dfnWorks Documentation

Release 2.3

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

Apr 20, 2020

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION

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

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

1.1 Citing dfnWorks

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

BibTex:

```
  @article{hyman2015dfnWorks,
    title={dfnWorks: A discrete fracture network framework
for modeling subsurface flow and transport},
    author={Hyman, Jeffrey D and Karra, Satish and Makedonska,
Nataliia and Gable, Carl W and Painter, Scott L
and Viswanathan, Hari S},
    journal={Computers \& Geosciences},
    volume={84},
    pages={10--19},
    year={2015},
    publisher={Elsevier}
  }
```

1.2 What's new in v2.3?

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

1.3 What's new in v2.2?

- 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

1.4 What's new in v2.1?

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

1.5 What's new in v2.0?

- New dfnGen C++ code which is much faster than the Mathematica dfnGen. This code has successfully generated networks with 350,000+ fractures.
- Increased functionality in the pydfnworks package for more streamlined workflow from dfnGen through visualization.

1.6 Where can one get dfnWorks?

dfnWorks 2.3 can be downloaded from https://hub.docker.com/r/ees16/dfnworks

dfnWorks 2.3 can be downloaded from https://github.com/lanl/dfnWorks/

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

1.7 Installation

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

1.7.1 Operating Systems

dfnWorks currently runs on Macs and Unix machine running Ubuntu.

1.7.2 Python

pydfnworks is supported on Python 3. The software authors 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`, and `re`.

1.7.3 pydfnworks

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

1.7.4 dfnGen

dfnGen primarily involves two steps: FRAM (the feature rejection algorithm for meshing) and LaGriT, the meshing tool box used to create a conforming Delaunay triangulation of the network.

FRAM

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

LaGriT

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

1.7.5 dfnFlow

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

PFLOTRAN

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

FEHM

FEHM is a subsurface multiphase flow code developed at Los Alamos National Laboratory.

1.7.6 dfnTrans

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

1.7.7 dfnGraph

dfnGraph is a suite of graph-based methods for use with DFN generated using dfnWorks DFN. This suite includes multiple methods to prune a DFN and simulate flow and transport in pipe-networks derived from a DFN. dfnGraph uses the networkX python software to handle graph representations.

1.7.8 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.

1.7.9 Paraview

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

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

1.8 Using pydfnworks in your Python scripts

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

```
import pydfnworks
```

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

1.9 About this manual

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

1.10 Contributors

- Jeffrey Hyman
- Satish Karra
- Nataliia Makedonska
- Carl Gable
- Hari Viswanathan
- Matt Sweeney
- Shriram Srinivasan
- Quan Bui (now at LLNL)
- Jeremy Harrod (now at Spectra Logic)
- Scott Painter (now at ORNL)
- Thomas Sherman (University of Notre Dame)

1.11 Contact

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

1.12 Copyright information

CHAPTER

TWO

SETTING UP DFNWORKS

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

2.1 Turn on X forwarding if on server

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

```
$ ssh -X SERVER_NAME
```

2.2 Go to the dnfWorks repository

```
$ cd ~/dfnWorks/
```

2.3 Fix paths in test directory

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

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

2.4 Set the PETSC, PFLOTRAN, Python, and LaGriT 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 /pydfnworks/pydfnworks/paths.py
```

For example:

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

2.5 Setup the Python package pydfnworks

Go up a directory:

```
$ cd ..
```

If the user has admin privelges:

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

If the user DOES NOT have admin priveleges:

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

CHAPTER

THREE

RUNNING DFNWORKS FROM DOCKER

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:v2.2
```

3.1 Setting Up X-Forwarding

On macOS:

To setup X-forwarding on macOS, you will need `homebrew`,``socat`` and `xquartz`. To install homebrew visit https://brew.sh/. or run

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/
↪master/install)"
```

To install, `socat` and `xquartz` run:

```
brew install socat
brew cask install xquartz
```

`socat` is used to expose the local XQuartz socker over a TCP port, enabling the Docker container to exchange display information with your local machine.

On Linux:

```
TODO
```

3.2 Running the dfnWorks container

The base command for running the dfnWorks container is:

```
docker run -ti ees16/dfnworks:v2.2
```

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.

In order to exchange display information, we will need to pass in the DISPLAY variable from host to container.

On macOS, this is `-e DISPLAY=docker.for.mac.host.internal:0`. On Linux, it is `TODO`.

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

On macOS:

```
open -a XQuartz
socat TCP-LISTEN:6000,reuseaddr,fork UNIX-CLIENT:\"$DISPLAY\"
docker run -ti \
       -e DISPLAY=docker.for.mac.host.internal:0 \
       -v /Users/yourname/dfnworks-example:/dfnWorks/work \
       dfnworks:latest
```

On Linux:

