dfnWorks Documentation

Release 2.5.3

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

Mar 08, 2022

CONTENTS

1 Welcome To dfnWorks **3**
- 1.1 Obtaining dfnWorks . . . 3
- 1.2 Citing dfnWorks . . . 3
- 1.3 Versions . . . 4
  - 1.3.1 v2.5 . . . 4
  - 1.3.2 v2.4 . . . 4
  - 1.3.3 v2.3 . . . 4
  - 1.3.4 v2.3 . . . 4
  - 1.3.5 v2.1 . . . 5
  - 1.3.6 v2.0 . . . 5
- 1.4 About this manual . . . 5
- 1.5 Contact . . . 5
- 1.6 Contributors . . . 5
  - 1.6.1 LANL . . . 5
  - 1.6.2 External . . . 6
- 1.7 Copyright Information . . . 6

2 Example Applications **9**
- 2.1 Carbon dioxide sequestration . . . 9
- 2.2 Shale energy extraction . . . 10
- 2.3 Nuclear waste repository . . . 10

3 Setting and Running up dfnWorks **13**
- 3.1 Docker . . . 13
  - 3.1.1 Running the dfnWorks container . . . 13
- 3.2 Native build from github repository . . . 13
  - 3.2.1 Clone the dnfWorks repository . . . 14
  - 3.2.2 Fix paths in test directory . . . 14
  - 3.2.3 Set the LagriT, PETSC, PFLOTRAN, Python, and FEHM paths . . . 14
  - 3.2.4 Setup the python package pydfnworks . . . 15
  - 3.2.5 Installation Requirements for Native Build . . . 15

4 Examples **17**
- 4.1 4_user_defined_rects . . . 17
- 4.2 4_user_defined_ell_uniform . . . 18
- 4.3 exp: Exponentially Distributed fracture lengths . . . 18
- 4.4 TPL: Truncated Power-Law . . . 19
- 4.5 Graph-based pruning . . . 19

5 pydfnworks: the dfnWorks python package **21**

5.1 Running dfnWorks from the command line using pydfnWorks . . . 21
5.2 Command Line Arguments: . . . 22
5.2.1 -name . . . 22
5.2.2 -input . . . 22
5.2.3 -ncpu . . . 22
5.2.4 -path . . . 23
5.2.5 -prune_file . . . 23
5.2.6 -cell . . . 23
5.3 pydfnWorks : Modules . . . 23
5.3.1 DFN Class and setup . . . 24
5.4 Detailed Doxygen Documentation . . . 24

6 pydfnworks: dfnGen **25**
6.1 dfnGen . . . 25
6.1.1 Processing generator input . . . 25
6.1.2 Running the generator . . . 25
6.1.3 Analysis of generated DFN . . . 26
6.1.4 Modification of hydraulic properties of the DFN . . . 27
6.2 Meshing - LaGriT . . . 28
6.2.1 Primary DFN meshing driver . . . 28
6.2.2 Meshing helper methods . . . 29
6.3 UDFM . . . 30
6.3.1 Creating an upscaled mesh of the DFN . . . 30

7 pydfnworks: dfnFlow **33**
7.1 Running Flow : General . . . 33
7.2 Running Flow: PFLOTRAN . . . 34
7.3 Running Flow: FEHM . . . 36
7.4 Processing Flow . . . 37

8 pydfnworks: dfnTrans **39**
8.1 Running Transport Simulations . . . 39

9 pydfnworks: dfnGraph **41**
9.1 General Graph Functions . . . 41
9.2 Graph-Based Flow and Transport . . . 44

10 pydfnworks: Well Package **47**
10.1 dfnWorks - Well Package . . . 47

11 dfnGen **51**
11.1 Domain Parameters . . . 51
11.1.1 domainSize . . . 51
11.1.2 domainSizeIncrease . . . 52
11.1.3 numOfLayers . . . 52
11.1.4 layers . . . 52
11.1.5 numOfRegions . . . 53
11.1.6 regions . . . 53
11.1.7 ignoreBoundaryFaces . . . 54
11.1.8 boundaryFaces . . . 54
11.2 General Network Generation Parameters . . . 54
11.2.1 stopCondition . . . 55
11.2.2 nPoly . . . 55
11.2.3 famProb . . . 55
11.2.4 orientationOption . . . 56

11.2.5 h . . . 56
11.2.6 disableFram . . . 57
11.2.7 printRejectReasons . . . 57
11.2.8 rejectsPerFracture . . . 58
11.2.9 radiiListIncrease . . . 58
11.2.10 visualizationMode . . . 58
11.2.11 seed . . . 59
11.2.12 keepOnlyLargestCluster . . . 59
11.2.13 keepIsolatedFractures . . . 59
11.2.14 tripleIntersections . . . 60
11.2.15 removeFracturesLessThan . . . 60
11.2.16 insertUserRectanglesFirst . . . 60
11.2.17 forceLargeFractures . . . 61
11.3 General Network Output Parameters . . . 61
11.3.1 outputAllRadii . . . 61
11.3.2 outputAcceptedRadiiPerFamily . . . 62
11.3.3 outputFinalRadiiPerFamily . . . 62
11.4 Fracture Family Generation Parameters: Ellipse . . . 62
11.4.1 Ellipse: General Parameters . . . 63
11.4.2 Ellipse: Fracture Orientation . . . 65
11.4.3 Ellipse: Fracture Radius Distributions . . . 70
11.5 Fracture Family Generation Parameters: Rectangle . . . 75
11.5.1 Rectangle: General Parameters . . . 75
11.5.2 Rectangle: Fracture Orientation . . . 77
11.5.3 Rectangle: Fracture Radius Distributions . . . 82
11.6 User Defined Fracture Generation Parameters . . . 87
11.6.1 User Defined Ellipses . . . 87
11.6.2 User Defined Rectangles . . . 95
11.6.3 Polygons . . . 100
11.7 Hydrological Properties . . . 102
11.7.1 Hydraulic Aperture . . . 102
11.7.2 Permeability . . . 105
11.8 Source Code Documentation (Doxygen) . . . 106

12 dfnFlow **107**

13 dfnTrans **111**
13.1 Documentation . . . 111

14 Output files **113**
14.1 dfnGen . . . 113
14.2 LaGrit . . . 114
14.3 PFLOTRAN . . . 115
14.4 dfnTrans . . . 116

15 dfnWorks Publications **119**

16 dfnWorks Gallery **123**

Index **129**

Contents:

CHAPTER

ONE

WELCOME TO DFNWORKS

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 Obtaining dfnWorks

dfnWorks 2.5 can be downloaded from https://hub.docker.com/r/ees16/dfnworks

dfnWorks 2.5 can be downloaded from https://github.com/lanl/dfnWorks/

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

1.2 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},
```

(continues on next page)

(continued from previous page)

```
    year={2015},
    publisher={Elsevier}
  }
```

1.3 Versions

1.3.1 v2.5

- New Generation parameters, family orientation by trend/plunge and dip/strike
- Define fracture families by region
- Updated output report

1.3.2 v2.4

- New meshing technique (Poisson disc sampling)
- Define fracture families by region
- Updated output report
- Well Package

1.3.3 v2.3

- Bug fixes in LaGrit Meshing
- Bug fixes in dfnTrans checking
- Bug fixes in dfnTrans output
- Expanded examples
- Added PDF printing abilities

1.3.4 v2.3

- pydfnWorks updated for python3
- Graph based (pipe-network approximations) for flow and transport
- Bug fixes in LaGrit Meshing
- Increased functionalities in pydfnworks including the path option
- dfn2graph capabilities
- FEHM flow solver
- Streamline routing option in dfnTrans
- Time Domain Random Walk in dfnTrans

1.3.5 v2.1

- Bug fixes in LaGrit Meshing
- Increased functionalities in pydfnworks including the path option

1.3.6 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.

1.4 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 `examples` directory of the dfnWorks repository, and a description of some applications of the dfnWorks suite.

1.5 Contact

Please email dfnworks@lanl.gov with questions about dfnWorks. Please let us know if you publish using dfnWorks and we'll add it to the *Publication Page*

1.6 Contributors

1.6.1 LANL

- Jeffrey D. Hyman
- Satish Karra
- Nataliia Makedonska
- Carl Gable
- Hari Viswanathan
- Matt Sweeney
- Shriram Srinivasan
- Aric Hagberg
- Yu Chen

1.6.2 External

- Quan Bui (now at LLNL)
- Jeremy Harrod (now at Spectra Logic)
- Scott Painter (now at ORNL)
- Thomas Sherman (University of Notre Dame)
- Johannes Krotz (Oregon State University)

1.7 Copyright Information

ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

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.

CHAPTER

TWO

EXAMPLE APPLICATIONS

2.1 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 CO_2 from anthropogenic sources and disposing it in geological formations such as deep saline aquifers and abandoned oil fields. Geological CO_2 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 CO_2 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 CO_2 , water (or brine) and fractured rock, is vital. dfnWorks capability to study multiphase flow in a DFN can be used to study potential CO_2 migration through cap-rock, a potential risk associated with proposed subsurface storage of CO_2 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 CO_2 –water interaction with rock.

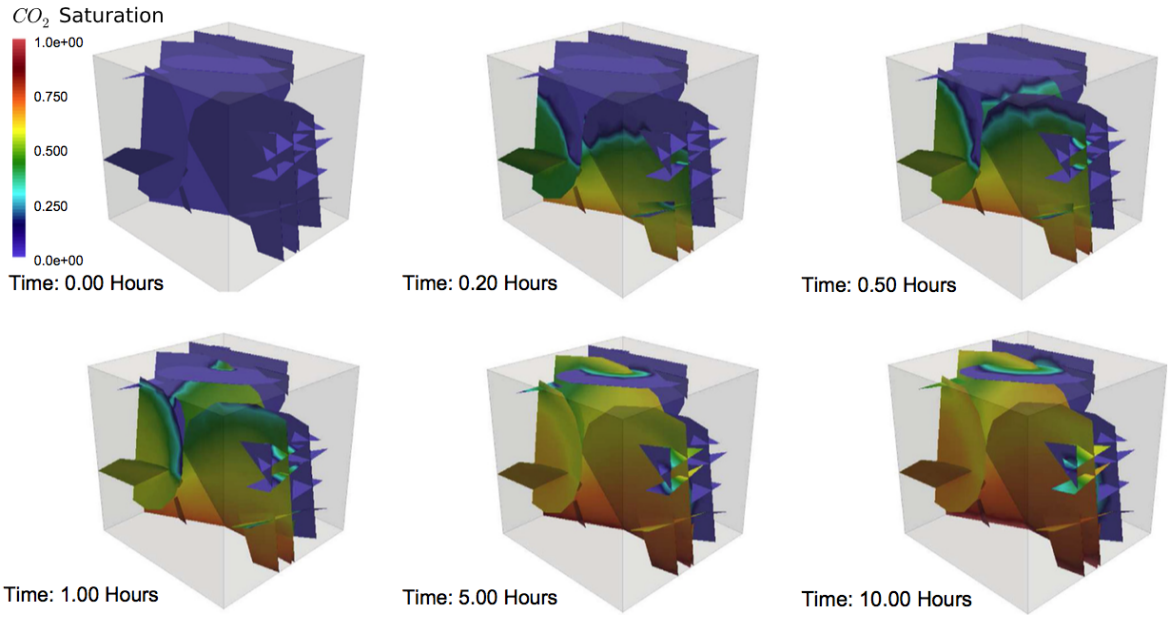


Fig. 1: *Temporal evolution of supercritical |CO2| displacing water in a meter cube DFN containing 24 fractures. The DFN is initially fully saturated with water, (top left time 0 hours) and supercritical |CO2| is slowly injected into the system from the bottom of the domain to displace the water for a total time of 10 h. There is an initial flush through the system during the first hour of the simulation, and then the rate of displacement decreases.*

2.2 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.

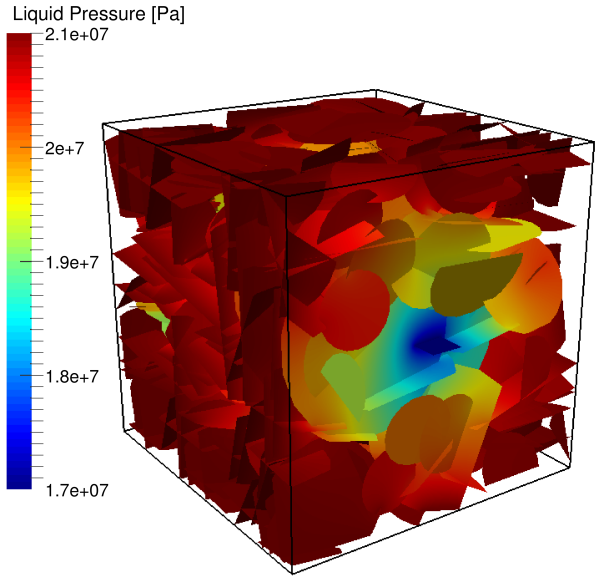


Fig. 2: *Pressure in a well used for hydraulic fracturing.*

2.3 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.

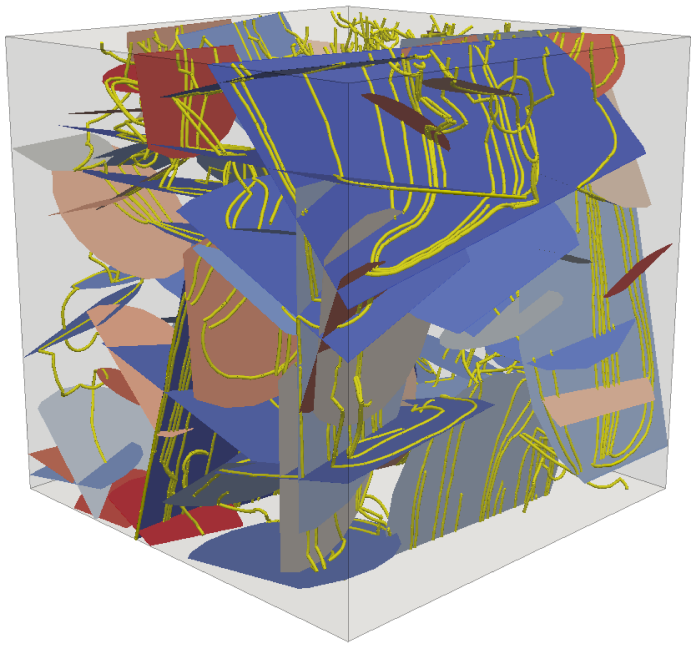


Fig. 3: *Simulated particle trajectories in fractured granite at Forsmark, Sweden.*

- Bachu, S., 2002. Sequestration of CO_2 in geological media in response to climate change: road map for site selection using the transform of the geological space into the CO_2 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.

CHAPTER

THREE

SETTING AND RUNNING UP DFNWORKS

3.1 Docker

The easiest way to get started with dfnWorks is using our docker container (https://hub.docker.com/r/ees16/dfnworks).

If you do not already have Docker installed on your machine, visit Getting Started with Docker.

The dfnWorks Docker image can be pulled from DockerHub using:

```
$ docker pull ees16/dfnworks:latest
```

3.1.1 Running the dfnWorks container

The base command for running the dfnWorks container is:

```
docker run -ti ees16/dfnworks:latest
```

However, to exchange files between the host and container, we will need to mount a volume.

The option `-v LOCAL_FOLDER:/dfnWorks/work` will allow all files present in the container folder `dfnWorks/work` to be exposed to `LOCAL_FOLDER`, where `LOCAL_FOLDER` is the absolute path to a folder on your machine.

With this is place, the final command for running the Docker container is:

On macOS:

```
docker run -ti -v <LOCAL_FOLDER>:/dfnWorks/work dfnworks:latest
```

3.2 Native build from github repository

This document contains instructions for setting up dfnWorks natively on your machine. To setup dfnWorks using Docker instead, see the next section.

3.2.1 Clone the dnfWorks repository

```
$ git clone https://github.com/lanl/dfnWorks.git
```

3.2.2 Fix paths in test directory

Fix the pathnames in files throughout pydfnworks. This can be done automatically by running the script `fix_paths.py`:

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

3.2.3 Set the LagriT, PETSC, PFLOTRAN, Python, and FEHM paths

Before executing dfnWorks, the following paths must be set:

- dfnWorks_PATH: the dfnWorks repository folder
- PETSC_DIR and PETSC_ARCH: PETSC environmental variables
- PFLOTRAN_EXE: Path to PFLOTRAN executable
- PYTHON_EXE: Path to python executable
- LAGRIT_EXE: Path to LaGriT executable

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

For example:

```
os.environ['dfnWorks_PATH'] = '/home/username/dfnWorks/'
```

Alternatively, you can create a `.dfnworksrc` file in your home directory with the following format

```
{
    "dfnworks_PATH": "<your-home-directory>/src/dfnworks-main/",
    "PETSC_DIR": "<your-home-directory>/src/petsc",
    "PETSC_ARCH": "arch-darwin-c-debug",
    "PFLOTRAN_EXE": "<your-home-directory>/src/pflotran/src/pflotran/pflotran",
    "PYTHON_EXE": "<your-home-directory>/anaconda3/bin/python",
    "LAGRIT_EXE": "<your-home-directory>/bin/lagrit",
    "FEHM_EXE": "<your-home-directory>//src/xfehm_v3.3.1"
}
```

3.2.4 Setup the python package pydfnworks

Go up into the pydfnworks sub-directory:

```
$ cd dfnWorks/pydfnworks/
```

If the user has admin privelges:

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

If the user DOES NOT have admin priveleges:

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

3.2.5 Installation Requirements for Native Build

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++.

Operating Systems

dfnWorks currently runs on Macs and Unix machine running Ubuntu.

Python

pydfnworks uses Python 3. We recommend using the Anaconda 3 distribution of Python, available at https://www.continuum.io/. pydfnworks requires the following python modules: `numpy`, `h5py`, `scipy`, `matplotlib`, `multiprocessing`, `argparse`, `shutil`, `os`, `sys`, `networkx`, `subprocess`, `glob`, `networkx`, `fpdf`, and `re`.

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.

PFLOTRAN

PFLOTRAN is a massively parallel subsurface flow and reactive transport code. PFLOTRAN solves a system of partial differential equations for multiphase, multicomponent and multi-scale 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.

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++ 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.

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. While not required for running dfnWorks, Paraview is very helpful for visualizing dfnWorks simulations.

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

CHAPTER

FOUR

EXAMPLES

This section contains a few examples of DFN generated using dfnWorks. All required input files for these examples are contained in the folder dfnWorks/examples/. 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 simplest so it is recommended that the user proceed in the order presented here.

All examples are in the examples/ directory. Within each subdirectory are the files required to run the example. The command line input is found in notes.txt. Be sure that you have created ~/test_output_files prior to running the examples.

4.1 4_user_defined_rects

Location: examples/4_user_defined_rects/

This test case consists of four user defined rectangular fractures within a a cubic domain with sides of length one meter. 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 job folder into Paraview.

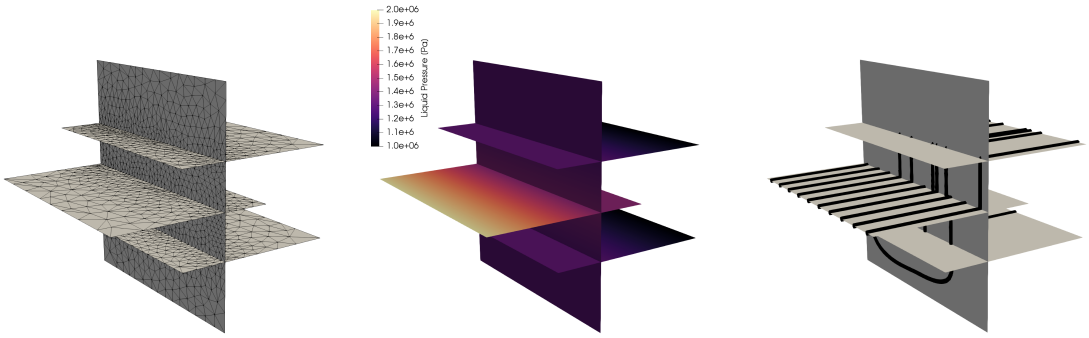


Fig. 1: *The meshed network of four rectangular fractures.*

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 parsed_vtk/dfn_explicit-001.vtk into Paraview.

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 veloc-

ity magnitude of the particle's velocity. Trajectories can be visualized by loading the files part_*.inp, in the folder 4_user_rectangles/traj/trajectories/

We have used the extract surface and tube filters in paraview for visual clarity.

4.2 4_user_defined_ell_uniform

Location: examples/4_user_defined_ell_uniform/

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 8 vertices. We have set the meshing resolution to be uniform by including the argument slope=0 into the mesh_networks function in run_explicit.py.

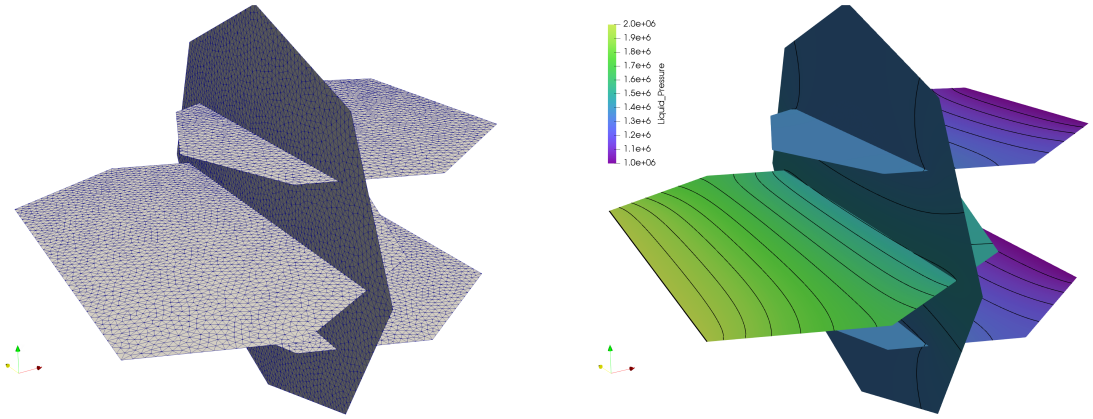


Fig. 2: *The uniformly meshed network of four circular fractures.*

4.3 exp: Exponentially Distributed fracture lengths

Location: examples/exp/

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. We have changed the flow direction to be aligned with the y-axis by modifying the PFLOTRAN input card dfn_explicit.in

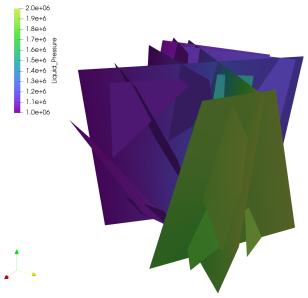
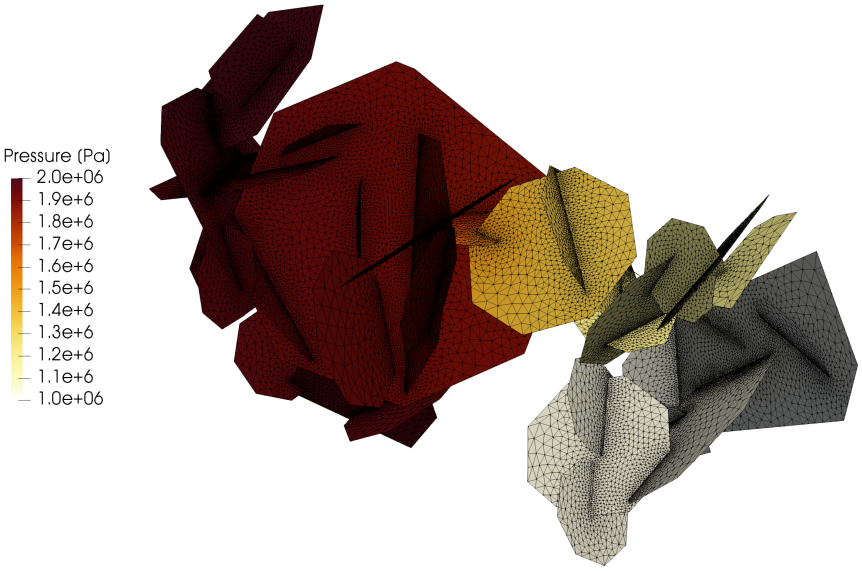


Fig. 3: *Pressure solution on with rectangular fractures whose lengths following a exponential distribution. Gradient is aligned with the Y-Axis*

4.4 TPL: Truncated Power-Law

Location: examples/TPL/

This test case consists of two families whose sizes have a truncated power law distribution with a minimum size of 1m and a maximum size of 5m an exponent 2.6. The domain size is cubic with an edge length of 15m.



4.5 Graph-based pruning

Location: examples/pruning/

This example uses a graph representation of a DFN to isolate the 2-core. The pruned DFN has all dead end fractures of the network are removed. This example has two run_explicit.py scripts. The first creates the original DFN and identifies the 2-core using networkx (https://networkx.github.io/). The second meshes the DFN corresponding to the 2-core of the graph and then runs flow and transport. The 2 core network is in a sub-directory 2-core. The original network has 207 fractures and the 2-core has 79 fractures.

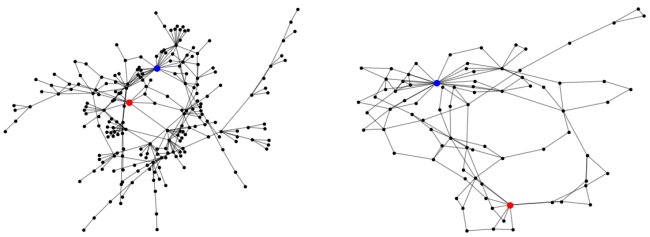


Fig. 4: *(left) Graph based on DFN topology. Each vertex is a fracture in the network. The inflow boundary is colored blue and the outflow is colored red. (right) 2-Core of the graph to the left.*

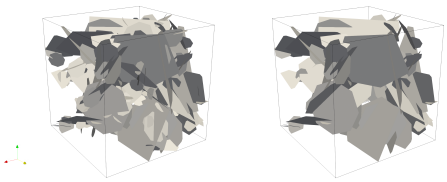


Fig. 5: *(left) Original DFN (right) DFN corresponding to the 2-core of the DFN to the left.*

CHAPTER

FIVE

PYDFNWORKS: THE DFNWORKS PYTHON PACKAGE

The pydfnworks package allows the user to 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.

The pydfnworks must be setup by the user using the following command in the directory `dfnWorks/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 basic information about the DFN class and pydfnworks setup.

5.1 Running dfnWorks from the command line using pydfnWorks

The recommended way to run dfnWorks is using a python call on the command line.

```
$ python run.py -name /dfnWorks/work/4_user_rects_example -input 4_user_defined_
↪rectangles_run_file.txt
```

Tip: Examples of command line calls for running simulations are provided in ascii files named `notes.txt` within each of the example directories.

The script `run.py` is the python control file that contains the workflow of the particular simulation. Below is a basic example taken from the 4_user_rects_example example:

```
from pydfnworks import *

DFN = create_dfn()
# General Work Flow
DFN.dfn_gen()
DFN.dfn_flow()
DFN.dfn_trans()
```

5.2 Command Line Arguments:

Along with the python script, each simulation is controlled by a set of command line arguments, which are described flow.

5.2.1 -name

Description: (Mandatory) Path of the simulation directory. Must be a valid path. The path is stored in `DFN.jobname` of the DFN object created by `create_dfn()`

Type: string

Example:

```
-name /dfnWorks/work/4_user_rects_example
```

Note: If the directory does not exist, `DFN.make_working_directory()` will create it.

5.2.2 -input

Description: (Mandatory) Path of the input file containing run files for dfnGen, dfnFlow (PFLOTRAN/FEHM/AMANZI), and dfnTrans. This file is parsed and the paths contained within are stored as `DFN.dfnGen_file`, `DFN.dfnFlow_file`, and `DFN.dfnTrans_file`. The local path for the files (string after the final `/` are stored as `DFN.local_dfnGen_file`, `DFN.local_dfnFlow_file`, and `DFN.local_dfnTrans_file`.

Type: string

Example:

```
-input 4_user_defined_rectangles_run_file.txt
```

Example of input run file:

```
dfnGen /dfnWorks/examples/4_user_rects/gen_4_user_rectangles.dat
dfnFlow /dfnWorks/examples/4_user_rects/dfn_explicit.in
dfnTrans /dfnWorks/examples/4_user_rects/PTDFN_control.dat
```

Note: The input file cannot have an empty line after dfnTrans.

5.2.3 -ncpu

Description: Number of processors to be used in the simulation. Stored as `DFN.ncpu`.

Type: integer

Example:

```
-ncpu 8
```

Note: Default is 4.

5.2.4 -path

Description: Path to parent directory. Useful for multiple runs using the same network with different meshing techniques, hydraulic properties, flow simulations, or pruned networks. Path is stored as `DFN.path`.

Type: string

Example:

```
-path /dfnWorks/work/4_user_rects_example
```

5.2.5 -prune_file

Description: Path to ascii file of fractures to be retained (not removed) in the network after pruning. See the pruning example for a workflow demonstration.

Type: string

Example:

```
-prune_file /dfnWorks/work/pruning_example/2_core.dat
```

Note: To prune the network, include `DFN.mesh_network(prune=True)` in the python run file.

5.2.6 -cell

Description: Toggle if the fracture apertures are cell based. If the option is included on the command line, then the workflow will assign cell-based apertures and permeabilities from the files `aper_node.dat` and `perm_node.dat`. These files consist of two columns, with a single line header value. The first column is the node number. The second column is the aperture/permeability value. See the See the in_fracture_var example for a workflow demonstration.

Type: Boolean

Example:

```
-cell
```

5.3 pydfnWorks : Modules

Information about the various pieces of pydfnworks is found in

pydfnGen - Network generation, meshing, and analysis

pydfnFlow - Flow simulations using PFLOTRAN and FEHM

pydfnTrans - Particle Tracking

pydfnGraph - Graph-based analysis and pipe-network simulations

Well-Package - Well simulations

5.3.1 DFN Class and setup

pydfnworks.general.dfnworks.**create_dfn**()

Parse command line inputs and input files to create and populate dfnworks class

Parameters None –

Returns **DFN** – DFN class object populated with information parsed from the command line. Information about DFN class is in dfnworks.py

Return type object

Notes

None

5.4 Detailed Doxygen Documentation

Doxygen

CHAPTER

SIX

PYDFNWORKS: DFNGEN

DFN Class functions used in network generation and meshing

6.1 dfnGen

6.1.1 Processing generator input

6.1.2 Running the generator

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

Execute dfnGen

Parameters **self** (*object*) – DFN Class

Returns

Return type None

Notes

After generation is complete, this script checks whether the generation of the fracture network failed or succeeded based on the existence of the file params.txt.

pydfnworks.dfnGen.generation.generator.**dfn_gen**(*self*, *output=True*, *visual_mode=None*)

Wrapper script the runs 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

Parameters

- **self** (*object*) – DFN Class object
- **output** (*bool*) – If True, output pdf will be created. If False, no pdf is made
- **visual_mode** (*None*) – If the user wants to run in a different meshing mode from what is in params.txt, set visual_mode = True/False on command line to override meshing mode

Returns

Return type None

Notes

Details of each portion of the routine are in those sections

pydfnworks.dfnGen.generation.generator.**make_working_directory**(*self*, *delete=False*)
Make working directory for dfnWorks Simulation

Parameters **self** (*object*) – DFN Class object

Returns

Return type None

Notes

If directory already exists, user is prompted if they want to overwrite and proceed. If not, program exits.

6.1.3 Analysis of generated DFN

filename gen_output.py

synopsis Main driver for dfnGen output report

version 1.0

maintainer Jeffrey Hyman

moduleauthor Jeffrey Hyman <jhyman@lanl.gov>

pydfnworks.dfnGen.generation.output_report.gen_output.**output_report**(*self*, *verbose=True*, *output_dir='dfnGen_output_report'*)
Creates a PDF output report for the network created by DFNGen. Plots of the fracture lengths, locations, orientations are produced for each family. Files are written into "output_dir/family_{id}/". Information about the whole network are also created and written into "output_dir/network/"

Parameters

- **self** (*object*) – DFN Class object
- **verbose** (*bool*) – Toggle for the amount of information printed to screen. If true, progress information printed to screen
- **output_dir** (*string*) – Name of directory where all plots are saved

Returns

Return type None

Notes

Final output report is named "jobname"_output_report.pdf User defined fractures (ellipses, rectangles, and polygons) are not supported at this time.

6.1.4 Modification of hydraulic properties of the DFN

pydfnworks.dfnGen.generation.hydraulic_properties.**dump_hydraulic_values**(*self*, *b*, *perm*, *T*, *prefix=None*)

Writes variable information to files.

Parameters

- **prefix** (*string*) – prefix of aperture.dat and perm.dat file names prefix_aperture.dat and prefix_perm.dat
- **b** (*array*) – aperture values
- **perm** (*array*) – permeability values
- **T** (*array*) – transmissivity values

Returns

Return type None

Notes

pydfnworks.dfnGen.generation.hydraulic_properties.**generate_hydraulic_values**(*self*, *variable*, *relationship*, *params*, *radii_filename='radii_Final.dat'*, *family_id=None*)

Generates hydraulic property values.

Parameters

- **self** (*object*) – DFN Class
- **relationship** (*string*) – name of functional relationship for apertures. options are log-normal, correlated, semi-correlated, and constant
- **params** (*dictionary*) – dictionary of parameters for functional relationship
- **family_id** (*int*) – family id of fractures

Returns

- **b** (*array*) – aperture values
- **perm** (*array*) – permeability values
- **T** (*array*) – transmissivity values
- **idx** (*array of bool*) – true / false of fracture families requested. If family_id = None, all entires are true. Only family members entires of b, perm, and T will be non-zero

Notes

See Hyman et al. 2016 "Fracture size and transmissivity correlations: Implications for transport simulations in sparse three-dimensional discrete fracture networks following a truncated power law distribution of fracture size" Water Resources Research for more details

6.2 Meshing - LaGriT

6.2.1 Primary DFN meshing driver

pydfnworks.dfnGen.meshing.mesh_dfn.**mesh_network**(*self*, *prune=False*, *uniform_mesh=False*, *production_mode=True*, *coarse_factor=8*, *slope=0.1*, *min_dist=1*, *max_dist=40*, *concurrent_samples=10*, *grid_size=10*, *visual_mode=None*, *well_flag=False*)

Mesh fracture network using LaGriT

Parameters

- **self** (*object*) – DFN Class
- **prune** (*bool*) – If prune is False, mesh entire network. If prune is True, mesh only fractures in self.prune_file
- **uniform_mesh** (*bool*) – If true, mesh is uniform resolution. If False, mesh is spatially variable
- **production_mode** (*bool*) – If True, all working files while meshing are cleaned up. If False, then working files will not be deleted
- **visual_mode** (*None*) – If the user wants to run in a different meshing mode from what is in params.txt, set visual_mode = True/False on command line to override meshing mode
- **coarse_factor** (*float*) – Maximum resolution of the mesh. Given as a factor of h
- **slope** (*float*) – slope of variable coarsening resolution.
- **min_dist** (*float*) – Range of constant min-distance around an intersection (in units of h).
- **max_dist** (*float*) – Range over which the min-distance between nodes increases (in units of h)
- **concurrent_samples** (*int*) – number of new candidates sampled around an accepted node at a time.
- **grid_size** (*float*) – side length of the occupancy grid is given by H/occupancy_factor
- **well_flag** (*bool*) – If well flag is true, higher resolution around the points in

Returns

Return type None

Notes

1. For uniform resolution mesh, set slope = 0
2. All fractures in self.prune_file must intersect at least 1 other fracture

6.2.2 Meshing helper methods

pydfnworks.dfnGen.meshing.mesh_dfn_helper.**create_mesh_links**(*self*, *path*)
Makes symlinks for files in path required for meshing

Parameters

- **self** (*DFN object*) –
- **path** (*string*) – Path to where meshing files are located

Returns

Return type None

Notes

None

pydfnworks.dfnGen.meshing.mesh_dfn_helper.**inp2gmv**(*self*, *inp_file=''*)
Convert inp file to gmv file, for general mesh viewer. Name of output file for base.inp is base.gmv

Parameters

- **self** (*object*) – DFN Class
- **inp_file** (*str*) – Name of inp file if not an attribure of self

Returns

Return type None

Notes

pydfnworks.dfnGen.meshing.mesh_dfn_helper.**inp2vtk_python**(*self*)
Using Python VTK library, convert inp file to VTK file.

Parameters **self** (*object*) – DFN Class

Returns

Return type None

Notes

For a mesh base.inp, this dumps a VTK file named base.vtk

pydfnworks.dfnGen.meshing.mesh_dfn_helper.**run_lagrit_script**(*lagrit_file*, *output_file=None*, *quiet=False*)

Runs LaGriT

Parameters ---------- –

lagrit_file [string] Name of LaGriT script to run

output_file [string] Name of file to dump LaGriT output

quiet [bool] If false, information will be printed to screen.

Returns **failure** – If the run was successful, then 0 is returned.

Return type int

pydfnworks.dfnGen.meshing.add_attribute_to_mesh.**add_variable_to_mesh**(*self*, *variable*, *variable_file*, *mesh_file_in*, *mesh_file_out=None*, *node_based=False*)

Adds a variable to the nodes of a mesh. Can be either fracture (material) based or node based.

Parameters

- **self** (*object*) – DFN Class
- **variable** (*string*) – name of variable
- **variable_file** (*string*) – name of file containing variable files. Must be a single column where each line corresponds to that node number in the mesh
- **mesh_file_in** (*string*) – Name of source mesh file
- **mesh_file_out** (*string*) – Name of Target mesh file. If no name if provide, mesh_file_in will be used
- **node_based** (*bool*) – Set to True if variable_file contains node-based values, Set to False if variable_file provide fracture based values

Returns **lagrit_file** – Name of LaGriT output file

Return type string

6.3 UDFM

6.3.1 Creating an upscaled mesh of the DFN

pydfnworks.dfnGen.meshing.udfm.map2continuum.**map_to_continuum**(*self*, *l*, *orl*, *path='./'*, *dir_name='octree'*)

This function generates an octree-refined continuum mesh using the reduced_mesh.inp as input. To generate the reduced_mesh.inp, one must turn visualization mode on in the DFN input card.

Parameters

- **self** (*object*) – DFN Class
- **l** (*float*) – Size (m) of level-0 mesh element in the continuum mesh

- **orl** (*int*) – Number of total refinement levels in the octree
- **path** (*string*) – path to primary DFN directory
- **dir_name** (*string*) – name of directory where the octree mesh is created

Returns

Return type None

Notes

octree_dfn.inp [Mesh file] Octree-refined continuum mesh

fracX.inp [Mesh files] Octree-refined continuum meshes, which contain intersection areas

pydfnworks.dfnGen.meshing.udfm.upscale.**upscale**(*self*, *mat_perm*, *mat_por*, *path='../'*)

Generate permeabilities and porosities based on output of map2continuum.

Parameters

- **self** (*object*) – DFN Class
- **mat_perm** (*float*) – Matrix permeability (in m^2)
- **mat_por** (*float*) – Matrix porosity

Returns

- **perm_fehm.dat** (*text file*) – Contains permeability data for FEHM input
- **rock_fehm.dat** (*text file*) – Contains rock properties data for FEHM input
- **mesh_permeability.h5** (*h5 file*) – Contains permeabilites at each node for PFLOTRAN input
- **mesh_porosity.h5** (*h5 file*) – Contains porosities at each node for PFLOTRAN input

Notes

None

pydfnworks.dfnGen.meshing.udfm.false_connections.**check_false_connections**(*self*, *path='../'*)

Parameters

- **self** (*object*) – DFN Class
- **fmc_filname** (*string*) – name of the pickled dictionary of mesh and fracture intersections

Returns

- **num_false_connections** (*int*) – number of false connections
- **num_cell_false** (*int*) – number of Voronoi cells with false connections
- **false_connections** (*list*) – list of tuples of false connections created by upscaling

Notes

map2continuum and upscale must be run first to create the fracture/mesh intersection dictionary. Thus must be run in the main job directory which contains connectivity.dat

CHAPTER

SEVEN

PYDFNWORKS: DFNFLOW

DFN Class functions used in flow simulations (PFLOTRAN and FEHM)

7.1 Running Flow : General

pydfnworks.dfnFlow.flow.**create_dfn_flow_links**(*self*, *path='../'*)

Create symlinks to files required to run dfnFlow that are in another directory.

Parameters

- **self** (*object*) – DFN Class
- **path** (*string*) – Absolute path to primary directory.

Returns

Return type None

Notes

1. Typically, the path is DFN.path, which is set by the command line argument -path
2. Currently only supported for PFLOTRAN

pydfnworks.dfnFlow.flow.**dfn_flow**(*self*, *dump_vtk=True*)

Run the dfnFlow portion of the workflow

Parameters

- **self** (*object*) – DFN Class
- **dump_vtk** (*bool*) – True - Write out vtk files for flow solutions False - Does not write out vtk files for flow solutions

Notes

Information on individual functions is found therein

pydfnworks.dfnFlow.flow.**set_flow_solver**(*self*, *flow_solver*)
Sets flow solver to be used

Parameters

- **self** (*object*) – DFN Class
- **flow_solver** (*string*) – Name of flow solver. Currently supported flow sovlers are FEHM and PFLOTRAN

Notes

Default is PFLOTRAN

7.2 Running Flow: PFLOTRAN

functions for using pflotran in dfnworks

pydfnworks.dfnFlow.pflotran.**lagrit2pflotran**(*self*, *inp_file=''*, *mesh_type=''*, *hex2tet=False*)
Takes output from LaGriT and processes it for use in PFLOTRAN. Calls the functuon write_perms_and_correct_volumes_areas() and zone2ex

Parameters

- **self** (*object*) – DFN Class
- **inp_file** (*str*) – Name of the inp (AVS) file produced by LaGriT
- **mesh_type** (*str*) – The type of mesh
- **hex2tet** (*bool*) – True if hex mesh elements should be converted to tet elements, False otherwise.

Returns

Return type None

Notes

None

pydfnworks.dfnFlow.pflotran.**parse_pflotran_vtk_python**(*self*, *grid_vtk_file=''*)
Adds CELL_DATA to POINT_DATA in the VTK output from PFLOTRAN. :param self: DFN Class :type self: object :param grid_vtk_file: Name of vtk file with mesh. Typically local_dfnFlow_file.vtk :type grid_vtk_file: string

Returns

Return type None

Notes

If DFN class does not have a vtk file, inp2vtk_python is called

pydfnworks.dfnFlow.pflotran.**pflotran**(*self*, *transient=False*, *restart=False*, *restart_file=''*)
Run PFLOTRAN. Copy PFLOTRAN run file into working directory and run with ncpus

Parameters

- **self** (*object*) – DFN Class
- **transient** (*bool*) – Boolean if PFLOTRAN is running in transient mode
- **restart** (*bool*) – Boolean if PFLOTRAN is restarting from checkpoint
- **restart_file** (*string*) – Filename of restart file

Returns

Return type None

Notes

Runs PFLOTRAN Executable, see http://www.pflotran.org/ for details on PFLOTRAN input cards

pydfnworks.dfnFlow.pflotran.**pflotran_cleanup**(*self*, *index_start=0*, *index_finish=1*, *filename=''*)
Concatenate PFLOTRAN output files and then delete them

Parameters

- **self** (*object*) – DFN Class
- **index** (*int*) – If PFLOTRAN has multiple dumps use this to pick which dump is put into cellinfo.dat and darcyvel.dat

Returns

Return type None

Notes

Can be run in a loop over all pflotran dumps

pydfnworks.dfnFlow.pflotran.**write_perms_and_correct_volumes_areas**(*self*)
Write permeability values to perm_file, write aperture values to aper_file, and correct volume areas in uge_file

Parameters **self** (*object*) – DFN Class

Returns

Return type None

Notes

Calls executable correct_uge

pydfnworks.dfnFlow.pflotran.**zone2ex**(*self*, *uge_file=''*, *zone_file=''*, *face=''*, *boundary_cell_area=0.1*)
Convert zone files from LaGriT into ex format for LaGriT

Parameters

- **self** (*object*) – DFN Class
- **uge_file** (*string*) – Name of uge file
- **zone_file** (*string*) – Name of zone file
- **Face** (*Face of the plane corresponding to the zone file*) –
- **zone_file** – Name of zone file to work on. Can be 'all' processes all directions, top, bottom, left, right, front, back
- **boundary_cell_area** (*double*) – should be a large value relative to the mesh size to force pressure boundary conditions.

Returns

Return type None

Notes

the boundary_cell_area should be a function of h, the mesh resolution

7.3 Running Flow: FEHM

pydfnworks.dfnFlow.fehm.**correct_stor_file**(*self*)
Corrects volumes in stor file to account for apertures

Parameters **self** (*object*) – DFN Class

Returns

Return type None

Notes

Currently does not work with cell based aperture

pydfnworks.dfnFlow.fehm.**fehm**(*self*)
Run FEHM

Parameters **self** (*object*) – DFN Class

Returns

Return type None

Notes

See https://fehm.lanl.gov/ for details about FEHM

7.4 Processing Flow

pydfnworks.dfnFlow.mass_balance.**effective_perm**(*self*, *inflow_pressure*, *outflow_pressure*, *boundary_file*, *direction*)

Computes the effective permeability of a DFN in the primary direction of flow using a steady-state PFLOTRAN solution.

Parameters

- **self** (*object*) – DFN Class
- **inflow_pressure** (*float*) – Pressure at the inflow boundary face. Units are Pascal
- **outflow_pressure** (*float*) – Pressure at the outflow boundary face. Units are Pascal
- **boundary_file** (*string*) – Name of inflow boundary file, e.g., pboundary_left.ex
- **direction** (*string*) – Primary direction of flow, x, y, or z

Returns

Return type None

Notes

1. Information is written to screen and to the file self.local_jobname_effective_perm.txt
2. Currently, only PFLOTRAN solutions are supported
3. Assumes density of water at 20c

CHAPTER

EIGHT

PYDFNWORKS: DFNTRANS

DFN Class functions used in particle transport simulations (DFNTrans)

8.1 Running Transport Simulations

pydfnworks.dfnTrans.transport.**check_dfn_trans_run_files**(*self*)
Ensures that all files required for dfnTrans run are in the current directory

Parameters **self** (*object*) – DFN Class

Returns

Return type None

Notes

None

pydfnworks.dfnTrans.transport.**copy_dfn_trans_files**(*self*)
Creates symlink to dfnTrans Execuateble and copies input files for dfnTrans into working directory

Parameters **self** (*object*) – DFN Class

Returns

Return type None

pydfnworks.dfnTrans.transport.**create_dfn_trans_links**(*self*, *path='../'*)
Create symlinks to files required to run dfnTrans that are in another directory.

Parameters

- **self** (*object*) – DFN Class
- **path** (*string*) – Absolute path to primary directory.

Returns

Return type None

Notes

Typically, the path is DFN.path, which is set by the command line argument -path

pydfnworks.dfnTrans.transport.**dfn_trans**(*self*)
Primary driver for dfnTrans.

Parameters **self** (*object*) – DFN Class

Returns

Return type None

pydfnworks.dfnTrans.transport.**run_dfn_trans**(*self*)
Execute dfnTrans

Parameters **self** (*object*) – DFN Class

Returns

Return type None

CHAPTER

NINE

PYDFNWORKS: DFNGRAPH

DFN Class functions used in graph analysis and pipe-network simulations

9.1 General Graph Functions

pydfnworks.dfnGraph.dfn2graph.**add_fracture_source**(*self*, *G*, *source*)

Returns the k shortest paths in a graph

Parameters

- **G** (*NetworkX Graph*) – NetworkX Graph based on a DFN
- **source_list** (*list*) – list of integers corresponding to fracture numbers
- **remove_old_source** (*bool*) – remove old source from the graph

Returns G

Return type NetworkX Graph

Notes

bipartite graph not supported

pydfnworks.dfnGraph.dfn2graph.**add_fracture_target**(*self*, *G*, *target*)

Returns the k shortest paths in a graph

Parameters

- **G** (*NetworkX Graph*) – NetworkX Graph based on a DFN
- **target** (*list*) – list of integers corresponding to fracture numbers

Returns G

Return type NetworkX Graph

Notes

bipartite graph not supported

pydfnworks.dfnGraph.dfn2graph.**create_graph**(*self*, *graph_type*, *inflow*, *outflow*)
Header function to create a graph based on a DFN

Parameters

- **self** (*object*) – DFN Class object
- **graph_type** (*string*) – Option for what graph representation of the DFN is requested. Currently supported are fracture, intersection, and bipartitie
- **inflow** (*string*) – Name of inflow boundary (connect to source)
- **outflow** (*string*) – Name of outflow boundary (connect to target)

Returns **G** – Graph based on DFN

Return type NetworkX Graph

Notes

pydfnworks.dfnGraph.dfn2graph.**dump_fractures**(*self*, *G*, *filename*)
Write fracture numbers assocaited with the graph G out into an ASCII file inputs

Parameters

- **self** (*object*) – DFN Class
- **G** (*NetworkX graph*) – NetworkX Graph based on the DFN
- **filename** (*string*) – Output filename

Notes

pydfnworks.dfnGraph.dfn2graph.**dump_json_graph**(*self*, *G*, *name*)
Write graph out in json format

Parameters

- **self** (*object*) – DFN Class
- **G** (*networkX graph*) – NetworkX Graph based on the DFN
- **name** (*string*) – Name of output file (no .json)

Notes

pydfnworks.dfnGraph.dfn2graph.**greedy_edge_disjoint**(*self*, *G*, *source='s'*, *target='t'*, *weight='None'*, *k=''*)
Greedy Algorithm to find edge disjoint subgraph from s to t. See Hyman et al. 2018 SIAM MMS

Parameters

- **self** (*object*) – DFN Class Object
- **G** (*NetworkX graph*) – NetworkX Graph based on the DFN
- **source** (*node*) – Starting node

- **target** (*node*) – Ending node
- **weight** (*string*) – Edge weight used for finding the shortest path
- **k** (*int*) – Number of edge disjoint paths requested

Returns **H** – Subgraph of G made up of the k shortest of all edge-disjoint paths from source to target

Return type NetworkX Graph

Notes

1. Edge weights must be numerical and non-negative.
2. See Hyman et al. 2018 "Identifying Backbones in Three-Dimensional Discrete Fracture Networks: A Bipartite Graph-Based Approach" SIAM Multiscale Modeling and Simulation for more details

pydfnworks.dfnGraph.dfn2graph.**k_shortest_paths_backbone**(*self*, *G*, *k*, *source='s'*, *target='t'*, *weight=None*)

Returns the subgraph made up of the k shortest paths in a graph

Parameters

- **G** (*NetworkX Graph*) – NetworkX Graph based on a DFN
- **k** (*int*) – Number of requested paths
- **source** (*node*) – Starting node
- **target** (*node*) – Ending node
- **weight** (*string*) – Edge weight used for finding the shortest path

Returns **H** – Subgraph of G made up of the k shortest paths

Return type NetworkX Graph

Notes

See Hyman et al. 2017 "Predictions of first passage times in sparse discrete fracture networks using graph-based reductions" Physical Review E for more details

pydfnworks.dfnGraph.dfn2graph.**load_json_graph**(*self*, *name*)

Read in graph from json format

Parameters

- **self** (*object*) – DFN Class
- **name** (*string*) – Name of input file (no .json)

Returns **G** – NetworkX Graph based on the DFN

Return type networkX graph

pydfnworks.dfnGraph.dfn2graph.**plot_graph**(*self*, *G*, *source='s'*, *target='t'*, *output_name='dfn_graph'*)

Create a png of a graph with source nodes colored blue, target red, and all over nodes black

Parameters

- **G** (*NetworkX graph*) – NetworkX Graph based on the DFN
- **source** (*node*) – Starting node
- **target** (*node*) – Ending node

- **output_name** (*string*) – Name of output file (no .png)

Notes

Image is written to output_name.png

9.2 Graph-Based Flow and Transport

pydfnworks.dfnGraph.graph_flow.**run_graph_flow**(*self*, *inflow*, *outflow*, *Pin*, *Pout*, *fluid_viscosity=0.00089*, *phi=1*, *G=None*)

Run the graph flow portion of the workflow

Parameters **self** – DFN Class

Returns **Gtilde** – Grtilde is updated with vertex pressures, edge fluxes and travel times

Return type NetworkX graph

Notes

Information on individual functions in found therein

pydfnworks.dfnGraph.graph_transport.**run_graph_transport**(*self*, *G*, *nparticles*, *partime_file*, *frac_id_file*, *initial_positions='uniform'*, *tdrw_flag=False*, *matrix_porosity=None*, *matrix_diffusivity=None*)

Run particle tracking on the given NetworkX graph

Parameters

- **self** (*object*) – DFN Class
- **G** (*NetworkX graph*) – obtained from graph_flow
- **nparticles** (*int*) – number of particles
- **initial_positions** (*str*) – distribution of initial conditions. options are uniform and flux (flux-weighted)
- **partime_file** (*string*) – name of file to which the total travel times and lengths will be written for each particle
- **frac_id_file** (*string*) – name of file to which detailed information of each particle's travel will be written
- **tdrw_flag** (*Bool*) – if False, matrix_porosity and matrix_diffusivity are ignored
- **matrix_porosity** (*float*) – Matrix Porosity used in TDRW
- **matrix_diffusivity** (*float*) – Matrix Diffusivity used in TDRW (SI units m^2/s)

Returns

Return type O if completed correctly

Notes

Information on individual functions is found therein

CHAPTER

TEN

PYDFNWORKS: WELL PACKAGE

DFN Class functions used for well package

10.1 dfnWorks - Well Package

pydfnworks.dfnGen.well_package.wells.**cleanup_wells**(*self*, *wells*)

Moves working files created while making wells into well_data directory

Parameters

- **self** (*object*) – DFN Class
- **well** – dictionary of information about the well. Contains the following:

  well["name"] [string] name of the well

  well["filename"] [string]

  filename of the well coordinates. "well_coords.dat" for example. Format is : x0 y0 z0 x1 y1 z1 … xn yn zn

  well["r"] [float] radius of the well

Returns

Return type None

Notes

Wells can be a list of well dictionaries

pydfnworks.dfnGen.well_package.wells.**combine_well_boundary_zones**(*self*, *wells*)

Processes zone files for particle tracking. All zone files are combined into allboundaries.zone

Parameters **None** –

Returns

Return type None

Notes

None

pydfnworks.dfnGen.well_package.wells.**find_well_intersection_points**(*self*, *wells*)

Identifies points on a DFN where the well intersects the network. These points are used in meshing the network to have higher resolution in the mesh in these points. Calls a sub-routine run_find_well_intersection_points.

Parameters

- **self** (*object*) – DFN Class
- **well** – dictionary of information about the well. Contains the following:

  well["name"] [string] name of the well

  well["filename"] [string]

  filename of the well coordinates. "well_coords.dat" for example. Format is : x0 y0 z0 x1 y1 z1 … xn yn zn

  well["r"] [float] radius of the well

Returns

Return type None

Notes

Wells can be a list of well dictionaries. Calls the subroutine run_find_well_intersection_points to remove redundant code.

pydfnworks.dfnGen.well_package.wells.**tag_well_in_mesh**(*self*, *wells*)

Identifies nodes in a DFN for nodes the intersect a well with radius r [m]

1. Well coordinates in well["filename"] are converted to a polyline that are written into "well_{well['name']}_line.inp"
2. Well is expanded to a volume with radius well["r"] and written into the avs file well_{well["name"]}_volume.inp
3. Nodes in the DFN that intersect with the well are written into the zone file well_{well["name"]}.zone
4. If using PFLOTRAN, then an ex file is created from the well zone file

Parameters

- **self** (*object*) – DFN Class
- **well** – Dictionary of information about the well that contains the following attributes

  well["name"] [string] name of the well

  well["filename"] [string] filename of the well coordinates with the following format x0 y0 z0

  x1 y1 z1

  …

  xn yn zn

Returns

Return type None

Notes

Wells can be a list of well dictionaries

CHAPTER

ELEVEN

DFNGEN

dfnGen creates the discrete fracture networks using the feature rejection algorithm for meshing (FRAM). Fractures can be created stochastically or as deterministic features.

The detailed description of FRAM and the implemented methodology is in J. D. Hyman, C. W. Gable, S. L. Painter, and N. Makedonska. Conforming Delaunay triangulation of stochastically generated three dimensional discrete fracture networks: A feature rejection algorithm for meshing strategy. SIAM J. Sci. Comput., 36(4):A1871–A1894, 2014.

11.1 Domain Parameters

The following parameters define the domain. All units are in SI (meters for length).

11.1.1 domainSize

Description: (Mandatory) Defines the domain size, which is centered at the origin (0,0,0). The first entry is span in x (east/west), the second entry is span in y (North/South), and the third entry is span in z (Top/Bottom).

Type: Array of three doubles

Example:

```
domainSize: {10,5,20}
// Create a domain of 10 m by 5 m by 20.
// Minimum/Maximum x is -5/+5
// Minimum/Maximum in y is -2.5/+2.5
// Minimum/Maximum in z is -10/+10.
```

Note: The minimum and maximum in each direction is 1/2 the input value.

11.1.2 domainSizeIncrease

Description: (Mandatory) Temporary *domainSize* increase for inserting fracture centers outside of the domain defined by *domainSize*. After generation is complete, the domain is truncated back to *domainSize*. First entry is expansion in x (east/west), second entry is expansion in y (North/South), and third entry is expansion in z (Top/Bottom). This is used to help mitigate edge density effects.

Type: Array of three doubles

Example:

```
domainSizeIncrease:{2,1,5}
// Increase the domain-size by:
// adding 1 to the +x, and subtracting 1 to the -x
// adding 0.5 to +y, and subtracting -0.5 to -y
// adding 2.5 to +z, and subtracting -2.5 to -z
```

Note: The domain size increase in each direction must be less than 1/2 the domain size in that direction.

Tip: A good rule of thumb is to set the expansion length to be *at least* the radius of the largest fracture.

11.1.3 numOfLayers

Description: (Mandatory) Defines the number of stratographic layers in the domain. If numOfLayers is 0, then there are no layers. For N layers, there must be N sets of minimum and maximum heights defined in *layers*. Each stochastic fracture set is assigned to a layer using *eLayer* for ellipses and *rLayer* for rectangles.

Type: Non-Negative Integer ($N > 0$)

```
numOfLayers: 2 // There will be two layers in the domain
```

11.1.4 layers

Description. Defines the lower and upper limits for each layer. The first layer listed is layer 1, the second is layer 2, etc. Every stochastic families *must* be assigned to a layer. If the family is assigned to layer 0, then the family in generated through the entire domain.

Type: Set of *numOfLayers* arrays with two elements. {zMin, zMax}

Example:

```
layers:
{-50, -30} // Minimum and Maximum height of layer 1 is -50 m and -30 m
{10, 40} // Minimum and Maximum height of layer 2 is 10 m and 40 m
```

Note:

Each layer must be on a new line

First entry (zMin) must be less than second entry (zMax)
Layers can overlap

11.1.5 numOfRegions

Description: (Mandatory) Defines the number of cuboid regions in the domain. If numOfRegions is 0, then there are no regions. There must be *N* sets of defined by *regions*. Each stochastic fracture set is assigned to a region using *eRegion* for ellipses and *rRegion* for rectangles.

Type: Non-Negative Integer ($N > 0$)

```
numOfRegions: 1 // There will be one region in the domain
```

11.1.6 regions

Description: (Mandatory) Defines the bounding box of each region. The first region listed is region 1, the region is region 2, etc. Stochastic families *must* be assigned to theses regions. If the family is assigned to region 0, then the family in generated through the entire domain.

Type: Set of *numOfRegions* arrays with six elements. {minX, maxX, minY, maxY, minZ, maxZ}.

Example:

```
regions:
{-5, 5, -10, 10, -20, 10}
// Will create a region for sampling with
// x-min: -5, x-max: 5
// y-min: -10, y-max: 10
// z-min: -20, z-max: 10
```

Note:

Each region must be on a new line.
Min/Max values for each direction do not need to be the same.
Minimum value must be less than the maximum value in each direction
Regions can overlap

11.1.7 ignoreBoundaryFaces

Description: Selection of using the boundary faces option.

Type: boolean (0/1)

0: use *boundaryFaces* option
1: ignore *boundaryFaces* option and keep all clusters

Warning: All clusters are retained only if *keepOnlyLargestCluster* is set to 0.

11.1.8 boundaryFaces

Description: (Mandatory) Selects domain boundaries for flow. The generation will only keep clusters of fractures with connections to domain boundaries which are set to 1.

Type: Array of six boolean values corresponding to each face of the domain.

boundaryFaces[1] = +X domain boundary
boundaryFaces[2] = -X domain boundary
boundaryFaces[3] = +Y domain boundary
boundaryFaces[4] = -Y domain boundary
boundaryFaces[5] = +Z domain boundary
boundaryFaces[6] = -Z domain boundary

Example:

```
boundaryFaces: {1,1,0,0,0,0} // Keep fractures within a cluster that connect the X␣
↪boundaries
```

Warning: Set *ignoreBoundaryFaces* to 0 when using this feature

---

11.2 General Network Generation Parameters

The following parameters define the general network properties. Individual fracture family parameters are described in *Fracture Family Generation Parameters: Ellipse*, and *Fracture Family Generation Parameters: Rectangle*.

11.2.1 stopCondition

Description: (Mandatory) Selection criteria for when network generation stops.

Type: boolean (0/1)

0: stop generation once *nPoly* fractures are accepted
1: stop generation once all fracture family p32 values have been meet

Note: p32 values are defined for each family in *e_p32Targets* for ellipses and *r_p32Targets* for rectangles

11.2.2 nPoly

Description: The total number of fractures requested in the domain. dfnGen will stop generation once there are *nPoly* number of fractures.

Type: Positive Integer (*nPoly* > 0)

