$UnBlocks^{gen}$: A Python Library for 3D Rock Mass Generation and Analysis

Leandro Lima Rasmussen

May 31, 2020

Abstract

$UnBlocks^{gen}$ is a Python 3 library for the generation and analysis of 3D rock block systems. This technical documentation gives an introduction to the three main features provided by the library: Discrete Fracture Network construction; block system generation; and rock blocks' shape and size analysis. Instructions as to the installation, compilation from source and library usage are also provided.

Contents

1 Introduction

Instructions on how to compile $UnBlocks^{gen}$ from source and installing the library are provided. The library has been developed for Python 3 and tested in Ubuntu 18.04 Bionic Beaver and Ubuntu 20.04 Focal Fossa. It has not been prepared for Windows or other Linux distributions, however with few modifications it may also work in these operational systems.

1.1 Compilation from Source and Installation

$UnBlocks^{gen}$ consists of a library file, a Python script and a PNG image. The library file 'unblocks.so' provides access to the Discrete Fracture Network construction and 3D block system generation commands; the Python script 'plotTools.py' offers the tools for blocks' shape and size analysis; and the image 'TriangularPlot.png' is used by 'plotTools' for displaying the block volume distribution, block shape diagram and block shape distribution. The mentioned files are all provided in the installation folder. Installation consists solely in keeping the three mentioned files together in the same folder where the Python script importing the library will be developed and run.

The 'unblocks.so' library file provided has been compiled in Ubuntu 18.04 Bionic Beaver. In case error messages are shown when attempting to import the library to Python, even after the libraries mentioned next have been installed, it is recommended that the library be re-compiled from source by following the instructions below.

$UnBlocks^{gen}$ have been developed in C++ based on a Cmake project. The source files are provided in the 'src' folder. In order to compile the library from source in Ubuntu 18.04 Bionic Beaver, the following packages available from Ubuntu Package archive must be previously installed:

- cmake
- build-essential
- python3
- python3-numpy
- python3-matplotlib
- coinor-clp
- coinor-libclp-dev
- libboost-python-dev
- paraview

In case Ubuntu 20.04 Focal Fossa is the operational system used, the following additional packages available from Ubuntu Package archive must also be installed:

- libboost1.67-dev
- libboost-python1.67-dev

A package can be installed directly from terminal using the command below. It may be necessary to apply the update command before installing the packages.

'sudo apt-get install package'
('package' should be replaced by the desired package name)

Constructed Discrete Fracture Networks and generated 3D block systems can be exported to VTK format. Therefore, the Paraview package [1] has been defined as required for visualizing the data from VTK files.

In order to compile, open Ubuntu terminal and access the 'build' folder inside 'src'. Subsequently, type the following commands within the terminal:

'cmake ..'
'make'.

Compilation will start and the library file 'unblocks.so' will be produced. In this way, compilation of $UnBlocks^{gen}$ from source is concluded. It should be reminded that, to use the library, the files 'unblocks.so', 'plotTools.py' and 'TriangularPlot.png' should be kept together in the same folder.

2 Examples

Three examples are given to introduce the user to the main features of $UnBlocks^{gen}$:

1. In the first example, a simple deterministic DFN model of a rock mass with four discontinuities is constructed and the block system generated.
2. In the second example, the stochastic DFN of a rock mass with three discontinuity sets is constructed and the block system generated, including a tunnel excavation. The rock blocks' shape and size are also analyzed and the results plotted.
3. in the third example, it is shown that complex excavation geometries, such as that of a open pit slope, can be accommodated by the library.

2.1 Simple Rock Mass

In this first example, it is shown how the user can interact with $UnBlocks^{gen}$ in Python 3. A simple deterministic Discrete Fracture Network containing two fracture sets will be created. Each fracture set will contain two fractures and will provide the input for the 3D block system generation.

In order to start, in Ubuntu terminal access the library file 'unblocks.so' location and run Python 3. In the Python terminal, write the following command:

```
>>> from unblocks import *
```

In case the library was successfully loaded, the screen should appear similar to the one presented in Fig. 1 below:

Figure 1: $UnBlocks^{gen}$ library loaded in Python 3.

In order to generate and analyze the block system of a fractured rock mass, first a Discrete Fracture Network has to be constructed. This can be achieved by means of the DFN class, which should be instantiated into a Python object. The DFN object provides then access to the commands for constructing a Discrete Fracture Network.

