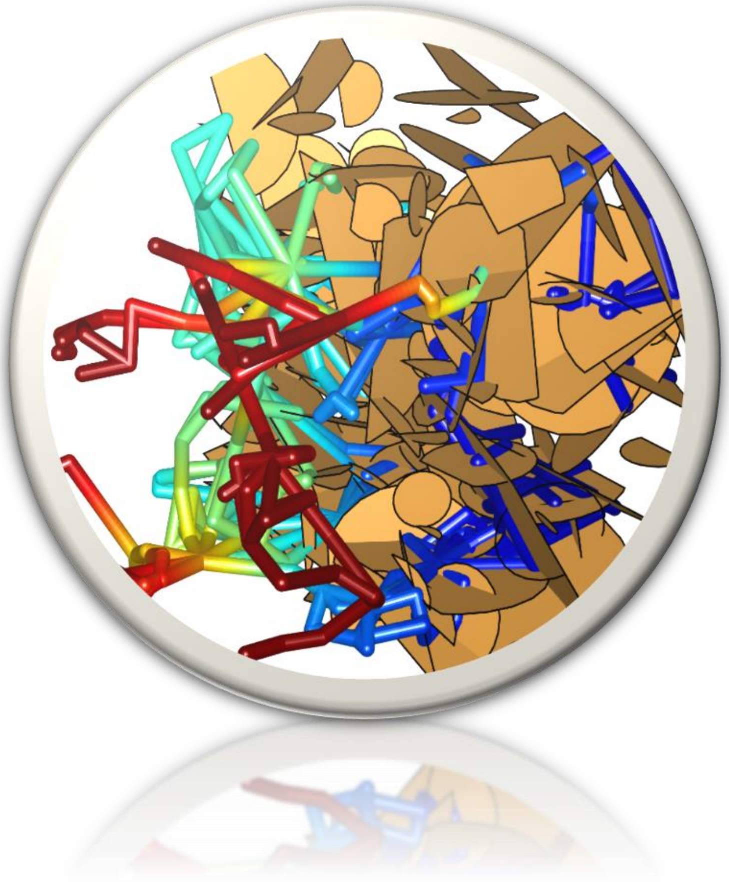
@lghalandis omputing R1.5

DFNE

Practices with ADFNE



Dr. Younes Fadakar Alghalandis
Alghalandis Computing @

DFNE Practices with ADFNE

Dr. Younes Fadakar Alghalandis

© 2018 Alghalandis Computing

This book serves as the comprehensive manual for the ADFNE1.5, an Open Source Software Package for DFNE, and as a practical guide on fundamentals of DFNE including simulations, characterizations and applications. It includes the complete list of all functions in ADFNE1.5, their formal syntax, example usages, and descriptions. In addition, several reproducible tutorials and case studies (with full program codes) are demonstrated. The official webpage of ADFNE1.5 and this book is: http://alghalandis.net/products/adfne/adfne15

Title: DFNE Practices with ADFNE

Author: Dr. Younes Fadakar Alghalandis

Publisher: Alghalandis Computing @

Date: Jan 2018 (First Edition)

Location: Toronto, ON Canada

Copyright: 2018 Alghalandis Computing @

All rights reserved.

Preface

DFNE (Discrete Fracture Network Engineering) is an exciting, fast growing and very influential engineering and science topic. The range of disciplines actively contributing to its development is massive including mining, geosciences, oil & gas, geothermal, water resources, nuclear, construction and many other industries. Wherever there is a problem associated with fractured rock masses the DFNE is the ultimate, comprehensive and modern solution.

This book, fair to say, is a product of years of research activities of mine in DFNE. Indeed, it is a product of passion and a mission to produce quality software and to release them to public as open source in order to assist researchers and practitioners around the world in academia and industries for completely transparent and trustworthy computations in DFNE.

ADFNE (R1.0) was first released in early 2017 together with my original paper published in Computers & Geosciences. Its root however goes back to my AFNM package published as an appendix to my PhD thesis in 2014. Now, I release the ADFNE1.5, a significant edition that offers several key developments in the software including in its structure, functionality, generic coding concept, performance, validation and documentation. The quality software engineering concepts implemented in ADFNE1.5 goes far beyond DFNE and would hence be of even greater use as real code tutorials for computations in *Matlab and numerical modeling.

Notice

I do not disclose the source code of the two functions 'Mesh' and 'Tensor' in this release of ADFNE1.5, because, I have papers based on them under review. I will however release an update likely mid to late 2018.

Dr. Younes Fadakar Alghalandis

Toronto, ON, Canada – Jan 2018

*Matlab is TM and product of Mathworks.

Contents

Introduction

My journey on DFNE started in 2011 at the University of Adelaide, Australia where I pursued my PhD study (with Prof. Peter Dowd and Prof. Chaoshui Xu) on a subject that was fairly brand new to me, the Fracture Networks Modeling. I remember that for the first months I did a lot of computer coding in order to familiarize myself with the concepts and terms; having multiple books always sitting on my desk for quick and or deep reads and reference. A thick book on point processes! Hmm… It took a while until I finally became relaxed (a little bit) with the terminology; it was the time that I started to explore more and more with enough confidence. Indeed, my computer-coding skills helped me massively to learn stuff rapidly by doing and exploring the results approach (quick and reliable, indeed), which is now a standard way of mine to tackle any research problem. I used to store all codes and progresses in folders organized by dates. A plenty of folders though, but very conservative and helpful commitment, in practice. Motivations to publish a conference paper just only 4 months after starting PhD, and later on as a trend of my entire PhD study, helped me to publish couple of noticeable conference and journal papers in DFNE, some deep theories and applications. The RANSAC method for fitting fractures to point clouds of micro-seismic events, the Connectivity Field to address long-challenged connectivity problem of fracture networks and many others all were triggered during my practices with codes, indeed. Later, I decided to prepare an appendix to my PhD thesis including program codes for fracture network modeling in Matlab language. That was named AFNM. More years of practices, working as post-doctoral research fellow at the University of British Columbia (with Prof. Erik Eberhardt and Prof. Davide Elmo), and Simon Fraser University (with Prof. Doug Stead) demonstrated more need and use to my Matlab package. By the end of 2016, I decided to introduce ADFNE1.0 to the community by publishing a pioneering paper in the Journal of Computers & Geosciences. Simultaneously, I was working hard to release the ADFNE1.0 package in a quality releasable to public. No claim of perfection or even any close, but I am still very proud of that accomplishment. My recent short visit to University of Toronto (with Prof. Giovanni Grasselli) also emphasized that the DFNE community (in a broad sense) is in need of a more polished package. Many motivated people including professors, post-doctoral research fellows, PhD and graduate students as well as industry experts from all continents, have contacted me through the years to share their feedback, ideas and even expectations. I have also been contacted numerously in regards to dedicated support on ADFNE package, doing further developments and industry / research customized solutions. All these ultimately motivated me to prepare ADFNE1.5, a truly significant release, for which I conducted months of timeless developments. I hope that my work excites, inspires and motivates people for similar solutions ultimately for the benefits of the communities, DFNE and beyond. Here, I sincerely welcome you to go through this document and to practice with ADFNE1.5.

Discrete Fracture Network Engineering

DFNE, in a nutshell, includes three main stages: simulations (generating), characterizations and applications. A DFNE simulation starts with defining fractures one by one, and then distributing them as a network spatially featured in the space. If the model is 2D, fractures are simply straight line-segments (hereafter lines). If the case was 3D, then fractures are convex polygons (hereafter polygons). Even though these two setups are very simplified but they are practically capable of modeling any sort of complexity in the shape of fractures. Hence, they are standards. For implementation of any curve or surface, it can be divided into straight lines and convex polygons, respectively. Functions such as Exponential distribution are used to define the length (size) of fractures. Having the fracture shapes (geometry) ready, every fracture is distributed into the space by translation and rotation. The translation may follow Uniform, Poisson or any other random distribution functions. Spatial correlations (e.g., Geostatistics: Variogram and Kriging) may also be used to enforce location-specific density variations. The rotation determines directional characteristics of fractures, and often follows Fisher distribution. Additional characteristics for every fracture or whole DFN model such as aperture, surface roughness, geomechanical properties, flow properties, and so on are incorporated depending on the purpose of study. Special characteristics such as connectivity are defined for the DFN model not individual fractures, and often is implemented in a try-and-test loop as some of these are the "response" of the DFN model, indeed.

The diversity of applications of DFNE is huge, and so is linked communities. The best to master DFNE is reading and more reading. A shortcut is "to practice". ADFNE1.5 provides a comprehensive and generic collection of tools (functions, classes and scripts), to help one to speed up the learning progress and to deepen the understanding. It is open source, free and under very easy license, to facilitate one's development of: programming knowledge, the sense of DFNE concepts, novel functionalities, new ideas, new solutions for new problems, and to energize the DFNE and even broader communities for transparent, free and easy sharing of works and achievements.

ADFNE 1.5

In this chapter, I list all available functions in ADFNE1.5 as of 2018-01-11. Every function, class or script in ADFNE1.5 accompanies internal documentation in three forms: "introduction" which appears at the beginning of the file (right after the declaration of function or class name, for example), "comments" which usually appear at the end of lines, and "additional" which, if exists, appears at the end of files. The "introduction" section consists of the name of the function, its purpose, formal usage syntax, input and output arguments, examples, copyright, authorship, rights, references and links, and date. The "comments" sections add further information, notes or any other descriptions to the lines. The "additional", if exists, adds further information relevant to the function, class or script.

Angle

Syntax:

```
ags = Angle(ags,lim);
```

Example(s):

```
ags = Angle([0,90,135,180,225,270,355,-10,-95,-180,-230,-365,723],360);
```

The 'Angle' function converts the given angles (in degree) to one of the four formats determined by the second argument 'lim', if 0, the results are azimuth angles; 90, the results are dip angles i.e., [0..90]; 180, the results are projections onto [0..180), and finally if 360, the results are projections onto [0..360). Any value other than the above is interpreted as 180. This function is especially useful while dealing with dip and dip-direction values.

Axes

Syntax:

```
Axes(varargin);
```

Example(s):

```
Axes('view','top');                          % sets 2d view
Axes(0,'p',5);                               % expands 2d axes by 5%
Axes('box',[0,0,0,2,2,2],'view',[30,30]);    % sets bounding box and view in 3d
```

The 'Axes' function conveniently sets axes' parameters for 2D and 3D visualizations. As shown in the above examples (also applicable to all functions in ADFNE1.5), the arguments are defined by names and their corresponding values in an order. For this function, a name of argument can be 'box', 'p', 'view' and or 'light'. The 'box' defines the boundary box to be drawn if there was a bounding box vector e.g., [0,0,1,1] or [0,0,0,1,1,1]. The 'p' is used to expand the axes for 2D drawing by a percentage of its value. The 'view' is used to set the viewing angles in 3D. The 'light' is of size (nx3) to define positions for lights in visualization. Function 'Axes' also provides a unique synchronization between rotation and camera light. While in effect, any change in the angle of view will automatically adjust the light such to avoid shadowed (dimed) views.

Backbone

Syntax:

```
dfn = Backbone(dfn,tol);
```

Example(s):

```
dfn = Backbone(Pipe(be,dfn,'cnt'));          % see Pipe function for details
```

The 'Backbone' function extracts efficiently the backbone (skeleton) structure from the given pipe model. Note that the governing coding convention in ADFNE1.5 is such to unify everything as much as possible i.e., fewer functions but more capabilities. This convention also applies to the arguments. Indeed, for the modeling of fluid flow through fractures in ADFNE1.5 a variety of stages (including the functions: Pipe, Backbone, Graph and Solve) work together seamlessly to achieve the best consistency in between. Hence, as in its syntax, the 'Backbone' function receives the argument 'dfn' right from the 'Pipe' function (page 27), and being loyal it updates the 'dfn' with the resulting backbone structure if any. The 'Backbone' function works smartly with 2D and 3D pipes. The 'tol' value is used to determine the tolerance for isolation test between pipes. If the 'tol' argument was missing the default 'Tolerance' declared in 'Globals' (page 40) is used.

