pydfnworks Documentation

Release 2.0

Computational Earth Science (EES-16), LANL, LA-CC-17-027

March 09, 2017

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION

dfnWorks is a parallelized computational suite to generate three-dimensional discrete fracture networks (DFN) and simulate flow and transport. Developed at Los Alamos National Laboratory, it has been used to study flow and transport in fractured media at scales ranging from millimeters to kilometers. The networks are created and meshed using dfnGen, which combines FRAM (the feature rejection algorithm for meshing) methodology to stochastically generate three-dimensional DFNs with the LaGriT meshing toolbox to create a high-quality computational mesh representation. The representation produces a conforming Delaunay triangulation suitable for high performance computing finite volume solvers in an intrinsically parallel fashion. Flow through the network is simulated with dfnFlow, which utilizes the massively parallel subsurface flow and reactive transport finite volume code PFLOTRAN. A Lagrangian approach to simulating transport through the DFN is adopted within dfnTrans to determine pathlines and solute transport through the DFN. Applications of the dfnWorks suite include nuclear waste repository science, hydraulic fracturing and CO_2 sequestration.

To run a workflow using the dfnWorks suite, the pydfnworks package is highly recommended. pydfnworks calls various tools in the dfnWorks suite; its aim is to provide a seamless workflow for scientific applications of dfnWorks.

1.1 Citing dfnWorks

Hyman, J. D., Karra, S., Makedonska, N., Gable, C. W., Painter, S. L., & Viswanathan, H. S. (2015). dfnWorks: A discrete fracture network framework for modeling subsurface flow and transport. Computers & Geosciences, 84, 10-19.

BibTex:

```
  @article{hyman2015dfnworks,
    title={dfnWorks: A discrete fracture network framework
for modeling subsurface flow and transport},
    author={Hyman, Jeffrey D and Karra, Satish and Makedonska,
Nataliia and Gable, Carl W and Painter, Scott L
and Viswanathan, Hari S},
    journal={Computers \& Geosciences},
    volume={84},
    pages={10--19},
    year={2015},
    publisher={Elsevier}
  }
```

1.2 What's new in v2.0?

- New dfnGen C++ code which is much faster than the Mathematica dfnGen. This code has successfully generated networks with 350,000+ fractures.

- Increased functionality in the pydfnworks package for more streamlined workflow from dfnGen through visualization.

1.3 Where can one get dfnWorks?

dfnWorks 2.0 can be downloaded from https://github.com/dfnWorks/dfnWorks-Version2.0

v1.0 can be downloaded from https://github.com/dfnWorks/dfnWorks-Version1.0

1.4 Installation

Tools that you will need to run the dfnWorks work flow are described in this section. VisIt and ParaView, which enable visualization of desired quantities on the DFNs, are optional, but at least one of them is highly recommended for visualization. CMake is also optional but allows faster IO processing using C++.

1.4.1 Python

pydfnworks is supported on Python 2.7. The software authors recommend using the Anaconda 2.7 distribution of Python, available at https://www.continuum.io/. pydfnworks requires the `numpy` and `h5py` modules to be installed.

1.4.2 pydfnworks

The source for pydfnworks can be found in the dfnWorks suite, in the folder pydfnworks.

1.4.3 dfnGen

dfnGen primarily involves two steps:

1. FRAM - Create DFN: Using the fractured site characterization networks are constructed using the feature rejection algorithm for meshing
2. LaGriT - Mesh DFN: The LaGriT meshing tool box is used to create a conforming Delaunay triangulation of the network.

FRAM

FRAM (the feature rejection algorithm for meshing) is executed using the dfnGen C++ source code, contained in the dfnGen folder of the dfnWorks repository.

LaGriT

The *LaGriT* (http://lagrit.lanl.gov) meshing toolbox is used to create a high resolution computational mesh representation of the DFN in parallel. An algorithm for conforming Delaunay triangulation is implemented so that fracture intersections are coincident with triangle edges in the mesh and Voronoi control volumes are suitable for finite volume flow solvers such as FEHM and PFLOTRAN.

1.4.4 dfnFlow

You will need one of either PFLOTRAN or FEHM to solve for flow using the mesh files from LaGriT.

PFLOTRAN

PFLOTRAN (http://www.pflotran.org) is a massively parallel subsurface flow and reactive transport code. PFLOTRAN solves a system of partial differential equations for multiphase, multicomponent and multiscale reactive flow and transport in porous media. The code is designed to run on leadership-class supercomputers as well as workstations and laptops.

FEHM

FEHM (http://fehm.lanl.gov) is a subsurface multiphase flow code developed at Los Alamos National Laboratory.

1.4.5 dfnTrans

dfnTrans is a method for resolving solute transport using control volume flow solutions obtained from dfnFlow on the unstructured mesh generated using dfnGen. We adopt a Lagrangian approach and represent a non-reactive conservative solute as a collection of indivisible passive tracer particles.

1.4.6 CMake

CMake (https://cmake.org/) is an open-source, cross-platform family of tools designed to build, test and package software. It is needed to use C++ for processing files at a bottleneck IO step of dfnWorks. Using C+C++ for this file processing optional but can greatly increase the speed of dfnWorks for large fracture networks. Details on how to use C++ for file processing are in the scripts section of this documentation.

1.4.7 VisIt

VisIt is a parallel, open-source visualisation software. PFLOTRAN can output in `.xmf` and `.vtk` format. These can be imported in VisIt for visualization.

Instructions for downloading and installing VisIt can be found at https://wci.llnl.gov/codes/visit/download.html

1.4.8 Paraview

Paraview is a parallel, open-source visualisation software. PFLOTRAN can output in `.xmf` and `.vtk` format. These can be imported in Paraview for visualization.

Instructions for downloading and installing Paraview can be found at http://www.paraview.org

1.5 Using pydfnworks in your Python scripts

To access the functionality of pydfnworks, the user must include the following line at the top of any Python script

```
import pydfnworks
```

Before doing this, one needs to ensure that the pydfnworks directory is in the PYTHONPATH. This can be done by configuring `cshrc` or `bashrc` files. Alternatively, one can add the pydfnworks path using `sys.path.append()` in their driver script.

1.6 About this manual

This manual comprises of information on setting up inputs to dfnGen, dfnTrans and PFLOTRAN, as well as details on the pydfnworks module: *pydfnworks*. Finally, the manual contains a short tutorial with prepared examples that can be found in the `tests` directory of the dfnWorks repository, and a description of some applications of the dfnWorks suite.

1.7 Contributors

- Satish Karra
- Nataliia Makedonska
- Jeffrey Hyman
- Jeremy Harrod (now at Spectra Logic)
- Quan Bui (now at University of Maryland)
- Carl Gable
- Scott Painter (now at ORNL)
- Hari Viswanathan
- Nathaniel Knapp

1.8 Contact

For any questions about dfnWorks, please email dfnworks@lanl.gov.

1.9 Copyright information

CHAPTER

TWO

DFNGEN

2.1 Keywords

The following is an example input file with all keywords and explanation of each keyword.

```
//========================================================================
//Gereral Options & Fracture Network Parameters:
/*
stopCondition: 0 /* 0: stop once nPoly fractures are accepted (Defined
below) 1: stop once all family's p32 values are equal or greater than the
families target p32 values (defined in stochastic family sections) */

nPoly: 3 /* Used when stopCondition = 0 Total number of fractures you would
like to have in the domain you defined. The program will complete once you
have nPoly number of fractures, maxPoly number of polygon/fracture
rejections, rejPoly number of rejections in a row, or reach a specified
fracture cluster size if using stoppingParameter = -largestSize  */


outputAllRadii: 0  /* 0: Do not output all radii file.  1: Include file of
all raddii (acepted+rejected fractures) in output files.  */


domainSize: {1,1,1} /* Mandatory Parameter.  Creates a domain with dimension
x*y*z centered at the origin.*/

numOfLayers: 0    //number of layers


layers: {-500,0} {0,500}

/*  Layers need to be listed line by line Format: {minZ, maxZ}

    The first layer listed is layer 1, the second is layer 2, etc Stochastic
    families can be assigned to theses layers (see stochastic shape familiy
    section) */


h: 0.050 /* Minimum fracture length scale(meters) Any fracture with a
feature, such as and intersection, of less than h will be rejected. */


//========================================================================//
/* Fracture Network Parameters:
*/
```

```
tripleIntersections: 1 /* Options:     0: Off 1: On    */

printRejectReasons: 0 /* Useful in debugging, This option will print all
fracture rejection reasons as they occur.  0: disable 1: print all rejection
reasons to screen */

visualizationMode: 0 /* Options: 0 or 1 Used during meshing: 0: creates a
fine mesh, according to h parameter; 1: produce only first round of
triangulations. In this case no modeling of flow and transport is possible.
*/

seed: 92731535 //Seed for random generator.


domainSizeIncrease: {0,0,0} //temporary size increase for inserting fracture
centers outside domain //increases the entire width by this ammount. So,
{1,1,1} will increase //the domain by adding .5 to the +x, and subbtracting
.5 to the -x, etc


keepOnlyLargestCluster: 0 /* 0 = Keep any clusters which connects the
specified boundary faces in boundaryFaces option below

       1 = Keep only the largest cluster which connects the specified
       boundary faces in boundaryFaces option below */

ignoreBoundaryFaces: 1  /* 0 = use boundaryFaces option below

     1 = ignore boundaryFaces option and keep all clusters and will still
     remove fractures with no intersections               */


boundaryFaces: {1,1,0,0,0,0} /*  DFN will only keep clusters with
connections to domain boundaries which are set to 1:

    boundaryFaces[0] = +X domain boundary boundaryFaces[1] = -X domain
    boundary boundaryFaces[2] = +Y domain boundary boundaryFaces[3] = -Y
    domain boundary boundaryFaces[4] = +Z domain boundary boundaryFaces[5] =
    -Z domain boundary

    Be sure to set ignoreBoundaryFaces to 0 when using this feature.     */


rejectsPerFracture: 10  /*If fracture is rejected, it will be re-translated
to a new position this number of times.

                         This helps hit distribution targets for stochastic
                         families (Set to 1 to ignore this feature)    */




//========================================================================
//                  Shape and Probability Parameters
//========================================================================

//user rectangles and user Ellipses defined in their cooresponding files
```

```
famProb: {.5,.5} /* Probability of occurrence of each family of randomly
distrubuted rectangles and ellipses.  User-ellipses and user-rectangles
insertion will be attempted with 100% likelihood, but with possability they
may be rejected.  The famProb elements should add up to 1.0 (for %100).  The
probabilities are listed in order of families starting with all stochastic
ellipses, and then all stochastic rectangles.

   For example: If  then there are two ellipse families, each with
   probabiliy .3, and two rectangle families, each with probabiliy .2,
   famProb will be: famProb: {.3,.3,.2,.2} Notice: famProb elements add to 1
   */



/*==========================================================================*/
//==========================================================================
//                      Elliptical Fracture Options
//      NOTE: Number of elements must match number of ellipse families  //
(first number in nShape input parameter)
//==========================================================================
/*==========================================================================*/

//Number of ellipse families nFamEll: 0 //Having this option = 0 will ignore
all rectangle family variables

eLayer: {0,0} /* Defines which domain the family belings to.  Layer 0 is the
entire domain.  Layers numbered > 0 coorespond to layers defined above 1
corresponts to the first layer listed, 2 is the next layer listed, etc */

//edist is a mandatory parameter if using statistically generated ellipses
edistr: {2,3}   /* Ellipse statistical distribution options: 1 - lognormal
distribution 2 - truncated power law distribution   3 - exponential
distribution 4 - constant */


ebetaDistribution: {1,1}   /* Beta is the rotation around the polygon's
normal vector, with the polygon centered on x-y plane at the orgin

                    0 - uniform distribution [0, 2PI]    1 - constant angle
                    (specefied below by "ebeta")    */


e_p32Targets: {.1,.1} /* Elliptical families target fracture intensity per
family.  When using stopCondition = 1 (defined at the top of the input
file), families will be inserted untill the families desired fracture
intensity has been reached.  Once all families desired fracture intensity
has been met, fracture generation will be complete.  */

//==========================================================================
// Parameters used by all stochastic ellipse families // Mandatory
Parameters if using statistically generated ellipses

easpect: {1,1}  /* Aspect ratio. Used for lognormal and truncated power law
distribution. */

enumPoints: {12, 12} /*Number of vertices used in creating each elliptical
fracture family. Number of elements must match number of ellipse families
```

```
(first number in nShape) */

eAngleOption: 0     /* All angles for ellipses: 0 - degrees 1 - radians
(Must use numerical value for PI) */

etheta: {-45, 45,} /*Ellipse fracture orientation.  The angle the normal
vector makes with the z-axis */

ephi: {0,0}   /* Ellipse fracture orientation.  The angle the projection of
the normal onto the x-y plane makes with the x-axis */

ebeta: {0, 0}   /* rotation around the normal vector */


ekappa: {8,8}  /*Parameter for the fisher distribnShaprutions. The bigger,
the more similar (less diverging) are the elliptical familiy's normal
vectors */

//========================================================================
// Options Specific For Ellipse Lognormal Distribution (edistr=1): //
Mandatory Parameters if using ellispes with lognormal distribution

//          NOTE: Number of elements must match number of //
ellipse families (first number in nShape)

eLogMean: {2}  //Mean value For Lognormal Distribution.

eLogMax: {100} eLogMin: {1}

esd: {.5} // Standard deviation for lognormal distributions of ellipses

//========================================================================
//     Options Specific For Ellipse Exponential Distribution (edistr=3): //
Mandatory Parameters if using ellispes with exponential distribution


eExpMean: {2}  //Mean value for Exponential  Distribution     eExpMax: {3}
//Mean value for Exponential  Distribution     eExpMin: {1}  //Mean value
for Exponential  Distribution

//========================================================================
//    Options Specific For Constant Size of ellipses (edistr=4):

econst: {10, 10, 10}  // Constant radius, defined per family

//========================================================================
// Options Specific For Ellipse Truncated Power-Law Distribution (edistr=2)
// Mandatory Parameters if using ellipses with truncated power-law dist.

// NOTE: Number of elements must match number //       of ellipse families
(first number in nShape)

emin: {1} // Minimum radius for each ellipse family.  // For power law
distributions.

emax: {6}  // Maximum radius for each ellipse family.  // For power law
distributions.
```

```
ealpha: {2.4} // Alpha. Used in truncated power-law // distribution
calculation




/*=====================================================================*/
/*=====================================================================*/ /*
Rectangular Fractures Options           */ /* NOTE: Number of elements must
match number of rectangle families   */ /*       (second number in nShape
parameter)                            */
/*=====================================================================*/
/*=====================================================================*/

//Number of rectangle families nFamRect: 0 //Having this option = 0 will
ignore all rectangle family variables


rLayer: {0,0} /* Defines which domain the family belings to.  Layer 0 is the
entire domain.  Layers numbered > 0 coorespond to layers defined above 1
corresponts to the first layer listed, 2 is the next layer listed, etc */


/*rdist is a mandatory parameter if using statistically generated rectangles
*/ rdistr: {2,3}   /*  Rectangle statistical distribution options: 1 -
lognormal distribution 2 - truncated power law distribution 3 - exponential
distribution 4 - constant */

rbetaDistribution: {1,1}   /* Beta is the rotation/twist about the z axis
with the polygon centered on x-y plane at the orgin before rotation into 3d
space

                    0 - uniform distribution [0, 2PI]    1 - constant angle
                    (specefied below by "rbeta")

                */

r_p32Targets: {.1,.1} /* Rectangle families target fracture intensity per
family.  When using stopCondition = 1 (defined at the top of the input
file), familiies will be inserted untill the families desired fracture
intensity has been reached. Once all families desired fracture intensity has
been met, fracture generation will be complete.  */

//=====================================================================
// Parameters used by all stochastic rectangle families // Mandatory
Parameters if using statistically generated rectangles

raspect: {1,1}  /* Aspect ratio */

rAngleOption: 0     /* All angles for rectangles: 0 - degrees 1 - radians
(must be numerical value, cannot use "Pi") */

rtheta: {-45,45} /*Rectangle fracture orientation.  The angle the normal
vector makes with the z-axis */

rphi: {0,45} /* Rectangle fracture orientation.  The angle the projection of
the normal onto the x-y plane makes with the x-axis */
```

```
rbeta: {0,0}   /* rotation around the normal vector */

rkappa: {8,8}  /*Parameter for the fisher distributions. The bigger, the
more similar (less diverging) are the rectangle familiy's normal vectors  */

//========================================================================
// Options Specific For Rectangle Lognormal Distribution (rdistr=1): //
Mandatory Parameters if using rectangles with lognormal distribution

rLogMean: {1.6}   /*For Lognormal Distribution.  Mean radius (1/2 rectangle
length) in lognormal distribution for rectangles. */


rLogMax: {100} rLogMin: {1}

rsd: {.4}     /* Standard deviation for lognormal distributions of
rectangles */

//========================================================================
// Options Specific For Rectangle Truncated Power-Law Distribution
(rdistr=2): // Mandatory Parameters if using rectangles with power-law
distribution

 rmin: {1,1}         /* Minimum radius for each rectangle family.  For power
 law distributions. */

 rmax: {6,5}   /* Maximum radius for each rectangle family.  For power law
 distributions. */

 ralpha: {2.4,2.5}   // Alpha. Used in truncated power-law // distribution
 calculation


/*========================================================================*/
/* Options Specific For Rectangle Exponential Distribution (edistr=3):
*/ /* Mandatory Parameters if using rectangules with exponential
distribution   */

rExpMean: {2}  //Mean value for Exponential  Distribution rExpMax: {100}
rExpMin: {1}

/*========================================================================*/
/* Options Specific For Constant Size of rectangles (edistr=4):
*/

rconst: {4,4}  // Constant radius, defined per rectangular family

/*========================================================================*/
/*========================================================================*/
/* User-Specified Ellipses
*/ /* Mandatory Parameters if using user-ellipses
*/ /* NOTE: Number of elements must match number of user-ellipse families
*/ /*(third number in nShape parameter)
*/
/*========================================================================*/
/* NOTE: Only one user-ellipse is placed into the domain per defined
user-ellipse, with possibility of being rejected  */
```

```
userEllipsesOnOff: 0    //0 - User Ellipses off //1 - User Ellipses on

UserEll_Input_File_Path: ./TestCases/test/uEllInput.dat

/*=========================================================================*/
/*=========================================================================*/
/*  User-Specified Ellipses
*/ /*  Mandatory Parameters if using user-ellipses
*/ /*  NOTE: Number of elements must match number of user-ellipse families.
*/ /*  NOTE: Only one user-ellipse is placed into the domain per defined
*/ /*        user-ellipse, with possibility of being rejected
*/
/*=========================================================================*/
/*=========================================================================*/

userEllByCoord: 0 /*  0 - User ellipses defined by coordinates off 1 - User
ellipses defined by coordinates on */

EllByCoord_Input_File_Path:
/home/jharrod/GitProjects/DFNGen/DFNC++Version/inputFiles/
userPolygons/ellCoords.dat



/*=========================================================================*/
/* User-Specified Rectangles
*/ /* Mandatory Parameters if using user-rectangles
*/ /* NOTE: Number of elements must match number of user-ellipse families
*/ /* (fourth number in nShape parameter)
*/
/*=========================================================================*/
/* NOTE: Only one user-rectangle is placed into the domain per defined
user-rectangle, with possibility of being rejected  */


userRectanglesOnOff: 1    //0 - User Rectangles off //1 - User Rectangles on

UserRect_Input_File_Path: /home/nknapp/dfnworks-main/
tests/define_4_user_rects.dat

/*=========================================================================*/
/* If you would like to input user specified rectangles according to their
coordinates, you can use the parameter userDefCoordRec. In that case, all
of the user specified rectangles will have to be according to coordinates.
*/

userRecByCoord: 0 //  0 - user defined rectangles not used //  1 - user
defined rectangles used and defined by input file:

RectByCoord_Input_File_Path: ./inputFiles/userPolygons/rectCoords.dat


/*WARNING: userDefCoordRec can cause LaGriT errors because the polygon
vertices are not put in clockwise or counter-clockwise order.  If errors
(Usualy seg fualt during meshing if using LaGriT), try to reorder the points
till u get it right.  Also, coordinates must be co-planar */

/*=========================================================================*/
```

```
// Aperture [m] /* Mandatory parameter, and can be specified in several
ways: - 1)meanAperture and stdAperture for using LogNormal distribution.  -
2)apertureFromTransmissivity, first transmissivity is defined, and then,
using a cubic law, the aperture is calculated; - 3)constantAperture, all
fractures, regardless of their size, will have the same aperture value; -
4)lengthCorrelatedAperture, aperture is defined as a function of fracture
size*/

//NOTE: Only one aperture type may be used at a time

aperture: 3  //choise of aperture option described above

//(**** 1)meanAperture and stdAperture for using LogNormal
distribution.********) meanAperture:  -3 /*Mean value for aperture using
normal distribution */ stdAperture: 0.8  //Standard deviation

/*(****** 2)apertureFromTransmissivity, first transmissivity is defined, and
then, using a cubic law, the aperture is calculated;***************/
apertureFromTransmissivity: {1.6e-9, 0.8} /* Transmissivity is calculated as
transmissivity = F*R^k, where F is a first element in
aperturefromTransmissivity, k is a second element and R is a mean radius of
a polygon.  Aperture is calculated according to cubic law as
b=(transmissivity*12)^1/3 */

/*(****** 3)constantAperture, all fractures, regardless of their size, will
have the same aperture value;    **********************************/

constantAperture: 1e-5  //Sets constant aperture for all fractures

/*(******** 4)lengthCorrelatedAperture, aperture is defined as a function of
fracture size ******************/

lengthCorrelatedAperture: {5e-5, 0.5} /*Length Correlated Aperture Option:
Aperture is calculated by: b=F*R^k, where F is a first element in
lengthCorrelatedAperture, k is a second element and R is a mean radius of a
polygon.*/


//========================================================================
//Permeability /* Options: 0: Permeability of each fracture is a function of
fracture aperture, given by k=(b^2)/12, where b is an aperture and k is
permeability 1: Constant permeabilty for all fractures */

permOption: 1  //See above for options

constantPermeability: 1e-12  //Constant permeability for all fractures

//========================================================================

outputAcceptedRadiiPerFamily:1 /* output radii files for each family
containing the final radii chosen */

disableFram:0 /* 0 if FRAM (feature rejection algorithm for meshing) is
disabled, 1 otherwise */

outputFinalRadiiPerFamily:1 /* output radii files for each family containing
the final radii chosen */
```

```
insertUserRectanglesFirst:1 /* 1 if user defined rectangles should be
inserted first, 0 otherwise */

forceLargeFractures:0 /* Force large fractures to be included in the network
*/

radiiListIncrease: 0.1 /* Increase the length of the initially generated
radii list (before rejections) by this percentage */

removeFracturesLessThan: 0 /*Used to change the lower cutoff of fracture
size*/
```