The commands below show the instantiation of a Python object from the DFN class and the definition of the model region size: 100m x 100m x 100m. Subsequently, two fracture sets are created with ids equal to 0 and 1 respectively.

```
>>> dfn = DFN()
>>> dfn.set_RegionMaxCorner([100,100,100])
>>> dfn.add_FractureSet()
Fracture Set number 0 added!
>>> dfn.add_FractureSet()
Fracture Set number 1 added!
```

It is now necessary that the fracture sets be filled with one or more fractures. A triangular and a circular fracture will be added to each set:

```
>>> dfn.fractureSets[0].add_TriangularFracture
    ([10,10,10],[80,90,70],[35,60,15])
Triangular fracture id 0 added!
>>> dfn.fractureSets[0].add_CircularFracture
    ([50,50,50],90,45,40)
Fracture id 1 added!
>>> dfn.fractureSets[1].add_TriangularFracture
    ([60,35,75],[15,45,20],[85,70,40])
Triangular fracture id 0 added!
>>> dfn.fractureSets[1].add_CircularFracture
    ([30,30,60],180,90,30)
Fracture id 1 added!
```

When inserting a triangular fracture, the three corner points that form the triangle should be informed. When inserting a circular fracture, the center point of the fracture is first provided, followed by the dip direction, dip angle and fracture radius values. $UnBlocks^{gen}$ assumes that the north direction is defined by the X axis and dip direction is a clockwise angular rotation from X. Fig. 2 illustrates the convention used with an example.

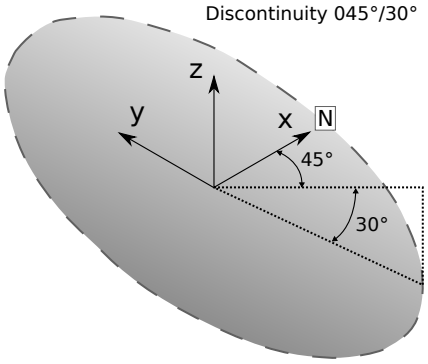

Figure 2: Fracture orientation convention adopted in $UnBlocks^{gen}$.

The constructed DFN and the model region can be exported to VTK files so as to be visualized. The commands below show the export commands which will generate the VTK files named 'dfnCreated.vtk' and 'modelRegion.vtk'. These can be visualized in Paraview, as shown in Fig. 3

```
>>> dfn.export_DFNVtk("dfnCreated")
DFN Vtk exported!
>>> dfn.export_RegionVtk("modelRegion")
DFN Region Vtk exported!
```

Once the desired DFN has been constructed, the next step consists in the generation of the 3D block system. This can be achieved by means of the Generator class, which should be instantiated into a Python object. The Generator object provides access to the commands for generating the 3D block system from a constructed DFN.

The commands below shows the instantiation of a Python object from the

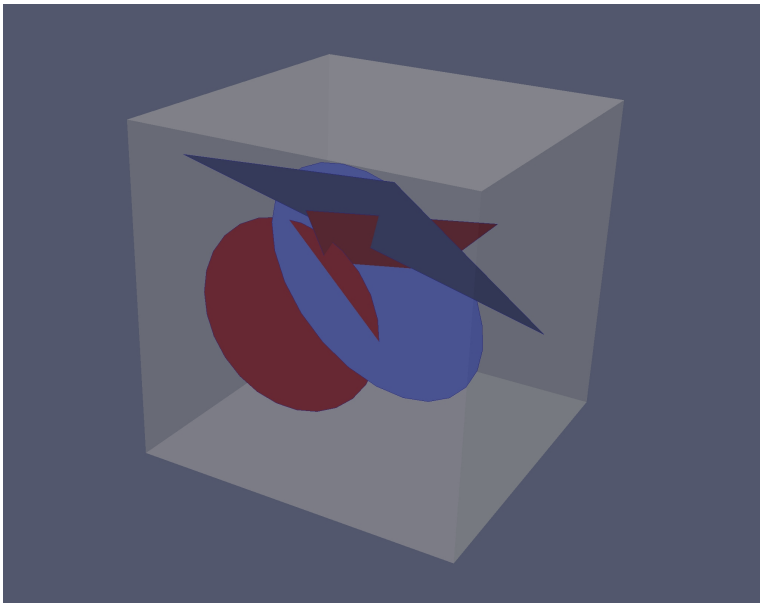

Figure 3: DFN fractures and model region. Fracture sets indicated by the red and blue colors.

Generator class, the generation of the block system based on the DFN object previously constructed, and the creation of the VTK file 'rockBlocks.vtk' for visualization in Paraview.

```
>>> generator = Generator()
>>> generator.generate_RockMass(dfn)
Fracture id 0 from Fracture Set 0 is being analysed!
Fracture id 1 from Fracture Set 0 is being analysed!
Fracture id 0 from Fracture Set 1 is being analysed!
Fracture id 1 from Fracture Set 1 is being analysed!
Generated Block 0 Geometry
Generated Block 1 Geometry
Generated Block 2 Geometry
Generated Block 3 Geometry
Generated Block 4 Geometry
Generated Block 5 Geometry
Generated Block 6 Geometry
Generated Block 7 Geometry
Generated Block 8 Geometry
Generated Block 9 Geometry
Generated Block 10 Geometry
Generated Block 11 Geometry
Generated Block 12 Geometry
>>> generator.export_BlocksVtk("rockBlocks")
```

The 3D block system generated in shown in Fig. 4. In order to understand this block system, it is important to keep in mind that $UnBlocks^{gen}$ employs the sequential rock slicing method proposed by Boon et al. [2]. In this

approach, the model region is sliced sequentially, starting with the first fracture of the first fracture set until the last fracture of the last fracture set. Therefore, it is strongly recommended that fracture sets with more persistent fractures are defined first during the DFN construction phase.

