

% CSE loop replacement

% calculate the combination of F_int and F_cse

for el=(nel_bulk+1):nel,
    % forces
    Fcsedthint(:,el) = [ ...
        F_cse_el(:,el)-kdth_cse_el(:,el)*d_el(el,(nel_bulk_dof+1):nel_bulk_dof)'; ...
        zeros(6,1); -fth1_cse_el(:,el)-fth2_cse_el(:,el) ];
    dFintdCse=[ dfdcseintdd(:,el)    zero12x6 ...
        -kdth_cse_el(:,;el) ... 
        zero6x18    zero6x4; ...
        -dth1csedd(:,;el)-dth2csedd(:,;el) zero4x6 ... 
        -dth1csedd(:,;el)-dth2csedd(:,;el) ];
    dFextdCse=[ zero18x18    zero18x4; zero4x18    zero4x4 ];
    temp=LM(:,el);
    % calculate the combination of F_int and F_cse
    for i=1:nel_bulk_dof,
        I=temp(i);
        if I>0
            Fint(I) = Fint(I) + Fcsedthint(i,el);
            Fext(I) = Fext(I) + Fcsedthext(i,el);
            for j=1:nel_bulk_dof,
                J=temp(j);
                if J>0
                    dR(I,J)=dR(I,J)+dFintdCse(i,j)-dFextdCse(i,j);
                end
            end
        end
    end %end cse element loop

dR=alpha*dt*dR+C;
R0=C*v+Fint-Fext;
%R0=Fint-Fext;
R=R0;
normR=1;
Rtol=1;
k=0;
if n==time
output
increment
counter *time
output
increment

Rtol
save(time
output
increment
counter,k+1)=Rtol;
Atol
save(time
output
increment
counter,k+1)=normR;
end
for el=1:nel,
    porosity_el_last (el,:)=porosity_el (el,:);
end
force_flag = 1; % 1: use state to calculate the cse force
%
while (Rtol > tolr) && (normR > tola) %
k = k + 1;
del_v = dR \(-R);
v = v + del_v;
d = dtilde + alpha*dt*v;
Delta_d = d - d_last;
dR = zeros(ndof, ndof);
Fint = zeros(ndof, 1);
Fext = zeros(ndof, 1);
%
build_d_el and Delta_d_el matrices from vectors
for el = 1:nel
    temp = LM(:, el);
    for i = 1:nel_bulk_dof,
        I = temp(i);
        if I>0
            d_el(el, i) = d(I);
            Delta_d_el(el, i) = Delta_d(I);
        else
            d_el(el, i) = G(i, el);
            Delta_d_el(el, i) = G(i, el) - G.n(i, el);
        end
    end
end
% BULK ELEMENTS

del_epsv_el(1:nel_bulk, :) = BKLP.element_del_vol_quad(coords, ...
    Delta_d_el, nel_bulk_dof, nel_bulk);
porosity_el(1:nel_bulk, :) = BKLP.element_porous_quad(del_epsv_el, ...
    porosity_el_last, nel_bulk);
perm_el(1:nel_bulk, :) = BKLP.element_perm_quad(porosity_el, perm, ...
    nel_bulk);
density_el(1:nel_bulk, :) = BKLP.element_density_quad(porosity_el, ...
    rhoS, rhoF, nel_bulk);
[stress_el, strain_el] = BKLP.element_stress_quad(coords, D, d_el, ...
    nel_bulk);
fdint = BKLP.fdint_quad(coords, stress_el, nel_bulk);
pf_el = BKLP.element_pf_quad(d_el, nel_bulk);
fthint = BKLP.element_fluid_int_force(coords, d_el, perm_el, rhoF, ...
    grav, nel_bulk, nel_bulk_dof, nel_bulk_dof);
fdf = BKLP.fdf.qdr_quad(coords, density_el, grav, nel_bulk);
Fthint = BKLP.F_Bulk_Combine01(fdint, kdtheta, d_el, fthint, ...
    nel_bulk, nel_bulk_dof, nel_bulk_dof);
Fdhext(1:nel_bulk_dof,:) = ratio_grav*fdf + ratio*fdt;
dfdintdd = BKLP.dfdintdd_quad(coords, Dep_el, nel_bulk);
dfthintdd = BKLP.dfthdd_fluid_int(coords, del_epsv_el, ...
    porosity_el, rhoF, grav, nel_bulk, nel_bulk_dof, nel_bulk_dof);
dfthintdth = BKLP.dfthd_th_fluid_int(coords, del_epsv_el, ...
    rhoF, grav, nel_bulk);
fdfdd = BKLP.dfdff_quad(coords, porosity_el, del_epsv_el, ...
    rhoF, rhoS, nel_bulk);
dFintD = [ dfdintdd -kdtheta;
    -dfthintdth];
dFextD = zeros(nel_bulk_dof, nel_bulk_dof, nel_bulk);
dFextD(1:neld_bulk_dof, 1:neld_bulk_dof,:) = fdfdd;

% begin bulk element loop
for el=1:nel_bulk,
    temp=LM(:,el);
    % assemble components of Fint and dR for bulk elements
    for i=1:neld_bulk_dof,
        I=temp(i);
        if I>0
            Fint(I)=Fint(I)+Fdthint(i,el);
            Fext(I)=Fext(I)+Fdthext(i,el);
            for j=1:neld_bulk_dof,
                J=temp(j);
                if J>0
                    dR(I,J)=dR(I,J)+dFintdD(i,j) - dFextdD(i,j);
                end
            end
        end
    end
end

% BULK ELEMENTS

% COHESIVE SURFACE ELEMENTS
fcth1cse_el = fth1_mult*CSLP.cse_el_fth1(coords, del_epsv_el, ...
    perm_cse_params, nel_bulk, nel_cse);
fcth2cse_el = fth2_mult*CSLP.cse_el_fth2(coords, del_epsv_el, ...
    perm_cse_params, nel_bulk, nel_cse);
Fcsedthext = fth2_mult*CSLP.cse_el_fth2dth(coords, del_epsv_el, ...
    perm_cse_params, Sflux, nel_bulk, nel_cse);

% tangents
dfdcseintdd = CPL.cse_element_stiffness(coords, Kcse, del_epsv_el, ...
    cse_para, flag, force_flag, n_cse_ip, ...
    n_cse_isv, nel_bulk, nel_cse);
dfth1csedd = fth1_mult*CSLP.cse_el_dfth1dth(coords, del_epsv_el, ...
    perm_cse_params, nel_bulk, nel_cse);
dfth2csedd = fth2_mult*CSLP.cse_el_dfth2dth(coords, del_epsv_el, ...
    perm_cse_params, nel_bulk, nel_cse);
dfth2csedd = fth2_mult*CLP.cse_el_dfth2dth(coords, d.el, n.cse_ip, ...
perm.cse_params, nel_bulk, nel_cse);

for el=(nel_bulk+1):nel
  %forces
  Fcsedthint(:,el)= [ ...
    F_cse_el(:,el)-kdth_cse_el(:,el)*d.el(el,(neld_bulk_dof+1):nel*bulk_dof)'
    ...
    zeros(6,1) ; ...
    -fth1_cse_el(:,el)-fth2_cse_el(:,el) ];
  dFintdDcse=[ dfdcestdd(:,el)
    zero12x6 ...
    -kdth_cse_el(:,el) ; ...
    zero6x18 ...
    -dflhcsedd(:,el)-dfth2csedd(:,el) zero4x6 ...
    -dfth1csedd(:,el)-dfth2csedd(:,el) ];
  dFextdDcse=[ zero18x18 zero18x4 ; zero4x18 zero4x4 ];
%
assemble
  temp=LM(:,el);
  % calculate the combination of F_int and F_cse
  for i=1:nel_bulk_dof,
    I=temp(i);
    if I>0
      Fint(I) = Fint(I) + Fcsedthint(i,el);
      Fext(I) = Fext(I) + Fcsedthext(i,el);
      for j=1:nel_bulk_dof,
        J=temp(j);
        if J>0
          dR(I,J)=dR(I,J)+dFintdDcse(i,j)-dFextdDcse(i,j);
        end
    end
  end
%
COHESIVE SURFACE ELEMENTS—------------------------------
%

  dR = alpha*dt*dR + C;
  R=C*v + Fint - Fext;
  if (k==1) & (n >= 1)
    R0 = R;
  end
  Rtol = norm(R)/norm(R0);
  normR = norm(R);
  if n == time_output_increment_counter*time_output_increment
    Rtol_save(time_output_increment_counter,k+1)=Rtol;
    Atol_save(time_output_increment_counter,k+1)=normR;
  end
  if k == iter_break
    error('reached max number of iterations')
  end
end %iteration loop
d_last = d;
G.n = G;
if n==time
output
increment
counter *time
output
increment
end
dsolve(time
output
increment
counter+1,:)=d;
tsolve(time
output
increment
counter+1)=t;
stress.solve(:, :, time.output.increment.counter+1)=stress.el;
strain.solve(:, :, time.output.increment.counter+1)=strain.el;
 pf.solve(:, :, time.output.increment.counter+1)=pf.el;
end

%begin cse element loop
for el=1:nel_cse,
  d_cse_local = d_cse_local.f(d_el(el+nel_bulk,:), Q_cse_el(:, :, el), ...
  n_cse.ip);
  state.n(:, :, el) = cse_2D(Kcse, d_cse_local, state.n(:, :, el), cse_para, ...
  flag, n_cse_ip, n_cse.isv);
end %end cse element loop

%save at each time output increment
if n==time.output.increment_counter*time.output.increment
  dg(time.output.increment.counter+1)=A;
  Freact = Freact.f(Kk, G, LM, nel, nel_bulk_dof, ndispDOF, nnodes, d_el);
  Fsolve(time.output.increment_counter+1,:) = Freact;
  state.solve(:, :, time.output.increment_counter+1) = state.n;
  time.output.increment_counter = time.output.increment_counter + 1;
end

force_flag = 0;  % 0: use state.n to calculate the cse force

V.NDCh.t = t;
V.NDCh.d = d;

if plotlive % if plotlive is true, plot the current displacement
  PDef.PlotLive;
  Stitle = sprintf('Time Step %d', n);
  ht = title(Stitle);
  set(ht, 'FontSize', 18);
  colorbar;
  mov(n+1) = getframe;
else % if not plotting live, update the status bar
  waitbar((n/time_steps), hwb);
end

end %time step loop

if plotlive == false % close status bar
  close(hwb)
end
DC.NameTime('Run Time', toc)
DC.WriteDetails;

% V.MeshDef.PlotData(V);

%% Plot results

% WRITE MOVIE OF DEFORMATION WITH TIME
if plotlive % if plotlive is true, plot the current displacement
  Mname = sprintf('Mesh_%u_Tsteps_%u_Aperture%1.2e.avi', MeshType, time_steps, ...
                 AI.aperture);
  movie2avi(mov, Mname, 'compression', 'None');
end

%% save results

%global dofs
d;

figure;

nodecse = 149; % node on cse
nodeblk = 47; % node on bulk element
pcse = V.NDCh.NodePt(nodecse,:);
pblk = V.NDCh.NodePt(nodeblk,:);
T = V.NDCh.T;
plot(T,pcse,T,pblk)

% SAVE FILES TO DIRECTORY IN V, where class(V) = AI
V.WriteFile('block_cse_steps',time_outputIncrements)
V.WriteFile('block_cse_t',tsolve)
V.WriteFile('block_cse_d',dsolve)
V.WriteFile('block_cse_dfinal',d)
V.WriteFile('block_cse_d',time_outputIncrements)

Rtol_end = Rtol_save';
Atol_end = Atol_save';
state_end = state_n;

%plot yield surface
Tn = -1e5:100:1.6e4;

%initial
Fyieldfunc0 = (cse_para(8) - Tn*tan(cse_para(9))).^2 - (cse_para(8) - ...
               cse_para(7)*tan(cse_para(9))).^2;
Fyield0 = 0.5*(Fyieldfunc0+abs(Fyieldfunc0));
T0 = sqrt(Fyield0);
T0neg = -sqrt(Fyield0);

%final at ip of cse
ip=1; %cse ip
el=1; %cse
Fyieldfunc = (state_solve(ip,10,el,time_outputIncrements+1) - ...
             Tn*state_solve(ip,11,el,time_outputIncrements+1)).^2 - ...
             (state_solve(ip,10,el,time_outputIncrements+1) - ...
             state_solve(ip,9,el,time_outputIncrements+1)*state_solve(ip,11,el,time_outputIncrements+1));
Fyield = 0.5*(Fyieldfunc+abs(Fyieldfunc));
Tt = sqrt(Fyield);
\[ T_{\text{neg}} = -\sqrt{F_{\text{yield}}}; \]

\[
\text{at ip 1}
\]

\[ T_{\text{solve}}(:,1) = \text{state \_solve(ip,1,el,:)}; \]

\[ T_{\text{solve}}(:,2) = \text{state \_solve(ip,2,el,:)}; \]

\[
\text{for } n=1:\text{time \_output \_increments}+1
\]

\[ \text{plot \_state \_solve \_trans(:,n)} = \text{state \_solve(ip,:,el,n)}; \]

\[
\text{end}
\]

\[ \text{plot \_state \_solve} = \text{plot \_state \_solve \_trans}; \]

\[
\text{% save('block\_cse\_state.txt', 'plot\_state\_solve', '-ASCII');}
\]

\[ \text{V.WriteFile('block\_cse\_state', plot\_state\_solve)} \]

\[
\]

\[ \text{el=1;} \]

\[ \text{ip=1;} \]

\[ \text{stress=2; %plot effective stress in y direction} \]

\[ \text{plot\_stress} = \text{zeros(time\_output\_increments}+1,1); \]

\[ \text{plot\_strain} = \text{zeros(time\_output\_increments}+1,1); \]

\[ \text{plot\_pf} = \text{zeros(time\_output\_increments}+1,1); \]

\[
\text{for } n=1:\text{time \_output \_increments}
\]

\[ \text{plot\_stress}(n+1) = \text{stress \_solve(ip,} stress,el,n+1); \]

\[ \text{plot\_strain}(n+1) = \text{strain \_solve(ip,} stress,el,n+1); \]

\[ \text{plot\_pf}(n+1) = \text{pf \_solve(ip,} 1,el,n+1); \]

\[
\text{end}
\]

\[
\text{% SAVE FILES}
\]

\[ \text{V.WriteFile('block\_cse\_stress', plot\_stress)} \]

\[ \text{V.WriteFile('block\_cse\_strain', plot\_strain)} \]

\[ \text{V.WriteFile('block\_cse\_pf', plot\_pf)} \]

\[
\text{% CALL SCRIPT TO PLOT ALL RESULTS}
\]

\[ \text{block\_cse\_subplot} \]
B.3 General Data Handling, Manipulating, and Plotting

B.3.1 Node Definition Class

classdef NodeDefClass < handle

% class to handle all info associated with each node
% 11.03.2012 – added ability to plot pressure dof #
% 11.12.2012 – fixed error where center nodes would have aperture added
% 11.23.2012 – Adding in ability to plot displacement and pressure

properties

NN; % current node

% DOFs---------------------------------------------------------------
C dof = 0; % current dof, initialize to = 0 (disp. or press.)
NPdofs = 0; % Number of pressure DOFs, initiate to zero
Nddofs = 0; % number of displacement DOF's, initiate to zero

%---------------------------------------------------------------

X0; Y0; % Initial coordinates

% displacement info
% if displ fixed, g of node = 0 and LM of node = 0
Xdf; Ydf; % is node fixed? true/false
Xdp; Ydp; % is displ. prescribed? true/false
Xdof; Ydof; % DOF associated with node
UX; UY; % x and y displacement at node <nnodes x 1>

% Variables for recording the entire solution
Ux; Uy; % node pressure at each time step
NodePt; % x and y displacements at each time step
t; % vector of times
c_tstep = 0; % current time step (0 in initial state)

% Forces on nodes
XF; YF; % x and y forces on nodes

% Pressure properties------------------------------------------
PressEnable; % logical, false—->no pressure set/calculated at node
PressDisable; % disable pressure at location at all costs
Pdof; % Pressure DOF’s ID (actual DOF number)

% Pressure on nodes-------------------------------------------
% trying to replace everything below (11.07.2012)
NodeP0; % prescribed initial pressure

Pbc; % Pressure BC, if true, goto same spot in NodeP for val
NodeP; % value of pressure

% time step info--------------------------------------------------
tsteps = 200; % number of time steps (to preallocate)
end

properties (Hidden)
NN_est = 10e3; % estimated number of nodes for preallocation
DupNodeIDs; % if node was duplicated, this is id to new node
\% PlotInfo
PlotNodeIDsG = true; % true/false to plot any given node id (master)
\% if false, NO node IDs will be plotted
PlotNodeIDs; % true/false to plot each node ID <nnodes x 1>
PlotNodeConsG = true; % true/false, false = never plot constraints EVER
PlotNodeCons; % true/false on node by node basis, false = no plot
PlotNodePDOFs = false; % true = label node pressure dof
Nodexos;
Nodeyos;
OS = 6; % general plotting offset (e.g. offset = o.xos/o.OS)
\% Graph Legend Info
Nh; % node handle
XFh; % handle to symbol for x being fixed
YFh; % handle to symbol for y being fixed
Pdofh; % handle to symbol for fixed pressure
Pbch; % handle to pressure boundary condition
DNh; % handle to double noded symbol
\% end
\% properties (Dependent, Hidden)
xos; % x plotting offset
yos; % y plotting offset
XFmax; % max x force
YFmax; % max y force
\% NodePbc; % node pressure BC,
\% end
\% properties (Dependent)
Fext; % external force <ndof x 1>
d; % initial pressure <ndof x 1>
X; Y; % current position of nodes with exaggeration
t; % current time, gets assigned to Times
\% methods
function o = NodeDefClass
\% LoadDefaults;
end
function LoadDefaults(o)
\% Preallocate all possible arrays
n = o.NN_est; % estimated number of nodes
o.NN = 0;
o.Nh = zeros(n,1);
o.X0 = zeros(n,1); o.Y0 = zeros(n,1);
o.XF = zeros(n,1); o.YF = zeros(n,1);
o.Xdf = false(n,1); o.Ydf = false(n,1);
o.Xdp = false(n,1); o.Ydp = false(n,1);
o.Xdof = zeros(n,1); o.Ydof = zeros(n,1);
o.Ux = zeros(n,1); o.Uy = zeros(n,1);
o.Uxt = zeros(n, o.tsteps+1); o.Uyt = zeros(n, o.tsteps+1);
o.NodePt = zeros(n, o.tsteps+1);
o.T = zeros(o.tsteps + 1, 1);
o.DupNodeIDs = zeros(n,1);

% new pressure info
o.Pdof = zeros(n,1); % pressure DOF for node

% disable all pressure at all nodes (elements will enable pressure
% at the correct nodes when initiated)
o.PressEnable = false(n,1); % false--> no pressure at node

% specify which nodes have a fixed pressure.
o.Pbc = false(n,1); % nodes with pressure fixed

% Pressure info
o.NodeP = zeros(n,1); % value of pressure bc
o.NodeP0 = zeros(n,1); % nodes with initial pressure

% Plotting Info
o.PlotNodeIDs = false(n,1); % if plot nodes, true--> plot node ID's 2
o.Nodexos = zeros(n,1); % x offset of nodes
o.Nodeyos = zeros(n,1); % y offset of nodes
o.PlotNodeCons = false(n,1);

function NodeIDs = AddNodes(o, X, Y, Xdf, Ydf, Xdp, Ydp)
% length of x = n
% X/Y = X/Y coordinate vectors of length n
% Xdf/Ydf = true/false for displacement being fixed
% Xdp/Ydp = if displacement prescribed
N = numel(X);
NodeIDs = zeros(N,1);
for n = 1:N
    NodeIDs(n) = ...
    o.AddNode(X(n), Y(n), Xdf(n), Ydf(n), Xdp(n), Ydp(n));
end
% distance to offset label by
o.Nodexos(1:N) = -1*(o.xos/o.OS); % in x direction
o.Nodeyos(1:N) = -1*(o.yos/o.OS);

function NewNodeIDs = DuplicateNodes(o, NodeIDs, dx, dy)
% duplicate the nodes specified by NodeIDs and shift the new
% nodes by (dx, dy) ONLY if nodes have not already been
% duplicated before
n = numel(NodeIDs);
NewNodeIDs = zeros(1,n);
for i = 1:n
    NewNodeIDs(i) = o.DuplicateNode(NodeIDs(i), dx, dy);
end

function NewNodeID = DuplicateNode(o, NodeID, dx, dy)
% create a new node with the same properties as NodeID
% dx and dy are the offsets from NodeID for the new node
% NOTE: these will NOT be applied if node has already been
% duplicated

% first, check to see if node has already been duplicated
DupNodeID = o.DupNodeIDs(NodeID);
if DupNodeID == 0 % if node has NOT been duplicated
  % copy all parameters from NodeID
  x = o.X0(NodeID); y = o.Y0(NodeID);
  xf = o.Xdf(NodeID); yf = o.Ydf(NodeID);
  xp = o.Xdp(NodeID); yp = o.Ydp(NodeID);
  % create the new node
  NewNodeID = o.AddNode(x, y, xf, yf, xp, yp);
  % set new node to have the same constraints on a pressure dof
  o.Pbc(NewNodeID) = o.Pbc(NodeID);
  o.NodeP(NewNodeID) = o.NodeP(NodeID);
  o.PressEnable(NewNodeID) = o.PressEnable(NodeID);
  % set new node id to have same offsets as NodeID
  o.Nodexos(NewNodeID) = o.Nodexos(NodeID);
  o.Nodeyos(NewNodeID) = o.Nodeyos(NodeID);
  % set node being copied to plot its ID to the left
  o.Nodexos(NodeID) = -2*o.xos/o.OS;
  % set new node to plot its id to the right
  o.Nodexos(NewNodeID) = o.xos/o.OS;
  % reference new node id to show that node has been duplicated
  o.DupNodeIDs(NodeID) = NewNodeID;
  % offset nodes of aperture
  % offset the new node by half amount specified
  o.ShiftNode(NewNodeID, dx/2, dy/2);
  % offset old node ID by half amount specified (-) direction
  o.ShiftNode(NodeID, -dx/2, -dy/2);
else % node has been duplicated
  NewNodeID = DupNodeID; % return the existing node id
end
% Inserted for testing — plot nodes that are copied and plot the
% resulting nodes
o.PlotNodeIDs(NodeID) = true; % plot node that was copied
o.PlotNodeIDs(NewNodeID) = true; % plot new node
end

function NodeID = AddNode(o, x, y, xdf, ydf, xdp, ydp)
  % x and y = coordinates
  % xdf and ydf --> true if node fixed, false if node free
  % xp and yp --> true if displ prescribed, false if not
  o.NN = o.NN + 1;
  NodeID = o.NN;
  % add properties to class
  o.X0(NodeID) = x; o.Y0(NodeID) = y;
  o.Xdf(NodeID) = xdf; o.Ydf(NodeID) = ydf;
  o.Xdp(NodeID) = xdp; o.Ydp(NodeID) = ydp;
  % add degrees of freedom
  o.AddNodeDOF(NodeID);
  % add plotting info to node
  o.PlotNodeIDs(NodeID) = true; % = true to plot node
  % let node plot constraints
  o.PlotNodeCons(NodeID) = true;
end
function AddNodeDOF(o, NodeID)
xf = o.Xdf(NodeID); yf = o.Ydf(NodeID);
xp = o.Xdp(NodeID); yp = o.Ydp(NodeID);
if (xp || xf) if node isn't prescribed or fixed in x direction
    if o.Xdof(NodeID) == 0 % node doesn't already have a dof
        o.Nddofs = o.Nddofs + 1; % increment # displ. DOF's
        o.Cdof = o.Cdof + 1;
        o.Xdof(NodeID) = o.Cdof;
    end
end
if (yp || yp) if node isn't prescribed or fixed in y direction
    if o.Ydof(NodeID) == 0 % node doesn't already have a dof
        o.Nddofs = o.Nddofs + 1; % increment # displ. DOF's
        o.Cdof = o.Cdof + 1;
        o.Ydof(NodeID) = o.Cdof;
    end
end
function AddNodePDOFs(o)
    % adds all pressure DOFs possible
    for node = 1:o.NN
        o.AddNodePDOF(node);
    end
end
function AddNodePDOF(o, NodeID)
    % add pressure degree of freedom to node NodeID
    pdof = o.Pdof(NodeID); % current pressure DOF (must = 0)
    pbc = o.Pbc(NodeID); % pressure bc
    penable = o.PressEnable(NodeID);
    % Only assign a dof if:
    % --Has not already been assigned
    % --There is no pressure BC
    if (pdof == 0) && (pbc == false) && (penable == true)
        if o.PressEnable(NodeID) % only if true
            o.NPdofs = o.NPdofs + 1; % increment # of pressure DOF's
            o.Cdof = o.Cdof + 1; % increment # of total DOF's
            o.Pdof(NodeID) = o.Cdof; % assign dof # to Pdof(NodeID)
        end
    end
end
function AddNodeForcesXY(o, NodeIDs, fxy)
    % adds forces when in the format of [x1 y1 x2....]
    temp = zeros(2,numel(fxy)/2);
    temp(:) = fxy(:);
    fx = temp(1,:);
    fy = temp(2,:);
    o.AddNodeForces(NodeIDs, fx, fy);
end
function AddNodeForces(o, NodeIDs, XForces, YForces)
    % add each force by node
    N = numel(NodeIDs);
    for i = 1:N
        o.AddNodeForce(NodeIDs(i), XForces(i), YForces(i));
    end
function AddNodeForce(o, NodeID, xF, yF)
    xf = o.Xdf(NodeID); yf = o.Ydf(NodeID); % if fixed or not
    if ~xf % if node is NOT fixed
        % force = current force + new force
        o.XF(NodeID) = o.XF(NodeID) + xF;
    end
    if ~yf % if node is NOT fixed
        % force = current force + new force
        o.YF(NodeID) = o.YF(NodeID) + yF;
    end
end

function MoveNodes(o, NodeIDs, Xnew, Ynew)
    % moves every node in NodeIDs to positions defined in Xnew and Ynew
    N = numel(NodeIDs);
    for i = 1:N
        o.MoveNode(NodeIDs(i), Xnew(i), Ynew(i));
    end
end

function MoveNode(o, NodeID, xnew, ynew)
    % move a node to position (xnew, ynew)
    o.X0(NodeID) = xnew; o.Y0(NodeID) = ynew;
end

function ShiftNodes(o, NodeIDs, dX, dY)
    % shifts every node in NodeIDs by [dX dY]
    N = numel(NodeIDs);
    if numel(dX) == 1
        dX = ones(N,1)*dX;
    end
    if numel(dY) == 1
        dY = ones(N,1)*dY;
    end
    for i = 1:N
        o.ShiftNode(NodeIDs(i), dX(i), dY(i));
    end
end

function ShiftNode(o, NodeID, dx, dy)
    % move a node to position (x + dx, y + dy)
    o.X0(NodeID) = o.X0(NodeID) + dx;
    o.Y0(NodeID) = o.Y0(NodeID) + dy;
end

function FixNodes(o, NodeID, Node0, Node1)
    % updates node NodeID to be on midpoint between nodes
    % Node0 and Node1
    xn = (o.X0(Node0) + o.X0(Node1))/2;
    yn = (o.Y0(Node0) + o.Y0(Node1))/2;
    o.MoveNode(NodeID, xn, yn)
end

function Trim(o)
% trims matrices to be the length of # of nodes
n = o.NN;
o.X0 = o.X0(1:n);
o.Y0 = o.Y0(1:n);
o.Xdf = o.Xdf(1:n);
o.Ydf = o.Ydf(1:n);
o.Xdp = o.Xdp(1:n);
o.Ydp = o.Ydp(1:n);
o.Xdof = o.Xdof(1:n);
o.Ydof = o.Ydof(1:n);
o.Ux = o.Ux(1:n);
o.Uy = o.Uy(1:n);
o.XF = o.XF(1:n);
o.YF = o.YF(1:n);
o.PressEnable = o.PressEnable(1:n);
o.Pdof = o.Pdof(1:n);
o.NodeP0 = o.NodeP0(1:n);
o.Pbc = o.Pbc(1:n);
o.NodeP = o.NodeP(1:n);
end
end

methods % set and get functions

function out = get.X(o)
% get current position of X to plot
out = o.X0 + (o.Ux).*o.exag;
end

function out = get.Y(o)
% get current position of Y to plot
out = o.Y0 + (o.Uy).*o.exag;
end

function out = get.xos(o)
p = 0.05;
xmin = min(o.X0); xmax = max(o.X0);
out = (xmax - xmin)*p;
end

function out = get.yos(o)
p = 0.05;
ymin = min(o.Y0); ymax = max(o.Y0);
out = (ymax - ymin)*p;
end

function out = get.XFmax(o)
out = max(abs(o.XF));
end

function out = get.YFmax(o)
out = max(abs(o.YF));
function out = get.Fext(o)
    out = zeros(o.Cdof,1);
    for NodeID = 1:o.NN % for each node
        x dof = o.Xdof(NodeID); y dof = o.Ydof(NodeID);
        if x dof ~= 0 % if x dof exists
            out(x dof) = o.XF(NodeID);
        end
        if y dof ~= 0 % if y dof exists
            out(y dof) = o.YF(NodeID);
        end
    end
end

function out = get.d(o)
    % get d vector (where d is value at each dof <ndof x 1>)
    out = zeros(o.Cdof,1);
    for node = 1:o.NN % for each node
        % node DOF's
        x dof = o.Xdof(node); % x dof (if it exist)
        y dof = o.Ydof(node); % y dof (if it exist)
        p dof = o.Pdof(node); % pressure dof (if it exist)
        if p dof > 0 % if p dof exists
            out(p dof) = o.NodeP0(node);
        end
        if x dof > 0 % if x dof exists
            out(x dof) = o.Ux(node);
        end
        if y dof > 0 % if y dof exists
            out(y dof) = o.Uy(node);
        end
    end
end

function set.d(o, val)
    % set the d vector (for plotting)
    % where val = current d vector
    % fill in values at each node
    d = val;
    for node = 1:o.NN % for each node
        x dof = o.Xdof(node); % x dof (if it exist)
        y dof = o.Ydof(node); % y dof (if it exist)
        p dof = o.Pdof(node); % pressure dof (if it exist)
        % x displacement
        if x dof > 0 % if x dof exists
            o.Ux(node) = d(x dof); % get value from d vector
        end
        % y displacement
        if y dof > 0 % if y dof exists
            o.Uy(node) = d(y dof); % get value from d vector
        end
        % pressure
        if p dof > 0 % if x dof exists
            o.NodeP(node) = d(p dof); % get value from d vector
        end
end

function set.t(o, val)
o.T(o.c.tstep + 1) = val;
end

function out = get.NodePbc(o)
N = numel(o.Pbc);
out = zeros(N, 1);
for i = 1:N
if o.Pbc(i) == true
out(i) = o.NodeP(i);
end
end
end

methods % plotting functions and node info function

function PlotNodes(o)
% plot all nodes in original positions (no displacements)
% update the pressure bc so that nodes without pressure aren't
% plotted
o.Pbc = o.Pbc & o.PressEnable;
for i = 1:o.NN
NodeID = i;
% fprintf('
Plotting node %u
', i);
o.PlotNode(NodeID);
end
o.PlotForces; % Plot all forces
o.AddLeg; % add legend
end

function PlotNode(o, NodeID)
N = NodeID;
x = o.X(N); y = o.Y(N); % position of node to plot
PlotNodeID = o.PlotNodeIDs(N) & o.PlotNodeIDsG;
% Determine offset to plot node to plot node's ID
% plot below and to the left by defualt but can be changed in the
% PlotConstraints function
if o.PlotNodeCons(N) && o.PlotNodeConsG
% plot constraints (whether or not node fixed in x/y)
o.PlotConstraints(NodeID, x, y);
else % if not plotting constraints, then plot * to mark node
% decides to plot/not to plot node id
nh = o.Nh(NodeID);
if nh > 0
    delete(nh);
    nh = 0;
end
nh = plot(x,y,'r*'); % mark exact position of node
o.Nh(NodeID) = nh;
end