2.2 Fracture Cluster Management

2.2.1 Introduction

This section covers dfnGen 2.0's cluster group management system and the isolated fracture removal process.

Fracture clusters are used in dfnGen for isolated fracture removal after the DFN has been generated and before dfnGen generates its output files. An isolated fracture is a fracture that does not intersect any other fractures and will not contribute to flow. Fracture clusters are also considered isolated when the cluster does not connect the users defined domain boundary faces.

NOTE: Isolated fracture removal only removes fractures with no intersections when the input option `ignoreBoundaryFaces` is set to 1.

Fracture cluster data is kept and updated with each new polygon/fracture added to a DFN.

2.2.2 Algorithm Overview

In the dfnGen source code, relevant functions are: 1. `intersectionChecking()`, found in `computationalGeometry.cpp` 2. `assignGroup()`, found in `clusterGroups.cpp` 3. `updateGroups()`, found in `clusterGroups.cpp` 4. `getCluster()`, found in `clusterGroups.cpp`

As a new polygon is being tested for intersections and for feature sizes less than `h` (these checks happen one intersection at a time), three lists are maintained: -a. Intersected polygons list (variable `tempIntersectList` in `intersectionChecking()`). This list contains indices/pointers to all the polygons which the new polygon has intersected in the order that they occur. -b. Intersections list (variable `tempIntPts` in `intersectionChecking()`). This list contains all new intersections (`IntPoints` structures) created by the new polygon in the order that they occur. -c. Encountered cluster groups list (variable `encounteredGroups` in `intersectionChecking()`). This list contains all other cluster group numbers which the new polygon has intersected with after the new polygon already has been assigned a group number.

E.g. If from the first intersection, the new polygon is assigned to group 5, and the next intersection is with a fracture in group 2, '2' is the first group saved to the encountered groups.

When a polygon bridges more than one group, there will be several different cluster groups to update.

If for any reason the fracture is rejected (FRAM rejects it while checking an intersection for features of size less than `h`), these lists are deleted and the fracture is either re-translated to a new position, or a new fracture is generated. If the fracture is accepted, the data in these lists are used to update the permanent fracture cluster data.

Code overview

1. Go through previously accepted polygons and test for intersections with the new polygon being added to the DFN. Once an intersection is found (by function `intersectionChecking()`) and has passed the FRAM tests, several things happen: 2. The intersection structure for the newest intersection is appended to the

   temp intersection array `tempIntPts`.

3. The index of the fracture the new polygon intersects with is appended to the

   intersected polygons list `tempIntersectList`.

4. The index to the new intersection structure's place in the permanent intPts

   array, if the new polygon is accepted, is calculated and appended to the new polygons list `intersectionIndex`. That is, the index that is saved is the index the intersection will have once moved to the permanent array if it is not rejected.

5. Any triple intersection points are saved to a temporary list of structure

   tempData. This structure contains the triple intersection point, and the index to the place in the permanent triplePoints list of where it will go if the polygon is not rejected (similar to step 4).

6. New Polygon Gets a Cluster Group Number (`groupNum` in the Poly struct).

   a. If it is the first intersection found, the new polygon inherits the cluster group number of the intersecting polygon. b. If the new polygon has already been given a cluster group number from intersecting another fracture), the intersecting polygon's cluster group number is added to the encountered cluster groups list `encounteredGroups`. This will be used to update the fractures and cluster groups (merging the two groups together) IF the new polygon does not end up being rejected (it still has more polygons to check for intersections with).

Numbers 2 to 5 repeat until all fractures have been checked for intersections with the new polygon. If the polygon has not been rejected during the process:

7. If no intersections were found after searching through previously accepted

   polygons, the new polygon is given a new cluster group number using the `assignGroup()` function (details below).

8. The new polygon is moved to the permanent `acceptedPoly` list.

9. If there were new intersections, they are now appended to the permanent

   `intPts` list.

10. All intersected polygons will have their `intersectionIndex` list updated

    with the indices of the new intersections. We do this by adding the index of each new intersection to its corresponding polygon in the same order which they were found. The list for polygons we encountered is in the variable `tempIntersectList`.

E.g. if the permanent `intPts` intersection list already has 10 (indexes 0 - 9) intersections from previous fractures and we just added 3 more fractures and intersections, and each fracture can only intersect with the new polygon once, the indexes to the new intersections once they are moved to the permanent `intPts` list will be indexes 10, 11, and 12 (indexes start at 0). So, we append to the first polygon listed in the tempIntersectList index 10, the second polygon in the list index 11, and the third index 12.

11. If there are new triple intersection points, they are now appended to the

    permanent `triplePoints` list. The temporary triple intersection points are held in a list of `TrieplePtTempData` structures. This structure contains the triple intersection point, and the index

for each of the intersections it belongs to (three total). One of the intersections will be a new intersection just created by the new polygon, and the other two will be a triple intersection point on previously accepted intersections.

The new triple intersection point is added to the permanent `triplePoints` array, and then its index in that permanent array is appended to the intersection structure variable `triplePointsIdx` for the intersection that it belongs to.

12. The last thing that is done is a call to the function `updateGroups()`

    (details below).

2.2.3 Function `assignGroup()`: assign polygon to cluster group

The function `assignGroup()`, defined in clusterGroups.cpp, is used to assign a new polygon to a new cluster group. This function is for polygons that do not intersect with any other polygons; otherwise a cluster group will be inherited from the intersected polygon.

Arguments to this function: 1. Poly structure reference. A reference to the new polygon being assigned a new group. 2. Stats structure reference. The program statistics object (variable name pstats throughout the code). The Stats structure contains two structures within it that contain all the cluster group information. These structures are `FractureGroups` and `GroupData` (details below). 3. Index (integer) of the new polygons place in the permanent polygon list `acceptedPoly`.

Code Overview (See sections on GroupData and FractureGroups structures for their details)

1. The new polygon is assigned the next available group number. This comes from

   the Stats variable `nextGroupNum`.

2. A `GroupData` structure is created.

3. Inside the `GroupData` structure, there is a boolean array of six

   elements. This array, faces, contains connectivity information for the cluster. There is an element for each of the six faces, or walls, of the domain. False meaning it is not touching that face, true meaning it is touching the face (see GroupData section for more details). Likewise, there is another faces array in the polygon Poly structure.

The polygon's faces array and the `GroupData's` faces array are bitwise ORed together so that anywhere there is a true in the polygons faces array, there will be a true in the `GroupData's` faces array. After many polygons go through this process for a single cluster group, by looking at the GroupData's faces array we are able to see which domain faces the cluster connects.

4. Next, the variable size inside of the structure `GroupData` is

   incremented. This contains the number of fractures contained in the fracture cluster group. 5. The `GroupData` structure is now saved to a permanent location within the `Stats` structure. 6. A `FractureGroups` structure is now created. 7. The new `FractureGroups` structure is assigned the same group number from step 1 using the same `nextGroupNum` variable. 8. `nextGroupNum` is incremented. 9. Inside the `FractrueGroups` structure is the list (polyList) of polygons belonging to the group. The index for the location in the permanent polygon list, `acceptedPoly`, for the new polygon is added to this list. 10. The `FractureGroup` structure is then saved to a permanent location within the `Stats` structure.

2.2.4 Function `updateGroups()` : update fracture cluster group information

The function `updateGroups()`, defined in clusterGroups.cpp, is used to update the fracture cluster group information for new polygons that have intersected other polygons. When updating the cluster group information, there are two cases: A. The new polygon only intersected with polygons of a single group. B. The new polygon intersected and connected more than one group. The groups now need to be merged together into a single group.

Arguments to this function: 1. `Poly` structure reference. A reference to the new polygon being added to fracture cluster groups. 2. Permanent list of accepted polygons already in the DFN (variable `acceptedPoly`). 3. List of cluster groups which the new polygon has intersected with, if more than one group (see example in part c on page 1). 4. `Stats` structure reference. The program statistics object (variable name `pstats` throughout the code). The `Stats` structure contains two structures within it that contain all the cluster group information. These structures are `FractureGroups` and `GroupData` (details below). 5. Index (integer) of the new polygons place in the permanent polygon list `acceptedPoly`.

Case A

1. The new polygons faces data is ORed into its corresponding `GroupData` structure.

The `GroupData` array, (in variable pstats) is always aligned with cluster group numbers. Group numbers start at 1, the indexes to the array start at 0. E.g. to access the `GroupData` structure for cluster group 12, it is the variable `pstats.groupData[12 - 1]`.

2. The corresponding `GroupData` structure's variable size is incremented

   (number of polygons in the group).