Bbox

Syntax:

```
out = Bbox(in,varargin);
```

Example(s):

```
out = Bbox(rand(10,4));                     % bounding box for all 2d lines
out = Bbox(rand(10,4),'pp');               % bounding box as point-to-point format
out = Bbox(rand(10,6));                     % bounding box for all 3d lines
out = Bbox(rand(10,4),'all');               % bounding boxes for every 2d line
out = Bbox(rand(10,6),'all');               % bounding boxes for every 3d line
out = Bbox(rand(10,4),'intersect');    % indices of intersecting bbox for 2d lines
out = Bbox({rand(3,3)});                    % bounding box for single 3d polygon
out = Bbox({rand(3,3);rand(3,3)});          % bounding box for 3d polygons
out = Bbox(plys,'intersect');      % indices of intersecting bbox for 3d polygons
```

The 'Bbox' function is a super function. It smartly recognizes variety of input arguments and acts correspondingly. It returns the bounding box for lines and polygons in 2D and 3D. If the keyword 'all' was in the arguments, then the bounding box for every individual entity of the input is returned, e.g., for every polygon. If the keyword 'intersect' was in the arguments, then 'Bbox' function acts in two stages. It first finds the bounding boxes, and then applies intersection analysis to the found bounding boxes. Any resulting intersection indices are then returned.

Boundary

Syntax:

```
vls = Boundary(pts,be,bv,p);
```

Example(s):

```
pts = rand(30,2);                           % points
be = [0,0,0,1;1,0,1,1];                     % boundary elements i.e., lines
bv = [1,0];                                 % boundary values
vls = Boundary(pts,be,bv,2);
```

The ‘Boundary’ function interpolates boundary values given by ‘bv’ for boundary elements ‘be’ at the points ‘pts’. The exponent ‘p’ affects the interpolation via distance weighting factor. This function is extremely versatile (i.e., works with any number of boundary elements) and useful especially for solving the fluid flow through fractures, accounting for various setups for boundary conditions.

Clip

Syntax:

```
[y,b] = Clip(x,bbx);
```

Example(s):

```
[y,b] = Clip(rand(10,4),[0,0,1,1]);         % clipping 2d lines
[y,b] = Clip(plys,[0,0,0,1,1,1]);           % clipping 3d polygons
```

The ‘Clip’ function clips 2D lines or 3D polygons by the given bounding box (2D or 3D). A bounding box is defined in point-to-point format (mins to maxs) i.e., like a line. In ADFNE1.5 2D lines are of size nx4, while 3D polygons are of cell type. A polygon cell can have any number of edges independent one to another. Polygons are convex.

Cluster

Syntax:

```
c = Cluster(s);
```

Example(s):

```
c = Cluster([1,2;3,5;2,9]);                 % = {[1,2,9],[3,5]}
```

The ‘Cluster’ function groups the given data into clusters based on any common elements. This function is used to determine the fracture clusters following the intersection analysis.

Contour

Syntax:

```
Contour(pts,vls,k,filled,mtd,varargin);
```

Example(s):

```
Contour(rand(10,2),rand(10,1),20,true,'idw');
Contour(rand(10,2),rand(10,1),20,true,'idw','linewidth',2,'linestyle',':');
```

The ‘Contour’ function provides a very convenient interface for creating contour maps based on any scattered point data as input. In addition, this function can use either Matlab’s ‘griddata’ interpolation or ADFNE1.5 own defined ‘IDW’ interpolation methods for contouring.

Convert

Syntax:

```
varargout = Convert(to,varargin);
```

Example(s):

```
% ------- strcut to cell
s.a = 1; s.b = '2';
c = Convert('cell',s);                      % = {'a',[1],'b','2'}
c = Convert('cell',[7,8]);                  % = {[1,2]}
% ------- seconds to clock
clk = Convert('clock',1000);                % = '00:16:40.00'
% ------- vector to colors
cls = Convert('color',rand(1,10));          % returns 10 colors
cls = Convert('color',rand(1,10),@hot,128); % ...from hot colormap(128)
```

```
% ------- radians to degrees
deg = Convert('deg',pi);                  % = 180
% ------- degrees to radians
rad = Convert('rad',180);                 % = pi
% ------- any to normalized
y = Convert('norm',rand(1,3));            % normalized (3)
% ------- polygons to triangles
trs = Convert('tri',[0,0;1,0;1,1;0,1]);   % = {4 triangles}
% ------- varargin to struct
s = Convert('struct','a',1,'b',34);       % = {a:1, b:34}
% ------- 3d points to 2d points
pts = Convert('2d',rand(10,3));           % size(pts) == [10,2]
% ------- 2d points to 3d points
pts = Convert('3d',rand(10,2));           % size(pts) == [10,3]
```

The 'Convert' is a super function that converts many things to many other things. The type of conversion is defined by the first argument including 'cell', 'clock', 'color', 'deg', 'rad', 'norm', 'tri', 'struct', '2d' and '3d'. See the examples above for information.

DFN

Syntax:

```
out = DFN(varargin);
```

Example(s):

```
fnm = DFN;                                % 2d fnm
fnm = DFN('dim',2,'n',300,'dir',45,'ddir',0,'minl',0.1,'mu',0.2,'maxl',0.3); % 2d
lns = [Field(DFN('dim',2,'n',30,'dir',45,'ddir',-100,'minl',0.1,'mu',0.2,...
    'maxl',0.3),'Line'); Field(DFN('dim',2,'n',30,'dir',135,'ddir',-100,'minl',...
    0.1,'mu',0.2,'maxl',0.3),'Line')];      % combined complex 2d fnm
ply = Field(DFN('dim',3),'Poly');         % 3d fnm
ply = Field(DFN('dim',3,'dip',45,'ddip',0,'dir',180,'ddir',0),'Poly'); % 3d fnm
```

The 'DFN' function generates 2D and 3D fracture networks. In the first example above, a 2D fracture network is simulated entirely based on the default settings. The second example shows how the arguments can be defined. In the third example, a combined complex fracture network is simulated. This example demonstrates the amazingly simple concatenation of multiple fracture sets (each with own setup) into one set. The fourth example creates a 3D fracture network entirely based on the default values. Any custom setups can be used as shown in the last example. The efficient concatenation of multiple sets of fractures works the same easy way for both 2D and 3D fracture network, empowering the construction of any complex or scenario-based dfn models.

Display

Syntax:

```
Display(varargin);
```

Example(s):

```
Display('a',1,'b',723);
```

The 'Display' function provides a convenient way to display information stored as cell of names and values.

Distance

Syntax:

```
d = Distance(pts,in);
```

Example(s):

```
d = Distance(rand(10,2),rand(3,4));         % 2d points and lines
d = Distance(rand(10,3),rand(3,6));         % 3d points and lines
d = Distance(rand(10,3),{rand(3,3);rand(3,3)}); % 3d points and polygons
```

The 'Distance' is a super function that finds the distances between points and lines or polygons. It works efficiently in both 2D and 3D cases; all made easy. This function is a very useful tool to define zoning area / volume around fractures, for example.

Draw

Syntax:

```
ax = Draw(dfn,varargin);
```

Example(s):

```
Draw(rand(10,4));                           % 2d lines
Draw(rand(10,6));                           % 3d lines
Draw('cub',[0,0,0,1,1,1]);                  % cube
Draw('sph',[0,0,0]);                        % sphere
Draw('cyl',[0,0,0;1,1,1],'cix',[1,64],'ec','k'); % cylinder
Draw('lin',rand(10,4));                     % 2d lines
Draw('lin',rand(10,6));                     % 3d lines
Draw('ply',rand(3,2));                      % 2d polygons
Draw('ply',rand(3,3));                      % 3d polygons
Draw(dfn,'what','fsen');                    % fnm, solution (edges,nodes)
Draw(dfn,'what','fpxiqseng');               % all possible drawing
```

The ‘Draw’ is a super function that does many various drawings in perfection. The examples above are self-explanatory. In the two last, if the first argument is a dfn, then the keyword ‘what’ determines what to be drawn. In 'fpxiqseng' used above, ‘f’ refers to the fractures, ‘p’ to the pipes, ‘x’ to the intersections, ‘i’ to a filter to mask but contributing fractures to the pipes, ‘q’ to a filter to mask but contributing fractures to the flow, ‘s’ to the solution (pressure), ‘e’ to the edges, ‘n’ to the nodes, and ‘g’ to the graph. Any sensible combination of these characters can be used to achieve desired combined visualization. The ‘Draw” function is smart enough to auto-adjust any overlapping entities.

Export

Syntax:

```
Export(in,fname,varargin);
```

Example(s):

```
Export(rand(10,4),'lns.htm');               % 2d htm
Export(rand(10,4),'lns.svg');               % 2d svg
Export(Field(DFN('dim',3),'Poly'),'ply.htm'); % 3d WebGL
Export(Field(DFN('dim',3),'Poly'),'ply.vtk'); % 3d vtk
Export(Field(DFN('dim',3),'Poly'),'ply.xyz'); % 3d text
Export(Field(DFN('dim',3),'Poly'),'ply.txt'); % 3d text
Export(Field(DFN('dim',3),'Poly'),'ply.geo'); % 3d geo
```

The ‘Export’ is a super function that exports 2D lines or 3D polygons to various standard file formats including ‘vtk’, ‘txt’, ‘xyz’, ‘geo’ (GMSH’s format), ‘htm’, ‘html’ (WebGL ready), ’svg’ (vector format) and ‘inp’ (Abaqus’ format). It can also export the workspace in ‘mat’ format accompanied with a snapshot of any current figure, as well as exporting the current figure in high quality ‘png’ (image) or ‘pdf’ (vector) formats.

Field

Syntax:

```
field = Field(in,field);
```

Example(s):

```
lns = Field(DFN,'Line');
```

The ‘Field’ function selects the requested field ‘field’ from the first argument ‘in’. It is very useful while working with structured format variable such as ‘dfn’.

Filter

Syntax:

```
b = Filter(ids,k);
```

Example(s):

```
b = Filter(dfn.dfn.xID,2);
```

The ‘Filter’ function masks out indices ‘ids’ based on the cardinality compared to ‘k’. See ‘Draw’ function’s source code for a working example, where this function helps to match fractures with select pipes.

Flatri

Syntax:

```
y = Flatri(x,k);
```

Example(s):

```
y = Flatri(Reshape(1:16,[],4))';            % = [5,9,13,10,14,15]'
```

The ‘Flatri’ function returns the lower triangle of the matrix ‘x’ in the flattened format, i.e., as a vector.

Geometry

Syntax:

```
out = Geometry(wht,varargin);
```

Example(s):

```
elp = Geometry('elp','cn',[0,0],'q',12,'r',[1,0.5],'ang',45);  % an ellipse
```

The ‘Geometry’ function creates geometries such as ellipse (‘elp’ or ‘ellipse’), ellipsoid (‘els’ or ‘ellipsoid’) and sphere (‘sph’ or ‘sphere’). It returns the point coordinates.

Graph

Syntax:

```
dfn = Graph(dfn,aFun,dFun);
```

Example(s):

```
dfn = Graph(dfn,@(x)0.0001);
```

The ‘Graph’ function constructs the graph structure (nodes and edges) based on the given ‘dfn’ with all necessary information such as the neighboring. It also accepts two functions as optional arguments, the first to apply to aperture field, and the second to apply to depth field of fractures. These two functions can be defined anonymously as shown in the above example. In the example above, the apertures are all set constant to value of 0.0001m. The argument ‘x’ in the anonymous function is fed inside the Graph function by the length of fractures; hence, by using a function such as `@(x)0.0001*x`, the aperture variation becomes relative to the length of every individual fracture (i.e., the scenario of correlated aperture and length is achieved).