% Plot node id if PlotNodeIDs(NodeID) == true
if PlotNodeID % if PlotNodeID is true
    % find offset to plot node id
    % plot node IDs offset of nodes
    S_id = sprintf('%u',N);
    % add in node specific offsets
    x = x + o.Nodexos(NodeID);
    y = y + o.Nodeyos(NodeID);
    % plot the node id
    text(x, y, S_id);
end

if o.PlotPDOFs % if Pdof = true
    o.PlotPDOF(NodeID, x, y);
end

function PlotPDOF(o, NodeID, x, y)
    % plot the pressure DOF label
    pdof = o.Pdof(NodeID);
    if pdof \neq 0
        xos = o.xos/o.OS; yos = o.yos/o.OS;
        S_id = sprintf('%u',pdof);
        % add in node specific offsets
        x = o.X(NodeID); y = o.Y(NodeID);
        x = x + o.Nodexos(NodeID);
        y = y + yos;
        % plot the node id
        text(x, y, S_id);
    end
end

function PlotConstraints(o, NodeID, x, y)
    % plot constraints for node NodeID at position (x,y)
    % plotnode location
    plotnode = true; % only if not marked with pressure or
    xos = o.xos/o.OS; yos = o.yos/o.OS;
    % X displ Fixed
    if o.Xdf(NodeID) % if x fixed
        % plot symbold to the left of node
        h = plot((x-xos),y,'<>');
        o.XFh = h;
    end
end
% Y displ Fixed
if o.Ydf(NodeID) % if y fixed
    % plot symbol below node
    h = plot(x, (y-yos), 'c-');
    o.YFh = h;
    o.Nodeyos(NodeID) = yos; % put node ID above node
end

% Pressure DOF Exists
if o.Pdof(NodeID)>0
    h = plot(x,y,'ko');
    o.Pdofh = h;
plotnode = false;
    o.Nh(NodeID) = h;
end

% if pressure specified
if o.Pbc(NodeID)>0
    h = plot(x,y,'gd'); % fixed pressure = red diamond
    o.Pbch = h;
plotnode = false;
    o.Nh(NodeID) = h;
end

% if node is not marked with a pressure dof or pressure
if plotnode
    nh = plot(x,y,'r*'); % mark exact position of node
    o.Nh(NodeID) = nh;
end

function PlotForces(o)
    for i = 1:o.NN;
        rx = o.xos*2; ry = o.yos*2;
        xF = o.XF(i); yF = o.YF(i);
        xl = (abs(xF)/o.XFmax)*rx; yl = (abs(yF)/o.YFmax)*ry;
        o.PlotForce(i, xF, yF, xl, yl);
    end
end

function PlotForce(o, NodeID, xF, yF, xl, yl)
    x = o.X(NodeID); y = o.Y(NodeID);
    r = xl;
    if xF > 0 % arrow right
        xl = [(x-r) (x-r/4)];
        yl = [y y];
        plot(xl(2), yl(2), 'k>', 'LineWidth', 2);
        plot(xl, yl, 'k', 'LineWidth', 2);
    elseif xF < 0 % arrow left
        xl = [(x+r) (x+r/4)];
        yl = [y y];
        plot(xl(2), yl(2), 'k<', 'LineWidth', 2);
        plot(xl, yl, 'k', 'LineWidth', 2);
    end
    r = yl;
    if yF > 0 % arrow up
        xl = [x x];
        yl = [(y-r) (y-r/4)];
        plot(xl(2), yl(2), 'k^', 'LineWidth', 2);
plot(xl, yl, 'k-', 'LineWidth', 2);

elseif yF < 0 % arrow down
    xl = [x x];
    yl = [(y+r) (y+r/4)];
    plot(xl(2), yl(2), 'kv', 'LineWidth', 2);
    plot(xl, yl, 'k-', 'LineWidth', 2);
end

plot(x, y, 'k-');

function AddLeg(o)
    h = [o.Nh(1), o.XFh, o.YFh, o.Pdofh o.Pbch];
    legend(h, 'Node', 'Ux=0', 'Uy=0', ...
         'Pressure dof', 'Pressure BC');
end

function GetNodeInfo(o,NodeID)

    % print out all info for node NodeID
    x dof = o.Xdof(NodeID);
    y dof = o.Ydof(NodeID);
    stars = '********';
    fprintf('%s Info for Node %u%s
', stars, NodeID, stars);
    fprintf('%s\nNodeDOF\n', 'n');
    fprintf('%s X DOF %u\n', 't', x dof);
    fprintf('%s Y DOF %u\n', 't', o.Ydof(NodeID));
    fprintf('%s Pressure DOF %u\n', 't', o.Pdof(NodeID));
    fprintf('%s\nDisplacement Values\n', 't');
    fprintf('%s Ux: %2.3e\n', 't', o.Ux(NodeID));
    fprintf('%s Uy: %2.3e\n', 't', o.Uy(NodeID));
    fprintf('%s\nPlotting Info\n', 't');
    fprintf('%s X: %2.4f\n', 't', o.X(NodeID));
    fprintf('%s Y: %2.4f\n', 't', o.Y(NodeID));
    fprintf('%s X offset: %2.4f\n', 't', o.Nodexos(NodeID));
    fprintf('%s Y offset: %2.4f\n', 't', o.Nodeyos(NodeID));
    DupNodeID = o.DupNodeIDs(NodeID);
    if DupNodeID ≠ 0
        fprintf('%s This node is double noded with %u\n', 't', DupNodeID)
        fprintf('%s\n', 's');
        o.GetNodeInfo(DupNodeID)
    elseif DupNodeID ≠ 0
        fprintf('%s This node is not double noded\n');
        fprintf('%s\n', 's');
    end
end

B.3.2 Element Manager
classdef ElementManager < handle
%
Class to manage all ElemDef instances
% created by MeshDefR
% 11.12.2012 — added compatibility with new NodeDefClass

properties

Elems;    % vector of handles to ElementClass instances
ElemIDs;  % vector of element ID’s
V;        % handle to AI.m, filled in when AI is initialized

PDCh;     % ProbDefClass
nel_est = 100;     % estimate of number of elements

ERow;     % vector (length = nel), row each element is in
ECol;     % vector (length = nel), column each element is in

% number of elements created
nel = 0;    % number of elements, initialize to zero
nel_bulk = 0;
nel_cse = 0;

% Arrays for elements
isCSE;     % is element a CSE? true/false
NDCh;      % Node def class handle

% plotting stuff
PlotElemIDsG = true;    % true/false, true = plot all element IDs

end
properties (Dependent)

LM;    % LM Matrix (nel_dof x nel) — contains each disp dof
LMP;   % LM Matrix for fluid pressures
LMAll; % LM and LMP
coords;
g;     % g matrix — contains the pressure values
gp;    % g matrix for fluid pressure
ElemNodeIDs; % matrix — each row = element and vals = node IDs
d_el; % value of each element dof
NRows; % number of rows of elements
NCols; % number of columns of elements

methods

function o = ElementManager(ProbDefClass_handle)
% created by MeshDefR
    o.PDCh = ProbDefClass_handle;
    o.NDCh = o.PDCh.NDCh; % get handle to node def class
    o.nel = 0;

    temp(o.nel_est) = ElementClass;
    o.Elems = temp;
end

function CreateElements(o, ElemM, isCSE, ERow, ECol)
    N = numel(ElemM(:,1));
    for i = 1:N
        NodeIDs = ElemM(i,:);
iscse = isCSE(i);
eRow = ERow(i);
eCol = ECol(i);
o.CreateElement(NodeIDs, iscse, eRow, eCol)
end
end

function CreateElement(o, nodeIDs, iscse, eRow, eCol)
o.nel = o.nel + 1;
if iscse % if element is a CSE
  o.nel_cse = o.nel_cse + 1; % inc. # of CSE's
else % if element is not a CSE
  o.nel_bulk = o.nel_bulk + 1; % inc. # of bulk E's
end
ElementID = o.nel;
o.Elems(o.nel) = ElementClass(o, o.NDCh, ElementID, nodeIDs, iscse, ... 
eRow, eCol);
o.ElemIDs(o.nel) = o.nel;
o.isCSE(o.nel) = iscse;

% save the row and column of the element
o.ERow(o.nel) = eRow;
o.ECol(o.nel) = eCol;
end

function SortElemNodes(o)
  for i = 1:o.nel
    o.Elems(i).SortNodes;
  end
end

function InsertCSEBetweenElems(o, ElemAID, ElemBID, dx, dy)
  % inserts a CSE between bulk elements ElemA and ElemB.
  % dx and dy specify the x and y aperture
  % find the nodes in common between element A and B
  ElemABNodes = o.GetCommonNodes(ElemAID, ElemBID); % nodes in A & B
  ElemABNodes = o.SortNodes(ElemAID, ElemABNodes);
  % insert a cse defined by ElemABNodes
  o.InsertCSE(ElemABNodes, ElemBID, dx, dy)
  % fix elements A and B (from offsetting for aperture)
  o.Elems(ElemAID).FixMidpoints;
o.Elems(ElemBID).FixMidpoints;
end

function InsertCSEsBetweenRows(o, row1, row2, aperture)
  % inserts CSE's between row1 and row2
  % dy specifies the aperture
dx = 0;
dy = aperture;
[Row1IDs_I, ~] = o.GetElemsInRow(row1);
[Row2IDs_I, ~] = o.GetElemsInRow(row2);
  for col = 1:o.NCols % for each column
    ElemA = Row1IDs_I(col);
    ElemB = Row2IDs_I(col);
o.InsertCSEBetweenElems(ElemA, ElemB, dx, dy);
end

function InsertCSEBetweenCols(o, col1, col2, aperture)
% inserts CSE's between row1 and row2
% aperture = offset distance
dx = aperture;
dy = 0;
[Col1IDs, ~] = o.GetElemsInCol(col1);
[Col2IDs, ~] = o.GetElemsInCol(col2);
for col = 1:o.NCols % for each column
    ElemA = Col1IDs(col);
    ElemB = Col2IDs(col);
    o.InsertCSEBetweenElems(ElemA, ElemB, dx, dy);
end
end

function InsertCSE(o, OldNodeIDs, ElemID, dx, dy)
% adds CSE defined by NodeIDs
% replaced the node IDs on element ElemID with new nodes
CSENodeIDs = zeros(6,1);
CSENodeIds(1:3) = OldNodeIDs;
% DupNodeIDs = OldNodeIDs([2, 1, 3]); % put in correct order to duplicate
DupNodeIDs = OldNodeIDs; % node IDs to duplicate
% create new nodes for CSE
NewNodeIDs = o.NDCh.DuplicateNodes(DupNodeIDs, dx, dy);
% Removed 11.11.2012 so nodes don't get offset applied twice
% shift new node IDs by dx, dy
o.NDCh.ShiftNodes(NewNodeIDs, dx, dy);
% Replace reference in ElemID to newly created nodes
o.Elems(ElemID).ReplaceNodeIDs(OldNodeIDs, NewNodeIDs);
% Add CSE new element
% put nodes in correct order
CSENodeIDs(4:6) = NewNodeIDs([2 1 3]);
% Create CSE
o.CreateElement(CSENodeIDs, true, 0, 0);
end

function AddPressureDOFs(o)
% for i = 1:o.nel % for each element (CSE's will be skipped)
    o.Elems(i).AddPressureDOFs
end

function ElemABNodes = GetCommonNodes(o, ElemAID, ElemBID)
% returns the node ID's of the nodes that are shared between elements
% ElemA and ElemB
ElemANodes = o.Elems(ElemAID).NodeIDs; % nodes in ElemA
ElemBNodes = o.Elems(ElemBID).NodeIDs; % nodes in ElemB
m = max([ElemANodes(:); ElemBNodes(:)]); % highest node ID
iA = false(m,1); % logical vector of size = maximum node ID
iB = iA;
ElemABNodes = (1:m)'; % all node ID's through highest ID

iA(ElemANodes) = true; % logical vector of nodes in A
iB(ElemBNodes) = true; % logical vector of nodes in B
ElemABNodes_i = iA & iB; % logical vector of nodes in A and B

% integer vector of nodes in a and b
ElemABNodes = ElemABNodes(ElemABNodes_i);

end

function NodesOut = SortNodes(o, ElemID, nodeIDs)
% sorts the nodes nodeIDs to match the local node numbering of
% ElemID

NodeIDs = o.Elems(ElemID).NodeIDs;
N = numel(NodeIDs);
n = numel(nodeIDs);
NodesOut = zeros(n,1);

k = 1;
for i = 1:N
    id = NodeIDs(i);
    t = nodeIDs(nodeIDs==id);
    if isempty(t) == false
        NodesOut(k) = id;
        k = k + 1;
    end
end
end

function [ElemIDs_I ElemIDs_L] = GetElemsInRow(o, ROW)
% returns the element ID's in row ROW
% returns both an integer index and a logical index
ElemIDs_I = o.ElemIDs(o.ERow == ROW);
ElemIDs_L = false(o.nel,1);
ElemIDs_L(ElemIDs_I) = true;
end

function [ElemIDs_I ElemIDs_L] = GetElemsInCol(o, COL)
% returns the element ID's in column COL
% returns both an integer index and a logical index
ElemIDs_I = o.ElemIDs(o.ECol == COL);
ElemIDs_L = false(o.nel,1);
ElemIDs_L(ElemIDs_I) = true;
end

function FixMidpoints(o)
% loops through all elements fixing the midpoints
% midpoints = local nodes 5-9 (bulk elements only)
for i = 1:o.nel
    o.Elems(i).FixMidpoints;
methods % set and get functions

function out = get.ElemNodeIDs(o)
    out = zeros(o.nel,9);
    for i = 1:o.nel
        a = o.Elems(i).NodeIDs;
        n = numel(a);
        out(i,1:n) = a;
    end
end

function out = get.LMAll(o)
    out = [o.LM; o.LMP];
end

function out = get.LM(o)
    % Assemble LM matrix from each element's LM
    lm = o.Elems(1).lm;
    n = numel(lm);
    out = zeros(n,o.nel);
    for i = 1:o.nel
        lm = o.Elems(i).lm;
        n = numel(lm);
        out(1:n,i) = lm;
    end
end

function out = get.LMP(o)
    lmp = o.Elems(1).lmp;
    n = numel(lmp);
    out = zeros(n, o.nel);
    for i = 1:o.nel
        lmp = o.Elems(i).lmp;
        n = numel(lmp);
        out(1:n, i) = lmp;
    end
end

function out = get.g(o)
    g = o.Elems(1).g;
    n = numel(g);
    out = zeros(n,o.nel);
    for i = 1:o.nel
        g = o.Elems(i).g;
        n = numel(g);
        out(1:n,i) = g;
    end
end

function out = get.gp(o)
    gp = o.Elems(1).gp;
    n = numel(gp);
    out = zeros(n, o.nel);
    for i = 1:o.nel
        gp = o.Elems(i).gp;
        n = numel(gp);
        out(1:n,i) = gp;
function out = get.coords(o)
    out = zeros(o.nel, 18);
    for i = 1:o.nel
        c = o.Elems(i).coords;
        n = numel(c);
        out(i, 1:n) = c;
    end
end

function out = get.NRows(o)
    % number of rows of elements
    out = max(o.ERow(:));
end

function out = get.NCols(o)
    % number of columns of elements
    out = max(o.ECol(:));
end

function out = get.d_el(o)
    n = o.nel;
    out = zeros(n, 22);
    for el = 1:n
        out(el,:) = o.Elems(el).d_el;
    end
end

methods
    % functions to plot elements
    function PlotElems(o)
        for i = 1:o.nel
            o.Elems(i).PlotElem;
        end
    end
    function PlotElemsPress(o)
        for i = 1:o.nel
            o.Elems(i).PlotElemPress;
        end
    end
end

B.3.3 Element Class to Manage Each Individual Element

classdef ElementClass < handle
    % ELEMENTCLASS -- instanced once per element
1
2
3
properties
  ElemID;
  isCSE;
  % Node Info
  NodeIDsAll; % ID's of nodes element is made up by
  % location of element in mesh
eRow; % row element is in
eCol; % column element is in
end

properties(Hidden)
  NodeIDsActive = false(9,1);
  % handles
  NDCh;
PCh;
hf; % handle to element fill plot (see PlotElemPress)
end

properties(Dependent)
e1,NN; % number of nodes in element
  % Node coordinates
  X; % x coordinates of nodes in element
  Y; % y coordinates of nodes in element
  x dof; % x degrees of freedom
  y dof; % y degrees of freedom
  p dof; % pressure degrees of freedom
  NodeIDs; % Active Node ID's
  Fextxy; % external force in xy format
d_e1; % element displacements in format [Ux1, Uy1, Ux2, Uy2....]
aperture; % aperture between nodes 3 and 6 if is a CSE
end

properties(Dependent, Hidden)
  % all of the following are dependent because they grab the actual
  % values from the node def class (NDCh) when called
  lm; % = LM(:,elementID)
lmp; % IM(:,elementID), for pore pressure
  coords; % = coords of element in [x1 y1 x2 y2...] format
  X0; Y0; % initial positions of all coordinates
  g; % g matrix (prescribed disp)
gp; % fluid pressure g matrix (fixed pressure nodes)
xg; % g for x DOF's
yg; % g for y DOF's
  Fext; % Applied external force
  Fx; % applied external force in the x direction
  Fy; % applied external force in the y direction
  P; % pressure at nodes
  P0; % initial pressure at nodes
  % center of element (may not be actual center of mass)
  CMx; % x position of node 9
  CMy; % y position of node 9
  % plotting
  PlotID; % grabbed from parent class
end
methods

function o = ElementClass(PCh, NDCh, ElemID, NodeIDs, isCSE, eRow, eCol)
o.NodeIDsAll = zeros(9,1);
if nargin == 0
    ElemID = 0;
    isCSE = false;
    PCh = [];
    NDCh = [];
    NodeIDs = [11 22 33];
    eRow = 0;
    eCol = 0;
end
o.PCh = PCh; % parent class handle
o.NDCh = NDCh; % node def class handle
o.ElemID = ElemID; % element ID

% NodeIDs element is made up with
i = (NodeIDs>0);
o.NodeIDsAll(1:sum(i)) = NodeIDs(i);
o.isCSE = isCSE;
o.eRow = eRow;
o.eCol = eCol;
end

function ReplaceNodeIDs(o, OldNodeIDs, NewNodeIDs)
N = numel(OldNodeIDs);
for i = 1:N
    o.ReplaceNode(OldNodeIDs(i), NewNodeIDs(i));
end
end

function ReplaceNode(o, OldNodeID, NewNodeID)
% replace reference to OldNodeID with NewNodeID
o.NodeIDs(o.NodeIDs == OldNodeID) = NewNodeID;
end

function SortNodes(o)
% sort the nodes if element is not a CSE
if o.isCSE == false % only sort of not a CSE
    NodesO = zeros(1, o.el_NN);
    x = o.X0; y = o.Y0;
    xmin = min(x); ymin = min(y);
    xmax = max(x); ymax = max(y);
    % ul_____um______ur
    % | | | | |
    % c1____cm______cr
    % | | | | |
    % ll_____lm______lr
    ll = (x==xmin) & (y==ymin); ll_n = 1;
    lr = (x==xmax) & (y==ymin); lr_n = 2;
    ur = (x==xmax) & (y==ymax); ur_n = 3;
    ul = (x==xmin) & (y==ymax); ul_n = 4;
NodesO(ll_n) = o.NodeIDs(ll);
NodesO(lr_n) = o.NodeIDs(lr);
NodesO(ur_n) = o.NodeIDs(ur);
NodesO(ul_n) = o.NodeIDs(ul);

lm = (x>xmin) & (x<xmax) & (y==ymin); lm_n = 5;
cr = (y>ymin) & (y<ymax) & (x==xmax); cr_n = 6;
um = (x>xmin) & (x<xmax) & (y==ymax); um_n = 7;
c1 = (y>ymin) & (y<ymax) & (x==xmin); c1_n = 8;
cm = (x>xmin) & (x<xmax) & (y>ymin) & (y<ymax); cm_n = 9;

NodesO(lm_n) = o.NodeIDs(lm);
NodesO(cr_n) = o.NodeIDs(cr);
NodesO(um_n) = o.NodeIDs(um);
NodesO(c1_n) = o.NodeIDs(c1);
NodesO(cm_n) = o.NodeIDs(cm);

o.NodeIDs = NodesO;

function EnablePressureDOFs(o)
% add pressure DOF’s to first 4 nodes in each bulk element
% if the DOF already exists, NDC will not create another
% if node has a pressure BC, no dof will be created
if o.isCSE == false; % only if not a CSE
  for i = 1:4 % for local nodes 1-->4
    NodeID = o.NodeIDs(i);
    o.NDCh.PressEnable(NodeID) = true;
    o.NDCh.AddNodePDOF(NodeID);
  end
end

function FixMidpoints(o)
% loops through local nodes 5-->8 to make sure sides are square
% (only for bulk elements)
if o.isCSE == false
  % Node 5
  ID = o.NodeIDs(5);
  Node0 = o.NodeIDs(1);
  Node1 = o.NodeIDs(2);
  o.NDCh.FixNodes( ID, Node0, Node1);
  % Node 6
  ID = o.NodeIDs(6);
  Node0 = o.NodeIDs(2);
  Node1 = o.NodeIDs(3);
  o.NDCh.FixNodes( ID, Node0, Node1);
  % Node 7
  ID = o.NodeIDs(7);
Node0 = o.NodeIDs(3);
Node1 = o.NodeIDs(4);
NDCh.FixNodes( ID, Node0, Node1);

% Node 8
ID = o.NodeIDs(8);
Node0 = o.NodeIDs(4);
Node1 = o.NodeIDs(1);
NDCh.FixNodes( ID, Node0, Node1);

% Node 9
% vertical 'line'
ID = o.NodeIDs(9);
Node0 = o.NodeIDs(5);
Node1 = o.NodeIDs(7);
NDCh.FixNodes( ID, Node0, Node1);
% horizontal 'line'
ID = o.NodeIDs(9);
Node0 = o.NodeIDs(6);
Node1 = o.NodeIDs(8);
NDCh.FixNodes( ID, Node0, Node1);
end
end

methods % set and get functions

function out = get.X(o)
% get x values
out = NDCh.X(o.NodeIDs);
out = out(:);
if ~isempty(o.NodeIDs)
out = NDCh.X(o.NodeIDs);
end

function out = get.Y(o)
out = NDCh.Y(o.NodeIDs);
out = out(:);
end

function out = get.X0(o)
% get x values
out = NDCh.X0(o.NodeIDs);
out = out(:);
if ~isempty(o.NodeIDs)
out = NDCh.X(o.NodeIDs);
end

function out = get.Y0(o)
out = NDCh.Y0(o.NodeIDs);
out = out(:);
end

function out = get.CMx(o)
if o.isCSE == false
out = o.X(9);
end
function out = get.CMy(o)
    if o.isCSE == false
        out = o.Y(9);
    end
end

function out = get.elNN(o)
    out = numel(o.NodeIDs);
end

function out = get.lm(o)
    temp = [o.xdof o.ydof];
    out = temp';
    out = out(:);
end

function out = get.xdof(o)
    out = o.NDCh.Xdof(o.NodeIDs);
    out = out(:);
end

function out = get.ydof(o)
    out = o.NDCh.Ydof(o.NodeIDs);
    out = out(:);
end

function out = get.pdof(o)
    if o.isCSE
        localnodes = [1 2 4 5];
    else
        localnodes = [1 2 3 4];
    end
    nodes = o.NodeIDs(localnodes);
    out = o.NDCh.Pdof(nodes);
end

function out = get.g(o)
    temp = [o.xg o.yg];
    out = temp';
    out = out(:);
end

function out = get.xg(o)
    out = o.NDCh.Xdp(o.NodeIDs);
    out = out(:);
end

function out = get.yg(o)
    out = o.NDCh.Ydp(o.NodeIDs);
    out = out(:);
end

function out = get.gp(o) % fluid pressure g matrix
    if o.isCSE == false % if not a cse
        IDs = o.NodeIDs(1:4);
    else % if it is a cse
        IDs = o.NodeIDs([1 2 4 5]);
    end
    out = o.NDCh.Pbc(IDs);
    out = out(:);
end
function out = get.lmp(o) % fluid pressure lm matrix
    if o.isCSE == false % if not a cse
        IDs = o.NodeIDs(1:4);
        out = o.NDCh.Pdof(IDs);
    else % if it is a cse
        IDs = o.NodeIDs([1 2 4 5]);
        out = o.NDCh.Pdof(IDs);
    end
    out = out(:);
end

function out = get.coords(o)
    C = [o.X o.Y];
    C = C';
    out = C(:); % format to [x1 y1 x2 y2...]
end

function out = get.NodeIDs(o)
    out = o.NodeIDsAll(o.NodeIDsAll > 0);
end

function set.NodeIDs(o, val)
    o.NodeIDsAll = val;
end

function out = get.Fext(o)
    % gets the applied force on the nodes of the element
    out = [o.Fx o.Fy];
end

function out = get.Fextxy(o)
    % gets the applied force on the nodes in the xy format
    fxy = o.Fext';
    out = fxy(:);
end

function out = get.Fx(o)
    out = o.NDCh.XF(o.NodeIDs);
end

function out = get.Fy(o)
    out = o.NDCh.YF(o.NodeIDs);
end

function out = get.P(o)
    if o.isCSE
        localnodes = [1 2 4 5];
    else
        localnodes = [1 2 3 4];
    end
    nodes = o.NodeIDs(localnodes);
    out = o.NDCh.NodeP(nodes);
end

function out = get.P0(o)
    % get the initial pressure at all the nodes
    if o.isCSE
        localnodes = [1 2 4 5];
    else
        localnodes = [1 2 3 4];
    end
    nodes = o.NodeIDs(localnodes);
out = o.NDCh.NodeP0(nodes);
end

function out = get.d_el(o)
% get element displacement/pressure vector
out = zeros(1,22);
% DISPLACEMENT
Ux = o.X - o.X0; % x displacements
Uy = o.Y - o.Y0; % y displacements
temp = [Ux(:) Uy(:)]; % combine into one matrix of displacements

% convert into format [x1 y1 x2 y2...]
temp = temp'; % now temp(1,:) = [Ux1 Ux2 ...] temp(2,:) = [Uy1 Uy2
if o.isCSE
out(1:12) = temp(:,);
else
out(1:18) = temp(:,);
end
% Pressure
p = o.P;
out(19:22) = p(:,); % pressure in last 4 of vector
end

function out = get.aperture(o)
% return the aperture size between CSE center nodes
if o.isCSE % if element is a CSE
X = o.X; Y = o.Y;
ax = X(6) - X(3);
ay = Y(6) - Y(3);
out = sqrt(ax^2 + ay^2);
else
out = 0; % for bulk elements
end
end

methods % plot functions
function PlotElem(o)
% plots the element with a line through the outer nodes
if o.isCSE == false % if element is a bulk element
x = o.X; x = x([1 5 2 6 3 7 4 8 1]);
y = o.Y; y = y([1 5 2 6 3 7 4 8 1]);
LW = 1; % bulk element line width
else % if element is a CSE
x = o.X([1 3 2 4 6 5 1]);
y = o.Y([1 3 2 4 6 5 1]);
LW = 2; % bulk element line width
end
plot(x, y, 'LineWidth', LW)
if (o.PlotID == true) && (o.isCSE == false)
% if PlotID is true and element is NOT a CSE
x = o.X(9); y = o.Y(9)+0.020;
S_id = sprintf('Elem%d',o.ElemID);
text(x, y, S_id);
end

function PlotElemPress(o)
% plots the element with a filled in polygon colored to indicate
% the pressure
if o.isCSE % if element is a CSE
  x = o.X([1 2 4 5]);
y = o.Y([1 2 4 5]);
else % if element is a bulk
  x = o.X([1 2 3 4]);
y = o.Y([1 2 3 4]);
end
if isempty(o.hf) == false;
  delete(o.hf)
end

end % function

function PlotElemNodeOrder(o)
% diagnostic function to show the local node ordering
x = o.X;
y = o.Y;
x = x(:);
y = y(:);
N = numel(x);
figure;
hold on
r = 0.05*(max(x) - min(x));
xmin = min(x) - r; ymin = min(y) - r;
xmax = max(x) + r; ymax = max(y) + r;
axis([xmin xmax ymin ymax]);
r = r/2;
for i = 1:N
  plot(x(i), y(i),'r*');
  lb = sprintf('Node %u',o.NodeIDs(i));
  text(x(i) + r, y(i) + r, lb);
  beep;
  pause(1);
end

function out = get.PlotID(o)
out = o.PCh.PlotElemIDsG;
end
end
B.3.4 All Info Class to Allow Access and Storage of Important Parameters

```matlab
classdef AI < handle

% AI = All Info — a combination of PreCalc and VarClass2
% --Built to ensure only one version of each variable exists, and
% --Contains
% +handles to all classes
% +Contains all parameters
% +High level functions to turn off variables
% +get functions for important values

properties (Constant) % constant props can be accessed with "x = AI.x" syntax
n_cse_ip = 3;
n_cse_isv = 17;
% conversion factors
%time_factor = 1 %s
%time_factor = 60 %min
%time_factor = 60*60 %hr
time_factor = 24*60*60 %day
%time_factor = 365*24*60*60 %year
%time_factor = 10*365*24*60*60 %decade
%time_factor = 100*365*24*60*60 %century
stress_factor = 1 %Pa, N
stress_factor = 1e3 %kPa, kN
stress_factor = 1e6 %MPa, MN

Tract = 10e3/AI.stress_factor; % confining pressure
% Tract = 0.0; % confining pressure

% elastic constants of bulk elements
E_p = 1.0e6/AI.stress_factor; %Pa
nu_p = 0.3; %

porepress_2m=9810*2/AI.stress_factor; % (N/m^3)*m=Pa, (kN/m^3)*m=kPa, ...
porepress_1m=9810*1/AI.stress_factor; % (N/m^3)*m=Pa, (kN/m^3)*m=kPa, ...
porepress_1pt5m=9810*1.5/AI.stress_factor; % (N/m^3)*m=Pa, (kN/m^3)*m=kPa, ...
porepress_1pt5m=9810*0.5/AI.stress_factor; % (N/m^3)*m=Pa, (kN/m^3)*m=kPa, ...

% BUILT IN STATIC FUNCTIONS NEAR BOTTOM-----------------------------------------%
% perm_cse_paras
% D (plane strain)
% cse_para
% BUILT IN STATIC FUNCTIONS NEAR BOTTOM-----------------------------------------%

%initial aperture
aperture = 5.0e-2; %m
% aperture = 0;

% elastic stiffness matrix of cse
kt = 1e10/AI.stress_factor; %Pa/m or kPa/m
kn = 1e10/AI.stress_factor; %Pa/m or kPa/m
% kt = 1e6/stress_factor; %Pa/m or kPa/m
% kn = 2e7/stress_factor; %Pa/m or kPa/m
Kcse = [AI.kt 0; 0 AI.kn]; %for now, assume same for all cses
```