```
TODO
```

3.3 Troubleshooting

If you recieve a warning that port 6000 is in use, run either `lsof -i TCP:6000` or `netstat -vanp tcp | grep 6000` to isolate the process listening on this port.

Then, use `kill -i <PID>` to kill the process and re-run `xquartz`, `socat`, and `docker run`.

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.

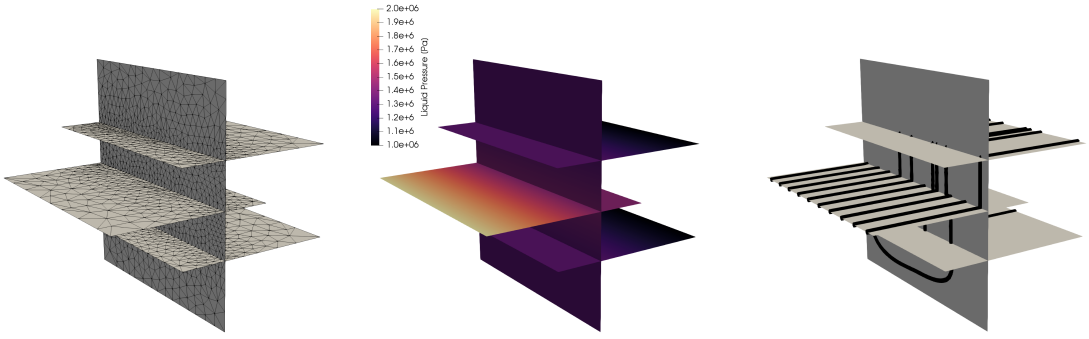


Fig. 1: *The meshed network of four rectangular fractures.*

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

velocity 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.

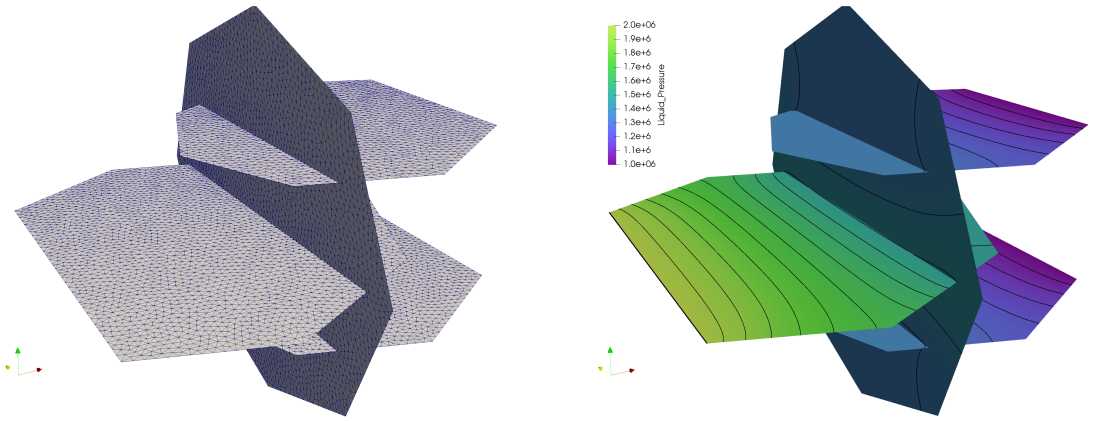


Fig. 2: *The uniformly meshed network of four circular fractures.*

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

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.

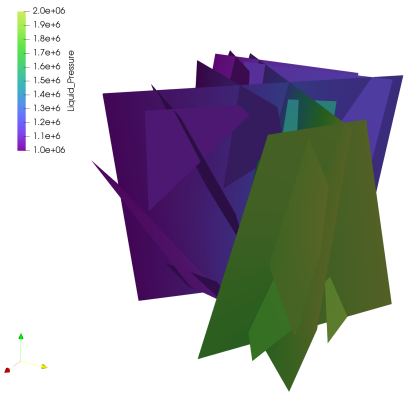
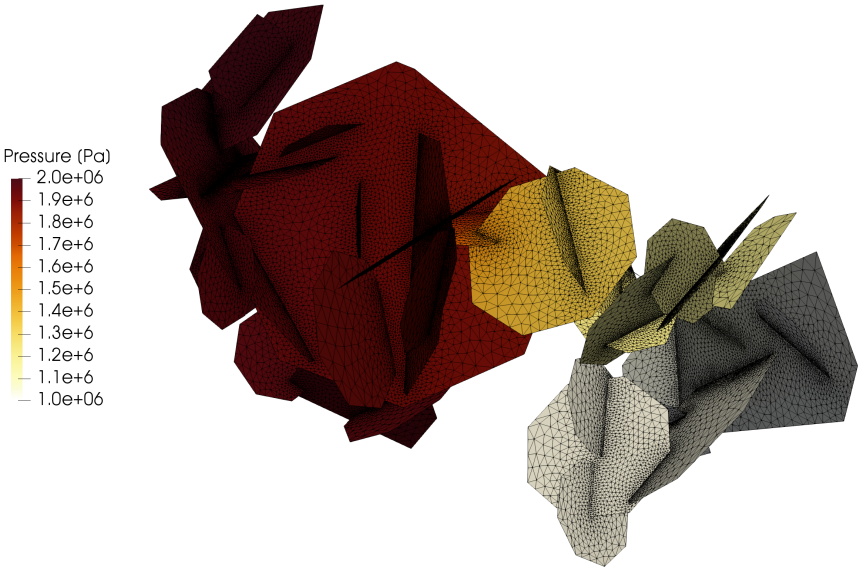


Fig. 3: *Pressure solution on with rectangular fractures whose lengths following a exponential distribution. Gradient is aligned with the Y-Axis*



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.

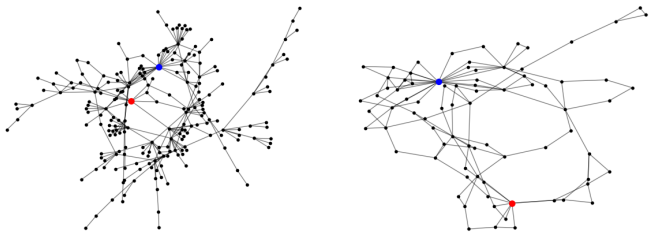


Fig. 4: *(left) Graph based on DFN topology. Each vertex is a fracture in the network. The inflow boundary is colored blue and the outflow is colored red. (right) 2-Core of the graph to the left.*

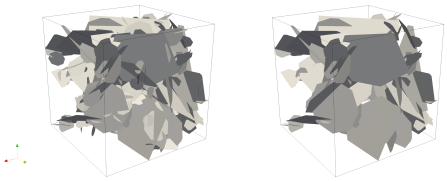


Fig. 5: *(left) Original DFN (right) DFN corresponding to the 2-core of the DFN to the left.*

4.6 In Fracture Variability

Location: examples/in_fracture_var/

This example runs the four rectangular fracture case with variable fracture aperture in each plane. The aperture field is modeled as a correlated multi-variant Gaussian random field. The aperture values are in the aper_node.dat file and the permeabilities are in perm_node.dat. The command line argument indicating that there is spatially variable aperture field is -cell. In fracture variability is not supported for FEHM runs at this time.

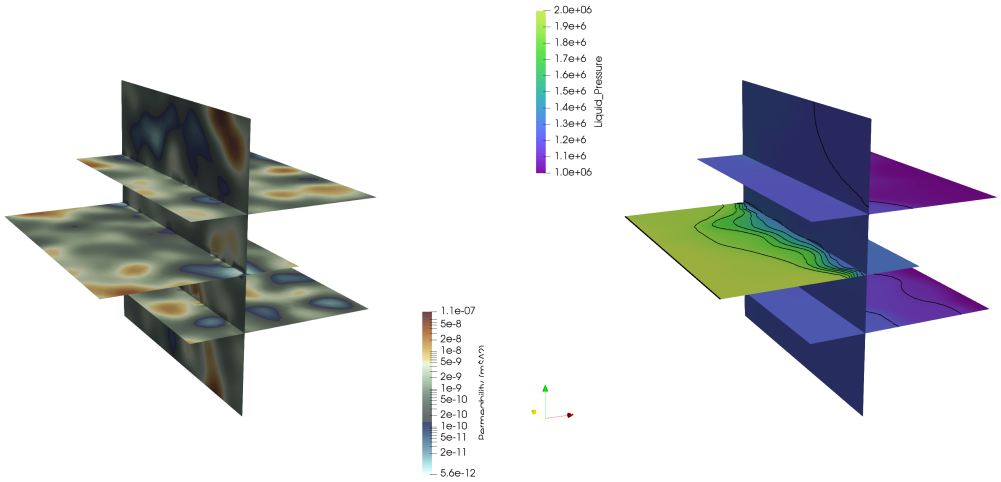


Fig. 6: *(left) The meshed network of four rectangular fractures colored by permeability, which is spatially variable on each fracture. (right) The network of four fractures, colored by pressure solution. Black lines are contours in the pressure field.*

CHAPTER

FIVE

EXAMPLE APPLICATIONS

5.1 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.

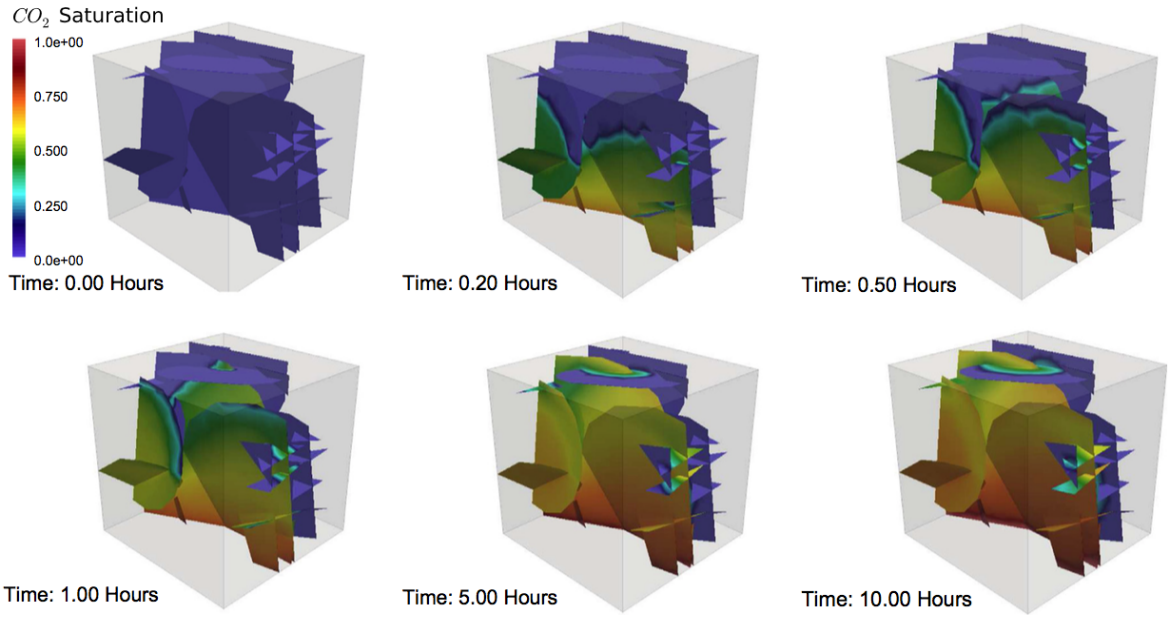


Fig. 1: *Temporal evolution of supercritical |CO2| displacing water in a meter cube DFN containing 24 fractures. The DFN is initially fully saturated with water, (top left time 0 hours) and supercritical |CO2| is slowly injected into the system from the bottom of the domain to displace the water for 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.*

5.2 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.

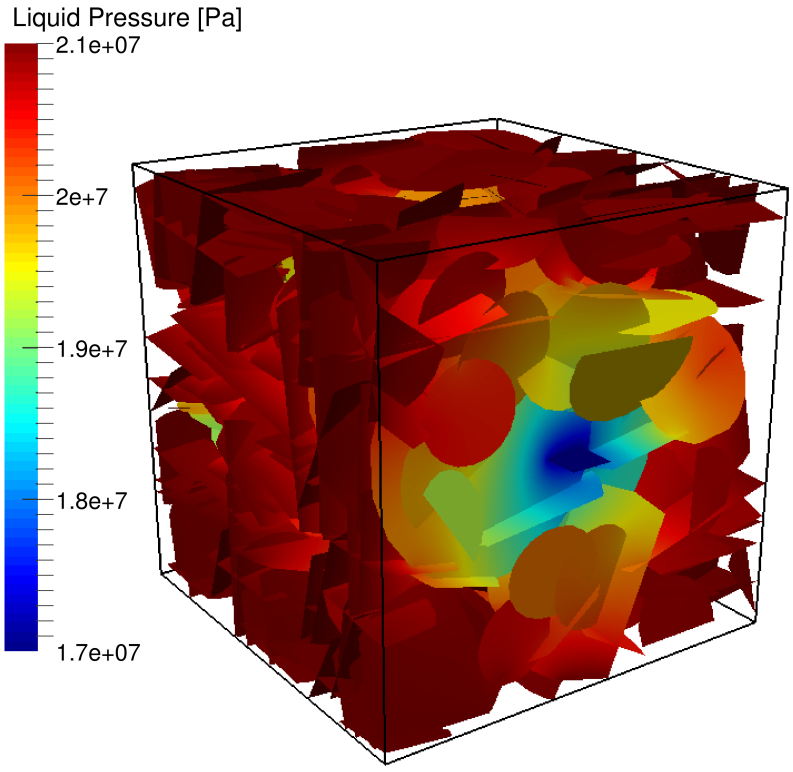


Fig. 2: *Pressure in a well used for hydraulic fracturing.*

5.3 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.

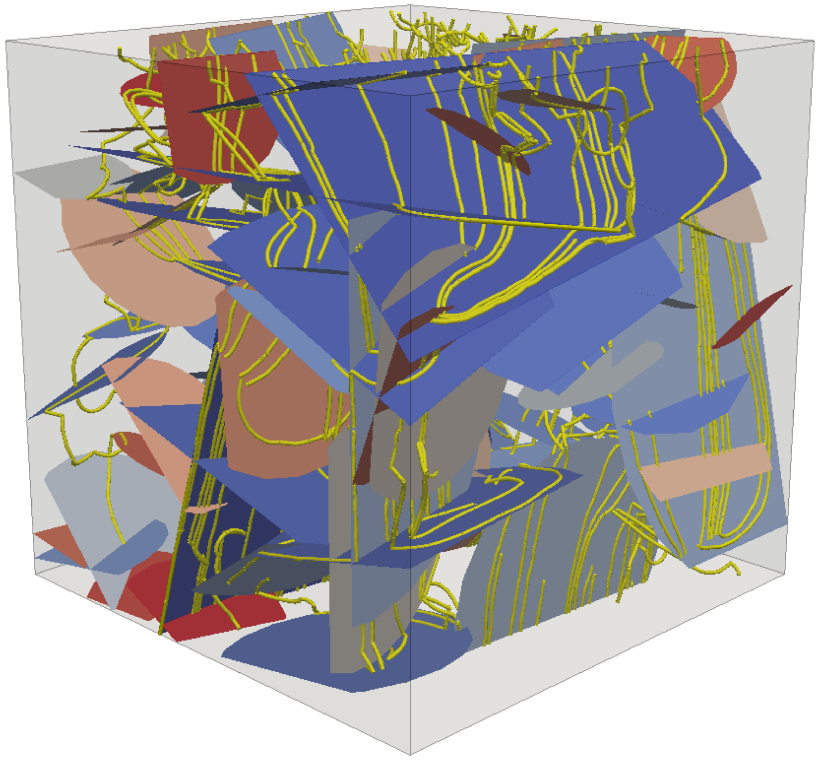


Fig. 3: *Simulated particle trajectories in fractured granite at Forsmark, Sweden.*

Sources:

- Adler, P.M., Thovert, J.-F., Mourzenko, V.V., 2012. Fractured Porous Media. Oxford University Press, Oxford, United Kingdom.
- Bachu, S., 2002. Sequestration of 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

SIX

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., 36(4):A1871–A1894, 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, 147:500–509, 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, 51(9):7289–7308, 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, 51(10):8646–8657, 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, 84:10–19, 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, 19(5):1123–1137, 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, 54(4):488–497, 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, 374(2078):20150426, 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, 23(8):1896–1909, Aug 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, 94:486 – 497, 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), pages 1532–1539, May 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, 10(5):326–342, 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, 119:25 – 33, 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, 54(5):3243–3258, 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, 16(4):1948– 1968, 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, 99(1):013110, 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, 46. 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, 123. 248501.

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

CHAPTER

SEVEN

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-Version2.0/pydfnworks/`:

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

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

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

7.1 DFN Class and Setup

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

Class for DFN Generation and meshing

*** jobname**
name of job, also the folder where output files are stored

*** ncpu**
number of CPUs used in the job

*** dfnGen file**
the name of the dfnGen input file

*** dfnFlow file**
the name of the dfnFlow input file

*** dfnTrans file**
the name of the dfnFlow input file

*** local prefix**
indicates that the name contains only the most local directory

* **vtk_file**
  the name of the VTK file

* **inp_file**
  the name of the INP file

* **uge_file**
  the name of the UGE file

* **mesh_type**
  the type of mesh

* **perm_file**
  the name of the file containing permeabilities

* **aper_file**
  the name of the file containing apertures

* **perm_cell file**
  the name of the file containing cell permeabilities

* **aper_cell_file**
  the name of the file containing cell apertures

* **freeze**
  indicates whether the class attributes can be modified

* **h**
  FRAM length scale

add_fracture_source(*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

add_fracture_target(*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

check_dfn_trans_run_files()
Ensures that all files required for dfnTrans run are in the current directory

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

Returns

Return type None

Notes

None

check_input(*input_file=''*, *output_file=''*)
Check input file for DFNGen to make sure all necessary parameters are defined

Input Format Requirements:

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

Parameters

- **input_file** (*string*) – name of dfnGen input file
- **output_file** (*string*) – Name of stripped down input file for DFNGen (input_file_clean.dat)

Returns

Return type None

Notes

There are warnings and errors raised in this function. Warning will let you continue while errors will stop the run. Continue past warnings are your own risk.

copy_dfn_trans_files()
Creates symlink to dfnTrans Execuateble and copies input files for dfnTrans into working directory

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

Returns

Return type None

correct_stor_file()
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

create_dfn_flow_links(*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

create_dfn_trans_links(*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

create_graph(*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

create_network()
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.

define_paths()
Defines enviromental variables for use in dfnWorks. The user must change these to match their workspace.

Parameters **None** –

Returns

Return type None

Notes

Enviromental variables are set to executables

dfn_flow(*dump_vtk=True*, *effective_perm=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

dfn_gen(*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

dfn_trans()
Primary driver for dfnTrans.

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

Returns

Return type None

dump_fractures(*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

dump_json_graph(*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

dump_time(*function_name*, *time*)
Write run time for a funcion to the jobname_run_time.txt file

Parameters

- **self** (*object*) – DFN Class
- **function_name** (*string*) – Name of function that was timed
- **time** (*float*) – Run time of function in seconds

Returns

Return type None

Notes

While this function is working, the current formulation is not robust through the entire workflow

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

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

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

fehm()
Run FEHM

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

Returns

Return type None

Notes

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

greedy_edge_disjoint(*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

inp2gmv(*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

inp2vtk_python()

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

k_shortest_paths_backbone(*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

lagrit2pflotran(*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

- **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

legal()
Print the legal LANL statement for dfnWorks.

