dfnWorks Documentation

Release 2.0

Subsurface Flow and Transport Team, LANL, LA-UR-17-22216

April 13, 2017

Contents:

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 with the aim 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 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 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 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 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/download/

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

Documentation:

LA-UR-17-22216

Software copyright:

LA-CC-17-027

Copyright (2017). Los Alamos National Security, LLC. This material was produced under U.S. Government contract DE-AC52-06NA25396 for Los Alamos National Laboratory (LANL), which is operated by Los Alamos National Security, LLC for the U.S. Department of Energy. The U.S. Government has rights to use, reproduce, and distribute this software. NEITHER THE GOVERNMENT NOR LOS ALAMOS NATIONAL SECURITY, LLC MAKES ANY WARRANTY, EXPRESS OR IMPLIED, OR ASSUMES ANY LIABILITY FOR THE USE OF THIS SOFTWARE. If software is modified to produce derivative works, such modified software should be clearly marked, so as not to confuse it with the version available from LANL.

Additionally, this program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version. Accordingly, this program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

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: 400
/*  Used when stopCondition = 0 (nPoly option).

    The total number of fractures you would
    like to have in the DFN.
*/


domainSize: {10,10,10}
/*  Mandatory Parameter.
    Creates a domain with dimension x*y*z centered at the origin.
*/


numOfLayers: 2
//  Number of layers


layers:
{-25,0}
{0,25}
/*  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: .1
/*  Minimum fracture length scale(meters).
    Any fracture with a feature, such as and intersection,
    of less than h will be rejected.
```

```
*/


radiiListIncrease: 0.05
/*  Percent to increase the size of the pre-generated radii lists, per family.
    Example: 0.2 will increase the size of the list by %20.

    Lists are pre-generated and are used to insert fractures from largest to
    smallest in order to help hit target distributions.

    If the accepted+rejected count of fractures is much different than the
    estimated number of fractures needed with the percentage extra added in,
    you will not get your expected distribution.
*/



removeFracturesLessThan: 0
/*
    Options:
        0 - Ignore this option, keep all fractures.

        Size of minimum fracture radius. Fractures smaller than
        defined radius will be removed AFTER DFN generation.

        Minimum and maximum size options under fracture family
        distributions will still be used while generating the DFN.
*/



/*=============================================================================*/
/*  File Output Options:                                                       */
/*=============================================================================*/

/* An output file (radii.dat) is always generated and contains all radii of
   fractures in the domain (before and after truncation).
*/

outputAllRadii: 0
/*  Caution: Can create very large files.
    Outputs all fractures which were generated during
    DFN generation (Accepted + Rejected).
        0: Do not output all radii file.
        1: Include file of all raddii, acepted + rejected fractures,
           in output files (radii_All.dat).
*/


outputFinalRadiiPerFamily: 1
/*  Outputs radii files after isolated fracture removal.
    One file per family.
        0: Do not create output files of radii per family
        1: Creates output files per family, containing a list
           of the family's fracture radii that is in the final DFN
*/

outputAcceptedRadiiPerFamily: 1
```

```
/*  Outputs radii files before isolated fracture removal.
    One file per family.
        0: Do not create output files of radii per family
        1: Creates output files per family, containing a list
           of the family's fracture radii in the domain before isolated
           fracture removal.
*/




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

tripleIntersections: 1
/*  Accept or reject triple intersections
        0 - Off (Reject)
        1 - On  (Accept)
*/


forceLargeFractures: 1
/*  Inserts the largest possible fracture for each defined fracture family,
    defined by the user-defined maxium radius
        0 - Off (Do not force insertion of larest fractures)
        1 - On  (Force insertion of largest fractures)
*/

printRejectReasons: 0
/*  Useful for debugging,
    This option will print all fracture rejection reasons as they occur.
        0 - Disable
        1 - Print all rejection reasons to screen
*/

disableFram: 0
/*  This option disables the FRAM algorithm. There will be no
    fracture rejections or fine mesh. Defaults visualizationMode to 1.
        0 - Enable  FRAM
        1 - Disable FRAM

*/

visualizationMode: 0
/*  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: 0
/*  Seed for random generator.
    Enter 0 for time based seed.
*/

domainSizeIncrease: {0,0,0}
/*  Size increase for inserting fracture centers outside the domain.
    Fracture will be truncated based on domainSize above.
```

```
    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.

    If ignoreBoundaryFaces is also set to 1, dfnGen will keep the largest
    cluster which connects at least any two sides of the domain.
*/

ignoreBoundaryFaces: 1
/*
     0 - Use boundaryFaces option below
     1 - Ignore boundaryFaces option and keep all clusters,
         and remove fractures with no intersections
*/

boundaryFaces: {1,1,1,1,1,1}
/*  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: 350
/*  If a fracture is rejected, it will be re-translated
    to a new position this number of times.

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


insertUserRectanglesFirst: 1
/*  0 - User ellipses will be inserted first
    1 - User rectangles will be inserted first
*/

/*============================================================================*/
/*  Probability Parameters                                                    */
/*============================================================================*/

famProb: {.33,.33,.34}
/*  Each element is the probability of chosing a fracture from
    the element's corresponding family to be inserted into the DFN.
```

```
    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}, famProb elements must add to 1
*/




/*=============================================================================*/
/*=============================================================================*/
/*                                                                           */
/*  Elliptical Fracture Options                                              */
/*  NOTE: Number of elements must match number of ellipse families           */
/*                                                                           */
/*=============================================================================*/
/*=============================================================================*/


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

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

edistr: {2,3}
/*  Mandatory parameter if using statistically generated ellipses.
    Statistical distribution options:
        1 - Log-normal distribution
        2 - Truncated power law distribution
        3 - Exponential distribution
        4 - Constant
*/

ebetaDistribution: {0,0}
/*  Beta is the rotation around the polygon's normal vector
        0 - Uniform distribution [0, 2PI)
        1 - Constant angle (specefied below by 'ebeta')
*/

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



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

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

easpect: {1.1,1.2}
//  Aspect ratio for stochastic ellipses.

enumPoints: {8, 12}
/*  Number of vertices used in creating each elliptical
    fracture family. Number of elements must match number
    of ellipse families
*/

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

etheta: {0, 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 fractures' normal vector.

ekappa: {8,10}
/*  Parameter for the fisher distribnShaprutions. The
    bigger, the more similar (less diverging) are the
    elliptical familiy's normal vectors.
*/



/*============================================================================*/
/*  Ellipse Log-normal Distribution Options (edistr = 1)                      */
/*  Mandatory Parameters if using ellispes with log-normal distribution       */
/*============================================================================*/

eLogMean: {4}
//  Mean of the underlying normal distribution

esd: {1}
//  Standard deviation of the underlying normal distribution

eLogMin: {1}
//  Minimum radius

eLogMax: {15}
//  Maximum radius
```

```
/*============================================================================*/
/*  Ellipse Exponential Distribution Options (edistr = 3)                     */
/*  Mandatory Parameters if using ellipses with exponential distribution      */
/*============================================================================*/


eExpMean: {2}
//  Mean values for exponential distributions, defined per family.


eExpMin: {1}
//  Minimum radius, defined per family.


eExpMax: {10}
//  Maximum radius, defined per family.