Grid

Syntax:

```
lns = Grid(n,varargin);
```

Example(s):

```
lns = Grid(7,'ang',45);
```

The ‘Grid’ function generates grid lines, horizontal and vertical, in 2D. The resulting grid can also be rotated if the argument ‘ang’ is passed to the function.

Group

Syntax:

```
[gxs,gds] = Group(xts,ids,n);
```

Example(s):

```
[gxs,gds] = Group(xts,ids,n);
```

The ‘Group’ function groups the intersection points (lines) and indices resulted from the ‘Intersect’ function.

IDW

Syntax:

```
pvs = IDW(rts,rvs,pts);
```

Example(s):

```
pvs = IDW(rand(3,2),rand(3,1),rand(10,2));
```

The ‘IDW’ function interpolates values for points (‘pts’, 2D or 3D) based on the inverse distance weighting system applying to the reference points ‘rts’ and the corresponding values ‘rvs’. This function is generic and very efficient, works with even higher dimensions.

Info

Syntax:

```
info = Info(in);
```

Example(s):

```
info = Info(Field(DFN,'Line'));
info = Info(Field(DFN('dim',3),'Poly'));
```

The ‘Info’ function extracts the location, size and orientation characteristics from the given fracture network, 2D lines or 3D polygons.

Inpoly

Syntax:

```
o = Inpoly(pts,ply,tol);
```

Example(s):

```
o = Inpoly(rand(100,2),rand(3,2));          % 2d case
o = Inpoly(rand(100,3),rand(3,3),0.2);      % 3d case
```

The ‘Inpoly’ function tests the points ‘pts’ for being inside or outside of the polygon ‘ply’. This function works in 2D and 3D cases, as exemplified above. The third argument defines the tolerance of distance. Any point closer than ‘d’ to the polygon is considered inside.

Intersect

Syntax:

```
[xts,ids,La] = Intersect(in);
```

Example(s):

```
[xts,ids,La] = Intersect(Field(DFN('dim',2),'Line'));
[xts,ids,La] = Intersect(Field(DFN('dim',3),'Poly'));
```

The ‘Intersect’ super-function finds intersections between 2D lines or 3D polygons. It results in the intersection lines or points (xts), intersection indices (ids) and cluster labels.

Isolated

Syntax:

```
b = Isolated(lns,tol);
```

Example(s):

```
b = Isolated(lns);
```

The ‘Isolated’ function tests 2D or 3D lines for isolation. This is the core function for backbone extraction.

Kriging

Syntax:

```
[krg,err] = Kriging(pts,vrl,d,mdl,gns,draw);
```

Example(s):

```
[d,~,~] = Variocloud([x,y],w,[],true);
[krg,err] = Kriging([x,y],w,d,{'name','sph','nugget',0,'sill',4,'range',50},
            [50,50],true);
```

The ‘Kriging’ function estimates ordinary kriging values for 2D or 3D points. It also results in the error of estimation. This function works best in combination with the functions ‘Variocloud’, ‘Variogram’ and ‘Variomodel’.

Label

Syntax:

```
La = Label(c,n);
```

Example(s):

```
La = Label(c,n);
```

The ‘Label’ function creates cluster labels. It is a significant part of any intersection analysis. Note that, any isolated fracture is associated with a distinct label.

Length

Syntax:

```
lts = Length(in);
```

Example(s):

```
lts = Length(rand(10,4));                    % 2d lines
lts = Length(rand(10,6));                    % 3d lines
lts = Length(Field(DFN('dim',3),'Poly'));    % 3d polygons
```

The ‘Length’ function computes the length of 2D lines and size of 3D polygons.

Markov

Syntax:

```
cs = Markov(P,cs,n);
```

Example(s):

```
s = Markov([0.3,0.5;0.1,0.3;0.6,0.2],1,40);
```

The ‘Markov’ function generates a Markov chain, useful for stochastic based decision-making.

Mesh

Syntax:

```
pip = Mesh(lns,r,pre,mdl);
```

Example(s):

```
msh = Mesh(rand(3,4),0.03,false,'r');
```

The ‘Mesh’ function generates various types (random or systematic) of meshes conditioned to the predefined lines (edges) ‘lns’ as well as a distance ‘r’. If ‘pre’ set to true, the lines will first be split at their intersections. The argument ‘mdl’ can be ‘s’ or ‘r’ defining the model of meshing. If it was ‘s’, then the systematic (regular, structured) otherwise random (unstructured) mesh is produced.

Naned

Syntax:

```
varargout = Naned(lns);
```

Example(s):

```
[x,y] = Naned(rand(10,4));
```

The ‘Naned’ function converts 2D or 3D lines to their axial coordinates with additional ‘nan’ values for separation in between. The resulting can directly be used in ‘plot’ or ‘plot3’ functions for ultimately fast drawing.

Net

Syntax:

```
pts = Net(x,y,varargin);
```

Example(s):

```
pts = Net(1:10,3:4);
pts = Net(1:10,3:4,6:7);
```

The ‘Net’ function creates ‘meshgrid’ in a convenient 2D or 3D point format.

Not

Syntax:

```
y = Not(x,ids);
```

Example(s):

```
y = Not([1,0,4,3,2,1],[1,3]);                % = [0,3,2,1]
```

The ‘Not’ function excludes the items indexed in ‘ids’ from data in ‘x’.

Occurrence

Syntax:

```
k = Occurrence(pts,rts,tol);
```

Example(s):

```
n = 1000;
pts = rand(n,3);
f = randi(10,1,100);
i = randi(n,1,100);
j = repelem(i,f);
rts = pts(j,:);
k = Occurrence(rts,rts);
```

The ‘Occurrence’ function counts the frequency of points ‘pts’ among the reference points ‘rts’ with a distance tolerance.

Option

Syntax:

```
opt = Option(opt,varargin);
```

Example(s):

```
opt = Option(varargin,'a',1);
```

The ‘Option’ function provides a significant convenience while dealing with multiple arguments and defaults in the definition of functions.

Orientation

Syntax:

```
out = Orientation(in);
```

Example(s):

```
out = Orientation(rand(10,4));              % 2D case, [out.Angle]
```

The ‘Orientation’ function computes the orientation angles (dip and dip-direction) for 3D polygons, and the direction angles for 2D lines.

Percolate

Syntax:

```
f = Percolate(a,b);
```

Example(s):

```
f = Percolate(a,b);
```

The ‘Percolate’ function is a helper used conveniently to check percolation state and to show a desired message.

Pipe

Syntax:

```
dfn = Pipe(be,fnm,mtd);
```

Example(s):

```
dfn = Pipe(be,fnm,'cnt');
```

The ‘Pipe’ super-function builds the pipe model based on 2D lines or 3D polygons. It also allows advanced options for creating hybrid mesh-pipe models. The keyword ‘cnt’ refers to the ‘center’ method for generating pipes in 3D case. Other options are ‘tris’ and trir’ referring to the systematic and random triangulation methods, respectively (Fadakar-A and Xu 2018).

Plane

Syntax:

```
pln = Plane(pts,normed);
```

Example(s):

```
pln = Plane(rand(3,3),true);
```

The ‘Plane’ function creates a plane base on the given 3D points. The argument ‘normed’ is a Boolean that directs if a normalized plane is required.

Pointline

Syntax:

```
lns = Pointline(pts,ids);
```

Example(s):

```
lns = Pointline(rand(5,2),[1,2;2,3;1,4]);   % = 3 lines
```

The ‘Pointline’ function creates lines from the points ‘pts’ based on the indices ‘ids’ that refer to the points.

Polyline

Syntax:

```
lns = Polyline(ply);
```

Example(s):

```
lns = Polyline(rand(3,2));                  % = 3 lines
lin = Polyline(rand(1,3));                  % = (6), point to line
```

The ‘Polyline’ function converts the polygon ‘ply’ to polyline. It can also be used to duplicate a row, e.g., for converting a point to a line.

Polynorm

Syntax:

```
nrm = Polynorm(ply);
```

Example(s):

```
nrm = Polynorm(Field(DFN('dim',3),'Poly'));
```

The ‘Polynorm’ function finds the normal vectors to given 3D polygons.

Project

Syntax:

```
[ots,msk] = Project(pts,pln);
```

Example(s):

```
ots = Project(rand(10,3),[0,0,0,1,0,0,0,1,0]);
```

The ‘Project’ function projects the 3D points ‘pts’ onto the plane ‘pln’. This function can be used to reduce 3D cases to 2D, for example.

Rand

Syntax:

```
out = Rand(varargin);
```

Example(s):

```
x = Rand(10,'fun','f','mu',45,'k',1,'ab',[0,90]); % Fisher random numbers,[0..90]
x = Rand(10,'fun','f','mu',45,'k',0);             % Fisher random numbers
x = Rand(10,'fun',@rand,'ab',[0,1]);              % uniform
x = Rand(10,'fun','e','mu',0.5,'ab',[0,1]);       % exponential
ags = Rand(10,'fun','a','mu',10,'ab',[0,90]);     % angle conditioned, 10 degree
pts = Rand(10,'fun','d','d',0.1,'ab',[0,0,0;1,1,1]);  % distance conditioned
lns = Rand(10,'fun','l','ab',rand(3,2),'d',[0.03,0.01],'mu',[0,10]); % rand. lines
```

The ‘Rand’ is a super function that generates random numbers according to the given function ‘keyword’. The implemented random functions are the ‘uniform’, ‘Fisher’, and ‘exponential’ distributions. More importantly, one can also define own random function to pass it to ‘Rand’ function as the argument ‘fun’, as shown in the above examples. In addition, generating conditional random angles, and distances are available. Furthermore, random 2D lines can be generated by passing ‘l’ or ‘line’ to the argument ‘fun’.

Ray

Syntax:

```
ray = Ray(ply);
```

Example(s):

```
ray = Ray([0,0;1,0;1,1]); % = {3 lines}
```

The ‘Ray’ function creates rays from the center of 2D or 3D polygons to all their vertices.

Relabel

Syntax:

```
Ra = Relabel(La);
```

Example(s):

```
Ra = Relabel([1,1,-1,2,2,-2,1,2,3]);        % = [2,2,-1,3,3,-2,2,3,1]
```

The ‘Relabel’ function generates new labels for clusters based on their frequencies (cardinality or population).

Reshape

Syntax:

```
y = Reshape(x,m,n);
```

Example(s):

```
y = Reshape([1,2,3,4,5,6],2,3);             % = [1,2,3; 4,5,6]
y = reshape([1,2,3,4,5,6],2,3);             % = [1,3,5; 2,4,6] Matlab's function
```

The ‘Reshape’ function correctly reshapes the matrix ‘x’ into (m, n) size. It works column-wise in contrast with Matlab’s ‘reshape’ function that works row-wise. See the examples above for comparison.

Rotate

Syntax:

```
out = Rotate(in,varargin);
```

Example(s):

```
pts = Rotate(rand(10,2),'ang',45,'mov',[1,10]);  % 2d points or polygons
pts = Rotate(rand(10,3),'x',45,'y',30,'z',10);   % 3d points
lns = Rotate(rand(10,4),'ang',45);               % 2d lines
lns = Rotate(rand(10,6),'x',45);                 % 3d lines
ply = Rotate(Poly.Top,'x',45);                   % 3d polygons
```

The 'Rotate' is a supper function that rotates 2D or 3D points, lines and or polygons. One can define the center (2D) or axis (3D) of rotation. The optional argument 'mov' translates the results to the given position.