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

Returns

Return type None

Notes

None

load_json_graph(*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

make_working_directory(*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.

map_to_continuum(*l*, *orl*)
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

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

mesh_dfm()
This function generates a conforming DFM mesh using the full_mesh.inp as input. Visualization mode must be turned off in the DFN input card.

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

Returns

Return type None

Notes

mesh_network(*prune=False*, *uniform_mesh=False*, *production_mode=True*, *refine_factor=1*, *slope=2*, *visual_mode=None*)
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
- **refine_factor** (*float*) – Determines distance for mesh refinement (default=1)
- **slope** (*float*) – Slope of piecewise linear function determining rate of coarsening.
- **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

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

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

Notes:

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

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

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

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

Returns

Return type None

Notes

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

pflotran_cleanup(*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

plot_graph(*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

print_run_time()
Read in run times from file and and print to screen with percentages

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

Returns

Return type None

Notes

This will dump out all values in the run file, not just those from the most recent run

pydfnworks = <module 'pydfnworks' from '/Users/jhyman/.local/lib/python3.7/site-packag

run_dfn_trans()
Execute dfnTrans

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

Returns

Return type None

run_graph_flow(*inflow*, *outflow*, *Pin*, *Pout*, *fluid_viscosity=0.00089*)
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

run_graph_transport(*Gtilde*, *nparticles*, *partime_file*, *frac_id_file*, *frac_porosity=1.0*, *tdrw_flag=False*, *matrix_porosity=0.02*, *matrix_diffusivity=1e-11*)
Run particle tracking on the given NetworkX graph

Parameters

- **self** (*object*) – DFN Class
- **Gtilde** (*NetworkX graph*) – obtained from graph_flow
- **nparticles** (*int*) – number of particles
- **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
- **frac_porosity** (*float*) – porosity of fracture, default is 1.0
- **tdrw_flag** (*Bool*) – if False, matrix_porosity, matrix_diffusivity are ignored
- **matrix_porosity** (*float*) – default is 0.02
- **matrix_diffusivity** (*float*) – default is 1e-11 in SI units

Notes

Information on individual functions is found therein

set_flow_solver(*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

uncorrelated(*mu*, *sigma*, *path='../'*)
Creates Fracture Based Log-Normal Permeability field with mean mu and variance sigma. Aperture is dervived using the cubic law

Parameters

- **mu** (*double*) – Mean of LogNormal Permeability field
- **sigma** (*double*) – Variance of permeability field
- **path** (*string*) – path to original network. Can be current directory

Returns

Return type None

Notes

mu is the mean of perm not log(perm)

upscale(*mat_perm*, *mat_por*)
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

write_perms_and_correct_volumes_areas()
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

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

Parameters

- **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*) – Boundary cells are moved a distance of boundary_cell_area 1e-1

Returns

Return type None

Notes

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

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

Parameters **None** –

Returns **options** – command line options

Return type argparse function

Notes

Options:

-name [string] Path to working directory (Mandatory)

-ncpu [int] Number of CPUS (Optional, default=4)

-input [string] Input file with paths to run files (Mandatory if the next three options are not specified)

-gen [string] Generator Input File (Mandatory, can be included within the input file)

-flow [string] PFLORAN Input File (Mandatory, can be included within the input file)

-trans [string] Transport Input File (Mandatory, can be included within the input file)

-prune_file [string] Absolute path to the prune Input File

-path [string] Path to another DFN run that you want to base the current run from

-cell [bool] True/False Set True for use with cell based aperture and permeabuility (Optional, default=False)

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

7.2 dfnGen

7.2.1 Processing generator input

pydfnworks.dfnGen.gen_input.**check_input**(*self*, *input_file=''*, *output_file=''*)
Check input file for DFNGen to make sure all necessary parameters are defined

Input Format Requirements:

- Each parameter must be defined on its own line (separate by newline)

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

Parameters

- **input_file** (*string*) – name of dfnGen input file
- **output_file** (*string*) – Name of stripped down input file for DFNGen (input_file_clean.dat)

Returns

Return type None

Notes

There are warnings and errors raised in this function. Warning will let you continue while errors will stop the run. Continue past warnings are your own risk.

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

*** params**
list of parameters specified in the input file.

Type list

*** minFracSize**
the minimum fracture size.

Type float

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

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

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

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

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

error(*errString*)
print an error

Parameters **errString** (*str*) – a string describing the error

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

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

Returns

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Input:

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

Output:

- returns negative value of list length to indicate incorrect length and provide meaningful error message
- prints error and exits if a value of the wrong type is found in the list

- returns None if successful

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

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

class pydfnworks.dfnGen.distributions.**distr**(*params*, *numEdistribs*, *numRdistribs*, *minFracSize*)
Verifies the fracture distribution input parameters for dfnGen.

params
list Parameters for dfnGen

numEdistribs
int Number of ellipse family distributions

numRdistribs
int Number of rectangle family distributions

minFracSize
double Minimum fracture size

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

Parameters **prefix** – str Indicates shapes that the beta distribution describes. 'e' if they are ellipses, 'r' if they are rectangles.

constant_dist(*prefix*)
Verifies parameters for constant distribution of fractures

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

exponential_dist(*prefix*)
Verifies parameters for exponential distribution of fractures.

lognormal_dist(*prefix*)
Verifies all logNormal Parameters for ellipses and Rectangles.

tpl_dist(*prefix*)
Verifies parameters for truncated power law distribution of fractures.

7.2.2 Running the generator

pydfnworks.dfnGen.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.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.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.

7.2.3 Analysis of Generated DFN

pydfnworks.dfnGen.gen_output.**output_report**(*self*, *radiiFile='radii.dat'*, *familiesFile='families.dat'*, *translationsFile='translations.dat'*, *rejectionsFile='rejections.dat'*, *output_name=''*)

Create PDF report of generator

Notes:

- Set the number of histogram buckets (bins) by changing numBuckets variable in his graphing functions

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

7.3 Meshing - LaGriT

7.3.1 Mesh DFN

pydfnworks.dfnGen.mesh_dfn.**mesh_network**(*self*, *prune=False*, *uniform_mesh=False*, *production_mode=True*, *refine_factor=1*, *slope=2*, *visual_mode=None*)

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
- **refine_factor** (*float*) – Determines distance for mesh refinement (default=1)
- **slope** (*float*) – Slope of piecewise linear function determining rate of coarsening.
- **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

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

7.3.2 LaGrit scripts

pydfnworks.dfnGen.lagrit_scripts.**create_lagrit_scripts**(*visual_mode*, *ncpu*, *refine_factor=1*, *production_mode=True*)

Creates LaGriT script to be mesh each polygon

Parameters

- **visual_mode** (*bool*) – Sets if running if visual mode or in full dump
- **ncpu** (*int*) – Number of cpus

- **refine_factor** (*int*) – Number of times original polygon gets refined