/*============================================================================*/
/*  Ellipse Constant Size Option (edistr=4)                                   */
/*  Mandatory Parameters if using ellipses with constant size.                */
/*============================================================================*/


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



/*============================================================================*/
/*  Ellipse Truncated Power-Law Distribution Options (edistr=2)               */
/*  Mandatory Parameters if using ellipses with power-law distributions.      */
/*============================================================================*/


emin: {1}
//  Minimum radius


emax: {6}
//  Maximum radius


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



/*============================================================================*/
/*============================================================================*/
/*                                                                            */
/*  Rctangular Fractures Options                                              */
/*  NOTE: Number of elements must match number of rectangle families          */
/*                                                                            */
/*============================================================================*/
/*============================================================================*/


nFamRect: 3
/*  Number of rectangular families defined below.
    Having this option = 0 will ignore all rectangular family variables.
*/


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

```
*/

rdistr: {3,3,3}
/* Mandatory parameter if using statistically generated rectangles.
    Rectangle statistical distribution options:
        1 - log-normal distribution
        2 - truncated power law distribution
        3 - exponential distribution
        4 - constant
*/

rbetaDistribution: {0,0,0}
/* Beta is the rotation around the polygon's normal vector
        0: Uniform distribution [0, 2PI)
        1: Constant angle (specefied below by 'rbeta')
*/

r_p32Targets: {.2,.2,.2}
/* Rectangular families target fracture intensities per family.
    When using stopCondition = 1 (defined at the top of the input file),
    families will be inserted until the families desired fracture
    intensity has been reached. Once all families desired fracture
    intensity has been met, fracture generation will complete.
*/



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

raspect: {1,1,1}
// Aspect ratio for stochasic rectangles.

rAngleOption: 1
/* All angles for rectangles are in:
        1 - Degrees
        0 - Radians (must be numerical value, cannot use "Pi")
*/

rtheta: {90,90,0}
/* Rectangle fracture orientation.
    The angle the normal vector makes with the z-axis
*/

rphi: {90,0, 0}
/* Rectangle fracture orientation.
    The angle the projection of the normal
    onto the x-y plane makes with the x-axis
*/

rbeta: {0,0,0}
// Rotation around the normal vector.

rkappa: {1,1,1}
/* Parameter for the fisher distribnShaprutions. The
    bigger, the more similar (less diverging) are the
    rectangular familiy's normal vectors.
```

```
*/


/*=============================================================================*/
// Rectangle Log-normal Distribution Options (rdistr = 1)                      */
// Mandatory Parameters if using rectangles with log-normal distribution       */
/*=============================================================================*/

rLogMean: {2,2}
//  Mean of the underlying normal distribution

rsd: {.3,.3}
//  Standard deviation of the underlying normal distribution

rLogMin: {5,5}
//  Minimum radius

rLogMax: {20,20}
//  Maximum radius



/*=============================================================================*/
// Rectangle Truncated Power-Law Distribution Options (rdistr=2)               */
// Mandatory Parameters if using rectangles with power-law distributions.      */
/*=============================================================================*/

rmin: {1}
//  Minimum radius

rmax: {10}
//  Maximum radius

ralpha: {2.4}
//  Alpha. Used in truncated power-law distribution calculations.



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

rExpMean: {10,10,10}
//  Mean radius

rExpMin: {6, 6, 3}
//  Minimum radius

rExpMax: {45, 45, 30}
//  Maximum radius



/*=============================================================================*/
/* Rectangle Constant Size of rectangles (edistr = 4)                          */
/*=============================================================================*/
rconst: {2,2,2}
//  Constant radius, defined per family.
```

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


userEllipsesOnOff: 0
/*  0 - User ellipses off
    1 - User ellipses on
*/


UserEll_Input_File_Path: /home/jharrod/GitProjects/DFNGen/DFNC++Version/inputFiles/userPolygons/uEll


/*==============================================================================*/
/*==============================================================================*/
/*  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: 1
/*  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/el


/*==============================================================================*/
/*==============================================================================*/
/*  User-Specified Rectangles                                                   */
/*  NOTE: Number of elements must match number of user-ellipse families         */
/*  NOTE: Only one user-rectangle is placed into the domain per defined         */
/*        user-rectangle, with possibility of being rejected                    */
/*==============================================================================*/
/*==============================================================================*/


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


UserRect_Input_File_Path: /home/jharrod/GitProjects/DFNGen/DFNC++Version/inputFiles/userPolygons/ignc


/*==============================================================================*/
/*==============================================================================*/
/*                                                                              */
/*  User Rectangles Defined By Coordinates                                      */
/*                                                                              */
```

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


userRecByCoord: 0
/*  0 - User defined rectangles by coordinates off
    1 - User defined rectangles by coordinates on
*/


RectByCoord_Input_File_Path: ~/GitProjects/DFNGen/DFNC++Version/inputFiles/userPolygons/rectCoords.da

/*  WARNING: This option can cause LaGriT errors if the polygon
            vertices are not put in clockwise or counter-clockwise order.
            If errors (Usualy seg fualt during meshing in LaGriT),
            make sure the vertices are in clockwise or counter clockwise
            order. Also, coordinates must be co-planar.
*/



/*============================================================================*/
/*============================================================================*/
/*  Aperture Options                                                          */
/*  Mandatory Paramter                                                        */
/*  NOTE: Only one aperture type may be used at a time                        */
/*============================================================================*/
/*============================================================================*/


aperture: 3
/*  1 - Log-normal distribution
    2 - Aperture from transmissivity, first transmissivity is defined,
        and then, using a cubic law, the aperture is calculated.
    3 - Constant aperture (same aperture for all fractures)
    4 - Length Correlated Aperture
        Apertures are defined as a function of fracture size.
*/



/*============================================================================*/
/*  Log-normal Aperture Distribution Options (aperture = 1)                   */
/*============================================================================*/


meanAperture:  3
// Mean value

stdAperture: 0.8
// Standard deviation



/*============================================================================*/
/*  Aperture From Transmissivity Options (aperture = 2)                       */
/*============================================================================*/


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)
*/
```

```
/*==============================================================================*/
/*  Constant Aperture Option (aperture = 3)                                     */
/*==============================================================================*/


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


/*==============================================================================*/
/*  Length Correlated Aperture (aperture = 4)                                   */
/*==============================================================================*/

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                                                        */
/*  Mandaatory Parameter                                                        */
/*==============================================================================*/
/*==============================================================================*/


permOption: 1
/*  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
*/

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

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

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
```

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 compu

/********************** 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 rota

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 o
```

```
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 genera

/* flow field: 3D Darcy velocities: output ASCII file "Velocity3D" has an each nodes position and its
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 to
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 coordi
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
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 f
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 flo