Scale

Syntax:

```
y = Scale(x,varargin);
```

Example(s):

```
y = Scale(rand(1,10));                         % = scaled to [0..1]
y = Scale(rand(1,10),3,7);                     % = scaled to [3..7]
y = Scale([0,3,0,9],-1,1,3,7);                 % = [5,11,5,23]
y = Scale(rand(10,4),'lin',0.5);               % = 2d lines scaled to half, centered
y = Scale(rand(10,6),'lin',0.5);               % = 3d lines scaled to half, centered
y = Scale({rand(3,2)},'ply',0.5);              % = 2d polygons scaled to half
y = Scale({rand(3,3);rand(3,3)},'ply',1/3); % = 3d polygons scaled to one third
```

The 'Scale' is a super function that scales (up or down) anything including numbers, points, lines and polygons. For geometries, it works seamlessly efficient in both 2D and 3D cases. See the examples above for information.

Snap

Syntax:

```
varargout = Snap(pts,lns,tol);
```

Example(s):

```
pts = Snap(rand(1000,2),rand(2,4)); % 2d
pts = Snap(rand(1000,3),rand(2,6)); % 2d
d = Snap(rand(1000,2),rand(3,4),0); % 2d distances, = 1000x3
d = Snap(rand(1000,3),rand(3,6),0); % 3d distances, = 1000x3
```

The ‘Snap’ is a super function that snaps 2D or 3D points ‘pts’ to the lines ‘lns’ based on the ‘tol’ value. If ‘tol’ was set 0 (zero) this function returns the distances between the points and the lines. It works in both dimensions (2D and 3D) automatically and efficiently.

Solve

Syntax:

```
dfn = Solve(dfn,bv,kv);
```

Example(s):

```
dfn = Solve(dfn);
dfn = Solve(dfn,[1,0],1.787e-6);
```

The ‘Solve’ function solves the developed ‘dfn’ (i.e., it includes graph structure) for fluid flow through the fractures. It computes the pressures field, flow distribution, permeability and many other parameters. All the resulting parameters are added to the graph. It works efficiently for both 2D and 3D dimensions. For very large ‘dfn’ models, one should consider modification of the ‘A’ matrix to sparse one in order to manage memory requirements. Also sparse solver and other setups are helpful.

Sort

Syntax:

```
out = Sort(in,dir);
```

Example(s):

```
pts = Sort(rand(10,2),nan);                     % = topologically sorted points
```

The 'Sort' function sorts 2D or 3D points in a topological order. If 'dir' was 'nan', the direction of topology is computed automatically based on the given points.

Split

Syntax:

```
[ols,jds,sn,xts,ids,La] = Split(lns,itv);
```

Example(s):

```
ols = Split(rand(10,4),0);                      % split 2d lines at intersections
ols = Split(rand(10,4),0.01);                   % split 2d lines at intervals = 0.01
ols = Split(rand(10,6),0.01);                   % split 3d lines at intervals = 0.01
```

The 'Split' function splits 2D lines at their intersections. It can also split 2D or 3D lines at any defined intervals. This function is used in 'Mesh' function, for example.

Stack

Syntax:

```
a = Stack(c);
```

Example(s):

```
a = Stack({[1,2],[3,4]}); % = [1,2;3,4]
```

The 'Stack' function stacks up cell values resulting a matrix. This function provides a useful tool to generate a matrix from structured data, such as graph data.

Stats

Syntax:

```
[x1,m,x2,s,n] = Stats(x,lbl);
```

Example(s):

```
[x1,m,x2,s,n] = Stats(rand(100,10),'X');
```

The ‘Stats’ function returns and prints nicely the number of data, minimum, mean, maximum and standard deviation values of the input data ‘x’.

Stereonet

Syntax:

```
cells = Stereonet(dip,ddir,varargin);
```

Example(s):

```
c = Stereonet([0,45,90],[0,180,270]);
```

The ‘Stereonet’ function computes and draws Stereonet diagram for dip and dip-direction angles. It also offers the density map.

Summary

Syntax:

```
Summary(h);
```

Example(s):

```
Summary({{'a','b'}});                           % = a:b
```

The ‘Summary’ function is a convenient way to displays cell structures.

Syncax

Syntax:

```
Syncax(varargin);
```

Example(s):

```
Syncax(ax1,ax2);
```

The ‘Syncax’ function synchronizes the rotation between multiple axes e.g., subplots.

Tensor

Syntax:

```
out = Tensor(g,varargin);
```

Example(s):

```
out = Tensor(dfn.grh,'dim',3);
```

The ‘Tensor’ function computes and draws the tensor ellipse or ellipsoid based on the information (permeability, pressure etc.) in the informed Graph i.e., the resulting graph from the ‘Solve’ function.

Ticot

Syntax:

```
Ticot(msg);
```

Example(s):

```
Ticot('Graphing');
{…code…}                                  % code to be timed
Ticot;                                    % to be used always in pairs
```

The ‘Ticot’ function manages and displays the elapsed time for the execution of code in between pairs. See the example above for clarification.

Time

Syntax:

```
s = Time(f);
```

Example(s):

```
s = Time;
```

The ‘Time’ function returns the current time in two formatted strings. It is very useful for auto-naming files or processes, for example.

Translate

Syntax:

```
y = Translate(x,mov);
```

Example(s):

```
pts = Translate(rand(10,2),[1,10]);         % 2d points
pts = Translate(rand(10,3),[1,10,3]);       % 3d points
lns = Translate(rand(10,4),[1,10]);         % 2d lines
lns = Translate(rand(10,6),[1,10,3]);       % 3d lines
ply = Translate({rand(3,2)},[1,10]);        % 2d polygon
ply = Translate({rand(3,3)},[1,10,3]);      % 3d polygon
```

The ‘Translate’ is a super function to move (translate) points, lines and polygons around. It works efficiently in both 2D and 3D cases.

Unique

Syntax:

```
[out,ia,ic] = Unique(in,tol);
```

Example(s):

```
pts = Unique(pts);
lns = Unique(lns);
```

The ‘Unique’ function removes duplicate points and lines in 2D and 3D. It is smart enough to address reversed line problem, automatically.

Upscaling

Syntax:

```
out = Upscaling(in,mtd,up);
```

Example(s):

```
out = Upscaling(rand(2^4,2^4),'geometric',2);  % = 2x2 matrix
```

The 'Upscaling' function up-scales the input matrix 'in' following the method 'mtd' to the level defined by 'up'. Available methods are 'arithmetic', 'harmonic' and 'geometric'. The two later methods might be suitable for upscaling non-additive variables such as the permeability.

Variocloud

Syntax:

```
[d,g,v] = Variocloud(pts,vls,d,draw);
```

Example(s):

```
[d,g,v] = Variocloud(rand(10,2),rand(10,1),[],true);
```

The 'Variocloud' function computes and draws the variogram cloud of the 2D or 3D points 'pts' with values defined in 'vls'.

Variogram

Syntax:

```
[h,v,p] = Variogram(d,g,s,md,draw);
```

Example(s):

```
[h,v,p] = Variogram(d,g,v,3,true);          % inputs from Variocloud
```

The 'Variogram' function computes and draws the variogram of data. It accepts input from 'Variocloud' function.

Variomodel

Syntax:

```
out = Variomodel(mdl,lgs,draw);
```

Example(s):

```
Variomodel({'name','sph','nugget',0,'sill',4,'range',40},0:0.5*max(d),true);
```

The ‘Variomodel’ function results in a variogram model. The available standard models are ‘spherical’, ‘exponential’ and ‘linear’ models.

Wrap

Syntax:

```
r = Wrap(i,n);
```

Example(s):

```
r = Wrap(1:10,4);                           % = [1,2,3,4,1,2,3,4,1,2]
```

The ‘Wrap’ function wraps numeric input ‘i’ around number ‘n’.

CBackbone

Syntax:

```
Cbb = CBackbone(dfn,tol);
```

Example(s):

```
cbb = CBackbone(dfn,1e-12);                 % = class, cbb.dfn
dfn = CBackbone(dfn,1e-12).Graph.Solve;     % structure
```

The ‘CBackbone’ class provides helper for Backbone function, resulting in ‘.’ interface.

CData

Syntax:

```
OUT = CData.FUNCTION(ARGUMENTS);
```

Example(s):

```
b = CData.Almost(rand(1,10),0.5,0.1);
b = CData.Almost(rand(1,10)*2*eps,0,eps);   % close to zero
[x,y] = CData.Deal(rand(3,2));
dts = CData.Discretize(rand(1,100),4);
CData.Filewrite('this.txt','This');
y = CData.Get([1,2,3,4]',1);                % = 1
y = CData.Get([1,2,3,4]',10);               % = 4
[les,equ,grt] = CData.Leg(rand(1,10),0.5,0.01);
CData.Map([1,2,3],@max,@min,@mean,@std,@var,@(x)x(end));  % = [3,1,2,1,1,3]
y = CData.Array(rand(1,4),rand(3,2));       % 2d line
y = CData.Array(rand(10,4),rand(3,2));      % 2d line sets
y = CData.Array(rand(5,3),rand(3,3));       % points, single polygon
y = CData.Array(rand(3,2),rand(3,2));       % 2d polygon sets
y = CData.Array({rand(3,2)},rand(3,2));     % single polygon
y = CData.Array({rand(3,2);rand(3,2)},rand(4,2));  % polygons
```

The ‘CData’ class contains several useful functions that apply to data. The included functions are ‘Almost’, ‘Deal’, ‘Discretize’, ‘Filewrite’, ‘Get’, ‘Leg’, ‘Map’ and ‘Array’. See the examples above for clarification of each.

CGraph

Syntax:

```
Cgr = CGraph(dfn,aFun,dFun);
```

Example(s):

```
cgr = CGraph(dfn);                          % = class, cgr.dfn
dfn = CGraph(dfn).Solve;                    % structure
```

The ‘CGraph’ class provides helper for the Graph function, resulting in ‘.’ Interface.

CPipe

Syntax:

```
cpp = CPipe(be,fnm,mtd);
```

Example(s):

```
cpp = CPipe(be,fnm,'cnt');                    % = class, cpp.dfn
dfn = CPipe(be,fnm,'cnt').Backbone.Graph.Solve;  % structure
```

The ‘CPipe’ class provides helper for the Pipe function, resulting in ‘.’ Interface.

Globals

Syntax:

```
Globals;
```

Example(s):

```
Globals; Interval = 0.01;                     % after invoke, modification possible
```

The ‘Globals’ script stores global variables, settings, functions, initializations, etc. for ADFNE1.5 package. For example, it includes predefined polygons (accessible through ‘Poly.’) and lines (accessible through ‘Line.’), Tolerance value and so on. It also updates Matlab’s ‘path’ list to be prepared for necessary subfolders within ADFNE1.5. This script must be called at the first line of any code.

Demo

Syntax:

```
Demo
```

The ‘Demo’ script includes multiple case studies all made using ADFNE1.5 package. The example runs are reported in the next chapter.

Demo Runs – Case Studies

All of the following case studies are full codes. Use the following template for running each of the following full codes. Note that, all these codes and more are available in the file "Demo.m" for your convenience.

```
Globals;
rng(1234567890);                                         % Reproducibility
{ any of the following full codes }
```

Removing "`rng(1234567890);`" allows one to experience own case studies e.g., random setups / simulations.

The ADFNE1.5 was developed based on Matlab R2016b version. I had a chance to try the demonstrations on the Matlab R2015a and R2017a versions as well. No issues on R2017a but I noticed that on R2015a the two functions "pad" and "contains" (string functions) were not available; therefore, I wrote them for your convenience. The two are located in the 'R2015a' folder. I also noticed that the following operations which are legal and valid in R2016b (and later versions) were not allowed in R2015a, hence I addressed them by the following solution.