% Poromechanical variables
% initial constituent mass densities
rhoS = 2700; %kg/m^3 Solid
rhoF = 1000; %kg/m^3 Fluid
% initial constituent volume fractions
nS0 = 0.58; % Solid
nF0 = 0.42; % Fluid
% permeability coefficient
init_dens = AI.nF0*AI.rhoF + AI.nS0*AI.rhoS; %kg/m^3
perm0 = (le-9)*AI.time_factor*AI.stress_factor; %m^2/(Pa.s) or ...
(m^2)/(kPa.d), already accounts for fluid viscosity
perm0sat = AI.perm0*9810/AI.stress_factor; %(m^2)/(Pa.s)*N/m^3=m/s ...
or ((m^2)/(kPa.d))*(kN/m^3)=m/d
perm = AI.perm0 * (1 - AI.nF0^2) / (AI.nF0^3); %m^2/(Pa.s) or ...
(m^2)/(kPa.d)
kp = AI.perm * [1 0; 0 1]; %(m^2)/(Pa.s) or (m^2)/(kPa.d)
grav=9.81/AI.stress_factor; %m/(s^2), to convert to kN/m^3 when ...
multiply by mass density
% crack hydraulic parameters
fluid_visc = (le-3)/(AI.time_factor*AI.stress_factor); %Pa.s or kPa.d
perm_crack = le-11; %m^2
kcrack = AI.perm_crack/AI.fluid_visc; %m^2/(Pa.s) or (m^2)/(kPa.d)
small_num = le-6; %m, limiting aperture for transverse conductivity calc
small_num0 = 0.0; %m, limiting aperture for longitudinal conductivity calc
% perm_cse_params is built in static funciton below
% perm_cse_params(1) = kcrack;
% perm_cse_params(2) = perm_crack;
% perm_cse_params(3) = fluid_visc;
% perm_cse_params(4) = small_num;
% perm_cse_params(5) = rhoF;
% perm_cse_params(6) = grav;
% perm_cse_params(7) = small_num0;
end
properties (Dependent)
% Meshing variables
nel;
nel_cse;
nel_bulk;
LM;
gd; % displ. g matrix
gp; % pressure g matrix
g; % both g matrices
coords;
Fext;
nnodes; % number of nodes
% degrees of freedom (DOF)
n dof; % total number of DOF's
nporoDOF; % # of pressure DOF's
ndispDOF; % number of displ. DOF's
% nel_d bulk dof = 18; % displ dofs per bulk element
% nel_pf bulk dof = 4; % num el bulk poro DOF's (pore pressure)
% nel bulk dof = 22; % num bulk el DOF's
end
% For Post Processing-------------------------------
dof_displ_plot; % displacement dof to plot
dof_press_plot; % pressure dof to plot
% For Post Processing
Sflux;

neld_cse_dof = 12;
nelpf_cse_dof = 4; % num el cse poro DOF's (pore pressure)
nel_cse_dof = 16; % num el cse DOF's

% Important handles

end

properties
% important handles
PDef;
MDCh; % mesh definition class handle
NDCh; % node definition class handle
MeshDef; % handle to mesh definition implemented
EMCh; % element manager class handle
FolderName; % folder name for output file
SavePath; % path of where to save files

end

properties(Hidden)
    SetOutput_man = false; % set directory output manually
end

methods
    function o = AI(PDef)
        % PDef is an instance of ProbDefManager
        if nargin -= 0
            o.PDef = PDef;
            o.LoadHandles;
            o.GetOutputFolder; % make a folder for all output files
        end
    end

    function LoadHandles(o)
        % load all handles
        o.MDCh = o.PDef.MDCh;
        o.NDCh = o.PDef.NDCh;
        o.MeshDef = o.PDef.MDCh.MeshDef;
        o.EMCh = o.PDef.MDCh.EMCh;
        o.EMCh.V = o; % give handle to element manager class
    end

    function GetOutputFolder(o)
        % gets the name of the output folder to be created if any files are
        % written
        c = clock;
y = c(1); m = c(2); d = c(3); h = c(4); mi = c(5);
o.FolderName = sprintf('RESULTS_Mesh%d_%u.%u.%u.%u.%u',... 'MeshType, m, d, y, h, mi);
        mkdir(o.FolderName);
    end
o.SavePath = sprintf('%s', o.FolderName);
end

function SetOutputFolder(o)
    path = uigetdir;
    o.SavePath = sprintf('%s', path);
end

function WriteFile(o, FileName, Var)
    if exist(o.FolderName, 'dir') ~= 7
        mkdir(o.FolderName);
    end
    w = sprintf('%s.txt', o.SavePath, FileName);
    save(w, ['Var', '-ASCII']);
end

function pathname = GetFilePath(o, FileName)
    % get the full path and file name to the file
    if o.SetOutput
        pathname = sprintf('%s.txt', o.SavePath, FileName);
    else
        pathname = sprintf('%s/%s.txt', pwd, o.SavePath, FileName);
    end
end

function LabelsOff(o)
    % turn off all plotting labels
    o.NDCh.PlotNodeConsG = false;   % dont plot constraints
    o.NDCh.PlotNodeIDsG = false;    % dont plot Node IDs
    o.NDCh.PlotNodePDOFs = false;   % dont plot pressure DOF's
    o.EMCh.PlotElemIDsG = false;   % dont plot element IDs
end

methods % set and get functions for variables NOT in AI
    function out = get.nel(o)
        out = o.nel_bulk + o.nel_cse;
    end

    function out = get.nel_cse(o)
        out = o.PDef.MDCh.nel_cse;
    end

    function out = get.nel_bulk(o)
        out = o.PDef.MDCh.nel_bulk;
    end

    function out = get.coords(o)
        out = o.MDCh.coords;
    end

    function out = get.Fext(o)
        out = o.NDCh.Fext;
    end

    function out = get.gd(o)
        out = o.MDCh.g;
    end

    function out = get.gp(o)
        out = o.MDCh.gp;
function out = get.g(o)
    out = [o.gd; o.gp];
end

function out = get.LM(o)
    out = o.MDCh.LMAAll;
end

function out = get.nporoDOF(o)
    out = o.NDCh.NPdofs;
end

function out = get.ndispDOF(o)
    out = o.NDCh.Nddofs;
end

function out = get.ndof(o)
    out = o.nporoDOF + o.ndispDOF;
end

function out = get.nnodes(o)
    out = o.NDCh.NN;
end

function out = get.d(o)
    out = o.NDCh.d;
end

function out = get.dof_displ_plot(o)
    out = o.NDCh.Ydof(o.node_disp_plot);
end

function out = get.dof_press_plot(o)
    out = o.NDCh.Pdof(o.node_press_plot);
end

function out = get.node_disp_plot(o)
    out = o.MeshDef.node_disp_plot;
end

function out = get.node_press_plot(o)
    out = o.MeshDef.node_press_plot;
end

function out = get.Sflux(o)
    out = o.MeshDef.f_Sflux(o);
end

methods (Static)

function out = perm_cse_params
    % function to build perm_cse_params
    out(1) = AI.kcrack;
    out(2) = AI.perm_crack;
    out(3) = AI.fluid_visc;
    out(4) = AI.small_num;
    out(5) = AI.rhoF;
    out(6) = AI.grav;
    out(7) = AI.small_num0;
end

function out = D
% Load constants
E = AI.E_p;
nu = AI.nu_p;
mu = E/(2*(1+nu));
lambda = 2*mu*nu/(1-2*nu);
out = ...
[ lambda+2*mu lambda 0 ;
  lambda lambda+2*mu 0 ;
  0 0 mu];

function out = cse_para
out(1) = 1e4/AI.stress_factor;  % fGf_I, N/m or kN/m
out(2) = 1e4/AI.stress_factor;  % fGf_II, N/m or kN/m
out(3) = 2e2;  % falpha_chi
out(4) = 2e2;  % falpha_c
out(5) = 900;  % falpha_phi
out(6) = 900;  % falpha_psi
%out(5) = 0.;  % falpha_phi
%out(6) = 0.;  % falpha_psi
% out(7) = 3000/AI.stress_factor;   % fchi_p, Pa or kPa
out(8) = 9053/AI.stress_factor;  % fc_p, Pa or kPa
out(9) = 0.5236;  % fphi_p, rad
out(7) = out(8)/tan(out(9));  % fchi_p, Pa or kPa
out(7) = 1e20;  % fchi_p, Pa, to avoid yielding
out(8) = 1e20;  % fc_p, Pa, to avoid yielding
out(10) = 0.087;  % fpsi_p, rad
%out(10) = 0.0;  % fpsi_p, rad
out(11) = 0./AI.stress_factor;  % fchi_r, Pa or kPa
out(12) = 0./AI.stress_factor;  % fc_r, Pa or kPa
out(13) = 0.;  % fphi_r, rad
out(14) = 1e-3;  % fTol_1
out(15) = 1e-8;  % fTol_2
end
end
end
B.4 Problem Definition Format and Problem Definitions

B.4.1 Problem Definition Manager

```matlab
classdef ProbDefManager < handle
    %PROBDEFMANAGER — Load problem and manage variables

    properties
        % handles to other classes
        MDCh; % Mesh def class handle
        NDCh; % Node Def Class
        EMCh;
        % mesh type
        MeshType; % mesh type
        % handles for plotting
        figh; % figure handle
    end

    methods
        function o = ProbDefManager(MeshType)
            % MeshType specifies what type of mesh to use
            % (see MeshDefR>LoadMesh)

            if nargin == 0 % for debugging
                MeshType = 13;
            end
            o.MeshType = MeshType;
            o.MDCh = MeshDefR(o.MeshType, o);
            o.EMCh = o.MDCh.EMCh;
        end

        methods % plotting functions
            function PlotAll(o)
                %
                o.NDCh.PlotNodeIDs(:) = true;
                o.InitPlot;
                o.PlotNodes;
                o.PlotElems;
                o.AddTitle
            end

            function PlotNodes(o)
                o.NDCh.PlotNodes;
            end

            function PlotElems(o)
                o.MDCh.EMCh.PlotElems
            end

            function AddTitle(o)
                t_el = sprintf('%u bulk elements, %u CSE''s', ... 
                                o.MDCh.nel_bulk,o.MDCh.nel_cse);
                t_node = sprintf('%u nodes, %u displ. DOF''s %u pressure DOF''s', ... 
                                o.MDCh.nel_node,o.MDCh.nel_displDOF,o.MDCh.nel_pressureDOF);
            end
        end
    end
```

function InitPlot(o)
% initialize plot
% setup figure
    o.figh = figure; % make figure and get handle
    hold on; % allow further plots on same figure
% set figure to fullscreen
    set(o.figh,'units','normalized','outerposition',[0.6 0.2 0.4 0.9]);
    set(o.figh,'units','normalized','outerposition',[0 0 1 1]);
% set axis of figure
    axis([min(o.NDCh.X)-o.NDCh.xos] [max(o.NDCh.X)+o.NDCh.xos] ...  
         [min(o.NDCh.Y)-o.NDCh.yos] [max(o.NDCh.Y)+...  
             o.NDCh.yos*4]);
end

function PlotPressure(o)
% plots pressure onto axis

    % initialize plot
    o.InitPlot;

    % plot element pressures
    o.MDCh.EMCh.PlotElemsPress;
    colorbar; % add in a colorbar

    % plot nodes ontop
    o.PlotNodes;

    % plot element outlines on top
    o.PlotElems;
end

function PlotCurrent(o)
% plot the displaced nodes WITH pressures
% assumes d has already been set with NDCh.d = d;
% Do NOT plot node ID's __________________________
    o.NDCh.PlotNodeID = false;
%_____________________________________________
    o.InitPlot; % initialize the plot
    o.PlotNodes; % plot the nodes
    o.PlotElems; % plot the elements
    o.AddTitle; % add a title to the plot
end

function PlotLive(o)
% plots the live view of everything
    if isempty(o.figh) % if figure has not yet been setup
        o.InitPlot;
    end
    o.EMCh.PlotElemsPress;
B.4.2 Problem Initialization and Operations

```matlab
function PlotHistory(o)
    % plot a history of the solid
    end
end
```

```matlab
classdef MeshDefR < handle
    % New Mesh creation class
    % manages
    %    -Mesh definition (e.g. Mesh4)
    %    -NodeDefClass
    %    -ElementManagerClass

    properties
        MeshType;
        MeshDef;
        PDCh; % prob def class handle
        % loaded from MeshDef (MeshDef = ProbDef13 or ProbDef14, ...)
        ElemM;
        ERow; % vector (length = nel), = which row the element is in
        ECol; % vector (length = nel), = which col the element is in
        EMCh; % element manager class h
        Vh; % handle to variable class
    end

    properties (Dependent)
        NDCh; % Node Definition Class handle
        LM; % LM matrix of displ. DOF's
        LMAAll; % LM matrix including pressure DOF's
        g;
        gp;
        coords;
        nel;
        nel_cse;
        nel_bulk;
    end

    methods
```
function o = MeshDefR(Mesh, ProbDefh)
% Mesh = integer corresponding to the mesh type number (LoadMesh)
% ProbDef = parent class handle
% setupmode allows to partially run to design a new mesh
% CREATED BY ProbDefManager
o.MeshType = Mesh;
o.PDCh = ProbDefh;
o.NDCh = NodeDefClass;
o.EMCh = ElementManager(o.PDCh);
o.LoadMesh;
if o.MeshDef.setupmode == false % do not run if in a setup mode
    o.ApplyExternalForces;
o.MeshDef.ApplyNodePressures(o.NDCh);
o.NDCh.NodeP = o.NDCh.NodeP0;
end
% o.SetupVarClass;
end

function LoadMesh(o)
switch o.MeshType
    case 1
        o.MeshDef = Mesh1;
    case 4
        o.MeshDef = Mesh4;
    case 5
        o.MeshDef = Mesh5;
    case 6
        o.MeshDef = Mesh6;
    case 7
        o.MeshDef = Mesh7;
    case 8
        o.MeshDef = ProbDef8;
    case 11
        o.MeshDef = ProbDef11;
    case 10
        o.MeshDef = ProbDef10;
    case 12
        o.MeshDef = ProbDef12;
    case 13
        o.MeshDef = ProbDef13;
    case 14
        o.MeshDef = ProbDef14;
    case 140
        o.MeshDef = ProbDef140;
    case 15
        o.MeshDef = ProbDef15;
    case 16
        o.MeshDef = ProbDef16;
    case 17
        o.MeshDef = ProbDef17;
    case 18
        o.MeshDef = ProbDef18;
end
o.CreateNodes % load nodes and element def matrix
% Get info from MeshDef class
o.ElemM = o.MeshDef.NodesM; % get element matrix from mesh file
o.ERow = o.MeshDef.ERow;
o.ECol = o.MeshDef.ECol;
end

function CreateNodes(o)
% load the nodes defined in o.MeshDef
% o.MeshDef = some problem definition class (e.g. ProbDef10.m)
[X, Y, Xp, Yp,Xd0, Yd0, Pbc, NodeP] = o.MeshDef.GetNodes;
% load in pressure and pressure bc vectors
N = numel(Pbc);
o.NDCh.Pbc(1:N) = Pbc;
o.NDCh.NodeP(1:N) = o.NDCh.NodeP{1:N} + NodeP;
end

function CreateBulkElements(o)
N = numel(o.ElemM(:,1));
o.EMCh.CreateElements(o.ElemM, false(N,1), o.ERow, o.ECol);
end

function CreateCSElements(o)
% create CSE's by calling the function in MeshDef
if o.MeshDef.setupmode == false
  o.MeshDef.MakeCSEs(o.EMCh)
end
end
function AddCSE(o, ElemID, OldNodeIDs)
% replaces nodes OldNodeIDs in element ElemID with newly created
% nodes for the CSE
% dX and dY correspond to the aperture
dx = AI.aperture;
dy = 0;
o.EMCh.InsertCSE(OldNodeIDs, ElemID, dx, dy);
fprintf('CSE Has been created');
end

function GetElemsM(o)
for i = 1:o.nel_bulk
  % bulk element1
  kdd(:,i)=kdd_qdr_quad(o.coords(i,:),D);
ktheta(:,i)=ktheta_qdr_quad(o.coords(i,:));
kthth=kthth_quad(o.coords(i,:), kp);
Kk(:,i) = [ kdd(:,i) -ktheta(:,i) zeros(4,1) kthth ];
Cc(:,i) = [ zeros(18,1) zeros(18,4) ktheta(:,i)' zeros(4,4) ];
fdt(:,i)=zeros(18,1);
end
function AddPressureDOFs(o)
    % must be called last so that all displ. DOF IDs < pressure DOF IDs
    o.NDCh.AddNodePDOFs;
end

function delete(o)
    % function called when instance is cleared
    % clear all prop handles
    delete(o.EMCh);
    delete(o.PDCh);
end

methods % functions to calculate external forces
    function ApplyExternalForces(o)
        [ElementIDs, FuncHandle, tract] = o.MeshDef.ApplyExternalForces(o.EMCh);
        if isempty(ElementIDs) == false % if ElementIDs is NOT empty
            for j = 1:numel(ElementIDs)
                el = ElementIDs(j);
                coords = o.EMCh.Elems(el).coords;
                fxy = FuncHandle(coords, tract);
                NodeIDs = o.EMCh.Elems(el).NodeIDs;
                o.NDCh.AddNodeForcesXY(NodeIDs, fxy);
            end
        end
    end

methods % set and get functions
    function set.NDCh(o,val)
        o.PDCh.NDCh = val;
    end
    function out = get.NDCh(o)
        out = o.PDCh.NDCh;
    end
    function out = get.LM(o)
        out = o.EMCh.LM;
    end
    function out = get.LMAll(o)
        out = o.EMCh.LMAll;
    end
    function out = get.coords(o)
        out = o.EMCh.coords;
    end
    function out = get.nel(o)
        out = o.EMCh.nel;
B.4.3 Problem Definition Standard Format and Operations

classdef ProbDefXX < handle
    % all ProbDefXX classes will inherit from this class

    properties
        MeshName = 'NAME NOT SPECIFIED';
        NN;          % number of nodes
        % element props
        nel_bulk;    % number of bulk elements
        nel_cse;    % number of CS elements
        % number of elements in each direction
        NEBx;       % number of bulk elements in the x direction
        NEBy;       % number of bulk elements in the y direction
        % dimensions of the domain
        x0; x1;     % x boundaries of the domain
        y0; y1;     % y boundaries of the domain
        % element info
        % Node Matrix (# of elements) x (# nodes/element)
        NodesM;     % row i = nodes in element i
        isCSE;
        ERow;       % vector (length = nel), = which row the element is in
        ECol;       % vector (length = nel), = which col the element is in
        % node properties
        % node location
        XCoords;    YCoords;
        % does node have an applied force?
        Xf; Yf;
        % does node have a prescribed displacement?
        Xp; Yp;
        Pf; % pressure fixed (i.e. no pressure DOF)
        Pbc; % logical vector of nodes with pressure bc (true if pressure BC)
        NodeP; % value of node pressures
        NodeActive; % specifies if node is active
        % Time step info
t0;  % initial time step
 tf;  % final time step;
 dt;  % step size

end

properties % specific to setup
  tract;
  setupmode = false;
  node_disp_plot;  % node(s) to plot vertical displ. at
  node_press_plot;  % node(s) to plot fluid pressure at
end

properties (Dependent)
  nel;  % number of elements
end

methods

function LoadMesh(o)
  % CREATE THE MESH FOR NOW, IGNORE CSEs
  % --Specify the location of each node
  % --Specify the boundary conditions
  % Number of nodes in x and y directions
  NNX = o.NEBx*2 + 1;
  NNY = o.NEBy*2 + 1;
  % define coordinates of nodes
  x = linspace(o.x0, o.x1, NNX);
  y = linspace(o.y0, o.y1, NNY);
  [X, Y] = meshgrid(x, y);
  n = o.NN;
  o.XCoords = X(:);
  o.YCoords = Y(:);
  % Load BC's
  o.LoadBCs;
  % set the nodes created to be active
  o.NodeActive(1:numel(o.XCoords)) = true;
  % load matrix containing which elements hold which nodes
  o.LoadElemMatrix
end

function LoadElemMatrix(o)
  % LoadMesh has already created the node positions
  % LoadElemMatrix associates those node positions with a specific
  % element
  % load x and y node positions
  X = o.XCoords;
  Y = o.YCoords;
  % treat position as node ID (i.e. X(2) = node 2)
  NodeIDs = (1:numel(X))';
  M = o.NodesM;  % element matrix
elem = 0;
% divide up region to see what element each node belongs to
x = linspace(o.x0, o.x1, o.NEBx + 1);
y = linspace(o.y0, o.y1, o.NEBy + 1);
for j = 1:(numel(y)-1) \% y variable
    for i = 1:(numel(x)-1) \% x variable
        elem = elem + 1;
        o.ECol(elem) = i;
        o.ERow(elem) = j;
        % find region element exists within
        xmin = x(i); xmax = x(i + 1);
        ymin = y(j); ymax = y(j + 1);
        tempi = ((X >= xmin) & (X <= xmax)) & ((Y >= ymin) & (Y <= ymax));
        temp = NodeIDs(tempi);
        M(elem,:) = temp;
    end
end
o.NodesM = M(1:elem,:);
o.ECol = o.ECol(1:elem);
o.ERow = o.ERow(1:elem);
o.nel_bulk = elem;

function LoadDefaults(o)
% preallocate all values
o.Xf = false(o.NN,1); o.Yf = false(o.NN,1);
o.Xp = o.Xf; o.Yp = o.Xf;
o.Pbc = o.Xf;
o.NodeActive = false(o.NN,1);
o.NodeP = zeros(o.NN,1);
o.NodesM = zeros(o.NN,9);
o.isCSE = false(o.NN,1);
o.ECol = zeros(o.NN,1);
o.ERow = zeros(o.NN,1);
o.nel_bulk = 0;
o.nel_cse = 0;
fprintf('
Mesh Loaded: %s\n', o.MeshName);
end

function ApplyNodePressures(o, NDCh)
% applies hydrostatic pressure to nodes
NodeIDs = 1:NDCh.NN;
for NodeID = NodeIDs
    y = NDCh.Y(NodeID); \% height of node
    pdof = NDCh.Pdof(NodeID);
    if pdof > 0 \% if pressure degree of freedom exists
        p = o.ApplyNodePressure(y);
        NDCh.NodeP0(NodeID) = NDCh.NodeP0(NodeID) + p;
    end
end
for NodeID = NodeIDs
    y = NDCh.Y(NodeID); \% height of node
function p = ApplyNodePressure(o, y)
    \% constants
    rho = AI.rhoF; \% density
    g = AI.grav; \% gravitational constant
    \% hydrostatic pressure
    h = 1 - y;
    p = rho*g*h/AI.stress_factor;
end

function [X Y Xp Yp Xf Yf Pbc NodeP] = GetNodes(o)
    \% returns the info for all nodes in a vector
    \% called by MeshDefR>LoadMesh>CreateNodes
    i = o.NodeActive; \% nodes that are active (i.e. defined)
    X = o.XCoords(i); \% Y = o.YCoords(i);
    Xp = o.Xp(i); \% Yp = o.Yp(i);
    Xf = o.Xf(i); \% Yf = o.Yf(i);
    \% Pf = o.Pf(i);
    Pbc = o.Pbc(i);
    NodeP = o.NodeP(i);
end

function out = get.nel(o)
    out = o.nel_bulk + o.nel_cse;
end
end

B.4.4 Problem Definition 14

classdef ProbDef14 < ProbDefXX
    \% 64 bulk elements (8 x 8) with 8 CSE's located in the center, \% oriented vertically
    \% created on 11/03/2012
    \% UPDATED: 11.07.2012
    \% UPDATED: 12.02.2012 – Fixed fracture aperture
methods
function o = ProbDef14
    o.LoadMeshProps;
o.LoadDefaults; % preallocate all values
o.LoadMesh; % load the mesh which then calls to load ElemM
end

% ********** MOST CHANGES FROM MESH TO MESH GO HERE ********

function LoadMeshProps(o)
    % these are the main mesh properties that need to be changed to
    % define a new problem
    o.MeshName = 'Problem Definition 14';
    % traction on top elements
    o.tract = AI.Tract;

    % define the region
    o.x0 = 0; o.x1 = 1; % physical x domain
    o.y0 = 0; o.y1 = 1; % physical y domain

    % Number of *bulk* elements in x and y directions
    o.NEBx = 8; o.NEBy = 8;

    % estimate the number of nodes in the problem
    o.NN = o.NEBx*o.NEBy*10; % # of nodes (estimate) for preallocation

    % Specify which nodes should be used to plot vertical displ and
    % pressure for post processing
    o.node_disp_plot = 1; % node to plot vertical displ. at
    o.node_press_plot = 1; % node to plot fluid pressure at
end

function LoadBCs(o)
    n = o.NN;
    xf = false(n, 1); yf = false(n, 1);

    % specify which nodes have a fixed displ.
    Yi = (o.YCoords==o.y0);
    % Nodes at y = 0 are fixed in x and y directions
    xf(Yi) = true; yf(Yi) = true;

    Xi = (o.XCoords==o.x0);
    % Nodes at x = 0 are fixed in x direction
    xf(Xi) = true; yf(Xi) = true;

    Xi = (o.XCoords==0.5);
    % Nodes at fracture (x=0.5) are fixed in x direction

    Xi = (o.XCoords==0.50);
    % Nodes at x = 1 are fixed in x direction
    xf(Xi) = true; yf(Xi) = true;

    % Add to o.Xf and o.Yf
    o.Xf(1:n) = xf; o.Yf(1:n) = yf;

    % specify which nodes have a fixed displ.