3. Next, the corresponding `FractureGroup` structure must be found. This has

   to be done by searching through the array (`pstats.fractGroup`) and comparing the new polygons `groupNum` and the group number in the `FractureGroup` structure.

See below for an explanation as to why we have to search for the group number, and why the `GroupData` and `FractureGroup` structures are not combined a single structure.

4. Once the correct FractureGroup structure is found, the index to the new

   polygon in the permanent polygon list acceptedPoly is appended to the list polyList in the `FractureGroups` structure.

Case B

1. The new polygon's corresponding `FractureGroup` structure is searched and found. The poly is added to the `FractureGroup` structure (see 3 and 4 in Case A).

2. The new polygon's faces data is ORed into the new polygons corresponding

   `GroupData` structure (see 1 in Case A).

3. The new polygon's corresponding `GroupData` structure has it's size

   incremented (see 2 in case A).

Merge Cluster Groups

4. For all groups in the `encounteredGroups` list (see part c under Algorithm Overview at the beginning of this document), the `GroupData's` size variable, is added to and the `GroupData` structure corresponding to the new polygons group number.

5. The GroupData's faces array for each of the groups in

   encounteredGroups is ORed together with the GroupData structure corresponding to the new polygons group.

6. While doing steps 4 and 5, the GroupData's valid variable for each group

   in encounteredGroups is set to false. This means that that GroupData's data is no longer valid and it should be disregarded (see next section of this document for more details).

7. Search for the corresponding FractureGroup for the group numbers listed

   in encounteredGroups.

8. For each of the corresponding FractureGroups for the group numbers

   listed in encounteredGroups, change the groupNum variable in FractureGroups to the new polygon's group number.

9. Inside the FractureGroups structure, go through all the polygons listed

   there and change their groupNum group number variables to match the new polygon's group number.

2.2.5 Group data structures: GroupData and FractureGroups

Structure Definitions:

NOTE: Both structures use a constructor to initialize their variables (see code in structures.cpp).

```
struct GroupData { unsigned int size; bool valid; bool faces[6]; /* Domain
boundary sides/faces that this cluster connects to..  Index Key: [0]: -x
face, [1]: +x face [2]: -y face, [3]: +y face [4]: -z face, [5]: +z face */
};

struct FractureGroups { unsigned long long int groupNum;
std::vector<unsigned int> polyList; };
```

The reason we do not combine the GroupData and FractureGroups into a single structure is for performance reasons.

If the two structures were combined, a problem arises when two different fracture groups merge together. The structures could no longer be aligned with the group numbers in an array because the group numbers will be changing whenever groups merge together. This would cause constant searching every time you needed to access any of the data. We still need to search when dealing with the FractureGroups array, but save some performance costs by being able to access everything in the GroupData array for any group number without any searching.

If you tried to force the alignment by having empty structures where groups were merged to another group, it would require constantly deleting and reallocating the arrays, and copying polygons to the new group every time groups merged to make everything fit as it should. This would be a huge performance hit and probably the worst solution.

The solution implemented was to keep the two structures separate. When clusters merge together, we simply have to set the old cluster's GroupData valid bit false (no search required), add its size and OR the faces to the GroupData structure that it is being merged into. We then need to find (search required) the group number that is about to go away in the FractureGroups list and change it to the new group number, and change the polygons in that group to the same group number. Nothing is ever re-allocated.

NOTE: When the group number changes in FractureGroups after clusters merge together, there will be two FractureGroups with the same group number but with different polygons listed. To get all the polygons from a single group, the two lists (or more if clusters continued to merge) need to be concatenated.

2.2.6 Funciton `getCluster()` : get a cluster of fractures

The `getCluster()` function is responsible for returning a list of indexes to the polygons which match the user's connectivity option.

Arguments to this function: 1. The program statistics Stats object (named pstats throughout the code). There are three user options that deal with fracture connectivity: 1. `boundaryFaces` a. This option provides a way to select which faces or walls of the domain the user wants the fractures to connect with. It is an array of 6 elements. A zero means not to enforce a connection, a 1 means fractures must have a connection to that face. i. Array elements match to each boundary wall as follows: [0]: -x face, [1]: +x face [2]: -y face, [3]: +y face [4]: -z face, [5]: +z face

2. `ignoreBoundaryFaces` a. This option ignores the `boundaryFaces`

   connectivity option completely and causes `getCluster()` to return a list of all polygons containing at least one intersection. 3. `keepOnlyLargestCluster` a. This option keeps causes getCluster() to return the largest cluster using the above two options as well. If `ignoreBoundaryFaces` is being used, `getCluster()` will return the largest cluster of fractures in the DFN, even if they do not connect to any of the domain walls. If the `boundaryFaces` option is being used, `getCluster()` will return the largest cluster which connects the user's required domain walls.

Code Overview

Part 1: Find cluster groups that match the user's connectivity option 1. If the user is using the `boundaryFaces` option, search through the GroupData and compare the `GroupData's` faces array to the users `boundaryFaces` array. If the groups faces connectivity array connects the required user defined domain walls, add that group number to a list (`matchingGroups` in the code).

2. If the user is using the `ignoreBoundaryFaces` option, go through the

   `GroupData` array and add all the valid groups to the `matchingGroups` array.

3. If the user is using the `keepOnlyLargestCluster` option, go through the

   `matchingGroups` array and compare each group's `GroupData.size` variable. Keep group with the largest size.

4. Search for each group in the `FractureGroups` array and concatenate their

   polygon lists in a list to be returned by the function.

2.3 Exponential Distribution Class Implementation

2.3.1 Introduction

This document is intended for new developers working on dfnGen. It covers the implementation of the `Distributions` class, and its composed exponential distribution class `ExpDist` in dfnGen V2.0.

During dfnGen 2.0 development, new functionality was needed to allow for the control of the range of numbers produced by the exponential distribution. Previously, dfnGen V2.0 was developed using the C++ standard library, `random`.

2.3.2 Need for a Customized Exponential Distribution

There was need to control the minimum fracture size for exponential distributed fracture families for research purposes. Also, all fracture radii must always be greater than the minimum feature size `h`.

The exponential distribution favors small numbers that caused a lot of re-sampling when the distribution generated fracture radii of less than h or smaller than the user's defined minimum radius. Re-sampling the standard library's exponential distribution when the distribution produced numbers outside of the user's defined ranged was found to be very inefficient and could halt program execution when the exponential mean did not match the range which the user had chosen. The program could re-sample the distribution thousands of times before an acceptable radius was generated.

With the standard library's implementation, complete randomness is forced from the distribution. There was no way to control the range of numbers produced by the distribution. A way of limiting the output of the distribution was needed that did not involve re-sampling.

Implementation Overview ********** Our implementation uses the CDF determine the random variable range from which we need to sample. When the inverse CDF is sampled uniformly between 0 and 1, an exponential distribution will be produced that matches that of the standard library's exponential distribution output. By limiting the random variable range, we can sample between the users desired minimum and maximum without generating numbers outside of that range.

To limit the range of output, we use the exponential CDF formula: `rv = 1 - e (-lambda * output)`, where rv is the random variable needed to produce output when plugged into the inverse CDF function: `output = -log(1-rv)/lambda`.

When the user's defined minimum and maximum are plugged in to output, we get the range which the distribution should be sample from in order to get a exponential distribution bounded by the users defined minimum and maximum.

These variables, the range to sample the exponential distribution, are saved to minDistInput and maxDistInput in the family's corresponding Shape structure.

2.3.3 Implementation Details

Our implementation uses composition for increased modularity and to increase the ease of adding additional distribution types in the future.

The `ExpDist` class is a sub-class of the `Distributions` class. This allows the programmer to only create one instance of the `Distributions` class, and the `ExpDist` class and any other distribution classes added in the future will be automatically set up and initialized by `Distributions` constructor.

`Distributions` Class ********** The `Distributions` class contains functions and variables that are needed to initialize the `ExpDist` class, and likely other distribution classes added in the future. It also contains the `ExpDist` class within it.

When the `Distributions` class is created, its constructor function is called. This function creates and initializes the `ExpDist` class within the `Distributions` class.

One of the issues with the exponential distribution is that if given 1.0 as a random variable, the distribution returns inf. To maximize the range of numbers which can be produced, we need to know the largest value less than 1.0 that the computer is able to produce.

The `Distributions` class has a function called `getMaxDecimalDouble()`. During `Distributions` creation, `getMaxDecimalDouble()` returns the largest number less than 1, e.g. 0.999....9, to its maximum precision. This variable is saved to variable `maxInput` in the `Distributions` class. It is also passed to the `ExpDist` class during its creation.

Also in the `Distributions` class constructor, the function `checkDistributionUserInput()` is called. This function error checks user exponential input options and finishes initializing the exponential distribution. The function is written with the expectation for other distributions to be added and will be easy to modify.

In `checkDistributionUserInput()`, `minDistInput` and `maxDistInput` are initialized for each family using exponential distribution (see Implementation Overview). Error checks are performed to ensure `minDistInput` and `maxDistInput` are within the machines capabilities to produce. If they are set very high,

plugging in `maxInput` (see above) into the distribution can produce a number smaller than the requested maximum, and possibly minimum. If the user defined maximum cannot be produced stochastically, the user is warned and the user defined maximum is set to the largest possible number that the machine can produce. The minimum is then checked to ensure it is still less than the maximum. If it is not, the error is reported to the user and the program terminates. Otherwise, everything is okay and the `ExpDist` class is ready to use.

2.3.4 `ExpDist` Class

After the `ExpDist` class has been initialized, the `getValue()` function can be used to return random numbers from the exponential distribution. The function has been overloaded to either be given the random input variable (random variable between 0 and 1) as an argument, or be given a range between 0 and 1 to generate random input variables from.

2.3.5 Other Details

The C++ standard random library is still used for generating uniform random reals. The 64-bit Mersenne twister engine random generator is the random generator used for all dfnGen's random variables. It is created in main() and passed as a reference to the Distributions class during its creation.

2.4 Hotkey ~

If the dfnGen takes too long, one can use ~ to abort fracture generation process and contine to the next step of outputting the data related the fractures generated until that point in time.

2.4.1 Developer notes: Variables that might need adjusting

Due to the recent changes in the LaGriT meshing script, there are a couple parts of the code that might need adjusting.

Distance between intersections

After updates to the meshing script, there are cases where intersections can have only one triangular element between them. If the distance between intersections needs to be increased, adjust the last argument in `checkDistDistToOldIntersections()` and `checkDistToNewIntersections()`, lines 645 and 653 in computationalGeometry.cpp

Allowed Intersection Angles

The changes to the LaGriT meshing script might allow for smaller angles without causing problems in the mesh. This is for intersection angles crossing the edge of a polygon, not for triple intersections.

To change the angle, adjust the variable `const static double minDist2` found on line 1260 in `computationalGeometry.cpp`.

`minDist2` is the minimum distance allowed to the edge of a polygon from the first discretized intersection point, not including the end points (the first node in from the end point).

2.4.2 Adding new user input variables to dfnGen 2.0

1. Add option/variable to an existing input file. Tag the option's name with `:` at the end. There must be at least 1 space or a new line in between the `:` and the data. E.g. `newUserOption: 12`

2. Add `extern varType varName` to `input.h`. Most user input variables are

   stored globally. `input.h` must be included in any files that need access to them.

3. Update `readInput.cpp`. Declare the new global variable (the same variable

   as in step 1 but without the `extern` keyword) at the top of this file.

This file contains the function `getInput()`. This function is responsible for reading in user input files. `getInput()` needs to be updated to read in the new variable. I suggest looking for a similar variable, whether it be an array, a flag, or a number, and use that as an example to read in the new input option.

The function `searchVar()` is very helpful in reading variables from the user input file. The first argument is the file object (C++ ifstream object), the second argument is a string of the variable/option name in the input file including the `:` at the end. After this function runs, the file pointer will be pointing to the data directly after the input options name (e.g. in step 1, the file pointer will be pointing to the white space directly after the colon.) All that is left is to read the input variable in to a C++ variable e.g. `file >> var`. NOTE: C++ is smart and will skip multiple spaces and/or new line characters.

If the option requires a list or array as the options parameters, see similar options in readInput.cpp. Instead of reading in directly to a variable (`file >> var`), a function will be required to parse the list. See `readInputFunctions.cpp` and `readInputFunctions.h` for some examples on how to do this.

4. The last thing to do is to write/edit the code that will use the new option. Include `input.h` in any new file to access the global variable. If the new variable is an array, don't forget to use `delete[]` to free its memory after the variable is no longer needed. If a new file was created, be sure to edit the makefile to include it in the built.

CHAPTER

THREE

DFNFLOW

dfnFlow involves using flow solver such as PFLOTRAN or FEHM. PFLOTRAN is recommended if large number of fractures (> O(1000)) are involved in a network. Using the function calls that are part of dfnworks_python, one can create the mesh files needed to run PFLOTRAN. This will involve creating unstructured mesh file `*uge` as well as the boundary `*ex` files. Please see the PFLOTRAN user manual at http://www.pflotran.org under unstructured *explicit* format usage for further details. An example input file for PFLOTRAN is provided in the repository. Please use this as a starting point to build your input deck.

Below is a sample input file. Refer to the PFLOTRAN user manual at http://www.pflotran.org for input parameter descriptions.

```
# Jan 13, 2014
# Nataliia Makedonska, Satish Karra, LANL
#================================================

SIMULATION
  SIMULATION_TYPE SUBSURFACE
  PROCESS_MODELS
    SUBSURFACE_FLOW flow
      MODE RICHARDS
    /
  /
END
SUBSURFACE

DFN

#=========================== discretization ===================================
GRID
  TYPE unstructured_explicit full_mesh_vol_area.uge
  GRAVITY 0.d0 0.d0 0.d0
END


#=========================== fluid properties =================================
FLUID_PROPERTY
  DIFFUSION_COEFFICIENT 1.d-9
END

DATASET Permeability
  FILENAME dfn_properties.h5
END

#=========================== material properties ==============================
MATERIAL_PROPERTY soil1
  ID 1
```

```
  POROSITY 0.25d0
  TORTUOSITY 0.5d0
  CHARACTERISTIC_CURVES default
  PERMEABILITY
    DATASET Permeability
  /
END


#========================== characteristic curves ===========================
CHARACTERISTIC_CURVES default
  SATURATION_FUNCTION VAN_GENUCHTEN
    M 0.5d0
    ALPHA  1.d-4
    LIQUID_RESIDUAL_SATURATION 0.1d0
    MAX_CAPILLARY_PRESSURE 1.d8
  /
  PERMEABILITY_FUNCTION MUALEM_VG_LIQ
    M 0.5d0
    LIQUID_RESIDUAL_SATURATION 0.1d0
  /
END

#========================== output options ==================================
OUTPUT
  TIMES s 0.01 0.05 0.1 0.2 0.5 1
#  FORMAT TECPLOT BLOCK
  PRINT_PRIMAL_GRID
  FORMAT VTK
  MASS_FLOWRATE
  MASS_BALANCE
  VARIABLES
    LIQUID_PRESSURE
    PERMEABILITY
  /
END

#========================== times ===========================================
TIME
  INITIAL_TIMESTEP_SIZE  1.d-8 s
  FINAL_TIME 1.d0 d==
  MAXIMUM_TIMESTEP_SIZE 10.d0 d
  STEADY_STATE
END

# REFERENCE_PRESSURE 1500000.

#========================== regions =========================================
REGION All
  COORDINATES
    -1.d20 -1.d20 -1.d20
    1.d20 1.d20 1.d20
  /
END

REGION inflow
  FILE pboundary_left_w.ex
END
```

```
REGION outflow
  FILE pboundary_right_e.ex
END

#========================== flow conditions ===================================
FLOW_CONDITION initial
  TYPE
     PRESSURE dirichlet
  /
  PRESSURE 1.01325d6
END


FLOW_CONDITION outflow
  TYPE
     PRESSURE dirichlet
  /
  PRESSURE 1.d6
END

FLOW_CONDITION inflow
  TYPE
    PRESSURE dirichlet
  /
  PRESSURE 2.d6
END

#========================== condition couplers ================================
# initial condition
INITIAL_CONDITION
  FLOW_CONDITION initial
  REGION All
END


BOUNDARY_CONDITION INFLOW
  FLOW_CONDITION inflow
  REGION inflow
END

BOUNDARY_CONDITION OUTFLOW
  FLOW_CONDITION outflow
  REGION outflow
END

#========================== stratigraphy couplers =============================
STRATA
  REGION All
  MATERIAL soil1
END

END_SUBSURFACE
```

CHAPTER

FOUR

DFNTRANS

dfnTrans is a method for resolving solute transport using control volume flow solutions obtained from dfnFlow on the unstructured mesh generated using dfnGen. We adopt a Lagrangian approach and represent a non-reactive conservative solute as a collection of indivisible passive tracer particles. Particle tracking methods (a) provide a wealth of information about the local flow field, (b) do not suffer from numerical dispersion, which is inherent in the discretizations of advection–dispersion equations, and (c) allow for the computation of each particle trajectory to be performed in an intrinsically parallel fashion if particles are not allowed to interact with one another or the fracture network. However, particle tracking on a DFN poses unique challenges that arise from (a) the quality of the flow solution, (b) the unstructured mesh representation of the DFN, and (c) the physical phenomena of interest. The flow solutions obtained from dfnFlow are locally mass conserving, so the particle tracking method does not suffer from the problems inherent in using Galerkin finite element codes.

dfnTrans starts from reconstruction of local velocity field: Darcy fluxes obtained using dfnFlow are used to reconstruct the local velocity field, which is used for particle tracking on the DFN. Then, Lagrangian transport simulation is used to determine pathlines through the network and simulate transport. It is important to note that dfnTrans itself only solves for advective transport, but effects of longitudinal dispersion and matrix diffusion, sorption, and other retention processes are easily incorporated by post-processing particle trajectories. The detailed description of dfnTrans algorithm and implemented methodology is in Makedonska, N., Painter, S. L., Bui, Q. M., Gable, C. W., & Karra, S. (2015). Particle tracking approach for transport in three-dimensional discrete fracture networks. Computational Geosciences, 19(5), 1123-1137.