/*** 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
```

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-Version2.0/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

argparse = <module 'argparse' from '/n/swdev/packages/Ubuntu-14.04-x86_64/anaconda-python/2.4.1/lib/python2.7/ar

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)

copy_dfn_trans_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.

dfn_flow()

> dfnFlow Run the dfnFlow portion of the workflow.

dfn_gen()

> **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

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

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

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

class 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.

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

    check_mean(*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.

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

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

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

    error(*errString*)
        print an error
        Args: errString (str): a string describing the error

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

    find_key(*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

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

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

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

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

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

process_line(*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]

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

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

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

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

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

verify_list(*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)

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

DFNWORKS.**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.

DFNWORKS.**legal**()
> Print the legal LANL statement for dfnWorks.

DFNWORKS.**make_working_directory**()
> make working directories for fracture generation

DFNWORKS.**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

DFNWORKS.**os = <module 'os' from '/n/swdev/packages/Ubuntu-14.04-x86_64/anaconda-python/2.4.1/lib/python2.7/os.py**

DFNWORKS.**output_report** (*radiiFile='radii.dat'*,   *famFile='families.dat'*,   *trans-File='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.

DFNWORKS.**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.

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

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

DFNWORKS.**pflotran_cleanup** ()
    Concatenate PFLOTRAN output files and then delete them

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

DFNWORKS.**re = <module 're' from '/n/swdev/packages/Ubuntu-14.04-x86_64/anaconda-python/2.4.1/lib/python2.7/re.py**

DFNWORKS.**run_dfn_trans** ()
    run dfnTrans simulation

DFNWORKS.**subprocess = <module 'subprocess' from '/n/swdev/packages/Ubuntu-14.04-x86_64/anaconda-python/2.4.1**

DFNWORKS.**sys = <module 'sys' (built-in)>**

DFNWORKS.**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

DFNWORKS.**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

beta_distribution(*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.

constant_dist(*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.

exponential_dist(*prefix*)

Verifies parameters for exponential distribution of fractures.

lognormal_dist(*prefix*)

Verifies all logNormal Parameters for ellipses and Rectangles.

tpl_dist(*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.**dfn_gen**(*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.**dfn_flow**(*self*)
    dfnFlow 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_dfn_trans_files**(*self*)

> create link to DFNTRANS and copy input file into local directory

pydfnworks.transport.**dfn_trans**(*self*)

> dfnTrans Copy input files for dfnTrans into working directory and run DFNTrans

pydfnworks.transport.**run_dfn_trans**(*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*, *h*, *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*, *re-
fine_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*, *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: A

>> •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)

`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.

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

`check_mean`(*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.

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

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

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

error(*errString*)
    print an error

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

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

find_key(*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

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

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

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

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

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

process_line(*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]

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

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

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

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

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

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

    Input:

5.6. Helper methods 37

- 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)

zero_in_std_devs(*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

SCRIPTS

The pydfnworks package has four Python scripts that assist with compiling, testing, and running the software. These scripts are all in the folder dfnWorks-Version2.0/pydfnworks/bin/ .

6.1 fix_paths.py: fix the paths in the tests folder

Syntax: python fix_paths.py

6.2 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.

6.3 test.py: test dfnWorks

The test.py script can be used to easily test functionality of dfnWorks. It runs a test suite (input files contained in the tests folder) that test the exponential, power law, and lognormal distributions as well as user-defined ellipses and user-defined rectangle fracture inputs. Syntax: python test.py [TEST_NAME_PART], where [TEST_NAME_PART] is all or part of the name of the tests the user wants to run.

6.4 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 [IN-PUT_FILE_NAME]

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

OUTPUT FILES

dfnWorks outputs about a hundred different output files. This section describes the contents and purpose of each file.

7.1 File locations

Here is a list of file locations after dfnWorks has finished running:

7.1.1 dfnworks-main

- *allboundaries.zone*
- *aperture.dat*
- *cellinfo.dat*
- *darcyvel.dat*
- *dfnTrans_output_dir*
- *params.txt*
- *PTDFN_control.dat*
- *pboundary_back_n.zone*
- *pboundary_bottom.zone*
- *pboundary_front_s.zone*
- *pboundary_left_w.zone*
- *pboundary_right_e.zone*
- *pboundary_top.zone*

7.1.2 dfnGen

- *connectivity.dat*
- *DFN_output.txt*
- *families.dat*
- *input_generator.dat*
- *input_generator_clean.dat*

- *normal_vectors.dat*
- *radii*
- *radii.dat*
- *radii_Final.dat*
- *rejections.dat*
- *rejectsPerAttempt.dat*
- *translations.dat*
- *triple_points.dat*
- *user_rects.dat*
- *warningFileDFNGen.txt*

7.1.3 LaGriT

- *bound_zones.lgi*
- *boundary_output.txt*
- *finalmesh.txt*
- *full_mesh.inp*
- *full_mesh.gmv*
- *full_mesh.lg*
- *intersections*
- *lagrit_logs*
- *logx3dgen*
- *outx3dgen*
- *parameters*
- *polys*
- *tri_fracture.stor*
- *user_function.lgi*

7.1.4 PFLOTRAN

- *aperture.dat*
- *cellinfo.dat*
- *dfn_explicit-000.vtk*
- *dfn_explicit-001.vtk*
- *dfn_explicit-mas.dat*
- *dfn_explicit.in*
- *dfn_explicit.out*
- *dfn_properties.h5*

- *full_mesh.uge*
- *full_mesh_viz.inp*
- *full_mesh_vol_area.uge*
- *materialid.dat*
- *parsed_vtk*
- *perm.dat*
- *pboundary_back_n.ex*
- *pboundary_bottom.ex*
- *pboundary_front_s.ex*
- *pboundary_left_w.ex*
- *pboundary_right_e.ex*
- *pboundary_top.ex*

7.2 dfnGen

connectivity.dat: Fracture connection list. Each row corresponds to a single fracture. The integers in that row are the fractures that fracture intersects with. These are the non-zero elements of the adjacency matrix.

convert_uge_params.txt: Input file do conver_uge executable.

DFN_output.txt: Detailed information about fracture network. Output by DFNGen.

families.dat: Information about fracture families. Produced by DFNGen.

input_generator.dat: Input file for DFN generator.

input_generator_clean.dat: Abbreviated input file for DFN generator.

normal_vectors.dat: Normal vector of each fracture in the network.

poly_info.dat: Fracture information output by DFNGen. Format: Fracture Number, Family number, rotation angle for rotateln in LaGriT, x0, y0, z0, x1, y1, z1 (end points of line of rotation).

user_rects.dat: User defined rectangle file.

radii: Subdirectory containing fracture radii information.

radii.dat: Concatentate file of fracture radii. Contains fractures that are removed due to isolation.

radii_Final.dat: Concatentated file of final radii in the DFN.

rejections.dat: Summary of rejection reasons.

rejectsPerAttempt.dat: Number of rejections per attempted fracture.

translations.dat: Fracture centriods.

triple_points.dat: x,y,z location of triple intersection points.

warningFileDFNGen.txt: Warning file output by DFNGen.

7.3 LaGrit

bound_zones.lgi: LaGriT run file to identify boundary nodes. Dumps zone files.

boundary_output.txt: Output file from bound_zones.lgi.

finalmesh.txt: Brief summary of final mesh.

full_mesh.inp: Full DFN mesh in AVS format.

full_mesh.gmv: Full DFN mesh in GMV (general mesh viewer) format.

full_mesh.lg: Information: Full DFN mesh in LaGriT binary format.

intersections: Directory containing intersection avs files output by the generator and used by LaGrit.

lagrit_logs: Directory of output files from individual meshing.

logx3dgen: LaGriT output.

outx3dgen: LaGriT output.

parameters: Directory of parameter*.mgli files used for fracture meshing.

params.txt: Parameter information about the fracture network used for meshing. Includes number of fractures, h, visualmode, expected number of dudded points, and x,y,z dimensions of the domain.

polys: Subdirectory contiaining AVS file for polygon boundaries.

tri_fracture.stor: FEHM stor file. Information about cell volume and area.

user_function.lgi: Function used by LaGriT for meshing. Defines coarsening gradient.

7.4 PFLOTRAN

aperture.dat: Fracture based aperture value for the DFN. Used to rescale volumes in full_mesh_vol_area.uge.

cellinfo.dat: Mesh information output by PFLOTRAN.

dfn_explicit-000.vtk: VTK file of initial conditions of PFLOTRAN. Mesh is not included in this file.

dfn_explicit-001.vtk: VTK file of steady-state solution of PFLOTRAN. Mesh is not included in this file.

dfn_explicit-mas.dat: pflotran information file.

dfn_explicit.in: pflotran input file.

_dfn_explicit.out: pflotran output file.

dfn_properties.h5: h5 file of fracture network properties, permeability, used by pflotran.

full_mesh.uge: Full DFN mesh in UGE format. NOTE volumes are not correct in this file. This file is processed by convert_uge to create full_mesh_vol_area.uge, which has the correct volumes.

full_mesh_viz.inp: Full DFN mesh with limited attributes in AVS format.

full_mesh_vol_area.uge: Full DFN in uge format. Volumes and areas have been corrected.

materialid.dat: Material ID (Fracture Number) for every node in the mesh.

parsed_vtk: Directory of pflotran results.

perm.dat: Fracture permeabilities in FEHM format. Each fracture is listed as a zone, starting index at 7.

pboundary_back_n.ex: Boundary file for back of the domain used by PFLOTRAN.

pboundary_bottom.ex: Boundary file for bottom of the domain used by PFLOTRAN.

pboundary_front_s.ex: Boundary file for front of the domain used by PFLOTRAN.

pboundary_left_w.ex: Boundary file for left side of the domain used by PFLOTRAN.

pboundary_right_e.ex: Boundary file for right of the domain used by PFLOTRAN.

pboundary_top.ex: Boundary file for top of the domain used by PFLOTRAN.

7.5 dfnTrans

allboundaries.zone: Concatenated file of all zone files.

darcyvel.dat: Concatenated file of darcy velocities output by PFLOTRAN.

dfnTrans_output_dir: Outpur directory from DFNTrans. Particle travel times, trajectories, and reconstructed Velocities are in this directory.

PTDFN_control.dat: Input file for DFNTrans.

pboundary_back_n.zone: Boundary zone file for the back of the domain.

pboundary_bottom.zone: Boundary zone file for the bottom of the domain.

pboundary_front_s.zone: Boundary zone file for the front of the domain.

pboundary_left_w.zone: Boundary zone file for the left side of the domain.

pboundary_right_e.zone: Boundary zone file for the bottom of the domain.

pboundary_top.zone: Boundary zone file for the top of the domain.

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-Version2.0/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.

8.1 Turn on X forwarding if on server

Ensure that X forwarding is turned on if you are running dfnWorks from an ssh connection. This requires that the ssh login have the -X option:

```
$ ssh -X SERVER_NAME
```

8.2 Go to the dnfWorks-Version2.0 repository

```
$ cd ~/dfnWorks-Version2.0/
```

8.3 Fix paths in test directory

Fix the pathnames for all files in the folder /tests/ . This can be done automatically by running the script fix_paths.py:

```
$ cd /pydfnworks/bin/
$ python fix_paths.py
```

8.4 Set the PETSC, PFLOTRAN, Python, and LaGriT paths correctly

Before executing dfnWorks, the following paths must be set:

- DFNWORKS_PATH: the dfnWorks-Version2.0 repository folder

- 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

```
$ vi /pydfnworks/pydfnworks/paths.py
```

For example:

```
os.environ['DFNWORKS_PATH'] = '/home/nknapp/dfnWorks-Version2.0/'
```

8.5 Setup the Python package pydfnworks

Go up a directory:

```
$ cd ..
```

If the user has admin privelges:

```
$ python setup.py install
```

If the user DOES NOT have admin priveleges:

```
$ python setup.py install --user
```

8.6 Executing dfnWorks

To run one of the test cases enter the following command:

- python run.py -name [JOBNAME] -input [INPUT_FILE] -ncpu [NUMBER_OF_CPUS]

The arguments are:

- [JOBNAME]: The name of the run, which is also the absolute path name of 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.

For example, to run the 4_user_defined rectangles test below, on 4 CPUs, the command line input would be:

```
$ cd bin
$ python run.py -name /home/nknapp/4_user_defined_rectangles
 -input /home/nknapp/dfnWorks-Version2.0/tests/4_user_defined_rectangles.txt
 -ncpu 4
```

This command will run the 4_user_defined_rectangles test and create a new folder 4_user_defined_rectangles where all output files will be located. Descriptions of each output file are in the documentation. In the following sections, we provide descriptions of the output you should expect for each of the five examples.

8.7 Summary

Here is a quick summary of the steps described above, with commands for the 4_user_defined_rectangles test case. **You MUST edit the paths in paths.py, and change ~ to the location of the repository, for these commands to work! Also, note that the last command should be on one line.**

```
$ ssh -X SERVER_NAME
$ cd ~/dfnWorks-Version2.0/pydfnworks/bin/
$ python fix_paths.py
$ vi /pydfnworks/pydfnworks/paths.py
$ cd ..
$ python setup.py install --user
$ cd bin
$ python run.py -name /home/nknapp/4_user_defined_rectangles
 -input /home/nknapp/dfnWorks-Version2.0/tests/4_user_defined_rectangles.txt
 -ncpu 4
```

8.8 Testing dfnWorks

The test.py script, located in dfnWorks-Version2.0/bin/, can be used to easily test functionality of dfnWorks. It runs a test suite (input files contained in the tests folder) that test the exponential, power law, and lognormal distributions as well as user-defined ellipses and user-defined rectangle fracture inputs. Syntax: python test.py [TEST_NAME_PART], where [TEST_NAME_PART] is all or part of the name of the tests the user wants to run.

8.9 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-Version2.0/tests, and is named define_4_user_rectangles.dat. To run the test on 4 cpus, enter the following command line input:

```
$ cd ~/dfnWorks-Version2.0/pydfnworks/bin/
$ python test.py rect
```

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.
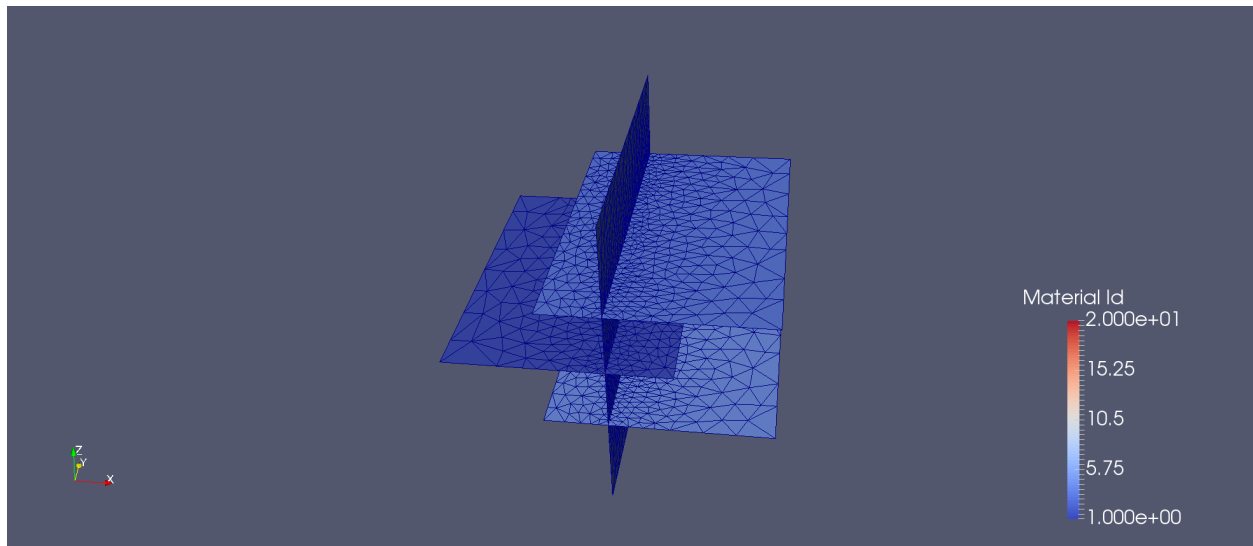
The network of four fractures, colored by pressure solution. High pressure (red) Dirichlet boundary conditions are applied on the edge of the single fracture along the boundary x = -0.5, and low pressure (blue) boundary conditions are