    % specify pressure info
    Yi = (o.YCoords==o.y1);
    o.Pbc(Yi) = true;

    Xi = (o.XCoords==0.5);
    % center nodes
    % Top nodes
    o.Pbc(Yi) = true;
end
% set the nodes created to be active
o.NodeActive(1:numel(o.XCoords)) = true;
end

function MakeCSEs(o, EMCh)
    % create the CSE's for the mesh given
    % EMCh = handle to element manager class
    % create CSE's by calling the function in Element manager class
    if o.setupmode == false
        EMCh.InsertCSEsBetweenCols(4, 5, AI.aperture);
    end
end

function [ElementIDs FuncHandle tract] = ApplyExternalForces(o, EMCh)
    [ElementIDs, ~] = EMCh.GetElemsInRow(8); % elements to apply forces to
    FuncHandle = @fdt_qdr_quad; % handle to function to apply external force
    tract = o.tract; % traction
end

function Sflux = fSflux(o,V)
    %zero all crack fluxes: 1 is Sflux0, 2 is SfluxL
    Sflux=zeros(V.nel,2);
    for i = (V.nel_bulk+1):V.nel
        Sflux(i,:)=[0 0]*V.time_factor; %m/s, or m/d
    end
    % Sflux(72,:)=[0 -1e-2]*V.time_factor; %m/s, or m/d
    % Sflux(72,:)=[-1e-2 0]*V.time_factor*10; %m/s, or m/d
end

function PlotData(o, V)
    % function to plot data for this mesh def
    V.NDCh.Trim;

    % d = V.NDCh.d;
    T = 10;
    % vertical pressures away from crack
    Node=35; % node outside of crack
    x = V.NDCh.X0(Node);
    nodes = 1:V.nnodes;
    xi = (V.NDCh.X0 == x);
    xi = xi & V.NDCh.PressEnable;
    NodeIDs1 = nodes(xi);
    NodeP1 = V.NDCh.NodePt(NodeIDs1,T); % node pressures
    Y1 = V.NDCh.Y0(xi); % node y positions
    leg1 = 'Pressure away from crack';
    % vertical pressures at crack
    nodes = 1:V.nnodes;
    xi = (V.NDCh.X0 == 0.5);
    xi = xi & V.NDCh.PressEnable;
    NodeIDs2 = nodes(xi);
    NodeP2 = V.NDCh.NodePt(NodeIDs2,T);
    Y2 = V.NDCh.Y0(NodeIDs2);
    leg2 = 'Pressure at crack';
    % plot pressures
B.4.5 Problem Definition 140

```matlab
figure; hold on;
h1 = plot(NodeP1, Y1, 'b', 'LineWidth', 2);
h2 = plot(NodeP2, Y2, 'r', 'LineWidth', 2);
legend([h1 h2], leg1, leg2);
end
```

classdef ProbDef140 < ProbDefXX

% 64 bulk elements (8 x 8) with 0 CSE’s located in the center,
% same as ProbDef14 without except for 0 CSEs
% oriented vertically
% created on 11/03/2012
% UPDATED: 11.07.2012
% UPDATED: 11.25.2012

methods

function o = ProbDef140
    o.LoadMeshProps;
    o.LoadDefaults; % preallocate all values
    o.LoadMesh; % load the mesh which then calls to load ElemM
end

% *******MOST CHANGES FROM MESH TO MESH GO HERE********

function LoadMeshProps(o)
    % these are the main mesh properties that need to be changed to
    % define a new problem
    o.MeshName = 'Problem Definition 14';
    % traction on top elements
    o.tract = AI.Tract;
    % define the region
    o.x0 = 0; o.x1 = 1; % physical x domain
    o.y0 = 0; o.y1 = 1; % physical y domain
    % Number of *bulk* elements in x and y directions
    o.NEBx = 8; o.NEBy = 8;
    % estimate the number of nodes in the problem
    o.NN = o.NEBx*o.NEBy*10; % # of nodes (estimate) for preallocation
    % Specify which nodes should be used to plot vertical displ and
    % pressure for post processing
    o.node_disp_plot = 17; % node to plot vertical displ. at
```matlab
% node to plot fluid pressure at
o.node_press_plot = 1;

function LoadBCs(o)
    n = o.NN;
    xf = false(n, 1);
    yf = false(n, 1);
    % specify which nodes have a fixed displ.
    % Nodes at y = 0 are fixed in x and y directions
    Yi = (o.YCoords==0);
    xf(Yi) = true; % fixed in x direction
    yf(Yi) = true; % fixed in y direction
    % Nodes at x = 0 are fixed in x direction
    Xi = (o.XCoords==0);
    xf(Xi) = true; % fixed in x direction
    % Nodes at x = 1 are fixed in x direction
    Xi = (o.XCoords==1);
    xf(Xi) = true; % fixed in x direction
    % Add to o.Xf and o.Yf
    o.Xf(1:n) = xf; o.Yf(1:n) = yf;
    % specify which nodes do not have a pressure DOF
    % Top nodes
    Yi = (o.YCoords==1);
    o.Pbc(Yi) = true;
    % center nodes
    Xi = (o.XCoords==0.5);
    o.Pbc(Xi) = true;
    % set the nodes created to be active
    o.NodeActive(1:numel(o.XCoords)) = true;
end

function MakeCSEs(o, EMCh)
    % create the CSE's for the mesh given
    % EMCh = handle to element manager class
    % create CSE's by calling the function in Element manager class
    % if o.setupmode == false
    %    EMCh.InsertCSEsBetweenCols(4, 5, AI.aperture);
    end
end

function [ElementIDs FuncHandle tract] = ApplyExternalForces(o, EMCh)
    [ElementIDs, ~] = EMCh.GetElemsInRow(8); % elements to apply forces to
    FuncHandle = @fdt_qdr_quad; % handle to function to apply external force
    tract = o.tract; % traction
end

function Sflux = f_Sflux(o,V)
    % zero all crack fluxes: 1 is Sflux0, 2 is SfluxL
    Sflux=zeros(V.nel,2);
    for i = (V.nel_bulk+1):V.nel
        Sflux(i,:)=[0 0]*V.time_factor; %m/s, or m/d
    end
    % Sflux(72,:)=[0 -1e-2]*V.time_factor; %m/s, or m/d
end
```
B.4.6 Problem Definition 16

classdef ProbDef16 < ProbDefXX
    % MESH WITH 72 ELEMENTS AND FLUID FLUX SPECIFIED AT LEFT SIDE
    % 64 bulk elements,
    % 8 cohesive surface elements inserted horizontally along midplane
    % A fluid flux is specified at the left side of the mesh (element 65)
    methods
        function o = ProbDef16
            o.LoadMeshProps;
            o.LoadDefaults; % preallocate all values
            o.LoadMesh; % load the mesh which then calls to load ElemM
        end
    end

    % ***********MOST CHANGES FROM MESH TO MESH GO HERE***********
    function LoadMeshProps(o)
        % these are the main mesh properties that need to be changed to
        % define a new problem
        o.MeshName = 'Problem Definition 16';
        % traction on top elements
        o.tract = AI.Tract;
        % define the region
        o.x0 = 0; o.x1 = 1; % physical x domain
        o.y0 = 0; o.y1 = 1; % physical y domain
        % Number of *bulk* elements in x and y directions
        o.NEBx = 8; o.NEBy = 8;
        % estimate the number of nodes in the problem
        o.NN = o.NEBx*o.NEBy*10; % # of nodes (estimate) for preallocation
        % Specify which nodes should be used to plot vertical displ and
        % pressure for post processing
        o.node_disp_plot = 17; % node to plot vertical displ. at
        o.node_press_plot = 1; % node to plot fluid pressure at
    end

    function LoadBCs(o)
        % specifies the boundary conditions for the problem
        n = o.NN;
        xf = false(n, 1); yf = false(n, 1);
% specify which nodes have a fixed displ.
% Nodes at bottom are fixed in the y directions
Yi = (o.YCoords==o.y0); % index of nodes at bottom of mesh
xf(Yi) = true; % fixed in x direction
yf(Yi) = true; % fixed in y direction

% Nodes at top of mesh
Yi = (o.YCoords==o.y1); % index of nodes at top of mesh
yf(Yi) = true; % fixed in the y direction

% Nodes at x = 0 are fixed in x direction
Xi = (o.XCoords==o.x0); % index of nodes at x = 0
xf(Xi) = true; % fixed in x direction

% Nodes at x = 1 are fixed in x direction
Xi = (o.XCoords==o.x1); % index of nodes at x = 1
xf(Xi) = true; % fixed in x direction

% Add to o.Xf and o.Yf
o.Xf(1:n) = xf; o.Yf(1:n) = yf;

% specify which nodes have a fixed displ.

% specify pressure info
% specify which nodes do not have a pressure DOF
% Top nodes
Yi = (o.YCoords==1); % index of top nodes
o.Pbc(Yi) = true;

% center nodes
% set the nodes created to be active
o.NodeActive(1:numel(o.XCoords)) = true;

function MakeCSEs(o, EMCh)
% create the CSE's for the mesh given
% EMCh = handle to element manager class
% create CSE's by calling the function in Element manager class
if o.setupmode == false
    EMCh.InsertCSEsBetweenRows(4, 5, AI.aperture);
    EMCh.InsertCSEBetweenElems(33, 34, AI.aperture, 0);
    EMCh.InsertCSEBetweenElems(25, 26, AI.aperture, 0);
end

end

function [ElementIDs FuncHandle tract] = ApplyExternalForces(o, EMCh)
% [ElementIDs,tracts] = EMCh.GetElemsInRow(8); % elements to apply forces to
ElementIDs = [];
FuncHandle = @fdt_qdr_quad; % handle to function to apply external force
tract = o.tract; % traction

end

function Sflux = f_Sflux(o,V)
% zero all crack fluxes: 1 is Sflux0, 2 is SfluxL
Sflux=zeros(V.nel,2);
for i = (V.nel_bulk+1):V.nel
    Sflux(i,:)=[0 0]*V.time_factor; %m/s, or m/d
end
Sflux(65,:)=[0 1e-2]*V.time_factor; %m/s, or m/d
100   end
101   end
102
103   end
B.5 Bulk Element Functions

```matlab
classdef BKLP
    
    static
    
    methods (Static)
    
    % code for 3x3 Gaussian Integration of body force tangent
    function df = dfdfdd_qdr_quad(coords, porosity, del_eps, grav, ...
         rho_f, rho_s, nel_bulk)
        df = zeros(18, 18, nel_bulk);
        for el = 1:nel_bulk
            coordsx = coords(el, :);
            porosity_el = porosity(el,:);
            del_eps_el = del_eps(el,:);
            
            % nodal coordinates of elements in local node numbering
            x1 = coordsx(1); y1 = coordsx(2);
            x2 = coordsx(3); y2 = coordsx(4);
            x3 = coordsx(5); y3 = coordsx(6);
            x4 = coordsx(7); y4 = coordsx(8);
            x5 = coordsx(9); y5 = coordsx(10);
            x6 = coordsx(11); y6 = coordsx(12);
            x7 = coordsx(13); y7 = coordsx(14);
            x8 = coordsx(15); y8 = coordsx(16);
            x9 = coordsx(17); y9 = coordsx(18);
            
            % initialize stiffness matrix
            df_e=zeros(18,18);
            W=zeros(9,1);
            
            % set Gauss point coordinates in xi,eta space
            const1=sqrt(3/5);
            const2=25/81;
            const3=40/81;
            const4=64/81;
            xi_vect(1,:)=[-const1 -const1];
            xi_vect(2,:)=[const1 -const1];
            xi_vect(3,:)=[const1 const1];
            xi_vect(4,:)=[-const1 const1];
            xi_vect(5,:)=[0 -const1];
            xi_vect(6,:)=[const1 0];
            xi_vect(7,:)=[0 const1];
            xi_vect(8,:)=[-const1 0];
            xi_vect(9,:)=[0 0];
            W(1)=const2;
            W(2)=const2;
            W(3)=const2;
            W(4)=const2;
            W(5)=const3;
            W(6)=const3;
            W(7)=const3;
            W(8)=const3;
```

\[ W(9) = \text{const}4; \]

% loop through the 4 Gauss points
for i = 1:9
    xi = xi_vect(i, 1);
    eta = xi_vect(i, 2);

% derivatives of shape functions with respect to \( \xi \)
\[
\begin{align*}
\text{dN}_1^{\text{d}\xi} &= 0.25 \times (2 \times \xi) \times \eta \times (\eta - 1); \\
\text{dN}_2^{\text{d}\xi} &= 0.25 \times (2 \times \xi + 1) \times \eta \times (\eta - 1); \\
\text{dN}_3^{\text{d}\xi} &= 0.25 \times (2 \times \xi + 1) \times \eta \times (\eta + 1); \\
\text{dN}_4^{\text{d}\xi} &= 0.25 \times (2 \times \xi - 1) \times \eta \times (\eta + 1); \\
\text{dN}_5^{\text{d}\xi} &= 0.5 \times (2 \times \xi) \times \eta \times (1 - \eta^2); \\
\text{dN}_6^{\text{d}\xi} &= 0.5 \times (2 \times \xi) \times \eta \times (\eta + 1); \\
\text{dN}_7^{\text{d}\xi} &= 0.5 \times (2 \times \xi - 1) \times \eta \times (1 - \eta^2); \\
\text{dN}_9^{\text{d}\xi} &= (2 \times \xi) \times (1 - \eta^2);
\end{align*}
\]

% derivatives of shape functions with respect to \( \eta \)
\[
\begin{align*}
\text{dN}_1^{\text{d}\eta} &= 0.25 \times (2 \times \eta) \times (\eta - 1); \\
\text{dN}_2^{\text{d}\eta} &= 0.25 \times (2 \times \eta + 1) \times (\eta - 1); \\
\text{dN}_3^{\text{d}\eta} &= 0.25 \times (2 \times \eta + 1) \times (\eta + 1); \\
\text{dN}_4^{\text{d}\eta} &= 0.25 \times (2 \times \eta - 1) \times (\eta + 1); \\
\text{dN}_5^{\text{d}\eta} &= 0.5 \times (2 \times \eta - 1) \times (1 - \xi^2); \\
\text{dN}_6^{\text{d}\eta} &= 0.5 \times (2 \times \eta - 1) \times (\xi + 1); \\
\text{dN}_7^{\text{d}\eta} &= 0.5 \times (2 \times \eta + 1) \times (1 - \xi^2); \\
\text{dN}_9^{\text{d}\eta} &= (2 \times \eta) \times (1 - \xi^2);
\end{align*}
\]

% calculate jacobian, its determinant, and its inverse
\[
\begin{align*}
\text{j} &= \det(\text{Je}) ; \\
\text{Je} &= \begin{bmatrix}
\text{dx}_\xi & \text{dx}_\eta \\
\text{dy}_\xi & \text{dy}_\eta
\end{bmatrix} ; \\
\text{Jeinv} &= \frac{1}{\text{j}} \times [ \text{dy}_\eta \times -\text{dx}_\eta ; -\text{dy}_\xi \times \text{dx}_\xi ];
\end{align*}
\]

% shape functions
\[
\begin{align*}
\text{N}_1 &= 0.25 \times \xi \times \eta \times (\xi - 1) \times (\eta - 1); \\
\text{Nmatrix}_1 &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}; \\
\text{N}_2 &= 0.25 \times \xi \times \eta \times (\xi + 1) \times (\eta - 1); \\
\text{Nmatrix}_2 &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}; \\
\text{N}_3 &= 0.25 \times \xi \times \eta \times (\xi + 1) \times (\eta + 1); \\
\text{Nmatrix}_3 &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}; \\
\text{N}_4 &= 0.25 \times \xi \times \eta \times (\xi - 1) \times (\eta + 1); \\
\text{Nmatrix}_4 &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}; \\
\text{N}_5 &= 0.5 \times \eta \times (1 - \xi^2) \times (\eta - 1); \\
\text{Nmatrix}_5 &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}; \\
\text{N}_6 &= 0.5 \times \xi \times (1 - \eta^2) \times (\xi + 1); \\
\text{Nmatrix}_6 &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}; \\
\text{N}_7 &= 0.5 \times \eta \times (1 - \eta^2) \times (\eta + 1); \\
\text{Nmatrix}_7 &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix};
\end{align*}
\]
Nmatrix7 = [ N7 0 ; 0 N7 ];
N8 = 0.5*xi*(1-etaˆ2)*(xi-1);
Nmatrix8 = [ N8 0 ; 0 N8 ];
N9 = (1-etaˆ2)*(1-xiˆ2);
Nmatrix9 = [ N9 0 ; 0 N9 ];

Nmatrix=[ Nmatrix1 Nmatrix2 Nmatrix3 Nmatrix4 Nmatrix5 Nmatrix6 ... Nmatrix7 Nmatrix8 Nmatrix9 ];

%for Btilde
dN1_dx_vect = [dN1_dx dN1_deta]*Jeinv;
B1 = dN1_dx_vect;
dN2_dx_vect = [dN2_dx dN2_deta]*Jeinv;
B2 = dN2_dx_vect;
dN3_dx_vect = [dN3_dx dN3_deta]*Jeinv;
B3 = dN3_dx_vect;
dN4_dx_vect = [dN4_dx dN4_deta]*Jeinv;
B4 = dN4_dx_vect;
dN5_dx_vect = [dN5_dx dN5_deta]*Jeinv;
B5 = dN5_dx_vect;
dN6_dx_vect = [dN6_dx dN6_deta]*Jeinv;
B6 = dN6_dx_vect;
dN7_dx_vect = [dN7_dx dN7_deta]*Jeinv;
B7 = dN7_dx_vect;
dN8_dx_vect = [dN8_dx dN8_deta]*Jeinv;
B8 = dN8_dx_vect;
dN9_dx_vect = [dN9_dx dN9_deta]*Jeinv;
B9 = dN9_dx_vect;
Btilde=[B1 B2 B3 B4 B5 B6 B7 B8 B9];
g=[0; -grav];
coeff = (rhof-rhos)*(1-porosity_el(i))/(1+del_epsv_el(i));
df_e = df_e+coeff*Nmatrix'*g*Btilde*j*W(i);
end
df(:,:,el) = df_e;
end
end

% code for bilinear quadrilateral element pore fluid pressure
function pf = element_pf_quad(d, nel_bulk)
% old call: pf_el(:,1,el)=element_pf_quad(coords(el,:),d_el(el,:));
% initialize output
pf = zeros(9, 1, nel_bulk);

for el = 1:nel_bulk
  % number Gauss points by number of pf values
  pf_el=zeros(9,1);
  d_el = d(el,:);
  % set Gauss point coordinates in xi,eta space
  const1=sqrt(3/5);
  const2=25/81;
  const3=40/81;
  const4=64/81;
  xi_vect(1,:)=[-const1 -const1];
  xi_vect(2,:)=[const1 -const1];
  xi_vect(3,:)=[const1 const1];
  xi_vect(4,:)=[-const1 const1];
  xi_vect(5,:)=[0 -const1];
  xi_vect(6,:)=[const1 0];
  xi_vect(7,:)=[0 const1];
  xi_vect(8,:)=[-const1 0];
  xi_vect(9,:)=[0 0];

  % loop through the 9 Gauss points
  for i=1:9
    xi=xi_vect(i,1);
    eta=xi_vect(i,2);

    % linear shape functions for pore pressure distribution
    N1_p = 0.25*(1-xi)*(1-eta);
    N2_p = 0.25*(1+xi)*(1-eta);
    N3_p = 0.25*(1+xi)*(1+eta);
    N4_p = 0.25*(1-xi)*(1+eta);
    Nmatrix = [ N1_p N2_p N3_p N4_p ];

    pf_el(i) = Nmatrix * d_el(19:22)';
  end
  pf(:,1,el) = pf_el;
end

% code for 3x3 Gaussian Integration of internal force
function fint = fdint_quad(coords, stress, nel_bulk)

  fint = zeros(18,nel_bulk);

  for el = 1:nel_bulk
    % setup variables for loop
    coordsx = coords(el,:);
    stress_el = stress(:,:,el);
    % loop variable
    fint_e = zeros(18,1);
% nodal coordinates of elements in local node numbering
x1 = coordsx(1); y1 = coordsx(2);
x2 = coordsx(3); y2 = coordsx(4);
x3 = coordsx(5); y3 = coordsx(6);
x4 = coordsx(7); y4 = coordsx(8);
x5 = coordsx(9); y5 = coordsx(10);
x6 = coordsx(11); y6 = coordsx(12);
x7 = coordsx(13); y7 = coordsx(14);
x8 = coordsx(15); y8 = coordsx(16);
x9 = coordsx(17); y9 = coordsx(18);

% initialize stiffness matrix
W=zeros(9,1);

% set Gauss point coordinates in xi,eta space
const1=sqrt(3/5);
const2=25/81;
const3=40/81;
const4=64/81;
xi_vect(1,:)=[-const1 -const1];
xi_vect(2,:)=[const1 -const1];
xi_vect(3,:)=[const1 const1];
xi_vect(4,:)=[-const1 const1];
xi_vect(5,:)=[0 -const1];
xi_vect(6,:)=[const1 0];
xi_vect(7,:)=[0 const1];
xi_vect(8,:)=[-const1 0];
xi_vect(9,:)=[0 0];
W(1)=const2;
W(2)=const2;
W(3)=const2;
W(4)=const2;
W(5)=const3;
W(6)=const3;
W(7)=const3;
W(8)=const3;
W(9)=const4;

% loop through the 4 Gauss points
for i=1:9
    xi=xi_vect(i,1);
    eta=xi_vect(i,2);
    % derivatives of shape functions with respect to xi
    dN1_dxi = 0.25*(2*xi-1)*eta*(eta-1);
    dN2_dxi = 0.25*(2*xi+1)*eta*(eta-1);
    dN3_dxi = 0.25*(2*xi+1)*eta*(eta+1);
    dN4_dxi = 0.25*(2*xi-1)*eta*(eta+1);
    dN5_dxi = 0.5*(-2*xi)*eta*(eta-1);
    dN6_dxi = 0.5*(2*xi+1)*(1-eta^2);
    dN7_dxi = 0.5*(-2*xi)*eta*(eta+1);
    dN8_dxi = 0.5*(2*xi-1)*(1-eta^2);
    dN9_dxi = (-2*xi)*(1-eta^2);
    % derivatives of shape functions with respect to eta
    dN1_deta = 0.25*(2*eta-1)*xi*(xi-1);
    dN2_deta = 0.25*(2*eta-1)*xi*(xi+1);
dN3_deta = 0.25*(2*eta+1)*xi*(xi+1);
dN4_deta = 0.25*(2*eta+1)*xi*(xi-1);
dN5_deta = 0.5*(2*eta-1)*(1-xi^2);
dN6_deta = 0.5*xi*(xi+1)*(-2*eta);
dN7_deta = 0.5*(1-xi^2)*(2*eta+1);
dN8_deta = 0.5*xi*(xi-1)*(-2*eta);
dN9_deta = (-2*eta)*(1-xi^2);

% calculate jacobian, its determinant, and its inverse
dx_dxi = dN1_dxi*x1 + dN2_dxi*x2 + dN3_dxi*x3 + dN4_dxi*x4 + dN5_dxi*x5 ...
   + dN6_dxi*x6 + dN7_dxi*x7 + dN8_dxi*x8 + dN9_dxi*x9;
dx_deta = dN1_deta*x1 + dN2_deta*x2 + dN3_deta*x3 + dN4_deta*x4 + ...
   dN5_deta*x5 + dN6_deta*x6 + dN7_deta*x7 + dN8_deta*x8 + dN9_deta*x9;
dy_dxi = dN1_dxi*y1 + dN2_dxi*y2 + dN3_dxi*y3 + dN4_dxi*y4 + dN5_dxi*y5 ...
   + dN6_dxi*y6 + dN7_dxi*y7 + dN8_dxi*y8 + dN9_dxi*y9;
dy_deta = dN1_deta*y1 + dN2_deta*y2 + dN3_deta*y3 + dN4_deta*y4 + ...
   dN5_deta*y5 + dN6_deta*y6 + dN7_deta*y7 + dN8_deta*y8 + dN9_deta*y9;
Je = [ dx_dxi dx_deta ; dy_dxi dy_deta ];

j=det(Je);
Jeinv = (1/j)*[ dy_deta -dx_deta ; -dy_dxi dx_dxi ];

B1 = [ dN1_dxi_dvec(1) 0 ; 0 dN1_dxi_dvec(2) ; dN1_dxi_dvec(2) ... 
      dN1_dxi_dvec(1) ];
B2 = [ dN2_dxi_dvec(1) 0 ; 0 dN2_dxi_dvec(2) ; dN2_dxi_dvec(2) ... 
      dN2_dxi_dvec(1) ];
B3 = [ dN3_dxi_dvec(1) 0 ; 0 dN3_dxi_dvec(2) ; dN3_dxi_dvec(2) ... 
      dN3_dxi_dvec(1) ];
B4 = [ dN4_dxi_dvec(1) 0 ; 0 dN4_dxi_dvec(2) ; dN4_dxi_dvec(2) ... 
      dN4_dxi_dvec(1) ];
B5 = [ dN5_dxi_dvec(1) 0 ; 0 dN5_dxi_dvec(2) ; dN5_dxi_dvec(2) ... 
      dN5_dxi_dvec(1) ];
B6 = [ dN6_dxi_dvec(1) 0 ; 0 dN6_dxi_dvec(2) ; dN6_dxi_dvec(2) ... 
      dN6_dxi_dvec(1) ];
B7 = [ dN7_dxi_dvec(1) 0 ; 0 dN7_dxi_dvec(2) ; dN7_dxi_dvec(2) ... 
      dN7_dxi_dvec(1) ];
B8 = [ dN8_dxi_dvec(1) 0 ; 0 dN8_dxi_dvec(2) ; dN8_dxi_dvec(2) ... 
      dN8_dxi_dvec(1) ];
B9 = [ dN9_dxi_dvec(1) 0 ; 0 dN9_dxi_dvec(2) ; dN9_dxi_dvec(2) ... 
      dN9_dxi_dvec(1) ];
B=[B1 B2 B3 B4 B5 B6 B7 B8 B9];
fint_e=fint_e+B' * stress_el(i,:) * j*W(i);
end
fint(:,el) = fint_e;
end
end

% assemble fdint, kdtheta, d_el, dthint
function Fdthint = F_Bulk_Combine01(fdint, kdtheta, d_el, fthint, nel_bulk, ...
      nel_bulk_dof, neldbulk_dof)
      % size(d_el(1:nel_bulk,(neld_bulk_dof+1):nel_bulk_dof)) = <4 x 4>
      % Fdthint(:,el) = [ fdint(:,el) - ... 
      %                 kdtheta(:,:,el)*d_el(el,(neld_bulk_dof+1):nel_bulk_dof) ];
      % M = zeros(18,nel_bulk);
      for el = 1:nel_bulk
          % nodal coordinates of elements in local node numbering
          x1 = coordsx(1); y1 = coordsx(2);
          x2 = coordsx(3); y2 = coordsx(4);
          x3 = coordsx(5); y3 = coordsx(6);
          x4 = coordsx(7); y4 = coordsx(8);
          x5 = coordsx(9); y5 = coordsx(10);
          x6 = coordsx(11); y6 = coordsx(12);
          x7 = coordsx(13); y7 = coordsx(14);
          x8 = coordsx(15); y8 = coordsx(16);
          x9 = coordsx(17); y9 = coordsx(18);

          % set Gauss point coordinates in xi,eta space
          const1=sqrt(3/5);
          const2=25/81;

          % code for biquadratic quadrilateral element stress
          function [stress, strain] = element_stress_quad(coords, D, d, nel_bulk)
const3 = 40/81;
const4 = 64/81;
xi_vec(1,:) = [const1 -const1];
xi_vec(2,:) = [const1 -const1];
xi_vec(3,:) = [const1 const1];
xi_vec(4,:) = [-const1 const1];
xi_vec(5,:) = [0 -const1];
xi_vec(6,:) = [0 const1];
xi_vec(7,:) = [-const1 0];
xi_vec(8,:) = [0 0];
xi_vec(9,:) = [0 0];
for i = 1:9
    xi = xi_vec(i,1);
    eta = xi_vec(i,2);
end

% derivatives of shape functions with respect to xi
dN1_dxi = 0.25*(2*xi-1)*eta*(eta-1);
dN2_dxi = 0.25*(2*xi+1)*eta*(eta-1);
dN3_dxi = 0.25*(2*xi+1)*eta*(eta+1);
dN4_dxi = 0.25*(2*xi-1)*eta*(eta+1);
dN5_dxi = 0.5*(-2*xi)*eta*(eta-1);
dN6_dxi = 0.5*(2*xi+1)*(1-eta^2);
dN7_dxi = 0.5*(-2*xi)*eta*(eta+1);
dN8_dxi = 0.5*(2*xi-1)*(1-eta^2);
dN9_dxi = (-2*xi)*(1-eta^2);

% derivatives of shape functions with respect to eta
dN1_deta = 0.25*(2*eta-1)*xi*(xi-1);
dN2_deta = 0.25*(2*eta+1)*xi*(xi+1);
dN3_deta = 0.25*(2*eta+1)*xi*(xi+1);
dN4_deta = 0.25*(2*eta-1)*xi*(xi-1);
dN5_deta = 0.5*(2*eta-1)*(1-xi^2);
dN6_deta = 0.5*xi*(xi+1)*(-2*eta);
dN7_deta = 0.5*(1-xi^2)*(2*eta+1);
dN8_deta = -0.5*xi*(xi+1)*(-2*eta);
dN9_deta = (2*eta)*(1-xi^2);

% calculate jacobian, its determinant, and its inverse
dx_dxi = dN1_dxi*x1 + dN2_dxi*x2 + dN3_dxi*x3 + dN4_dxi*x4 + ... + dN9_dxi*x9;
dx_deta = dN1_deta*x1 + dN2_deta*x2 + dN3_deta*x3 + dN4_deta*x4 + ... + dN9_deta*x9;
dy_dxi = dN1_dxi*y1 + dN2_dxi*y2 + dN3_dxi*y3 + dN4_dxi*y4 + ... + dN9_dxi*y9;
dy_deta = dN1_deta*y1 + dN2_deta*y2 + dN3_deta*y3 + dN4_deta*y4 + ... + dN9_deta*y9;
Je = [ dx_dxi dx_deta ; dy_dxi dy_deta ];
j = det(Je);
Jeinv = (1/j)*[ dy_deta -dx_deta ; -dy_dxi dx_dxi ];

dN1_dx_vect = [dN1_dxi dN1_deta]*Jeinv;
B1 = [ dN1_dx_vect(1) 0 ; 0 dN1_dx_vect(2) ; dN1_dx_vect(3) ... dN1_dx_vect(9) ];

dN2_dx_vect = [dN2_dxi dN2_deta]*Jeinv;
B2 = [ dN2_dx_vect(1) 0 ; 0 dN2_dx_vect(2) ; dN2_dx_vect(3) ... dN2_dx_vect(9) ];
dN3_dx_vect = [dN3_dxi dN3_deta]*Jeinv;
B3 = [ dN3_dx_vect(1) 0 ; 0 dN3_dx_vect(2) ; dN3_dx_vect(2) ... 
      dN3_dx_vect(1) ];

dN4_dx_vect = [dN4_dxi dN4_deta]*Jeinv;
B4 = [ dN4_dx_vect(1) 0 ; 0 dN4_dx_vect(2) ; dN4_dx_vect(2) ... 
      dN4_dx_vect(1) ];

dN5_dx_vect = [dN5_dxi dN5_deta]*Jeinv;
B5 = [ dN5_dx_vect(1) 0 ; 0 dN5_dx_vect(2) ; dN5_dx_vect(2) ... 
      dN5_dx_vect(1) ];

dN6_dx_vect = [dN6_dxi dN6_deta]*Jeinv;
B6 = [ dN6_dx_vect(1) 0 ; 0 dN6_dx_vect(2) ; dN6_dx_vect(2) ... 
      dN6_dx_vect(1) ];

dN7_dx_vect = [dN7_dxi dN7_deta]*Jeinv;
B7 = [ dN7_dx_vect(1) 0 ; 0 dN7_dx_vect(2) ; dN7_dx_vect(2) ... 
      dN7_dx_vect(1) ];

dN8_dx_vect = [dN8_dxi dN8_deta]*Jeinv;
B8 = [ dN8_dx_vect(1) 0 ; 0 dN8_dx_vect(2) ; dN8_dx_vect(2) ... 
      dN8_dx_vect(1) ];

dN9_dx_vect = [dN9_dxi dN9_deta]*Jeinv;
B9 = [ dN9_dx_vect(1) 0 ; 0 dN9_dx_vect(2) ; dN9_dx_vect(2) ... 
      dN9_dx_vect(1) ];

B=[B1 B2 B3 B4 B5 B6 B7 B8 B9];
strain_el(i,:)=B*d_el(1:18)';
stress_el(i,:)=D*B*d_el(1:18)';
end
strain(:,:,el) = strain_el;
stress(:,:,el) = stress_el;

end %

% code for calculating porosity at 9 Gauss Integration points
function porosity = element_porosity_quad(del_evpv, porosity_last, nel_bulk)
  % Original
  % del_eps_vol = <1 x 9>
  % porosity_last = <1 x 9>
  % Modified 10.20.2012
  % del_eps_vol = <6 x 9>
  % initialize output
  porosity = zeros(nel_bulk, 9);
  % initialize porosity at each ip
  porosity_el = zeros(9,1);
  for el = 1:nel_bulk
% loop through the 9 Gauss points
for i=1:9
    porosity_el(i)=(porosity_last_el(i)+del_epsv_el(i)) / (1 + ... 
    del_epsv_el(i));
end
porosity(el,:) = porosity_el;

% code for calculating porosity at 9 Gauss Integration points
function permeability = element_perm_quad(porosity, perm, nel_bulk)
    % Original
    % porosity = [1 x 9]
    % perm = scalar
    permeability = zeros(nel_bulk, 9); % initialize porosity at each ip
    permeability_el=zeros(9,1);
    for el = 1:nel_bulk
        porosity_el = porosity(el,:); % porosity for element
        % loop through the 9 Gauss points
        for i=1:9
            permeability_el(i)=perm*(porosity_el(i)^3)/(1-porosity_el(i)^2);
        end
        permeability(el,:) = permeability_el;
    end
end