All source files of C code of dfnTrans are in `ParticleTracking/` directory of dfnWorks 2.0. It compiles under linux/mac machines using `makefile`. In order to run transport, first, all the parameters and paths should be set up in the PTDFN Control file, PTDFN_control.dat. Then, the following command should be run: `./dfnTrans <PTDFN_control.dat`

PTDFN_control.dat file sets all necessary parameters to run particle tracking in dfnWorks. Below is one `PTDFN_control.dat` example that includes a short explanation of each parameter setting:

```
/*********************************************************************/ /*
CONTROL FILE FOR PARTICLE TRACKING IN DISCRETE FRACTURE NETWORK   */
/*********************************************************************/

// 1. Define paths of files that were generated by DFNGEN and contain all
the information about computational grid.

/**********************  INPUT FILES: grid ****************************/
/**** input files with grid of DFN, mainly it's output of dfnGen ******/
param: params.txt //generated by dfnGen, contains the number of fractures in
the DFN

poly: poly_info.dat //generated by dfnGen, contains the normal vectors of
each polygon; used for rotation each fracture

inp: full_mesh.inp  // AVS file of full mesh of DFN: positions of vertices
```

```
and connectivity list

stor: tri_fracture.stor // produced by LaGriT: the matrix of all connected
nearest neighbors faces of Voronoi polygons and their geometrical
coefficients

boundary: allboundaries.zone //list of nodes that located on domain
boundaries /* boundary conditions: reading the nodes that belong to in-flow
and out-flow boundaries. Should be consistent with those applied to obtain
steady state pressure solution (PFLOTRAN)   */ /*1 - top;  2 - bottom;  3 -
left_w;  4 - front_s;  5 - right_e;  6 - back_n */ in-flow-boundary: 3
out-flow-boundary: 5

/* The allboundaries.dat file can be modified according to task. For
example, inflow boundaries are left and top, and out flow is a bottom face
of the domain.  In this case, left and top list of boundary nodes should be
combined together in allboundaries.dat and only one number given (for
example 1). dfnTrans code can't read more than one defined number for
in-flow or out-flow boundaries.  Also, number of boundary zone is not
limited by 6. */

// 2. Input of Flow Solver results

/**************** INPUT FILES: PFLOTRAN flow solution ******************/
PFLOTRAN: yes //yes - the PFLOTRAN flow solver is used

PFLOTRAN_vel: darcyvel.dat // the Darcy flux for each Control Volume face of
the full mesh of DFN

PFLOTRAN_cell: cellinfo.dat

/*connectivity list and area of each Control Volume face of the full mesh of
DFN.  In case of using one of recent versions of PFLOTRAN, where
PFLOTRAN_vel file consists areas of connection, PFLOTRAN_uge file will not
be used.*/

PFLOTRAN_uge: lagrit_pflotran_dat/lagrit_pflotran_corrected.uge

// The current version of DFNWorks uses PFLOTRAN /**************** INPUT
FILES: FEHM flow solution **********************/ /*currently we are using
PFLOTRAN , but the code would work with FEHM, too */ FEHM: no

FEHM_fin:  tri_frac.fin // fluxes and pressure outputs of FEHM

// 3. Settings of OUTPUT files with particle tracking results.

/************************  OUTPUT FILES  ******************************/
/* initial grid info structure output, useful for debugging */ out_grid: yes
// yes means that the output ASCII files, named "nodes" and "fractures",
will be generated. If you don't want this output  - type "no" instead of
"yes".

/* flow field: 3D Darcy velocities: output ASCII file "Velocity3D" has an
each nodes position and its Darcy velocity, reconstructed from fluxes */
out_3dflow: yes

/* out initial positions of particles into separate ASCII file "initpos" */
out_init: yes
```

```
/* out particle trajectories tortuosity file, torts.dat */ out_tort: no

/*************** output options for particles trajectories ***************/
/* output frequency is set according to trajectories curvature. We check the
curvature of particles trajectory each segment, from intersection to
intersection.  If it's like a straight line, then the output is less
frequent (in case of "out_curv:yes", if "no",  the output file will contain
every time step of the simulation) */ out_curv: no

/* output into avs file (GMV visualization, Paraview visualization) */
out_avs: no

/* output into trajectories ascii files (veloc+posit+cell+fract+time) */
out_traj: no

/************* output directories ************************************/
out_dir: ptresults1 /* path and name of directory where all the particle
tracking results will be written, including those defined above*/


out_path: trajectories /*name of directory where all particle trajectories
will be saved, in out_dir path */

/* name of resultant file (in out_dir path), which contains total travel
time and final positions of particles */ out_time: partime

// 4. Options for Particles Initial Positions in DFN

/*************** PARTICLES INITIAL POSITIONS
*****************************/ /****init_nf: if yes - the same number of
particles (init_partn) will be placed on every boundary fracture edge on
in-flow boundary, equidistant from each other ****/ init_nf: yes init_partn:
5

/****init_eqd: if yes - particles will be placed on the same distance from
each other on all over in-flow boundary edges ***********************/  //
The difference between options init_eqd: and init_nf: is the following. In
case of init_eqd the total length of fracture edges on in-flow boundaries
will be calculated. Then, according to init_npart given number of particles,
the particles will be distributed equidistant over all fracture edges on
in-flow boundaries.  In init_nf option, the init_partn number of particles
will be equidist in each edge of fracture on in-inflow boundaries. In this
case distance between two neighboring particles in one fracture will not be
the same as distance between two particles in other fracture.   init_eqd: no
//maximum number of particles that user expects on one boundary edge
init_npart: 1

/*** all particles start from the same region at in-flow boundary, in a
range  {in_xmin, in_xmax,in_ymin, in_ymax, in_zmin, in_zmax} **************/
//  In this option, the region on in-flow boundary should be defined
according to x, y, and z coordination of the domain. Then particles will be
placed equidistant in those part of fracture edges that cross the defined
region. If there are no fracture edge found there, the program will be
terminated, and user should redefine the region.  init_oneregion: no
in_partn: 10 in_xmin: -20.0 in_xmax: 20.0 in_ymin: -20.0 in_ymax: 20.0
in_zmin: 499.0 in_zmax: 501.0

/**** all particles are placed randomly over all fracture surface (not only
```

```
on boundary edges!) ***********************************/ // In this option
the particles will be placed on the center of randomly chosen Control Volume
cell over all cells in DFN mesh (not only on in-flow boundary).
init_random: no // total number of particles in_randpart: 110000


// 5. Flow and Fracture Parameters

/****************** FLOW AND FRACTURE PARAMETERS **********************/
porosity: 0.25 // porosity

density: 997.73  //fluid density

satur: 1.0

thickness: 1.0 //DFN aperture  (used in case of no aperture file provided)

aperture: yes  //DFN aperture

aperture_type: frac //aperture is giving per cell (type "cell") //    or per
fracture (type "frac") // In the current version of DFNWorks the only an
aperture per fracture option is given.

aperture_file: aperture.dat  // The ASCII file aperture.dat is produced by
dfnGen

/********************  TIME ******************************************/
timesteps: 2000000

//units of time (years, days, hours, minutes) time_units:  seconds

/**** flux weighted particles*/ // in all the options of initial positions
of particles, particles can be weighted either by input flux  )in case of
placing particles in in-flow boundary) of by current cell aperture (in case
of randomly defined initial positions).  flux_weight: yes

/* random generator seed */ seed: 337799

// 6. Control Plane Output /********************  Control Plane/Cylinder
Output ********************/ // Here is another option for output. The
control Planes can be defined on any position along the flow direction. For
example, if fluid flow goes from top to bottom along Z direction
(flowdir=2), the imaginary control planes will be parallel  to x-y plane and
placed each 1 m (delta_Control: 1). Each particle will have it's data output
(location, current velocity) every time it crosses control plane.

/*** virtual Control planes will be build in the direction of flow.  Once
particle crosses the control plane, it's position, velocity, time will
output to an ascii file. ****/ ControlPlane: no

/* the path and directory name with all particles output files */
control_out: outcontroldir

/* Delta Control Plane - the distance between control planes */
delta_Control: 1

/* ControlPlane: direction of flow: x-0; y-1; z-2 */ flowdir: 1
```

```
/*****************************************************************************/
END
```

CHAPTER

FIVE

PYDFNWORKS: THE DFNWORKS PYTHON PACKAGE

The pydfnworks package allows the user to easily run dfnWorks from the command line and call dfnWorks within other python scripts. Because pydfnworks is a package, users can call individual methods from the package easily.

The pydfnworks must be setup by the user using the following command in the directory dfnworks-main/pydfnworks/ :

python setup.py install (if the user has admin privileges), OR:

python setup.py install –user (if the user does not have admin privileges):

The documentation below includes all the methods and classes of the pydfnworks package.

5.1 DFNWORKS

class pydfnworks.**DFNWORKS**(*jobname=''*, *local_jobname=''*, *dfnGen_file=''*, *output_file=''*, *local_dfnGen_file=''*, *ncpu=''*, *dfnFlow_file=''*, *local_dfnFlow_file=''*, *dfnTrans_file=''*, *inp_file='full_mesh.inp'*, *uge_file=''*, *vtk_file=''*, *mesh_type='dfn'*, *perm_file=''*, *aper_file=''*, *perm_cell_file=''*, *aper_cell_file=''*, *dfnTrans_version=''*, *num_frac=''*)

Class for DFN Generation and meshing

Attributes:

- _jobname: name of job, also the folder where output files are stored
- _ncpu: number of CPUs used in the job
- _dfnGen file: the name of the dfnGen input file
- _dfnFlow file: the name of the dfnFlow input file
- _local prefix: indicates that the name contains only the most local directory
- _vtk_file: the name of the VTK file
- _inp_file: the name of the INP file
- _uge_file: the name of the UGE file
- _mesh_type: the type of mesh
- _perm_file: the name of the file containing permeabilities
- _aper_file: the name of the file containing apertures
- _perm_cell file: the name of the file containing cell permeabilities
- _aper_cell_file: the name of the file containing cell apertures

- _dfnTrans_version: the version of dfnTrans to use
- _freeze: indicates whether the class attributes can be modified
- _large_network: indicates whether C++ or Python is used for file processing at the bottleneck

of inp to vtk conversion

check_input(*input_file=''*, *output_file=''*)

Input Format Requirements:

- Each parameter must be defined on its own line (separate by newline)
- A parameter (key) MUST be separated from its value by a colon ':' (ie. –> key: value)
- Values may also be placed on lines after the 'key'
- Comment Format: On a line containing // or / $*$, nothing after $*$ / or // will be processed but text before a comment will be processed

Kwargs:

- input_file (name): name of dfnGen input file
- output_file (name): stripped down input file for DFNGen

commandline_options()

Read command lines for use in dfnWorks.

Options:

- -name : Jobname (Mandatory)
- -ncpu : Number of CPUS (Optional, default=4)
- -input : input file with paths to run files (Mandatory if the next three options are not specified)
- -gen : Generator Input File (Mandatory, can be included within this file)
- -flow : PFLORAN Input File (Mandatory, can be included within this file)
- -trans: Transport Input File (Mandatory, can be included within this file)
- -cell: True/False Set True for use with cell based aperture and permeabuility (Optional, default=False)
- -large_network: True/False Set True to use CPP for file processing bottleneck (Optional, default=False)

copy_dfnTrans_files()

create link to DFNTRANS and copy input file into local directory

create_network()

Execute dfnGen and print whether the generation of the fracture network failed or succeeded. The params.txt file must be there for success.

dfnFlow()

Run the dfnFlow portion of the workflow.

dfnGen()

Run the dfnGen workflow:

- 1. make_working_directory: Create a directory with name of job
- 2. check_input: Check input parameters and create a clean version of the input file
- 3. create_network: Create network. DFNGEN v2.0 is called and creates the network

- 4. output_report: Generate a PDF summary of the DFN generation
- 5. mesh_network: calls module dfnGen_meshing and runs LaGriT to mesh the DFN

dfnTrans()
Copy input files for dfnTrans into working directory and run DFNTrans

lagrit2pflotran(*inp_file=''*, *mesh_type=''*, *hex2tet=False*)
Takes output from LaGriT and processes it for use in PFLOTRAN.

