Installation - and user guide for FracSim2D: A new methodology for generating geological constrained 2D discrete fracture networks

Software developed by: Nico Hardebol (TUDelft and SODM)

Guide written by: Quinten Boersma (TUDelft)

FracSim2D, is a python-based algorithm which uses defined statistical and geological controls in order to stochastically populate 2D DFN's using an iterative workflow. The tool has been developed on top of Quantum GIS 2.4 (QGIS 2.4) and uses multiple functions taken from the QGIS 2.4 library. Further, FracSim2D runs on a windows operating system. In this document we will: 1) explain how to install the software, 2) explain the design and organization of the algorithm, 3) explain the structure the input file, 4) demonstrate the essential steps for running a simulation, 5) exemplify the algorithm by running a simulating of an orthogonal fracture network, and 6) provide a list of potential pitfalls of the current version of the FracSim2D.

Contents

1. Software installation

FracSim2D makes use of Quantum GIS libraries which needs to be pre-installed. This step by step Guide will explain how to install all necessary software's, unzip the FracSim2D package and change the FracSim2D batch file.

Step-1: Install QGIS-2.4 32 bits. **So not versions 2.6-3.0 nor 64bits!**

- You can find this older QGIS installer at http://qgis.org/downloads/
- Look for QGIS-OSGeo4W-2.4.0-1-Setup-x86.exe (Figure 1)
- Install the QGIS software
- Remember the file path of where you installed the QGIS software. In my case the standard installation path was C:\Program Files (x86)\QGIS Chugiak.
- QGIS 2.4 can be installed in parallel to another version QGIS. This implies that you can still run QGIS 3.0 while having QGIS 2.4 installed.

Step-2: Install notepad++

- You can find the notepad++ installer at https://notepad-plus-plus.org/downloads/
- You can download and install the software you prefer
- This software is necessary for changing the console batch within the FracSim2D folder

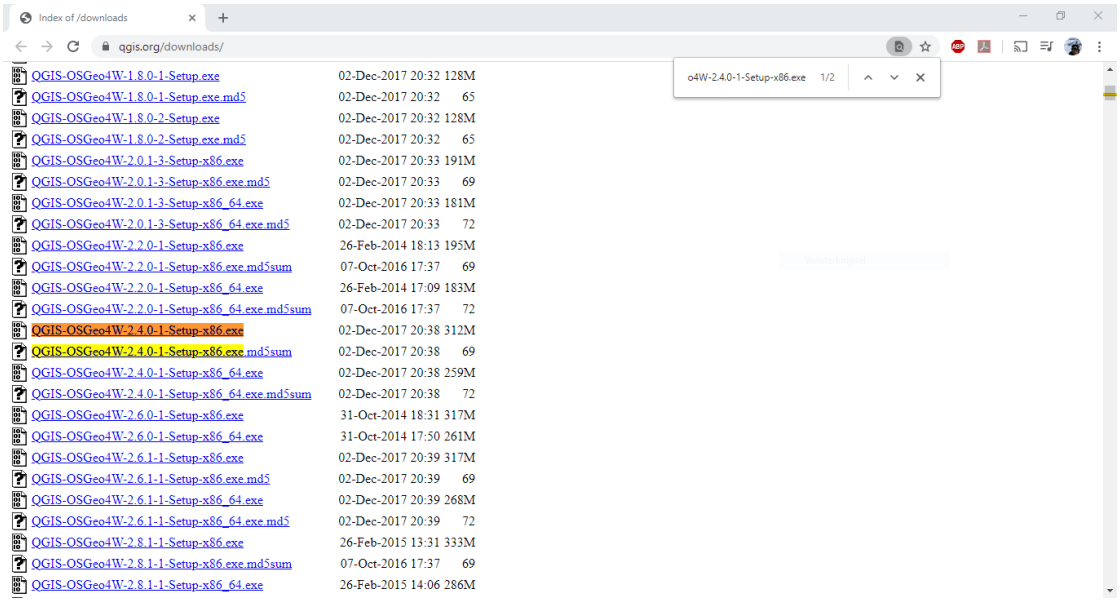


Figure 1: Finding the correct QGIS installer within the downloads folder

Step-3: Unzip the FracSim2D zip folder provided with this document

- Unzip the folder to a location of your choice

Step-4: Edit the reference path in the console batch file using notepad++

- Right click on the console.batch in your FracSim2D folder and select open with notepad++
- Change the file path for your QGIS installation. If you used the default QGIS installation settings you can use the following file path OSGEO4W_ROOT= C:\"Program Files (x86)"\"QGIS Chugiak" **(BEST NOT TO USE COPY-PASE) (BEST TO TYPE IT IN NOTEPAD++)** (Figure 2)
- If you have whitespaces (spacebar) in your file path you will need to use quotation marks (figure 2)

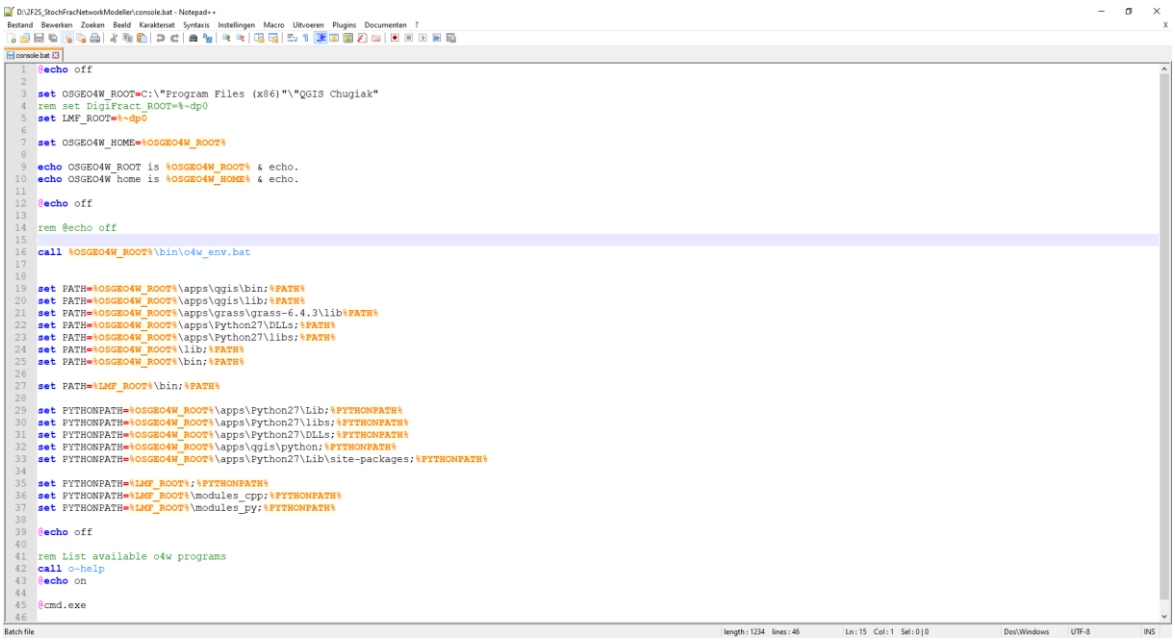


Figure 2: Changing the file-path using Notepad++

- Safe the file using the floppy disk icon
- You have now installed FracSim2D and are ready to run simulations

2. Algorithm design and organization

FracSim2D uses an iterative workflow to stochastically simulate 2D Discrete Fracture Networks (Figure 3). These simulations are controlled by user defined statistical distributions and geological rules (Figure 3). Different from other stochastic simulators the geological rules are regarded as more important than the assigned statistical distributions. This implies that resulting DFN's can deviate from the input statistics in order to acknowledge the assigned geological rules.