- **production_mode** (*bool*) – Determines if clean up of work files occurs on the fly.

Returns

Return type None

Notes

1. Only ncpu of these files are created
2. Symbolic links are used to rotate through fractures on different CPUs

pydfnworks.dfnGen.lagrit_scripts.**create_merge_poly_files**(*ncpu*, *num_poly*, *fracture_list*, *h*, *visual_mode*, *domain*, *flow_solver*)

Creates a LaGriT script that reads in each fracture mesh, appends it to the main mesh, and then deletes that mesh object. Then duplicate points are removed from the main mesh using EPS_FILTER. The points are compressed, and then written to file.

Parameters

- **ncpu** (*int*) – Number of Processors used for meshing
- **fracture_list** (*list of int*) – List of fracture numbers in the DFN
- **h** (*float*) – Meshing length scale
- **visual_mode** (*bool*) – If True, reduced_mesh.inp will be output. If False, full_mesh.inp is output
- **domain** (*dict*) – Dictionary of x,y,z domain size
- **flow_solver** (*string*) – Name of target flow solver (Changes output files)

Returns **n_jobs** – number of merge jobs

Return type int

Notes

1. Fracture mesh objects are read into different part_*.lg files. This allows for merging of the mesh to be performed in batches.

pydfnworks.dfnGen.lagrit_scripts.**create_parameter_mlgi_file**(*fracture_list*, *h*, *slope=2.0*, *refine_dist=0.5*)

Create parameteri.mlgi files used in running LaGriT Scripts

Parameters

- **num_poly** (*int*) – Number of polygons
- **h** (*float*) – Meshing length scale
- **slope** (*float*) – Slope of coarsening function, default = 2
- **refine_dist** (*float*) – Distance used in coarsening function, default = 0.5

Returns

Return type None

Notes

Set slope = 0 for uniform mesh

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

Parameters **None** –

Returns

Return type None

Notes

These functions are called within LaGriT. It controls the mesh resolution using slope and refine_dist

pydfnworks.dfnGen.lagrit_scripts.**define_zones**()
Processes zone files for particle tracking. All zone files are combined into allboundaries.zone

Parameters **None** –

Returns

Return type None

Notes

None

pydfnworks.dfnGen.lagrit_scripts.**edit_intersection_files**(*num_poly*, *fracture_list*, *path*)
If pruning a DFN, this function walks through the intersection files and removes references to files that are not included in the fractures that will remain in the network.

Parameters

- **num_poly** (*int*) – Number of Fractures in the original DFN
- **fracture_list** (*list of int*) – List of fractures to keep in the DFN

Returns

Return type None

Notes

1. Currently running in serial, but it could be parallelized
2. Assumes the pruning directory is not the original directory

7.3.3 Run meshing in parallel

pydfnworks.dfnGen.run_meshing.**merge_the_meshes**(*num_poly*, *ncpu*, *n_jobs*, *visual_mode*)
Runs the LaGrit Scripts to merge meshes into final mesh

Parameters

- **num_poly** (*int*) – Number of Fractures
- **ncpu** (*int*) – Number of Processors
- **n_jobs** (*int*) – Number of mesh pieces
- **visual_mode** (*bool*) – True/False for reduced meshing

Returns

Return type None

Notes

Meshes are merged in batches for efficiency

pydfnworks.dfnGen.run_meshing.**merge_worker**(*job*)
Parallel worker for merge meshes into final mesh

Parameters **job** (*int*) – job number

Returns **bool**

Return type True if failed / False if successful

Notes

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

Parameters

- **fracture_id** (*int*) – Current Fracture ID number
- **visual_mode** (*bool*) – True/False for reduced meshing
- **num_poly** (*int*) – Total Number of Fractures in the DFN

Returns

Return type None

Notes

If meshing fails, information about that fracture will be put into a directory failure_(fracture_id)

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

Creates a queue of fracture numbers ranging from 1, num_poly

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

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

Parameters

- **fracture_list** (*list*) – Fractures to be meshed
- **visual_mode** (*bool*) – True/False for reduced meshing
- **num_poly** (*int*) – Total Number of Fractures

Returns **True/False** – True - If failure.txt is empty then all fractures have been meshed correctly
False - If failure.txt is not empty, then at least one fracture failed.

Return type bool

Notes

If one fracture fails meshing, program will exit.

pydfnworks.dfnGen.run_meshing.**single_worker**(*work_queue*, *visual_mode*, *num_poly*)
Worker function for parallelized meshing

Parameters

- **work_queue** (*multiprocessing queue*) – Queue of fractures to be meshed
- **visual_mode** (*bool*) – True/False for reduced meshing
- **num_poly** (*int*) – Total Number of Fractures

Returns **True** – If job is complete

Return type bool

7.3.4 Mesh helper methods

pydfnworks.dfnGen.mesh_dfn_helper.**check_dudded_points**(*dudded*, *hard=False*)
Parses LaGrit log_merge_all.txt and checks if number of dudded points is the expected number

Parameters

- **dudded** (*int*) – Expected number of dudded points from params.txt
- **hard** (*bool*) – If hard is false, up to 1% of nodes in the mesh can be missed. If hard is True, no points can be missed.

Returns **True/False** – True if the number of dudded points is correct and False if the number of dudded points is incorrect

Return type bool

Notes

If number of dudded points is incorrect by over 1%, program will exit.

pydfnworks.dfnGen.mesh_dfn_helper.**clean_up_files_after_prune**(*self*)
After pruning a DFN to only include the fractures in prune_file this function removes references to those fractures from params.txt, perm.dat, aperature.dat, and poly_info.dat

Parameters

- **prune_file** (*string*) – Name for file with list of fractures to remain in the network
- **path** (*string*) – Path to files to be modified

Returns

Return type None

Notes

This function should always be run after pruning if flow solution is going to be run

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

Parameters **None** –

Returns

Return type None

Notes

Only runs if production_mode is True

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

Parameters **path** (*string*) – Path to where meshing files are located

Returns

Return type None

Notes

None

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

Parameters **local_jobname** (*string*) – Name of current DFN job (not path)

visual_mode [bool] Determines is reduced_mesh or full_mesh is dumped

Returns

Return type None

Notes

None

pydfnworks.dfnGen.mesh_dfn_helper.**parse_params_file**(*quite=False*)
Reads params.txt file from DFNGen and parses information

Parameters **quite** (*bool*) – If True details are not printed to screen, if False they area

Returns

- **num_poly** (*int*) – Number of Polygons
- **h** (*float*) – Meshing length scale h
- **dudded_points** (*int*) – Expected number of dudded points in Filter (LaGriT)
- **visual_mode** (*bool*) – If True, reduced_mesh.inp is created (not suitable for flow and transport), if False, full_mesh.inp is created
- **domain** (*dict*) – x,y,z domain sizes

Notes

None

7.3.5 Creating an upscaled mesh of the DFN

pydfnworks.dfnGen.map2continuum.**driver_parallel**(*self*, *num_poly*)
This function drives the parallelization of the area sums upscaling.

Parameters

- **self** (*object*) – DFN Class
- **num_poly** (*int*) – Number of fractures

Returns

Return type None

Notes

None

pydfnworks.dfnGen.map2continuum.**lagrit_build**()
This function creates the build_octree.mlgi lagrit script.

Parameters **None** –

Returns

Return type None

Notes

None

pydfnworks.dfnGen.map2continuum.**lagrit_driver**(*nx*, *ny*, *nz*, *num_poly*, *normal_vectors*, *points*)

This function creates the main lagrit driver script, which calls all lagrit scripts.

Parameters

- **ni** (*int*) – Number of cells in each direction
- **num_poly** (*int*) – Number of fractures
- **normal_vectors** (*array*) – Array containing normal vectors of each fracture
- **points** (*array*) – Array containing a point on each fracture

Returns

Return type None

Notes

None

pydfnworks.dfnGen.map2continuum.**lagrit_hex_to_tet**()

This function creates the hex_to_tet.mlgi lagrit script.

Parameters **None** –

Returns

Return type None

Notes

None

pydfnworks.dfnGen.map2continuum.**lagrit_intersect**()

This function creates the intersect_refine.mlgi lagrit scripts.

Parameters **None** –

Returns

Return type None

Notes

None

pydfnworks.dfnGen.map2continuum.**lagrit_parameters**(*orl*, *x0*, *x1*, *y0*, *y1*, *z0*, *z1*, *nx*, *ny*, *nz*, *h*)

This function creates the parameters_octree_dfn.mlgi lagrit script.

Parameters

- **orl** (*int*) – Number of total refinement levels in the octree
- **i1** (*i0,*) – Extent of domain in x, y, z directions
- **ni** (*int*) – Number of cells in each direction

Returns

Return type None

Notes

None

pydfnworks.dfnGen.map2continuum.**lagrit_remove**()
This function creates the remove_cells.mlgi lagrit script.

Parameters **None** –

Returns

Return type None

Notes

None

pydfnworks.dfnGen.map2continuum.**lagrit_run**()
This function executes the lagrit scripts.

Parameters **None** –

Returns

Return type None

Notes

None

pydfnworks.dfnGen.map2continuum.**lagrit_strip**(*num_poly*)