Kwargs:

- inp_file (str): name of the inp (AVS) file produced by LaGriT
- mesh_type (str): the type of mesh
- hex2tet (boolean): True if hex mesh elements should be converted to tet elements, False otherwise.

legal()
Print the legal LANL statement for dfnWorks.

make_working_directory()
make working directories for fracture generation

mesh_network(*production_mode=True*, *refine_factor=1*, *slope=2*)
Mesh Fracture Network using ncpus and lagrit meshing file is separate file: dfnGen_meshing.py

output_report(*radiiFile='radii.dat'*, *famFile='families.dat'*, *transFile='translations.dat'*, *rejectFile='rejections.dat'*, *output_name=''*)
Create PDF report of generator

Notes:

- Set the number of histogram buckets (bins) by changing numBuckets variable in his graphing functions
- Also change number of x-values used to plot lines by changing numXpoints variable in appropriate funcs
- Set show = True to show plots immediately and still make pdf
- NOTE future developers of this code should add functionality for radiiList of size 0.

parse_pflotran_vtk(*grid_vtk_file=''*)
Using C++ VTK library, convert inp file to VTK file, then change name of CELL_DATA to POINT_DATA.

parse_pflotran_vtk_python(*grid_vtk_file=''*)
Replace CELL_DATA with POINT_DATA in the VTK output.

pflotran()
Run pflotran Copy PFLOTRAN run file into working directory and run with ncpus

pflotran_cleanup()
Concatenate PFLOTRAN output files and then delete them

run_dfnTrans()
run dfnTrans simulation

write_perms_and_correct_volumes_areas(*inp_file=''*, *uge_file=''*, *perm_file=''*, *aper_file=''*)
Write permeability values to perm_file, write aperture values to aper_file, and correct volume areas in uge_file

zone2ex(*uge_file=''*, *zone_file=''*, *face=''*)
Convert zone files from LaGriT into ex format for LaGriT inputs: uge_file: name of uge file zone_file: name of zone file face: face of the plane corresponding to the zone file

zone_file='all' processes all directions, top, bottom, left, right, front, back

5.2 dfnGen

5.2.1 Processing generator input

pydfnworks.gen_input.**check_input**(*self*, *input_file=''*, *output_file=''*)

Input Format Requirements:

- Each parameter must be defined on its own line (separate by newline)
- A parameter (key) MUST be separated from its value by a colon ':' (ie. –> key: value)
- Values may also be placed on lines after the 'key'
- Comment Format: On a line containing // or / $*$, nothing after $*$ / or // will be processed but text before a comment will be processed

Kwargs:

- input_file (name): name of dfnGen input file
- output_file (name): stripped down input file for DFNGen

class pydfnworks.distributions.**distr**(*params*, *numEdistribs*, *numRdistribs*, *minFracSize*)

Verifies the fracture distribution input parameters for dfnGen.

Attributes:

- params (list): parameters for dfnGen
- numEdistribs (int): number of ellipse family distributions
- numRdistribs (int): number of rectangle family distributions
- minFracSize (double): minimum fracture size

betaDistribution(*prefix*)

Verifies both the "ebetaDistribution" and "rBetaDistribution". If either contain any flags indicating contant angle (1) then the corresponding "ebeta" and/or "rbeta" parameters are also verified.

Args: prefix (str): Indicates shapes that the beta distribution describes. 'e' if they are ellipses, 'r' if they are rectangles.

constantDist(*prefix*)

Verifies paramters for constant distribution of fractures

distr(*prefix*)

Verifies "edistr" and "rdistr" making sure one distribution is defined per family and each distribution is either 1 (log-normal), 2 (Truncated Power Law), 3 (Exponential), or 4 (constant). Stores how many of each distrib are in use in numEdistribs or numRdistribs lists.

exponentialDist(*prefix*)

Verifies parameters for exponential distribution of fractures.

lognormalDist(*prefix*)

Verifies all logNormal Parameters for ellipses and Rectangles.

tplDist(*prefix*)

Verifies parameters for truncated power law distribution of fractures.

5.2.2 Running the generator

pydfnworks.generator.**create_network**(*self*)
Execute dfnGen and print whether the generation of the fracture network failed or succeeded. The params.txt file must be there for success.

pydfnworks.generator.**dfnGen**(*self*)

Run the dfnGen workflow:

- 1. make_working_directory: Create a directory with name of job
- 2. check_input: Check input parameters and create a clean version of the input file
- 3. create_network: Create network. DFNGEN v2.0 is called and creates the network
- 4. output_report: Generate a PDF summary of the DFN generation
- 5. mesh_network: calls module dfnGen_meshing and runs LaGriT to mesh the DFN

pydfnworks.generator.**make_working_directory**(*self*)
make working directories for fracture generation

5.2.3 Graphing generator output

pydfnworks.gen_output.**output_report**(*self*, *radiiFile='radii.dat'*, *famFile='families.dat'*, *transFile='translations.dat'*, *rejectFile='rejections.dat'*, *output_name=''*)
Create PDF report of generator

Notes:

- Set the number of histogram buckets (bins) by changing numBuckets variable in his graphing functions
- Also change number of x-values used to plot lines by changing numXpoints variable in appropriate funcs
- Set show = True to show plots immediately and still make pdf
- NOTE future developers of this code should add functionality for radiiList of size 0.

5.3 dfnFlow

pydfnworks.flow.**dfnFlow**(*self*)
Run the dfnFlow portion of the workflow.

pydfnworks.flow.**inp2gmv**(*self*, *inp_file=''*)
Convert inp file to gmv file, for general mesh viewer .

Kwargs: inp_file (str): name of inp file

pydfnworks.flow.**lagrit2pflotran**(*self*, *inp_file=''*, *mesh_type=''*, *hex2tet=False*)
Takes output from LaGriT and processes it for use in PFLOTRAN.

Kwargs:

- inp_file (str): name of the inp (AVS) file produced by LaGriT
- mesh_type (str): the type of mesh
- hex2tet (boolean): True if hex mesh elements should be converted to tet elements, False otherwise.

pydfnworks.flow.**parse_pflotran_vtk**(*self*, *grid_vtk_file=''*)
Using C++ VTK library, convert inp file to VTK file, then change name of CELL_DATA to POINT_DATA.

pydfnworks.flow.**parse_pflotran_vtk_python**(*self*, *grid_vtk_file=''*)
Replace CELL_DATA with POINT_DATA in the VTK output.

pydfnworks.flow.**pflotran**(*self*)
Run pflotran Copy PFLOTRAN run file into working directory and run with ncpus

pydfnworks.flow.**pflotran_cleanup**(*self*)
Concatenate PFLOTRAN output files and then delete them

pydfnworks.flow.**write_perms_and_correct_volumes_areas**(*self*, *inp_file=''*, *uge_file=''*, *perm_file=''*, *aper_file=''*)
Write permeability values to perm_file, write aperture values to aper_file, and correct volume areas in uge_file

pydfnworks.flow.**zone2ex**(*self*, *uge_file=''*, *zone_file=''*, *face=''*)
Convert zone files from LaGriT into ex format for LaGriT inputs: uge_file: name of uge file zone_file: name of zone file face: face of the plane corresponding to the zone file

zone_file='all' processes all directions, top, bottom, left, right, front, back

5.4 dfnTrans

pydfnworks.transport.**copy_dfnTrans_files**(*self*)
create link to DFNTRANS and copy input file into local directory

pydfnworks.transport.**dfnTrans**(*self*)
Copy input files for dfnTrans into working directory and run DFNTrans

pydfnworks.transport.**run_dfnTrans**(*self*)
run dfnTrans simulation

5.5 LaGriT (meshing)

5.5.1 Mesh DFN

pydfnworks.meshdfn.**mesh_network**(*self*, *production_mode=True*, *refine_factor=1*, *slope=2*)
Mesh Fracture Network using ncpus and lagrit meshing file is separate file: dfnGen_meshing.py

5.5.2 LaGrit scripts

pydfnworks.lagrit_scripts.**create_lagrit_scripts**(*visual_mode*, *ncpu*, *refine_factor=1*, *production_mode=True*)
Creates LaGriT script to be run for each polygon

Inputs:

- visual_mode (True / False)
- ncpu: number of cpus
- refine_fractor: used rectangles
- production_mode (True/False)

pydfnworks.lagrit_scripts.**create_merge_poly_files**(*ncpu*, *num_poly*, *visual_mode*)

Section 4 [Create merge_poly file] Creates a lagrit script that reads in each mesh, appends it to the main mesh, and then deletes that mesh object Then duplicate points are removed from the main mesh using EPS_FILTER The points are compressed, and then written in the files full_mesh.gmv, full_mesh.inp, and an FEHM dump is preformed.

pydfnworks.lagrit_scripts.**create_parameter_mlgi_file**(*num_poly*, *h*, *slope=2*, *refine_dist=0.5*)
create parameter_mgli_files Outputs parameteri.mlgi files used in running LaGriT Scripts

Inputs:

- num_poly: Number of polygons
- h: meshing length scale
- slope: Slope of coarsening function, default = 2, set to 0 for uniform mesh resolution
- refine_dist: distance used in coarsing function, default = 0.5,

pydfnworks.lagrit_scripts.**create_user_functions**()
Create user_function.lgi files for meshing

pydfnworks.lagrit_scripts.**define_zones**(*h*, *domain*)
Creates and runs LaGriT script to define domain size

5.5.3 Run meshing in parallel

pydfnworks.run_meshing.**merge_the_meshes**(*num_poly*, *ncpu*, *n_jobs*, *visual_mode*)

Merges all the meshes together, deletes duplicate points, dumps the .gmv and fehm files

pydfnworks.run_meshing.**mesh_fracture**(*fracture_id*, *visual_mode*, *num_poly*)
Child function for parallelized meshing of fractures

pydfnworks.run_meshing.**mesh_fractures_header**(*num_poly*, *ncpu*, *visual_mode*)
Header function for Parallel meshing of fractures

Creates a queue of fracture numbers ranging form 1, num_poly

Each fractures is meshed using mesh_fracture called within the worker function.

If any fracture fails to mesh properly, then a folder is created with that fracture information and the fracture number is written into failure.txt.

Returns:

- True: If failure.txt is empty meaning all fractures meshed correctly
- False: If failure.txt is not empty, then at least one fracture failed.

pydfnworks.run_meshing.**worker**(*work_queue*, *done_queue*, *visual_mode*, *num_poly*)
Worker function for parallelized meshing

5.6 Helper methods

5.6.1 Mesh helper methods

pydfnworks.mesh_dfn_helper.**check_dudded_points**(*dudded*)
Parses Lagrit log_merge_all.txt and checks if number of dudded points is the expected number

Returns:

- True if the number of dudded points is correct
- False if the number of dudded points is incorrect

pydfnworks.mesh_dfn_helper.**cleanup_dir**()
Removes files from meshing

pydfnworks.mesh_dfn_helper.**output_meshing_report**(*visual_mode*)
Prints information about the final mesh to file

pydfnworks.mesh_dfn_helper.**parse_params_file**()
Reads params.txt file and parse information

Returns:

- num_poly: (int) Number of Polygons
- h: (float) meshing length scale h
- dudded_points: (int) Expected number of dudded points in Filter (LaGriT)
- visual_mode: True/False
- domain: dict: x,y,z domain sizes

5.6.2 Print legal statement

pydfnworks.legal.**legal**(*self*)
Print the legal LANL statement for dfnWorks.

5.6.3 Other helper methods

pydfnworks.helper.**commandline_options**()
Read command lines for use in dfnWorks.

Options:

- -name : Jobname (Mandatory)
- -ncpu : Number of CPUS (Optional, default=4)
- -input : input file with paths to run files (Mandatory if the next three options are not specified)
- -gen : Generator Input File (Mandatory, can be included within this file)
- -flow : PFLORAN Input File (Mandatory, can be included within this file)
- -trans: Transport Input File (Mandatory, can be included within this file)
- -cell: True/False Set True for use with cell based aperture and permeabuility (Optional, default=False)
- -large_network: True/False Set True to use CPP for file processing bottleneck (Optional, default=False)

pydfnworks.helper.**dump_time**(*local_jobname*, *section_name*, *time*)
keeps log of cpu run time, current formulation is not robust

pydfnworks.helper.**get_num_frac**()
Get the number of fractures from the params.txt file.

class pydfnworks.helper.**input_helper**(*params*, *minFracSize*)
Functions to help parse the input file and check input parameters.

Attributes:

- params (list): list of parameters specified in the input file.
- minFracSize (float): the minimum fracture size.

checkFamCount()
Makes sure at least one polygon family has been defined in nFamRect or nFamEll OR that there is a user input file for polygons.

checkMean(*minParam*, *maxParam*, *meanParam*, *warningFile=''*)
Warns the user if the minimum value of a parameter is greater than the family's mean value, or if the maximum value of the parameter is less than the family's mean value.

checkMinFracSize(*valList*)
Corrects the minimum fracture size if necessary, by looking at the values in valList.

checkMinMax(*minParam*, *maxParam*, *shape*)
Checks that the minimum parameter for a family is not greater or equal to the maximum parameter.

curlyToList(*curlyList*)
'{1,2,3}' –> [1,2,3]

error(*errString*)
print an error

Args: errString (str): a string describing the error

extractParameters(*line*, *inputIterator*)
Returns line without comments or white space.

findKey(*line*, *unfoundKeys*, *warningFile*)
Input: line containing a paramter (key) preceding a ":"

Returns:

- key – if it has not been defined yet and is valid
- None – if key does not exist
- exits – if the key has already been defined to prevent duplicate confusion

findVal(*line*, *key*, *inputIterator*, *unfoundKeys*, *warningFile*)
Extract the value for key from line.

getGroups(*line*, *valList*, *key*)
extract values between { and }

hasCurlys(*line*, *key*)
Checks to see that every { has a matching }.

isNegative(*num*)
"returns True if num is negative, false otherwise

listToCurly(*strList*)
[1,2,3] –> '{1,2,3}' for writing output

processLine(*line*, *unfoundKeys*, *inputIterator*, *warningFile*)
Find the key in a line, and the value for that key.

scale(*probList*, *warningFile*)
scales list of probabilities (famProb) that doesn't add up to 1 ie [.2, .2, .4] –> [0.25, 0.25, 0.5]

valHelper(*line*, *valList*, *key*)
pulls values from culry brackets

valueOf(*key*, *writing=False*)
Use to get key's value in params. writing always false

verifyFlag(*value*, *key=''*, *inList=False*)
Verify that value is either a 0 or a 1.

verifyFloat(*value*, *key=''*, *inList=False*, *noNeg=False*)
Verify that value is a positive float.

verifyInt(*value*, *key=''*, *inList=False*, *noNeg=False*)
Verify that value is a positive integer.

verifyList(*valList*, *key*, *verificationFn*, *desiredLength*, *noZeros=False*, *noNegs=False*)
verifies input list that come in format {0, 1, 2, 3}

Input:

- valList - list of values (flags, floats, or ints) corresponding to a parameter
- key - the name of the parameter whose list is being verified
- verificationFn - (either verifyflag, verifyfloat or verifyint) checks each list element
- desiredLength - how many elements are supposed to be in the list
- noZeros - (optional) True for lists than cannot contain 0's, false if 0's are ok
- noNegs - (optional) True for lists than cannot contain negative numbers, false otherwise

Output:

- returns negative value of list length to indicate incorrect length and provide meaningful error message
- prints error and exits if a value of the wrong type is found in the list
- returns None if successful

warning(*warnString*, *warningFile=''*)
print a warning to a file (currently does not work)

zeroInStdDevs(*valList*)
returns True is there is a zero in valList of standard deviations

pydfnworks.helper.**print_run_time**(*local_jobname*)
Read in run times from file and and print to screen with percentages

CHAPTER

SIX

SCRIPTS

The pydfnworks package has three Python scripts that assist with compiling, testing, and running the software. These scripts are all in the folder dfnworks-main/pydfnworks/bin/ .

6.1 compile.py: compile dfnWorks components

The compile.py script is called by run.py, but can also be called on its own. This script compiles the C and C++ components of dfnWorks. Without arguments, the script performs the compiling. With the argument 'clean,' the script cleans up C and C++ object files, before compiling.

6.2 test.py: test dfnWorks

The benchmark.py script can be used to easily test functionality of dfnWorks. It runs a benchmark suite (input files contained in the benchmarks folder) that test the exponential, power law, and lognormal distributions as well as user-defined ellipses and user-defined rectangle fracture inputs.

6.3 run.py: run dfnWorks

The run.py script allows the user to execute a single run of dfnWorks. The syntax for this script is:

python run.py -name [JOBNAME] -input [INPUT_FILE_NAME] -ncpu [NUMBER OF CPUS TO USE] -input [INPUT_FILE_NAME] -large_network

Please refer to the tutorial section (the next section for a detailed description of how to run dfnWorks.