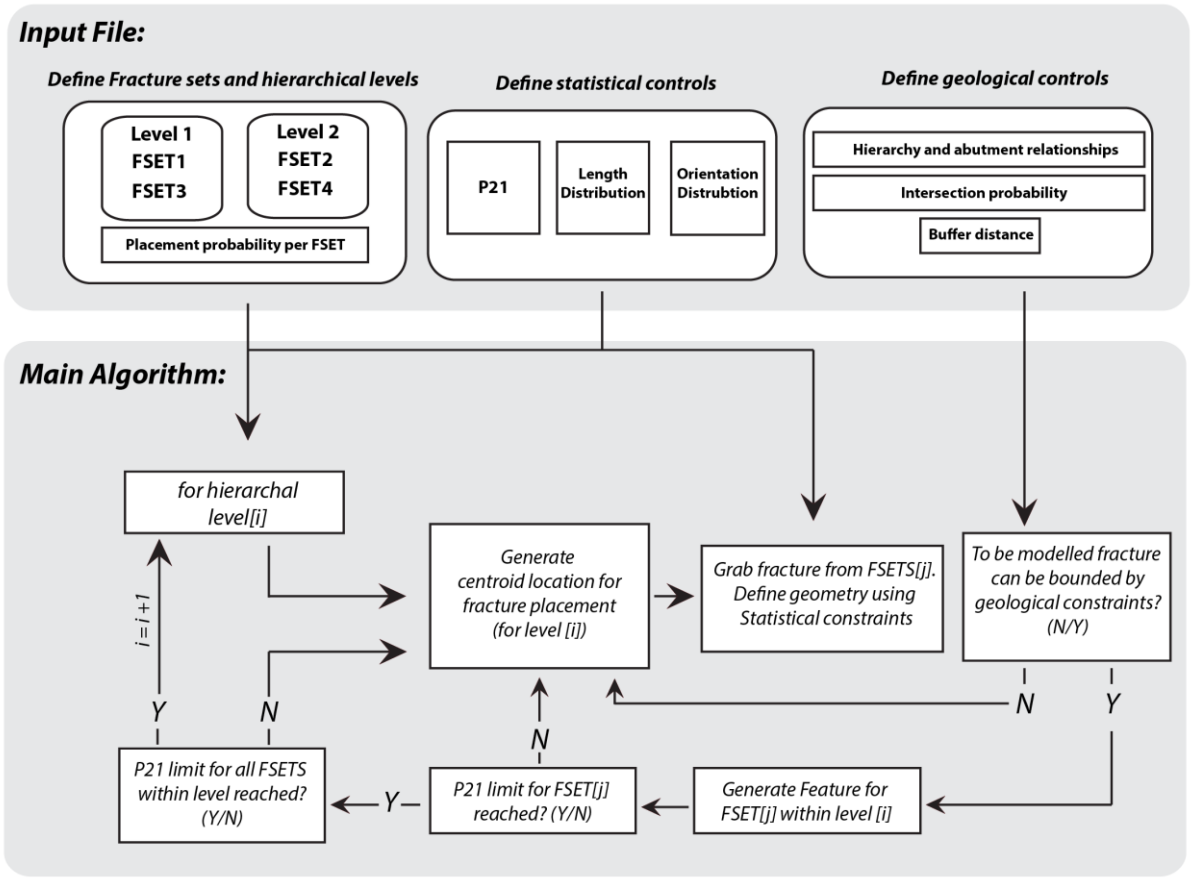


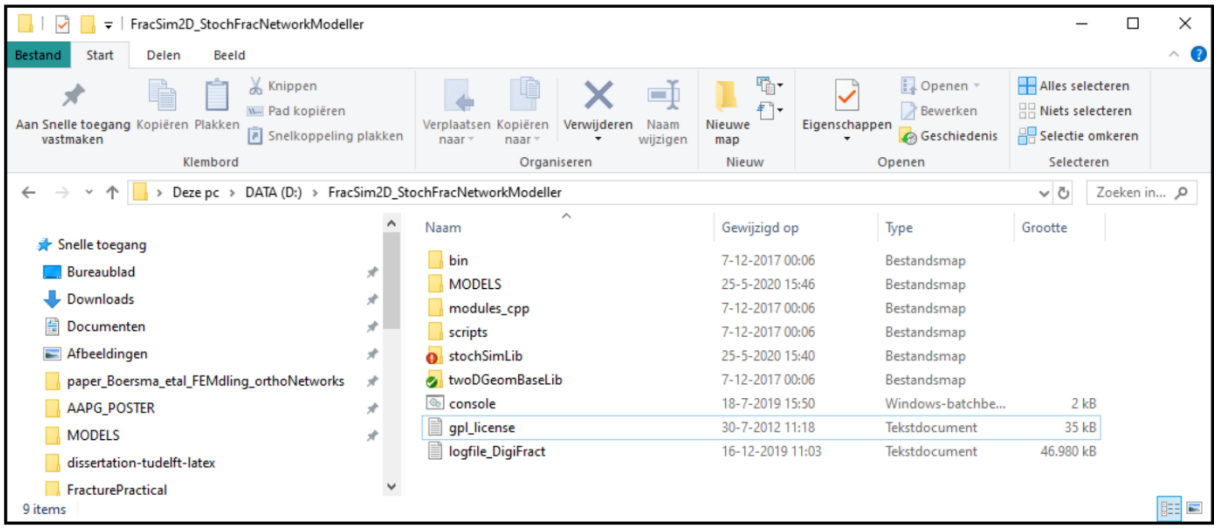
Figure 3: The workflow of the algorithm. This algorithm simulates a fracture network geometry for a number for fracture sets and levels, based on the assigned statistical (P21, Length distribution and the orientation distribution) and geological (Hierarchy, intersection probability and buffer distance) controls. This algorithm continues to place fractures until the P21 limit for each fracture set is reached.

Within FracSim2D, three main fracture network characteristics need to be defined in the input file, namely: 1) fracture levels, sets and placement probabilities, 2) statistical distributions and controls (e.g. fracture intensity (P21) and length – and orientation distributions) and 3) geological controls (i.e. hierarchy, placement constraints and buffer distances) (Figure 3). The algorithm uses these characteristics to iteratively create a 2D DFN.

The main iterator in FracSim2D are the fracture level(s) (hierarchical order). In the flow chart shown in figure 3, fractures belong to sets 1 and 3 will be placed using the assigned statistical and geological controls. Once the P21 limit for the fracture sets within level 1 are reached, the algorithm starts to populate fracture sets belonging the second hierarchical level (i.e. FSET2 and FSET 4) (Figure 3). Again, this is done using the assigned geological and statistical controls and continues until the assigned P21 limits are reached (Figure 3). The simulation is finished once all fracture sets have reached their P21 limit (Figure 3).

FracSim2D is based on a set of python scripts and folders structures which organized as follows: 1) the main algorithm, 2) background functions, 3) input python files and 4) model output folders (Figures 4 and 5). The main python script is stored under the folder scripts (Figure 4). In order to run a simulation this script needs to be run inside the console batch file (see section 4)

Main Folder



Main Algorithm

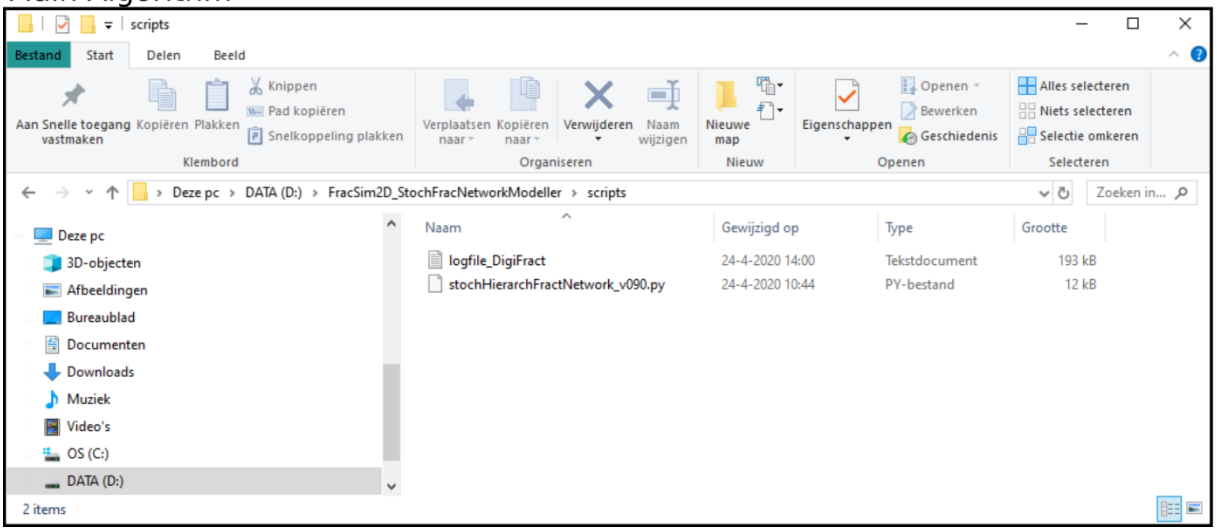
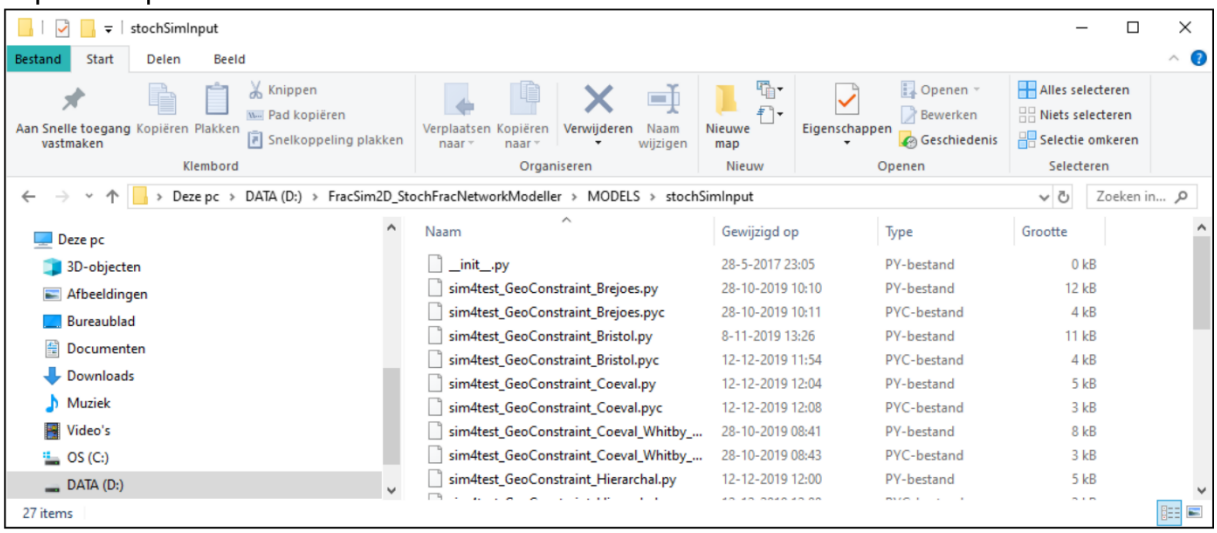


Figure 4: Folder structure of FracSim2D and location of the main python script

Input scrips



Model Output folders

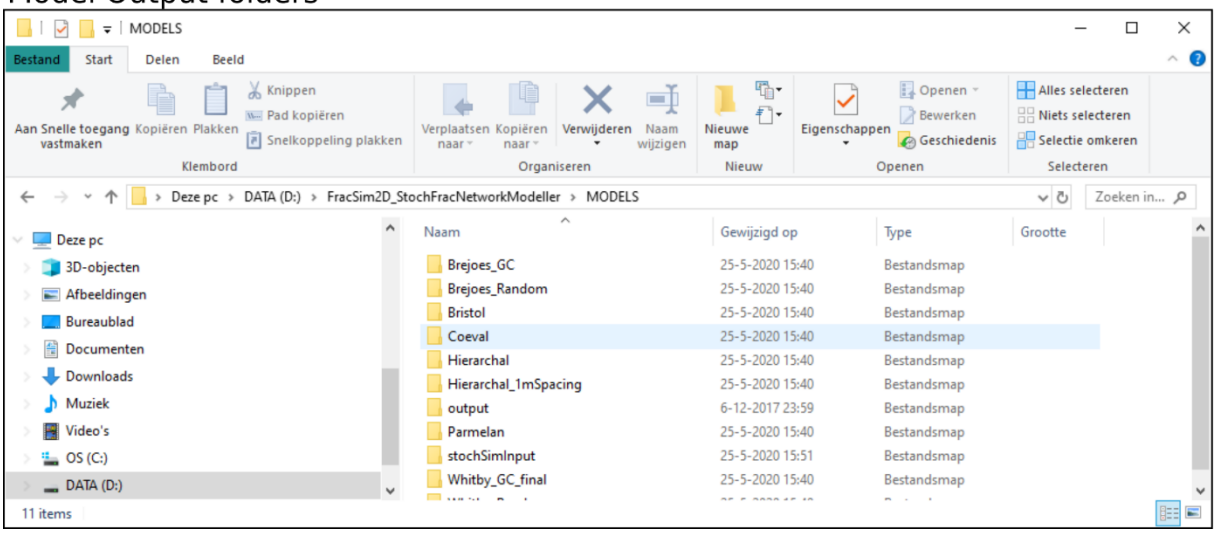


Figure 5: Top) Location of the python model input scripts. Bottom) Output folders of FracSim2D. In these folders, the produced shapefiles are stored.