This function strips and replaces the headers of the files, which is needed to assign the fracture areas to a mesh object.

Parameters **num_poly** (*int*) – Number of fractures

Returns

Return type None

Notes

None

pydfnworks.dfnGen.map2continuum.**map_to_continuum**(*self*, *l*, *orl*)
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

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.map2continuum.**upscale_parallel**(*f_id*)
Generates lagrit script that makes mesh files with area sums.

Parameters **f_id** (*int*) – Fracture index

Returns

Return type None

Notes

None

pydfnworks.dfnGen.map2continuum.**worker**(*tasks_to_accomplish*, *tasks_that_are_done*)
Worker function for python parallel. See multiprocessing module documentation for details.

Parameters

- **tasks_to_accomplish** – Processes still in queue
- **tasks_that_are_done** – Processes complete

Notes

None

pydfnworks.dfnGen.upscale.**upscale**(*self*, *mat_perm*, *mat_por*)
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

7.4 dfnFlow

7.4.1 Running Flow

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*, *effective_perm=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

pydfnworks.dfnFlow.flow.**uncorrelated**(*self*, *mu*, *sigma*, *path='../'*)

Creates Fracture Based Log-Normal Permeability field with mean mu and variance sigma. Aperture is dervived using the cubic law

Parameters

- **mu** (*double*) – Mean of LogNormal Permeability field
- **sigma** (*double*) – Variance of permeability field
- **path** (*string*) – path to original network. Can be current directory

Returns

Return type None

Notes

mu is the mean of perm not log(perm)

7.4.2 Running Flow: PFLOTRAN

functions for using pflotran in dfnworks

pydfnworks.dfnFlow.pflotran.**check_pflotran_convergence**(*pflotran_input_file*)

checks pflotran_input_file.out for convergence

Parameters **pflotran_input_file** (*string*) – pflotran_input_file

Returns

- *bool*
- *True if solver converged / False if not*

Notes

pydfnworks.dfnFlow.pflotran.**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.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

- **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=''*)

Replace CELL_DATA with POINT_DATA in the VTK output. :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*, *restart=False*, *restart_file=''*)

Run PFLOTRAN. Copy PFLOTRAN run file into working directory and run with ncpus

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

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

- **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*) – Boundary cells are moved a distance of boundary_cell_area 1e-1

Returns

Return type None

Notes

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

7.4.3 Running Flow: FEHM

pydfnworks.dfnFlow.fehm.**correct_perm_for_fehm**()
FEHM wants an empty line at the end of the perm file This functions adds that line return

Parameters **None** –

Returns

Return type None

Notes

Only adds a new line if the last line is not empty

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

7.4.4 Processing Flow

pydfnworks.dfnFlow.mass_balance.**dump_effective_perm**(*local_jobname*, *mass_rate*, *volume_rate*, *domain*, *direction*, *inflow_pressure*, *outflow_pressure*)
Compute the effective permeability of the DFN and write it to screen and to the file local_jobname_effective_perm.dat

Parameters

- **local_jobname** (*string*) – Jobname
- **mass_rate** (*float*) – Mass flow rate through inflow boundary
- **volume_rate** (*float*) – Volumetric flow rate through inflow boundary
- **direction** (*string*) – Primary direction of flow (x, y, or z)
- **domain** (*dict*) – Dictionary of domain sizes in x, y, z
- **inflow_pressure** (*float*) – Inflow boundary pressure
- **outflow_pressure** (*float*) – Outflow boundary pressure

Returns

Return type None

Notes

Information is written into (local_jobname)_effective_perm.txt

pydfnworks.dfnFlow.mass_balance.**effective_perm**(*self*)

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

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

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

pydfnworks.dfnFlow.mass_balance.**flow_rate**(*darcy_vel_file*, *boundary_file*)

Calculates the flow rate across the inflow boundary

Parameters

- **darcy_vel_file** (*string*) – Name of concatenated Darcy velocity file
- **boundary_file** (*string*) – ex file for the inflow boundary

Returns

- **mass_rate** (*float*) – Mass flow rate across the inflow boundary
- **volume_rate** (*float*) – Volumetric flow rate across the inflow boundary

Notes

None

pydfnworks.dfnFlow.mass_balance.**get_domain**()

Return dictionary of domain x,y,z by calling parse_params_file

Parameters **None** –

Returns **domain** – Dictionary of domain sizes in x, y, z

Return type dict

Notes

parse_params_file() is in mesh_dfn_helper.py

pydfnworks.dfnFlow.mass_balance.**parse_pflotran_input**(*pflotran_input_file*)
Walk through PFLOTRAN input file and find inflow boundary, inflow and outflow pressure, and direction of flow

Parameters **pflotran_input_file** (*string*) – Name of PFLOTRAN input file

Returns

- **inflow_pressure** (*double*) – Inflow Pressure boundary condition
- **outflow_pressure** (*float*) – Outflow pressure boundary condition
- **inflow_file** (*string*) – Name of inflow boundary .ex file
- **direction** (*string*) – Primary direction of flow x, y, or z

Notes

Currently only works for Dirichlet Boundary Conditions

7.5 dfnTrans

7.5.1 Running Transport

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

7.6 dfnGraph

7.6.1 General Graph Functions

pydfnworks.dfnGraph.dfn2graph.**add_area**(*G*, *fracture_info='fracture_info.dat'*)
Read Fracture aperture from fracture_info.dat and load on the edges in the graph. Graph must be intersection to node representation

Parameters

- **G** (*NetworkX Graph*) – networkX graph
- **fracture_info** (*str*) – filename for fracture information

Returns

Return type None

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.**add_perm**(*G*, *fracture_info='fracture_info.dat'*)
Add fracture permeability to Graph. If Graph representation is fracture, then permeability is a node attribute. If graph representation is intersection, then permeability is an edge attribute

Parameters

- **G** (*networkX graph*) – NetworkX Graph based on the DFN
- **fracture_infor** (*str*) – filename for fracture information

Notes

pydfnworks.dfnGraph.dfn2graph.**add_weight**(*G*)
Compute weight w = K*A/L associated with each edge :param G: networkX graph :type G: NetworkX Graph

Returns

Return type None

pydfnworks.dfnGraph.dfn2graph.**boundary_index**(*bc_name*)
Determines boundary index in intersections_list.dat from name

Parameters **bc_name** (*string*) – Boundary condition name

Returns **bc_index** – integer indexing of cube faces

Return type int

Notes

top = 1 bottom = 2 left = 3 front = 4 right = 5 back = 6

pydfnworks.dfnGraph.dfn2graph.**create_bipartite_graph**(*inflow*, *outflow*, *intersection_list='intersection_list.dat'*, *fracture_info='fracture_info.dat'*)
Creates a bipartite graph of the DFN. Nodes are in two sets, fractures and intersections, with edges connecting them.

Parameters

- **inflow** (*str*) – name of inflow boundary
- **outflow** (*str*) – name of outflow boundary
- **intersection_list** (*str*) – filename of intersections generated from DFN
- **fracture_infor** (*str*) – filename for fracture information

Returns B

Return type NetworkX Graph

Notes

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.**create_fracture_graph**(*inflow*, *outflow*, *topology_file='connectivity.dat'*, *fracture_info='fracture_info.dat'*)

Create a graph based on topology of network. Fractures are represented as nodes and if two fractures intersect there is an edge between them in the graph.

Source and Target node are added to the graph.

Parameters

- **inflow** (*string*) – Name of inflow boundary (connect to source)
- **outflow** (*string*) – Name of outflow boundary (connect to target)
- **topology_file** (*string*) – Name of adjacency matrix file for a DFN default=connectivity.dat
- **fracture_infor** (*str*) – filename for fracture information

Returns **G** – NetworkX Graph where vertices in the graph correspond to fractures and edges indicated two fractures intersect

Return type NetworkX Graph

Notes

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.**create_intersection_graph**(*inflow*, *outflow*, *intersection_file='intersection_list.dat'*, *fracture_info='fracture_info.dat'*)

Create a graph based on topology of network. Edges are represented as nodes and if two intersections are on the same fracture, there is an edge between them in the graph.

Source and Target node are added to the graph.

Parameters

- **inflow** (*string*) – Name of inflow boundary
- **outflow** (*string*) – Name of outflow boundary
- **intersection_file** (*string*) – File containing intersection information File Format: fracture 1, fracture 2, x center, y center, z center, intersection length
- **fracture_infor** (*str*) – filename for fracture information

Returns **G** – Vertices have attributes x,y,z location and length. Edges has attribute length

Return type NetworkX Graph

Notes

Aperture and Perm on edges can be added using add_app and add_perm functions