% code for 3x3 Gaussian Integration of coupling stiffness matrix
function fthint = element_fluid_int_force(coords, d_el, perm, rhof, grav, ...
    nel bulk, nel bulk dof, nel bulk dof)
    % theta = d el(el, (neld bulk dof+1):nel bulk dof)
    % Old call:
    % fthint(:,el)=element_fluid_int_force(coords(el,:),d el(el, (neld bulk dof+1):nel bulk dof),... 
    %                perm el(el,:),rhof,grav);
    fthint = zeros(4, nel bulk);
    Itheta = (neld bulk dof+1):(nel bulk dof);
    for el = 1:nel bulk
        theta = d el(el, Itheta);
        coordsx = coords(el,:);
        perm_el = perm(el,:);
        % nodal coordinates of elements in local node numbering
x1 = coordsx(1); y1 = coordsx(2);
x2 = coordsx(3); y2 = coordsx(4);
x3 = coordsx(5); y3 = coordsx(6);
x4 = coordsx(7); y4 = coordsx(8);
x5 = coordsx(9); y5 = coordsx(10);
x6 = coordsx(11); y6 = coordsx(12);
x7 = coordsx(13); y7 = coordsx(14);
x8 = coordsx(15); y8 = coordsx(16);
x9 = coordsx(17); y9 = coordsx(18);

% initialize stiffness matrix
fthint_el=zeros(4,1);
W=zeros(9,1);

% set Gauss point coordinates in xi,eta space
const1=sqrt(3/5);
const2=25/81;
const3=40/81;
const4=64/81;
xi_vect(1,:)=[const1 -const1];
xi_vect(2,:)=[const1 -const1];
xi_vect(3,:)=[const1 const1];
xi_vect(4,:)=[-const1 const1];
xi_vect(5,:)=[0 -const1];
xi_vect(6,:)=[const1 0];
xi_vect(7,:)=[0 const1];
xi_vect(8,:)=[-const1 0];
xi_vect(9,:)=[0 0];

% loop through the 4 Gauss points
for i=1:9
    xi=xi_vect(i,1);
    eta=xi_vect(i,2);
    
    % derivatives of shape functions with respect to xi
dN1_dxi = 0.25*(2*xi-1)*eta*(eta-1);
dN2_dxi = 0.25*(2*xi+1)*eta*(eta+1);
dN3_dxi = 0.25*(2*xi-1)*eta*(eta+1);
dN4_dxi = 0.25*(2*xi+1)*eta*(eta-1);
dN5_dxi = 0.5*(-2*xi)*eta*(eta-1);
dN6_dxi = 0.5*(2*xi+1)*(1-eta^2);
dN7_dxi = 0.5*(-2*xi)*eta*(eta+1);
dN8_dxi = 0.5*(2*xi-1)*(1-eta^2);
dN9_dxi = (-2*xi)*(1-eta^2);

    % derivatives of shape functions with respect to eta
dN1_deta = 0.25*(2*eta-1)*xi*(xi-1);
dN2_deta = 0.25*(2*eta+1)*xi*(xi+1);
dN3_deta = 0.25*(2*eta+1)*xi*(xi+1);
\[ dN_4 = 0.25 \times (2 \times \eta + 1) \times x_i \times (x_i - 1); \]
\[ dN_5 = 0.5 \times (2 \times \eta - 1) \times (1 - x_i^2); \]
\[ dN_6 = 0.5 \times x_i \times (x_i + 1) \times (-2 \times \eta); \]
\[ dN_7 = 0.5 \times (1 - x_i^2) \times (2 \times \eta + 1); \]
\[ dN_8 = 0.5 \times x_i \times (x_i - 1) \times (-2 \times \eta); \]
\[ dN_9 = (-\eta) \times (1 - x_i^2); \]

\[ \frac{\partial N_4}{\partial \eta} = 0.25 \times (2 \times \eta + 1) \times x_i \times (x_i - 1); \]
\[ \frac{\partial N_5}{\partial \eta} = 0.5 \times (2 \times \eta - 1) \times (1 - x_i^2); \]
\[ \frac{\partial N_6}{\partial \eta} = 0.5 \times x_i \times (x_i + 1) \times (-2 \times \eta); \]
\[ \frac{\partial N_7}{\partial \eta} = 0.5 \times (1 - x_i^2) \times (2 \times \eta + 1); \]
\[ \frac{\partial N_8}{\partial \eta} = 0.5 \times x_i \times (x_i - 1) \times (-2 \times \eta); \]
\[ \frac{\partial N_9}{\partial \eta} = (-\eta) \times (1 - x_i^2); \]

\[ \frac{\partial N_1}{\partial x_i} = dN_1 \times x_i + dN_2 \times x_2 + dN_3 \times x_3 + dN_4 \times x_4 + dN_5 \times x_5 \ldots \]
\[ + dN_6 \times x_6 + dN_7 \times x_7 + dN_8 \times x_8 + dN_9 \times x_9; \]
\[ \frac{\partial N_1}{\partial \eta} = \frac{1}{2} \times (2 \times \eta + 1) \times x_i \times (x_i - 1); \]
\[ \frac{\partial N_1}{\partial x_i} = \frac{1}{2} \times \eta \times x_i \times (x_i - 1); \]
\[ \frac{\partial N_2}{\partial x_i} = \frac{1}{2} \times (2 \times \eta + 1) \times x_i \times (x_i - 1); \]
\[ \frac{\partial N_2}{\partial \eta} = \frac{1}{2} \times \eta \times (x_i - 1); \]
\[ \frac{\partial N_3}{\partial x_i} = \frac{1}{2} \times \eta \times x_i \times (x_i - 1); \]
\[ \frac{\partial N_3}{\partial \eta} = \frac{1}{2} \times \eta \times (x_i - 1); \]
\[ \frac{\partial N_4}{\partial x_i} = \frac{1}{2} \times \eta \times x_i \times (x_i - 1); \]
\[ \frac{\partial N_4}{\partial \eta} = \frac{1}{2} \times \eta \times (x_i - 1); \]

\[ \frac{\partial N_5}{\partial x_i} = \frac{1}{2} \times \eta \times x_i \times (x_i - 1); \]
\[ \frac{\partial N_5}{\partial \eta} = \frac{1}{2} \times \eta \times (x_i - 1); \]
\[ \frac{\partial N_6}{\partial x_i} = \frac{1}{2} \times \eta \times x_i \times (x_i - 1); \]
\[ \frac{\partial N_6}{\partial \eta} = \frac{1}{2} \times \eta \times (x_i - 1); \]
\[ \frac{\partial N_7}{\partial x_i} = \frac{1}{2} \times \eta \times x_i \times (x_i - 1); \]
\[ \frac{\partial N_7}{\partial \eta} = \frac{1}{2} \times \eta \times (x_i - 1); \]
\[ \frac{\partial N_8}{\partial x_i} = \frac{1}{2} \times \eta \times x_i \times (x_i - 1); \]
\[ \frac{\partial N_8}{\partial \eta} = \frac{1}{2} \times \eta \times (x_i - 1); \]
\[ \frac{\partial N_9}{\partial x_i} = \frac{1}{2} \times \eta \times (x_i - 1); \]
\[ \frac{\partial N_9}{\partial \eta} = \frac{1}{2} \times \eta \times (x_i - 1); \]

% calculate jacobian, its determinant, and its inverse
\[ \frac{\partial x}{\partial \eta} = dN_1 \frac{\partial x_i}{\partial \eta} + dN_2 \frac{\partial x_i}{\partial \eta} + dN_3 \frac{\partial x_i}{\partial \eta} + dN_4 \frac{\partial x_i}{\partial \eta} + dN_5 \frac{\partial x_i}{\partial \eta} \ldots \]
\[ + dN_6 \frac{\partial x_i}{\partial \eta} + dN_7 \frac{\partial x_i}{\partial \eta} + dN_8 \frac{\partial x_i}{\partial \eta} + dN_9 \frac{\partial x_i}{\partial \eta}; \]
\[ \frac{\partial y}{\partial \eta} = dN_1 \frac{\partial y_i}{\partial \eta} + dN_2 \frac{\partial y_i}{\partial \eta} + dN_3 \frac{\partial y_i}{\partial \eta} + dN_4 \frac{\partial y_i}{\partial \eta} + dN_5 \frac{\partial y_i}{\partial \eta} \ldots \]
\[ + dN_6 \frac{\partial y_i}{\partial \eta} + dN_7 \frac{\partial y_i}{\partial \eta} + dN_8 \frac{\partial y_i}{\partial \eta} + dN_9 \frac{\partial y_i}{\partial \eta}; \]
\[ \frac{\partial x}{\partial \xi} = \frac{\partial x_i}{\partial \xi} + \frac{\partial y_i}{\partial \eta}; \]
\[ \frac{\partial y}{\partial \xi} = \frac{\partial y_i}{\partial \xi} + \frac{\partial x_i}{\partial \eta}; \]
\[ \text{Je} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} \end{bmatrix}; \]
\[ j = \text{det} (\text{Je}); \]
\[ \text{Je}^{-1} = \frac{1}{j} \begin{bmatrix} \frac{\partial y}{\partial \eta} & -\frac{\partial x}{\partial \eta} \\ -\frac{\partial y}{\partial \xi} & \frac{\partial x}{\partial \xi} \end{bmatrix}; \]

% derivatives of linear shape functions with respect to x_i
\[ dN_1 \frac{\partial x_i}{\partial \xi} = -0.25 \times (1 - \eta); \]
\[ dN_2 \frac{\partial x_i}{\partial \xi} = -dN_1 \frac{\partial x_i}{\partial \xi}; \]
\[ dN_3 \frac{\partial x_i}{\partial \xi} = 0.25 \times (1 + \eta); \]
\[ dN_4 \frac{\partial x_i}{\partial \xi} = -dN_3 \frac{\partial x_i}{\partial \xi}; \]

% derivatives of linear shape functions with respect to f_i
\[ dN_1 \frac{\partial f_i}{\partial \eta} = 0.25 \times (1 - \eta); \]
\[ dN_2 \frac{\partial f_i}{\partial \eta} = -dN_1 \frac{\partial f_i}{\partial \eta}; \]
\[ dN_3 \frac{\partial f_i}{\partial \eta} = -dN_1 \frac{\partial f_i}{\partial \eta}; \]
\[ dN_4 \frac{\partial f_i}{\partial \eta} = -dN_1 \frac{\partial f_i}{\partial \eta}; \]

\[ dN_1 \frac{\partial f_i}{\partial \xi} = dN_1 \frac{\partial x_i}{\partial \xi} + dN_1 \frac{\partial y_i}{\partial \eta}; \]
\[ B_1 = dN_1 \frac{\partial x_i}{\partial \xi} + dN_1 \frac{\partial y_i}{\partial \eta}; \]
\[ B_2 = dN_2 \frac{\partial x_i}{\partial \xi} + dN_2 \frac{\partial y_i}{\partial \eta}; \]
\[ B_3 = dN_3 \frac{\partial x_i}{\partial \xi} + dN_3 \frac{\partial y_i}{\partial \eta}; \]
\[ B_4 = dN_4 \frac{\partial x_i}{\partial \xi} + dN_4 \frac{\partial y_i}{\partial \eta}; \]
\[ B = [B_1, B_2, B_3, B_4]; \]
\[ \text{gravity} = [0; -\text{grav}]; \]
\[ \text{velocity} = \text{perm} \times B \times \text{theta}' + \text{rhof} \times \text{perm} \times \text{gravity}; \]
\[ f\text{thint} = f\text{thint} + B' \times \text{velocity} \times j \times W(i); \]
\[ \text{fthint}(:, i) = f\text{thint} = f\text{thint} ; \]
\[ \text{end} \]
\[ \text{fthint}(:, i) = f\text{thint} = f\text{thint} ; \]
\[ \text{end} \]
\[ \text{end} \]
% code for calculating porosity at 9 Gauss Integration points

function density = element_density_quad(porosity, rhos, rhof, nel_bulk)

    % initialize output
    density = zeros(nel_bulk, 9);

    % initialize porosity at each ip
    density_el = zeros(1, 9);

    for el = 1:nel_bulk
        porosity_el = porosity(el,:);
        % loop through the 9 Gauss points
        for i=1:9
            density_el(i) = rhos + porosity_el(i) * (rhof - rhos);
        end
        density(el,:) = density_el;
    end

end

% code for calculating Delta eps_v at 9 Gauss Integration points

function eps_vol = element_del_vol_quad(coords, Delta_d_el, neld_bulk_dof, ... nel_bulk)

    eps_vol = zeros(nel_bulk, 9); % # rows = nel, # rows filled here = nel_bulk

    for el = 1:nel_bulk
        coordsx = coords(el,:); % coords for element el
        A_d = Delta_d_el(el, 1:neld_bulk_dof);

        % nodal coordinates of elements in local node numbering
        x1 = coordsx(1); y1 = coordsx(2);
        x2 = coordsx(3); y2 = coordsx(4);
        x3 = coordsx(5); y3 = coordsx(6);
        x4 = coordsx(7); y4 = coordsx(8);
        x5 = coordsx(9); y5 = coordsx(10);
        x6 = coordsx(11); y6 = coordsx(12);
        x7 = coordsx(13); y7 = coordsx(14);
        x8 = coordsx(15); y8 = coordsx(16);
        x9 = coordsx(17); y9 = coordsx(18);

        % initialize volumetric strain at each ip
        eps_vol_el = zeros(1,9);

        % set Gauss point coordinates in xi,eta space
        const1=sqrt(3/5);
        xi_vect(1,:)=[-const1 -const1];
        xi_vect(2,:)=[const1 -const1];
        xi_vect(3,:)=[const1 const1];
        xi_vect(4,:)=[-const1 const1];
        xi_vect(5,:)=[0 -const1];
        xi_vect(6,:)=[const1 0];
        xi_vect(7,:)=[0 const1];
        xi_vect(8,:)=[-const1 0];
        xi_vect(9,:)=[0 0];

        % loop through the 9 Gauss points
for i=1:9
    xi=xi_vect(i,1);
    eta=xi_vect(i,2);

    % derivatives of shape functions with respect to xi
    dN1_dxi = 0.25*(2*xi-1)*eta*(eta-1);
    dN2_dxi = 0.25*(2*xi+1)*eta*(eta-1);
    dN3_dxi = 0.25*(2*xi+1)*eta*(eta+1);
    dN4_dxi = 0.25*(2*xi-1)*eta*(eta+1);
    dN5_dxi = 0.5*(-2*xi)*eta*(eta-1);
    dN6_dxi = 0.5*(2*xi+1)*(1-eta^2);
    dN7_dxi = 0.5*(-2*xi)*eta*(eta+1);
    dN8_dxi = 0.5*(2*xi-1)*(1-eta^2);
    dN9_dxi = (-2*xi)*(1-eta^2);

    % derivatives of shape functions with respect to eta
    dN1_deta = 0.25*(2*eta-1)*xi*(xi-1);
    dN2_deta = 0.25*(2*eta+1)*xi*(xi+1);
    dN3_deta = 0.25*(2*eta+1)*xi*(xi+1);
    dN4_deta = 0.25*(2*eta-1)*xi*(xi-1);
    dN5_deta = 0.5*(2*eta-1)*(1-xi^2);
    dN6_deta = 0.5*xi*(xi+1)*(-2*eta);
    dN7_deta = 0.5*(1-xi^2)*(2*eta+1);
    dN8_deta = 0.5*xi*(xi-1)*(-2*eta);
    dN9_deta = (-2*eta)*(1-xi^2);

    % calculate jacobian, its determinant, and its inverse
    dx_dxi = dN1_dxi*x1 + dN2_dxi*x2 + dN3_dxi*x3 + dN4_dxi*x4 + ...
    dy_dxi = dN1_deta*y1 + dN2_deta*y2 + dN3_deta*y3 + dN4_deta*y4 + ...
    dN5_dxi*y5 + dN6_dxi*y6 + dN7_dxi*y7 + dN8_dxi*y8 + dN9_dxi*y9;
    dy_deta = dN1_deta*x1 + dN2_deta*x2 + dN3_deta*x3 + dN4_deta*x4 + ...
    dN5_deta*x5 + dN6_deta*x6 + dN7_deta*x7 + dN8_deta*x8 + dN9_deta*x9;
    Je = [ dx_dxi dx_d eta ; dy_dxi dy_d eta ];
    j=det(Je);
    Jeinv= (1/j)*[ dy_deta -dx_d eta ; -dy_dxi dx_d_dxi ];

    dN1_dx_vect = [dN1_dxi dN1_deta]*Jeinv;
    B1 = dN1_dx_vect;
    dN2_dx_vect = [dN2_dxi dN2_deta]*Jeinv;
    B2 = dN2_dx_vect;
    dN3_dx_vect = [dN3_dxi dN3_deta]*Jeinv;
    B3 = dN3_dx_vect;
    dN4_dx_vect = [dN4_dxi dN4_deta]*Jeinv;
    B4 = dN4_dx_vect;
    dN5_dx_vect = [dN5_dxi dN5_deta]*Jeinv;
    B5 = dN5_dx_vect;
    dN6_dx_vect = [dN6_dxi dN6_deta]*Jeinv;
    B6 = dN6_dx_vect;
    dN7_dx_vect = [dN7_dxi dN7_deta]*Jeinv;
\[ B7 = dN7_{dx\_vect}; \]
\[ dN8_{dx\_vect} = [dN8_{dx\_i} dN8_{dx\_eta}] J_{inv}; \]
\[ B8 = dN8_{dx\_vect}; \]
\[ dN9_{dx\_vect} = [dN9_{dx\_i} dN9_{dx\_eta}] J_{inv}; \]
\[ B9 = dN9_{dx\_vect}; \]
\[ B = [B1 B2 B3 B4 B5 B6 B7 B8 B9]; \]
\[ \varepsilon_{vol}(i) = B \cdot A_d'; \]
\[ \text{end} \]
\[ \varepsilon_{vol}(el,:) = \varepsilon_{vol}(el); \]
\[ \text{end} \% END FUNCTION \]

---

% code for 3x3 Gaussian Integration of tangent matrix

function \( dfthdth = dfthdth\_fluid\_int(\text{coords}, \text{perm}, \text{nel\_bulk}) \)

% Old call: \( dfthdth(:,,:) = dfthdth\_fluid\_int(\text{coords}(el,:),\text{perm}(el,:)); \)
\( dfthdth = \text{zeros}(4,4,\text{nel\_bulk}); \)

for \( el = 1:\text{nel\_bulk} \)
    \( \text{coordsx} = \text{coords}(el,:); \)
    \( \text{perm}_e1 = \text{perm}(el,:); \)
    \% nodal coordinates of elements in local node numbering
    \( x1 = \text{coordsx}(1); \quad y1 = \text{coordsx}(2); \)
    \( x2 = \text{coordsx}(3); \quad y2 = \text{coordsx}(4); \)
    \( x3 = \text{coordsx}(5); \quad y3 = \text{coordsx}(6); \)
    \( x4 = \text{coordsx}(7); \quad y4 = \text{coordsx}(8); \)
    \( x5 = \text{coordsx}(9); \quad y5 = \text{coordsx}(10); \)
    \( x6 = \text{coordsx}(11); \quad y6 = \text{coordsx}(12); \)
    \( x7 = \text{coordsx}(13); \quad y7 = \text{coordsx}(14); \)
    \( x8 = \text{coordsx}(15); \quad y8 = \text{coordsx}(16); \)
    \( x9 = \text{coordsx}(17); \quad y9 = \text{coordsx}(18); \)

    \% initialize tangent matrix
    \( dfthdth_e = \text{zeros}(4,4); \)

end

\% set Gauss point coordinates in xi,eta space
\( \text{const}_1 = \sqrt{\frac{3}{5}}; \)
\( \text{const}_2 = \frac{25}{81}; \)
\( \text{const}_3 = \frac{40}{81}; \)
\( \text{const}_4 = \frac{64}{81}; \)
\( xi\_vec(1,:) = [-\text{const}_1 \ -\text{const}_1]; \)
\( xi\_vec(2,:) = [\text{const}_1 \ -\text{const}_1]; \)
\( xi\_vec(3,:) = [\text{const}_1 \ \text{const}_1]; \)
\( xi\_vec(4,:) = [-\text{const}_1 \ \text{const}_1]; \)
\( xi\_vec(5,:) = [0 \ -\text{const}_1]; \)
\( xi\_vec(6,:) = [\text{const}_1 \ 0]; \)
\( xi\_vec(7,:) = [0 \ \text{const}_1]; \)
\( xi\_vec(8,:) = [-\text{const}_1 \ 0]; \)
\( xi\_vec(9,:) = [0 \ 0]; \)
\% loop through the 4 Gauss points
for i=1:9
    xi=xi_vec(i,1);
    eta=xi_vec(i,2);

\% derivatives of shape functions with respect to \( \xi \)
    dN1_dxi = 0.25*(2*xi-1)*eta*(eta-1);
    dN2_dxi = 0.25*(2*xi+1)*eta*(eta+1);
    dN3_dxi = 0.25*(2*xi-1)*eta*(eta+1);
    dN4_dxi = 0.25*(2*xi+1)*eta*(eta+1);
    dN5_dxi = 0.5*(-2*xi)*eta*(eta-1);
    dN6_dxi = 0.5*(2*xi+1)*(1-eta^2);
    dN7_dxi = 0.5*(-2*xi)*eta*(eta+1);
    dN8_dxi = 0.5*(2*xi-1)*(1-eta^2);
    dN9_dxi = (-2*xi)*(1-eta^2);

\% derivatives of shape functions with respect to \( \eta \)
    dN1_deta = 0.25*(2*eta-1)*xi*(xi-1);
    dN2_deta = 0.25*(2*eta+1)*xi*(xi+1);
    dN3_deta = 0.25*(2*eta-1)*xi*(xi+1);
    dN4_deta = 0.25*(2*eta+1)*xi*(xi+1);
    dN5_deta = 0.5*(2*eta-1)*(1-xi^2);
    dN6_deta = 0.5*xi*(xi+1)*(-2*eta);
    dN7_deta = 0.5*(1-xi^2)*(2*eta+1);
    dN8_deta = 0.5*xi*(xi-1)*(-2*eta);
    dN9_deta = (-2*eta)*(1-xi^2);

\% calculate jacobian, its determinant, and its inverse
    dx_dxi = dN1_dxi*x1 + dN2_dxi*x2 + dN3_dxi*x3 + dN4_dxi*x4 + ...
            dN5_dxi*x5 + dN6_dxi*x6 + dN7_dxi*x7 + dN8_dxi*x8 + dN9_dxi*x9;
    dx_deta = dN1_deta*x1 + dN2_deta*x2 + dN3_deta*x3 + dN4_deta*x4 + ...
            dN5_deta*x5 + dN6_deta*x6 + dN7_deta*x7 + dN8_deta*x8 + dN9_deta*x9;
    dy_dxi = dN1_dxi*y1 + dN2_dxi*y2 + dN3_dxi*y3 + dN4_dxi*y4 + ...
            dN5_dxi*y5 + dN6_dxi*y6 + dN7_dxi*y7 + dN8_dxi*y8 + dN9_dxi*y9;
    dy_deta = dN1_deta*y1 + dN2_deta*y2 + dN3_deta*y3 + dN4_deta*y4 + ...
            dN5_deta*y5 + dN6_deta*y6 + dN7_deta*y7 + dN8_deta*y8 + dN9_deta*y9;
    Je = [ dx_dxi dx_deta ; dy_dxi dy_deta ];
    j=det(Je);
    Jeinv= (1/j)*[ dy_deta -dx_deta ; -dy_dxi dx_dxi ];

\% derivatives of linear shape functions with respect to \( \xi \)
    dN1_dxi = -0.25*(1-eta);
    dN2_dxi = -dN1_dxi;
    dN3_dxi = 0.25*(1+eta);
    dN4_dxi = -dN3_dxi;

\% derivatives of linear shape functions with respect to \( \eta \)
    dN1_deta = -0.25*(1-xi);
\[dN_2 \delta = -0.25 \times (1 + \xi)\]
\[dN_3 \delta = -dN_2 \delta;\]
\[dN_4 \delta = -dN_1 \delta;\]
\[dN_1 \delta = \frac{1}{2} \times (1 + \xi) \times \text{Jeinv};\]
\[B_1 p = [dN_1 \delta, dN_1 \delta];\]
\[B_2 p = [dN_2 \delta, dN_2 \delta];\]
\[B_3 p = [dN_3 \delta, dN_3 \delta];\]
\[B_4 p = [dN_4 \delta, dN_4 \delta];\]
\[B_p = [B_1 p, B_2 p, B_3 p, B_4 p];\]
\[\text{dfthintd}_e = \text{dfthintd}_e \times \text{perm}(i) \times B_p^t \times B_p \times j \times \Omega(i);\]
\[\text{dfthintd}(::, :, el) = \text{dfthintd}_e;\]

% code for 3x3 Gaussian Integration of tangent matrix
% function dfthint = dfthdd_fluid_int(coords, d_el, perm, del_epsv, ...
% porosity, rhoF, grav, nel_bulk, nel_bulk_dof, ...
% nel_bulk_dof)
% Old call: dfthintd(:, :, el)=
% ...
% dfthdd_fluid_int(coords(el,:), d_el(el,(nel_bulk_dof+1):nel_bulk_dof), perm_el(el,:), ...% ...% del_epsv_el(el,:), porosity_el(el,:), rhoF, grav);
% dfthintd = zeros(4, 18, nel_bulk);
% for el = 1:nel_bulk
% coordsx = coords(el,:);
% theta = d_el(el,(nel_bulk_dof+1):nel_bulk_dof);
% perm_el = perm(el,:);
% del_epsv_el = del_epsv(el,:);
% porosity_el = porosity(el,:);
% % nodal coordinates of elements in local node numbering
% x1 = coordsx(1); y1 = coordsx(2);
% x2 = coordsx(3); y2 = coordsx(4);
% x3 = coordsx(5); y3 = coordsx(6);
% x4 = coordsx(7); y4 = coordsx(8);
% x5 = coordsx(9); y5 = coordsx(10);
% x6 = coordsx(11); y6 = coordsx(12);
% x7 = coordsx(13); y7 = coordsx(14);
% x8 = coordsx(15); y8 = coordsx(16);
% x9 = coordsx(17); y9 = coordsx(18);
% % initialize tangent matrix
dfthintd_e=zeros(4,18);
W=zeros(9,1);

% set Gauss point coordinates in xi, eta space
const1=sqrt(3/5);
const2=25/81;
const3=40/81;
const4=64/81;
xi_vect(1,:)=[const1 const1];
xi_vect(2,:)=[const1 const1];
xi_vect(3,:)=[const1 const1];
xi_vect(4,:)=[const1 const1];
xi_vect(5,:)=[-const1 const1];
xi_vect(6,:)=[-const1 const1];
xi_vect(7,:)=[-const1 const1];
xi_vect(8,:)=[-const1 const1];
xi_vect(9,:)=[0 0];
W(1)=const2;
W(2)=const2;
W(3)=const2;
W(4)=const2;
W(5)=const3;
W(6)=const3;
W(7)=const3;
W(8)=const3;
W(9)=const4;

% loop through the 4 Gauss points
for i=1:9
  xi=xi_vect(i,1);
  eta=xi_vect(i,2);
  % derivatives of shape functions with respect to xi
  dN1_dxi = 0.25*(2*xi-1)*eta*(eta-1);
  dN2_dxi = 0.25*(2*xi+1)*eta*(eta-1);
  dN3_dxi = 0.25*(2*xi+1)*eta*(eta+1);
  dN4_dxi = 0.25*(2*xi-1)*eta*(eta+1);
  dN5_dxi = 0.5*(-2*xi)*eta*(eta-1);
  dN6_dxi = 0.5*(2*xi+1)*(1-eta^2);
  dN7_dxi = 0.5*(-2*xi)*eta*(eta+1);
  dN8_dxi = 0.5*(2*xi-1)*(1-eta^2);
  dN9_dxi = (-2*xi)*(1-eta^2);
  % derivatives of shape functions with respect to eta
  dN1_deta = 0.25*(2*eta-1)*xi*(xi-1);
  dN2_deta = 0.25*(2*eta+1)*xi*(xi+1);
  dN3_deta = 0.25*(2*eta+1)*xi*(xi+1);
  dN4_deta = 0.25*(2*eta-1)*xi*(xi-1);
  dN5_deta = 0.5*(2*eta-1)*(1-xi^2);
  dN6_deta = 0.5*xi*(xi+1)*(-2*eta);
  dN7_deta = 0.5*(1-xi^2)*(2*eta+1);
  dN8_deta = 0.5*xi*(xi-1)*(-2*eta);
  dN9_deta = (-2*eta)*(1-xi^2);
  % calculate jacobian, its determinant, and its inverse
  dx_dxi = dN1_dxi*x1 + dN2_dxi*x2 + dN3_dxi*x3 + dN4_dxi*x4 + ...
  dN5_dxi*x5 + dN6_dxi*x6 + dN7_dxi*x7 + dN8_dxi*x8 + dN9_dxi*x9;
dx_deta = dN1_deta*x1 + dN2_deta*x2 + dN3_deta*x3 + dN4_deta*x4 + ... 
   dN5_deta*x5 + dN6_deta*x6 + dN7_deta*x7 + dN8_deta*x8 + dN9_deta*x9;

dy_dxi = dN1_dxi*y1 + dN2_dxi*y2 + dN3_dxi*y3 + dN4_dxi*y4 + ... 
   dN5_dxi*y5 + dN6_dxi*y6 + dN7_dxi*y7 + dN8_dxi*y8 + dN9_dxi*y9;

dy_deta = dN1_deta*y1 + dN2_deta*y2 + dN3_deta*y3 + dN4_deta*y4 + ... 
   dN5_deta*y5 + dN6_deta*y6 + dN7_deta*y7 + dN8_deta*y8 + dN9_deta*y9;

Je = [ dx_dxi dx_deta ; dy_dxi dy_deta ];

j = det(Je);

Jeinv = (1/j)*[ dy_deta -dx_deta ; -dy_dxi dx_dxi ];

% for Btilde

B1 = dN1_dxi;
B2 = dN2_dxi;
B3 = dN3_dxi;
B4 = dN4_dxi;
B5 = dN5_dxi;
B6 = dN6_dxi;
B7 = dN7_dxi;
B8 = dN8_dxi;
B9 = dN9_dxi;

Btilde = [B1 B2 B3 B4 B5 B6 B7 B8 B9];

% derivatives of linear shape functions with respect to xi

dN1_dxi = -0.25*(1-eta);
dN2_dxi = -dN1_dxi;
dN3_dxi = 0.25*(1+eta);
dN4_dxi = -dN3_dxi;