3. The input file

In order to generate different DFN's, the user has to define a model input file. In this file the user can define the characteristics of the DFN such as, 1) the modelling domain and 2) different fracture characteristics including fracture sets and levels, length distributions, orientation distributions, hierarchy, intersection probability and spacing (Figure 6). FracSim2D reads the input file, and together with multiple function libraries, a DFN is iteratively generated (Figures 1 and 6). The main algorithm and background functions do not need to be changed by the user. In the following we will explain the organization of the input file and describe the different geological - and statistical controls.

The input file should be organized as follows (e.g. figure 6):

1. Define the model boundary and epsg code. This can be done by adding lines: 1) `epsgCode = 28992` and 2) `surfBoundGeomWkt = "POLYGON (( -5.0 -5.0, \ -5.0 5.0, 5.0 5.0, \ 5.0 -5.0 , -5.0 -5.0 ))"` (Figure 6)
2. Add a fracture set in the main class within the input file. This can be done by adding a line: `self.add_simFractSet_n('L1-NS',generator_type='randomFracture', modelDomain= myDataSpace.domainBound )` (Figure 6). Here you can also define

the generator type, which determines how the respective fracture set will be populated with respect to other fracture sets.

3. Add a fracture level to each of the fracture sets within your model. This can be done by adding a line: `self.add_Level('L1', ['L1-NS'], [1.0])` (Figure 6). Here you can also define the placement probabilities of each set within the defined level.
4. Define the fracture intensity for each defined fracture set. This can be done by adding a line: `self.set_P21Limit('L1-NS', 1.0)` (Figure 6).
5. Define the length distribution for each defined fracture set. This can be done by adding a line: `self.fractSets[0].generator.set_sizeGenerator('fixed',[2.5])` (Figure 6). Here, it should be noted that the first defined fracture set (e.g. L1-NS (Figure 6)) corresponds to [0] within the algorithm. The fracture set which is defined second will be referred to as [1], etc. See Figure 12 for an example of an input file with multiple fracture sets and levels.
6. Define the orientation distribution for each defined fracture set. This can be done by adding a line: `self.fractSets[0].generator.set_orientGenerator('vonmises', [numpy.radians(0.0), 10.0, None , 1])` (Figure 6). Again, it should be noted that the first defined fracture set (e.g. L1-NS (Figure 6)) corresponds to [0] within the algorithm. See Figure 12 for an example of an input file with multiple fracture sets and levels.
7. Define the buffer distance for each defined fracture set. This can be done by adding a line: `self.fractSets[0].generator.set_buffer_dist(0.5)` (Figure 6). If you want this buffer distance activated you should set the proximity condition to 'True': `self.fractSets[0].generator.set_proximity_condition(True)`. Again, it should be noted that the first defined fracture set (e.g. L1-NS (Figure 6)) corresponds to [0] within the algorithm. See Figure 12 for an example of an input file with multiple fracture sets and levels.
8. Define the non-intersecting fracture sets. This can be done by adding a line: `self.fractSets[0].generator.set_nonIntersecting_fractSets(['L1-NS'], [1.0], [1.0])` (Figure 6). This line defines which fractures sets the to be placed fracture set cannot intersect. Again, it should be noted that the first defined fracture set (e.g. L1-NS (Figure 6)) corresponds to [0] within the algorithm. See Figure 12 for an example of an input file with multiple fracture sets and levels.

Finally, for each of the input lines, additional information is given by the blocked comments behind the respective input lines within the provided explementary input files (e.g. Figure 6).

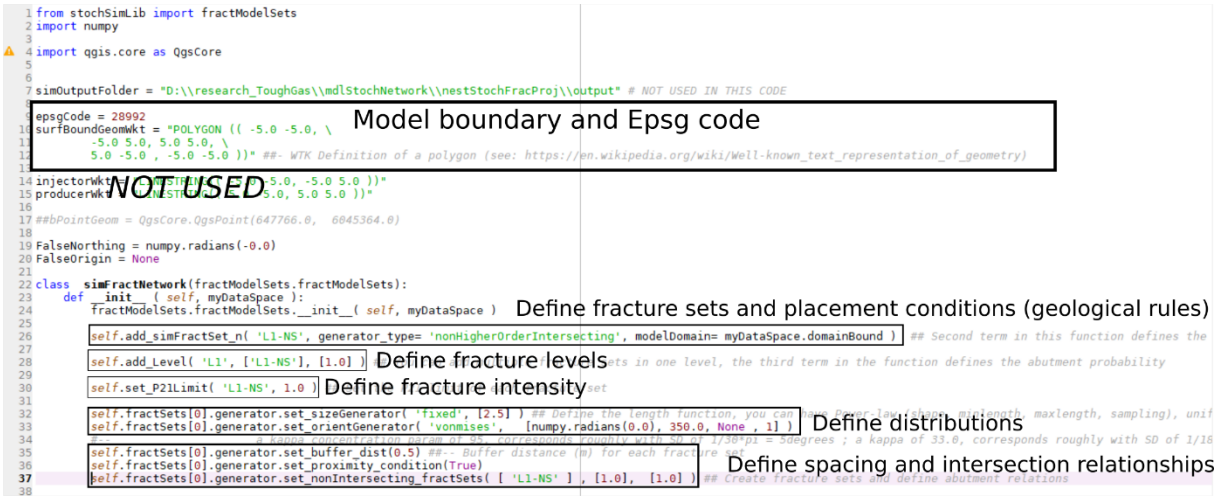


Figure 6: General organization of a FracSim2D input file: 1) Define model boundary (WTK description) and epsg code, 2) define fracture sets and selection fracture generators, 3) define the fracture levels and sub-divide the fracture sets, 4) set the fracture intensity for each set, 5) define the statistical distributions and 6) set spacing and intersection relationships.

3.1. The modelling domain and EPSG code (Lines 9-12, Figure 6)

The modelling domain defines the size of your DFN and is defined using a WTK description (Polygon). The EPSG code defines the type of CRS used by the algorithm and by default it is set at RD-Netherlands-New (EPSG: 28992) (Figure 6).

3.2. Adding a fracture set and defining placement constraints (generator types) (Line 26, Figure 6)

Within FracSim2D different fracture sets can be added to a simulation and in order to generate fracture network geometries which show organization and hierarchy, placement constraints can be added to the model (Figure 3 and 6)

Four different placement constraints can be assigned, namely:

1) Random (in the code defined as `'randomFracture'`) (no geological controls)
2) No Self Intersect (NSI) (in the code defined as `'noSelfIntersecting'`) (fractures belonging to the same set cannot intersect)
3) No Equal Level Fracture Intersect (NELFI) (in the code defined as `'noEqualLvlIntersecting'`) (Fractures belonging to the same level cannot intersect each other)
4) No Higher Order Fracture Intersect (NHOFI) (in the code defined as `'nonHigherOrderIntersecting'`) (fractures belonging to this level cannot intersect fractures of a higher level)
5) Place at Higher Order Fracture (PHOF) (in the code defined as `'placedAtHOrderFract'`) (explicitly place fractures to fractures belonging to a higher order)

These different placement constraints are exemplified by Figure 7. See the caption for more information on the simulation settings.

Finally, it should be noted that FracSim2D assumes that geological constraints are more important that the statistical distributions. This implies that by assigning different placement constraints (shown above), the resulting fracture could have lengths and/or orientations which differ from the input distributions. This is also shown by figure 7, where the assigned length distributions were set at fixed and 5.0 m. However, due to the implemented placement constraints, modelled fracture lengths differ from the assigned input value (i.e. the fracture lengths are clipped in order to honor the placement constraint). More information on the impact of placement constraints will be given in section 5.

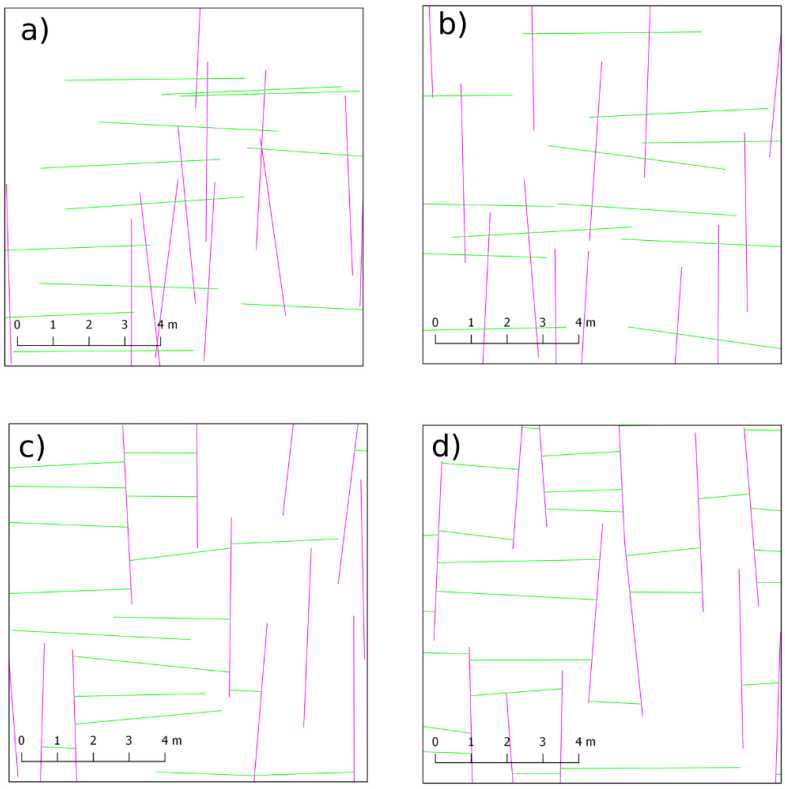


Figure 7: Different placement constraints. All network realizations had a fixed length distribution (l = 5.0m) and two fracture sets (FSET1 (N-S) (Pink) and FSET2 (E-W) (Green)). Both fracture sets have a von-mises orientation distribution (FSET1: μ = 0.0, κ = 250, FSET2: μ = 90.0, κ = 250) and have a P21 limit of 0.5 (1/m). The buffer distance was set at 0.5m. a) Random fracture placement. b) No self-intersect constraint. c) No higher order fracture intersect constraint. d) Place at higher order fracture. See text for additional information on the assigned parameters.