```
nPoly: 100 // Stop generation once 100 fractures are accepted into the network
```

Note: Only used if *stopCondition* is set to 0

11.2.3 famProb

Description: Probability of occurrence for each family of stochastically generated fractures. Values of *famProb* elements must add up to 1.0. The probabilities are listed in order of families starting with all stochastic ellipses, and then all stochastic rectangles.

Type: Array of length number of stochastic fracture families (*nFamEll* + *nFamRect*)

Example:

Listing 1: If there are two ellipse families, each with probability 30%, and two rectangle families, each with probability 20%:

```
famProb: {0.3,0.3,0.2,0.2}
```

Note: User defined ellipses, rectangles, and polygons are inserted into the domain prior to any stochastic fractures. However, there is a possibility they will be rejected if FRAM constraints are not met.

11.2.4 orientationOption

Description: (Mandatory) Selection of fracture family orientation definition.

Type: Value of 0,1,2

0 : Spherical Coordinates
1 : Trend / Plunge
2 : Dip / Strike

```
orientationOption: 1 // Fracture Orientations will be defined using Trend/Plunge
```

Note:

For spherical coordinates, values are defined using *etheta*/*ephi* for ellipses and *rtheta*/*rphi*.
For Trend/Plunge, values are defined using *etrend*/*eplunge* for ellipses and *rtrend*/*rplunge*.
For Dip/Strike, values are defined using *edip*/*estrike* for ellipses and *rdip*/*estrike*.

Warning:

When using Trend / Plunge or Dip / Strike, *eAngleOption*/*rAngleOption* must be set to degrees, *eAngleOption*/*rAngleOption* = 1

11.2.5 h

Description: (Mandatory) Minimum feature size accepted into the network.

Type: Positive Double

Note:

The following constraints are imposed on h to keep the final mesh size reasonable, unless *disableFram* is turned on.

1. h must be greater than $10^{-5} \cdot \sqrt{x^2 + y^2 + z^2}$ where x,y,z are the elements of *domainSize*.
2. h must be smaller than 1/10th the minimum fracture size
3. h must be larger than 1/1000th than minimum fracture size
4. h must be non-zero

Warning: A clear understanding of h is required for network generation and meshing. Refer to J. D. Hyman, C. W. Gable, S. L. Painter, and N. Makedonska. Conforming Delaunay triangulation of stochastically generated three dimensional discrete fracture networks: A feature rejection algorithm for meshing strategy. SIAM J. Sci. Comput., 36(4):A1871–A1894, 2014. for a complete discussion of h.

11.2.6 disableFram

Danger: disableFram: If FRAM is turned off (disableFram: 1) the resulting network can only be meshed using the coarse visual mode. You cannot mesh DFN or run flow and transport if the network is generated with disableFram: 1.

Description: (Mandatory) Turn FRAM on/off. Having FRAM on is required for full DFN capabilities of meshing, flow, and transport. If FRAM is off then capabilities will be limited.

Type: boolean (0/1)

0: FRAM is on
1: FRAM is off

Note: disableFram: 1 and *visualizationMode* :1 are recommended if the intention is to use an upscaled octree mesh using the *UDFM* module in pydfnWorks.

11.2.7 printRejectReasons

Description: (Mandatory) Option to print rejection information to screen during network generation.

Type: boolean (0/1)

0: off. Limited rejection information will be printed to screen during generation.
1: on. Detailed fracture information will be printed to screen during generation.

Tip: Turning this feature on is useful for debugging and initial network construction. Having this turned off is more efficient for network generation.

11.2.8 rejectsPerFracture

Description: (Mandatory) If fracture is rejected, it will be re-translated to a new position *rejectsPerFracture* number of times. Increasing this value can help hit distribution targets for stochastic families.

Type: Positive Integer

```
rejectsPerFracture: 10 // If a fracture is rejected, it will be translated to a new␣
↪point in the domain 10 times before being completely rejected
```

Note: Suggested default is 10. Set equal to 1 to ignore.

11.2.9 radiiListIncrease

Description: (Mandatory) Increases the length of the radii list in the sampling queue by this percentage. Fracture radii are sampled and ordered (largest to smallest) prior to beginning network generation. If the target distributions are not being properly resolved due to rejection, then increasing this value can help provide a more uniform representation of the distribution. Once the original list is exhausted, then fracture radii are sampled from the distribution at random and only smaller fractures are likely to be accepted.

Type: Positive Double

Example:

```
radiiListIncrease: 0.1 // Increase the length of the possible samples by 10%.
```

Note: Set to 0 to ignore.

Tip: Examine the dfnGen output report to check if the prescribed distributions are being properly resolved. Run DFN.output_report() in the python work file after generation is complete to generate the output report.

11.2.10 visualizationMode

Warning: No longer supported. Selection of visual mode should be done using the pydfnWorks function DFN.mesh_network(visual_mode=True)

Description: (Mandatory) Selection if you want to mesh to be coarse, for quick visualization but cannot run flow and transport, or standard mesh.

Type: boolean (0/1)

0: Create full DFN mesh (full_mesh.inp) for flow and transport

1: Create reduced DFN mesh (reduced_mesh.inp) for quick visualization

Note: Default should be 1. However, the selection of this value does not affect network generation.

11.2.11 seed

Description: (Mandatory) Seed for random generator. Setting the seed equal to 0 will seed off the clock and a unique network will be produced. Setting the seed equal to a value > 0 will create the same network every time, which is useful for reproducibility.

Type: Non-negative integer

Tip: If you set seed to 0, the seed used in the generation is saved in the file DFN_output.txt created by dfnGen.

11.2.12 keepOnlyLargestCluster

Description: (Mandatory) Selection to retain multiple clusters that connect boundaries or only the largest cluster. The largest cluster is defined by the number of fractures in the cluster.

Type: boolean (0/1)

0: Keep all clusters that connect the specified boundary faces in *boundaryFaces*
1: Keep only the largest cluster that connects the specified boundary faces in *boundaryFaces*

11.2.13 keepIsolatedFractures

Description: (Mandatory) Selection to keep isolated fractures in the domain after generation is complete.

Type: boolean (0/1)

0: Remove isolated fractures from the domain
1: Keep isolated in the domain.

Note: Isolated fractures do not contribute to flow in the DFN as they are not connected to flow boundaries. If you are running a DFN, keepIsolatedFractures should be set to 0. You can keep isolated fractures in the domain for UDFM meshing.

Danger: Full DFN-meshing will fail if isolated fractures are not removed. Reduced meshing for visualization can still be performed.

11.2.14 tripleIntersections

Description: (Mandatory) Selection of whether triple intersection are accepted into the network.

Type: boolean (0/1)

0: Reject all triple intersections
1: Accept triple intersections that meet FRAM criteria.

Note: Even if *tripleIntersections: 1*, triple intersections can be rejected if they create a feature on the network smaller than h.

Warning: dfnTrans does not support triple intersections.

11.2.15 removeFracturesLessThan

Description: (Mandatory) All fractures with radius less than *removeFracturesLessThan* are removed from the network after generation is complete.

Type: Non-Negative double

Example:

```
removeFracturesLessThan: 5 // Remove all fracture with radius less than 5 meters.
```

Note: The lower cutoff of fracture size is defined using fracture family generation, e.g., *emin* for ellipses sampled from a truncated powerlaw. If this parameter is non-zero, then the network will be generated with fractures down to the lower cutoff, but only those with a radius greater than removeFracturesLessThan will be output for meshing.

11.2.16 insertUserRectanglesFirst

Description: (Mandatory) Select order for how user defined rectangles and user defined ellipses are inserted into the domain.

Type: boolean (0/1)

0: Insert user defined ellipses first
1: Insert user defined rectangles first

Note: User defined fractures (ellipses, rectangles, and polygons) are *always* inserted prior to stochastic fractures.

11.2.17 forceLargeFractures

Description: (Mandatory) Insert the largest fracture from each family into the domain prior to sampling sequential from family based on their respective probabilities.

Type: boolean (0/1)

0: Do not force the largest fractures
1: Force the largest fractures

Warning: No Longer Supported. Fractures are sorted by size prior to being inserted into the domain. Larger fractures are inserted first to minimize rejections.

11.3 General Network Output Parameters

11.3.1 outputAllRadii

Description: (Mandatory) Create an output file of all fracture radii, both accepted and rejected fractures.

Filename: radii_AllAccepted.dat
Format: xRadius yRadius Distribution # (-2 = userPolygon, -1 = userRectangle, 0 = userEllipse, > 0 is family in order of famProb)

Type: boolean (0/1)

0: Do not create file
1: Create file

11.3.2 outputAcceptedRadiiPerFamily

Description: (Mandatory) Create one file that contains the radius of every fracture per family prior to isolated fractures removed.

Filename: radii/radii_AllAccepted_Fam_1.dat
Format: xRadius yRadius DistributionNumber (-2 = userPolygon, -1 = userRectangle, 0 = userEllipse, > 0 is family in order of famProb)

Type: boolean (0/1)

0: Do not create file
1: Create file

11.3.3 outputFinalRadiiPerFamily

Description: (Mandatory) Create one file that contains the radius of every fracture per family after to isolated fractures removed.

Filename: radii/radii_Final_Fam_1.dat
Format: xRadius yRadius DistributionNumber (-2 = userPolygon, -1 = userRectangle, 0 = userEllipse, > 0 is family in order of famProb)

Type: boolean (0/1)

0: Do not create file
1: Create file

11.4 Fracture Family Generation Parameters: Ellipse

This section describes generation parameters for families of disc-shaped fractures. The number of elements in each parameter must match number of ellipse families defined by *nFamEll*. The index of the elements corresponds to the fracture family. The first element of each parameter corresponds to the first family, the second element corresponds to the second family, etc.

11.4.1 Ellipse: General Parameters

nFamEll

Description: (Mandatory) Number of ellipse families

Type: Non-Negative Integer

```
nFamEll: 3 // There will be 3 ellipse families
```

Note: If this option is set to 0, then all ellipses parameters will be ignored.

---

eLayer

Description: Assign each ellipse family to a layer in domain.

Type: Array of *nFamEll* integers

Example:

```
eLayer: {0,2,1}
// Family 1 is assigned to the whole domain
// Family 2 is assigned to layer 2
// Family 3 is assigned to layer 1
```

Note: Layer 0 is the entire domain. Numbers > 0 correspond to those defined in *layers*.

Warning: Families can only be assigned to either a layer or a region, not both.

---

eRegion

Description: Assign each ellipse family to a region in domain.

Type: Array of *nFamEll* integers

Example:

```
eRegion: {0,2,1}
// Family 1 is assigned to the whole domain
// Family 2 is assigned to region 2
// Family 3 is assigned to region 1
```

Note: Region 0 is the entire domain. Numbers > 0 correspond to those defined in *regions*.

Warning: Families can only be assigned to either a layer or a region, not both.

---

e_p32Targets

Description: Target fracture intensity per family. Fracture intensity $P_{32}[\mathrm{m}^{-1}]$ is defined as total surface area of each fracture family divided by the total domain volume. Fractures from each family are inserted into the domain until provided target values are obtained. Generation stops once all desired fracture intensity are obtained.

Type: Array of *nFamEll* doubles

Example:

```
e_p32Targets: {0.02,0.4,0.05}
// Family 1 has a target p32 of 0.02
// Family 2 has a target p32 of 0.4
// Family 3 has a target p32 of 0.05
```

Note: Only used when *stopCondition* = 1

Warning: The fracture surface area is defined using *both* sides of a fracture.

---

enumPoints

Description: Number of vertices defining the boundary of each elliptical fracture

Type: Array of *nFamEll* integers

Example:

```
enumPoints: {8,12,16}
// Fractures from family 1 are defined using 8 points
// Fractures from family 2 are defined using 12 points
// Fractures from family 3 are defined using 16 points
```

Note:

1. Values must be greater than 4, which corresponds to a rectangle
2. Increasing this value lead to more challenging acceptance criteria via FRAM due to smaller edge lengths between vertices on the fracture boundary
3. Suggested value: 8

---

easpect

Description: Aspect ratio of fractures

Type: Array of *nFamEll* doubles

Example:

```
easpect: {1,2,0.5}
// Family 1 has an aspect ratio of 1 (circles)
// Family 2 has an aspect ratio of 2 - y radius is twice the x radius
// Family 3 has an aspect ratio of 0.5 - y radius is 1/2 the x radius
```

Note: A value of 1 makes circles

Tip: As the aspect ratio increases, the shape of the ellipse can degrade accuracy unless *enumPoints* is also increased

11.4.2 Ellipse: Fracture Orientation

The fracture orientations are sampled from the three-dimensional von Mises-Fisher distribution,

$$f(\mathbf{x}; \boldsymbol{\mu}, \kappa) = \frac{\kappa \exp(\kappa \boldsymbol{\mu}^T \mathbf{x})}{4\pi \sinh(\kappa)} .$$

where $\boldsymbol{\mu}$ is the mean orientation vector and T denotes transpose. The distribution is sampled using the algorithm provided Simulation of the von Mises Fisher distribution Communications in statistics-simulation and computation 23.1 (1994): 157-164. by Andrew Wood..

dfnGen accepts spherical coordinates (θ/ϕ), Trend/Plunge, or Dip/Strike to define the mean orientation of a fracture family.

Tip: *orientationOption* selects which of these options are used. The same option must be used for all families.

0 : Spherical Coordinate
1 : Trend / Plunge
2 : Dip / Strike.

eAngleOption

Description: Selection of angle option for ellipse families

Type: boolean (0/1)

0 - radians (Must use numerical value for π)
1 - degrees

Note: Use of radians is only supported for spherical coordinates (*etheta*/*ephi*). Degrees must be used for *etrend*/*eplunge* and *edip*/*estrike*

ekappa

Description: The concentration parameter of the von Mises-Fisher distribution, which determines the degree of clustering around the mean orientation.

Type: Array of *nFamEll* doubles

Example:

```
ekappa: {0.1, 20, 17}
// Fracture Family 1 has a theta value of 45 degrees
// Fracture Family 2 has a theta value of 78 degrees
// Fracture Family 3 has a theta value of 0 degrees
```

Note: Values of κ approaching zero result in a uniform distribution of points on the sphere. Larger values create points with a small deviation from mean direction.

Warning: The numerical method for sampling the von Mises-Fisher distribution becomes unstable for values greater than 100.

Ellipse: Spherical Coordinates

The mean normal vector of the fracture family $\vec{n}$ is related to the spherical coordinates θ and ϕ by

$$\vec{n}_x = \sin(\theta)\cos(\phi)$$
$$\vec{n}_y = \sin(\theta)\sin(\phi) \tag{11.2}$$
$$\vec{n}_z = \cos(\theta) \tag{11.3}$$

etheta

Description: Angle the normal vector of the fracture makes with the z-axis

Type: Array of *nFamEll* double

Example:

```
etheta: {45, 78, 0}
// Fracture Family 1 has a theta value of 45 degrees
// Fracture Family 2 has a theta value of 78 degrees
// Fracture Family 3 has a theta value of 0 degrees
```

Note: Both radians and degrees are supported. Use *eAngleOption* to select one. If *eAngleOption* is set to radians *eAngleOption* : 0, and the value provided must be less than 2π.

ephi

Description: Angle that the projection of the normal vector of a fracture onto the x-y plane makes with the x-axis.

Type: Array of *nFamEll* double

Example:

```
etheta: {0, 56, 12}
// Fracture Family 1 has a phi value of 0 degrees
// Fracture Family 2 has a phi value of 56 degrees
// Fracture Family 3 has a phi value of 12 degrees
```

Note: Both radians and degrees are supported. Use *eAngleOption* to select one. If *eAngleOption* is set to radians *eAngleOption* : 0, then the value provided must be less than 2π.

Ellipse: Trend & Plunge

The mean normal vector of the fracture family $\vec{n}$ is related to trend and plunge by

$$\vec{n}_x = \cos(\text{trend})\cos(\text{plunge})$$
$$\vec{n}_y = \sin(\text{trend})\cos(\text{plunge}) \quad (11.5)$$
$$\vec{n}_z = \sin(\text{trend}) \quad (11.6)$$

etrend

Description: Trend of fracture families

Type: Array of *nFamEll* double

Example

```
etrend: {0, 56, 12}
// Fracture Family 1 has a trend value of 0 degrees
// Fracture Family 2 has a trend value of 56 degrees
// Fracture Family 3 has a trend value of 12 degrees
```

Note: *eAngleOption* must be set to degree (eAngleOption: 1) to use Trend & Plunge

---

eplunge

Description: Plunge of fracture families

Type: Array of *nFamEll* double

Example

```
eplunge: {0, 56, 12}
// Fracture Family 1 has a plunge value of 0 degrees
// Fracture Family 2 has a plunge value of 56 degrees
// Fracture Family 3 has a plunge value of 12 degrees
```

Note: *eAngleOption* must be set to degree (eAngleOption: 1) to use Trend & Plunge

---

Ellipse: Dip & Strike

The mean normal vector of the fracture family $\vec{n}$ is related to dip and strike by

$$\vec{n}_x = \sin(\text{dip})\sin(\text{strike})$$
$$\vec{n}_y = -\sin(\text{dip})\cos(\text{strike}) \tag{11.8}$$
$$\vec{n}_z = \cos(\text{dip}) \tag{11.9}$$

---

estrike

Description: Strike of fracture families

Type: Array of *nFamEll* double

Example

```
estrike: {0, 56, 12}
// Fracture Family 1 has a strike value of 0 degrees
// Fracture Family 2 has a strike value of 56 degrees
// Fracture Family 3 has a strike value of 12 degrees
```

Note: *eAngleOption* must be set to degree (eAngleOption: 1) to use Dip & Strike

---

edip

Description: Dip of fracture families

Type: Array of *nFamEll* double

Example

```
edip: {0, 56, 12}
// Fracture Family 1 has a dip value of 0 degrees
// Fracture Family 2 has a dip value of 56 degrees
// Fracture Family 3 has a dip value of 12 degrees
```

Note: *eAngleOption* must be set to degree (eAngleOption: 1) to use Dip & Strike

---

Ellipse: In Plane Rotation

ebetaDistribution

Description: Prescribe a rotation around each fracture's normal vector, with the fracture centered on x-y plane at the origin

Type: Array of *nFamEll* boolean values

0: Uniform distribution on [0, 2π)
1: Constant rotation specified by *ebeta*

```
ebetaDistribution: {0, 1, 1}
// Fracture Family 1 will have a random rotation
// Fracture Family 2 will have a constant angle of rotation defined in the first entry␣
↪of ebeta
// Fracture Family 3 will have a constant angle of rotation defined in the second entry␣
↪of ebeta
```

---

ebeta

Description: Values for constant angle of rotation around the normal vector

Type: Array of boolean (0/1)

Example:

```
ebetaDistribution: {45, 270} // For ebetaDistribution: {0, 1, 1}
// Fracture Family 2 will have a constant angle of rotation of 45 degrees
// Fracture Family 3 will have a constant angle of rotation of 270 degrees
```

Note:

1. The length of ebeta corresponds to the number of *non-zero* entries in *ebetaDistribution*
2. *eAngleOption* defines if the values are in radians or degrees

---

11.4.3 Ellipse: Fracture Radius Distributions

Fracture radii can be defined using four different distributions: (1) Log-Normal, (2) Truncated Power-law, (3) Exponential, and (4) Constant. Minimum and maximum values must be provided for 1-3 along with the distribution parameters.

---

edistr

Description: Assigns fracture radius distribution for each family

Type: Array of *nFamEll* Integers (1,2,3,4)

Example:

```
edistr: {1, 2, 4}
// Fracture Family 1 will use a LogNormal Distribution
// Fracture Family 2 will use a Truncated powerlaw distribution
// Fracture Family 3 will have a constant sized fractures
```

Note: Number of elements in the parameters for each distribution must match number of families assigned to that distribution.

Ellipse: Lognormal Distribution

Fracture radii are sampled from a Lognormal distribution with the following probability density function

$$\frac{1}{x\sigma\sqrt{2\pi}} \exp\left(-\frac{(\ln x - \mu)^2}{2\sigma^2}\right)$$

with mean μ and variance σ^2.

Warning: dfnGen uses the mean and standard deviation of the underlying normal distribution that creates the lognormal distribution not the mean and variance of the lognormal distribution.

In order to produce a LogNormal distribution with a desired mean (μ) and variance (σ^2) one uses

$$eLogMean = \ln\left(\frac{\mu^2}{\sqrt{\mu^2 + \sigma^2}}\right)$$

and

$$esd = \sqrt{\ln\left(1 + \frac{\sigma^2}{\mu^2}\right)}$$

For more details see https://en.wikipedia.org/wiki/Log-normal_distribution.

---

eLogMean

Warning: This value is not the mean of the Log Normal distribution. Use the equations above to convert between the values if needed.

Description: Mean value of the underlying normal distribution

Type: Array of doubles. Length is the number of elements in *edistr* set to 1.

Example:

```
eLogMean: {1.609} // This value along with that in esd produce a lognormal with mean 5 m␣
↪and variance of 0.1
```

---

esd

Warning: This value is not the standard deviation of the Log Normal distribution. Use the equations above to convert between the values if needed.

Description: Standard deviation value of the underlying normal distribution

Type: Array of Positive Doubles. Length is the number of elements in *edistr* set to 1.

Example:

```
esd: {0.040} // This value along with that in eLogMean produce a lognormal with mean 5 m␣
↪and variance of 0.1
```

eLogMin

Description: Minimum radius created by the LogNormal distribution for each family

Type: Array of Positive Doubles. Length is the number of elements in *edistr* set to 1.

Example:

```
eLogMin: {1,0.4}
// Lognormal family 1 has a minimum radius of 1 m
// Lognormal family 2 has a maximum radius of 0.4 m
```

Note: eLogMin must be less than *eLogMax* within each family.

eLogMax

Description: Maximum radius created by the LogNormal distribution for each family

Type: Array of Positive Doubles. Length is the number of elements in *edistr* set to 1.

Example:

```
eLogMax: {10,12}
// Lognormal family 1 has a maximum radius of 10 m
// Lognormal family 2 has a maximum radius of 12 m
```

Note: eLogMax must be greater than *eLogMin* within each family.

Ellipse: Truncated Powerlaw Distribution

Fracture radii are sampled from a truncated power-law distribution with lower bound r_0, upper bound r_u, and exponent α defined by the following probability density function

$$\frac{\alpha}{r_0}\frac{(r/r_0)^{-1-\alpha}}{1-(r_u/r_0)^{-\alpha}}.$$

ealpha

Description: Exponent of the truncated powerlaw distribution

Type: Array of Positive Doubles. Length is the number of elements in *edistr* set to 2.

Example:

```
ealpha: {1.6, 2.2}
// TPL family 1 has an alpha of 1.6
// TPL family 2 has an alpha of 2.2
```

Note: A value of 0 creates constant sized fractures of size *emin*

---

emin

Description: Lower cutoff of the truncated powerlaw distribution

Type: Array of Positive Doubles. Length is the number of elements in *edistr* set to 2.

Example:

```
emin: {1.2, 5}
// TPL family 1 has an lower cutoff of 1.2 m
// TPL family 2 has an lower cutoff of 5 m
```

Note: emin must be less than *emax* within each family.

---

emax

Description: Upper cutoff of the truncated powerlaw distribution

Type: Array of Positive Doubles. Length is the number of elements in *edistr* set to 2.

Example:

```
emax: {10, 50}
// TPL family 1 has an upper cutoff of 10 m
// TPL family 2 has an upper cutoff of 50 m
```

Note: emax must be greater than *emin* within each family.

Ellipse: Exponential Distribution

Fracture radii are sampled from a exponential distribution with the following probability density function

$$\lambda e^{-\lambda x}$$

Where λ is referred to as the rate parameter.

---

eExpMean

Description: Mean value of each exponential distribution

Type: Array of Positive Doubles. Length is the number of elements in *edistr* set to 3.

Example:

```
eExpMean: {10, 25}
// Exponential family 1 has a mean value of 10 m
// Exponential family 2 has a mean value of 25 m
```

Note: eExpMean equal to $1/\lambda$ where λ is the rate parameter of the distribution.

---

eExpMin

Description: Lower cutoff of the exponential distribution families

Type: Array of Positive Doubles. Length is the number of elements in *edistr* set to 3.

Example:

```
eExpMin: {1, 7}
// Exponential family 1 has a lower cutoff value of 1 m
// Exponential family 2 has a lower cutoff value of 7 m
```

Note: eExpMin must be less than *eExpMax* within each family.

---

eExpMax

Description: Upper cutoff of the exponential distribution families

Type: Array of Positive Doubles. Length is the number of elements in *edistr* set to 3.

Example:

```
eExpMax: {527, 89}
// Exponential family 1 has a upper cutoff value of 527 m
// Exponential family 2 has a upper cutoff value of 89 m
```

Note: eExpMax must be greater than *eExpMin* within each family.

Ellipse: Constant

Constant sized fracture families are defined using a single parameter *econst*. These families are also referred to as uniform or mono-disperse.

econst

Description: Constant radius for each family

Type: Array of Positive Doubles. Length is the number of elements in *edistr* set to 4.

Example:

```
econst: {1, 7}
// Constant family 1 has a x-radius of 1 m
// Constant family 2 has a x-radius of 7 m
```

11.5 Fracture Family Generation Parameters: Rectangle

This section describes generation parameters for families of rectangular fractures. The number of elements in each parameter must match number of rectangle families defined by *nFamRect*. The index of the elements corresponds to the fracture family. The first element of each parameter corresponds to the first family, the second element corresponds to the second family, etc.

11.5.1 Rectangle: General Parameters

nFamRect

Description: (Mandatory) Number of rectangle families

Type: Non-Negative Integer

```
nFamRect: 3 // There will be 3 rectangle families
```

Note: If this option is set to 0, then all rectangle parameters will be ignored.

rLayer

Description: (Mandatory) Assign each rectangle family to a layer in domain.

Type: Array of *nFamRect* integers

Example:

```
rLayer: {0,2,1}
// Family 1 is assigned to the whole domain
// Family 2 is assigned to layer 2
// Family 3 is assigned to layer 1
```

Note: Layer 0 is the entire domain. Numbers > 0 correspond to those defined in *layers*.

Warning: Families can only be assigned to either a layer or a region, not both.

rRegion

Description: (Mandatory) Assign each rectangle family to a region in domain.

Type: Array of *nFamRect* integers

Example:

```
rRegion: {0,2,1}
// Family 1 is assigned to the whole domain
// Family 2 is assigned to region 2
// Family 3 is assigned to region 1
```

Note: Region 0 is the entire domain. Numbers > 0 correspond to those defined in *regions*.

Warning: Families can only be assigned to either a layer or a region, not both.

r_p32Targets

Description: Target fracture intensity per family. Fracture intensity $P_{32}[\mathrm{m}^{-1}]$ is defined as total surface area of each fracture family divided by the total domain volume. Fractures from each family are inserted into the domain until provided target values are obtained. Generation stops once all desired fracture intensity are obtained.

Type: Array of *nFamRect* doubles

Example:

```
r_p32Targets: {0.02,0.4,0.05}
// Family 1 has a target p32 of 0.02
// Family 2 has a target p32 of 0.4
// Family 3 has a target p32 of 0.05
```

Note: Only used when *stopCondition* = 1

Warning: The fracture surface area is defined using *both* sides of a fracture.

raspect

Description: Aspect ratio of fractures

Type: Array of *nFamRect* doubles

Example:

```
raspect: {1,2,0.5}
// Family 1 has an aspect ratio of 1 (circles)
// Family 2 has an aspect ratio of 2 - y radius is twice the x radius
// Family 3 has an aspect ratio of 0.5 - y radius is 1/2 the x radius
```

Note:

1. A value of 1 makes squares

11.5.2 Rectangle: Fracture Orientation

The fracture orientations are sampled from the three-dimensional von Mises-Fisher distribution,

$$f(\mathbf{x}; \boldsymbol{\mu}, \kappa) = \frac{\kappa \exp(\kappa \boldsymbol{\mu}^T \mathbf{x})}{4\pi \sinh(\kappa)} .$$

where $\boldsymbol{\mu}$ is the mean orientation vector and T denotes transpose. The distribution is sampled using the algorithm provided Simulation of the von Mises Fisher distribution, Communications in statistics-simulation and computation 23.1 (1994): 157-164. by Andrew Wood..

dfnGen accepts spherical coordinates (θ/ϕ), Trend/Plunge, or Dip/Strike to define the mean orientation of a fracture family.

Tip: *orientationOption* selects which of these options are used. The same option must be used for all families.

0 : Spherical Coordinate
1 : Trend / Plunge

2 : Dip / Strike.

---

rAngleOption

Description: Selection of angle option for rectangle families

Type: boolean (0/1)

0 - radians (Must use numerical value for π)
1 - degrees

Note: Use of radians is only supported for spherical coordinates (*rtheta*/*rphi*). Degrees must be used for *rtrend*/*rplunge* and *rdip*/*rstrike*

---

rkappa

Description: The concentration parameter of the von Mises-Fisher distribution, which determines the degree of clustering around the mean orientation.

Type: Array of *nFamRect* doubles

Example:

```
rkappa: {0.1, 20, 17}
// Fracture Family 1 has a theta value of 45 degrees
// Fracture Family 2 has a theta value of 78 degrees
// Fracture Family 3 has a theta value of 0 degrees
```

Note: Values of κ approaching zero result in a uniform distribution of points on the sphere. Larger values create points with a small deviation from mean direction.

Warning: The numerical method for sampling the von Mises-Fisher distribution becomes unstable for values greater than 100.

Rectangle: Spherical Coordinates

The mean normal vector of the fracture family $\vec{n}$ is related to the spherical coordinates θ and ϕ by

$$\vec{n}_x = \sin(\theta)\cos(\phi)$$
$$\vec{n}_y = \sin(\theta)\sin(\phi) \quad (11.11)$$
$$\vec{n}_z = \cos(\theta) \quad (11.12)$$

---

rtheta

Description: Angle the normal vector of the fracture makes with the z-axis

Type: Array of *nFamRect* double

Example:

```
rtheta: {45, 78, 0}
// Fracture Family 1 has a theta value of 45 degrees
// Fracture Family 2 has a theta value of 78 degrees
// Fracture Family 3 has a theta value of 0 degrees
```

Note: Both radians and degrees are supported. Use *rAngleOption* to select one. If *rAngleOption* is set to radians *rAngleOption* : 0, and the value provided must be less than 2π.

---

rphi

Description: Angle that the projection of the normal vector of a fracture onto the x-y plane makes with the x-axis.

Type: Array of *nFamRect* double

Example:

```
rtheta: {0, 56, 12}
// Fracture Family 1 has a phi value of 0 degrees
// Fracture Family 2 has a phi value of 56 degrees
// Fracture Family 3 has a phi value of 12 degrees
```

Note: Both radians and degrees are supported. Use *rAngleOption* to select one. If *rAngleOption* is set to radians *rAngleOption* : 0, then the value provided must be less than 2π.

Rectangle: Trend & Plunge

The mean normal vector of the fracture family $\vec{n}$ is related to trend and plunge by

$$\vec{n}_x = \cos(\text{trend})\cos(\text{plunge})$$
$$\vec{n}_y = \sin(\text{trend})\cos(\text{plunge}) \tag{11.14}$$
$$\vec{n}_z = \sin(\text{trend}) \tag{11.15}$$

---

rtrend

Description: Trend of fracture families

Type: Array of *nFamRect* double

Example

```
rtrend: {0, 56, 12}
// Fracture Family 1 has a trend value of 0 degrees
// Fracture Family 2 has a trend value of 56 degrees
// Fracture Family 3 has a trend value of 12 degrees
```

Note: *nFamRect* must be set to degree (rAngleOption: 1) to use Trend & Plunge

---

rplunge

Description: Plunge of fracture families

Type: Array of *nFamRect* double

Example

```
eplunge: {0, 56, 12}
// Fracture Family 1 has a plunge value of 0 degrees
// Fracture Family 2 has a plunge value of 56 degrees
// Fracture Family 3 has a plunge value of 12 degrees
```

Note: *rAngleOption* must be set to degree (rAngleOption: 1) to use Trend & Plunge

---

Rectangle: Dip & Strike

The mean normal vector of the fracture family $\vec{n}$ is related to dip and strike by

$$\vec{n}_x = \sin(\text{dip})\sin(\text{strike})$$
$$\vec{n}_y = -\sin(\text{dip})\cos(\text{strike}) \quad (11.17)$$
$$\vec{n}_z = \cos(\text{dip}) \quad (11.18)$$

---

rstrike

Description: Strike of fracture families

Type: Array of *nFamRect* double

Example

```
rstrike: {0, 56, 12}
// Fracture Family 1 has a strike value of 0 degrees
// Fracture Family 2 has a strike value of 56 degrees
// Fracture Family 3 has a strike value of 12 degrees
```

Note: *rAngleOption* must be set to degree (rAngleOption: 1) to use Dip & Strike

---

rdip

Description: Dip of fracture families

Type: Array of *nFamRect* double

Example

```
rdip: {0, 56, 12}
// Fracture Family 1 has a dip value of 0 degrees
// Fracture Family 2 has a dip value of 56 degrees
// Fracture Family 3 has a dip value of 12 degrees
```

Note: *rAngleOption* must be set to degree (rAngleOption: 1) to use Dip & Strike

---

Rectangle: In Plane Rotation

rbetaDistribution

Description: Prescribe a rotation around each fracture's normal vector, with the polygon centered on x-y plane at the origin.

Type: Array of *nFamRect* boolean values

0: Uniform distribution on [0, 2π]
1: Constant rotation specified by *rbeta*

```
rbetaDistribution: {0, 1, 1}
// Fracture Family 1 will have a random rotation
// Fracture Family 2 will have a constant angle of rotation defined in the first entry␣
↪of rbeta
// Fracture Family 3 will have a constant angle of rotation defined in the second entry␣
↪of rbeta
```

---

rbeta

Description: Values for constant angle of rotation around the normal vector

Type: Array of boolean (0/1)

Example:

```
rbetaDistribution: {45, 270} // For rbetaDistribution: {0, 1, 1}
// Fracture Family 2 will have a constant angle of rotation of 45 degrees
// Fracture Family 3 will have a constant angle of rotation of 270 degrees
```

Note:

1. The length of rbeta corresponds to the number of *non-zero* entries in *rbetaDistribution*
2. *rAngleOption* defines if the values are in radians or degrees

---

11.5.3 Rectangle: Fracture Radius Distributions

Fracture radii can be defined using four different distributions: (1) Log-Normal, (2) Truncated Power-law, (3) Exponential, and (4) Constant. Minimum and maximum values must be provided for 1-3 along with the distribution parameters.

---

rdistr

Description: Assigns fracture radius distribution for each family

Type: Array of *nFamRect* Integers (1,2,3,4)

Example:

```
rdistr: {1, 2, 4}
// Fracture Family 1 will use a LogNormal Distribution
// Fracture Family 2 will use a Truncated powerlaw distribution
// Fracture Family 3 will have a constant sized fractures
```

Note: Number of elements in the parameters for each distribution must match number of families assigned to that distribution.

Rectangle: Lognormal Distribution

dfnGen uses the mean and standard deviation of the underlying normal distribution that creates the lognormal distribution.

These parameters define fracture radii sampled from a Lognormal distribution with the following probability density function

$$\frac{1}{x\sigma\sqrt{2\pi}} \exp\left(-\frac{(\ln x - \mu)^2}{2\sigma^2}\right)$$

In order to produce a LogNormal distribution with a desired mean (μ) and variance (σ^2) one uses

$$rLogMean = \ln\left(\frac{\mu^2}{\sqrt{\mu^2 + \sigma^2}}\right)$$

and

$$rsd = \sqrt{\ln\left(1 + \frac{\sigma^2}{\mu^2}\right)}$$

For more details see https://en.wikipedia.org/wiki/Log-normal_distribution.

rLogMean

Description: Mean value of the underlying normal distribution

Type: Array of doubles. Length is the number of elements in *rdistr* set to 1.

Example:

```
rLogMean: {1.609} // This value along with that in esd produce a lognormal with mean 5 m
↪and variance of 0.1
```

Warning: This value is not the mean of the Log Normal distribution. Use the equations above to convert between the values if needed.

---

rsd

Description: Standard deviation value of the underlying normal distribution

Type: Array of Positive Doubles. Length is the number of elements in *rdistr* set to 1.

Example:

```
rsd: {0.040} // This value along with that in rLogMean produce a lognormal with mean 5 m
↪and variance of 0.1
```

Warning: This value is not the standard deviation of the Log Normal distribution. Use the equations above to convert between the values if needed.

---

rLogMin

Description: Minimum radius created by the LogNormal distribution for each family

Type: Array of Positive Doubles. Length is the number of elements in *rdistr* set to 1.

Example:

```
rLogMin: {1,0.4}
// Lognormal family 1 has a minimum radius of 1 m
// Lognormal family 2 has a maximum radius of 0.4 m
```

Note: rLogMin must be less than *rLogMax* within each family.

---

rLogMax

Description: Maximum radius created by the LogNormal distribution for each family

Type: Array of Positive Doubles. Length is the number of elements in *rdistr* set to 1.

Example:

```
rLogMax: {10,12}
// Lognormal family 1 has a maximum radius of 10 m
// Lognormal family 2 has a maximum radius of 12 m
```

Note: rLogMax must be greater than *rLogMin* within each family.

Rectangle: Truncated Powerlaw Distribution

These parameters define fracture radii sampled from a truncated power-law distribution with lower bound r_0, upper bound r_u, and exponent α defined by the following probability density function

$$\frac{\alpha}{r_0}\frac{(r/r_0)^{-1-\alpha}}{1-(r_u/r_0)^{-\alpha}}.$$

---

ralpha

Description: Exponent of the truncated powerlaw distribution

Type: Array of Positive Doubles. Length is the number of elements in *rdistr* set to 2.

Example:

```
ralpha: {1.6, 2.2}
// TPL family 1 has an alpha of 1.6
// TPL family 2 has an alpha of 2.2
```

Note: A value of 0 creates constant sized fractures of size *rmin*

---

rmin

Description: Lower cutoff of the truncated powerlaw distribution

Type: Array of Positive Doubles. Length is the number of elements in *rdistr* set to 2.

Example:

```
rmin: {1.2, 5}
// TPL family 1 has an lower cutoff of 1.2 m
// TPL family 2 has an lower cutoff of 5 m
```

Note: rmin must be less than *rmax* within each family.

---

rmax

Description: Upper cutoff of the truncated powerlaw distribution

Type: Array of Positive Doubles. Length is the number of elements in *rdistr* set to 2.

Example:

```
rmax: {10, 50}
// TPL family 1 has an upper cutoff of 10 m
// TPL family 2 has an upper cutoff of 50 m
```

Note: rmax must be greater than *rmin* within each family.

Rectangle: Exponential Distribution

These parameters define fracture radii sampled from a exponential distribution with the following probability density function

$$\lambda e^{-\lambda x}$$

Where λ is referred to as the rate parameter.

rExpMean

Description: Mean value of each exponential distribution

Type: Array of Positive Doubles. Length is the number of elements in *rdistr* set to 3.

Example:

```
rExpMean: {10, 25}
// Exponential family 1 has a mean value of 10 m
// Exponential family 2 has a mean value of 25 m
```

Note: rExpMean equal to $1/\lambda$ where λ is the rate parameter of the distribution.

rExpMin

Description: Lower cutoff of the exponential distribution families

Type: Array of Positive Doubles. Length is the number of elements in *rdistr* set to 3.

Example:

```
rExpMin: {1, 7}
// Exponential family 1 has a lower cutoff value of 1 m
// Exponential family 2 has a lower cutoff value of 7 m
```

Note: rExpMin must be less than *rExpMax* within each family.

rExpMax

Description: Upper cutoff of the exponential distribution families

Type: Array of Positive Doubles. Length is the number of elements in *rdistr* set to 3.

Example:

```
rExpMax: {527, 89}
// Exponential family 1 has a upper cutoff value of 527 m
// Exponential family 2 has a upper cutoff value of 89 m
```

Note: rExpMax must be greater than *rExpMin* within each family.

Rectangle: Constant

Constant sized fracture families are defined using a single parameter *rconst*. These families are also referred to as uniform or mono-disperse.

---

rconst

Description: Constant radius for each family

Type: Array of Positive Doubles. Length is the number of elements in *rdistr* set to 4.

Example:

```
rconst: {1, 7}
// Constant family 1 has a x-radius of 1 m
// Constant family 2 has a x-radius of 7 m
```

11.6 User Defined Fracture Generation Parameters

User defined deterministic features can be included into dfnWorks in a number of ways. There are two format options for ellipses and rectangles. One can also put in a selection of convex *n* vertex polygons.

11.6.1 User Defined Ellipses

Ellipses can be included using two different formats: general or coordinates.

userEllipsesOnOff

Description: Selection if general user defined ellipses are going to be used. If this option is activated, then the file *UserEll_Input_File_Path* is read. The path to that file must be valid.

Type: boolean (0/1)

0: Do not include general user defined ellipses
1: Do include general user defined ellipses

UserEll_Input_File_Path

Description: Filepath for general user defined ellipses

Type: string

Example:

```
UserEll_Input_File_Path: /dfnWorks/examples/4_user_ell_uniform/define_4_user_ellipses.dat
```

General user defined ellipses parameters

Below are the required parameters for the general user defined ellipses

nUserEll

Description: Number of User Defined Ellipses

Type: Int

Example:

```
nUserEll: 2 // 2 ellipses are expected in the file
```

Number_of_Vertices

Description: Number of vertices defining the boundary of each ellipse. One per line per fracture.

Type: Int

Example:

```
Number_of_Vertices:
12 // fracture 1 has 12 vertices
8 // fracture 2 has 8 vertices
```

Radii

Description: Radius of each ellipse

Type: Double, one value per line per fracture

Example:

```
Radii:
0.5 // fracture 1 has a radius of 0.5 m
1 // fracture 1 has a radius of 1 m
```

Aspect_Ratio

Description: Aspect Ratio for each ellipse

Type: Double, one value per line per fracture

Example:

```
Aspect_Ratio:
1 // fracture 1 has a radius of 1
2 // fracture 2 has a radius of 2
```

AngleOption

Description: Selection of Radius / Degrees

Type: boolean

0: All angles in radians
1: All angles in degrees

Example:

```
AngleOption:
0 // fracture 1 has angles in radians
1 // fracture 2 has angles in degrees
```

Beta

Description: Rotation around center for each ellipse (one per line)

Type: Double, one value per line per fracture

Example:

```
AngleOption:
0 // no rotation for fracture 1
45 // rotate fracture 2 by 45 degrees
```

Translation

Description: Translation of each ellipse according to its center {x,y,z} (one per line)

Type: Set of double values {x,y,z}

Example:

```
Translation:
{-0.2,0,0} // Fracture 1 has a center at -0.2, 0, 0
{0,1,0} // Fracture 2 has a center at 0, 1, 0
```

---

userOrientationOption

Description: Selection of fracture orientation definition. The same orientation option must be used for all fractures.

0 : Normal Vector -> Normal {x,y,z}
1 : Trend / Plunge
2 : Dip / Strike

Type: Value of 0,1,2

Example:

```
userOrientationOption: 0 // Fracture orientation will be defined using normal vectors
```

Note:

If option 0 is selected, the keyword normal is required.
If option 1 is selected, the keyword Trend_Plunge is required.
If option 2 is selected, the keyword Dip_Strike is required.

Warning: You cannot mix orientation options

Normal

Description: Normal vector of each ellipse according {x,y,z} (one per line)

Type: Set of double values {x,y,z}

Example:

```
Normal:
{0,0,1} // Fracture 1 has a normal vector of [0,0,1]
{1,1,0} // Fracture 1 has a normal vector of [1,1,0]
```

Note: Vectors *do not* need to be normalized

Trend_Plunge

Description: Trend and plunge for each fracture

Type: Set of double values {trend,plunge}

Example:

```
Trend_Plunge:
{45, 90} // Fracture 1 has a trend of 45 and plunge of 90
{0, 37} // Fracture 2 has a trend of 0 and plunge of 37
```

Warning: *AngleOption* must be set to degrees

Dip_Strike

Description: Dip and Strike for each fracture

Type: Set of double values {dip,strike}

Example:

```
Trend_Plunge:
{45, 90} // Fracture 1 has a dip of 45 and strike of 90
{0, 37} // Fracture 2 has a dip of 0 and strike of 37
```

Warning: *AngleOption* must be set to degrees

General Ellipse Input Example

```
/************************ USER DEFINED ELLIPSES ***************************/
/**************************************************************************/

/**************************************************************************/
//Number of User Defined Ellipses
/**************************************************************************/
nUserEll: 4


/**************************************************************************/
//Radius for each ellipse (one per line)
/**************************************************************************/
Radii:
0.5
0.5
0.4
0.4


/**************************************************************************/
//Aspect Ratio for each ellipse (one per line)
/**************************************************************************/
Aspect_Ratio:
1
1
1
1


/**************************************************************************/
//Angle Option: 0 - All angles in radians
//              1 - All angles in degrees
/**************************************************************************/
AngleOption:
1
1
1
1

/**************************************************************************/
//Rotation around center for each ellipse (one per line)
/**************************************************************************/
Beta:
0
0
0
0

/**************************************************************************/
//Translation of each ellipse according to its center {x,y,z} (one per line)
```

(continues on next page)

(continued from previous page)

```
/**************************************************************************/
Translation:
{-0.2,0,0}
{0,0,0}
{0.2,0,0.2}
{0.2,0,-0.2}


/**************************************************************************/
//  userOrientationOption:
//  0 - Normal Vector -> Normal {x,y,z}
//  1 - Trend / Plunge -> Trend_Plunge {trend, plunge} -> Must be degrees
//  2 - Dip / Strike - > Dip_Strike {dip, strike} -> Must be degrees
/**************************************************************************/

userOrientationOption: 0


/**************************************************************************/
//Normal Vector for each ellipse (one per line)
/**************************************************************************/
Normal:
{0,0,1}
{1,0,0}
{0,0,1}
{0,0,1}


Number_of_Vertices:
8
8
8
8
```

userEllByCoord

Description: Selection if user defined ellipses by coordinate are going to be used. If this option is activated, then the file *EllByCoord_Input_File_Path* is read. The path to that file must be valid.

Type: boolean (0/1)

0: Do not include user defined ellipses by coordinate
1: Include user defined ellipses by coordinate

Warning: The same number of vertices must be used for all fractures.

EllByCoord_Input_File_Path

Description: File path name for user defined ellipses by coordinate

Type: string

Example:

```
EllByCoord_Input_File_Path: /dfnWorks/example/4_user_ell/ellCoords.dat
```

User defined ellipses by coordinate parameters

Below are the required parameters for the user defined ellipses by coordinate

---

nEllipses

Description: Number of User Defined Ellipses

Type: Integer

Example:

```
nEllipses: 2
```

---

nNodes

Description: Number of nodes for all ellipses

Type: Integer

```
nNodes: 5
```

Coordinates

Description: Coordinates / Vertices for each ellipse.

Type: Set of *nNodes* triples $x_0, y_0, z_0 x_1, y_1, z_1 \ldots x_n, y_n, z_n$

```
Coordinates:
{-2,-1,0} {1,-2,0} {2,0,0} {0,2,0} {-2,1,0}
{0,-0.3,-1} {0,.5,-.7} {0,.7,1} {0,-.7,1} {0,-1,0}
```

Warning: Coordinates must be listed in clockwise, or counterclockwise order. Coordinates must be co-planar

---

Ellipse By Coordinate Example

```
/*************************************************************************/
/*                  ELLIPSES SPECIFIED BY COORDINATES                    */
/*************************************************************************/
// NOTE: Coordinates must be listed in clockwise, or counterclockwise order
//       Coordinates must be co-planar

/*************************************************************************/
// Number of Ellipses Defined
/*************************************************************************/

nEllipses: 2


/*************************************************************************/
// Number of nodes for all ellipses
/*************************************************************************/

nNodes: 5


/*************************************************************************/
// Coordinates (4 vertice coordinates per line/One rectangle per line)
/*************************************************************************/
// One Ellipse per line (White space and new lines should not matter)
// Format: {x1,y1,z1} {x2,y2,z2} {x3,y3,z3} {x4,y4,z4} ... {xn, yn, zn}

Coordinates:

{-2,-1,0} {1,-2,0} {2,0,0} {0,2,0} {-2,1,0}
{0,-0.3,-1} {0,.5,-.7} {0,.7,1} {0,-.7,1} {0,-1,0}
```

11.6.2 User Defined Rectangles

Deterministic Rectangles can be included using two different formats: general or coordinates.

userRectanglesOnOff

Description: Selection if general user defined rectangles are going to be used. If this option is activated, then the file *UserRect_Input_File_Path* is read. The path to that file must be valid.

Type: boolean (0/1)

0: Do not include general user defined rectangles
1: Do include general user defined rectangles

UserRect_Input_File_Path

Description: Filepath for general user defined rectangles

Type: string

Example:

```
UserRect_Input_File_Path: /dfnWorks/examples/4_user_rects/define_4_user_rects.dat
```

General user defined rectangles parameters

Below are the required parameters for the general user defined rectangles

---

nUserRect

Description: Number of user defined rectangles

Type: Int

Example:

```
nUserRect: 2 // There will be 2 rectangles expected.
```

Additional parameters have the same definitions as for user defined ellipses:

- *Radii*
- *Aspect_Ratio*
- *AngleOption*
- *Beta*
- *Translation*
- *userOrientationOption*
- *Normal*
- *Trend_Plunge*
- *Dip_Strike*

General Rectangle Input Example

```
/************************ USER DEFINED rectangles ***************************/
/****************************************************************************/

/****************************************************************************/
//Number of User Defined rectangles
/****************************************************************************/
nUserRect: 4


/****************************************************************************/
//Radius for each rectangle (one per line)
/****************************************************************************/
Radii:
0.5
0.5
0.4
0.4


/****************************************************************************/
//Aspect Ratio for each rectangle (one per line)
/****************************************************************************/
Aspect_Ratio:
1
1
1
1


/****************************************************************************/
//Angle Option: 0 - All angles in radians
//              1 - All angles in degrees
/****************************************************************************/
AngleOption:
1
1
1
1

/****************************************************************************/
//Rotation around center for each rectangle (one per line)
/****************************************************************************/
Beta:
0
0
0
0

/****************************************************************************/
//Translation of each rectangle according to its center {x,y,z} (one per line)
```

(continues on next page)

(continued from previous page)

```
/**************************************************************************/
Translation:
{-0.2,0,0}
{0,0,0}
{0.2,0,0.2}
{0.2,0,-0.2}


/**************************************************************************/
//  userOrientationOption:
//  0 - Normal Vector -> Normal {x,y,z}
//  1 - Trend / Plunge -> Trend_Plunge {trend, plunge} -> Must be degrees
//  2 - Dip / Strike - > Dip_Strike {dip, strike} -> Must be degrees
/**************************************************************************/

userOrientationOption: 0


/**************************************************************************/
//Normal Vector for each rectangle (one per line)
/**************************************************************************/
Normal:
{0,0,1}
{1,0,0}
{0,0,1}
{0,0,1}
```

userRecByCoord

Description: Selection if user defined rectangles by coordinate are going to be used. If this option is activated, then the file *RectByCoord_Input_File_Path* is read. The path to that file must be valid.

Type: boolean (0/1)

0: Do not include user defined rectangles by coordinate
1: include user defined rectangles by coordinate

Warning: The same number of vertices must be used for all fractures.

RectByCoord_Input_File_Path

Description: File path name for user defined rectangles by coordinate

Type: string

Example:

```
RectByCoord_Input_File_Path: /dfnWorks/example/4_user_rect/rectCoords.dat
```

User defined rectangles by coordinate parameters

Below are the required parameters for the user defined rectangles by coordinate

---

nRectangles

Description: Number of user defined rectangles

Type: Integer

Example:

```
nRectangles: 2
```

Fracture coordinates are defined using the same method as for ellipses. See *Coordinates*.

Rectangle By Coordinate Example

```
/*****************************************************************************/
/*                  rectangles SPECIFIED BY COORDINATES                      */
/*****************************************************************************/
// NOTE: Coordinates must be listed in clockwise, or counterclockwise order
//       Coordinates must be co-planar

/*****************************************************************************/
// Number of rectangles Defined
/*****************************************************************************/

nRectangles: 2

/*****************************************************************************/
// Coordinates (4 vertices coordinates per line/One rectangle per line)
/*****************************************************************************/
// One rectangle per line (White space and new lines should not matter)
// Format: {x1,y1,z1} {x2,y2,z2} {x3,y3,z3} {x4,y4,z4} ... {xn, yn, zn}

Coordinates:
```

(continues on next page)

(continued from previous page)

```
{-2,-1,0} {1,-2,0} {2,0,0} {0,2,0} {-2,1,0}
{0,-0.3,-1} {0,.5,-.7} {0,.7,1} {0,-.7,1} {0,-1,0}
```

11.6.3 Polygons

An example DFN for this input is found in dfnWorks-main/examples/user_polygons.

---

userPolygonByCoord

Description: Selection if user defined polygons are going to be used. If this option is activated, then the file :refPolygonByCoord_Input_File_Path is read. The path to that file must be valid.

Type: boolean (0/1)

0: Do not include user defined polygons
1: Do include user defined polygons

Note:

Each polygon can have a different number of vertices.
dfnGen automatically outputs the file *polygons.dat* which can be read back in using this option to create the same DFN.

PolygonByCoord_Input_File_Path:

Description: File path for user defined polygons

Type: string

Example:

```
UserEll_Input_File_Path: /dfnWorks/examples/user_polygons/polygons.dat
```

Polygon file format

The first line is a keyword *nPolygons*, the number of fractures in the file. Then each line after nPolygons is a different fracture. The first entry is an integer with the number of vertices in that fracture. The remaining entries are the vertex coordinates. The coordinate format is {x1,y1,z1} {x2,y2,z2} {x3,y3,z3} {x4,y4,z4} ... {xn, yn, zn}

General Example:

```
nPolygons: 1
4 {x1,y1,z1} {x2,y2,z2} {x3,y3,z3} {x4,y4,z4} // fracture 1 has 4 vertices with these␣
↪coordinates
```

Example generated by dfnGen:

```
nPolygons: 13
5 {0.614809755508, -5, 10} {0.215302589745, -5, 5.14052566974} {0.545132019741, -3.
↪76296408041, 5.01135305297} {1.46205561432, -1.11169723114, 7.28911258873} {1.
↪7422671348, -0.903335314403, 10}
5 {-5, 5, -7.34275683413} {-5, -3.19676496723, -8.52840001844} {-2.54841054822, -3.
↪3215018603, -8.70265907788} {2.28455866847, 0.968743065242, -8.39004354883} {2.
↪5300049043, 5, -7.82257143457}
7 {0.169537908304, -1.10365780918, 7.89449659536} {3.39229757501, -1.47097810869, 9.
↪7368040684} {5, -0.396173810104, 9.9199696155} {5, 5, 6.76351044547} {0.192549006049,␣
↪5, 4.33581920939} {-0.157462422292, 4.76600532497, 4.29594241204} {-1.30082072643, 1.
↪47978607543, 5.64081751301}
6 {-0.793278268371, 5, 1.81778296407} {-0.561983329033, 3.45819042638, 3.44636375172} {1.
↪4541413343, -3.51825209439, 3.66719739398} {2.0912558548, -5, 2.1739128129} {3.
↪74246641127, -5, -10} {0.809628560699, 5, -10}
8 {-3.56729801834, -4.10664963232, -7.17895163585} {-2.75834169562, -5, -7.23742727251}
↪{2.80010131269, -5, -7.26666449977} {3.40228077595, -4.45665913893, -7.23685468994} {3.
↪54844410108, -1.5749811435, -7.06272440502} {1.61045433532, 0.565188057474, -6.
↪92263630123} {-1.27643915621, 0.710166484812, -6.89865208954} {-3.42113429321, -1.
↪22497133918, -7.00482133497}
6 {1.61970921408, 0.322281059007, 8.27249031626} {1.91468659533, -0.549550927249, 10} {2.
↪19700291895, -5, 10} {1.3154355331, -5, 3.64584005023} {1.20544838483, -3.98437001886,␣
↪3.31745142962} {1.22589362057, -0.804792681255, 4.91861492809}
8 {-4.70593999999, -3.29639431251, -7.25224389054} {-3.10094407568, -4.6562865987, -7.
↪03245087974} {-0.992518232701, -4.50612429324, -7.10741842947} {0.384249987443, -2.
↪93387033715, -7.43323157864} {0.222868762162, -0.860529584164, -7.81903344627} {-1.
↪38212753352, 0.499362646346, -8.03882643837} {-3.4905528176, 0.349200223043, -7.
↪9638588789} {-4.8673209588, -1.22305329999, -7.63804581282}
6 {5, 5, -3.39564264412} {2.56312416852, 5, -4.08462835745} {2.41898893677, 4.
↪65124788861, -4.11909954775} {3.10520773081, 2.8645938936, -3.89290711889} {4.
↪80872562941, 2.11160312936, -3.39770554375} {5, 2.19173173493, -3.34506895229}
6 {-5, 2.42895858094, 6.20485792137} {-5, -5, 4.85177752427} {-1.81351205493, -5, 4.
↪99702019362} {0.0853427250495, -2.66597219167, 5.50868187563} {-0.289753327606, 0.
↪588931258258, 6.08441961571} {-2.88990150472, 2.64805531643, 6.34094328974}
8 {-0.922507014307, -1.71519798881, 2.85778630995} {-0.841071899053, 0.310044679731, 1.
↪25820816606} {-1.26426496474, 2.84566856205, 1.49988903618} {-1.94418542615, 4.
↪40633923745, 3.44125563949} {-2.48254521057, 4.07783735953, 5.94508200102} {-2.
↪5639802392, 2.05259419185, 7.54466008318} {-2.14078724661, -0.483028953749, 7.
↪30297907019} {-1.4608669326, -2.04369954706, 5.36161304612}
8 {2.18143095055, -0.610312831185, -4.21771261931} {2.80909183961, 0.814183043594, -4.
↪47247873318} {2.26469088948, 2.29432736239, -4.44327042469} {0.867130728085, 2.
↪96307141873, -4.14719749191} {-0.564917036364, 2.42867426252, -3.75769541671} {-1.
↪19257780773, 1.00417817281, -3.50292931657} {-0.648176891617, -0.475965761662, -3.
↪53213764385} {0.74938298608, -1.14470985076, -3.82821050463}
```

(continues on next page)

(continued from previous page)

```
6 {-1.27165416308, 5, 10} {-1.97370232204, 1.13046740212, 10} {-2.01807341608, 0.
↪962283071202, 9.35725325459} {-1.83562647916, 2.23286105726, 7.12746635144} {-1.
↪40616097581, 4.67958113114, 6.45759785589} {-1.34398137987, 5, 6.64526675756}
```

nPolygons

Description: Number of polgyons/fractures

Type: integer

Example:

```
nPolygons: 3 // Read in three polygons/fractures
```

11.7 Hydrological Properties

Warning: These parameters are not going to be supported in future versions. Use the pydfnWorks function dump_hydraulic_values to specific hydraulic properties of fractures (hydraulic aperture, permeability, and transmissivity) after generation is complete. Using that function, values can be assigned by fracture family, which can correlated with depth using layers or regions.

11.7.1 Hydraulic Aperture

Description: Specification of hydraulic aperture option.

1) Log Normal distribution with parameters *meanAperture* and *stdAperture*.

2) Aperture from Transmissivity defined using *apertureFromTransmissivity*

3) Constant Aperture defined using *constantAperture*

4) Aperture perfectly correlated to fracture length using *lengthCorrelatedAperture*

Type: Integer 1,2,3,4

Note: Detailed description of each option is provided below.

Option 1) Log Normal Distribution

Aperture values are uncorrelated with fracture size and sampled from a log normal distribution with assigned mean and standard deviation. This model is often referred to as the *uncorrelated* model.

dfnGen uses the mean and standard deviation of the underlying normal distribution that creates the lognormal distribution.

These parameters define the aperture sampled from a Lognormal distribution with the following probability density function

$$\frac{1}{x\sigma\sqrt{2\pi}} \exp\left(-\frac{(\ln x - \mu)^2}{2\sigma^2}\right)$$

In order to produce a LogNormal distribution with a desired mean (μ) and variance (σ^2) one uses

$$meanAperture = \ln\left(\frac{\mu^2}{\sqrt{\mu^2 + \sigma^2}}\right)$$

and

$$stdAperture = \sqrt{\ln\left(1 + \frac{\sigma^2}{\mu^2}\right)}$$

For more details see https://en.wikipedia.org/wiki/Log-normal_distribution.

---

meanAperture

Description: Mean value of the underlying normal distribution

Type: Double

> **Warning:** This value is not the mean of the Log Normal distribution. Use the equations above to convert between the values if needed.

---

stdAperture

Description: Standard deviation of the underlying normal distribution

Type: Double

> **Warning:** This value is not the mean of the Log Normal distribution. Use the equations above to convert between the values if needed.

Option 2: Aperture from Transmissivity

Description: Determines the hydraulic aperture based on an assigned fracture transmissivity. First, the transmissivity is defined using

$$T = F \cdot r^k$$

where F and k are user defined parameters assigned here and r is the mean fracture radius. Once T is defined, then the aperture is determined using the cubic law

$$b = \sqrt[3]{12T}$$

---

apertureFromTransmissivity

Description: Parameters F and k to define transmissivity from fracture radius.

Type: Tuple of double values.

Example:

```
apertureFromTransmissivity: {1.6e-9, 0.8} // F = 1.6-9 and k = 0.8
```

Option 3: Constant Aperture

All fractures are assigned the same aperture value regardless of family, location, or size.

constantAperture

Description: Value of fracture aperture in meters

Type: Double

```
constantAperture: 1e-5
```

Option 3: Aperture perfectly correlated to fracture radius

Fracture aperture is assigned as a function of fracture size using the following equation

$$b = F \cdot r^k$$

where F and k are user defined parameters assigned here and r is the mean fracture radius.

---

lengthCorrelatedAperture

Description: Parameters F and k to define the relationship between fracture size and aperture.

Type: Tuple of double values.

Example:

```
lengthCorrelatedAperture: {5e-5, 0.5} // F = 5e-5 and k =  0.5
```

11.7.2 Permeability

Permeability can be defined in two ways. Either derived from the fracture aperture using the cubic law or assigned a constant value.

---

permOption

Description: Selection of permeability option

Type: Integer value of 0 or 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 Permeability for all fractures defined using *constantPermeability*

---

constantPermeability

Danger: Using this option will result in inconsistent permeability values of aperture with mesh volume rescaling for FEHM and PFLOTRAN.

Description: Value of constant permeability for all fractures. Value is in m^2

Type: Double

Example:

```
constantPermeability: 1e-12 // all fractures will have a permeability of 1e-12 m^2
```

11.8 Source Code Documentation (Doxygen)

CHAPTER

TWELVE

DFNFLOW

dfnFlow involves using flow solver such as PFLOTRAN or FEHM. PFLOTRAN is recommended if a large number of fractures (> O(1000)) are involved in a network. Using the function calls that are part of pydfnworks, 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
```

(continues on next page)

(continued from previous page)

```
#========================== 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 ==========================================
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
  FLOW_CONDITION outflow
  REGION outflow
END

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

END_SUBSURFACE
```

CHAPTER

THIRTEEN

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.