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**(*G*, *k*, *source*, *target*, *weight*)

Returns 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 **paths** – a list of lists of nodes in the k shortest paths

Return type sets of nodes

Notes

Edge weights must be numerical and non-negative

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

pydfnworks.dfnGraph.dfn2graph.**pull_source_and_target**(*nodes*, *source='s'*, *target='t'*)
Removes source and target from list of nodes, useful for dumping subnetworks to file for remeshing

Parameters

- **nodes** (*list*) – List of nodes in the graph
- **source** (*node*) – Starting node
- **target** (*node*) – Ending node

Returns **nodes** – List of nodes with source and target nodes removed

Return type list

Notes

7.6.2 Graph-Based Flow and Transport

pydfnworks.dfnGraph.graph_flow.**get_laplacian_sparse_mat**(*G*, *nodelist=None*, *weight=None*, *dtype=None*, *format='lil'*)

Get the matrices D, A that make up the Laplacian sparse matrix in desired sparsity format. Used to enforce boundary conditions by modifying rows of L = D - A

Parameters

- **G** (*object*) – NetworkX graph equipped with weight attribute
- **nodelist** (*list*) – list of nodes of G for which laplacian is desired. Default is None in which case, all the nodes
- **weight** (*string*) – For weighted Laplacian, else all weights assumed unity
- **dtype** (*default is None, cooresponds to float*) –
- **format** (*string*) – sparse matrix format, csr, csc, coo, lil_matrix with default being lil

Returns

- **D** (*sparse 2d float array*) – Diagonal part of Laplacian
- **A** (*sparse 2d float array*) – Adjacency matrix of graph

pydfnworks.dfnGraph.graph_flow.**prepare_graph_with_attributes**(*inflow*, *outflow*)

Create a NetworkX graph, prepare it for flow solve by equipping edges with attributes, renumber vertices, and tag vertices which are on inlet or outlet

Parameters

- **inflow** (*string*) – name of file containing list of DFN fractures on inflow boundary
- **outflow** (*string*) – name of file containing list of DFN fractures on outflow boundary

Returns **Gtilde**

Return type NetworkX graph

pydfnworks.dfnGraph.graph_flow.**run_graph_flow**(*self*, *inflow*, *outflow*, *Pin*, *Pout*, *fluid_viscosity=0.00089*)

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_flow.**solve_flow_on_graph**(*Gtilde*, *Pin*, *Pout*, *fluid_viscosity=0.00089*)

Given a NetworkX graph prepared for flow solve, solve for vertex pressures, and equip edges with attributes (Darcy) flux and time of travel

Parameters

- **Gtilde** (*NetworkX graph*) –
- **Pin** (*double*) – Value of pressure (in Pa) at inlet
- **Pout** (*double*) – Value of pressure (in Pa) at outlet
- **fluid_viscosity** (*double*) – optional, in Pa-s, default is for water

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

Return type NetworkX graph

class pydfnworks.dfnGraph.graph_transport.**Particle**

Class for graph particle tracking, instantiated for each particle

*** time**
Total advective time of travel of particle [s]

*** tdrw_time**
Total advection+diffusion time of travel of particle [s]

*** dist**
total distance travelled in advection [m]

*** flag**
True if particle exited system, else False

*** frac_seq**
Dictionary, contains information about fractures through which the particle went

add_frac_data(*frac*, *t*, *t_tdrw*, *L*)
add details of fracture through which particle traversed

Parameters

- **self** (*object*) –
- **frac** (*int*) – index of fracture in graph
- **t** (*double*) – time in seconds
- **t_tdrw** (*double*) – time in seconds
- **L** (*double*) – distance in metres

set_start_time_dist(*t*, *L*)
Set initial value for travel time and distance

Parameters

- **self** (*object*) –
- **t** (*double*) – time in seconds
- **L** (*double*) – distance in metres

track (*Gtilde*, *nbrs_dict*, *frac_porosity*, *tdrw_flag*, *matrix_porosity*, *matrix_diffusivity*)
track a particle from inlet vertex to outlet vertex

Parameters

- **Gtilde** (*NetworkX graph*) – graph obtained from graph_flow
- **nbrs_dict** (*nested dictionary*) – dictionary of downstream neighbors for each vertex

write_file (*partime_file=None*, *frac_id_file=None*)
write particle data to output files, if supplied

Parameters

- **self** (*object*) –
- **partime_file** (*string*) – name of file to which the total travel times and lengths will be written for each particle, default is None
- **frac_id_file** (*string*) – name of file to which detailed information of each particle's travel will be written, default is None

pydfnworks.dfnGraph.graph_transport.**create_neighbor_list** (*Gtilde*)
Create a list of downstream neighbor vertices for every vertex on NetworkX graph obtained after running graph_flow

Parameters **Gtilde** (*NetworkX graph*) – obtained from output of graph_flow

Returns **dict**

Return type nested dictionary.

Notes

dict[n]['child'] is a list of vertices downstream to vertex n dict[n]['prob'] is a list of probabilities for choosing a downstream node for vertex n

pydfnworks.dfnGraph.graph_transport.**dump_particle_info** (*particles*, *partime_file*, *frac_id_file*)
If running graph transport in parallel, this function dumps out all the particle information is a single pass rather then opening and closing the files for every particle

Parameters

- **particles** (*list*) – list of particle objects
- **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

Returns **pfailcount** – Number of particles that do not exit the domain

Return type int

pydfnworks.dfnGraph.graph_transport.**prepare_output_files** (*partime_file*, *frac_id_file*)
opens the output files partime_file and frac_id_file and writes the header for each

Parameters

- **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

Returns

Return type None

pydfnworks.dfnGraph.graph_transport.**run_graph_transport**(*self*, *Gtilde*, *nparticles*, *partime_file*, *frac_id_file*, *frac_porosity=1.0*, *tdrw_flag=False*, *matrix_porosity=0.02*, *matrix_diffusivity=1e-11*)

Run particle tracking on the given NetworkX graph

Parameters

- **self** (*object*) – DFN Class
- **Gtilde** (*NetworkX graph*) – obtained from graph_flow
- **nparticles** (*int*) – number of particles
- **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
- **frac_porosity** (*float*) – porosity of fracture, default is 1.0
- **tdrw_flag** (*Bool*) – if False, matrix_porosity, matrix_diffusivity are ignored
- **matrix_porosity** (*float*) – default is 0.02
- **matrix_diffusivity** (*float*) – default is 1e-11 in SI units

Notes

Information on individual functions is found therein

pydfnworks.dfnGraph.graph_transport.**track_particle**(*data*)

Tracks a single particle through the graph

all input parameters are in the dictionary named data

Parameters

- **Gtilde** (*NetworkX graph*) – obtained from graph_flow
- **nbrs_dict** (*dict*) – see function create_neighbor_list
- **frac_porosity** (*float*) – porosity of fracture, default is 1.0
- **tdrw_flag** (*Bool*) – if False, matrix_porosity, matrix_diffusivity are ignored
- **matrix_porosity** (*float*) – default is 0.02
- **matrix_diffusivity** (*float*) – default is 1e-11 m^2/s

Returns **particle** – particle trajectory information

Return type object

7.7 General Workflow functions

7.7.1 Print legal statement

pydfnworks.general.legal.**legal**(*self*)

Print the legal LANL statement for dfnWorks.

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

Returns

Return type None

Notes

None

7.7.2 Helper functions

pydfnworks.general.general_functions.**dump_time**(*self*, *function_name*, *time*)

Write run time for a funcion to the jobname_run_time.txt file

Parameters

- **self** (*object*) – DFN Class
- **function_name** (*string*) – Name of function that was timed
- **time** (*float*) – Run time of function in seconds

Returns

Return type None

Notes

While this function is working, the current formulation is not robust through the entire workflow

pydfnworks.general.general_functions.**print_run_time**(*self*)

Read in run times from file and and print to screen with percentages

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

Returns

Return type None

Notes

This will dump out all values in the run file, not just those from the most recent run

7.7.3 Set up run paths

pydfnworks.general.paths.**compile_dfn_exe**(*directory*)

Compile executables used in the DFN workflow including: DFNGen, DFNTrans, correct_uge, correct_stor, mesh_checking. The executables LaGriT, PFLOTRAN, and FEHM are not compiled in this function

Parameters **directory** (*string*) – Path to dfnWorks executable

Returns

Return type None

Notes

This function is only called if an executable is not found.

pydfnworks.general.paths.**define_paths**()

Defines enviromental variables for use in dfnWorks. The user must change these to match their workspace.

Parameters **None** –

Returns

Return type None

Notes

Enviromental variables are set to executables

pydfnworks.general.paths.**valid**(*name*)

" Check that path is valid for a executable

Parameters **name** (*string*) – Path to file or executable

Returns

Return type None

Notes

If file is not found, program exits

CHAPTER

EIGHT

DFNGEN

8.1 Keywords

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