3.3. Fracture sets and levels (Line 28 Figure 6)

Within FracSim2D, fracture sets describe a group of fractures which have a shared orientation. Fracture levels describe the hierarchy of the network, and within this tool multiple fracture sets can belong to one level. For each set within a level, a placement probability needs to be defined (Figures 3 and 6). This can be done by changing the number within the square brackets, which in this example is set at 1.0 for the sole fracture set within the level (Figures 6, line 28). Finally, an example of a simulation which had distinct fracture levels and sets will be shown later in this guide.

3.4. Fracture intensity (Line 30, Figure 6)

The fracture intensity (P21) limit describes the maximum cumulative trace length of fractures within a set over the modelling area (Dershowitz & Herda, 1992; Sanderson & Nixon, 2015) (Figure 6). Within FracSim2D, this limit is a single scalar value defined by the user. The P21 should be assigned for each fracture set present in your simulation. A P21 distribution or map cannot be implemented in the current version of the algorithm. The impact of assigning different fracture intensity values is illustrated by Figure 8, where the assigned P21 values are 0.1, 1.0 and 5.0, respectively.

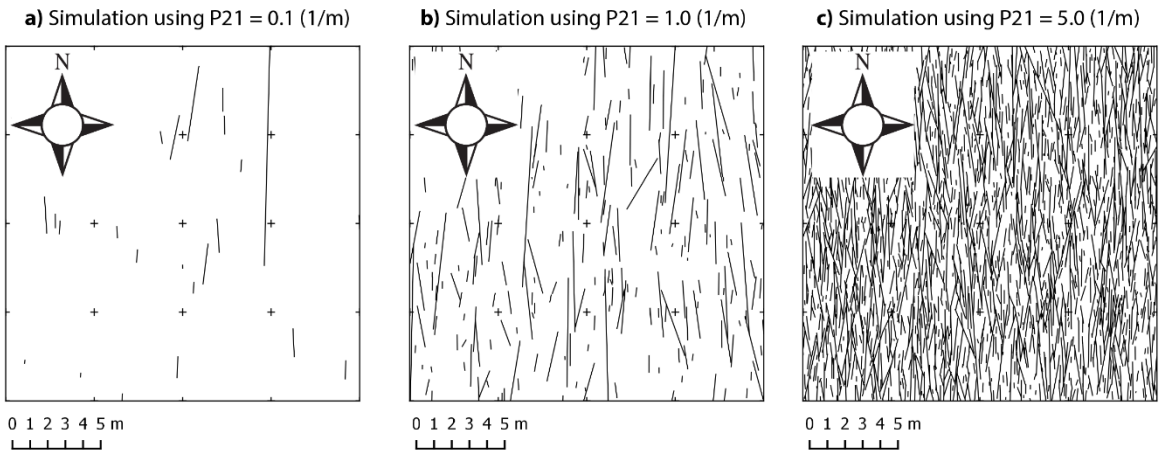


Figure 8: Different realization having a maximum assigned fracture intensity (P21) of 0.1 (1/m) (a), 1.0 (1/m) (d) and 5.0 (1/m) c), respectively. For all figures, the length and orientation distribution are identical and set at: A Log-normal length distribution (μ=1.0, σ =1.0) and a Von-Mises orientation distribution (μ = 0.0, κ= 50). See text for additional information on the assigned parameters.

3.5. Fracture length distributions (Line 32, Figure 6)

Within this algorithm, the user can choose from five different length distributions, namely:

1) Fixed length, in code: `'fixed'`
2) Uniform distribution, in code: `'uniform'`
3) Log-Normal distribution, in code: `'lognorm'`
4) Negative power-law distribution, in code: `'powerlaw'`
5) Negative exponential distribution, in code: `'expon'`

For the fixed length and uniform distribution, the algorithm aims to generate 1) fractures having the assigned length (e.g. 10m), and 2) fractures which are uniformly distributed in between a user defined minimum and maximum length, respectively (Figure 9a-b).

The log-normal distribution uses the log-normal function taken from the scipy (python) library (Figure 9b), so that the length distribution can be defined as follows:

$$P(\ln l;\ \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[\frac{(\ln l - \mu)^2}{2\sigma^2}\right], l >\ 0 \quad (1)$$

Here, l is the fracture length (m), σ is the shape parameter (standard deviation of the logarithm of the variable) and μ is the scale parameter (i.e. $\text{Median} = \exp(\mu)$) of the function, respectively (Figure 9b).

For the negative power-law distribution we use a python package called power-law (Alstott, Bullmore, & Plenz, 2014), so that the length population is defined by equation 2:

$$P(l; a) = a \cdot l^{-a} \quad (2)$$

Here, l is the fracture length (m) and a is the power-law exponent and scaling parameter (Figure 9c).

Finally, for the exponential distribution (Figure 9d), we use the exponential function within the python scipy library so that the length distribution can be defined by equation 3:

$$P(l; \lambda) =\ \lambda e^{-\lambda l} \quad (3)$$

Where l is the fracture length (m) and λ is the shape parameter (in the scipy function, $\text{shape} = \text{scale} = 1/\lambda$). Further, a loc parameter is added in the function, which can be used to additionally shift the exponential function.

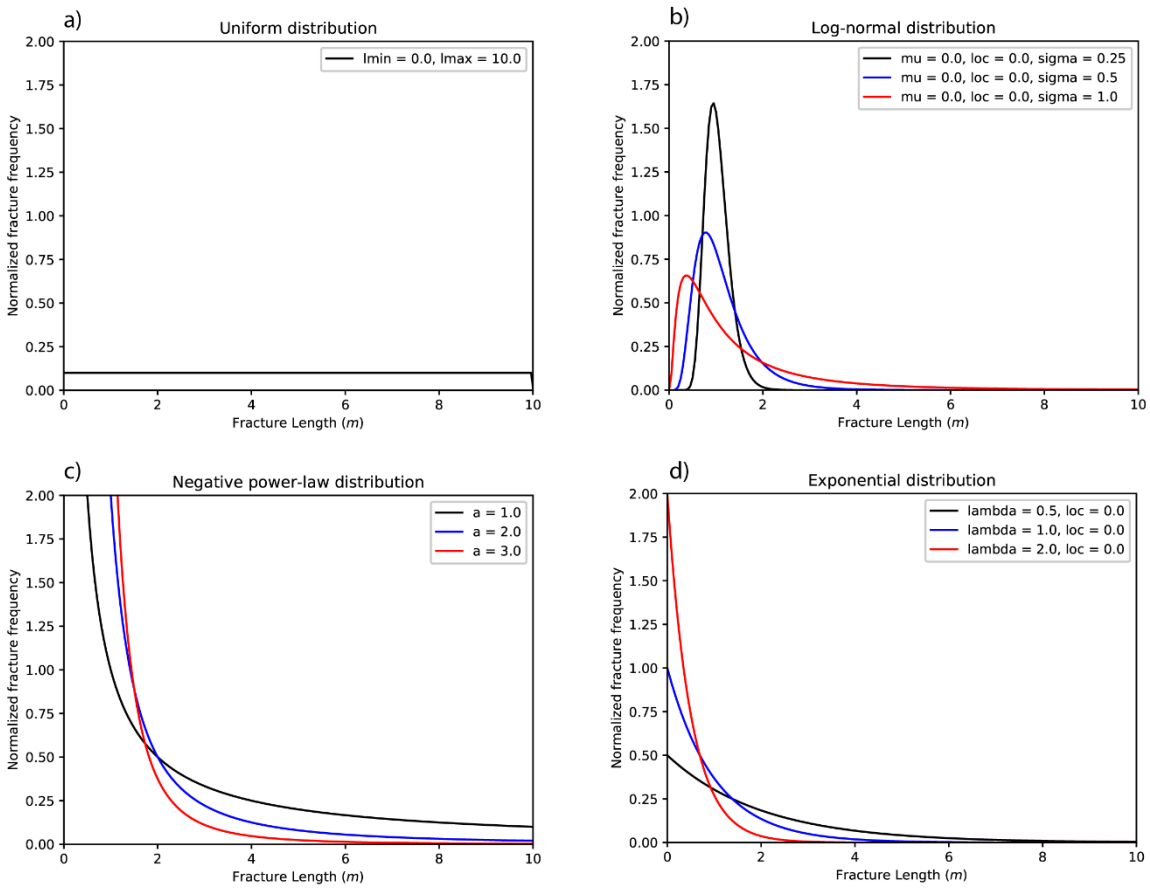


Figure 9: Different length distributions used within the algorithm. a) Uniform length distribution. b) Different log-normal length distributions. c) Different negative power-law length distributions. d) Different exponential distributions.

3.6. Fracture orientation distributions (Line 33, Figure 6)

Two different orientation distributions can be chosen, namely: 1) uniform (in code: `'uniform'`) and 2) Von-Mises (in code: `'vonmises'`). For the uniform distribution, the user assigns a minimum and maximum value, and the algorithm populates fractures which have an angle ranging in between the two defined values. For the Von-Mises distribution we also use a scipy package so that fractures are normally (i.e. Gaussian distribution) populated around a user defined mean angle (equation 4) (Figure 10):

$$P(\theta;\ \mu, \kappa) = \frac{e^{\kappa \cos(\theta - \mu)}}{2\pi I_0(\kappa)} \quad (4)$$

where, θ is the fracture orientation (radians) (in this algorithm strike), μ is the user defined mean orientation and $1/\kappa$ is analogues to the variance of the orientation distribution (σ^2) (Figure 10). $I_0(\kappa)$ is the modified Bessel function of the zeroed order.