% derivatives of linear shape functions with respect to eta

dN1_deta = -0.25*(1-xi);
dN2_deta = -dN1_deta;
dN3_deta = -dN2_deta;
dN4_deta = -dN1_deta;

B1p = [ dN1_dx_vect(1) ; dN1_dx_vect(2) ];
B2p = [ dN2_dx_vect(1) ; dN2_dx_vect(2) ];
\[ \text{dN3\_dx\_vect} = [\text{dN3\_dxi} \ \text{dN3\_deta}] \cdot \text{Jeinv}; \]
\[ \text{B3p} = [\text{dN3\_dx\_vect}(1) \ ; \ \text{dN3\_dx\_vect}(2)]; \]
\[ \text{dN4\_dx\_vect} = [\text{dN4\_dxi} \ \text{dN4\_deta}] \cdot \text{Jeinv}; \]
\[ \text{B4p} = [\text{dN4\_dx\_vect}(1) \ ; \ \text{dN4\_dx\_vect}(2)]; \]
\[ \text{Bp} = [\text{B1p} \ \text{B2p} \ \text{B3p} \ \text{B4p}]; \]
\[ \text{gravity} = [0 ; \text{grav}]; \]
\[ \text{dkdn} = \text{perm\_el}(i) \cdot \frac{(3 - \text{porosity\_el}(i)^2)}{(\text{porosity\_el}(i) \cdot (1 - \text{porosity\_el}(i)^2)}; \]
\[ \text{dncoeff} = \frac{(1 - \text{porosity\_el}(i))}{(1 + \text{del\_epsv\_el}(i))}; \]
\[ \text{dvdd} = -\text{dkdn} \cdot \text{dncoeff} \cdot (\text{Bp} \cdot \text{theta}' - \text{rhof} \cdot \text{gravity}) \cdot \text{Btilde}; \]
\[ \text{dfthintd\_e} = \text{dfthintd\_e} + \text{Bp}' \cdot \text{dvdd} \cdot j \cdot \text{W}(i); \]
\[ \text{dfthintd}(:, :, \text{el}) = \text{dfthintd\_e}; \]
\[ \% \text{code for 3x3 Gaussian Integration of consistent tangent matrix} \]
\[ \text{function dfdintdd} = \text{dfdintdd\_quad}(\text{coords}, \text{Dep}, \text{nel\_bulk}) \]
\[ \% \text{old call: dfdintdd}(:, :, \text{el}) = \text{dfdintdd\_quad}(\text{coords}(:, \text{el}), \text{Dep}(:, :, :, \text{el})); \]
\[ \text{dfdintdd} = \text{zeros}(18, 18, \text{nel\_bulk}); \]
\[ \% \text{nodal coordinates of elements in local node numbering} \]
\[ \text{x1} = \text{coordsx}(1); \ \text{y1} = \text{coordsx}(2); \]
\[ \text{x2} = \text{coordsx}(3); \ \text{y2} = \text{coordsx}(4); \]
\[ \text{x3} = \text{coordsx}(5); \ \text{y3} = \text{coordsx}(6); \]
\[ \text{x4} = \text{coordsx}(7); \ \text{y4} = \text{coordsx}(8); \]
\[ \text{x5} = \text{coordsx}(9); \ \text{y5} = \text{coordsx}(10); \]
\[ \text{x6} = \text{coordsx}(11); \ \text{y6} = \text{coordsx}(12); \]
\[ \text{x7} = \text{coordsx}(13); \ \text{y7} = \text{coordsx}(14); \]
\[ \text{x8} = \text{coordsx}(15); \ \text{y8} = \text{coordsx}(16); \]
\[ \text{x9} = \text{coordsx}(17); \ \text{y9} = \text{coordsx}(18); \]
\[ \% \text{initialize stiffness matrix} \]
\[ \text{dfdintdd\_e} = \text{zeros}(18, 18); \]
\[ \text{W} = \text{zeros}(9, 1); \]
\[ \% \text{set Gauss point coordinates in xi,eta space} \]
\[ \text{const1} = \text{sqrt}(3/5); \]
\[ \text{const2} = 25/81; \]
\[ \text{const3} = 40/81; \]
\[ \text{const4} = 64/81; \]
\[ \text{xi\_vect}(1,:) = [-\text{const1} \ -\text{const1}]; \]
% Calculate jacobian, its determinant, and its inverse
\[
\begin{bmatrix}
\text{dx}_d & \text{dy}_d
\end{bmatrix} =
\begin{bmatrix}
\text{dN1}_d & \text{dN2}_d & \text{dN3}_d & \text{dN4}_d & \ldots
\text{dN5}_d & \text{dN6}_d & \text{dN7}_d & \text{dN8}_d & \text{dN9}_d
\end{bmatrix}
\begin{bmatrix}
\text{dx}_i & \text{dy}_i
\end{bmatrix}
\]
\[
J = \begin{bmatrix}
\text{dx}_d & \text{dy}_d
\end{bmatrix}
\]
\[
J^{-1} = \frac{1}{\text{det}(J)} \begin{bmatrix}
\text{dy}_d & -\text{dx}_d
\end{bmatrix}
\]
\[
\begin{align*}
\mathbf{dN}_1 \mathbf{dx}_\mathbf{vec} &= [\mathbf{dN}_1 \mathbf{dx}_1 \mathbf{dN}_1 \mathbf{deta}] \mathbf{Jeinv}; \\
\mathbf{B}_1 &= \begin{bmatrix}
\mathbf{dN}_1 \mathbf{dx}_\mathbf{vec}(1) & 0 & 0 & \mathbf{dN}_1 \mathbf{dx}_\mathbf{vec}(2) & \mathbf{dN}_1 \mathbf{dx}_\mathbf{vec}(2) & \cdots \\
\end{bmatrix}; \\
\mathbf{dN}_2 \mathbf{dx}_\mathbf{vec} &= [\mathbf{dN}_2 \mathbf{dx}_1 \mathbf{dN}_2 \mathbf{deta}] \mathbf{Jeinv}; \\
\mathbf{B}_2 &= \begin{bmatrix}
\mathbf{dN}_2 \mathbf{dx}_\mathbf{vec}(1) & 0 & 0 & \mathbf{dN}_2 \mathbf{dx}_\mathbf{vec}(2) & \mathbf{dN}_2 \mathbf{dx}_\mathbf{vec}(2) & \cdots \\
\end{bmatrix}; \\
\mathbf{dN}_3 \mathbf{dx}_\mathbf{vec} &= [\mathbf{dN}_3 \mathbf{dx}_1 \mathbf{dN}_3 \mathbf{deta}] \mathbf{Jeinv}; \\
\mathbf{B}_3 &= \begin{bmatrix}
\mathbf{dN}_3 \mathbf{dx}_\mathbf{vec}(1) & 0 & 0 & \mathbf{dN}_3 \mathbf{dx}_\mathbf{vec}(2) & \mathbf{dN}_3 \mathbf{dx}_\mathbf{vec}(2) & \cdots \\
\end{bmatrix}; \\
\mathbf{dN}_4 \mathbf{dx}_\mathbf{vec} &= [\mathbf{dN}_4 \mathbf{dx}_1 \mathbf{dN}_4 \mathbf{deta}] \mathbf{Jeinv}; \\
\mathbf{B}_4 &= \begin{bmatrix}
\mathbf{dN}_4 \mathbf{dx}_\mathbf{vec}(1) & 0 & 0 & \mathbf{dN}_4 \mathbf{dx}_\mathbf{vec}(2) & \mathbf{dN}_4 \mathbf{dx}_\mathbf{vec}(2) & \cdots \\
\end{bmatrix}; \\
\mathbf{dN}_5 \mathbf{dx}_\mathbf{vec} &= [\mathbf{dN}_5 \mathbf{dx}_1 \mathbf{dN}_5 \mathbf{deta}] \mathbf{Jeinv}; \\
\mathbf{B}_5 &= \begin{bmatrix}
\mathbf{dN}_5 \mathbf{dx}_\mathbf{vec}(1) & 0 & 0 & \mathbf{dN}_5 \mathbf{dx}_\mathbf{vec}(2) & \mathbf{dN}_5 \mathbf{dx}_\mathbf{vec}(2) & \cdots \\
\end{bmatrix}; \\
\mathbf{dN}_6 \mathbf{dx}_\mathbf{vec} &= [\mathbf{dN}_6 \mathbf{dx}_1 \mathbf{dN}_6 \mathbf{deta}] \mathbf{Jeinv}; \\
\mathbf{B}_6 &= \begin{bmatrix}
\mathbf{dN}_6 \mathbf{dx}_\mathbf{vec}(1) & 0 & 0 & \mathbf{dN}_6 \mathbf{dx}_\mathbf{vec}(2) & \mathbf{dN}_6 \mathbf{dx}_\mathbf{vec}(2) & \cdots \\
\end{bmatrix}; \\
\mathbf{dN}_7 \mathbf{dx}_\mathbf{vec} &= [\mathbf{dN}_7 \mathbf{dx}_1 \mathbf{dN}_7 \mathbf{deta}] \mathbf{Jeinv}; \\
\mathbf{B}_7 &= \begin{bmatrix}
\mathbf{dN}_7 \mathbf{dx}_\mathbf{vec}(1) & 0 & 0 & \mathbf{dN}_7 \mathbf{dx}_\mathbf{vec}(2) & \mathbf{dN}_7 \mathbf{dx}_\mathbf{vec}(2) & \cdots \\
\end{bmatrix}; \\
\mathbf{dN}_8 \mathbf{dx}_\mathbf{vec} &= [\mathbf{dN}_8 \mathbf{dx}_1 \mathbf{dN}_8 \mathbf{deta}] \mathbf{Jeinv}; \\
\mathbf{B}_8 &= \begin{bmatrix}
\mathbf{dN}_8 \mathbf{dx}_\mathbf{vec}(1) & 0 & 0 & \mathbf{dN}_8 \mathbf{dx}_\mathbf{vec}(2) & \mathbf{dN}_8 \mathbf{dx}_\mathbf{vec}(2) & \cdots \\
\end{bmatrix}; \\
\mathbf{dN}_9 \mathbf{dx}_\mathbf{vec} &= [\mathbf{dN}_9 \mathbf{dx}_1 \mathbf{dN}_9 \mathbf{deta}] \mathbf{Jeinv}; \\
\mathbf{B}_9 &= \begin{bmatrix}
\mathbf{dN}_9 \mathbf{dx}_\mathbf{vec}(1) & 0 & 0 & \mathbf{dN}_9 \mathbf{dx}_\mathbf{vec}(2) & \mathbf{dN}_9 \mathbf{dx}_\mathbf{vec}(2) & \cdots \\
\end{bmatrix}; \\
\mathbf{B} &= [\mathbf{B}_1 \mathbf{B}_2 \mathbf{B}_3 \mathbf{B}_4 \mathbf{B}_5 \mathbf{B}_6 \mathbf{B}_7 \mathbf{B}_8 \mathbf{B}_9]; \\
\mathbf{dfdintdd}_e &= \mathbf{dfdintdd}_e + \mathbf{B}' \mathbf{Dep}_e(:,:,i) \mathbf{B} \mathbf{j} \mathbf{W}(i); \\
\mathbf{dfdintdd}(:,:,el) &= \mathbf{dfdintdd}_e; \\
end
\end{align*}
\]

% code for 3x3 Gaussian Integration of body force vector

function fdf = fdf_qdr_quad(coords, density, grav, nel, bulk)
% Old function call fdf(:,el)=fdf_qdr_quad(coords(el,:),density_el(el,:),grav);
% preallocate output
fdf = zeros(18, nel, bulk);
for el = 1:nel, bulk
    coordsel = coords(el,:);
    density_el = density(el,:);
% nodal coordinates of elements in local node numbering
x1 = coordsx(1); y1 = coordsx(2);
x2 = coordsx(3); y2 = coordsx(4);
x3 = coordsx(5); y3 = coordsx(6);
x4 = coordsx(7); y4 = coordsx(8);
x5 = coordsx(9); y5 = coordsx(10);
x6 = coordsx(11); y6 = coordsx(12);
x7 = coordsx(13); y7 = coordsx(14);
x8 = coordsx(15); y8 = coordsx(16);
x9 = coordsx(17); y9 = coordsx(18);

% initialize stiffness matrix
fdf_el = zeros(18,1);
W = zeros(9,1);

% set Gauss point coordinates in xi,eta space
const1=sqrt(3/5);
const2=25/81;
const3=40/81;
const4=64/81;
xi_vect(1,:)=[-const1 -const1];
xi_vect(2,:)=[const1 -const1];
xi_vect(3,:)=[const1 const1];
xi_vect(4,:)=[-const1 -const1];
xi_vect(5,:)=[0 -const1];
xi_vect(6,:)=[const1 0];
xi_vect(7,:)=[0 const1];
xi_vect(8,:)=[-const1 0];
xi_vect(9,:)=[0 0];
W(1)=const2;
W(2)=const2;
W(3)=const2;
W(4)=const2;
W(5)=const3;
W(6)=const3;
W(7)=const3;
W(8)=const3;
W(9)=const4;

% loop through the 4 Gauss points
for i=1:9
  xi=xi_vect(i,1);
  eta=xi_vect(i,2);

  % derivatives of shape functions with respect to xi
  dN1_dxi = 0.25*(2*xi-1)*eta*(eta-1);
dN2_dxi = 0.25*(2*xi+1)*eta*(eta-1);
dN3_dxi = 0.25*(2*xi+1)*eta*(eta+1);
dN4_dxi = 0.25*(2*xi-1)*eta*(eta+1);
dN5_dxi = 0.5*(-2*xi)*eta*(eta-1);
dN6_dxi = 0.5*(2*xi+1)*(1-eta^2);
dN7_dxi = 0.5*(-2*xi)*eta*(eta+1);
dN8_dxi = 0.5*(2*xi-1)*(1-eta^2);
dN9_dxi = (-2*xi)*(1-eta^2);

  % derivatives of shape functions with respect to eta
  dN1_deta = 0.25*(2*eta-1)*xi*(xi-1);
\[ dN_{2,\eta} = 0.25 \times (2 \times \eta - 1) \times \xi \times (\xi + 1); \]
\[ dN_{3,\eta} = 0.25 \times (2 \times \eta + 1) \times \xi \times (\xi + 1); \]
\[ dN_{4,\eta} = 0.25 \times (2 \times \eta + 1) \times \xi \times (\xi - 1); \]
\[ dN_{5,\eta} = 0.5 \times (2 \times \eta - 1) \times (1 - \xi^2); \]
\[ dN_{6,\eta} = 0.5 \times \xi \times (\xi + 1) \times (-2 \times \eta); \]
\[ dN_{7,\eta} = 0.5 \times (1 - \xi^2) \times (2 \times \eta + 1); \]
\[ dN_{8,\eta} = 0.5 \times \xi \times (\xi - 1) \times (-2 \times \eta); \]
\[ dN_{9,\eta} = (-2 \times \eta) \times (1 - \xi^2); \]

\[ dx_{\xi} = dN_{1,\xi} \times x_1 + dN_{2,\xi} \times x_2 + dN_{3,\xi} \times x_3 + dN_{4,\xi} \times x_4 + \ldots + dN_{5,\xi} \times x_5 + dN_{6,\xi} \times x_6 + dN_{7,\xi} \times x_7 + dN_{8,\xi} \times x_8 + dN_{9,\xi} \times x_9; \]
\[ dx_{\eta} = dN_{1,\eta} \times x_1 + dN_{2,\eta} \times x_2 + dN_{3,\eta} \times x_3 + dN_{4,\eta} \times x_4 + \ldots + dN_{5,\eta} \times y_5 + dN_{6,\eta} \times y_6 + dN_{7,\eta} \times y_7 + dN_{8,\eta} \times y_8 + dN_{9,\eta} \times y_9; \]
\[ dy_{\xi} = dN_{1,\xi} \times y_1 + dN_{2,\xi} \times y_2 + dN_{3,\xi} \times y_3 + dN_{4,\xi} \times y_4 + \ldots + dN_{5,\xi} \times y_5 + dN_{6,\xi} \times y_6 + dN_{7,\xi} \times y_7 + dN_{8,\xi} \times y_8 + dN_{9,\xi} \times y_9; \]
\[ dy_{\eta} = dN_{1,\eta} \times y_1 + dN_{2,\eta} \times y_2 + dN_{3,\eta} \times y_3 + dN_{4,\eta} \times y_4 + \ldots + dN_{5,\eta} \times y_5 + dN_{6,\eta} \times y_6 + dN_{7,\eta} \times y_7 + dN_{8,\eta} \times y_8 + dN_{9,\eta} \times y_9; \]
\[ J_e = \begin{bmatrix} dx_{\xi} & dx_{\eta} \\ dy_{\xi} & dy_{\eta} \end{bmatrix}; \]
\[ j = \text{det}(J_e); \]

\% shape functions
\[ N_{1} = 0.25 \times \xi \times \eta \times (\xi - 1) \times (\eta - 1); \]
\[ N_{\text{matrix}1} = [ N_{1} \ 0 \ 0 \ N_{1} ]; \]
\[ N_{2} = 0.25 \times \xi \times \eta \times (\xi + 1) \times (\eta - 1); \]
\[ N_{\text{matrix}2} = [ N_{2} \ 0 \ 0 \ N_{2} ]; \]
\[ N_{3} = 0.25 \times \xi \times \eta \times (\xi + 1) \times (\eta + 1); \]
\[ N_{\text{matrix}3} = [ N_{3} \ 0 \ 0 \ N_{3} ]; \]
\[ N_{4} = 0.25 \times \xi \times \eta \times (\xi - 1) \times (\eta + 1); \]
\[ N_{\text{matrix}4} = [ N_{4} \ 0 \ 0 \ N_{4} ]; \]
\[ N_{5} = 0.5 \times \eta \times (1 - \xi^2) \times (\eta - 1); \]
\[ N_{\text{matrix}5} = [ N_{5} \ 0 \ 0 \ N_{5} ]; \]
\[ N_{6} = 0.5 \times \xi \times (1 - \eta^2) \times (\xi + 1); \]
\[ N_{\text{matrix}6} = [ N_{6} \ 0 \ 0 \ N_{6} ]; \]
\[ N_{7} = 0.5 \times \eta \times (1 - \xi^2) \times (\eta + 1); \]
\[ N_{\text{matrix}7} = [ N_{7} \ 0 \ 0 \ N_{7} ]; \]
\[ N_{8} = 0.5 \times \xi \times (1 - \eta^2) \times (\xi - 1); \]
\[ N_{\text{matrix}8} = [ N_{8} \ 0 \ 0 \ N_{8} ]; \]
\[ N_{9} = (1 - \eta^2) \times (1 - \xi^2); \]
\[ N_{\text{matrix}9} = [ N_{9} \ 0 \ 0 \ N_{9} ]; \]
\[ \text{Nmatrix} = [ \text{Nmatrix}1 \ \text{Nmatrix}2 \ \text{Nmatrix}3 \ \text{Nmatrix}4 \ \text{Nmatrix}5 \ \text{Nmatrix}6 \ \ldots \ \text{Nmatrix}7 \ \text{Nmatrix}8 \ \text{Nmatrix}9 ]; \]
\[ g = [0; -\text{grav}]; \]
\[ \text{fdf}_{el} = \text{fdf}_{el} + \text{density}_{el}(i) \times \text{Nmatrix}' \times g \times j \times W(i); \]
\[ \text{end} \]
\[ \text{fdf}(; , el) = \text{fdf}_{el}; \]
\[ \text{end} \]
\[ \text{end} \]
end
classdef CSLP
    % CSLP - static class of CSE looped functions (parallel compatible)
    % Created on 10/31/2012
    % Updated
    % datestr(clock)=03-Nov-2012 14:30:20
methods (Static)
    % Code for cse nodal force vector
    function F_cse = cse_element_force(coords, Kcse, d_el, state_n, cse_para, flag, ...
            force_flag, n_ip, n_isvs, nel_bulk, nel_cse)
        % F_cse_el(:,el) = cse_element_force(coords(el,:), Kcse, ...
        d_el(el,:), state_n(:,el-nel_bulk), cse_para, ...
        % flag, force_flag, n_cse_ip, ...
        n_cse_isv);
        % size(F_cse_el) = <12 x 6>
        nel = nel_bulk + nel_cse;
        F_cse = zeros(12, nel);
        % CSE_Els is input vector of CSE ID's (e.g. [5 6 7 8])
        for el = (nel_bulk+1):nel
            coordsx = coords(el,:);
            d_elx = d_el(el,:);
            state_n_el = state_n(:,el-nel_bulk);
            % define nodal coordinates of cse in local node numbering
            x1 = coordsx(1); y1 = coordsx(2);
            x2 = coordsx(3); y2 = coordsx(4);
            x3 = coordsx(5); y3 = coordsx(6);
            x4 = coordsx(7); y4 = coordsx(8);
            x5 = coordsx(9); y5 = coordsx(10);
            x6 = coordsx(11); y6 = coordsx(12);
            xI = (x1+x5)/2;
            xII = (x2+x4)/2;
            xIII = (x3+x6)/2;
            yI = (y1+y5)/2;
            yII = (y2+y4)/2;
            yIII = (y3+y6)/2;
            % initialize nodal force array of cse
            F_cse_el = zeros(12,1);
            state = zeros(n_ip, n_isvs);
            % calculate the coordinate transformation Q
            [Q,normal] = Qf(coordsx);
            % calculate the local displacements of cse at ips
            d_cse_local_ip = d_cse_local_f(d_elx, Q, n_ip);
            % calculate the state variables of cse at ips
            if force_flag == 0
                state = state_n_el;
            else
                state = cse2D(Kcse, d_cse_local_ip, state_n_el, cse_para, flag, ...
                                n_ip, n_isvs);
            end
end
% set Gauss point coordinates in xi space
if n_ip == 1
    const = 0;
    xi_vect = [0];
    w_vect = [2];
elseif n_ip == 2
    const = 1/sqrt(3);
    xi_vect = [-const; const];
    w_vect = [1; 1];
elseif n_ip == 3
    const = sqrt(3/5);
    xi_vect = [-const; 0; const];
    w_vect = [(5/9); (8/9); (5/9)];
end

% set trapezoidal rule point coordinates in xi space
% const = 1;
% xi_vect = [-const; const];

% loop through the Gauss points
for i = 1:n_ip
    xi = xi_vect(i);

    % shape functions at Gauss point i
    % Left node
    N1 = 1/2*(xi^2 - xi);
    N5 = N1;

    % Right node
    N2 = 1/2*(xi^2 + xi);
    N4 = N2;

    % Center Node
    N3 = (1 - xi^2);
    N6 = N3;

    N_cse = ...
    [-N1 0 -N2 0 -N3 0 N4 0 N5 0 N6 0; 0 -N1 0 -N2 0 -N3 0 N4 0 N5 0 N6];

    % calculate the elastoplastic consistent tangent of cse
    %K_EP = cse.K_EP_2D_f(state(i,:), Kcse, cse_para);

    % this should be tilde(T).(n+1), not using K_EP
    %B = N_cse'*Q'*K_EP*d_cse_local_ip(i,:);
    Tlocal = [state(i,1) state(i,2)];
    B = N_cse' * Q' * Tlocal' * w_vect(i);

    % derivatives of shape functions with respect to xi
    dNI_dxi = xi-0.5;
    dNII_dxi = xi+0.5;
    dNIII_dxi = -2*xi;
    dx_dxi = dNI_dxi*xI + dNII_dxi*xII + dNIII_dxi*xIII;
    dy_dxi = dNI_dxi*yI + dNII_dxi*yII + dNIII_dxi*yIII;
    A = sqrt((dx_dxi)^2 + (dy_dxi)^2);
% calculate nodal force vector of cse
F_cse_el = F_cse_el + A*B;
end

% Write output
F_cse(:,el) = F_cse_el;
end
end

% Code for cse stiffness matrix
function k_cse = ...
cse_element_stiffness(coords,Kcse,d_el,state_n,cse_para,flag,n_ip,n_isvs, ... nel_bulk, nel_cse)
  % tangents
  % dfdcsintdd(:,el) = cse_element_stiffness(coords(el,:), Kcse, ...
  %     d_el(el,:), ...]
  %     state_n(:,el-nel_bulk), cse_para, ...
  %     flag, ...]
  % updated using cse_element_stiffness.m 03.09.2012
  % added elseif for 3 gauss points
  % initialize cse stiffness matrix = <12 x 12>
  nel = nel_bulk + nel_cse;
  k_cse = zeros(12, 12, nel);
  for el = (nel_bulk+1):nel;
    coordsx = coords(el,:);
    d_elx = d_el(el,:);
    state_n_el = state_n(:,el-nel_bulk);
    % define nodal coordinates of cse in local node numbering
    x1 = coordsx(1); y1 = coordsx(2);
    x2 = coordsx(3); y2 = coordsx(4);
    x3 = coordsx(5); y3 = coordsx(6);
    x4 = coordsx(7); y4 = coordsx(8);
    x5 = coordsx(9); y5 = coordsx(10);
    x6 = coordsx(11); y6 = coordsx(12);
    xI = (x1+x5)/2;
    xII = (x2+x4)/2;
    xIII = (x3+x6)/2;
    yI = (y1+y5)/2;
    yII = (y2+y4)/2;
    yIII = (y3+y6)/2;
    % initialize cse stiffness matrix
    k_cse_el = zeros(12, 12);
    % calculate the coordinate transformation Q
    [Q,normal] = Q_f(coordsx);
    % calculate the local displacements of cse
    d_cse_local_ip = d_cse_local_f(d_elx, Q, n_ip);
    % calculate the state variables of cse
    if flag == 0
      K_EP = Kcse;
      state = state_n_el;
    else
state = cse2D(Kcse, d_cse.local.ip, state_n_el, cse_para, flag, n_ip, ...
   n_svs);

end

% set Gauss point coordinates in xi space
if n_ip == 1
    const = 0;
    xi_vect = [0];
    w_vect = [2];
elseif n_ip == 2
    const = 1/sqrt(3);
    xi_vect = [-const; const];
    w_vect = [ 1 ; 1 ];
elseif n_ip == 3
    const = sqrt(3/5);
    xi_vect = [-const; 0; const];
    w_vect = [ (5/9); (8/9); (5/9)];
end

% set trapezoidal rule coordinates in xi space
% const = 1;
% xi_vect = [-const ; const];

% loop through the Gauss points
for i = 1:n_ip
    xi = xi_vect(i);

    % derivatives of shape functions
    % N1 = (1 - xi)/2;
    % N2 = (1 + xi)/2;
    % N3 = N2;
    % N4 = N1;
    % N_cse = [-N1 0 -N2 0 N3 0 N4 0; 0 -N1 0 -N2 0 N3 0 N4];
    % shape functions at Gauss point i
    % Left node
    N1 = 1/2*(xi^2 - xi);
    N5 = N1;
    % Right node
    N2 = 1/2*(xi^2 + xi);
    N4 = N2;
    % Center Node
    N3 = (1 - xi^2);
    N6 = N3;

    N_cse = ...
    [-N1 0 -N2 0 -N3 0 N4 0 N5 0 N6 0;
     0 -N1 0 -N2 0 -N3 0 N4 0 N5 0 N6];

    % calculate the elastoplastic consistent tangent of cse
    if flag == 1
        K_EP = cse.K_EP_2D_f(state(i,:), Kcse, cse_para);
    end

    B = N_cse' * Q' * K_EP * Q * N_cse * w_vect(i);

    % derivatives of shape functions with respect to xi
dNI_dxi = xi-0.5;
dNII_dxi = xi+0.5;
dNIII_dxi = -2*xi;
dx_dxi = dNI_dxi*xI + dNII_dxi*xII + dNIII_dxi*xIII;
dy_dxi = dNI_dxi*yI + dNII_dxi*yII + dNIII_dxi*yIII;
A = sqrt((dx_dxi)^2 + (dy_dxi)^2);

% calculate cse stiffness matrix
k_cse_el = k_cse_el + A*B;
end
k_cse(:,;el) = k_cse_el;
end

% Code for cse nodal force vector
function k_cse = cse_el_kdth(coords, n_ip, nel_bulk, nel_cse)
% kdth_cse_el(:,;el) = cse_el_kdth(coords(el,:), n_cse_ip);
nel = nel_bulk + nel_cse;
for el = (nel_bulk+1):nel
    coordsx = coords(el,:);
    % define nodal coordinates of cse in local node numbering
    x1 = coordsx(1);  y1 = coordsx(2);
    x2 = coordsx(3);  y2 = coordsx(4);
    x3 = coordsx(5);  y3 = coordsx(6);
    x4 = coordsx(7);  y4 = coordsx(8);
    x5 = coordsx(9);  y5 = coordsx(10);
    x6 = coordsx(11); y6 = coordsx(12);
    xI  = (x1+x5)/2;
    xII = (x2+x4)/2;
    xIII = (x3+x6)/2;
    yI  = (y1+y5)/2;
    yII = (y2+y4)/2;
    yIII = (y3+y6)/2;

    % initialize nodal force array of cse
    k_cse_el = zeros(12,4);

    % calculate the coordinate transformation Q
    [Q,normal] = Q_f(coordsx);

    % set Gauss point coordinates in xi space
    if n_ip == 1
        const = 0;
        xi_vector = [0];
        w_vector = [1];
    elseif n_ip == 2
        const = 1/sqrt(3);
        xi_vector = [ -const; const ];
        w_vector = [ (5/9); (8/9); (5/9) ];
    elseif n_ip == 3
        const = sqrt(3/5);
        xi_vector = [-const; 0; const ];
        w_vector = [ (5/9); (8/9); (5/9) ];
    end
    k_cse = k_cse_el.*w_vector;
end

% set trapezoidal rule point coordinates in xi space
const = 1;
xi_vect = [-const, const];

% loop through the Gauss points
for i = 1:n_ip
    xi = xi_vect(i);

% quadratic shape functions for displacement dof at Gauss point i
% Left node
N1 = 1/2*(xi^2 - xi);
N5 = N1;

% Right node
N2 = 1/2*(xi^2 + xi);
N4 = N2;

% Center Node
N3 = (1 - xi^2);
N6 = N3;

N_cse_u = ...
[-N1 0 -N2 0 -N3 0 N4 0 N5 0 N6 0; 0 -N1 0 -N2 0 -N3 0 N4 0 N5 0 N6];

% linear shape functions for pore pressure dof
% Left node
N1p = 1/2*(1 - xi);
N5p = N1p;

% Right node
N2p = 1/2*(1 + xi);
N4p = N2p;

N_cse_p = 0.5*[N1p N2p N4p N5p];

% form 12x4 matrix
B = N_cse_u' * (normal') * N_cse_p * w_vect(i);