applied on the edges of the two fractures at the boundary x = 0.5. 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.
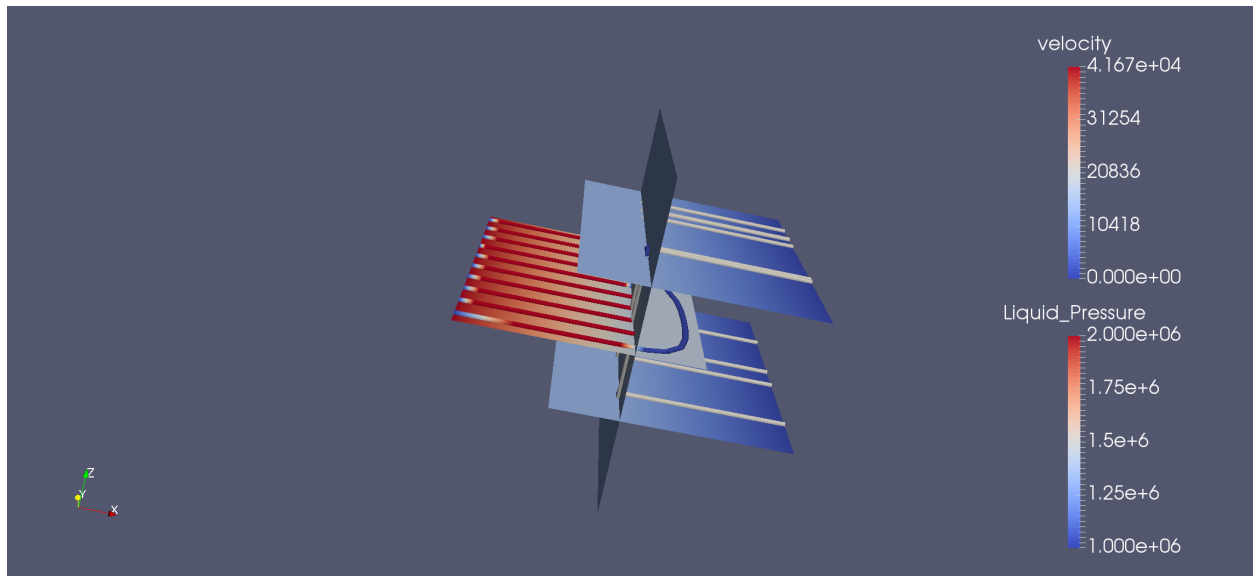
In the other tests, only a brief description and pictures are provided.

8.10 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-Version2.0/tests, and is named define_4_user_ellipses.dat.