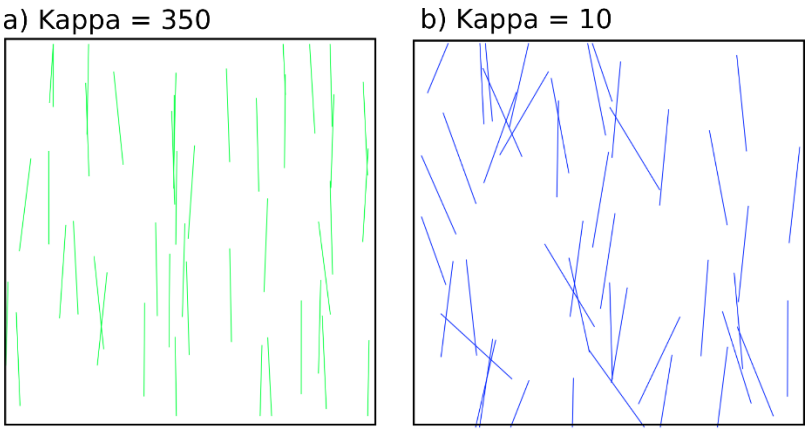


Figure 10: a-b) Random DFN simulations having a Von-Mises distribution with a κ of 350 and 10, respectively.

3.7. The buffer distance and non-intersecting fractures (Lines 35-37, Figure 6)

The fracture spacing is implemented by assigning a specific buffer distance to a fracture set. If activated, the algorithm tries to place fractures belonging to the same set at a spacing which is bigger than the defined buffer distance. For example, figure 11a shows a simulation with an assigned buffer distance of 0.1m whereas the simulation shown by figure 11b, has a buffer distance of 0.5m. In FracSim2D you can also assign non-intersecting fracture sets (Figure 6). By doing so, fractures belonging to the to be placed fracture set will not intersect the FSET defined by the user.

While showing promising results, it should be noted that some minor deviation from the assigned buffer distance can occur. This problem occurs because FracSim2D uses the central node of a to be placed fracture to calculate the relative spacing between the different fractures. In addition, it should be noted that assigning strict spacing relationships between fractures results in simulations which may take a long time to compute. Finally, it should be noted that when the random fracture placement generator type is assigned, no buffer distances or non-intersecting fracture sets can be assigned.

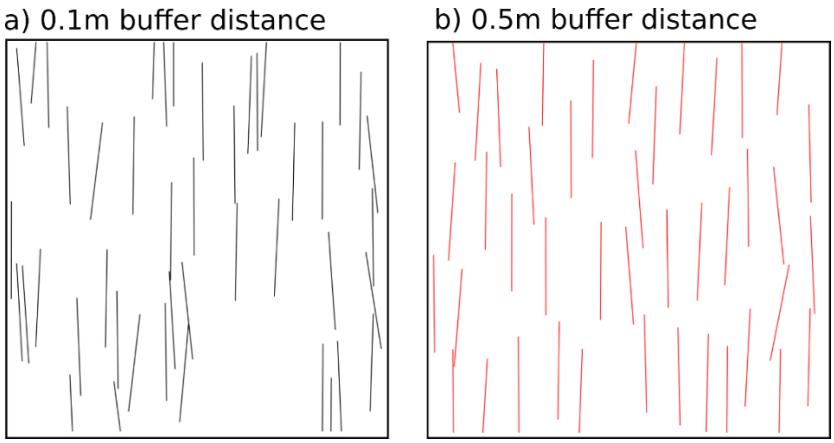


Figure 11: Changing the assigned buffer distance. Both network realizations had a fixed length distribution (l = 2.5m) and one fracture set (FSET1 (N-S) which has a von-mises orientation distribution (μ = 0.0, κ = 350). The P21 limit was set at 1.0 (1/m). Finally, the fracture placement constraint was set at no self intersect. a) Buffer distance = 0.1m. b) Buffer distance = 0.5m.

4. Steps for running a simulation

In the following, we will explain the essential steps you will need to undertake in order to run a simulation

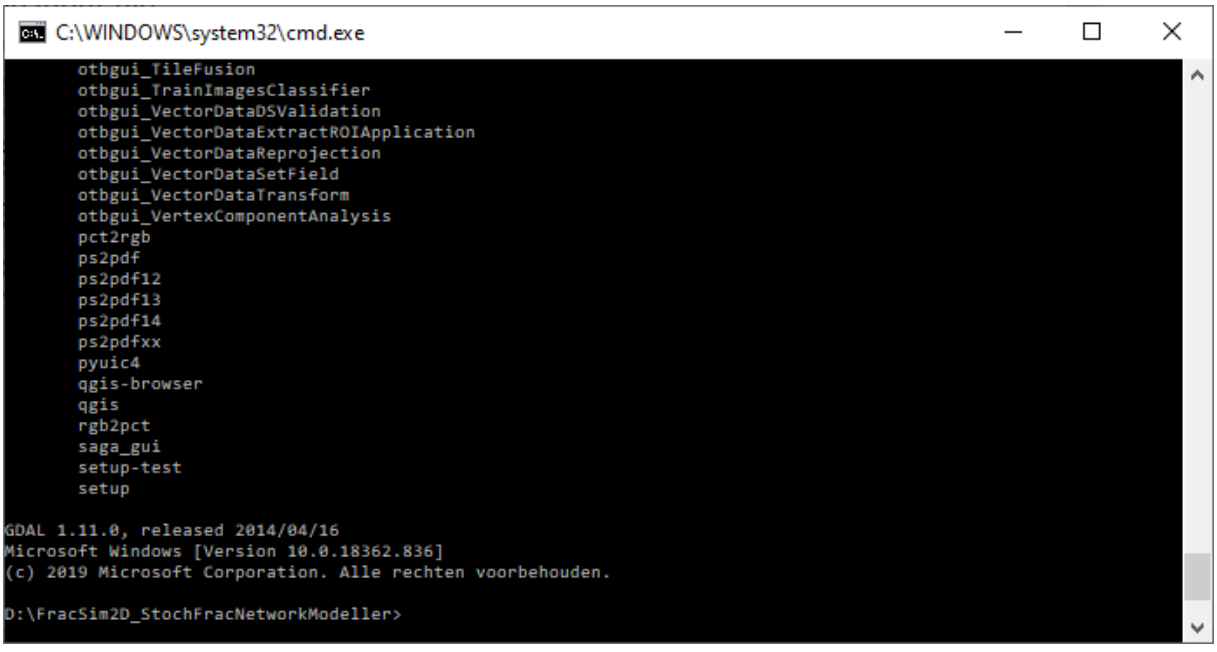
- Step 1: create the input file (section 3)

- Step 2: create the output folder (e.g. Figure 5)
- Step 3 change the file-paths in the main script (see figure below)

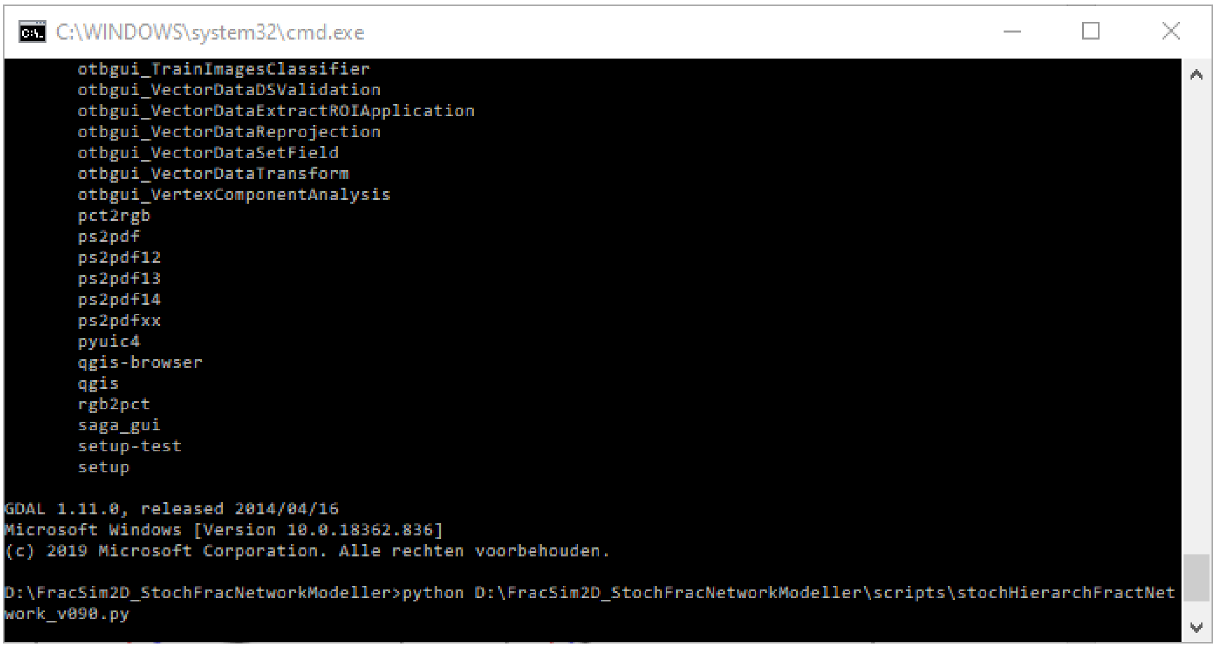
```
import os, sys,  imp
import logging

logger = logging.getLogger("DigiFractLogger")
logger.setLevel(logging.DEBUG)
#logger.setLevel(logging.INFO)

logFilePath = os.path.join( os.getcwd(),"logfile_DigiFract.log" )
if os.path.isfile(logFilePath): os.remove(logFilePath)

log2fileHandle = logging.FileHandler( logFilePath ,  mode='a' )
#log2fileHandle.setLevel(logging.DEBUG)
log2fileHandle.setLevel(logging.DEBUG)
##formatter = logging.Formatter('%(message)s')
formatter = logging.Formatter('%(asctime)s | %(name)s|%(levelname)s: %(message)s')
log2fileHandle.setFormatter(formatter)
logger.addHandler(log2fileHandle)

import numpy
from osgeo import ogr

from PyQt4 import QtCore
import qgis.core as QgsCore

QGIS_basePath = os.path.normpath( os.path.join( os.environ['OSGEO4W_ROOT'] , "apps\\qgis" ) )
QgsCore.QgsApplication.setPrefixPath( QGIS_basePath , True)
QgsCore.QgsApplication.initQgis()

from twoDGeomBaseLib.core import geomBasicOperations
from twoDGeomBaseLib.core import connectivityOperations

from stochSimLib import fractModelSets
from stochSimLib import StoSim_algorithms
#-- set simBaseFolder
simBaseFolder       = os.path.normpath("D:\\FracSim2D_StochFracNetworkModeller\\MODELS")
sys.path.insert(0 , os.path.join( os.path.join( simBaseFolder, "stochSimInput" ) ) )

simulationName    = "Step_By_Step_Simulation"
CfgFracMdlFileNm   = "sim4test_GeoConstraint_StepByStep"
print "sys.path: %s" % sys.path
mdlCgf = __import__( CfgFracMdlFileNm )

simoutputfolder = os.path.join( mdlCgf.simOutputFolder ,  simulationName )

translate_nd_rotate_domain_aligned_around_origin = False
translate_nd_rotate_fractstochparams                    = False
gen_intersection_points = False

_activeOutcropBoundIdx = 0
```