13.1 Documentation

Doxygen

CHAPTER

FOURTEEN

OUTPUT FILES

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

14.1 dfnGen

aperture.dat:

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.

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.

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.

intersection_list.dat:

List of intersections between fractures. Format is fracture1 fracture2 x y z length. Negative numbers correspond to intersections with boundaries.

14.2 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.lg:

Full DFN mesh in LaGriT binary format.

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:

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.

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.

14.3 PFLOTRAN

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 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.

14.4 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. Normal vector (0,1,0) +- pi/2

pboundary_bottom.zone:

Boundary zone file for the bottom of the domain. Normal vector (0,0,-1) +- pi/2

pboundary_front_s.zone:

Boundary zone file for the front of the domain. Normal vector (0,-1,0) +- pi/2

pboundary_left_w.zone:

Boundary zone file for the left side of the domain. Normal vector (-1,0,0) +- pi/2

pboundary_right_e.zone:

Boundary zone file for the bottom of the domain. Normal vector (1,0,0) +- pi/2

pboundary_top.zone:

Boundary zone file for the top of the domain. Normal vector (0,0,1) +- pi/2

CHAPTER

FIFTEEN

DFNWORKS PUBLICATIONS

The following are publications that use *dfnWorks*:

1. J. D. Hyman, C. W. Gable, S. L. Painter, and N. Makedonska. Conforming Delaunay triangulation of stochastically generated three dimensional discrete fracture networks: A feature rejection algorithm for meshing strategy. SIAM J. Sci. Comput. (2014).
2. R.S. Middleton, J.W. Carey, R.P. Currier, J. D. Hyman, Q. Kang, S. Karra, J. Jimenez-Martınez, M.L. Porter, and H.S. Viswanathan. Shale gas and non-aqueous fracturing fluids: Opportunities and challenges for supercritical CO2. Applied Energy, (2015).
3. J. D. Hyman, S. L. Painter, H. Viswanathan, N. Makedonska, and S. Karra. Influence of injection mode on transport properties in kilometer-scale three-dimensional discrete fracture networks. Water Resources Research (2015).
4. S. Karra, Nataliia Makedonska, Hari S Viswanathan, Scott L Painter, and Jeffrey D. Hyman. Effect of advective flow in fractures and matrix diffusion on natural gas production. Water Resources Research (2015).
5. J. D. Hyman, S. Karra, N. Makedonska, C. W Gable, S. L Painter, and H. S Viswanathan. dfnWorks: A discrete fracture network framework for modeling subsurface flow and transport. Computers & Geosciences (2015).
6. H. S. Viswanathan, J. D. Hyman, S. Karra, J.W. Carey, M. L. Porter, E. Rougier, R. P. Currier,Q. Kang, L. Zhou, J. Jimenez-Martınez, N. Makedonska, L. Chen, and R. S. Middleton. Using Discovery Science To Increase Efficiency of Hydraulic Fracturing While Reducing Water Usage, chapter 4, pages 71–88. ACS Publications, (2016).
7. N. Makedonska, S. L Painter, Q. M Bui, C. W Gable, and S. Karra. Particle tracking approach for transport in three-dimensional discrete fracture networks. Computational Geosciences (2015).
8. D. O'Malley, S. Karra, R. P. Currier, N. Makedonska, J. D. Hyman, and H. S. Viswanathan. Where does water go during hydraulic fracturing? Groundwater (2016).
9. J. D. Hyman, J Jiménez-Martínez, HS Viswanathan, JW Carey, ML Porter, E Rougier, S Karra, Q Kang, L Frash, L Chen, et al. Understanding hydraulic fracturing: a multi-scale problem. Phil. Trans. R. Soc. A, (2016).
10. G. Aldrich, J. D. Hyman, S. Karra, C. W. Gable, N. Makedonska, H. Viswanathan, J.Woodring, and B. Hamann. Analysis and visualization of discrete fracture networks using a flow topology graph. IEEE Transactions on Visualization and Computer Graphics (2017).
11. N. Makedonska, J. D. Hyman, S. Karra, S. L. Painter, C.W. Gable, and H. S. Viswanathan. Evaluating the effect of internal aperture variability on transport in kilometer scale discrete fracture networks. Advances in Water Resources (2016).
12. J. D. Hyman, G. Aldrich, H. Viswanathan, N. Makedonska, and S. Karra. Fracture size and transmissivity correlations: Implications for transport simulations in sparse three-dimensional discrete fracture networks following a truncated power law distribution of fracture size. Water Resources Research (2016).

13. H. Djidjev, D. O'Malley, H. Viswanathan, J. D. Hyman, S. Karra, and G. Srinivasan. Learning on graphs for predictions of fracture propagation, flow and transport. In 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) (2017).

14. J. D. Hyman, A. Hagberg, G. Srinivasan, J. Mohd-Yusof, and H. Viswanathan. Predictions of first passage times in sparse discrete fracture networks using graph-based reductions. Phys. Rev. E, 96:013304, Jul.

15. T Hadgu, S. Karra, N. Makedonska, J. D. Hyman, K. Klise, H. S. Viswanathan, and Y.Wang. A comparative study of discrete fracture network and equivalent continuum models for simulating flow and transport in the far field of a hypothetical nuclear waste repository in crystalline host rock. J. Hydrology, 2017.

16. V. Romano, J. D. Hyman, S. Karra, A. J. Valocchi, M. Battaglia, and S. Bigi. Numerical modeling of fluid flow in a fault zone: a case of study from majella mountain (Italy). Energy Procedia, 125:556 – 560, 2017.

17. M. Valera, Z. Guo, P. Kelly, S. Matz, A. Cantu, A.G. Percus, J. D. Hyman, G. Srinivasan, and H.S. Viswanathan. Machine learning for graph-based representations of three-dimensional discrete fracture networks. Computational Geosciences, (2018).

18. M. K. Mudunuru, S. Karra, N. Makedonska, and T. Chen. Sequential geophysical and flow inversion to characterize fracture networks in subsurface systems. Statistical Analysis and Data Mining: The ASA Data Science Journal (2017).

19. J. D. Hyman, Satish Karra, J. William Carey, Carl W. Gable, Hari Viswanathan, Esteban Rougier, and Zhou Lei. Discontinuities in effective permeability due to fracture percolation. Mechanics of Materials (2018).

20. S. Karra, D. O'Malley, J. D. Hyman, H.S. Viswanathan, and G. Srinivasan. Modeling flow and transport in fracture networks using graphs. Phys. Rev. E, (2018).

21. J. D. Hyman and J. Jimenéz-Martínez. Dispersion and mixing in three-dimensional discrete fracture networks: Nonlinear interplay between structural and hydraulic heterogeneity. Water Resources Research (2018).

22. D. O'Malley, S. Karra, J. D. Hyman, H. Viswanathan, and G. Srinivasan. Efficient Monte Carlo with graph-based subsurface flow and transport models. Water Resour. Res., (2018).

23. G. Srinivasan, J. D. Hyman, D. Osthus, B. Moore, D. O'Malley, S. Karra, E Rougier, A. Hagberg, A. Hunter, and H. S. Viswanathan. Quantifying topological uncertainty in fractured systems using graph theory and machine learning. Scientific Reports, (2018).

24. H. S. Viswanathan, J. D. Hyman, S. Karra, D. O'Malley, S. Srinivasan, A. Hagberg, and G. Srinivasan. Advancing graph-based algorithms for predicting flow and transport in fractured rock. Water Resour. Res., (2018).

25. S. Srinivasan, J. D. Hyman, S. Karra, D. O'Malley, H. Viswanathan, and G. Srinivasan. Robust system size reduction of discrete fracture networks: A multi-fidelity method that preserves transport characteristics. Computational Geosciences, 2018.

26. J. D. Hyman, Aric Hagberg, Dave Osthus, Shriram Srinivasan, Hari Viswanathan, and Gowri Srinivasan. Identifying backbones in three-dimensional discrete fracture net- works: A bipartite graph-based approach. Multiscale Modeling & Simulation (2018).

27. G. Aldrich, J. Lukasczyk, J. D. Hyman, G. Srinivasan, H. Viswanathan, C. Garth, H. Leitte, J. Ahrens, and B. Hamann. A query-based framework for searching, sorting, and exploring data ensembles. Computing in Science Engineering, (2018).

28. T. Sherman, J. D. Hyman, D. Bolster, N. Makedonska, and G. Srinivasan. Characterizing the impact of particle behavior at fracture intersections in three-dimensional discrete fracture networks. Physical Review E (2019).

29. J. D. Hyman, M. Dentz, A. Hagberg, and P. Kang. Linking structural and transport properties in three-dimensional fracture networks. J. Geophys. Res. Sol. Ea., (2019).

30. S. Srinivasan, S. Karra, J. D. Hyman, H. Viswanathan, and G. Srinivasan. Model reduction for fractured porous media: A machine-learning approach for identifying main flow pathways. Computational Geosciences (2018).

31. N. Makedonska, J.D, Hyman, E. Kwicklis, K. Birdsell, Conference Proceedings, Discrete Fracture Network Modeling and Simulation of Subsurface Transport for the Topopah Spring Aquifer at Pahute Mesa, 2nd International Discrete Fracture Network Engineering (2018).
32. N. Makedonska, C.W. Gable, R. Pawar, Conference Proceedings, Merging Discrete Fracture Network Meshes With 3D Continuum Meshes of Rock Matrix: A Novel Approach, 2nd International Discrete Fracture Network Engineering (2018).
33. A. Frampton, J.D, Hyman, L. Zou, Advective transport in discrete fracture networks with connected and disconnected textures representing internal aperture variability, Water Resources Research (2019).
34. J.D. Hyman, J. Jiménez-Martínez, C. W. Gable, P. H. Stauffer, and R. J. Pawar. Characterizing the Impact of Fractured Caprock Heterogeneity on Supercritical CO2 Injection. Transport in Porous Media (2019).
35. J.D. Hyman, H. Rajaram, S. Srinivasan, N. Makedonska, S. Karra, H. Viswanathan, H., & G. Srinivasan, (2019). Matrix diffusion in fractured media: New insights into power law scaling of breakthrough curves. Geophysical Research Letters (2019).
36. J.D. Hyman, M. Dentz, A. Hagberg, & P. K. Kang, (2019). Emergence of Stable Laws for First Passage Times in Three-Dimensional Random Fracture Networks. Physical Review Letters (2019).
37. M. R. Sweeney, C. W. Gable, S. Karra, P. H. Stauffer, R. J. Pawar, J. D. Hyman (2019). Upscaled discrete fracture matrix model (UDFM): an octree-refined continuum representation of fractured porous mediaComputational Geosciences (2019).
38. T. Sherman, J. D. Hyman, M. Dentz, and D. Bolster. Characterizing the influence of fracture density on network scale transport. J. Geophys. Res. Sol. Ea., (2019).
39. D. Osthus, J. D. Hyman, S. Karra, N. Panda, and G. Srinivasan. A probabilistic clustering approach for identifying primary subnetworks of discrete fracture networks with quantified uncertainty. SIAM/ASA Journal on Uncertainty Quantification, (2020).
40. V. Romano, S. Bigi, F. Carnevale, J. D. Hyman, S. Karra, A. Valocchi, M. Tartarello, and M. Battaglia. Hydraulic characterization of a fault zone from fracture distribution. Journal of Structural Geology, (2020).
41. S. Srinivasan, E. Cawi, J. D. Hyman, D. Osthus, A. Hagberg, H. Viswanathan, and G. Srinivasan. Physics-informed machine-learning for backbone identification in discrete fracture networks. Comput. Geosci., (2020).
42. N. Makedonska, S. Karra, H.S. Viswanathan, and G.D. Guthrie,. Role of Interaction between Hydraulic and Natural Fractures on Production. Journal of Natural Gas Science and Engineering (2020)..
43. H. Pham, R. Parashar, N. Sund, and K. Pohlmann. A Method to Represent a Well in a Three-dimensional Discrete Fracture Network Model. Groundwater. (2020).
44. M.R. Sweeney, and J.D. Hyman. Stress effects on flow and transport in three-dimensional fracture networks. Journal of Geophysical Research: Solid Earth. (2020).
45. J.D. Hyman. Flow Channeling in Fracture Networks: Characterizing the Effect of Density on Preferential Flow Path Formation. Water Resources Research (2020): e2020WR027986..
46. H. Pham, R. Parashar, N. Sund, and K. Pohlmann. Determination of fracture apertures via calibration of three-dimensional discrete-fracture-network models: application to Pahute Mesa, Nevada National Security Site, USA. Hydrogeol J (2020)..
47. S. Srinivasan, D. O'Malley, J. D. Hyman, s. Karra, H. S. Viswanathan, and G. Srinivasan Transient flow modeling in fractured media using graphs. (2020) Physical Review E..
48. Liangchao Zou and Vladimir Cvetkovic. Inference of Transmissivity in Crystalline Rock Using Flow Logs Under Steady-State Pumping: Impact of Multiscale Heterogeneity. Water Resources Research (2020).
49. P. K. Kang, J. D. Hyman, W. S. Han, & M. Dentz, Anomalous Transport in Three-Dimensional Discrete Fracture Networks: Interplay between Aperture Heterogeneity and Injection Modes. Water Resources Research (2020).

50. Hyman, J. D., & Dentz, M. Transport upscaling under flow heterogeneity and matrix-diffusion in three-dimensional discrete fracture networks. Advances in Water Resources (2021).

51. T. Sherman, G. Sole-Mari, J. Hyman, M. R. Sweeney, D. Vassallo, and D. Bolster. Characterizing Reactive Transport Behavior in a Three-Dimensional Discrete Fracture Network. Transport in Porous Media (2021).

52. S. Shriram, D. O'Malley, M. K. Mudunuru, M. R. Sweeney, J. D. Hyman, S. Karra, L. Frash et al. A machine learning framework for rapid forecasting and history matching in unconventional reservoirs. (2021) Scientific Reports.

53. J. D. Hyman, M. R. Sweeney, L. P. Frash, J. W. Carey, and H. S. Viswanathan. Scale-Bridging in Three-Dimensional Fracture Networks: Characterizing the Effects of Variable Fracture Apertures on Network-Scale Flow Channelization. Geophysical Research Letters (2021).

54. **`Liangchao Zou and Vladimir Cvetkovic. Evaluation of Flow-Log Data From Crystalline Rocks With Steady-State Pumping and Ambient Flow. Geophysical Research Letters (2021) < https://doi.org/10.1029/2021GL092741>`_**.

55. H. Ushijima-Mwesigwa, J. D. Hyman, A. Hagberg, I. Safro, S. Karra, C. W. Gable, M. R. Sweeney, and G. Srinivasan. Multilevel graph partitioning for three-dimensional discrete fracture network flow simulations. Mathematical Geosciences (2021).

56. Yingtao Hu, Wenjie Xu, Liangtong Zhan, Liangchao Zou, and Yunmin Chen. "Modeling of solute transport in a fracture-matrix system with a three-dimensional discrete fracture network." Journal of Hydrology (2021).

57. C. R. Romano, R. T. Williams; Evolution of Fault-Zone Hydromechanical Properties in Response to Different Cementation Processes. Lithosphere (2022).

58. J. Krotz, M.R. Sweeney, C.W. Gable, J.D. Hyman, & J.M. Restrepo, (2022). Variable resolution Poisson-disk sampling for meshing discrete fracture networks. Journal of Computational and Applied Mathematics (2022).

CHAPTER

SIXTEEN

DFNWORKS GALLERY

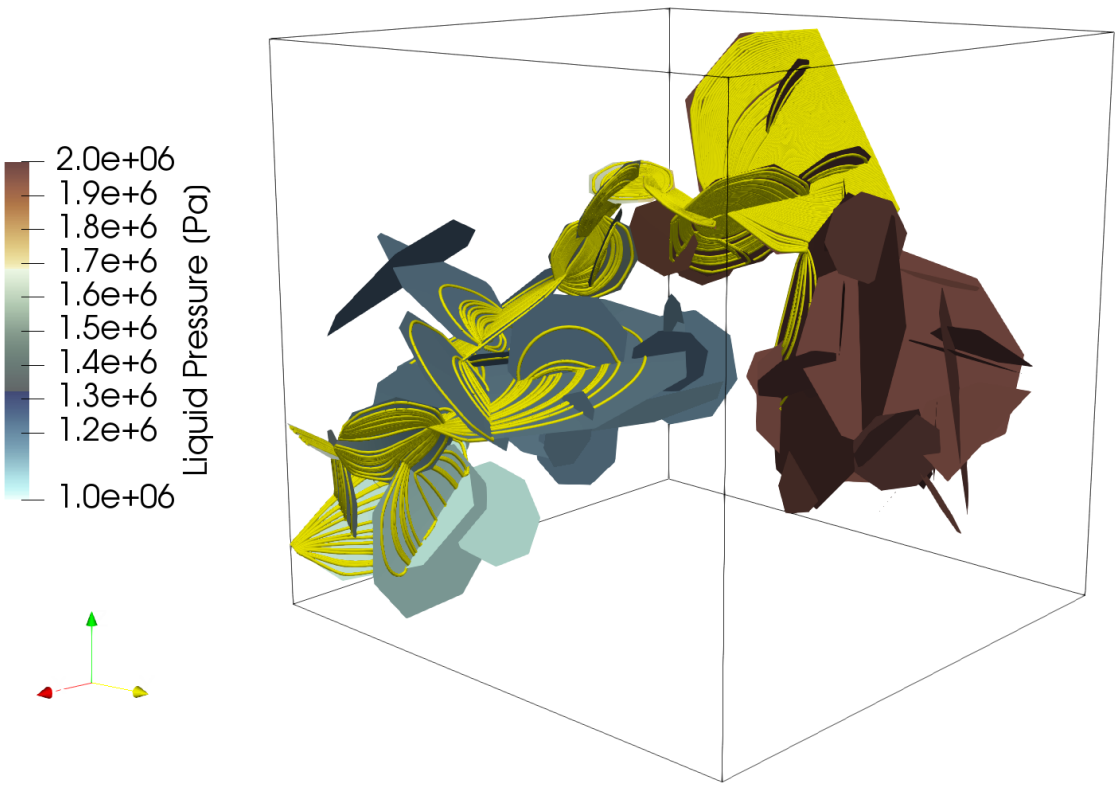


Fig. 1: *Particle Pathlines within a DFN composed of fractures whose radii follow a truncated powerlaw. Fractures are colored by Pressure.*