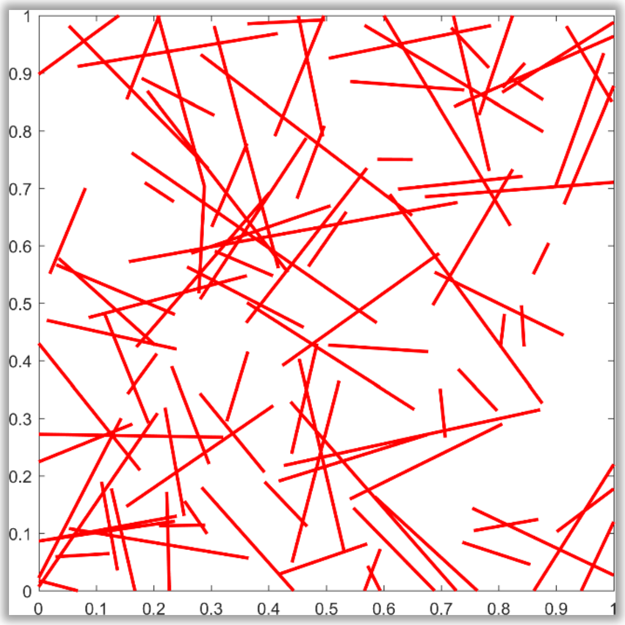
R2016b and later	**R2015a** and earlier
`pts = rand(10,3)-[1,2,3];`	`pts = bsxfun(@minus,rand(10,3),[1,2,3]);`

Similarly for "+" use "`bsxfun(@plus`", for "`.*`" use "`bsxfun(@times`" and for "`./`" use "`bsxfun(@rdivide`". Basically, on Matlab R2015a and earlier versions, if you received "Matrix dimensions must agree." error anywhere / anytime during your runs with ADFNE1.5, the above solutions are to be followed. You do not need any fix and should not receive any error if you run on Matlab R2016b and later versions. If you encountered by an error "Static method or constructor invocations cannot be indexed...", while using "`CPipe(...`" form, then use the other non-class form to get exactly the same results, as follows.

```
dfn = CPipe(be,fnm,'cnt').Backbone.Graph.Solve([1,0]);        % >= Matlab R2016b
dfn = Solve(Graph(Backbone(Pipe(be,fnm,'cnt'))),[1,0]);       % <= Matlab R2015a
```

1. 2D DFN Model

```
Draw('lin',Field(DFN,'Line'));                                % 2D DFN Model
Export(gcf,'dfn_2d_adfne1.5.png');                            % Export
```

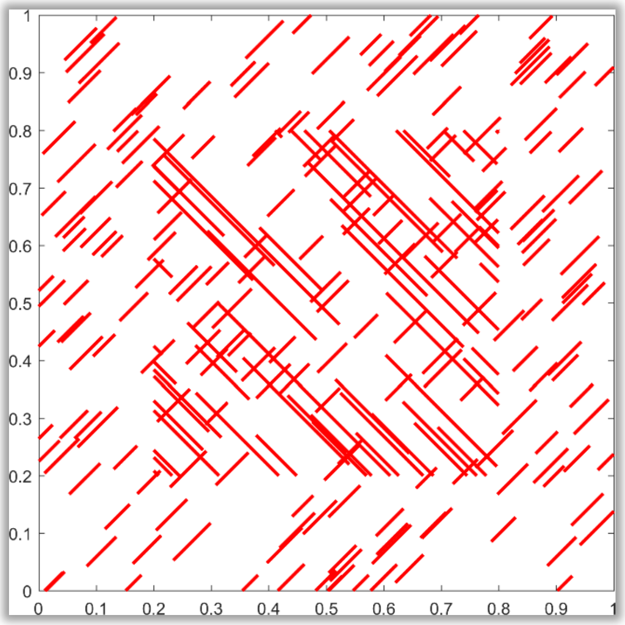


Notes

- *DFN, is the simulator, Field is the selector, Draw is the plotter and Export is the exporter;*
- *DFN when used with no arguments uses all default values;*
- *Export exports the figure in high quality 300dpi resolution;*
-
-
-
-
-

2. 2D DFN Model – Multiple Sets

```
set1 = Field(DFN('dim',2,'n',200,'dir',45,'ddir',-1e9,'minl',0.05,...
       'mu',0.07,'maxl',0.1,'bbx',[0,0,1,1]),'Line');          % DFN set 1
set2 = Field(DFN('dim',2,'n',100,'dir',135,'ddir',-1e9,'minl',0.1,...
       'mu',0.3,'maxl',0.5,'bbx',[0.2,0.2,0.8,0.8]),'Line');   % DFN set 2
fnm = [set1;set2];                                             % combined DFN
Draw('lin',fnm);                                               % Draw
Export(gcf,'dfn_2d_sets_adfne1.5.png');                        % Export
```

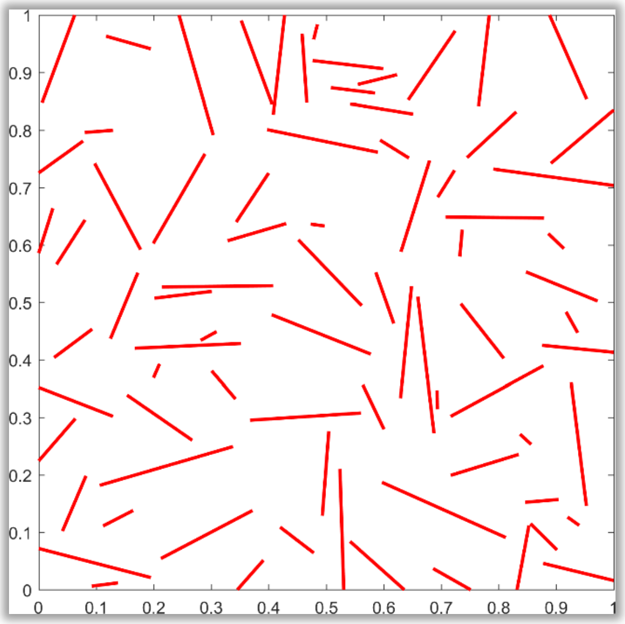


Notes

- *Combine any number of sets with ease as the above;*
- *Each fracture set can be different, can be clipped differently, can be located differently;*
- *'mu' determines the mean value of exponential distribution used for lengths;*
- *'minl' is minmum length and 'maxl' is maximum length;*
- *'ddir' if negative, its absolute equals to kappa for Fisher distribution of angles;*
- *Larger kappa means less dispersion;*
-
-
-

3. 2D DFN Model – Conditional

```
fnm = Field(DFN('dim',2,'n',100,'minl',0.02,'mu',0.2,'maxl',0.3,...
      'bbx',[0,0,1,1],'asep',0.1,'dsep',0.001,'mit',100),'Line'); %Conditional DFN
Draw('lin',fnm);
Export(gcf,'dfn_2d_cnd_adfne1.5.png');
```

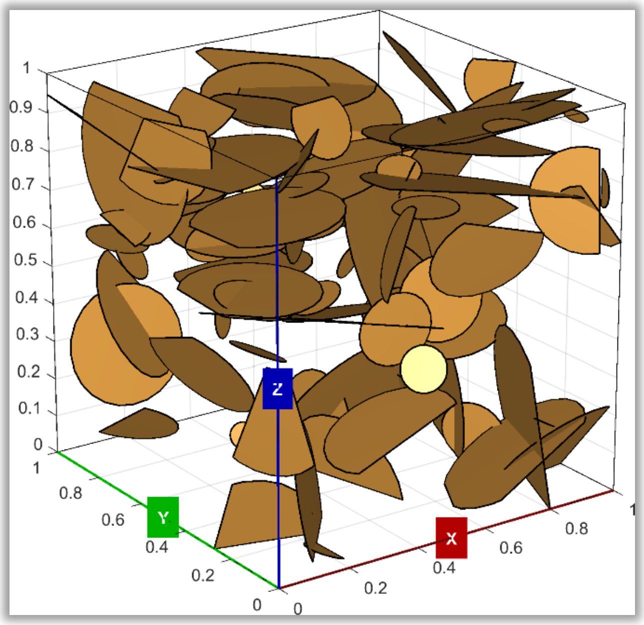


Notes

- *Two conditions can be applied: angle and distance conditions;*
- *asep = 0.1 forces fractures to have at least 0.1 degree angle difference…*
- *… if their bounding boxes are close but not closer than 0.001 unit;*
- *'mit' maximum repeats for any new fracture to meet the conditions;*
- *Your run may generate slightly different result;*
-
-
-
-
-
-

4. 3D DFN Model

```
Draw('ply',Field(DFN('dim',3),'Poly'));                 % 3D DFN Model
Export(gcf,'dfn_3d_adfne1.5.png');                      % Export
```

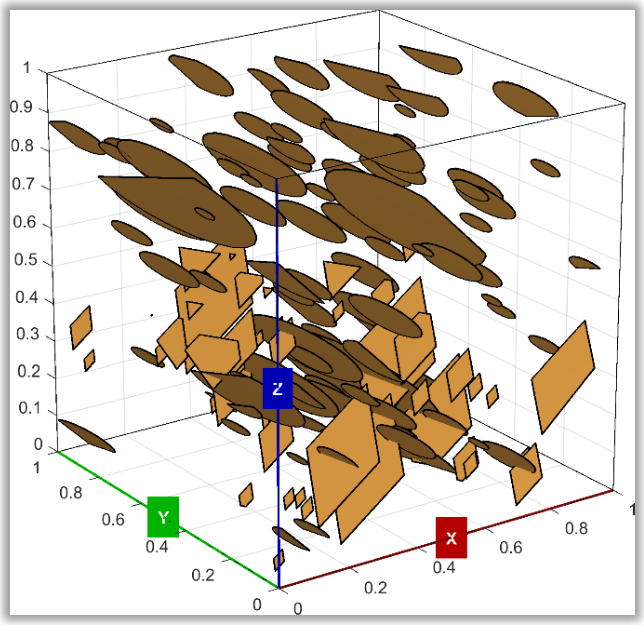


Notes

- *All default values are used in the above;*
- *Default shape is ellipse;*
-
-
-
-
-
-
-
-
-
-

5. 3D DFN Model – Multiple Sets

```
set1 = Field(DFN('dim',3,'n',100,'dir',15,'ddir',-1e9,'minl',0.05,...  % DFN set 1
       'mu',0.1,'maxl',0.5,'bbx',[0,0,0,1,1,1],'dip',45,'ddip',-1e7),'Poly');
set2 = Field(DFN('dim',3,'n',100,'dir',165,'ddir',-1e9,'minl',0.05,... % DFN set 2
       'mu',0.1,'maxl',0.5,'bbx',[0,0,0,1,1,0.5],'dip',45,'ddip',-1e7,...
       'shape','e','q',4),'Poly');
fnm = [set1;set2];                                             % Combined DFN
Draw('ply',fnm);
Export(gcf,'dfn_3d_sets_adfne1.5.png');
```

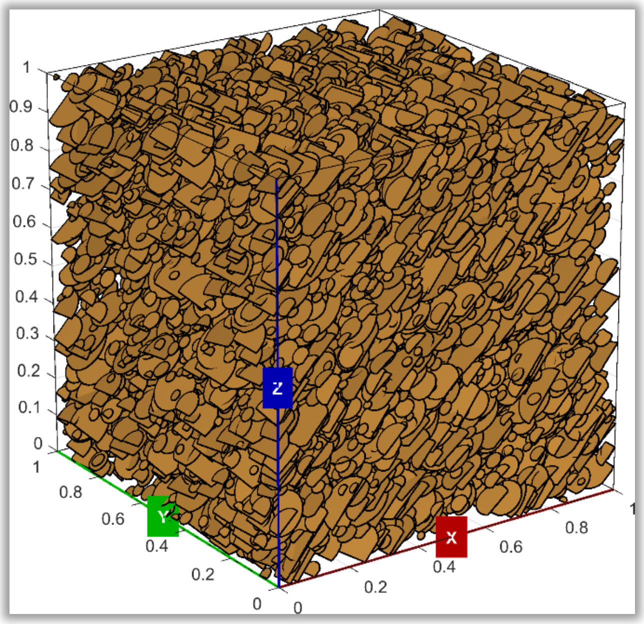


Notes

- *Other shapes for fractures are circle, square, and legacy;*
- *For Legacy (a random quadrangle) see Fadakar-A et al., 2011;*
- *'q' determines the discretization resolution; if 3 the result is a triangle;*
-
-
-
-
-
-