```
/*=========================================================================*/
/*  Gereral Options & Fracture Network Parameters:                         */
/*=========================================================================*/

stopCondition: 0
/*  0: Stop once nPoly fractures are accepted (Defined below)
    1: Stop once all family's p32 values are equal or greater than the
       families target p32 values (defined in stochastic family sections)
*/

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

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

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

numOfLayers: 2
//  Number of layers

layers:
{-25,0}
{0,25}
/*  Layers need to be listed line by line
    Format: {minZ, maxZ}

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

h: .1
/*  Minimum fracture length scale(meters).
    Any fracture with a feature, such as and intersection,
```

(continues on next page)

(continued from previous page)

```
    of less than h will be rejected.
*/


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

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

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



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

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

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



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

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

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


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

(continues on next page)

(continued from previous page)

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



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

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


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

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

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

*/

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

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

(continues on next page)

(continued from previous page)

```
domainSizeIncrease: {0,0,0}
/*  Size increase for inserting fracture centers outside the domain.
    Fracture will be truncated based on domainSize above.
    Increases the entire width by this ammount. So, {1,1,1} will increase
    the domain by adding .5 to the +x, and subbtracting .5 to the -x, etc
*/

keepOnlyLargestCluster: 0
/*  0 - Keep any clusters which connects the specified
        boundary faces in boundaryFaces option below
    1 - Keep only the largest cluster which connects
        the specified boundary faces in boundaryFaces option below.

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

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

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

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

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

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

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


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

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

(continues on next page)

(continued from previous page)

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

    The famProb elements should add up to 1.0 (for %100).
    The probabilities are listed in order of families starting with all
    stochastic ellipses, and then all stochastic rectangles.

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



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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
    intensity has been met, fracture generation will complete.
*/


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

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

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

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

etheta: {0, 45}
/*  Ellipse fracture orientation.
    The angle the normal vector makes with the z-axis,
*/

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

ebeta: {0, 0}
//  Rotation around the fractures' normal vector.

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


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

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

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

eLogMin: {1}
```

(continues on next page)

(continued from previous page)

```
//  Minimum radius

eLogMax: {15}
//  Maximum radius


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

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

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

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


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

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


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

emin: {1}
//  Minimum radius

emax: {6}
//  Maximum radius

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


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

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

(continues on next page)

(continued from previous page)