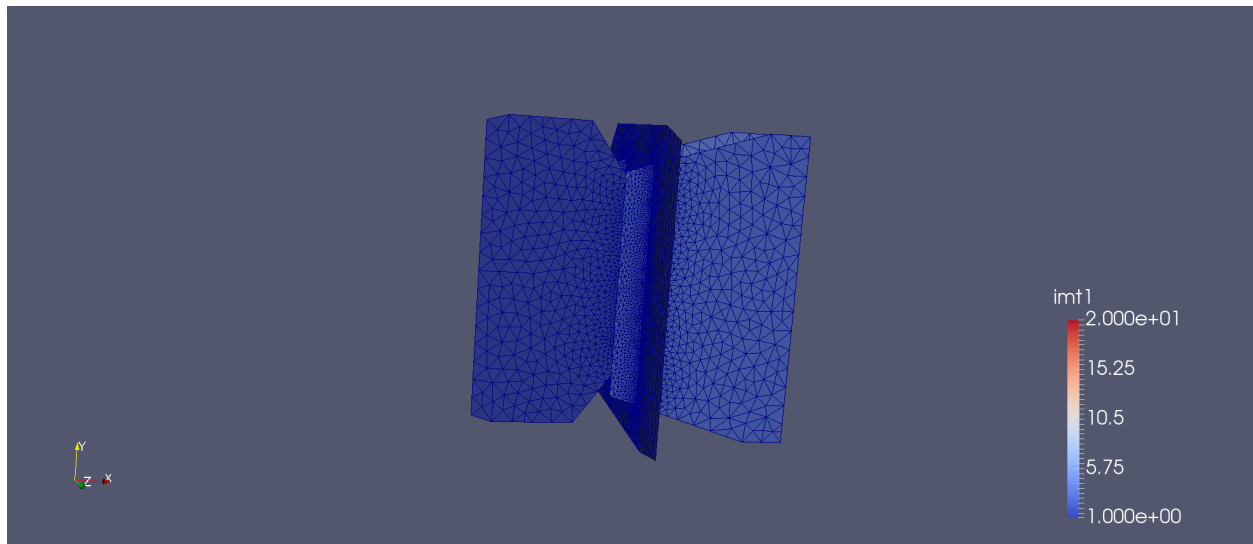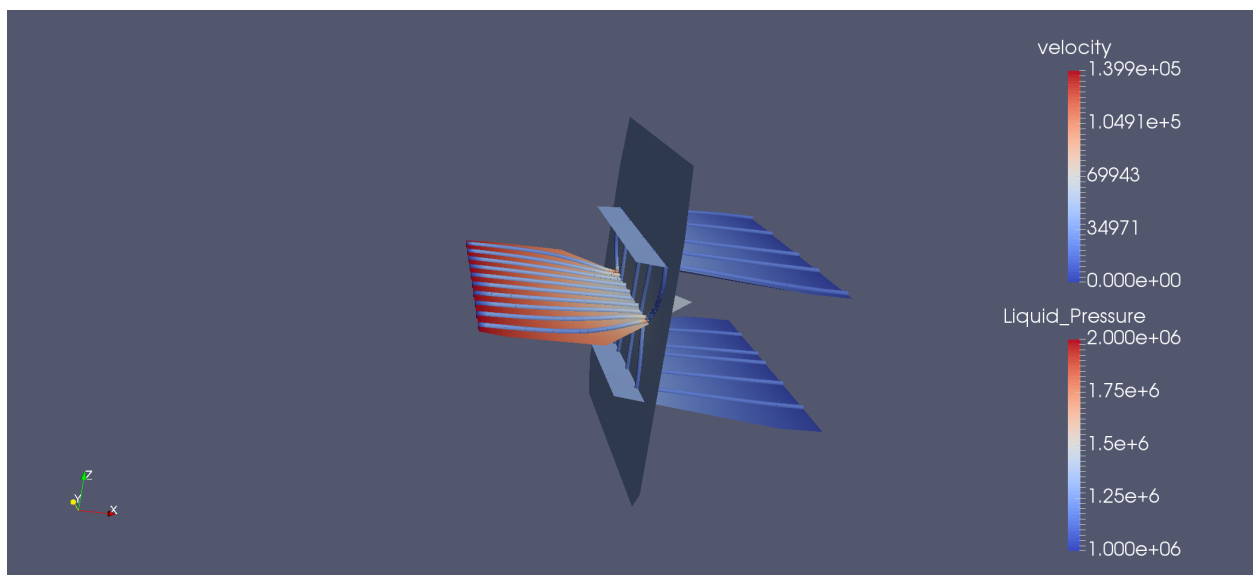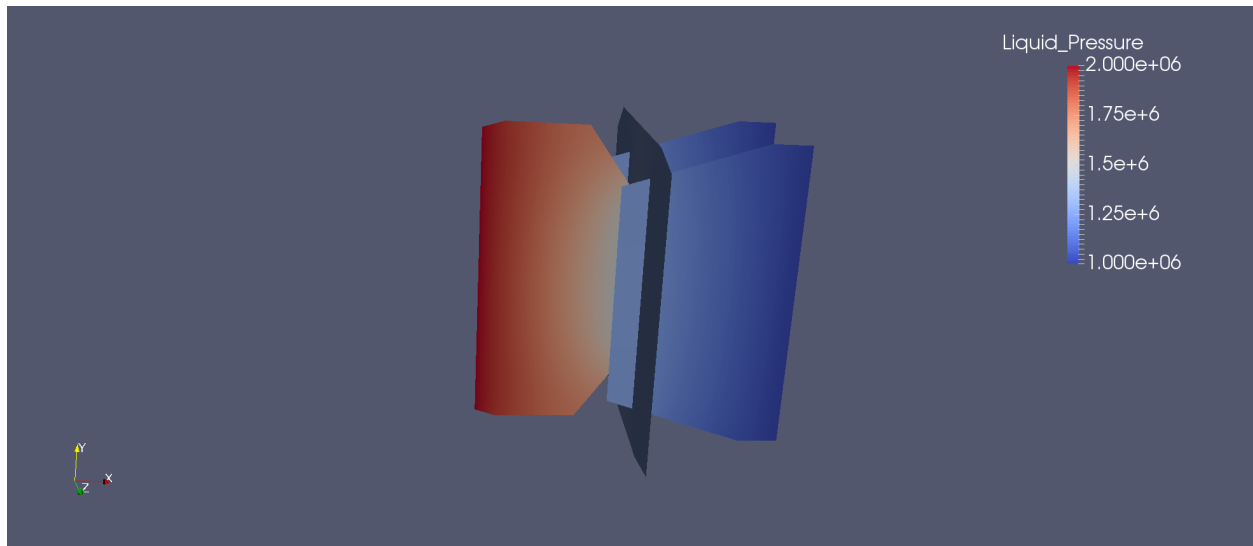
8.11 truncated_power_law_dist

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.
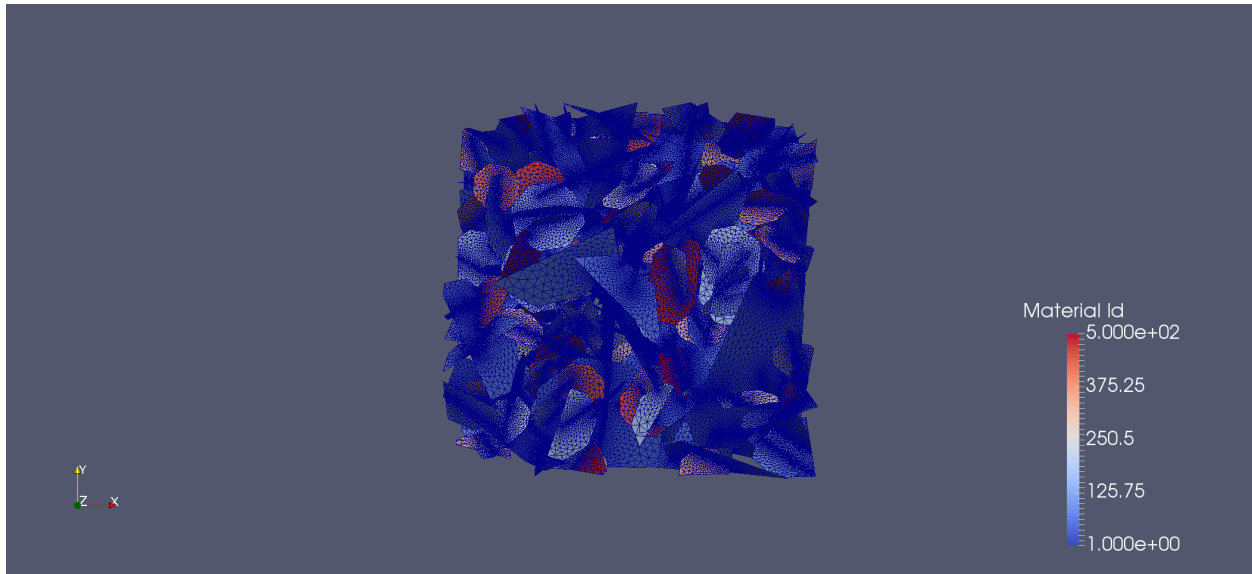
8.12 exponential_dist

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.
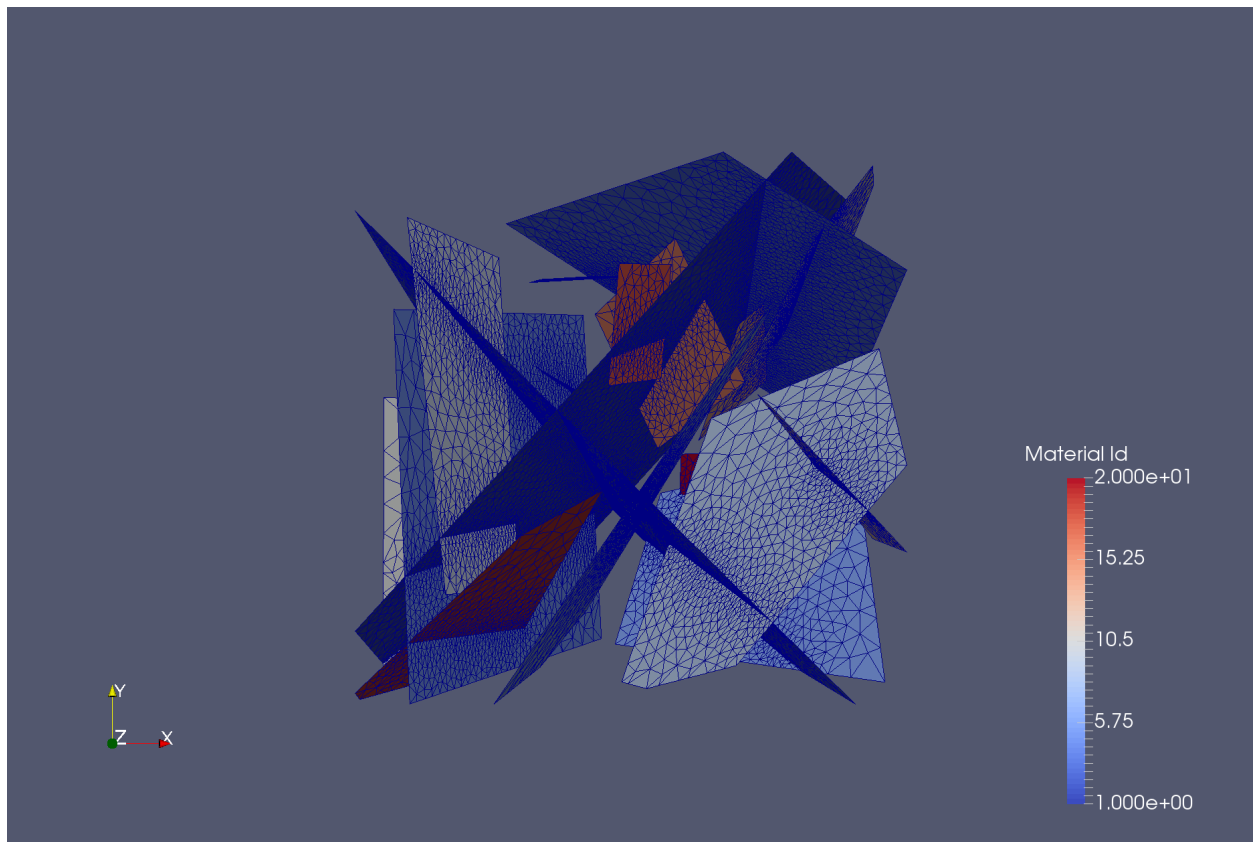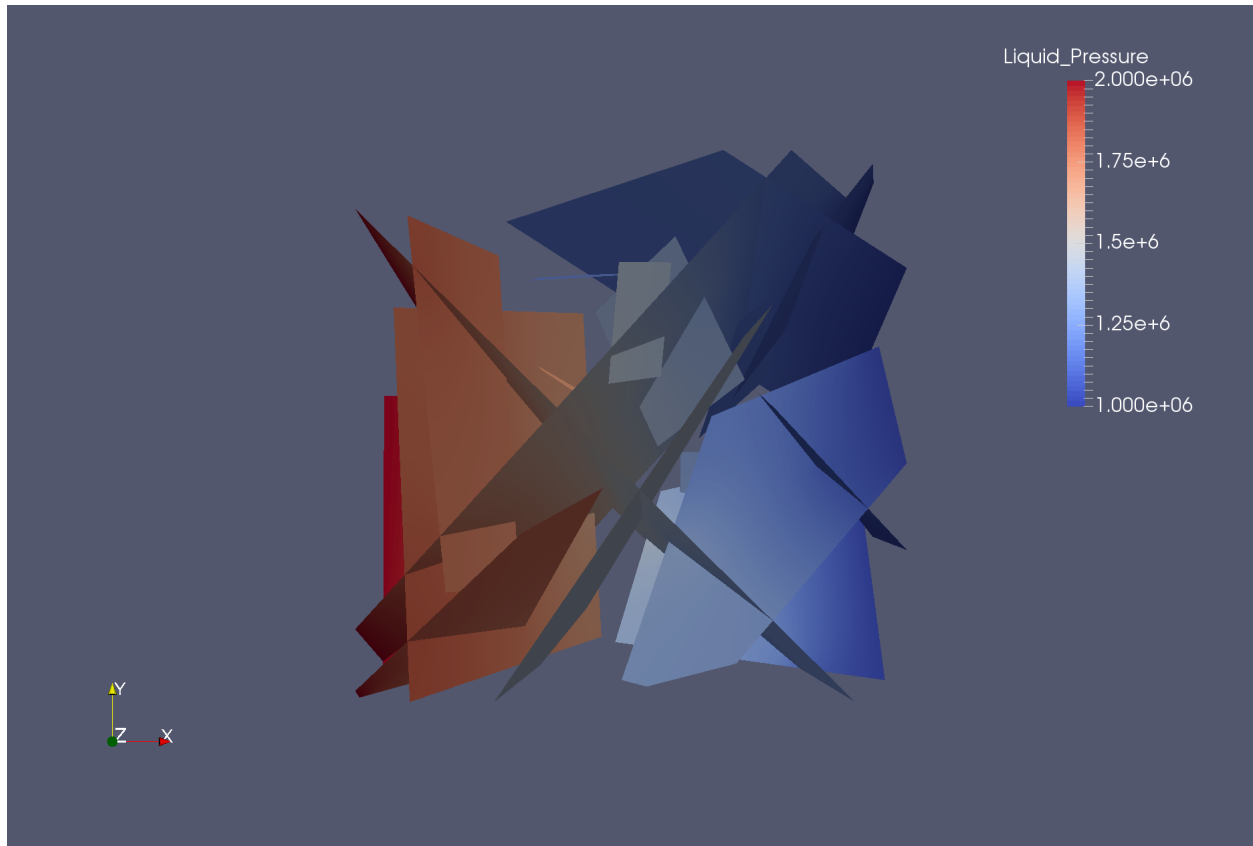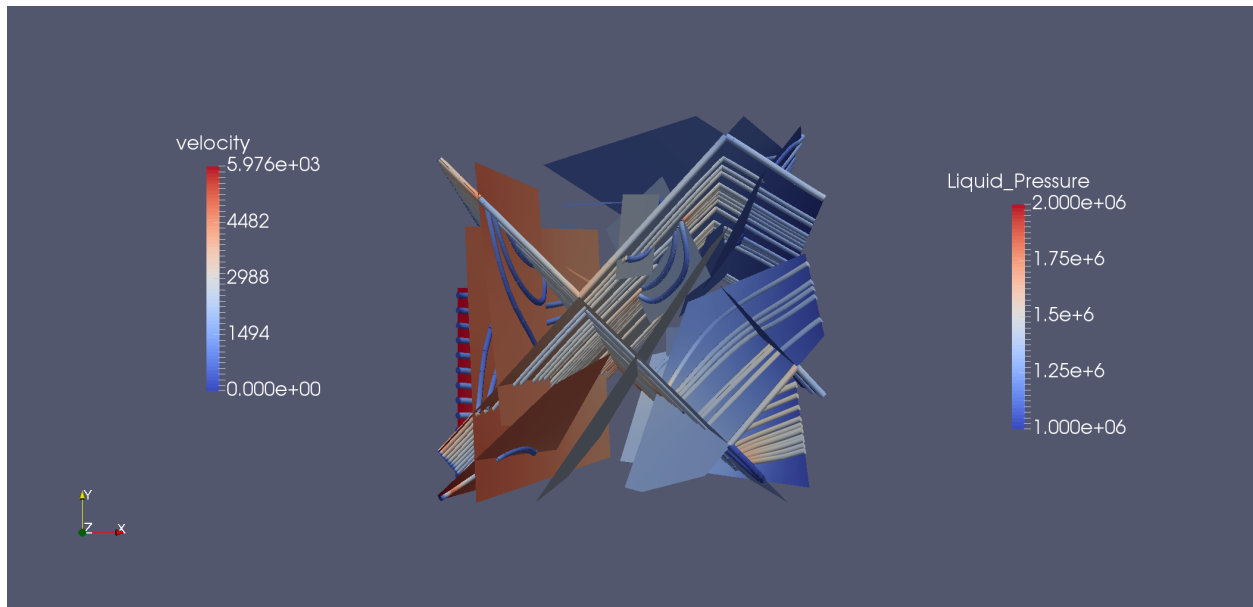