6. 3D DFN Model – Large

```
fnm = Field(DFN('dim',3,'n',10000,'dir',180,'ddir',10,'minl',0.02,...
      'mu',0.05,'maxl',0.15,'bbx',[0,0,0,1,1,1],'dip',45,'ddip',10),'Poly');
Draw('ply',fnm);
Export(gcf,'dfn_3d_large_adfne1.5.png');
```

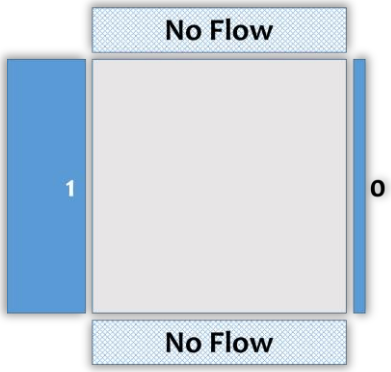


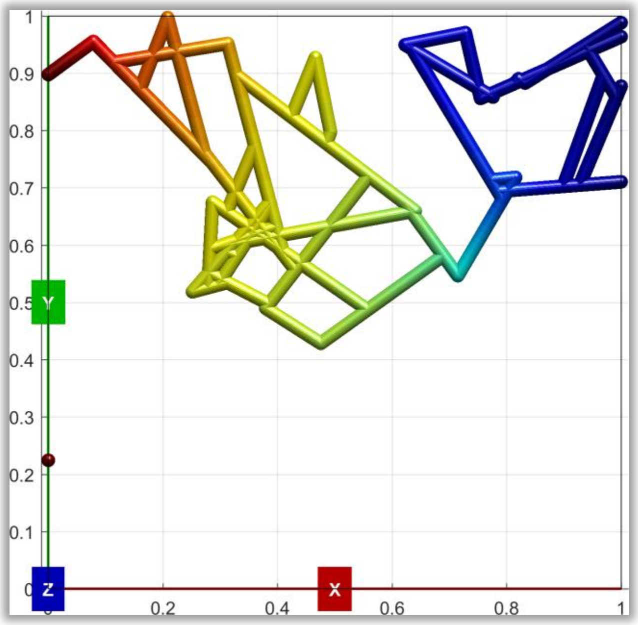
Notes

- *A 10,000 fracture network is more than enough in size for many applications;*
- *It takes couple of seconds to simulate;*
- *ADFNE1.5 is capable of simulating even millions;*
-
-
-
-
-
-
-
-

7. 2D DFN Model + Fluid Flow (BC: LR, *no-flow)

```
fnm = Field(DFN,'Line');                                    % DFN Model
be = [Line.Left;Line.Right];                                % Boundary Elements
dfn = CPipe(be,fnm,'cnt').Backbone.Graph.Solve([1,0]);      % All Stages
Draw(dfn,'what','seniq');                                   % Visualization
Export(gcf,'dfn_2d_pressure_adfne1.5.png');                 % Export
```



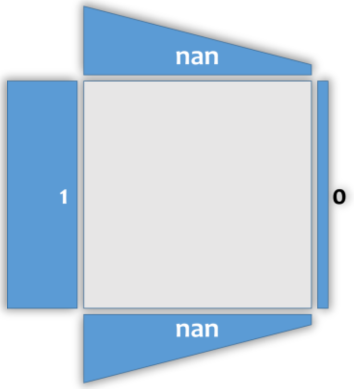


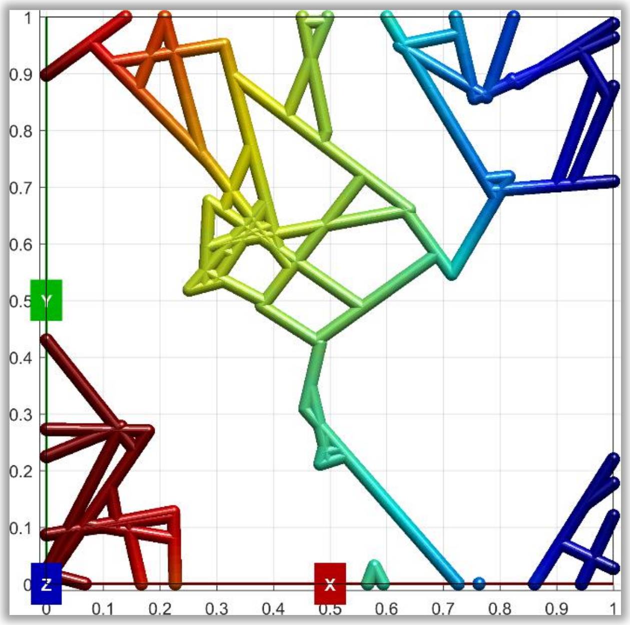
Notes

- *Boundary conditions can be defined quite easily;*
- *'be' boundary elements; their associated values are fed to 'Solve';*
- *In the above, left side is inlet (value 1) and the right side is outlet (value 0);*
- *Top and bottom sides are no-flow boundaries;*
- *Framework for flow modeling: DFN -> Pipes -> Backbone -> Graph -> Solution;*
- *'seniq': s=solution{e:edges, n:nodes}, i=only inner ones, q=only those contribute to flow;*
-
-
-
-

8. 2D DFN Model + Fluid Flow (BC: LR, *linear)

```
fnm = Field(DFN,'Line');                                     % 2D DFN Model
be = [Line.Left;Line.Right;Line.Top;Line.Bottom];            % BE:{LR,TB:linear}
dfn = CPipe(be,fnm,'cnt').Backbone.Graph.Solve([1,0,nan,nan]); % solution
Draw(dfn,'what','seniq');                                    % Visualization
Export(gcf,'dfn_2d_pressure_lin_adfne1.5.png');              % Export
```



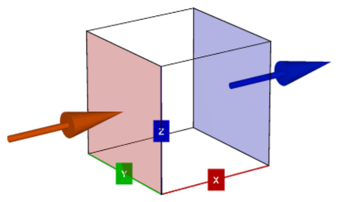


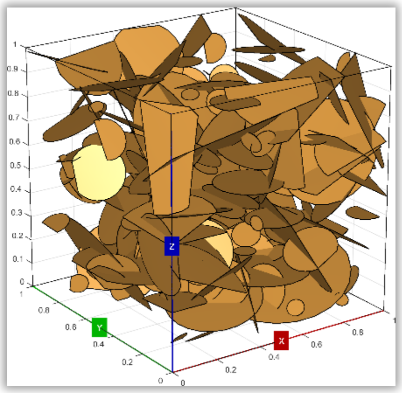
Notes

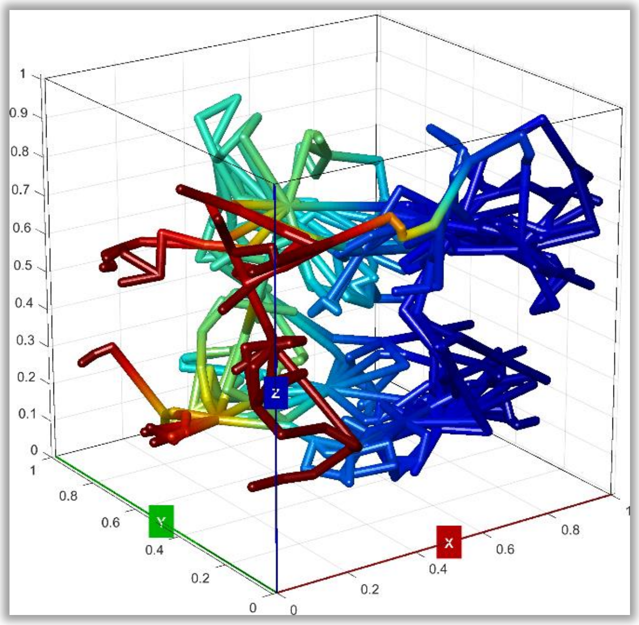
- *'nan' if set for a boundary element its value will be linearly interpolated;*
- *ADFNE1.5 is capable of handling countless boundary elements (lines);*
- *Intersecting boundary elements are also allowed;*
-
-
-
-
-
-
-

9. 3D DFN Model + Fluid Flow (BC: LR, *no-flow)

```
fnm = Field(DFN('dim',3,'n',200),'Poly');                    % 3D DFN Model
be = {Poly.Left;Poly.Right};                                 % BE
dfn = CPipe(be,fnm,'cnt').Backbone.Graph.Solve([1,0]);       % solution
Draw(dfn,'what','seniq');                                    % Visualization
Export(gcf,'dfn_3d_pressure_adfne1.5.png');                  % Export
```





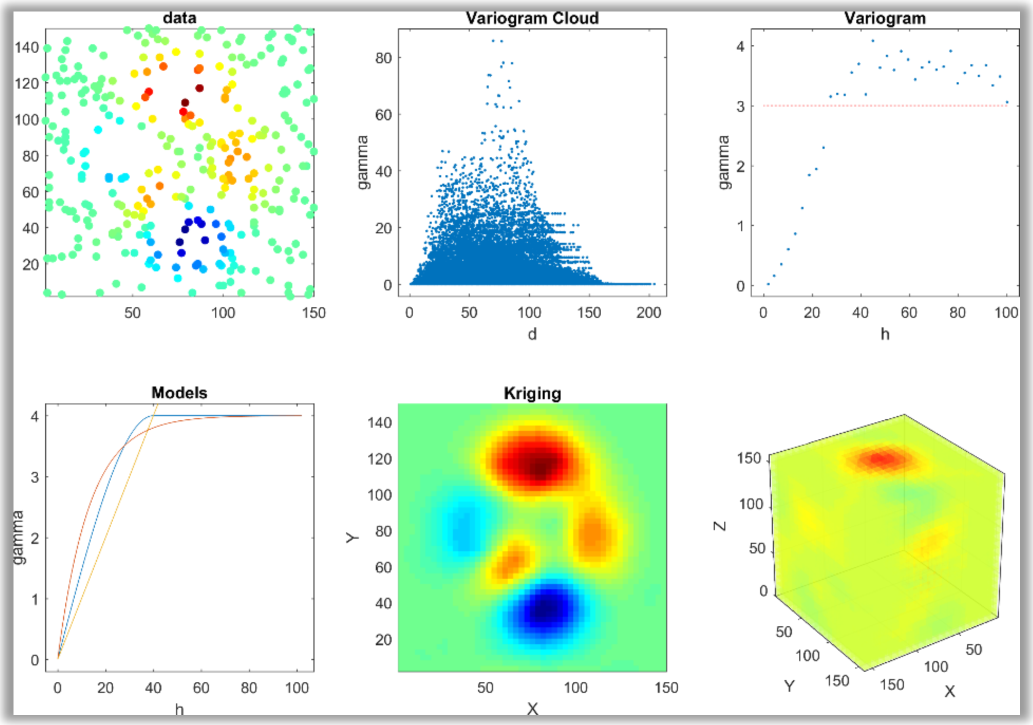


Notes

- *Framework and functions are exactly the same for 2D and 3D case;*
- *Everything in ADFNE1.5 is smart enough to distinguish what the case is;*
- *In the above example, Left side is inlet; Right is outlet; others no-flow;*
-
-
-
-
-
-
-