Location of the input python file and model output folder

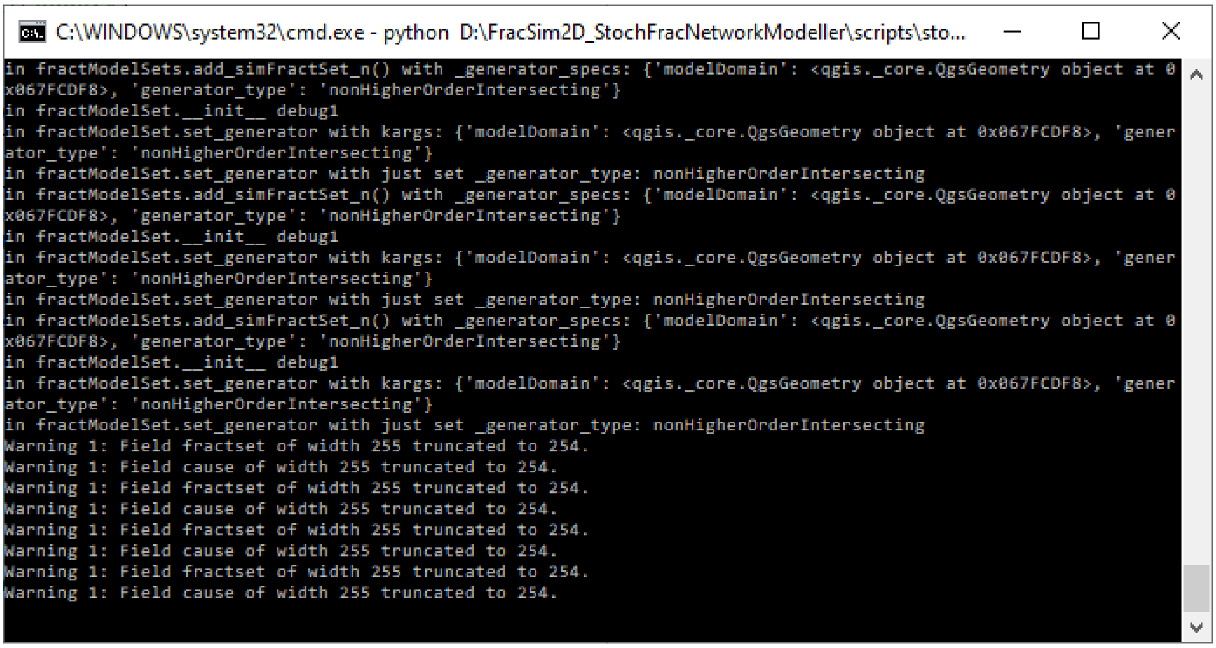
- Step 4: open the console. The console can be found in the main folder (Figure 4).



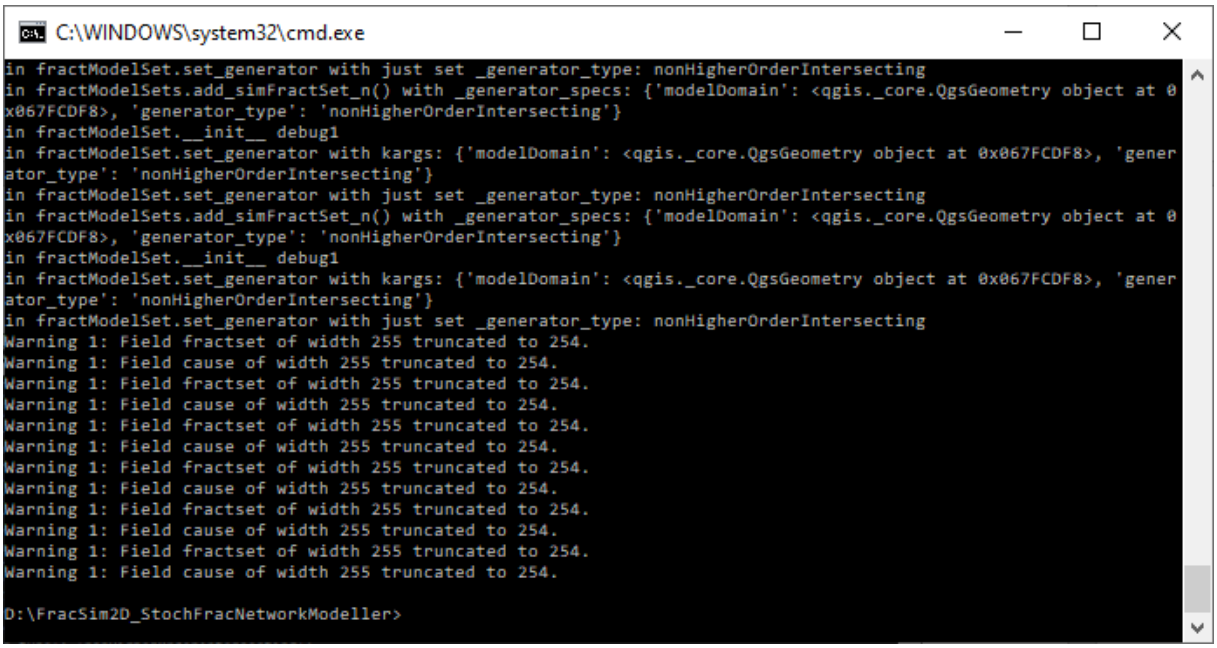
- Step 5: run the main algorithm (tip: type Python and drag and drop the script into the console) (see figure below).



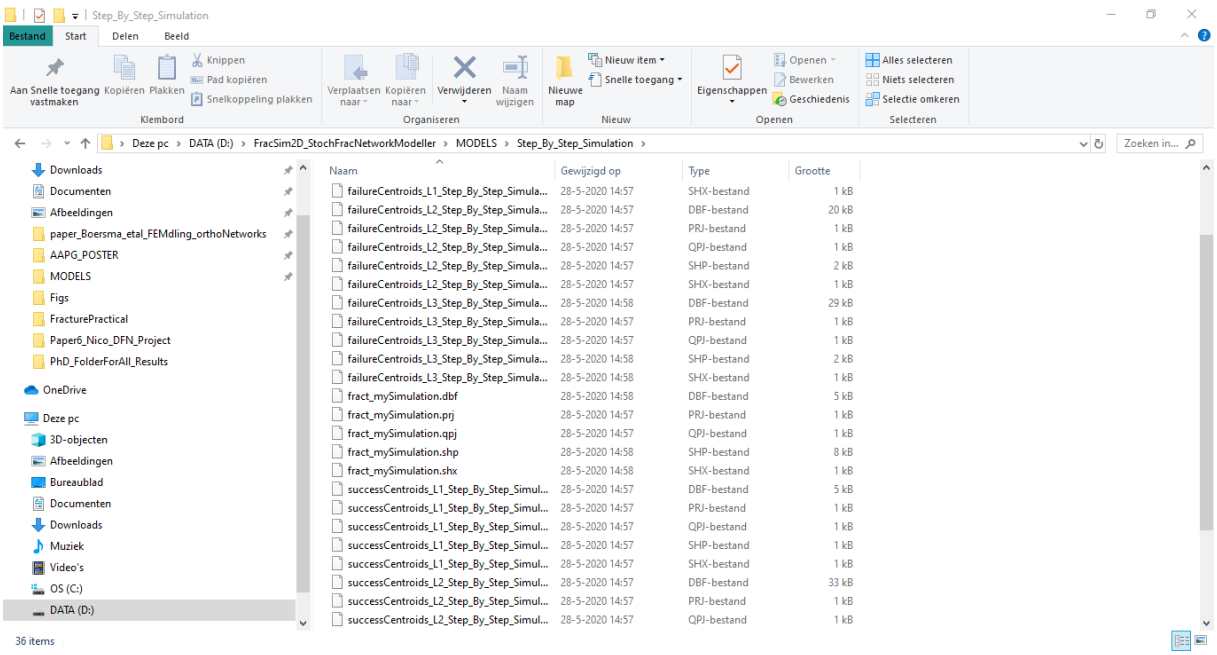
- Step 6: run the simulator by pressing enter (see figure below). If there is an error in your simulations this will be highlighted in the console.