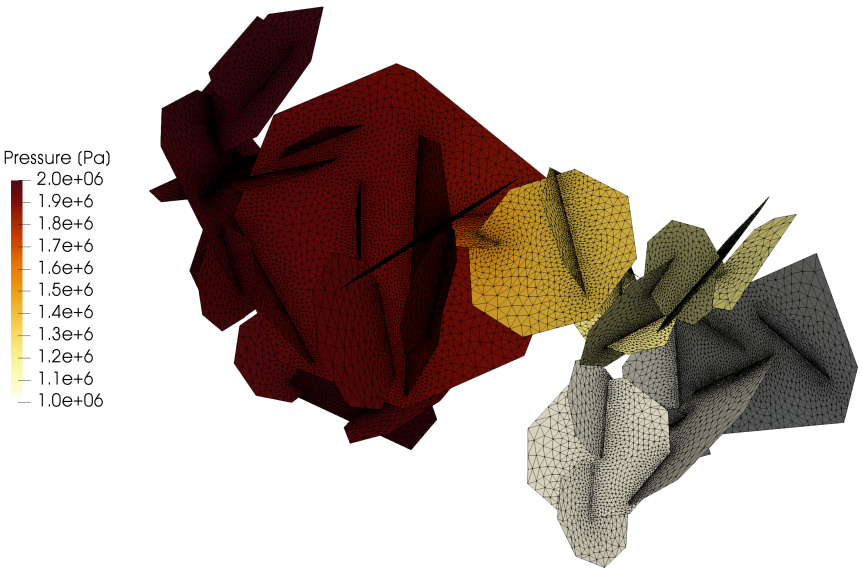


Fig. 2: *Mesh of a DFN composed of fractures whose radii follow a truncated powerlaw. Fractures are colored by Pressure.*

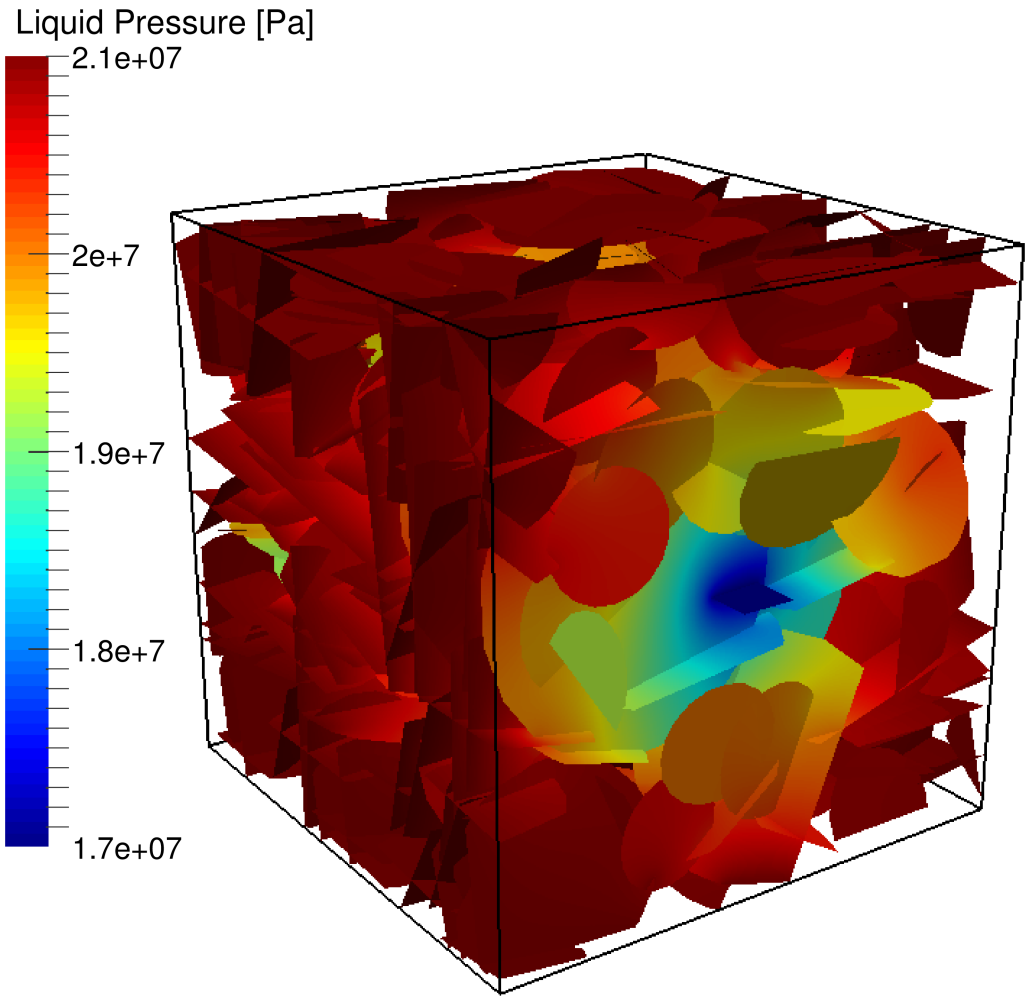


Fig. 3: *Pressure distribution with a hydraulic fracturing simulation*

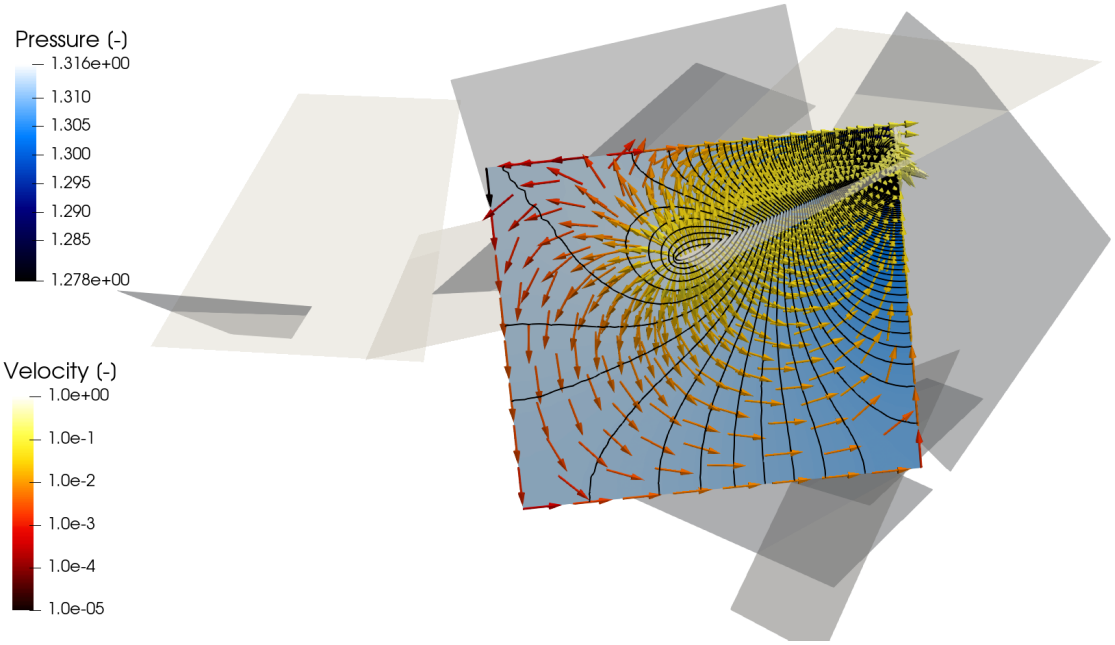


Fig. 4: *Pressure contours and velocity vector field within a dead-end fracture.*

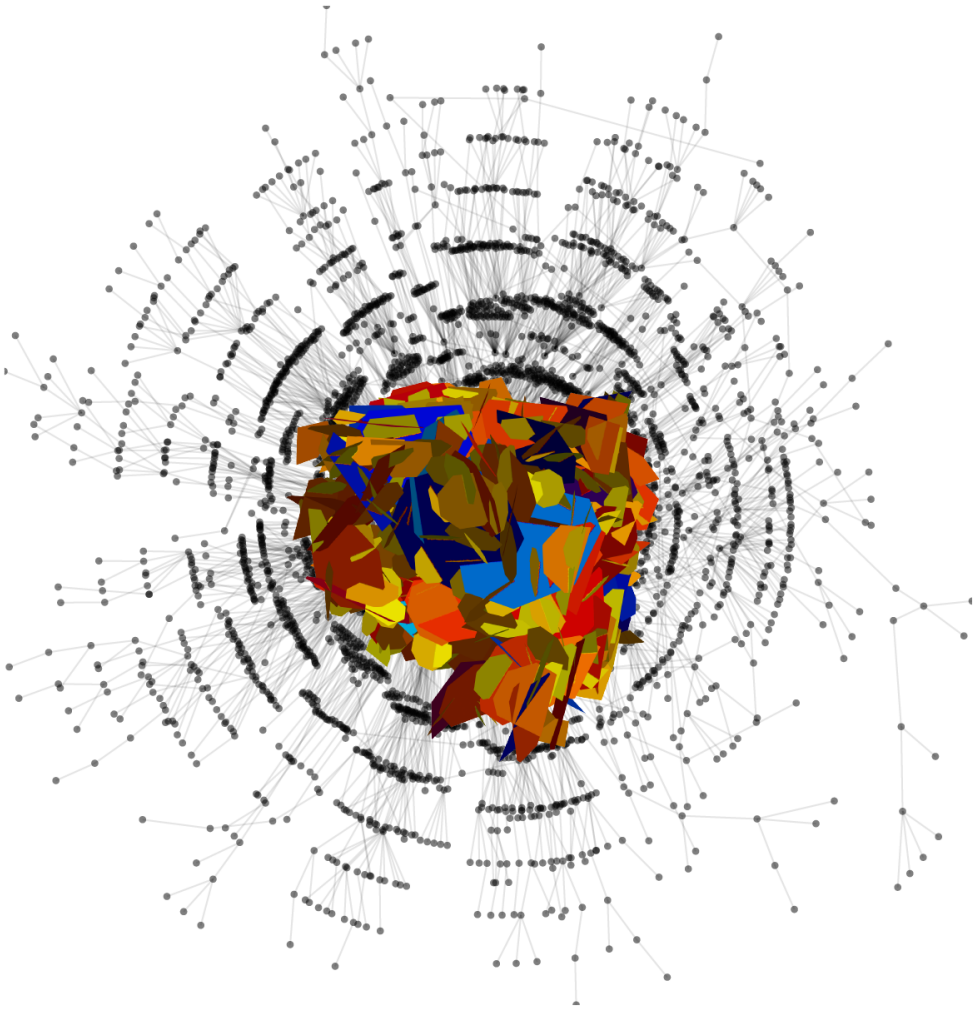


Fig. 5: *DFN along with it's graph representation*

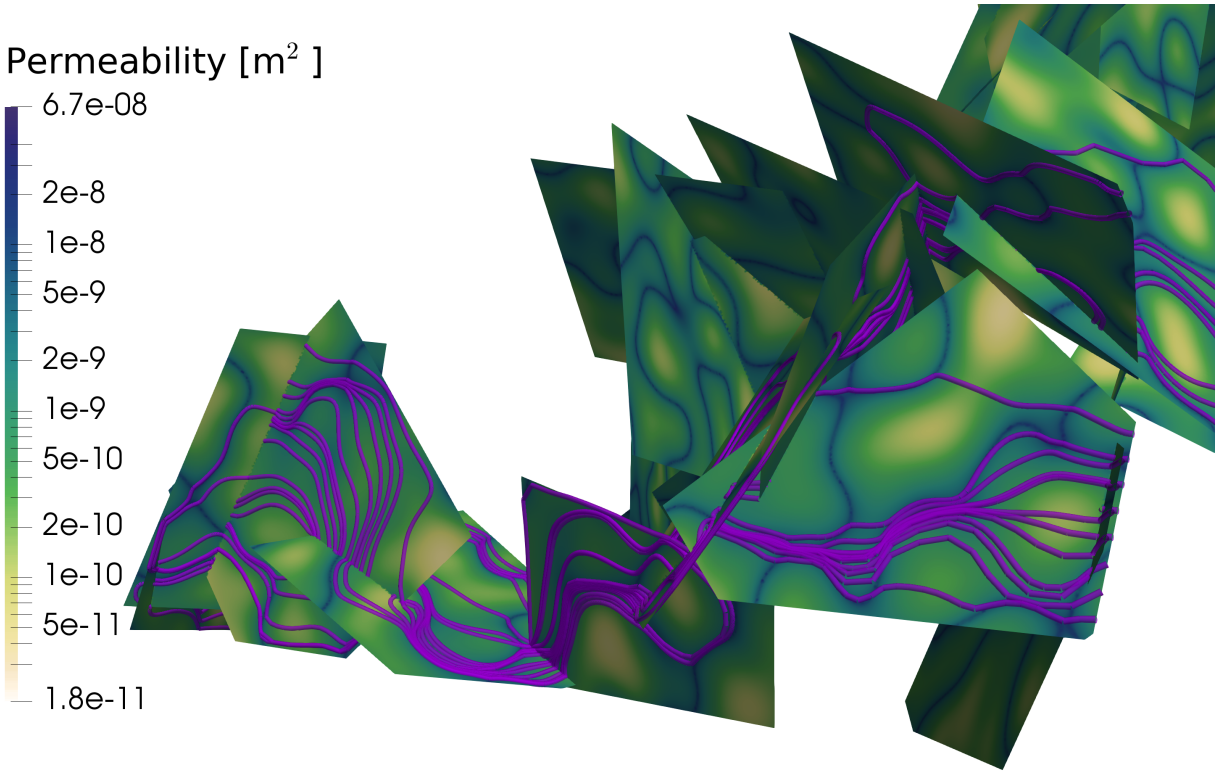


Fig. 6: *DFN with internal aperture variability. Particle Pathlines are shown in purple.*

INDEX

A

add_fracture_source() (*in module pydfnworks.dfnGraph.dfn2graph*), 41
add_fracture_target() (*in module pydfnworks.dfnGraph.dfn2graph*), 41
add_variable_to_mesh() (*in module pydfnworks.dfnGen.meshing.add_attribute_to_mesh*), 30

C

check_dfn_trans_run_files() (*in module pydfnworks.dfnTrans.transport*), 39
check_false_connections() (*in module pydfnworks.dfnGen.meshing.udfm.false_connections*), 31
cleanup_wells() (*in module pydfnworks.dfnGen.well_package.wells*), 47
combine_well_boundary_zones() (*in module pydfnworks.dfnGen.well_package.wells*), 47
copy_dfn_trans_files() (*in module pydfnworks.dfnTrans.transport*), 39
correct_stor_file() (*in module pydfnworks.dfnFlow.fehm*), 36
create_dfn() (*in module pydfnworks.general.dfnworks*), 24
create_dfn_flow_links() (*in module pydfnworks.dfnFlow.flow*), 33
create_dfn_trans_links() (*in module pydfnworks.dfnTrans.transport*), 39
create_graph() (*in module pydfnworks.dfnGraph.dfn2graph*), 42
create_mesh_links() (*in module pydfnworks.dfnGen.meshing.mesh_dfn_helper*), 29
create_network() (*in module pydfnworks.dfnGen.generation.generator*), 25

D

dfn_flow() (*in module pydfnworks.dfnFlow.flow*), 33
dfn_gen() (*in module pydfnworks.dfnGen.generation.generator*), 25
dfn_trans() (*in module pydfnworks.dfnTrans.transport*), 40
dump_fractures() (*in module pydfnworks.dfnGraph.dfn2graph*), 42
dump_hydraulic_values() (*in module pydfnworks.dfnGen.generation.hydraulic_properties*), 27
dump_json_graph() (*in module pydfnworks.dfnGraph.dfn2graph*), 42

E

effective_perm() (*in module pydfnworks.dfnFlow.mass_balance*), 37

F

false_connections.py
    module, 31
fehm() (*in module pydfnworks.dfnFlow.fehm*), 36
find_well_intersection_points() (*in module pydfnworks.dfnGen.well_package.wells*), 48

G

generate_hydraulic_values() (*in module pydfnworks.dfnGen.generation.hydraulic_properties*), 27
graph_transport.py
    module, 44
greedy_edge_disjoint() (*in module pydfnworks.dfnGraph.dfn2graph*), 42

I

inp2gmv() (*in module pydfnworks.dfnGen.meshing.mesh_dfn_helper*), 29
inp2vtk_python() (*in module pydfnworks.dfnGen.meshing.mesh_dfn_helper*), 29

K

k_shortest_paths_backbone() (*in module pydfnworks.dfnGraph.dfn2graph*), 43

L

lagrit2pflotran() (*in module pydfnworks.dfnFlow.pflotran*), 34
load_json_graph() (*in module pydfnworks.dfnGraph.dfn2graph*), 43

M

make_working_directory() (*in module pydfnworks.dfnGen.generation.generator*), 26
map2continuum.py
    module, 30
map_to_continuum() (*in module pydfnworks.dfnGen.meshing.udfm.map2continuum*), 30
mesh_dfn.py
    module, 28
mesh_dfn_helper.py
    module, 29
mesh_network() (*in module pydfnworks.dfnGen.meshing.mesh_dfn*), 28
module
    false_connections.py, 31
    graph_transport.py, 44
    map2continuum.py, 30
    mesh_dfn.py, 28
    mesh_dfn_helper.py, 29
    pydfnworks.dfnFlow.fehm, 36
    pydfnworks.dfnFlow.flow, 33
    pydfnworks.dfnFlow.mass_balance, 37
    pydfnworks.dfnFlow.pflotran, 34
    pydfnworks.dfnGen.generation.generator, 25
    pydfnworks.dfnGen.generation.hydraulic_properties, 27
    pydfnworks.dfnGen.generation.input_checking, 25
    pydfnworks.dfnGen.generation.output_report.gen_output, 26
    pydfnworks.dfnGen.meshing.add_attribute_to_mesh, 30
    pydfnworks.dfnGen.meshing.mesh_dfn, 28
    pydfnworks.dfnGen.meshing.mesh_dfn_helper, 29
    pydfnworks.dfnGen.meshing.udfm.false_connections, 31
    pydfnworks.dfnGen.meshing.udfm.map2continuum, 30
    pydfnworks.dfnGen.meshing.udfm.upscale, 31
    pydfnworks.dfnGen.well_package.wells, 47
    pydfnworks.dfnGraph.dfn2graph, 41
    pydfnworks.dfnGraph.graph_flow, 44
    pydfnworks.dfnGraph.graph_transport, 44
    pydfnworks.dfnTrans.transport, 39
    pydfnworks.general.dfnworks, 24
    upscale.py, 31

O

output_report() (*in module pydfnworks.dfnGen.generation.output_report.gen_output*), 26

P

parse_pflotran_vtk_python() (*in module pydfnworks.dfnFlow.pflotran*), 34
pflotran() (*in module pydfnworks.dfnFlow.pflotran*), 35
pflotran_cleanup() (*in module pydfnworks.dfnFlow.pflotran*), 35
plot_graph() (*in module pydfnworks.dfnGraph.dfn2graph*), 43
pydfnworks.dfnFlow.fehm
    module, 36
pydfnworks.dfnFlow.flow
    module, 33
pydfnworks.dfnFlow.mass_balance
    module, 37
pydfnworks.dfnFlow.pflotran
    module, 34
pydfnworks.dfnGen.generation.generator
    module, 25
pydfnworks.dfnGen.generation.hydraulic_properties
    module, 27
pydfnworks.dfnGen.generation.input_checking
    module, 25
pydfnworks.dfnGen.generation.output_report.gen_output
    module, 26
pydfnworks.dfnGen.meshing.add_attribute_to_mesh
    module, 30
pydfnworks.dfnGen.meshing.mesh_dfn
    module, 28
pydfnworks.dfnGen.meshing.mesh_dfn_helper
    module, 29
pydfnworks.dfnGen.meshing.udfm.false_connections
    module, 31
pydfnworks.dfnGen.meshing.udfm.map2continuum
    module, 30
pydfnworks.dfnGen.meshing.udfm.upscale
    module, 31
pydfnworks.dfnGen.well_package.wells
    module, 47
pydfnworks.dfnGraph.dfn2graph
    module, 41
pydfnworks.dfnGraph.graph_flow
    module, 44
pydfnworks.dfnGraph.graph_transport
    module, 44
pydfnworks.dfnTrans.transport

module, 39
pydfnworks.general.dfnworks
module, 24

R

run_dfn_trans() (*in module pydfnworks.dfnTrans.transport*), 40
run_graph_flow() (*in module pydfnworks.dfnGraph.graph_flow*), 44
run_graph_transport() (*in module pydfnworks.dfnGraph.graph_transport*), 44
run_lagrit_script() (*in module pydfnworks.dfnGen.meshing.mesh_dfn_helper*), 30

S

set_flow_solver() (*in module pydfnworks.dfnFlow.flow*), 34

T

tag_well_in_mesh() (*in module pydfnworks.dfnGen.well_package.wells*), 48

U

upscale() (*in module pydfnworks.dfnGen.meshing.udfm.upscale*), 31
upscale.py
module, 31

W

write_perms_and_correct_volumes_areas() (*in module pydfnworks.dfnFlow.pflotran*), 35

Z

zone2ex() (*in module pydfnworks.dfnFlow.pflotran*), 36