8.13 lognormal_dist

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.
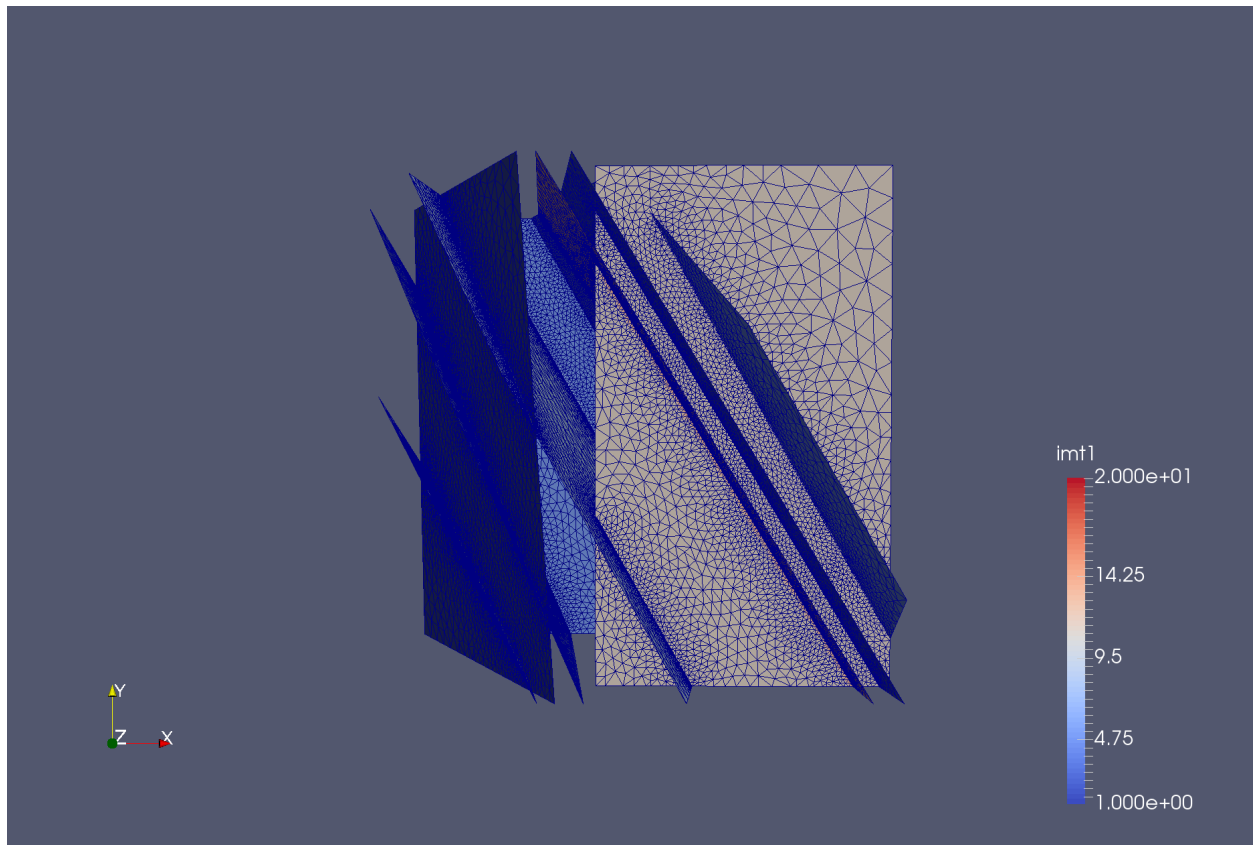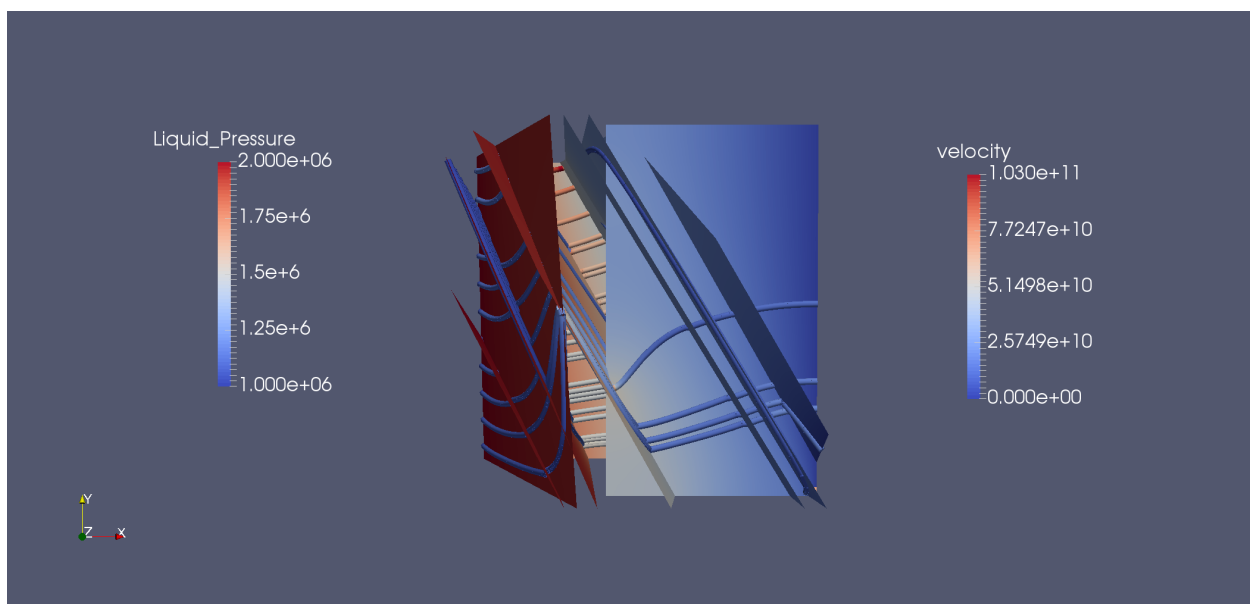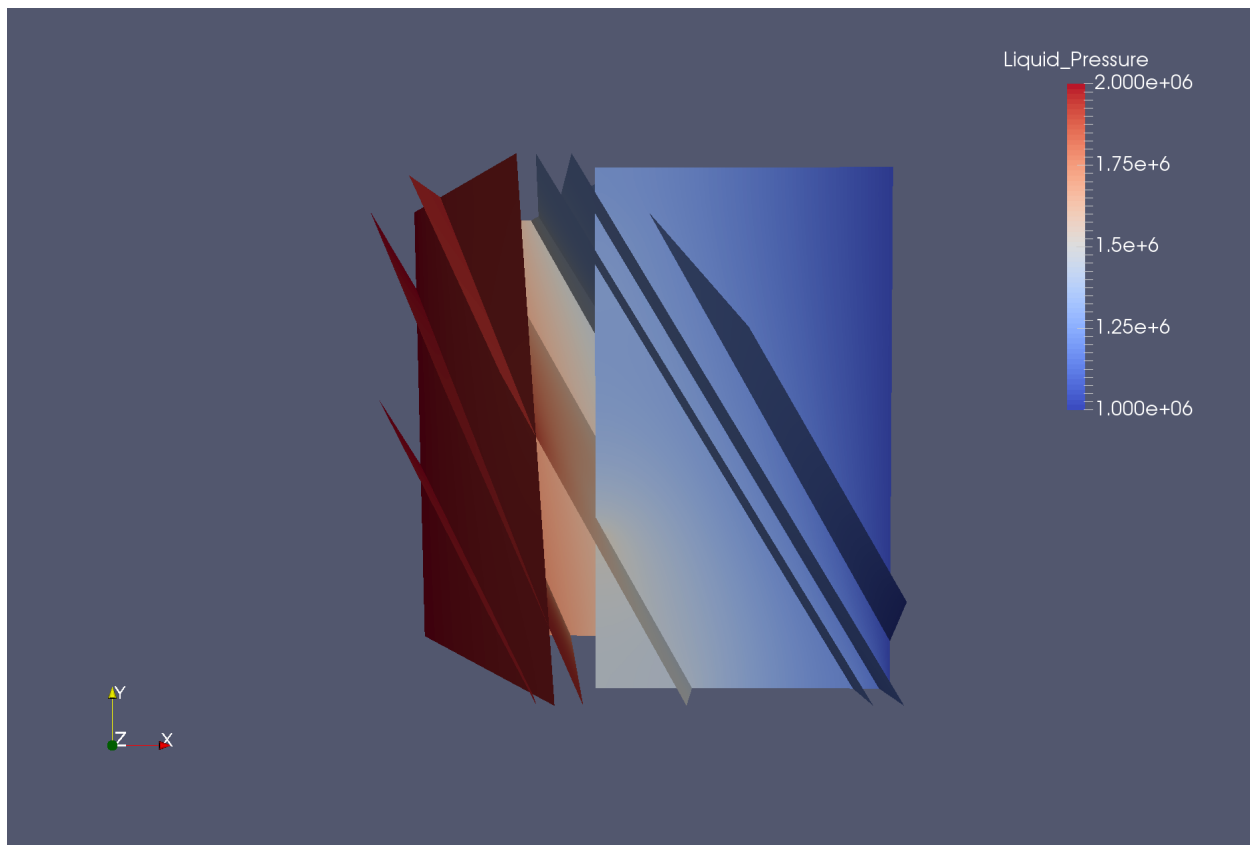
EXAMPLE APPLICATIONS

9.1 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.

9.2 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.

9.3 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

S

T

V

W

Z