CHAPTER

SEVEN

DFNWORKS TEST CASE TUTORIAL

This document contains a short, five example, tutorial for dfnWorks. The five test cases provided are for:

- 4_user_defined_rectangles: The user defines the precise location and size of four rectangular fractures.
- 4_user_defined_ellipses: The user defines the precise location and size of four elliptical fractures, where the ellipses are approximated as polygons.
- exponential_dist: The user specifies the parameters for two families of fractures with an exponential distribution of fracture size.
- lognormal_dist: The user specifies the parameters for two families of fractures with a lognormal distribution of fracture size.
- truncated_power_law_dist: The user specifies the parameters for two families of fractures with a truncated power-law distribution of fracture size.

All required input files for these examples are contained in the folder dfnworks-main/tests. The focus of this document is to provide visual confirmation that new users of dfnWorks have the code set up correctly, can carry out the following runs and reproduce the following images. All images are rendered using Paraview, which can be obtained for free at http : //www.paraview.org/. The first two examples are simpler than the last three so it is recommended that the user proceed in the order presented here.

7.1 Setting the paths correctly

Before executing dfnWorks, the following paths must be set. These are in the file dfnworks-main/pydfnworks/bin/run.py:

- PETSC_DIR and PETSC_ARCH: PETSC environmental variables
- PFLOTRAN_DIR: The PFLOTRAN directory
- python_dfn: The location of the Python distribution to use
- lagrit_dfn: The location of the LaGriT executable

7.2 Executing dfnWorks

To run one of the test cases enter either of the following types of command (INPUT PARAMETERS WILL CHANGE FOR ACTUAL RUNS). Both of the scripts invoked below are in the directory dfnworks-main/pydfnworks/bin/ :

- python test.py [JOBNAME], where name is one of the names above.
- python run.py -name [JOBNAME] -input [INPUT_FILE] -ncpus [NUMBER_OF_CPUS] -large_network

The second way of running dfnWorks can be used for any input, not only the examples presented here. The arguments are:

-[JOBNAME]: The name of the run, which is also the folder which will contain the run's output. -[INPUT_FILE]: An input file with three lines that have input files for dfnGen, dfnFlow, and dfnTrans, respectively. Any of the files with ending .txt in the directory tests can be used as examples of input files. -[NUMBER_OF_CPUS]: The number of CPUs that the user would like to use for the parralel computation of the meshing and flow solutions. -large_network (optional): Only use this flag if the user should use CPP for file processing.

For example, to run the demo lognormal on 4 CPUs, using C++ for file processing, the command line input would be either:

python test.py lognormal_dist

OR

python run.py -name lognormal_dist -input dfnworks-main/tests/lognormal_distribution.txt -ncpus 4 -large_network

Both of these command line inputs will run the lognormal_dist test and create a new folder lognormal_dist where all output files will be located. Descriptions of each output file are in the documentation. The only difference between the command line inputs above is that in the first, parameters such as number of CPUs and the input file are specified in the script test.py, while the second allows the user to specify these parameters on the command line. In the following sections, we provide descriptions of the output you should expect for each of the five examples.

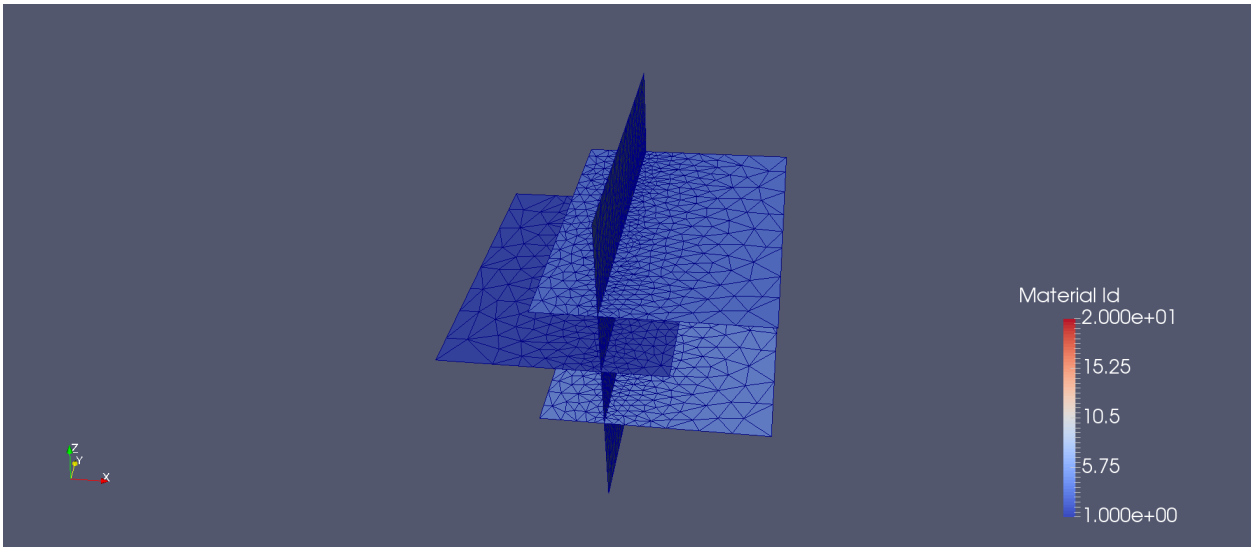
7.3 4_user_defined_rectangles

This test case consists of four user defined rectangular fractures within a a cubic domain with sides of length one meter. The input file specifiying the ellipses is in dfnworks-main/tests, and is named define_4_user_rectangles.dat. To run the test on 4 cpus, enter the following command line input:

python dfnworks-main/pydfnworks/bin/run.py -name 4_user_defined_rectangles -input dfnworks_main/tests/4_user_rectangles.txt -ncpus 4

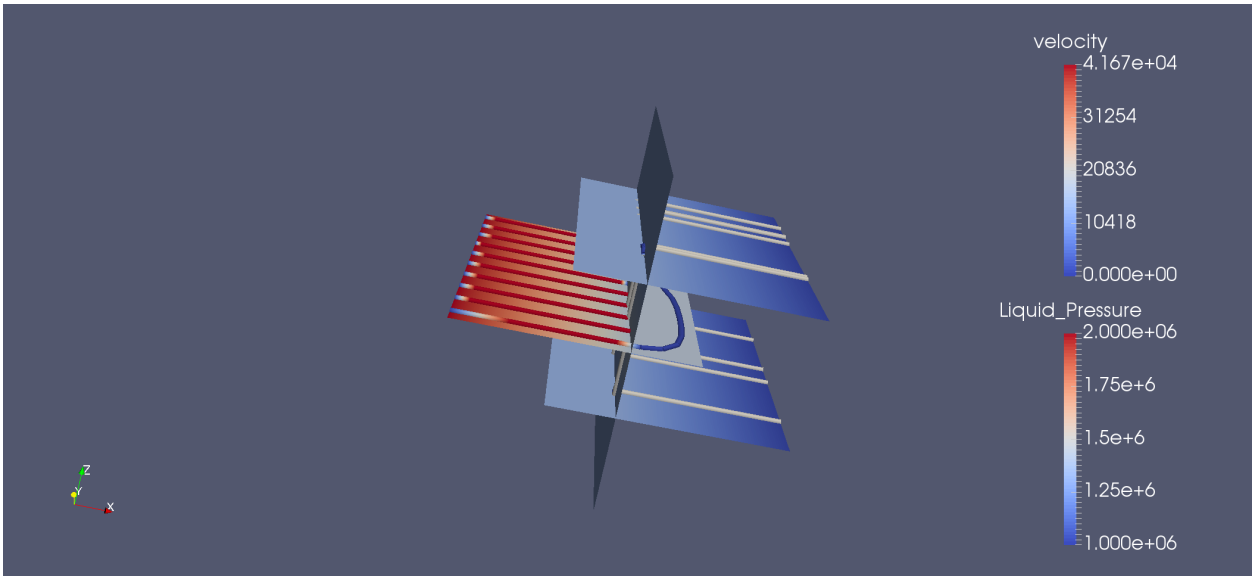
This will create a new folder, test 4fractures, where all of the output will be located. You can compare your results to the following images.

The network of four fractures, each colored by material ID. The computational mesh is overlaid on the fractures. This image is created by loading the file full_mesh.inp. located in the folder 4_user_defined_rectangles/LaGriT/, into Paraview.



The network of four fractures, colored by pressure solution. High pressure (red) Dirichlet boundary conditions are applied on the edge of the single fracture along the boundary x = -0.5, and low pressure (blue) boundary conditions are applied on the edges of the two fractures at the boundary x = 0.5. This image is created by loading the file 4_user_defined_rectangles/PFLOTRAN/parsed_vtk/dfn_explicit-001.vtk into Paraview.

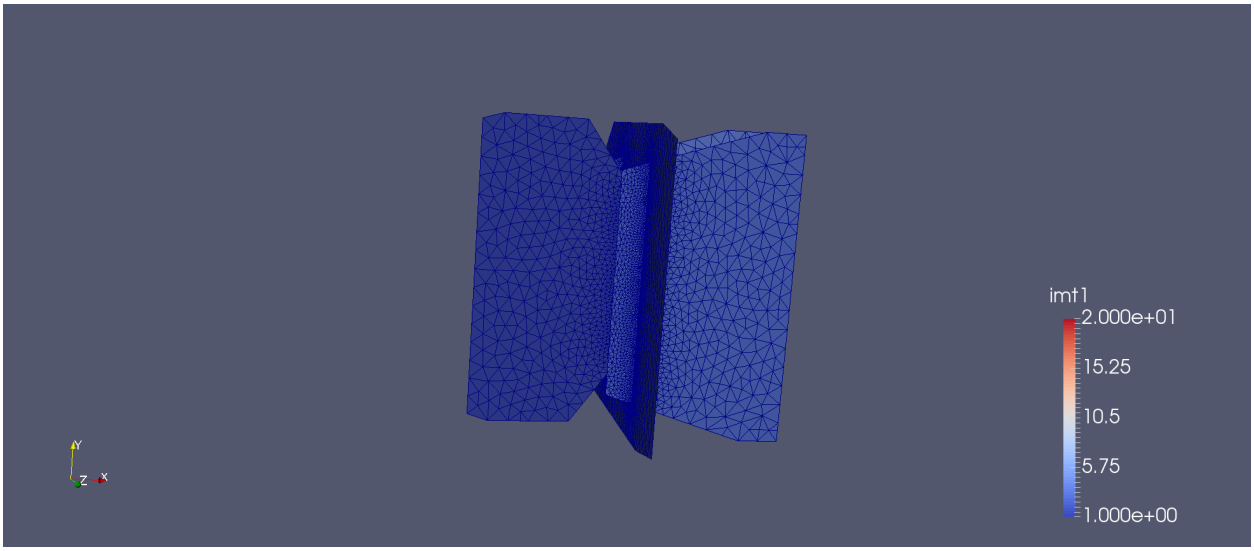
Particle trajectories on the network of four fractures. Particles are inserted uniformly along the inlet fracture on the left side of the image. Particles exit the domain through the two horizontal fractures on the right side of the image. Due to the stochastic nature of the particle tracking algorithm, your pathlines might not be exactly the same as in this image. Trajectories are colored by the current velocity magnitude of the particle's velocity. Trajectories can be visualized by loading the files part_*.inp, in the folder 4_user_rectangles/dfnTrans/trajectories/ We have used the extract surface and tube filters in paraview for visual clarity.

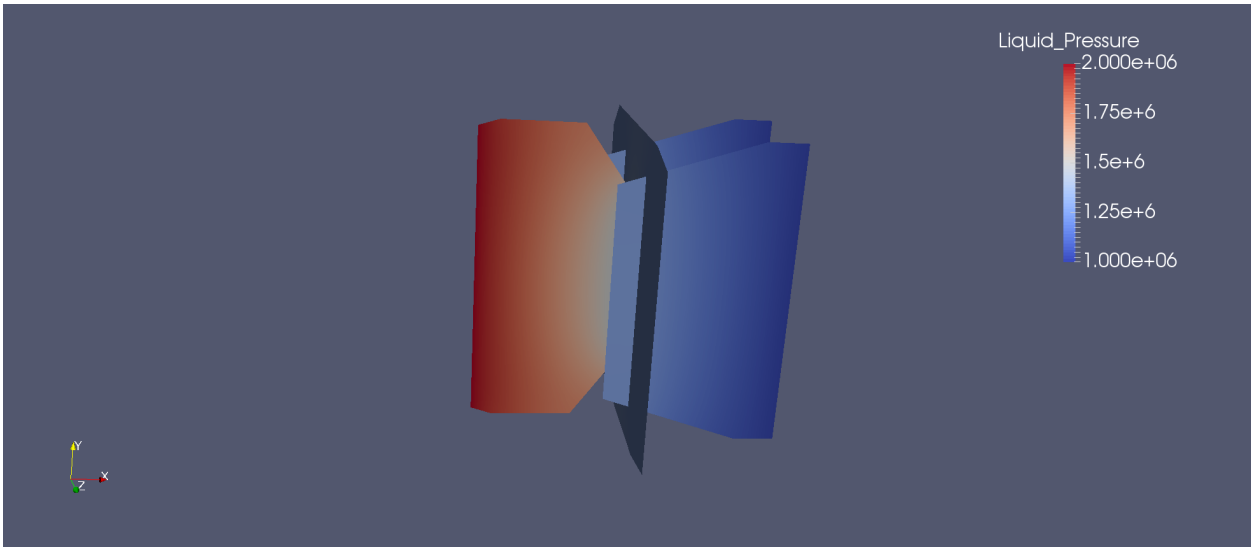


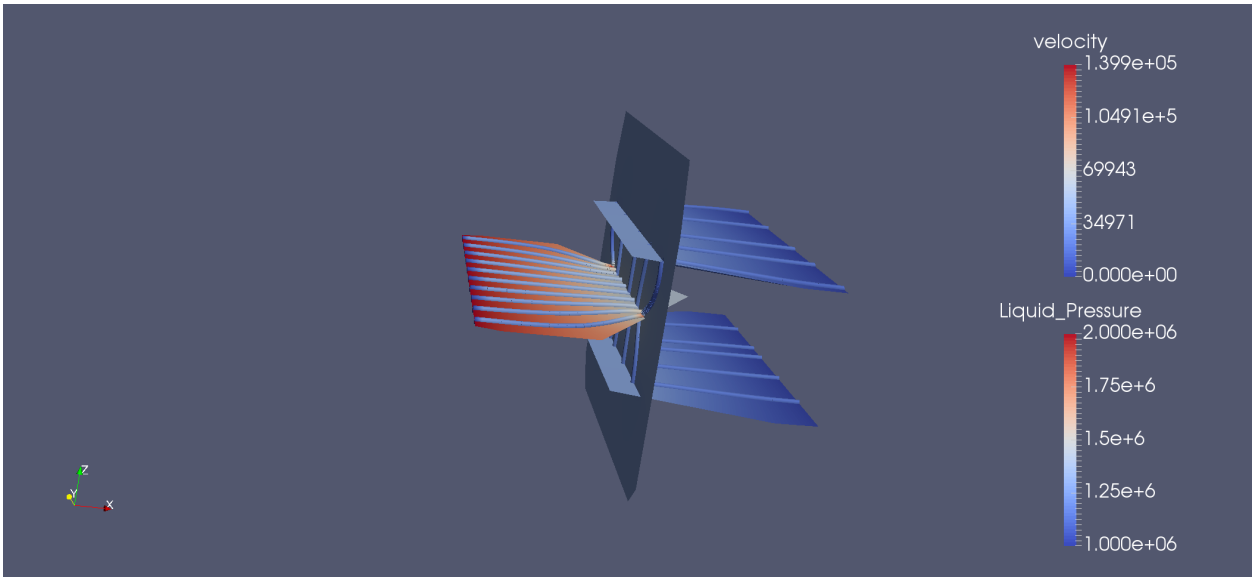
In the other tests, only a brief description and pictures are provided.