% derivatives of shape functions with respect to xi
dNI_dxi = xi-0.5;
dNII_dxi = xi+0.5;
dNIII_dxi = -2*xi;
dx_dxi = dNI_dxi*xI + dNII_dxi*xII + dNIII_dxi*xIII;
dy_dxi = dNI_dxi*yI + dNII_dxi*yII + dNIII_dxi*yIII;
A = sqrt((dx_dxi)^2 + (dy_dxi)^2);

% calculate nodal force vector of cse
k_cse_el = k_cse_el + A*B;

end
k_cse(:,:,el) = k_cse_el;
end
% Code for cse nodal force vector
function f_cse = cse_el_fth1(coords, del, n_ip, perm_cse_params, nel_bulk, ...
   nel_cse)

% fth1_cse_el(:,el) = cse_el_fth1(coords(:,el), del(el,:), n_cse_ip, ...
%  perm_cse_params);
nel = nel_bulk + nel_cse;
f_cse = zeros(4,nel);

for el = (nel_bulk+1):nel
   coordsx = coords(el,:);
delx = del(el,:);
   
   % define nodal coordinates of cse in local node numbering
   x1 = coordsx(1); y1 = coordsx(2);
   x2 = coordsx(3); y2 = coordsx(4);
   x3 = coordsx(5); y3 = coordsx(6);
   x4 = coordsx(7); y4 = coordsx(8);
   x5 = coordsx(9); y5 = coordsx(10);
   x6 = coordsx(11); y6 = coordsx(12);

   XI = (x1+x5)/2;
   XII = (x2+x4)/2;
   XIII = (x3+x6)/2;
   yI = (y1+y5)/2;
   yII = (y2+y4)/2;
   yIII = (y3+y6)/2;

   xIjump = x5-x1;
   xIIjump = x4-x2;
   xIIIjump = x6-x3;
   yIjump = y5-y1;
   yIIjump = y4-y2;
   yIIIjump = y6-y3;

   % determine aperture magnitudes at each fictitious node
   xIvec_jump_mag = sqrt(xIjump^2 + yIjump^2);
   xIIvec_jump_mag = sqrt(xIIjump^2 + yIIjump^2);
   xIIIvec_jump_mag = sqrt(xIIIjump^2 + yIIIjump^2);

   % retrieve hydraulic conductivity of crack
   kcrack = perm_cse_params(1);
   small_num = perm_cse_params(4);
   small_num0 = perm_cse_params(7);

   % determine aperture direction vectors at each fictitious node
   xIvec_jump = [ xIjump ; yIjump ];
   if (xIvec_jump_mag > small_num)
      xIvec_jump_dir = xIvec_jump/xIvec_jump_mag;
   else
      xIvec_jump_dir = [ small_num ; small_num ];
   end
   xIIvec_jump = [ xIIjump ; yIIjump ];
   if (xIIvec_jump_mag > small_num)
      xIIvec_jump_dir = xIIvec_jump/xIIvec_jump_mag;
   else
      xIIvec_jump_dir = [ small_num ; small_num ];
   end
   xIIIvec_jump = [ xIIIjump ; yIIIjump ];
   if (xIIIvec_jump_mag > small_num)
      xIIIvec_jump_dir = xIIIvec_jump/xIIIvec_jump_mag;
   else
      xIIIvec_jump_dir = [ small_num ; small_num ];
   end
else
    xIIIvec_jump_dir = [ small_num; small_num ];
end

% initialize nodal force array of cse
f_cse_el = zeros(4,1);

% extract pore pressure dofs
theta = d_elx(19:22);

% calculate the coordinate transformation Q
[Q,normal] = Q_f(coordsx);

% xIvec_jump_dir
% xIIvec_jump_dir
% xIIIvec_jump_dir

% determine if aperture attempts to overlap itself at each fictitious node
xIvec_jump_dir_dot_normal = dot(xIvec_jump_dir,normal);
if (xIvec_jump_dir_dot_normal ≤ 0.0)
    xIvec_jump_mag = small_num0;
end
xIIvec_jump_dir_dot_normal = dot(xIIvec_jump_dir,normal);
if (xIIvec_jump_dir_dot_normal ≤ 0.0)
    xIIvec_jump_mag = small_num0;
end
xIIIvec_jump_dir_dot_normal = dot(xIIIvec_jump_dir,normal);
if (xIIIvec_jump_dir_dot_normal ≤ 0.0)
    xIIIvec_jump_mag = small_num0;
end

% calculate the local displacements of cse at ips
d_cse_local_ip = d_cse_local.f(d_elx, Q, n_ip);

% set Gauss point coordinates in xi space
if n_ip == 1
    const = 0;
    xi_vect = [0];
    w_vect = [2];
elseif n_ip == 2
    const = 1/sqrt(3);
    xi_vect = [ -const; const ];
    w_vect = [ 1; 1 ];
elseif n_ip == 3
    const = sqrt(3/5);
    xi_vect = [ -const; 0; const ];
    w_vect = [ (5/9); (8/9); (5/9) ];
end

% set trapezoidal rule point coordinates in xi space
% const = 1;
% xi_vect = [-const; const];

% loop through the Gauss points
for i = 1:n_ip
    xi = xi_vect(i);

    % aperture interpolation along crack
NI = 0.5*(xi-1)*xi;
NII = 0.5*(xi+1)*xi;
NIII = 1 -xiˆ2;
aperture = NI*xIvec_jump_mag + NII*xIIvec_jump_mag + NIII*xIIIvec_jump_mag;

% linear shape functions for pore pressure dof
% Left node
N1p = 1/2*(1 - xi);
N5p = N1p;

% Right node
N2p = 1/2*(1 + xi);
N4p = N2p;

% jump shape function matrix for pore pressures
N_cse_p = [N1p  N2p  N4p  N5p];

% pore pressure jump
pore_jump = N_cse_p * theta';

% Macauley bracket on un, only positive or zero for perm calc
d_cse_un = 0.5*(abs(d_cse_local_ip(i,2)) + d_cse_local_ip(i,2));

% calculate transverse hydraulic conductivity
% if (abs(d_cse_un) > small_num)
  % kn = kcrack/d_cse_un;
% else
  % kn = kcrack/small_num;
end
% if (abs(aperture) > small_num)
  % kn = kcrack/aperture;
else
  % kn = kcrack/small_num;
end
kn;

% form 12x4 matrix
% check equations for where this "2" comes from
B = N_cse_p' * 2 * (-kn) * pore_jump * w_vect(i);

% derivatives of shape functions with respect to xi
dNI_dxi = xi-0.5;
dNII_dxi = xi+0.5;
dNIII_dxi = -2*xi;
dx_dxi = dNI_dxi*xI + dNII_dxi*xII + dNIII_dxi*xIII;
dy_dxi = dNI_dxi*yI + dNII_dxi*yII + dNIII_dxi*yIII;
A = sqrt((dx_dxi)^2 + (dy_dxi)^2);

% calculate nodal force vector of cse
f_cse_el = f_cse_el + A*B;
end
f_cse(:,el) = f_cse_el;
end
end

% Code for cse nodal force vector
function f_cse = cse_el_fth2(coords, d_el, n_ip, perm_cse_params, nel bulk, ...
    nel cse)

  % fth2_cse_el(:,el) = cse_el_fth2(coords(el,:), d_el(el,:), n_cse_ip, ...
  perm_cse_params);

  nel = nel bulk + nel cse;
  f_cse = zeros(4, nel);

  for el = (nel bulk+1):nel
    coordsx = coords(el,:);
    d_elx = d_el(el,:);

    % define nodal coordinates of cse in local node numbering
    x1 = coordsx(1);  y1 = coordsx(2);
    x2 = coordsx(3);  y2 = coordsx(4);
    x3 = coordsx(5);  y3 = coordsx(6);
    x4 = coordsx(7);  y4 = coordsx(8);
    x5 = coordsx(9);  y5 = coordsx(10);
    x6 = coordsx(11); y6 = coordsx(12);
    x1 = (x1+x5)/2;
    x2 = (x2+x4)/2;
    xIII = (x3+x6)/2;
    y1 = (y1+y5)/2;
    yII = (y2+y4)/2;
    yIII = (y3+y6)/2;

    xIjump = x5-x1;
    xIIjump = x4-x2;
    xIIIjump = x6-x3;
    yIjump = y5-y1;
    yIIjump = y4-y2;
    yIIIjump = y6-y3;

    % determine aperture magnitudes at each fictitious node
    xIvec_jump_m = sqrt(xIjump^2 + yIjump^2);
    xIIvec_jump_m = sqrt(xIIjump^2 + yIIjump^2);
    xIIIvec_jump_m = sqrt(xIIIjump^2 + yIIIjump^2);

    % retrieve hydraulic conductivity of crack
    kcrack = perm_cse_params(1);
    fluid visc = perm_cse_params(3);
    small num = perm_cse_params(4);
    rhoF = perm_cse_params(5);
    grav = perm_cse_params(6);
    small num0 = perm_cse_params(7);

    % determine aperture direction vectors at each fictitious node
    xIvec_jump = [xIjump; yIjump];
    if (xIvec_jump_m > small num)
      xIvec_jump_dir = xIvec_jump/xIvec_jump_m;
    else
      xIvec_jump_dir = [small num; small num];
    end
    xIIvec_jump = [xIIjump; yIIjump];
    if (xIIvec_jump_m > small num)
      xIIvec_jump_dir = xIIvec_jump/xIIvec_jump_m;
    else
      xIIvec_jump_dir = [small num; small num];
    end
    xIIIvec_jump = [xIIIjump; yIIIjump];
    if (xIIIvec_jump_m > small num)
      xIIIvec_jump_dir = xIIIvec_jump/xIIIvec_jump_m;
    else
      xIIIvec_jump_dir = [small num; small num];
    end
end

xIIIvec_jump = [ xIIIjump; yIIIjump ];
if (xIIIvec_jump_mag > small_num)
xIIIvec_jump_dir = xIIIvec_jump/xIIIvec_jump_mag;
else
xIIIvec_jump_dir = [ small_num; small_num ];
end

% initialize nodal force array of cse
f_cse_el = zeros(4,1);

% extract pore pressure dofs
theta = d_elx(19:22);

% calculate the coordinate transformation Q
[Q,normal] = Q_f(coordsx);

% determine if aperture attempts to overlap itself at each fictitious node
xIvec_jump_dir_dot_normal = dot(xIvec_jump_dir,normal);
if (xIvec_jump_dir_dot_normal <= 0.0)
xIvec_jump_mag = small_num0;
end
xIIvec_jump_dir_dot_normal = dot(xIIvec_jump_dir,normal);
if (xIIvec_jump_dir_dot_normal <= 0.0)
xIIvec_jump_mag = small_num0;
end
xIIIvec_jump_dir_dot_normal = dot(xIIIvec_jump_dir,normal);
if (xIIIvec_jump_dir_dot_normal <= 0.0)
xIIIvec_jump_mag = small_num0;
end

% calculate the local displacements of cse at ips
d_cse_local_ip = d_cse_local.f(d_elx, Q, n_ip);

% set Gauss point coordinates in xi space
if n_ip == 1
    const = 0;
    xi_vect = [0];
    w_vect = [2];
elseif n_ip == 2
    const = 1/sqrt(3);
    xi_vect = [-const; const];
    w_vect = [1; 1];
elseif n_ip == 3
    const = sqrt(3/5);
    xi_vect = [-const; 0; const];
    w_vect = [5/9; 8/9; 5/9];
end

% set trapezoidal rule point coordinates in xi space
% const = 1;
% xi_vect = [-const; const];

% loop through the Gauss points
for i = 1:n_ip
    xi = xi_vect(i);
    % aperture interpolation along crack
\[ N_{I} = 0.5 \times (x_{i-1}) \times x_{i}; \]
\[ N_{II} = 0.5 \times (x_{i+1}) \times x_{i}; \]
\[ N_{III} = 1 \times x_{i}^{2}; \]
aperture = \( N_{I} \times x_{Ivec} \times \text{jump mag} + N_{II} \times x_{IIvec} \times \text{jump mag} + N_{III} \times x_{IIIvec} \times \text{jump mag}; \]

% derivatives of shape functions with respect to \( x_{i} \)
\[ dN_{I} / dx_{i} = x_{i}-0.5; \]
\[ dN_{II} / dx_{i} = x_{i+0.5}; \]
\[ dN_{III} / dx_{i} = 2x_{i}; \]
d\( x / dx_{i} = dN_{I} / dx_{i} \times x_{I} + dN_{II} / dx_{i} \times x_{II} + dN_{III} / dx_{i} \times x_{III}; \]
\[ A = \sqrt{(d\( x / dx_{i}\))^2 + (dy / dx_{i})^2}; \]

% linear shape functions for pore pressure dof
% Left node
\[ N_{1p} = 1/2 \times (1 - x_{i}); \]
\[ N_{5p} = N_{1p}; \]

% Right node
\[ N_{2p} = 1/2 \times (1 + x_{i}); \]
\[ N_{4p} = N_{2p}; \]

% jump shape function matrix for pore pressures
\[ N_{cse} = 0.5 \times \begin{bmatrix} N_{1p} & N_{2p} & N_{4p} & N_{5p} \end{bmatrix}; \]

% Left node
\[ B_{1p} = -1/2; \]
\[ B_{5p} = B_{1p}; \]

% Right node
\[ B_{2p} = 1/2; \]
\[ B_{4p} = B_{2p}; \]

% jump shape function matrix derivative for pore pressures
\[ B_{cse} = 0.5 \times \begin{bmatrix} B_{1p} & B_{2p} & B_{4p} & B_{5p} \end{bmatrix} / A; \]

% Macauley bracket on \( u_n \), only positive or zero for perm calc
\[ d_{cse\_un} = 0.5 \times (\text{abs}(d_{cse\_local\_ip}(i,2)) + d_{cse\_local\_ip}(i,2)); \]

% calculate longitudinal hydraulic conductivity
% \[ k_{l} = \left(1/\left(4 \times \text{fluid-visc}\right) \right)^{3}; \]
% \[ k_{l} = 1e-4; \]
\[ k_{l} = \left(1/\left(4 \times \text{fluid-visc}\right) \right)^{3}; \]

% derivatives
\[ dpfds = B_{cse\_p} \times \text{theta}' ; \]
\[ dyds = dy / dx_{i} / A; \]

% longitudinal flow
\[ \text{vt} = -k_{l} \times (dpfds - \text{rhoF} \times \text{grav} \times dyds); \]
\[ \text{vt} = -k_{l} \times (dpfds + \text{rhoF} \times \text{grav} \times dyds); \]

% form 12x4 matrix
\[ B = B_{cse\_p}' \times \text{vt} \times w_{vect}(i); \]

% calculate nodal force vector of cse
\[ f_{cse\_el} = f_{cse\_el} + A \times B; \]
end

f_cse(:,el) = f_cse_el;
end

end

% Code for cse nodal force vector
function k_cse = cse_el_dfth1dth(coords, d_el, n_ip, perm_cse_params, nel_bulk, ...
  nel_cse)
  % dfth1dth(:,:,:) = cse_el_dfth1dth(coords(1,:), d_el(1,:), n_cse_ip, ...) %4x4
  nel = nel_bulk + nel_cse;
  k_cse = zeros(4,4,nel);
  for el = (nel_bulk+1):nel
    coordsx = coords(el,:);
    d_elx = d_el(el,:);
    % define nodal coordinates of cse in local node numbering
    x1 = coordsx(1); y1 = coordsx(2);
    x2 = coordsx(3); y2 = coordsx(4);
    x3 = coordsx(5); y3 = coordsx(6);
    x4 = coordsx(7); y4 = coordsx(8);
    x5 = coordsx(9); y5 = coordsx(10);
    x6 = coordsx(11); y6 = coordsx(12);
    xI = (x1+x5)/2;
    xII = (x2+x4)/2;
    xIII = (x3+x6)/2;
    yI = (y1+y5)/2;
    yII = (y2+y4)/2;
    yIII = (y3+y6)/2;
    xIjump = x5-x1;
    xIIjump = x4-x2;
    xIIIjump = x6-x3;
    yIjump = y5-y1;
    yIIjump = y4-y2;
    yIIIjump = y6-y3;
    xIvec_jump_mag = sqrt(xIjump^2 + yIjump^2);
    xIIvec_jump_mag = sqrt(xIIjump^2 + yIIjump^2);
    xIIIvec_jump_mag = sqrt(xIIIjump^2 + yIIIjump^2);
    % retrieve hydraulic conductivity of crack
    kcrack = perm_cse_params(1);
    small_num = perm_cse_params(4);
    small_num0 = perm_cse_params(7);
    xIvec_jump = [ xIjump ; yIjump ];
    if (xIvec_jump_mag > small_num)
      xIvec_jump_dir = xIvec_jump/xIvec_jump_mag;
else
  xIvec_jump_dir = [ small_num ; small_num ];
end
xIvec_jump = [ xIjump ; yIjump ];
if (xIvec_jump_mag > small_num)
  xIvec_jump_dir = xIvec_jump/xIvec_jump_mag;
else
  xIvec_jump_dir = [ small_num ; small_num ];
end
xIIvec_jump = [ xIIjump ; yIIjump ];
if (xIIvec_jump_mag > small_num)
  xIIvec_jump_dir = xIIvec_jump/xIIvec_jump_mag;
else
  xIIvec_jump_dir = [ small_num ; small_num ];
end
xIIIvec_jump = [ xIIIjump ; yIIIjump ];
if (xIIIvec_jump_mag > small_num)
  xIIIvec_jump_dir = xIIIvec_jump/xIIIvec_jump_mag;
else
  xIIIvec_jump_dir = [ small_num ; small_num ];
end
% initialize nodal force array of cse
k_cse_el = zeros(4,4);
% extract pore pressure dofs
theta = d_elx(19:22);
% calculate the coordinate transformation Q
[Q,normal] = Qf(coordsx);
xIvec_jump_dir.dot_normal = dot(xIvec_jump_dir,normal);
if (xIvec_jump_dir.dot_normal <= 0.0)
  xIvec_jump_mag = small_num0;
end
xIIvec_jump_dir.dot_normal = dot(xIIvec_jump_dir,normal);
if (xIIvec_jump_dir.dot_normal <= 0.0)
  xIIvec_jump_mag = small_num0;
end
xIIIvec_jump_dir.dot_normal = dot(xIIIvec_jump_dir,normal);
if (xIIIvec_jump_dir.dot_normal <= 0.0)
  xIIIvec_jump_mag = small_num0;
end
% calculate the local displacements of cse at ips
d_cse_local_ip = d_cse_local_f(d_elx, Q, n_ip);
% set Gauss point coordinates in xi space
if n_ip == 1
  const = 0;
  xi_vect = [0];
  w_vect = [2];
elseif n_ip == 2
  const = 1/sqrt(3);
  xi_vect = [-const; const];
  w_vect = [ 1; 1 ];
elseif n_ip == 3
  const = sqrt(3/5);
  xi_vect = [-const; 0; const];
  w_vect = [ (5/9); (8/9); (5/9) ];
end
% set trapezoidal rule point coordinates in xi space
% const = 1;
% xi_vect = [-const; const];

% loop through the Gauss points
for i = 1:n_ip
    xi = xi_vect(i);

    % aperture interpolation along crack
    NI = 0.5*(xi-1)*xi;
    NII = 0.5*(xi+1)*xi;
    NIII = 1-xi^2;
    aperture = NI*xIvec_jump_mag + NII*xIIvec_jump_mag + ...  
                 NIII*xIIIvec_jump_mag;

    % linear shape functions for pore pressure dof
    % Left node
    N1p = 1/2 *(1 - xi);
    N5p = N1p;

    % Right node
    N2p = 1/2 *(1 + xi);
    N4p = N2p;

    % jump shape function matrix for pore pressures
    N_cse_p = [-N1p  -N2p  N4p  N5p];

    % Macauley bracket on un, only positive or zero for perm calc
    d_cse_un = 0.5*(abs(d_cse_local_ip(i,2))+d_cse_local_ip(i,2));

    % calculate transverse hydraulic conductivity derivative
    % if (abs(d_cse_un) > small_num)
    %   kn = kcrack/(d_cse_un);
    % else
    %   kn = kcrack/(small_num);
    % end
    if (abs(aperture) > small_num)
        kn = kcrack/(aperture);
    else
        kn = kcrack/(small_num);
    end

    % form 12x4 matrix
    % check equations for where this "2" comes from
    B = N_cse_p' * 2 * (-kn) * N_cse_p * w_vect(i);

    % derivatives of shape functions with respect to xi
    dNI_dxi = xi-0.5;
    dNII_dxi = xi+0.5;
    dNIII_dxi = -2*xi;
    dx_dxi = dNI_dxi*xI + dNII_dxi*xII + dNIII_dxi*xIII;
    dy_dxi = dNI_dxi*yI + dNII_dxi*yII + dNIII_dxi*yIII;
    A = sqrt((dx_dxi)^2 + (dy_dxi)^2);

    % calculate nodal force vector of cse
    k_cse_el = k_cse_el + A*B;
end
k_cse(:,:,el) = k_cse_el;
end
% Code for cse nodal force vector
function k_cse = cse_el_dfth2dth(coords, d_el, n_ip, params, nel_bulk, nel_cse)
    % dfth2sedth(:, :, el) = cse_el_dfth2dth(coords(:, :), d_el(:, :), n_cse.ip, ...
    % perm_cse.params);
    % define nodal coordinates of cse in local node numbering
    nel = nel_bulk + nel_cse;
    k_cse = zeros(4,4,nel);
    for el = (nel_bulk+1):nel
        coordsx = coords(el,:);
        d_elx = d_el(el,:);
        % define nodal coordinates of cse in local node numbering
        x1 = coordsx(1); y1 = coordsx(2);
        x2 = coordsx(3); y2 = coordsx(4);
        x3 = coordsx(5); y3 = coordsx(6);
        x4 = coordsx(7); y4 = coordsx(8);
        x5 = coordsx(9); y5 = coordsx(10);
        x6 = coordsx(11); y6 = coordsx(12);
        xI = (x1+x5)/2;
        xII = (x2+x4)/2;
        xIII = (x3+x6)/2;
        yI = (y1+y5)/2;
        yII = (y2+y4)/2;
        yIII = (y3+y6)/2;
        xIjump = x5-x1;
        xIIjump = x4-x2;
        xIIIjump = x6-x3;
        yIjump = y5-y1;
        yIIjump = y4-y2;
        yIIIjump = y6-y3;
        xIvec_jump_mag = sqrt(xIjumpˆ2 + yIjumpˆ2);
        xIIvec_jump_mag = sqrt(xIIjumpˆ2 + yIIjumpˆ2);
        xIIIvec_jump_mag = sqrt(xIIIjumpˆ2 + yIIIjumpˆ2);
        % retrieve hydraulic conductivity of crack
        kcrack = params(1);
        fluid_visc = params(3);
        small_num = params(4);
        rhoF = params(5);
        grav = params(6);
        small_num0 = params(7);
        xIvec_jump = [ xIjump ; yIjump ];
        if (xIvec_jump_mag > small_num)
            xIvec_jump_dir = xIvec_jump/xIvec_jump_mag;
        else
            xIvec_jump_dir = [ small_num ; small_num ];
        end
        xIIvec_jump = [ xIIjump ; yIIjump ];
        if (xIIvec_jump_mag > small_num)
            xIIvec_jump_dir = xIIvec_jump/xIIvec_jump_mag;
        else
            xIIvec_jump_dir = [ small_num ; small_num ];
        end
        xIIIvec_jump = [ xIIIjump ; yIIIjump ];
        if (xIIIvec_jump_mag > small_num)
            xIIIvec_jump_dir = xIIIvec_jump/xIIIvec_jump_mag;
        else
            xIIIvec_jump_dir = [ small_num ; small_num ];
        end
    end
end

xIIIvec_jump = [ xIIIjump ; yIIIjump ];
if (xIIIvec_jump_mag > small_num)
    xIIIvec_jump_dir = xIIIvec_jump/xIIIvec_jump_mag;
else
    xIIIvec_jump_dir = [ small_num ; small_num ];
end

% initialize nodal force array of cse
k_cse_el = zeros(4,4);

% extract pore pressure dofs
theta = d_elx(19:22);

% calculate the coordinate transformation Q
[Q,normal] = Q_f(coordsx);

xIvec_jump_dir_dot_normal = dot(xIvec_jump_dir,normal);
if (xIvec_jump_dir_dot_normal <= 0.0)
    xIvec_jump_mag = small_num0;
end

xIIvec_jump_dir_dot_normal = dot(xIIvec_jump_dir,normal);
if (xIIvec_jump_dir_dot_normal <= 0.0)
    xIIvec_jump_mag = small_num0;
end

xIIIvec_jump_dir_dot_normal = dot(xIIIvec_jump_dir,normal);
if (xIIIvec_jump_dir_dot_normal <= 0.0)
    xIIIvec_jump_mag = small_num0;
end

% calculate the local displacements of cse at ips
d_cse_local_ip = d_cse_local.f(d_elx, Q, n_ip);

% set Gauss point coordinates in xi space
if n_ip == 1
    const = 0;
    x_i_vect = [0];
    w_vect = [2];
elseif n_ip == 2
    const = 1/sqrt(3);
    x_i_vect = [ -const; const];
    w_vect = [ 1 ; 1 ];
elseif n_ip == 3
    const = sqrt(3/5);
    x_i_vect = [ -const; 0; const];
    w_vect = [ (5/9); (8/9); (5/9) ];
end

% set trapezoidal rule point coordinates in xi space
% const = 1;
% x_i_vect = [ -const; const ];

% loop through the Gauss points
for i = 1:n_ip
    xi = x_i_vect(i);

    % aperture interpolation along crack
    NI = 0.5*(xi-1)*xi;
NII = 0.5*(xi+1)*xi;
NIII = 1-xi^2;
aperture = NII*xIvec_jump_mag + NII*xIvec_jump_mag + ...
    NIII*xIIIvec_jump_mag;

% derivatives of shape functions with respect to xi
dNIdx = xi-0.5;
dNIIdx = xi+0.5;
dNIIIdx = -2*xi;
dx dx = dNIdx*xI + dNIIdx*xII + dNIIIdx*xIII;
dy dx = dNIdx*yI + dNIIdx*yII + dNIIIdx*yIII;
A = sqrt((dx dx)^2 + (dy dx)^2);

% Left node
B1p = -1/2;
B5p = B1p;

% Right node
B2p = 1/2;
B4p = B2p;

% jump shape function matrix derivative for pore pressures
B_cse_p = 0.5*[B1p B2p B4p B5p]/A;

% Macauley bracket on un, only positive or zero for perm calc
dcse_un = 0.5*(abs(dcse_local_ip(i,2)) + dcse_local_ip(i,2));

% calculate longitudinal hydraulic conductivity
%kl = (1/(4*fluid_visc))*(dcse_un)^2;
%kl = 1e-4;
kl = (1/(4*fluid_visc))*(aperture)^2;

% form 12x4 matrix
B = B_cse_p' * (-kl) * B_cse_p * wvect(i);

% calculate nodal force vector of cse
k_cse_el = k_cse_el + A*B;
end

% Code for cse nodal force vector
function k_cse = cse_el_dfth1dd(coords, d_el, n_ip, perm_cse_params, nel_bulk, ...
    nel_cse)
    % dfth1csedd(:,el) = cse_el_dfth1dd(coords(el,:), d_el(el,:), n_cse_ip, ...
        perm_cse_params);
    nel = nel_bulk + nel_cse;
k_cse = zeros(4, 12, nel);
for el = (nel_bulk+1):nel
    coordsx = coords(el,:);
    d_elx = d_el(el,:);
    % define nodal coordinates of cse in local node numbering
    x1 = coordsx(1);   y1 = coordsx(2);
    x2 = coordsx(3);   y2 = coordsx(4);
    x3 = coordsx(5);   y3 = coordsx(6);
x4 = coordsx(7);  y4 = coordsx(8);
x5 = coordsx(9);  y5 = coordsx(10);
x6 = coordsx(11); y6 = coordsx(12);

xI = (x1+x5)/2;
xII = (x2+x4)/2;
xIII = (x3+x6)/2;
yI = (y1+y5)/2;
yII = (y2+y4)/2;
yIII = (y3+y6)/2;

xIjump = x5-x1;
xIIjump = x4-x2;
xIIIjump = x6-x3;
yIjump = y5-y1;
yIIjump = y4-y2;
yIIIjump = y6-y3;

xIvec_jump_mag = sqrt(xIjump^2 + yIjump^2);
xIIvec_jump_mag = sqrt(xIIjump^2 + yIIjump^2);
xIIIvec_jump_mag = sqrt(xIIIjump^2 + yIIIjump^2);

% retrieve hydraulic conductivity of crack
kcrack = perm_cse.params(1);
small_num = perm_cse.params(4);
small_num0 = perm_cse.params(7);

xIvec_jump = [ xIjump ; yIjump ];
if (xIvec_jump_mag > small_num)
   xIvec_jump_dir = xIvec_jump/xIvec_jump_mag;
else
   xIvec_jump_dir = [ small_num ; small_num ];
end

xIIvec_jump = [ xIIjump ; yIIjump ];
if (xIIvec_jump_mag > small_num)
   xIIvec_jump_dir = xIIvec_jump/xIIvec_jump_mag;
else
   xIIvec_jump_dir = [ small_num ; small_num ];
end

xIIIvec_jump = [ xIIIjump ; yIIIjump ];
if (xIIIvec_jump_mag > small_num)
   xIIIvec_jump_dir = xIIIvec_jump/xIIIvec_jump_mag;
else
   xIIIvec_jump_dir = [ small_num ; small_num ];
end

% initialize nodal force array of cse
k_cse_el = zeros(4,12);

% extract pore pressure dofs
theta = d_elx(19:22);

% calculate the coordinate transformation Q
[Q,normal] = Q_f(coordsx);

xIvec_jump_dir_dot_normal = dot(xIvec_jump_dir,normal);
if (xIvec_jump_dir_dot_normal < 0.0)
   xIvec_jump_mag = small_num0;
xIIvec_jump_dir dot normal = dot(xIIvec_jump_dir, normal);
if (xIIvec_jump_dir dot normal ≤ 0.0)
xIIvec_jump_mag = small_num0;
end
xIIvec_jump_dir dot normal = dot(xIIvec_jump_dir, normal);
if (xIIvec_jump_dir dot normal ≤ 0.0)
xIIvec_jump_mag = small_num0;
end

% calculate the local displacements of cse at ips
d_cse_local_ip = d_cse_local_f(d_elx, Q, n_ip);

% set Gauss point coordinates in xi space
if n_ip == 1
const = 0;
xi_vect = [0];
w_vect = [2];
elseif n_ip == 2
const = 1/sqrt(3);
xi_vect = [-const; const];
w_vect = [1; 1];
elseif n_ip == 3
const = sqrt(3/5);
xi_vect = [-const; 0; const];
w_vect = [(5/9); (8/9); (5/9)];
end