```
*/

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

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

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

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


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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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


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

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

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

rLogMin: {5,5}
//  Minimum radius

rLogMax: {20,20}
//  Maximum radius


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

rmin: {1}
//  Minimum radius

rmax: {10}
//  Maximum radius

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


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

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

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

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

(continues on next page)

(continued from previous page)

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


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

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

UserEll_Input_File_Path: /home/jharrod/GitProjects/DFNGen/DFNC++Version/inputFiles/
↪userPolygons/uEllInput.dat


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

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

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


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

userRectanglesOnOff: 0
```

(continues on next page)

(continued from previous page)

```
/*  0 - User Rectangles off
    1 - User Rectangles on
*/

UserRect_Input_File_Path: /home/jharrod/GitProjects/DFNGen/DFNC++Version/inputFiles/
↪userPolygons/ignoreConnTest.dat


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

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

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

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


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

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


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

meanAperture:  3
//  Mean value
```

(continues on next page)

(continued from previous page)

```
stdAperture: 0.8
//  Standard deviation


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

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

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

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

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

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

/*============================================================================*/
/*============================================================================*/
/*  Permeability Options                                                      */
/*  Mandaatory Parameter                                                      */
/*============================================================================*/
/*============================================================================*/

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

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

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

CHAPTER

NINE

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

#=========================== material properties ==============================
MATERIAL_PROPERTY soil1
```

(continues on next page)

(continued from previous page)

```
  ID 1
  POROSITY 0.25d0
  TORTUOSITY 0.5d0
  CHARACTERISTIC_CURVES default
  PERMEABILITY
    DATASET Permeability
  /
END


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

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

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

# REFERENCE_PRESSURE 1500000.

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

REGION inflow
```

(continues on next page)

(continued from previous page)

```
  FILE pboundary_left_w.ex
END

REGION outflow
  FILE pboundary_right_e.ex
END

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


FLOW_CONDITION outflow
  TYPE
     PRESSURE dirichlet
  /
  PRESSURE 1.d6
END

FLOW_CONDITION inflow
  TYPE
    PRESSURE dirichlet
  /
  PRESSURE 2.d6
END

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


BOUNDARY_CONDITION INFLOW
  FLOW_CONDITION inflow
  REGION inflow
END

BOUNDARY_CONDITION OUTFLOW
  FLOW_CONDITION outflow
  REGION outflow
END

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

END_SUBSURFACE
```

CHAPTER

TEN

DFNTRANS

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

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

All source files of C code of dfnTrans are in `DFNTrans/` directory of dfnWorks 2.2. It compiles under linux/mac machines using `makefile`. In order to run transport, first, all the parameters and paths should be set up in the PTDFN Control file, PTDFN_control.dat. Then, the following command should be run:

```
./DFNTrans PTDFN_control.dat
```

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

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

/*********************  INPUT FILES: grid ****************************/
/**** input files with grid of DFN, mainly it's output of DFNGen ******/
param: params.txt
poly: poly_info.dat
inp: full_mesh.inp
stor: full_mesh.stor
boundary: allboundaries.zone
/* boundary conditions: reading the nodes that belong to in-flow and
out-flow boundaries. Should be consistent with those applied to obtain
steady state pressure solution (PFLOTRAN)   */
/*1 - top;  2 - bottom;  3 - left_w;  4 - front_s;  5 - right_e;  6 - back_n */
```

(continues on next page)

(continued from previous page)

```
in-flow-boundary: 3
out-flow-boundary: 5


/**************** INPUT FILES: PFLOTRAN flow solution ******************/
PFLOTRAN: yes
PFLOTRAN_vel: darcyvel.dat
PFLOTRAN_cell: cellinfo.dat
PFLOTRAN_uge: full_mesh_vol_area.uge

/**************** INPUT FILES: FEHM flow solution ***********************/
/*currently we are using PFLOTRAN , but the code would work with FEHM, too */
FEHM: no
FEHM_fin: dfn.fin

/************************  OUTPUT FILES  ******************************/
/* initial grid info structure output, usefull for debugging */
out_grid: no
/* flow field: 3D Darcy velocities: output file has an each nodes position
and its Darcy velocity, reconstructed from fluxes */
out_3dflow: no
/* out initial positions of particles into separate file */
out_init: no
/* out particle trajectories tortuosity file, torts.dat */
out_tort: no

/*************** output options for particles trajectories ***************/
/* output frequency is set according to trajectories curvature. We check the
curvature of particles trajectory each segment, from intersection to intersection.
If it's like a straight line, then the output is less frequent (in case of
"out_curv:yes", if "no",  the output file will contain every time step) */
out_curv: yes
/* output into avs file (GMV visualization, Paraview visualization) */
out_avs: no
/* output into trajectories ascii files (veloc+posit+cell+fract+time) */
out_traj: yes

/* temporary outputs (every time step from intersection to intersection)*/
/* use outputs to file or memory buffer. Memory buffer by default */
out_filetemp: no

/************* output directories ***************************************/
out_dir: traj_SR /* path and name of directory where all the particle
                    tracking results will be written*/


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

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



/*************** PARTICLES INITIAL POSITIONS ***************************/
```

(continues on next page)

(continued from previous page)

```
/*****  particles positions according to in-flow flux weight *********/
init_fluxw: no //turn on this input option (don't forget to turn off rest of␣
↪PARTICLES INITIAL POSITIONS options)
init_totalnumber: 10000 // distance [m] between particles at inflow face for equal␣
↪flux weight calculation


/****init_nf: if yes - the same number of particles (init_partn) will be placed
     on every boundary fracture edge on in-flow boundary,
     equidistant from each other ****/
init_nf: yes
init_partn: 10

/****init_eqd: if yes - particles will be placed on the same distance from
     each other on all over in-flow boundary edges **********************/
init_eqd: no  //maximum number of particles that user expects on one boundary edge
init_npart: 100

/*** all particles start from the same region at in-flow boundary, in a range
    {in_xmin, in_xmax,in_ymin, in_ymax, in_zmin, in_zmax} **************/
init_oneregion: no
in_partn: 100000
in_xmin: -50.0
in_xmax: -50.0
in_ymin: -20.0
in_ymax:  20.0
in_zmin: -15.0
in_zmax:  0.0

/**** all particles are placed randomly over all fracture surface
     (not only on boundary edges!) ***********************************/
init_random: no
// total number of particles
in_randpart: 100

/**** all particles are seed randomly over matrix,
     they will start travel in DFN from the node/cell that is closest to
     their initial position in rock matrix ***************************/

init_matrix: no
// to obtain these files, run python script RandomPositGener.py
inm_coord: ParticleInitCoordR.dat
inm_nodeID: ClosestNodeR.inp
inm_porosity: 0.02
inm_diffcoeff: 1.0e-12

/*************** Intersection Mixing Rule *********************************/
/****streamline_routing: if yes - streamline routing is the selected subgrid process
     otherwise the complete mixing rule is selected ****/
streamline_routing: no


/************* TIME DOMAIN RANDOM WALK ****************************/
tdrw: no
tdrw_porosity: 0.02
tdrw_diffcoeff: 1.0e-11
```

(continues on next page)

(continued from previous page)

```
/****************** FLOW AND FRACTURE PARAMETERS ********************/
porosity: 1.0 // porosity
density: 997.73  //fluid density
satur: 1.0
thickness: 1.0 //DFN aperture  (used in case of no aperture file provided)

/************************ APERTURE ********************************/

aperture: yes  //DFN aperture
aperture_type: frac //aperture is giving per cell (type "cell")
//    or per fracture (type "frac")
// for now we use an aperture giving per fracture
aperture_file: aperture.dat


/********************  TIME ***********************************************/
timesteps: 2000000
//units of time (years, days, hours, minutes)
time_units: seconds

/**** flux weighted particles*/
/**** in case of random initial positions of particles - it's aperture weighted **/
flux_weight: yes
/* random generator seed */
seed: 337799

/*********************  Control Plane/Cylinder Output ********************/
/*** virtual Control planes will be build in the direction of flow.
Once particle crosses the control plane, it's position, velocity, time
will output to an ascii file. ****/
ControlPlane: yes

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

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

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


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

CHAPTER

ELEVEN

SCRIPTS

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

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

The tests that come with dfnWorks depend on pathnames which vary depending on where dfnWorks is installed. To fix these paths automatically, run the ''fix_paths.py" script accordingly:

```
python fix_paths.py
```

CHAPTER

TWELVE

OUTPUT FILES

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

12.1 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.

12.2 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.

12.3 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.

12.4 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

INDEX

D

E

F

N

O

P

R

S

T

U

V

W

Z