7.4 4_user_defined_ellipses

This test case consists of four user defined elliptical fractures within a a cubic domain with sides of length one meter. In this case the ellipses are approximated using 5 vertices. The input file specifiying the ellipses is in dfnworks-main/tests, and is named define_4_user_ellipses.dat.

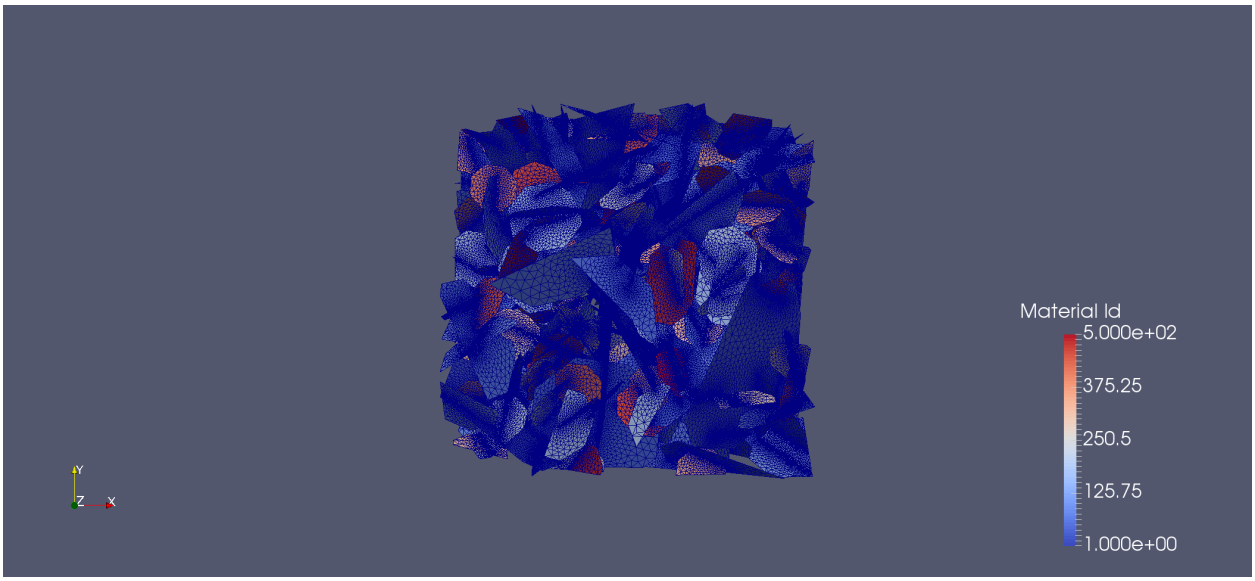


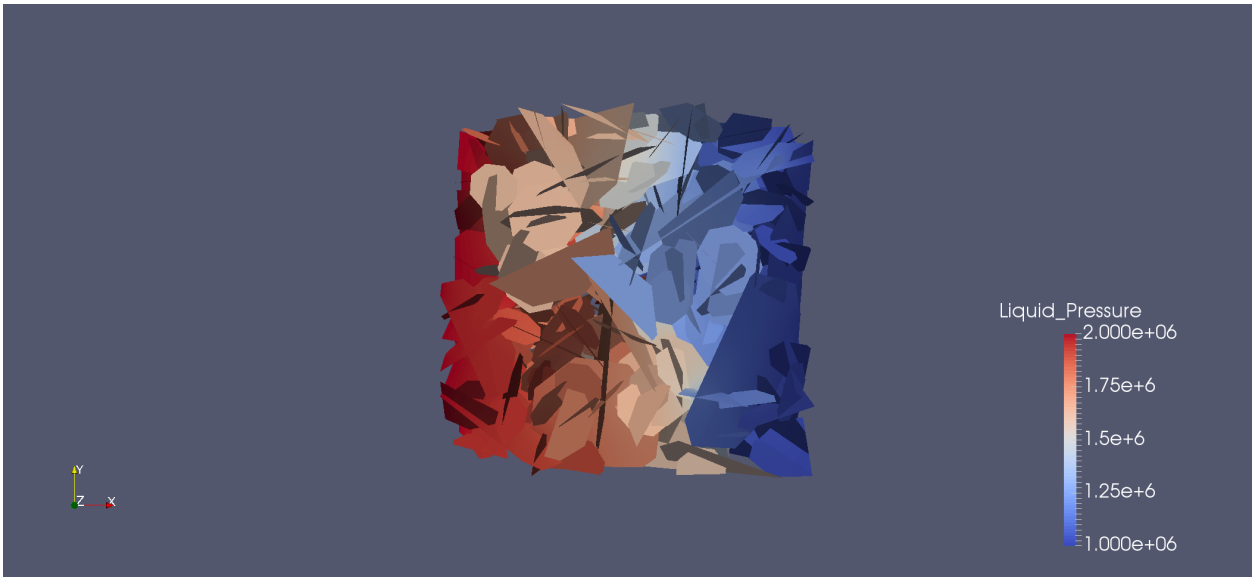


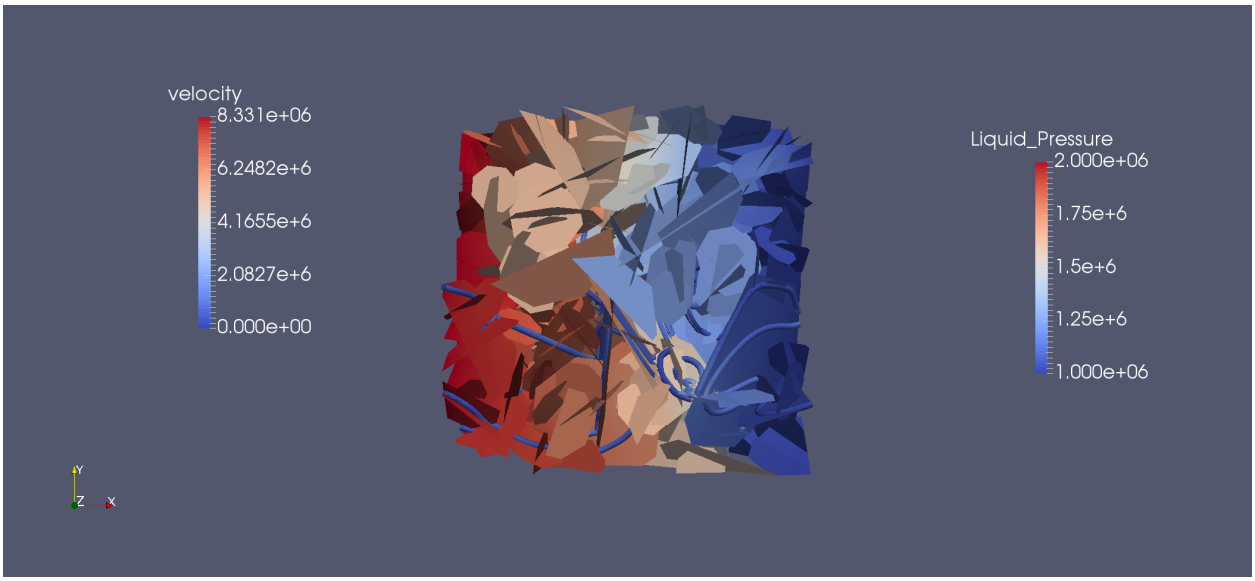


7.5 truncated_power_law_dist

This test case consists of two families whose sizes have a truncated power law distribution with a minimum size of 0.5m and a maximum size of 50m. The domain size is cubic with an edge length of 4m. The other input parameters can be found in tests/gen_truncated_power_law_dist.dat.

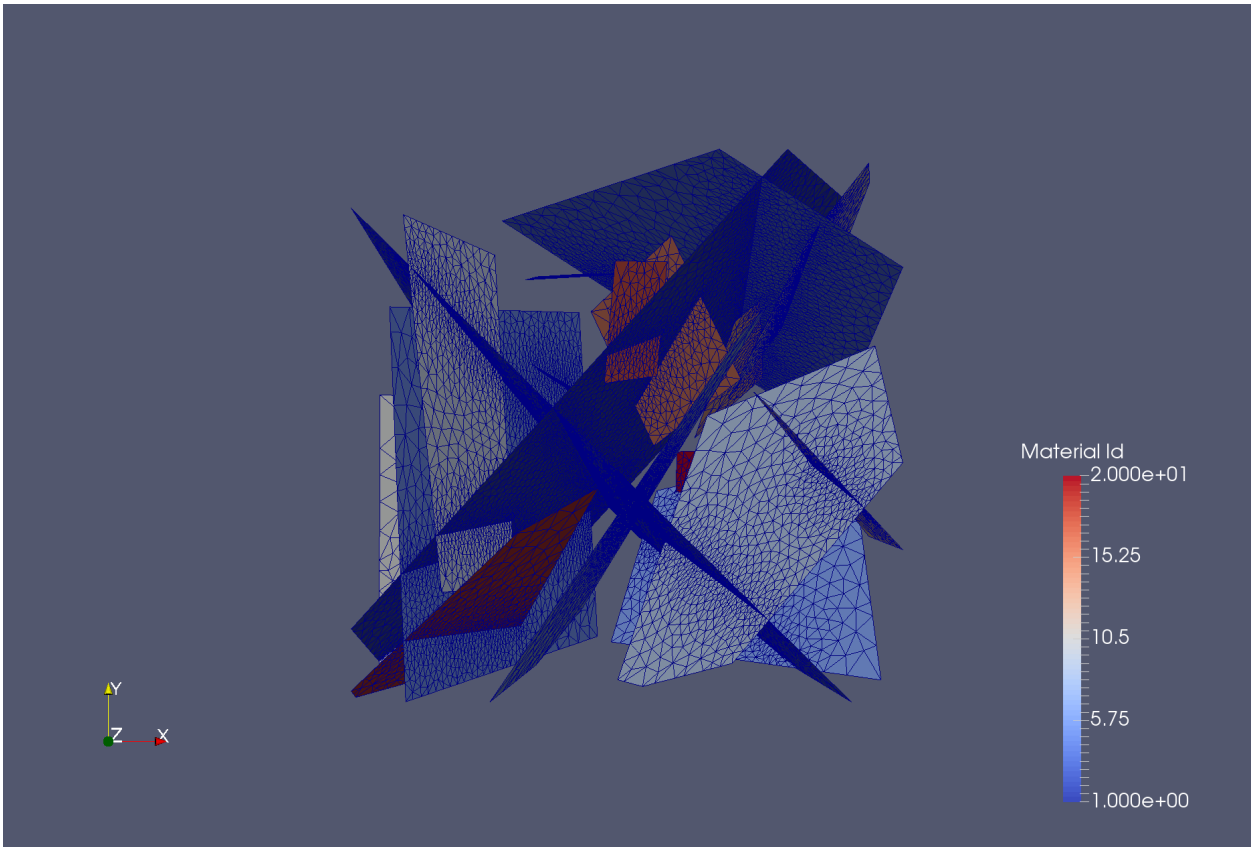


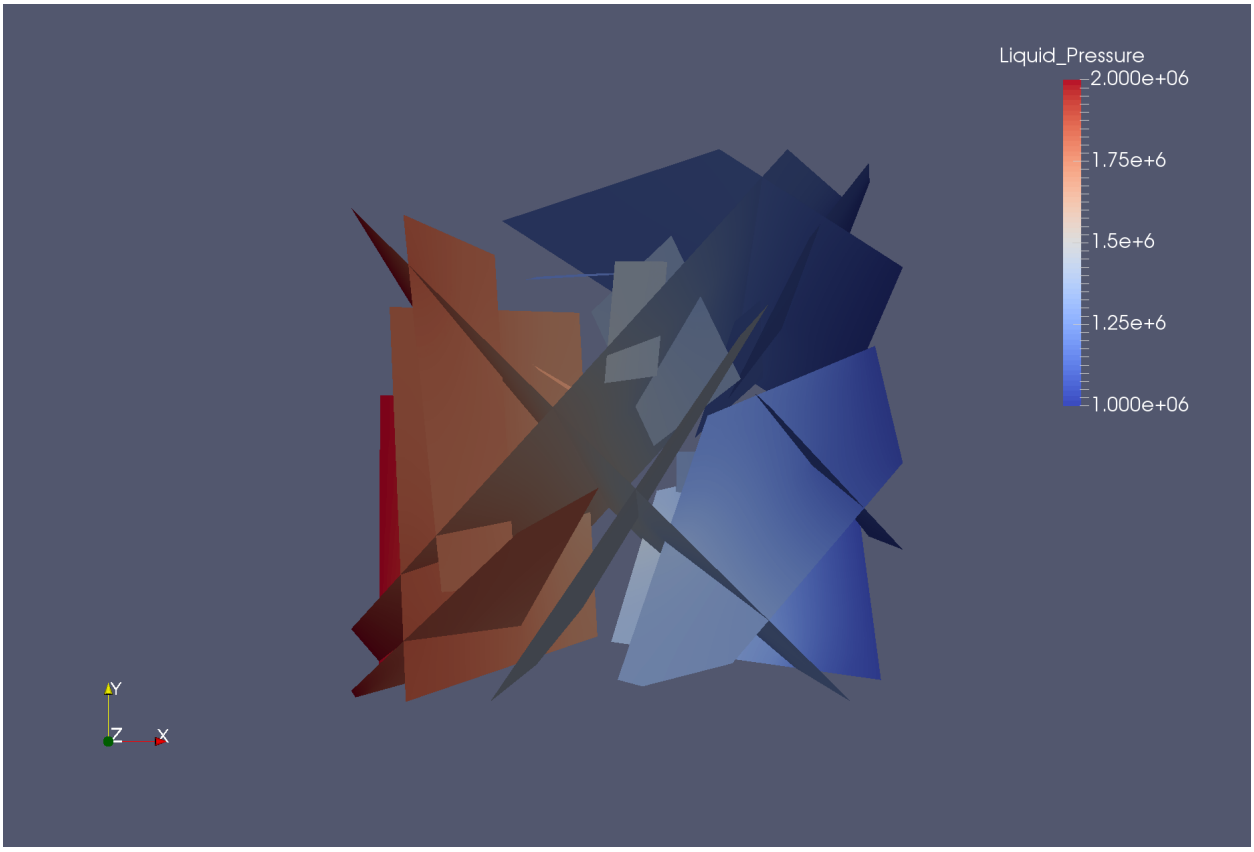


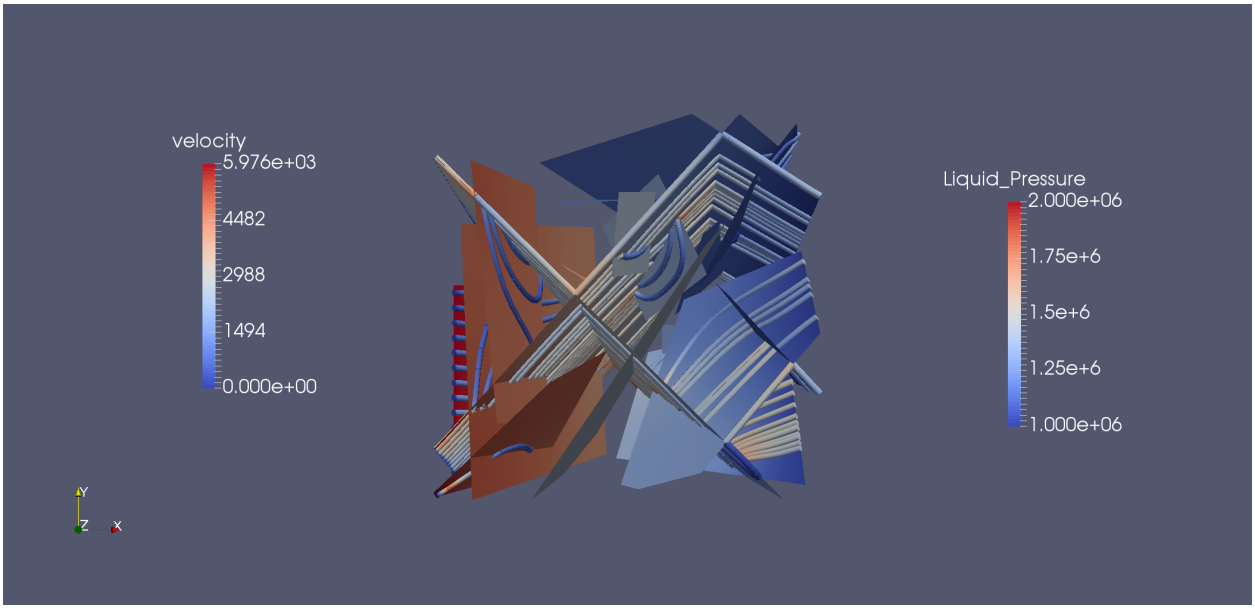


7.6 exponential_dist

This test case consists of a family of fractures whose size is exponentially distributed with a minimum size of 1m and a maximum size of 50m. The domain is cubic with an edge length of 10m. All input parameters for the generator can be found in tests/gen_exponential_dist.dat.