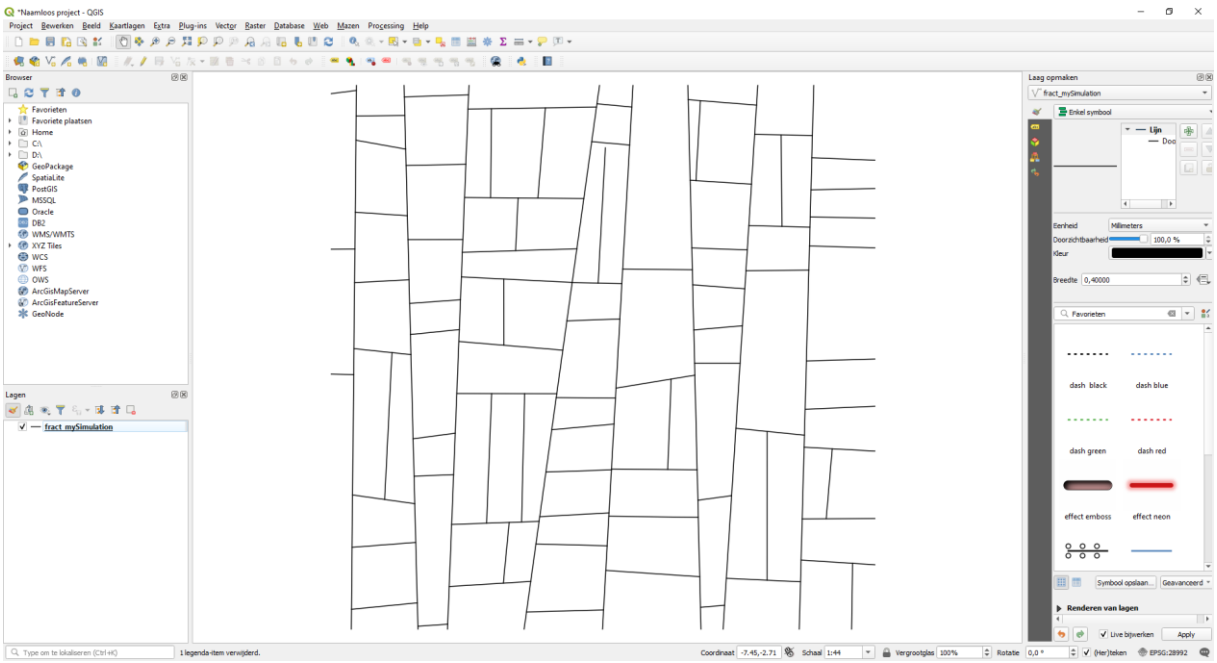
- Step 7: the simulation is finished once you can enter a new line (see figure below). Depending on the assigned conditions, this may take a while.



- Step 8: the results (shapefiles) are stored in the output folder (see figure below).



- Step 9: The simulation results (fract_mySimulation.shp) are shapefiles which can be opened in GIS based programs (e.g. QGIS).



- Step 10 **(optional):** If for any reason you want to re-run the simulation (e.g. did not converge or results not as expected), the created shapefiles will have to be deleted prior to running because FracSim2D does not overwrite previously generated shapefiles. This can be done in the windows file explorer.

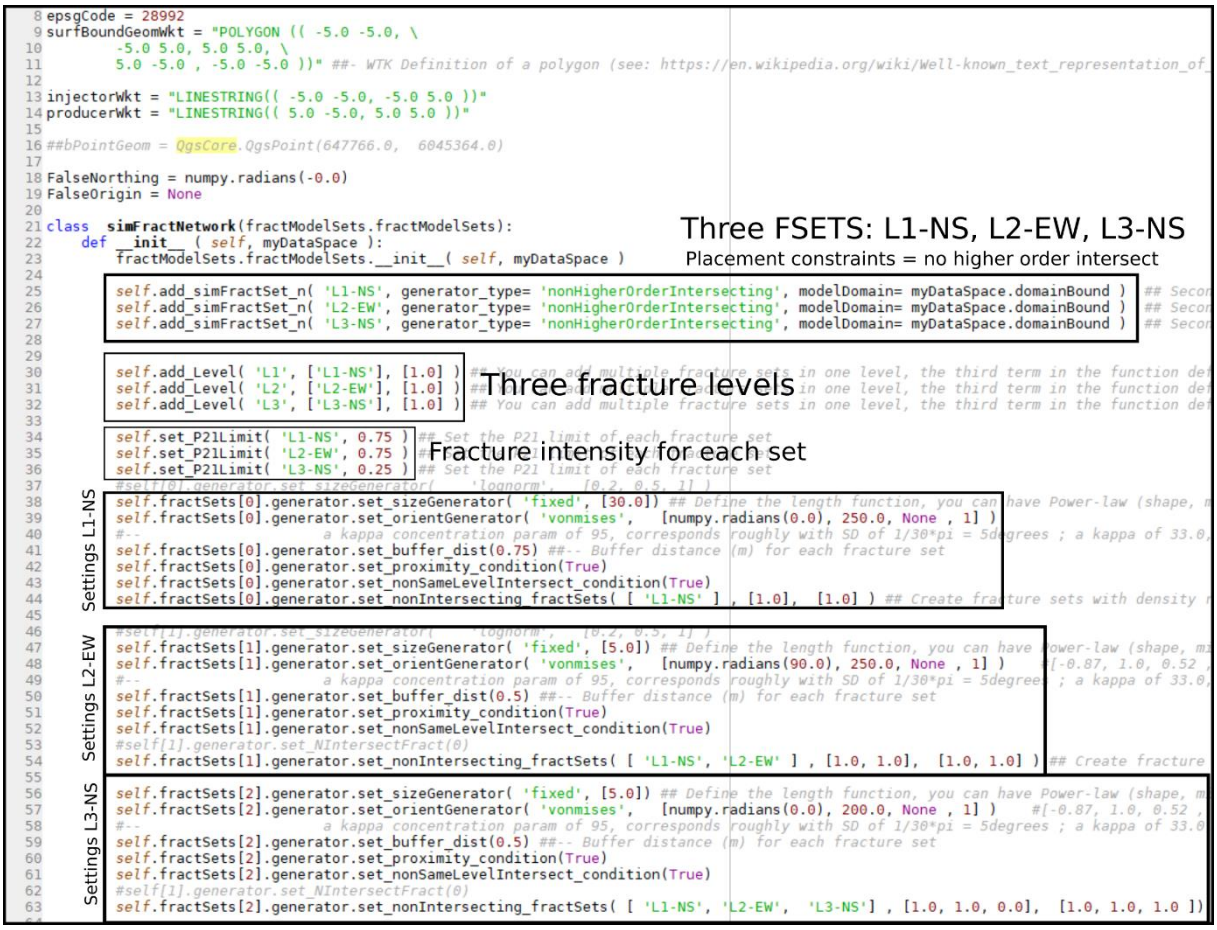
5. Example of an FracSim2D simulation: Orthogonal fracture network

To illustrate how the algorithm stochastically generates a 2D DFN (Figure 3), we show a simulation with distinct geological and statistical controls. The respective input file and resulting shapefiles have been provided with this document (Figures 12 and 13).

For this simulation we assume that three fracture sets orientated N-S or E-W are present. Further, we assume that the different fracture sets can be sub-divided into three levels and that a distinct hierarchy is present. To do this we explicitly tell the algorithm that fractures of a lower order cannot intersect fractures belonging to a higher order, by assigning the placement control: No Higher Order Fracture Intersect (NHFI) (see section 3.2.) (Table 1, Figures 12-13). Additionally, a fixed fracture spacing was assigned for all fracture sets (Table 1).

FSET	Fracture level	P21 limit	Length distribution	Buffer distance (m)	Orientation distribution	Placement constraint
FSET1	1	0.75	Fixed: 30.0m	0.75	VM: $\mu = 0$, $\kappa = 250$	NHFI
FSET2	2	0.75	Fixed: 5.0m	0.75	VM: $\mu = 90$, $\kappa = 250$	NHFI
FSET3	3	0.25	Fixed: 5.0m	0.5	VM: $\mu = 0$, $\kappa = 250$	NHFI
VM = Von Mises NHFI = Non higher order fracture intersection						