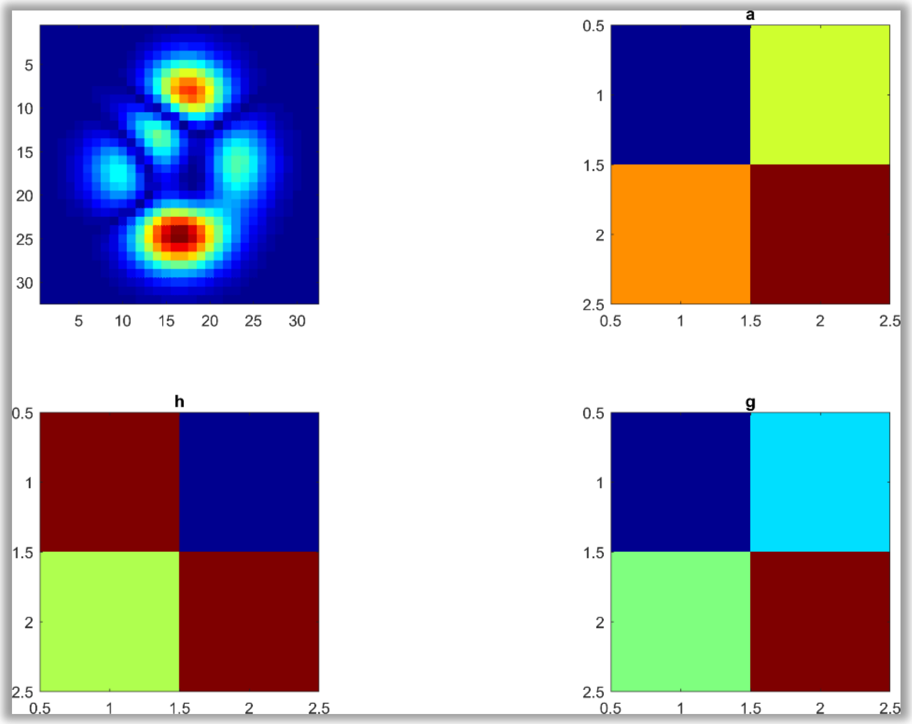
10. Geostatistical Modeling – 2D | 3D

```
n = 150;
[x,y] = meshgrid(1:n,1:n); w = peaks(n);            % map with spatial correlation
x = x(:); y = y(:); w = w(:); nn = randperm(n*n); f = nn(1:2*n);  % sampling
x = x(f); y = y(f); w = w(f);
clf;
subplot(231); scatter(x,y,10,w); axis image; box on; title('data');  % data
subplot(232); [d,g,v] = Variocloud([x,y],w,[],true);           % variocloud
subplot(233); Variogram(d,g,3,v,true);                         % variogram
subplot(234);                                                  % variomodels
Variomodel({'name','sph','nugget',0,'sill',4,'range',40},...   % spherical model
           0:0.5*max(d),true); hold on;
Variomodel({'name','exp','nugget',0,'sill',4,'range',40},...   % exponential model
           0:0.5*max(d),true);
Variomodel({'name','lin','nugget',0,'slope',0.1},...           % linear model
           0:0.5*max(d),true);
subplot(235);
[krg,err] = Kriging([x,y],w,d,{'name','sph','nugget',0,'sill',4,...  % 2d kriging
            'range',50},[50,50],true);
subplot(236);
xyz = [x,y,Scale(rand(size(x)),0,150)];                        % making 3d data
[krg3,err3] = Kriging(xyz,w,d,{'name','sph','nugget',0,'sill',4,...  % 3d kriging
              'range',50},[20,20,20],true);
Export(gcf,'geostatistics_adfne1.5.png');
```



11. Upscaling – 2D

```
in = abs(peaks(2^5));
subplot(221); imagesc(in); axis image                        % original data
subplot(222); imagesc(Upscaling(in,'arithmetic',2)); axis image  % arithmetic
subplot(223); imagesc(Upscaling(in,'harmonic',2)); axis image    % harmonic
subplot(224); imagesc(Upscaling(in,'geometric',2)); axis image   % geometric
Export(gcf,'upscaling_ahg_adfne1.5.png');                     % Export
```

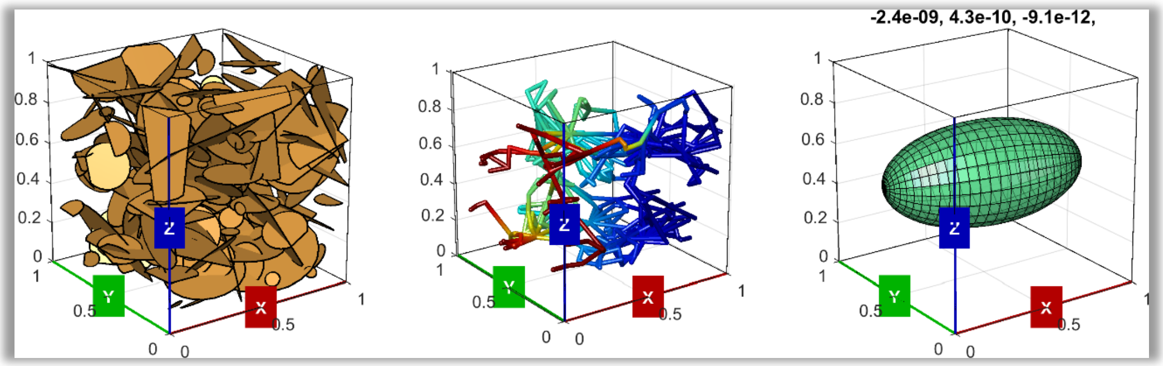


Notes

- *Upscaling is an averaging method, indeed;*
- *Upscaling in ADFNE1.5 is a recursive function;*
-
-
-
-
-
-

12. Tensor – 3D

```
fnm = Field(DFN('dim',3,'n',200),'Poly');                   % 3D DFN model
be = {Poly.Left;Poly.Right};                                % BE
dfn = CPipe(be,fnm,'cnt').Backbone.Graph.Solve([1,0]);      % all stages
if isfield(dfn,'grh')                                       % any solution?
    subplot(131); Draw('ply',fnm);
    subplot(132); Draw(dfn,'what','seniq');
    subplot(133); tns = Tensor(dfn.grh,'dim','3d');         % 3D Tensor
    title(sprintf('%0.1e, ',tns.Tensor));                   % tensor values
end
Export(gcf,'tensor_adfne1.5.png');
```

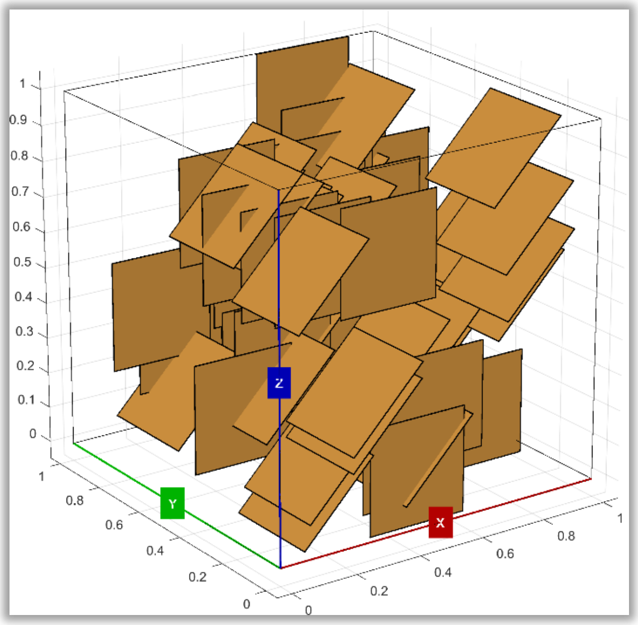


Notes

- *Tensors are computed easily in two setups: 2D and 3D;*
- *The above is 3D tensor; axes aligned values are shown for permeability;*
- *Framework: DFN -> Flow -> Permeability Tensors*
-
-
-
-
-
-
-
-

13. 3D DFN Model – Yet another Way

```
set1 = CData.Array(Rotate(Scale(Poly.Back,'ply',0.3),'y',0,'mov',...
       [0,0,0]),Scale(rand(30,3),0.1,0.9));
set2 = CData.Array(Rotate(Scale(Poly.Left,'ply',0.3),'y',45,'mov',...
       [0,0,0]),Scale(rand(30,3),0.1,0.9));
fnm = [set1;set2];                                      % combined DFN
clf; Draw('ply',fnm,'axes',false); Axes;
Export(gcf,'dfn_3d_other_adfne1.5.png');
```

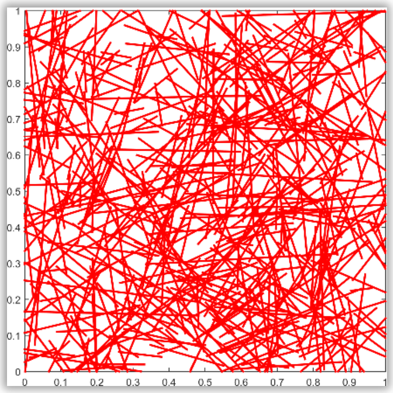
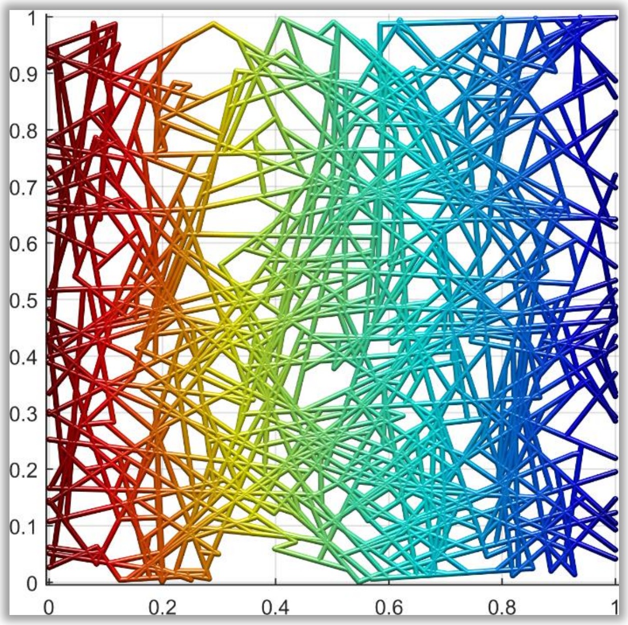


Notes

- *The generic and advanced functions in ADFNE1.5 allows to explore new possibilities;*
- *No DFN simulator is used for the above; instead operator functions are nicely used;*
- *By mastering ADFNE1.5 even more and more innovations are promised;*
-
-
-
-
-
-

14. 2D DFN – Flow, Medium Size (500)

```
fnm = Field(DFN('dim',2,'n',500),'Line');                    % DFN, 500 fractures
be = [Line.Left;Line.Right];                                 % boundary elements
dfn = CPipe(be,fnm,'cnt').Backbone.Graph.Solve([1,0]);       % all stages
Draw(dfn,'what','seniq','r',0.005);                          % visualization
Export(gcf,'flow_2d_500_adfne1.5.png');                      % save as image file
```