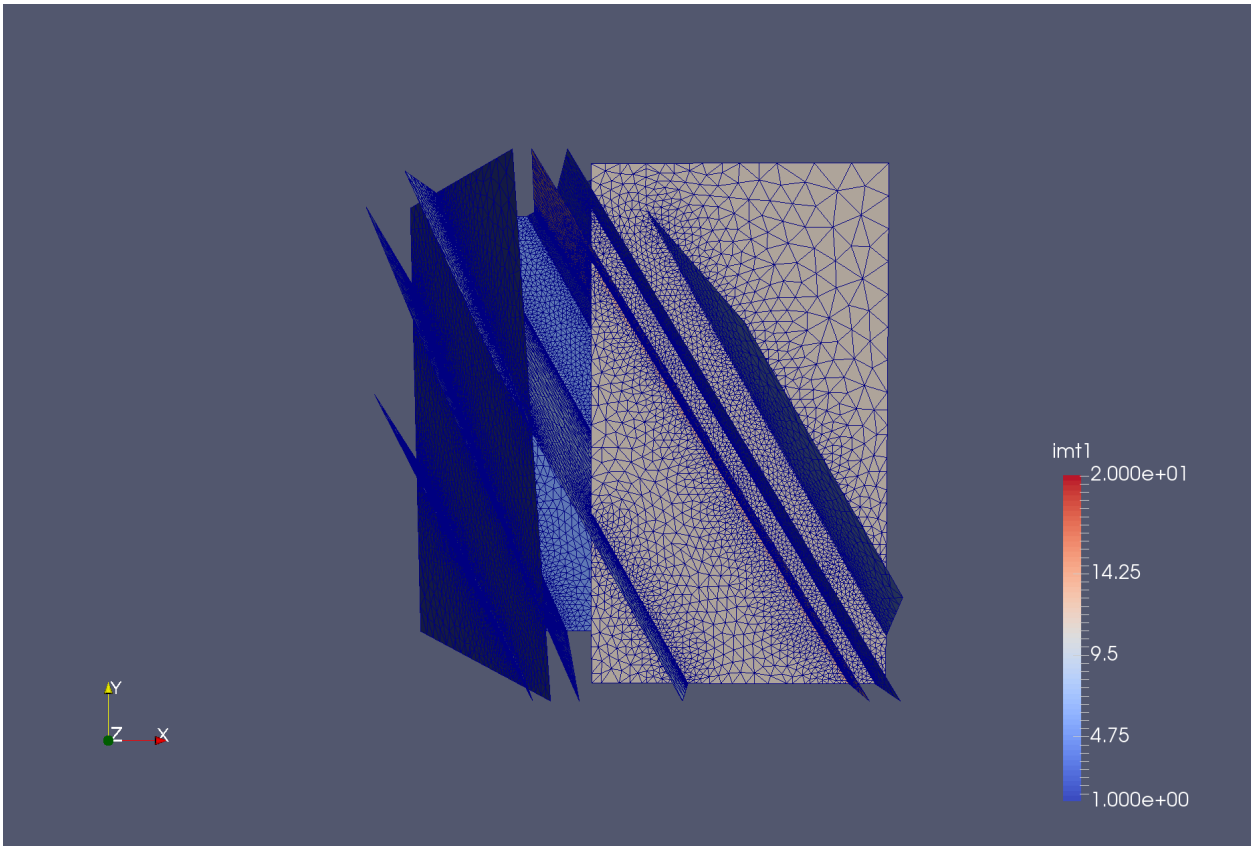

Material Id
2.000e+01
15.25
10.5
5.75
1.000e+00
Y
Z
X

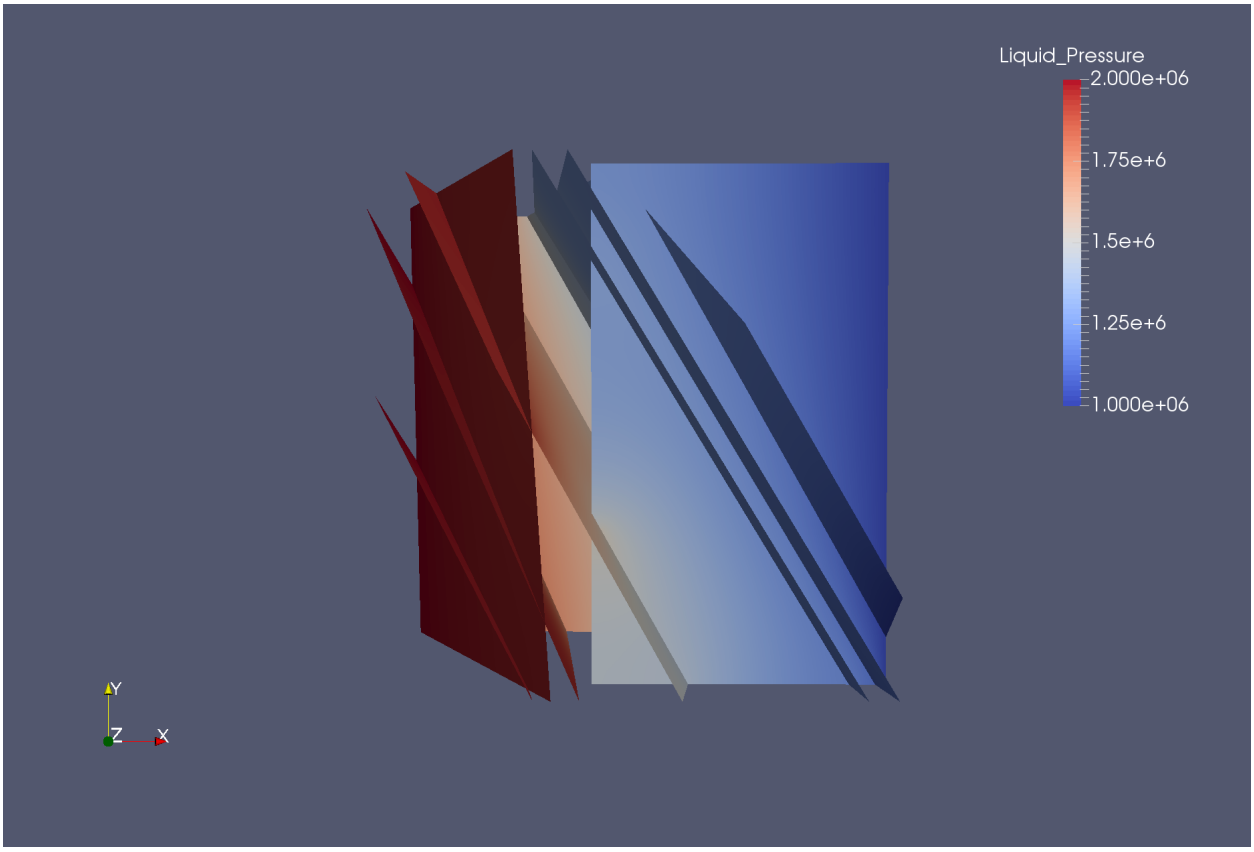

Liquid_Pressure
2.000e+06
1.75e+6
1.5e+6
1.25e+6
1.000e+06
Y
Z
X

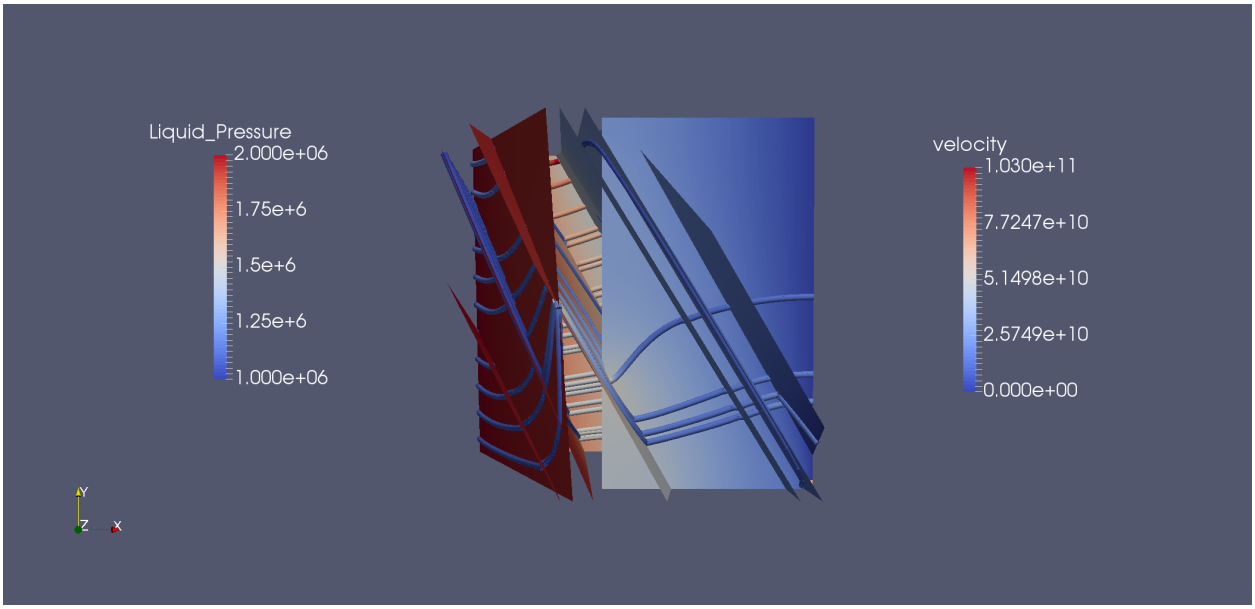

velocity
5.976e+03
4482
2988
1494
0.000e+00
Liquid_Pressure
2.000e+06
1.75e+6
1.5e+6
1.25e+6
1.000e+06
Y
Z
X

7.7 lognormal_dist

This test case consists of two fracture families whose sizes have a lognormal distribution with a minimum size of 0.5m and a maximum size of 50m. The domain size is cubic with an edge length of 10m. All input parameters for the generator can be found in tests/gen_lognormal_dist.dat.




Liquid_Pressure
2.000e+06
1.75e+6
1.5e+6
1.25e+6
1.000e+06
Y
Z
X


Liquid_Pressure
2.000e+06
1.75e+6
1.5e+6
1.25e+6
1.000e+06
velocity
1.030e+11
7.7247e+10
5.1498e+10
2.5749e+10
0.000e+00
Y
Z
X

CHAPTER

EIGHT

EXAMPLE APPLICATIONS

8.1 Carbon dioxide sequestration

DFNWORKS provides the framework necessary to perform multiphase simulations (such as flow and reactive transport) at the reservoir scale. A particular application, highlighted here, is sequestering CO2 from anthropogenic sources and disposing it in geological formations such as deep saline aquifers and abandoned oil fields. Geological CO2 sequestration is one of the principal methods under consideration to reduce carbon footprint in the atmosphere due to fossil fuels (Bachu, 2002; Pacala and Socolow, 2004). For safe and sustainable long-term storage of CO2 and to prevent leaks through existing faults and fractured rock (along with the ones created during the injection process), understanding the complex physical and chemical interactions between CO2, water (or brine) and fractured rock, is vital. DFNWORKS capability to study multiphase flow in a DFN can be used to study potential CO2 migration through cap-rock, a potential risk associated with proposed subsurface storage of CO2 in saline aquifers or depleted reservoirs. Moreover, using the reactive transport capabilities of PFLOTRAN coupled with cell-based transmissivity of the DFN allows one to study dynamically changing permeability fields with mineral precipitation and dissolution due to CO2–water interaction with rock.

8.2 Shale energy extraction

Hydraulic fracturing (fracking) has provided access to hydrocarbon trapped in low-permeability media, such as tight shales. The process involves injecting water at high pressures to reactivate existing fractures and also create new fractures to increase permeability of the shale allowing hydrocarbons to be extracted. However, the fundamental physics of why fracking works and its long term ramifications are not well understood. Karra et al. (2015) used DFNWORKS to generate a typical production site and simulate production. Using this physics based model, they found good agreement with production field data and determined what physical mechanisms control the decline in the production curve.

8.3 Nuclear waste repository

The Swedish Nuclear Fuel and Waste Management Company (SKB) has undertaken a detailed investigation of the fractured granite at the Forsmark, Sweden site as a potential host formation for a subsurface repository for spent nuclear fuel (SKB, 2011; Hartley and Joyce, 2013). The Forsmark area is about 120 km north of Stockholm in northern Uppland, and the repository is proposed to be constructed in crystalline bedrock at a depth of approximately 500 m. Based on the SKB site investigation, a statistical fracture model with multiple fracture sets was developed; detailed parameters of the Forsmark site model are in SKB (2011). We adopt a subset of the model that consist of three sets of background (non-deterministic) circular fractures whose orientations follow a Fisher distribution, fracture radii are sampled from a truncated power-law distribution, the transmissivity of the fractures is estimated using a power-law model based on the fracture radius, and the fracture aperture is related to the fracture size using the cubic law (Adler et al., 2012). Under such a formulation, the fracture apertures are uniform on each fracture, but vary among fractures.

The network is generated in a cubic domain with sides of length one-kilometer. Dirichlet boundary conditions are imposed on the top (1 MPa) and bottom (2 MPa) of the domain to create a pressure gradient aligned with the vertical axis, and noflow boundary conditions are enforced along lateral boundaries.

Sources:

- Adler, P.M., Thovert, J.-F., Mourzenko, V.V., 2012. Fractured Porous Media. Oxford University Press, Oxford, United Kingdom.
- Bachu, S., 2002. Sequestration of CO2 in geological media in response to climate change: road map for site selection using the transform of the geological space into the CO2 phase space. Energy Convers. Manag. 43, 87–102.
- Hartley, L., Joyce, S., 2013. Approaches and algorithms for groundwater flow modeling in support of site investigations and safety assessment of the Fors- mark site, Sweden. J. Hydrol. 500, 200–216.
- Karra, S., Makedonska, N., Viswanathan, H., Painter, S., Hyman, J., 2015. Effect of advective flow in fractures and matrix diffusion on natural gas production. Water Resour. Res., under review.
- Pacala, S., Socolow, R., 2004. Stabilization wedges: solving the climate problem for the next 50 years with current technologies. Science 305, 968–972.
- SKB, Long-Term Safety for the Final Repository for Spent Nuclear Fuel at Forsmark. Main Report of the SR-Site Project. Technical Report SKB TR-11-01, Swedish Nuclear Fuel and Waste Management Co., Stockholm, Sweden, 2011.

l

m

p

r

B

C

D

E

F

G

H

I

L

M

O

P

R

S

T

V

W

Z