% set trapezoidal rule point coordinates in xi space
% const = 1;
% xi_vect = [-const ; const];

% loop through the Gauss points
for i = 1:n_ip
xi = xi_vect(i);

% aperture interpolation along crack
NI = 0.5*(xi-1)*xi;
NII = 0.5*(xi+1)*xi;
NIII = 1-xi^2;
aperture = NI*xIIvec_jump_mag + NII*xIIvec_jump_mag + ... 
   NIII*xIIIvec_jump_mag;

% linear shape functions for pore pressure dof
% Left node
N1p = 1/2*(1 - xi);
N5p = N1p;

% Right node
N2p = 1/2*(1 + xi);
N4p = N2p;

% jump shape function matrix for pore pressures
N_cse_p = [-N1p  -N2p N4p N5p];

% pore pressure jump
pore_jump = N_cse_p * theta';
% Macauley bracket on un, only positive or zero for perm calc
\[
d_{cse\_un} = 0.5 \times (\text{abs}(d_{cse\_local\_ip}(i,2)) + d_{cse\_local\_ip}(i,2));
\]

% calculate transverse hydraulic conductivity derivative
\[
\% \text{if} (\text{abs}(d_{cse\_un}) > \text{small\_num})
\% \quad dkndun = kcrack / (d_{cse\_un}^2);
\% \else
\% \quad dkndun = kcrack / (\text{small\_num});
\% \end
\]

\[
\% \text{if} (\text{abs}(\text{aperture}) > \text{small\_num})
\% \quad dkndun = kcrack / (\text{aperture}^2);
\% \else
\% \quad dkndun = kcrack / (\text{small\_num});
\% \end
\]

\[
\% \text{shape functions at Gauss point i}
\% \text{Left node}
N_1 = 1/2 \times (x_i^2 - x_i);
N_5 = N_1;
\]

\[
\% \text{Right node}
N_2 = 1/2 \times (x_i^2 + x_i);
N_4 = N_2;
\]

\[
\% \text{Center Node}
N_3 = (1 - x_i^2);
N_6 = N_3;
\]

\[
N_{cse} = \ldots
\begin{bmatrix}
-N_1 & 0 & -N_2 & 0 & -N_3 & 0 & N_4 & 0 & N_5 & 0 & N_6 & 0;
0 & -N_1 & 0 & -N_2 & 0 & -N_3 & 0 & N_4 & 0 & N_5 & 0 & N_6;
\end{bmatrix};
\]

% form 12x4 matrix
% check equations for where this "2" comes from
B = N_{cse\_p}' \times 2 \times (\text{pore\_jump} \times \text{dkndun}) \times \text{normal} \times N_{cse} \times \text{w\_vect}(i);

% derivatives of shape functions with respect to xi
\[
dNI_{\_dxi} = x_i - 0.5;
dNII_{\_dxi} = x_i + 0.5;
dNIII_{\_dxi} = -2 \times x_i;
d_{dxi} = dNI_{\_dxi} \times x_I + dNII_{\_dxi} \times x_{II} + dNIII_{\_dxi} \times x_{III};
dy_{\_dxi} = dNI_{\_dxi} \times y_I + dNII_{\_dxi} \times y_{II} + dNIII_{\_dxi} \times y_{III};
A = \text{sqrt}((dx_{\_dxi})^2 + (dy_{\_dxi})^2);
\]

% calculate nodal force vector of cse
\[
k_{cse\_el} = k_{cse\_el} + A \times B;
end
\]

\[
k_{cse}(:, :, el) = k_{cse\_el};
end
\]

% Code for cse nodal force vector
function k_{cse} = cse\_el\_dfth2dd(coords, d_{el}, n_{ip}, perm\_cse\_params, nel\_bulk, ...
nel\_cse)
\[
% dfth2csedd(:, :, el) = cse\_el\_dfth2dd(coords(el,:), d_{el}(el,:), n_{cse\_ip}, ...
\% perm\_cse\_params);
\]

\[
\]
nel = nel_bulk + nel_cse;
k_cse = zeros(4, 12, nel);
for el = (nel_bulk+1):nel
coordx = coords(el,:);
d_elx = d_el(el,:);

% define nodal coordinates of cse in local node numbering
x1 = coordx(1); y1 = coordx(2);
x2 = coordx(3); y2 = coordx(4);
x3 = coordx(5); y3 = coordx(6);
x4 = coordx(7); y4 = coordx(8);
x5 = coordx(9); y5 = coordx(10);
x6 = coordx(11); y6 = coordx(12);

xI = (x1+x5)/2;
xII = (x2+x4)/2;
xIII = (x3+x6)/2;
yI = (y1+y5)/2;
yII = (y2+y4)/2;
yIII = (y3+y6)/2;

xIjump = x5-x1;
xIIjump = x4-x2;
xIIIjump = x6-x3;
yIjump = y5-y1;
yIIjump = y4-y2;
yIIIjump = y6-y3;

xIvec_jump_mag = sqrt(xIjump^2 + yIjump^2);
xIIvec_jump_mag = sqrt(xIIjump^2 + yIIjump^2);
xIIIvec_jump_mag = sqrt(xIIIjump^2 + yIIIjump^2);

% retrieve hydraulic conductivity of crack
kcrack = perm_cse_params(1);
fluid_visc = perm_cse_params(3);
small_num = perm_cse_params(4);
 rhoF = perm_cse_params(5);
grav = perm_cse_params(6);
small_num0 = perm_cse_params(7);

xIvec_jump = [ xIjump ; yIjump ];
if (xIvec_jump_mag > small_num)
    xIvec_jump_dir = xIvec_jump/xIvec_jump_mag;
else
    xIvec_jump_dir = [ small_num ; small_num ];
end
xIIvec_jump = [ xIIjump ; yIIjump ];
if (xIIvec_jump_mag > small_num)
    xIIvec_jump_dir = xIIvec_jump/xIIvec_jump_mag;
else
    xIIvec_jump_dir = [ small_num ; small_num ];
end
xIIIvec_jump = [ xIIIjump ; yIIIjump ];
if (xIIIvec_jump_mag > small_num)
    xIIIvec_jump_dir = xIIIvec_jump/xIIIvec_jump_mag;
else
    xIIIvec_jump_dir = [ small_num ; small_num ];
end
% initialize nodal force array of cse
k_cse_el = zeros(4,12);

% extract pore pressure dofs
theta = d_elx(19:22);

% calculate the coordinate transformation \( Q \)
[Q,normal] = Q_f(coordsx);

xIvec_jump_dir_dot_normal = dot(xIvec_jump_dir,normal);
if (xIvec_jump_dir_dot_normal < 0.0)
xIvec_jump_mag = small_num0;
end
xIIvec_jump_dir_dot_normal = dot(xIIvec_jump_dir,normal);
if (xIIvec_jump_dir_dot_normal < 0.0)
xIIvec_jump_mag = small_num0;
end
xIIIvec_jump_dir_dot_normal = dot(xIIIvec_jump_dir,normal);
if (xIIIvec_jump_dir_dot_normal < 0.0)
xIIIvec_jump_mag = small_num0;
end

% calculate the local displacements of cse at ips
d_cse_local_ip = d_cse_local_f(d_elx, Q, n_ip);

% set Gauss point coordinates in xi space
if n_ip == 1
    const = 0;
    xi_vect = [0];
    w_vect = [2];
elseif n_ip == 2
    const = 1/sqrt(3);
    xi_vect = [-const; const];
    w_vect = [1 1];
elseif n_ip == 3
    const = sqrt(3/5);
    xi_vect = [-const; 0; const];
    w_vect = [(5/9); (8/9); (5/9)];
end

% set trapezoidal rule point coordinates in xi space
% const = 1;
% xi_vect = [-const; const];

% loop through the Gauss points
for i = 1:n_ip
    xi = xi_vect(i);

    % aperture interpolation along crack
    NI = 0.5*(xi-1)*xi;
    NII = 0.5*(xi+1)*xi;
    NIII = 1-xi^2;
    aperture = NI*xIvec_jump_mag + NII*xIIvec_jump_mag + ...
               NIII*xIIIvec_jump_mag;

    % derivatives of shape functions with respect to xi
    dNI_dxi = xi-0.5;
\[ dN_{II}dx_i = xi + 0.5; \]
\[ dN_{II}dx_i = -2*xi; \]
\[ dx_{\text{dxi}} = dN_{I}dx_i*xi + dN_{II}dx_i*xI + dN_{III}dx_i*xIII; \]
\[ dy_{\text{dxi}} = dN_{I}dx_i*yI + dN_{II}dx_i*yII + dN_{III}dx_i*yIII; \]
\[ A = \sqrt{(dx_{\text{dxi}})^2 + (dy_{\text{dxi}})^2}; \]

\% linear shape functions for pore pressure dof
\% Left node
\[ N_{1p} = 1/2*(1 - xi); \]
\[ N_{5p} = N_{1p}; \]

\% Right node
\[ N_{2p} = 1/2*(1 + xi); \]
\[ N_{4p} = N_{2p}; \]

\% jump shape function matrix for pore pressures
\[ N_{\text{cse p}} = 0.5*[N_{1p} N_{2p} N_{4p} N_{5p}]; \]

\% Left node
\[ B_{1p} = -1/2; \]
\[ B_{5p} = B_{1p}; \]

\% Right node
\[ B_{2p} = 1/2; \]
\[ B_{4p} = B_{2p}; \]

\% jump shape function matrix derivative for pore pressures
\[ B_{\text{cse p}} = 0.5*[B_{1p} B_{2p} B_{4p} B_{5p}]/A; \]

\% Macauley bracket on un, only positive or zero for perm calc
\[ d_{\text{cse un}} = 0.5*(abs(d_{\text{cse local ip(i,2)}) + d_{\text{cse local ip(i,2)})}); \]

\% calculate longitudinal hydraulic conductivity
\[ \text{dkldun} = (3/(4*fluid_visc)) * (d_{\text{cse un}})^2; \]
\[ \text{dkldun} = 0; \]
\[ dkldun = (3/(4*fluid_visc)) * (aperture)^2; \]

\% derivatives
\[ dpfds = B_{\text{cse p}} * \theta'; \]
\[ dyds = dy_{\text{dxi}}/A; \]

\% longitudinal flow
\[ dvtdun = -(dpfds - rhoF*grav*dyds) * dkldun; \]
\[ dvtdun = -(dpfds + rhoF*grav*dyds) * dkldun; \]

\% shape functions at Gauss point i
\% Left node
\[ N_{1} = 1/2*(xi^2 - xi); \]
\[ N_{5} = N_{1}; \]

\% Right node
\[ N_{2} = 1/2*(xi^2 + xi); \]
\[ N_{4} = N_{2}; \]

\% Center Node
\[ N_{3} = (1 - xi^2); \]
\[ N_{6} = N_{3}; \]
N_cse = ...

\[
\begin{bmatrix}
-1 & 0 & -N2 & 0 & -N3 & 0 & N4 & 0 & N5 & 0 & N6 & 0; \\
0 & -1 & 0 & -N2 & 0 & -N3 & 0 & N4 & 0 & N5 & 0 & N6;
\end{bmatrix}
\]

% form 12x4 matrix
B = B_cse * p' * dvtdun * normal * N_cse * w_vect(i);

% calculate nodal force vector of cse
k_cse = k_cse + A * B;
end

% Code for cse nodal force vector due to end flux
function Fcsedthext = cse_el_Sflux(coords, d_el, n_ip, perm_cse_params, Sflux, ...
    ratio, nel_bulk, nel_cse)

    % fthS_cse_el(:,el) = fth2_mult*cse_el_Sflux(coords(el,:), d_el(el,:), ...
    % n_cse_ip, perm_cse_params, Sflux(el,:));
    % fthS_cse_el(:,el) = fth2_mult*cse_el_Sflux(coords(el,:), d_el(el,:), ...
    % n_cse_ip, perm_cse_params, Sflux(el,:));
    nel = nel_bulk + nel_cse;
    Fcsedthext = zeros(22,nel);

    % f_cse = zeros(4,nel);
    for el = (nel_bulk+1):nel
        coordsx = coords(el,:);
        Sflux_el = Sflux(el,:);
        % define nodal coordinates of cse in local node numbering
        x1 = coordsx(1); y1 = coordsx(2);
        x2 = coordsx(3); y2 = coordsx(4);
        x3 = coordsx(5); y3 = coordsx(6);
        x4 = coordsx(7); y4 = coordsx(8);
        x5 = coordsx(9); y5 = coordsx(10);
        x6 = coordsx(11); y6 = coordsx(12);

        xI = (x1+x5)/2;
        xII = (x2+x4)/2;
        xIII = (x3+x6)/2;
        yI = (y1+y5)/2;
        yII = (y2+y4)/2;
        yIII = (y3+y6)/2;

        xIjump = x5-x1;
        xIIjump = x4-x2;
        xIIIjump = x6-x3;
        yIjump = y5-y1;
        yIIjump = y4-y2;
        yIIIjump = y6-y3;

        % determine aperture magnitudes at each fictitious node
        xIvec_jump_mag = sqrt(xIjump^2 + yIjump^2);
        xIIvec_jump_mag = sqrt(xIIjump^2 + yIIjump^2);
        xIIIvec_jump_mag = sqrt(xIIIjump^2 + yIIIjump^2);
% retrieve hydraulic conductivity of crack
kcrack = perm_cse_params(1);
fluid_visc = perm_cse_params(3);
small_num = perm_cse_params(4);
rhoF = perm_cse_params(5);
grav = perm_cse_params(6);
small_num0 = perm_cse_params(7);

% determine aperture direction vectors at each fictitious node
xIvec_jump = [ xIjump ; yIjump ];
if (xIvec_jump_mag > small_num)
    xIvec_jump_dir = xIvec_jump/xIvec_jump_mag;
else
    xIvec_jump_dir = [ small_num ; small_num ];
end
xIIvec_jump = [ xIIjump ; yIIjump ];
if (xIIvec_jump_mag > small_num)
    xIIvec_jump_dir = xIIvec_jump/xIIvec_jump_mag;
else
    xIIvec_jump_dir = [ small_num ; small_num ];
end
xIIIvec_jump = [ xIIIjump ; yIIIjump ];
if (xIIIvec_jump_mag > small_num)
    xIIIvec_jump_dir = xIIIvec_jump/xIIIvec_jump_mag;
else
    xIIIvec_jump_dir = [ small_num ; small_num ];
end

% initialize nodal force array of cse
f_cse_el = zeros(4,1);

% calculate the coordinate transformation Q
[Q,normal] = Qf(coordsx);

% determine if aperture attempts to overlap itself at each fictitious node
xIvec_jump_dir_dot_normal = dot(xIvec_jump_dir,normal);
if (xIvec_jump_dir_dot_normal ≤ 0.0)
    xIvec_jump_mag = small_num0;
end
xIIvec_jump_dir_dot_normal = dot(xIIvec_jump_dir,normal);
if (xIIvec_jump_dir_dot_normal ≤ 0.0)
    xIIvec_jump_mag = small_num0;
end
xIIIvec_jump_dir_dot_normal = dot(xIIIvec_jump_dir,normal);
if (xIIIvec_jump_dir_dot_normal ≤ 0.0)
    xIIIvec_jump_mag = small_num0;
end

% retrieve end fluxes
Sflux0 = Sflux_el(1);
SfluxL = Sflux_el(2);

% at 0 end of element
f_cse_el = f_cse_el + Sflux0*xIvec_jump_mag*[1 0 0 1]';

% at L end of element
f_cse_el = f_cse_el + SfluxL*xIIvec_jump_mag*[0 1 1 0]';
1 % Code for qbar
2 function epsp = epsp_f(Sig, qn, dlam, epsp0, fGf_I, fGf_II)
3   % initialize epsp
4   epsp = zeros(1,2);
5   B1 = (Sig(2) + abs(Sig(2)))/(2.*fGf_I);
6   TNA = (Sig(2) - abs(Sig(2)))/2.;
7   Tt_excess = abs(Sig(1)) + TNA*qn(3);
8   B3func = 0.5*(Tt_excess + abs(Tt_excess));
9   Shear_Q = Sig(1)*Sig(1) + (qn(2) - qn(1)*qn(3))*(qn(2) - qn(1)*qn(4));
10  DQDN = qn(4);
11  DQDT = Sig(1)/sqrt(Shear_Q);
12  hshear = B3func*abs(DQDT)/fGf_II;
13  hnormal = B1*DQDN;
14  % calculate epsp
15  epsp(1) = epsp0(1) + dlam*hshear;
16  epsp(2) = epsp0(2) + dlam*hnormal;

1 % Code for yield value
2 function Yield = Yield_f(Sig, qn)
3   % calculate the yield value
4   Yield = sqrt((Sig(1))^2 + (qn(2) - qn(1)*qn(3))^2) - (qn(2) - Sig(2)*qn(3));
5 end
function state = state_f(u, state_n, fE_t, fE_n, fGf_I, fGf_II, falpha_chi, ...
    falpha_phi, falpha_psi, fchi_p, fc_p, fphi_p, fpsi_p, fchi_r, fc_r, fphi_r, ...
    fTol1, fTol2, n_ip, n_isvs)

state = zeros(n_ip,n_isvs);

% loop through Gauss points
for r = 1:n_ip

    % Initialize the necessary vectors at each Gauss pt
    I_mat = zeros(4,4);
    I_m = zeros(2,2);
    D = zeros(6,6);
    du = zeros(1,2);
    dup = zeros(1,2);
    up = zeros(1,2);
    upo = zeros(1,2);
    epsp = zeros(1,2);
    epsp0 = zeros(1,2);
    Sigo = zeros(1,2);
    Sigtr = zeros(1,2);
    dSig = zeros(1,2);
    dq = zeros(1,4);
    qn = zeros(1,4);
    qo = zeros(1,4);
    R = zeros(1,6);
    CMAT = zeros(6,6);
    Rvec = zeros(1,6);
    Cvec = zeros(1,6);
    Rmod = zeros(1,6);
    normr = 1.0;

    %dlam = state_n(r,14);
    dlam = 0;

    % Create the elastic stiffness
    KEE = zeros(2,2);
    KEE(1,1) = fE_t;
    KEE(2,2) = fE_n;
    KEE_inv = inv(KEE);

    for i = 1:2
        du(i) = u(r,i) - state_n(r,i+2);
        up(i) = state_n(r,i+4);
        upo(i) = up(i);
        epsp(i) = state_n(r,i+6);
        epsp0(i) = epsp(i);
        Sigo(i) = state_n(r,i);
        I_m(i,i) = 1.0;
    end

    for i = 1:4
        qn(i) = state_n(r,i+8);
        qo(i) = qn(i);
        I_mat(i,i) = 1.0;
end

ue = u(r,:); - upo;

% Sigtr = Sigo + (KEE*du')' % trial traction
Sigtr = (KEE*ue.')'; % trial traction

% Check the yield function
Ftr = Yield_f(Sigtr, qn);
% set to elastic for debugging
% Ftr = -1;

if Ftr < 0.
  iselastic = 0.;
  Sig = Sigtr;
  state(r,13) = Ftr;
  F = Ftr;
  kk = 0.;
else
  iselastic = 1.;
  kk = 0.;
  Sig = Sigo; % T'0 = T.n
  %Sig = Sigtr; % T'0 = T'tr
  F = Yield_f(Sig, qn);
  if abs(F) > 1e-12
    F0 = F;
  else
    F0 = 1;
  end
end

KEE.du = (KEE*du')';

dQdTsig = dQdTsig.f(Sig, qn);
qbar = qbar.f(Sig, qn, epsp, fGf_I, fGf_II, falpha_ch, falpha_c, ...
            falpha_Chi, falpha_ps, fchi_p, fc_p, fphi_p, fpsi_p, fchi_r, fc_r, ...
            fphi_r);
KEE.dQdTsig = (KEE*dQdTsig')';

% Create the residual equations
for i = 1:2
  R(i) = Sig(i) - Sigo(i) - KEE.du(i) + dlam*KEE.dQdTsig(i);
end

for i = 1:4
  R(i+2) = qo(i) - qn(i) + dlam*qbar(i);
end

normr = norm(R); % check norm
if normr > 1e-10
  normr0 = normr;
else
  normr0 = 1;
end

%while (F > fTol_1 || normr > fTol_2) % Start the local iterations
while ((abs(F/F0) > fTol_1 || normr/normr0 > fTol_2) && (abs(F) > ...
fTol_1 || normr > fTol_2)) % Start the local iterations
dQdTsig = dQdTsig.f(Sig, qn);
qbar = qbar.f(Sig, qn, epsp, fGf_I, fGf_II, falpha_ch, falpha_c, ...
falphanphi, falphaschi, fchip, fcphi, fpsi, fchip, ..., fcphi, fchip);
KEE_dQdSig = (KEE*dQdSig')';

% Create the residual equations
for i = 1:2
    R(i) = Sig(i) - Sig(i) - KEE*du(i) + dlam*KEE_dQdSig(i);
end

for i = 1:4
    R(i+2) = qo(i) - qo(i) + dlam*qbar(i);
end

dQdSig2 = dQdSig2_f(Sig, qn);
dQdSigdq = dQdSigdq_f(Sig, qn);
dqbardSig = dqbardSig_f(Sig, qn, epsp, dlam, fGf_I, fGf_II, ...
    falpha_chil, falphaschi, fchip, ..., fchip);
dqbardq = dqbardq_f(Sig, qn, epsp, dlam, fGf_I, fGf_II, falpha_chil, ...
    fchip, ..., falphaschi, fchip, fchip, ..., fchip);
KEE_dQdSig2 = KEE*dQdSig2;
KEE_dQdSigdq = KEE*dQdSigdq;

% Create the matrix D
for i = 1:6
    for j = 1:6
        if (i <= 2 && j <= 2)
            D(i,j) = I_m(i,j) + dlam*KEE_dQdSig2(i,j);
        elseif (i <= 2 && j > 2)
            D(i,j) = dlam*KEE_dQdSigdq(i,j-2);
        elseif (i > 2 && j <= 2)
            D(i,j) = dlam*dqbardSig(i-2,j);
        elseif (i > 2 && j > 2)
            D(i,j) = -I_mat(i-2,j-2) + dlam*dqbardq(i-2,j-2);
        end
    end
end

D_Inv = inv(D);
dfdSig = dfdSig_f(Sig, qn);
dfdq = dfdq_f(Sig, qn);

for i = 1:6
    if i <= 2
        Rvec(i) = dfdSig(i);
        Cvec(i) = KEE_dQdSig(i);
    elseif i > 2
        Rvec(i) = dfdq(i-2);
        Cvec(i) = qbar(i-2);
    end
end

% Rvecprint = Rvec'
% Cvecprint = Cvec'
% Rprint = R'
tmpRVec = D_Inv*R.';
top = F - dot(Rvec,tmpRVec);
%top = F - Rvec*tmpVec
tmpCVec = D_Inv*Cvec.';
bott = dot(Rvec,tmpCVec);
%bott = Rvec*tmpVec
%bott = -dot(Rvec,tmpVec)
dlam2 = top/bott;
if dlam2 < 0
dlam2 = 0
end

for i = 1:6
    for j = 1:6
        if (i <= 2 && j <= 2)
            CMAT(i,j) = KEE_Inv(i,j);
        elseif (i <= 2 && j > 2)
            CMAT(i,j) = 0.;
        elseif (i > 2 && j <= 2)
            CMAT(i,j) = 0.;
        elseif (i > 2 && j > 2)
            CMAT(i,j) = -I_mat(i-2,j-2);
        end
    end
end

for i = 1:6
    if i <= 2
        Rmod(i) = KEE_dQdSig(i);
    elseif i > 2
        Rmod(i) = qbar(i-2);
    end
end
Rmod = dlam2*Rmod;
R2 = R + Rmod;
%X = D_Inv*R2.';
%Y = CMAT*X;
X = -D_Inv*R2.';

for i = 1:6
    if i <= 2
        %dup(i) = Y(i);
        dSig(i) = X(i);
    elseif i > 2
        %dq(i-2) = Y(i);
        dq(i-2) = X(i);
    end
end
% dSigprint=dSig
% dqprint=dq

% Update the local variables
up = up + dup
Sig = Sig + dSig;
qn = qn + dq;
dlam = dlam + dlam2;
if dlam < 0
dlamprint = dlam;
dlam = 0;
end

% Check for yield for next iteration
ue = u(r,:) - up;
Sig = (KEE*ue.');
F = Yield_f(Sig, qn);
dQdSig = dQdSig_f(Sig, qn);
qbar = qbar_f(Sig, qn, epsp, fGf_I, fGf_II, falpha_chi, falpha_c, ...
falpha_phi, falpha_psi, fchi_p, fc_p, fphi_p, fpsi_p, fchi_r, ...fc_r, fphi_r);
KEE_dQdSig = (KEE*dQdSig');
for i = 1:2
    R(i) = Sig(i) - Sigo(i) - KEE_du(i) + dlam*KEE_dQdSig(i);
end
for i = 1:4
    R(i+2) = qo(i) - qn(i) + dlam*qbar(i);
end
%update epsp
ebsp = epsp_f(Sig, qn, dlam, epsp0, fGf_I, fGf_II);
normr = norm(R); % Update the value of normr
kk = kk + 1;
if kk>20
    kk
localrelF = abs(F/F0)
localF = F
localrelnorm = normr/normr0
localnorm = normr
error('reached maximum number of local iterations')
break
end %end while iteration loop
%kk
%localrelF = abs(F/F0)
%localF = F
%localrelnorm = normr/normr0
%localnorm = normr
%update plastic jump displacement
up = upo + dlam*jumpq;
end
% update the state vector
state(r,1) = Sig(1);
state(r,2) = Sig(2);
state(r,3) = u(r,1);
state(r,4) = u(r,2);
state(r,5) = up(1);
state(r,6) = up(2);
state(r,7) = epsp(1);
state(r,8) = epsp(2);
state(r,9) = qn(1);
state(r,10) = qn(2);
state(r,11) = qn(3);
state(r,12) = qn(4);
state(r,13) = F;
state(r,14) = dlam;
state(r,15) = iplastic;
state(r,16) = normr;
state(r,17) = kk;

end %end Gauss pt loop

% Code to determine transformation Q
function [Q,normal] = Q_f(coordsx)
% Updated 04.11.2012
% Changed top node comments to fix error
% -Updated to accept 6 nodes from coords
% -Changed to new node numbering for 6 node CSE

% Number of nodes to convert from 4 to 6 noded CSE
% 4 node 6 node
% 1 1
% 2 2
% 3 4
% 4 5

% Still need update main part to rotate 6 nodes
% Define nodal coordinates of cse in local node numbering
Q = zeros(2,2);
normal = zeros(2,1);

% Bottom Edge
x1 = coordsx(1);
y1 = coordsx(2);  % Left node
x2 = coordsx(3);
y2 = coordsx(4);  % Right node
x3 = coordsx(5);
y3 = coordsx(6);  % Center Node

% Top Edge
x4 = coordsx(7);
y4 = coordsx(8);  % Right node
x5 = coordsx(9);
y5 = coordsx(10); % Left node
x6 = coordsx(11);
y6 = coordsx(12); % Center Node

small_num = 1e-4;

% Calculate the coordinate transformation Q
if (abs(x1-x2)<small_num) && (abs(x4-x5)<small_num) % 90 degree case
    Q = [0 1; -1 0];
    normal = [-1 0];
else
    if (abs(x1-x2)<small_num) && x4 != x5

```matlab
function [Q,normal] = Q_f(coordsx)
% Updated 04.11.2012
% Changed top node comments to fix error
% -Updated to accept 6 nodes from coords
% -Changed to new node numbering for 6 node CSE

% Number of nodes to convert from 4 to 6 noded CSE
% 4 node 6 node
% 1 1
% 2 2
% 3 4
% 4 5

% Still need update main part to rotate 6 nodes
% Define nodal coordinates of cse in local node numbering
Q = zeros(2,2);
normal = zeros(2,1);

% Bottom Edge
x1 = coordsx(1);
y1 = coordsx(2);  % Left node
x2 = coordsx(3);
y2 = coordsx(4);  % Right node
x3 = coordsx(5);
y3 = coordsx(6);  % Center Node

% Top Edge
x4 = coordsx(7);
y4 = coordsx(8);  % Right node
x5 = coordsx(9);
y5 = coordsx(10); % Left node
x6 = coordsx(11);
y6 = coordsx(12); % Center Node

small_num = 1e-4;

% Calculate the coordinate transformation Q
if (abs(x1-x2)<small_num) && (abs(x4-x5)<small_num) % 90 degree case
    Q = [0 1; -1 0];
    normal = [-1 0];
else
    if (abs(x1-x2)<small_num) && x4 != x5
```
Alpha_1 = pi/2;
Alpha_2 = atan((y4 - y5)/(x4 - x5));
elseif x1 \neq x2 \&\& (abs(x4-x5)<small_num)
    Alpha_1 = atan((y2 - y1)/(x2 - x1));
    Alpha_2 = pi/2;
else
    Alpha_1 = atan((y2 - y1)/(x2 - x1));
    Alpha_2 = atan((y4 - y5)/(x4 - x5));
end

% Alpha = mean of Alpha_1 and Alpha_2
Alpha = (Alpha_1 + Alpha_2)/2;
Q = [cos(Alpha) sin(Alpha); -sin(Alpha) cos(Alpha)];

end

% calculate the coordinate transformation Q
% if x1 == x2 \&\& x4 == x5 % 90 degree case
% Q = [0 1; -1 0];
% normal = [-sin(Alpha) cos(Alpha)];
% else
% if x1 == x2 \&\& x4 \neq x5
%    Alpha_1 = pi/2;
%    Alpha_2 = atan((y4 - y5)/(x4 - x5));
% elseif x1 \neq x2 \&\& x4 == x5
%    Alpha_1 = atan((y2 - y1)/(x2 - x1));
%    Alpha_2 = pi/2;
% else
%    Alpha_1 = atan((y2 - y1)/(x2 - x1));
%    Alpha_2 = atan((y4 - y5)/(x4 - x5));
% end

% if x2 == x1 \&\& x3 == x4 % 90 degree case
% Q = [0 1; -1 0];
% else
% if x2 == x1 \&\& x3 \neq x4
%    Alpha_1 = pi/2;
%    Alpha_2 = atan((y3 - y4)/(x3 - x4));
% elseif x2 \neq x1 \&\& x3 == x4
%    Alpha_1 = atan((y2 - y1)/(x2 - x1));
%    Alpha_2 = pi/2;
% else
%    Alpha_1 = atan((y2 - y1)/(x2 - x1));
%    Alpha_2 = atan((y3 - y4)/(x3 - x4));
% end

% Alpha = (Alpha_1 + Alpha_2)/2;
% Q = [cos(Alpha) sin(Alpha); -sin(Alpha) cos(Alpha)];
% end