Table 1: Input parameters for the step by step modelling simulation. See section 3 for more information regarding the input parameters.




```
epsgCode = 28992
surfBoundGeomWkt = "POLYGON (( -5.0 -5.0, \
        -5.0 5.0, 5.0 5.0, \
        5.0 -5.0 , -5.0 -5.0 ))" ##- WTK Definition of a polygon (see: https://en.wikipedia.org/wiki/Well-known_text_representation_of_

injectorWkt = "LINESTRING(( -5.0 -5.0, -5.0 5.0 ))"
producerWkt = "LINESTRING(( 5.0 -5.0, 5.0 5.0 ))"

##bPointGeom = QgsCore.QgsPoint(647766.0,  6045364.0)

FalseNorthing = numpy.radians(-0.0)
FalseOrigin = None

class  simFractNetwork(fractModelSets.fractModelSets):
    def __init__ ( self, myDataSpace ):
        fractModelSets.fractModelSets.__init__( self, myDataSpace )

        self.add_simFractSet_n( 'L1-NS', generator_type= 'nonHigherOrderIntersecting', modelDomain= myDataSpace.domainBound ) ## Secon
        self.add_simFractSet_n( 'L2-EW', generator_type= 'nonHigherOrderIntersecting', modelDomain= myDataSpace.domainBound ) ## Secon
        self.add_simFractSet_n( 'L3-NS', generator_type= 'nonHigherOrderIntersecting', modelDomain= myDataSpace.domainBound ) ## Secon

        self.add_Level( 'L1', ['L1-NS'], [1.0] ) ## You can add multiple fracture sets in one level, the third term in the function def
        self.add_Level( 'L2', ['L2-EW'], [1.0] ) ## You can add multiple fracture sets in one level, the third term in the function def
        self.add_Level( 'L3', ['L3-NS'], [1.0] ) ## You can add multiple fracture sets in one level, the third term in the function def

        self.set_P21Limit( 'L1-NS', 0.75 ) ## Set the P21 limit of each fracture set
        self.set_P21Limit( 'L2-EW', 0.75 ) ## Set the P21 limit of each fracture set
        self.set_P21Limit( 'L3-NS', 0.25 ) ## Set the P21 limit of each fracture set
        #self[0].generator.set_sizeGenerator(    'lognorm',    [0.2, 0.5, 1] )
        self.fractSets[0].generator.set_sizeGenerator( 'fixed', [30.0]) ## Define the length function, you can have Power-law (shape, m
        self.fractSets[0].generator.set_orientGenerator( 'vonmises',   [numpy.radians(0.0), 250.0, None , 1] )
        #--                      a kappa concentration param of 95, corresponds roughly with SD of 1/30*pi = 5degrees ; a kappa of 33.0,
        self.fractSets[0].generator.set_buffer_dist(0.75) ##-- Buffer distance (m) for each fracture set
        self.fractSets[0].generator.set_proximity_condition(True)
        self.fractSets[0].generator.set_nonSameLevelIntersect_condition(True)
        self.fractSets[0].generator.set_nonIntersecting_fractSets( [ 'L1-NS' ] , [1.0],  [1.0] ) ## Create fracture sets with density m
        #self[1].generator.set_sizeGenerator(    'lognorm',    [0.2, 0.5, 1] )
        self.fractSets[1].generator.set_sizeGenerator( 'fixed', [5.0]) ## Define the length function, you can have Power-law (shape, m
        self.fractSets[1].generator.set_orientGenerator( 'vonmises',   [numpy.radians(90.0), 250.0, None , 1] )    #[-0.87, 1.0, 0.52 ,
        #--                      a kappa concentration param of 95, corresponds roughly with SD of 1/30*pi = 5degrees ; a kappa of 33.0,
        self.fractSets[1].generator.set_buffer_dist(0.5) ##-- Buffer distance (m) for each fracture set
        self.fractSets[1].generator.set_proximity_condition(True)
        self.fractSets[1].generator.set_nonSameLevelIntersect_condition(True)
        #self[1].generator.set_NIntersectFract(0)
        self.fractSets[1].generator.set_nonIntersecting_fractSets( [ 'L1-NS', 'L2-EW' ] , [1.0, 1.0],  [1.0, 1.0] ) ## Create fracture
        self.fractSets[2].generator.set_sizeGenerator( 'fixed', [5.0]) ## Define the length function, you can have Power-law (shape, m
        self.fractSets[2].generator.set_orientGenerator( 'vonmises',   [numpy.radians(0.0), 200.0, None , 1] )    #[-0.87, 1.0, 0.52 ,
        #--                      a kappa concentration param of 95, corresponds roughly with SD of 1/30*pi = 5degrees ; a kappa of 33.0
        self.fractSets[2].generator.set_buffer_dist(0.5) ##-- Buffer distance (m) for each fracture set
        self.fractSets[2].generator.set_proximity_condition(True)
        self.fractSets[2].generator.set_nonSameLevelIntersect_condition(True)
        #self[1].generator.set_NIntersectFract(0)
        self.fractSets[2].generator.set_nonIntersecting_fractSets( [ 'L1-NS', 'L2-EW',  'L3-NS'] , [1.0, 1.0, 0.0],  [1.0, 1.0, 1.0 ])
```

Figure 12: The input file for the step-by-step simulation. Please see section 3 for additional information on the utilized settings. This input file has been provided with the supplementary materials of this document.

The first loop within FracSim2D is over the hierarchal levels (Figure 3). Therefore, in this example, the algorithm first places N-S fractures (FSET1) at a roughly equal spacing within the modelling domain (Table 1 and Figures 12-13a). Once the P21 limit for FSET1 is reached, the simulator starts placing E-W striking fractures belonging to FSET2 (Table 1 and Figures 12-13b to e). Because of the assigned geological controls, these fractures cannot intersect the previously generated N-S fractures (Table 1) Therefore, these fractures are clipped (shortened in length) so that they fit in between the features belonging to FSET1 (Figure 13 and Table 1). Clipping fractures in order to honour the assigned geological controls is one of the key characteristics of FracSim2D. However, this does imply that the assigned statistical input is generally not completely respected by the resulting DFN (Table 1and Figure 13). Finally, in between the E-W fractures, new N-S features (FSET3) are placed (Figure 13e-f). Again, the modelled lengths are clipped due to the no higher order intersection control.

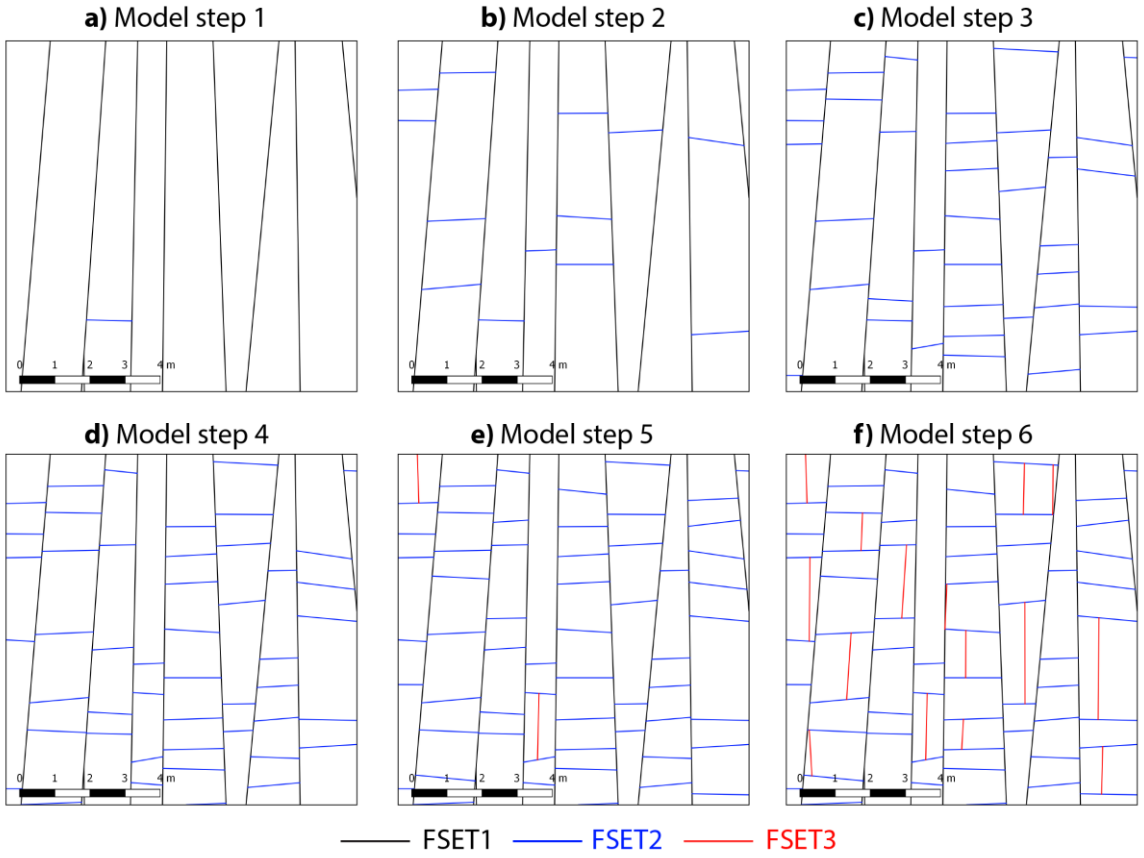


Figure 13: a-f) Step by step modelling workflow for generating a nested orthogonal network (see table 1 for modelling parameters).

6. Flaws and pitfalls to the current version of the algorithm

Although showing promising results, FracSim2D has some limitations. For instance, due to crude methodology used (i.e. trial and error), simulating large fracture networks with 1) high fracture intensities, 2) multiple hierarchical levels and 3) and strict spacing relationships, will take a relatively long time to complete, especially with respect to randomly populated DFN's. Further, in extreme cases with high fracture intensity and strict geological constraints (i.e. intersection and spacing rules), the algorithm has difficulties in finding suitable locations for fracture placement. This process repeats itself for each to be placed fracture, resulting in simulations which essentially don't converge. For the current version of the code, this issue can only be resolved by placing less strict geological rules to the model or by changing the fracture intensity limit.

Further, the current version of the code does not allow for the implementation of fracture intersection probabilities (i.e. the likelihood that two fractures will intersect or abut). This implies that fracture network geometries showing mixed topologies (e.g. 50 % crossing fractures and 50 % abutting fractures) are difficult to model using FracSim2D. We are planning to incorporate intersection probabilities in future versions of the code.

Another limitation is with changing fracture characteristics over the modelling domain. FracSim2D uses fixed behaviour for each fracture characteristic such as: 1) fracture intensity, 2) fracture length and orientation behaviour, or 3) topology and hierarchy. On outcrops, these characteristics can change. For example, based on its proximity to large scale features such as faults or fold hinges, the fracture intensity, dominant orientation and/or intersection probability of each fracture set can change

drastically (Bisdom, Bertotti, & Bezerra, 2017; Hanke, Fischer, & Pollyea, 2018). These local deviations cannot be modelled by the current version of FracSim2D. In future versions of the code we are planning to account for varying fracture characteristics within the modelling domain. This will be done by implementing an additional loop to the main workflow (Figure 3), so that laterally varying fracture characteristics (intensity, orientation, abutment and spacing) can be incorporated.

References

Alstott, J., Bullmore, E., & Plenz, D. (2014). Powerlaw: A python package for analysis of heavy-tailed distributions. *PLoS ONE*, *9*(1). https://doi.org/10.1371/journal.pone.0085777

Bisdom, K., Bertotti, G., & Bezerra, F. H. (2017). Inter-well scale natural fracture geometry and permeability variations in low-deformation carbonate rocks. *Journal of Structural Geology*, *97*, 23–36. https://doi.org/10.1016/j.jsg.2017.02.011

Dershowitz, W. S., & Herda, H. H. (1992). Interpretation of fracture spacing and intensity. In *The 33th U.S. Symposium on Rock Mechanics (USRMS), 3-5 June, Santa Fe, New Mexico*. https://doi.org/10.1080/00856401.2017.1295342

Hanke, J. R., Fischer, M. P., & Pollyea, R. M. (2018). Directional semivariogram analysis to identify and rank controls on the spatial variability of fracture networks. *Journal of Structural Geology*, *108*(January 2017), 34–51. https://doi.org/10.1016/j.jsg.2017.11.012

Sanderson, D. J., & Nixon, C. W. (2015). The use of topology in fracture network characterization. *Journal of Structural Geology*, *72*, 55–66. https://doi.org/10.1016/j.jsg.2015.01.005