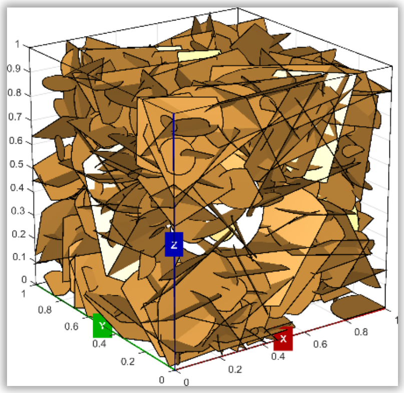



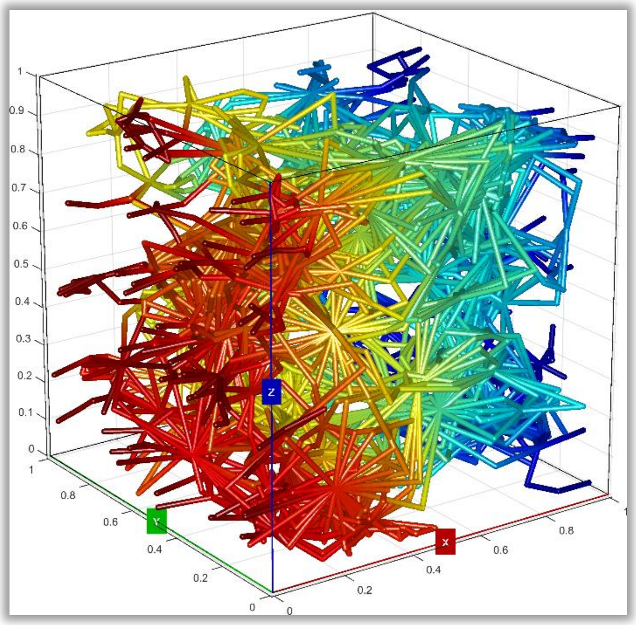
Notes

- *A medium size 2D DFN model, quite satisfactory for many applications;*
- *TIPS:*
- *there is no limit in ADFNE1.5 for DFN size;*
- *go for larger DFN models, fit memory consumption and computation time;*
- *go for 'sparse' matrices and associated functions to manage the computation cost;*
- *choose smart modeling solutions; go deeper and larger iteratively;*
-
-
-
-

15. 3D DFN – Flow, Medium Size (500)

```
fnm = Field(DFN('dim',3,'n',500),'Poly');                    % DFN 500 fractures
be = {Poly.Left;Poly.Right};
dfn = CPipe(be,fnm,'cnt').Backbone.Graph.Solve([1,0]);
Draw(dfn,'what','seniq','r',0.007);
Export(gcf,'flow_3d_500_adfne1.5.png');
```



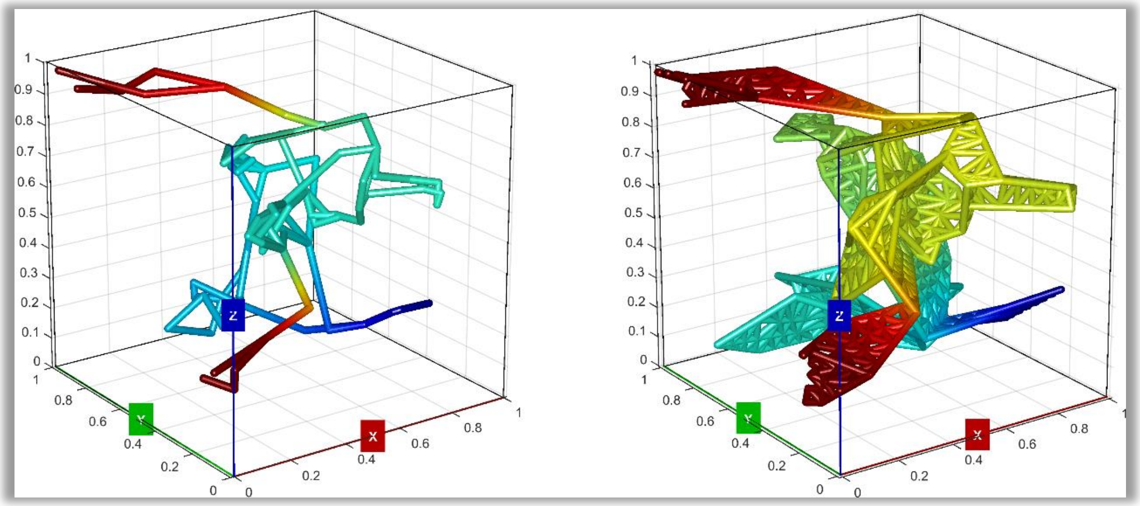


Notes

- *A medium size 3D DFN model, quite satisfactory for many applications;*
- *TIPS: for larger DFN models, follow recommendations listed in the previous page;*
-
-
-
-
-
-
-
-

16. 3D DFN – Flow, Hybrid Mesh-Pipe Model

```
fnm = Field(DFN('dim',3,'n',90),'Poly');                      % 3D DFN model
be = {Poly.Left;Poly.Right};                                   % BE
dfn1 = CPipe(be,fnm,'cnt').Backbone.Graph.Solve([1,0]);        % pipe solution
Draw(dfn1,'what','seniq','sub',121);
Interval = 0.05;                                               % split interval
dfn2 = CPipe(be,fnm,'tris').Backbone.Graph.Solve([1,0]);       % hybrid mesh-pipe
Draw(dfn2,'what','seniq','sub',122);                           % ...solution
Export(gcf,'flow_3d_hybrid_mesh_pipe_adfne1.5.png');
```

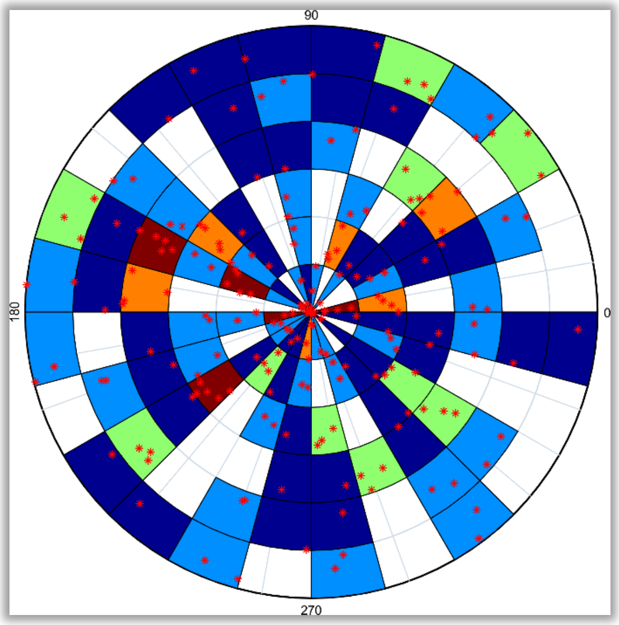


Notes

- *A new hybrid mesh-pipe model for modeling flow through 3D fracture networks;*
- *Reference:*
- *Fadakar-A Y, Xu C, 2018*, A New Hybrid Mesh-Pipe Method for Modeling Fluid Flow in 3D Discrete Fracture Networks, DFNE2018 (June 2018), Seattle, USA (*submitted).*
-
-
-
-
-
-

17. Stereonet Map – Characterization

```
fnm = Field(DFN('dim',3,'n',200),'Poly');                  % 3D DFN model
o = Orientation(fnm);                                       % orientation info
Stereonet([o.Dip],[o.Dir],'density',true,'marker','*',...   % Stereonet map
          'ndip',6,'ndir',24,'cmap',@jet,'color','r');
Export(gcf,'stereonet_adfne1.5.png');                       % exports the map
```



Notes

- *Stereonet map to visualize dip and dip-direction information;*
- *Density of poles can be plotted as the background;*
- *Very customizable;*
-
- *TIPS: Characterization of DFN models:*
- *Function "Info" can be used to extract Locations, Lengths and Orientations information in one step;*
- *Function "Intersect" can be used to extract clustering information;*
-
-

Performance

All functions, classes and scripts in ADFNE1.5 are pure Matlab codes for the maximum readability. For top performance, advanced optimizations have been implemented including bounding box optimization for intersection analysis, vectorized executions and many others. For all usual uses the executions are fulfilled in fraction of a second to couple of minutes. For very large DFN simulations (100,000+) that include intersection analysis, characterization, flow modeling and so on, one may consider the following solutions:

- Managing the memory consumption by modifying wherever possible and needed the matrices to sparse format. This can easily be done by Matlab's functions. Sparse matrices are definite solutions for almost all cases. For very specific situations if the sparse format was not possible or efficient, one may use advanced fast compressed disk solutions.
- Translating functions with heavy calculations into 'mex' compiled format by means of Matlab's Coder tool. In some cases, such translations would increase the performance significantly, occasionally 1000 times or even more.

TIPS: To determine what code takes the significant time in execution, one can use "Run & Time" functionality of Matlab to profile the runs. Matlab will then generate a comprehensive report of the runs, that would help to determine exactly the time consuming sections in the code.

TIPS: Note that it may take some time for one to master Matlab' Coder for every need.

Good Luck!

References for Further Reading

List of my select works related to DFNE.

Journal & Conference Papers

Fadakar-A Y, Elmo D, **2018***, Application of Graph Theory for Robust and Efficient Rock Bridge Analysis, DFNE2018 (June 2018), Seattle, USA (**submitted*).

Fadakar-A Y, Xu C, **2018***, A New Hybrid Mesh-Pipe Method for Modeling Fluid Flow in 3D Discrete Fracture Networks, DFNE2018 (June 2018), Seattle, USA (**submitted*).

Fadakar-A Y, **2017**, ADFNE: Open Source Software for Discrete Fracture Network Engineering, Two and Three Dimensional Applications. Journal of Computers and Geosciences, 102:1-11. DOI: http://dx.doi.org/10.1016/j.cageo.2017.02.002.

Fadakar-A Y, Elmo D, Eberhardt E, **2017**, Similarity Analysis of Discrete Fracture Networks. Geophysics, arXiv:1711.05257, https://arxiv.org/abs/1711.05257.

Fadakar-A Y, Dowd P.A, Xu C, **2014**, Connectivity Field: A Measure for Characterising Fracture Networks. Journal of Mathematical Geosciences, DOI: 10.1007/s11004-014-9520-7.

Fadakar-A Y, Dowd P.A, Xu C, **2013**, The RANSAC method for generating fracture networks from micro-seismic event data, J Mathematical Geosciences, DOI 10.1007/s11004-012-9439-9.

Fadakar-A Y, Dowd P.A, Xu C, **2013**, A Connectivity-Graph Approach to Optimising Well Locations in Geothermal Reservoirs, Australian Geothermal Energy Conference (AGEC2013), Brisbane, Australia.

Fadakar-A Y, Xu C, Dowd P.A, **2013**, Connectivity Index and Connectivity Field towards fluid flow in fracture-based geothermal reservoirs, in proceedings: 34th Stanford Geothermal Workshop Conference (SGW2013), Stanford, USA.

Fadakar-A Y, Dowd P.A, Xu C, **2012**, Application of connectivity measures in enhanced geothermal systems, in proceedings: Australian Geothermal Energy Conference (AGEC2012), Sydney, Australia.

Fadakar-A Y, Xu C, Dowd P.A, **2011**, A general framework for fracture intersection analysis: algorithms and practical applications, in proceedings: Australian Geothermal Energy Conference (AGEC2011), Melbourne, Australia.

Online Resources

ADFNE1.5	http://alghalandis.net/products/adfne/adfne15
ADFNE1.0	http://alghalandis.net/products/adfne
ADFNE2.0 Preview	http://alghalandis.net/note/adfne2
Applications	http://alghalandis.net/products/adfne/apps
Blog	http://alghalandis.net/blog
Notes	http://alghalandis.net/note
Publications	http://alghalandis.net/about/academia/publications
Presentations	http://alghalandis.net/about/academia/presentations

Other Useful Links

Products	http://alghalandis.net/products
Services	http://alghalandis.net/services
Rights	http://alghalandis.net/copyright-and-rights
Community Services	http://alghalandis.net/community
Contact	http://alghalandis.net/contact-us
About	http://alghalandis.net/about
CV	http://alghalandis.net/about/academia/curriculum-vitae