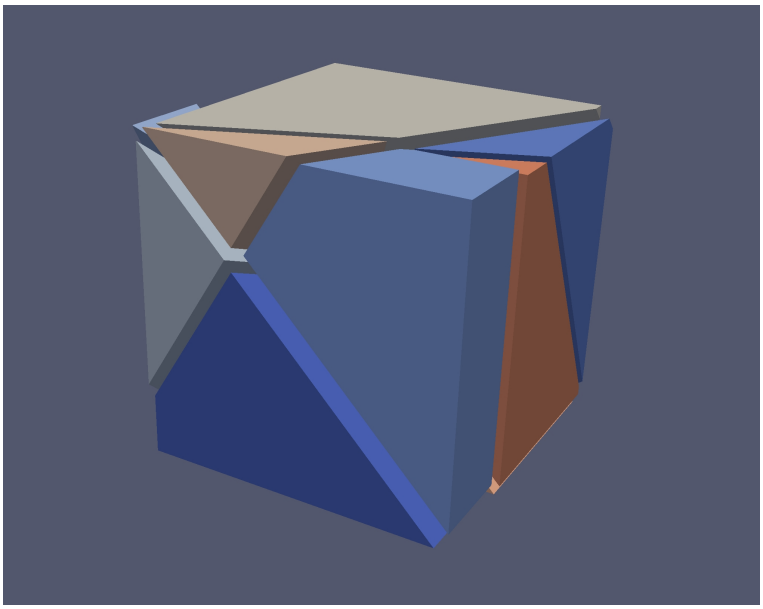

Figure 4: 3D block system generated from the constructed DFN. Blocks have been shrunk to facilitate their visualization.

If necessary, it is possible to iterative over each rock block within the blocks container of the generator object and export them individually to a VTK file. The commands below show how each rock block and related geometrical information can be exported to a separate VTK file:

```
>>> for i in range(len(generator.blocks)):
...     generator.blocks[i].export_BlockVtk("Block"+str(i))
```

2.2 Rock Mass with a Tunnel

This second example is based on the Monte Seco Tunnel, excavated through fractured gneiss in the state of Espírito Santo in Brazil. The tunnel geometry and calibrated stochastic DFN parameters are presented in the work of Rasmussen et al. [3]. In short, the rock mass contains a total of three fracture sets: one foliation set and two joint sets. Due to the stochastic nature of the fractures, the DFN parameters were estimated considering the following assumptions:

1. Fractures are circular entities in space, with location following a Poisson process.

2. Orientations are defined by the Fisher distribution.

3. Persistence of the joint sets are represented by the log-normal distribution.

4. Fracture intensity is defined by the Dershowitz and Herda [4] P-system.

The table below presents the calibrated stochastic DFN parameters from the Monte Seco tunnel section analyzed.

Fracture Set	Orientation Dip (°)	Dip Dir. (°)	Fisher K	Persistence* Mean (m)	Std. (m)	Dist.	Intensity (1/m)
Foliation	48	110	-	∞	-	-	1.2 (P10)
F1	71	180	23	0.725	0.52	Log-N.	1.24 (P32)
F2	65	258	30	1.02	0.445	Log-N.	2.12 (P32)

Figure 5: Discrete Fracture Network parameters for the second example (*Persistence refers to the radius distribution of the fractures).

In order to construct the stochastic DFN for the Monte Seco tunnel rock mass using $UnBlocks^{gen}$, it will be necessary to make use of two mapping classes: the line and volume mappings. While line mapping provides for the calculation of P10 fracture frequency, the volume mapping gives P30 and P32 fracture intensity values. P10, P30 and P32 are values based on the P-system proposed by Dershowitz and Herda [4].

In order to start, first $UnBlocks^{gen}$ library and 'plotTools' should be imported into Python. In this example, 'plotTools.py' (make sure this file is available together with 'TriangularPlot.png') will used to generate three blocks' shape and size analysis plots: block volume distribution, block shape diagram and block shape distribution. These plots have been suggested by Kalenchuk et al. [5] to assist with the rock block analysis process. The Python commands are:

```
>>> from unblocks import *
>>> import plotTools
```

Subsequently, the dfn object shoud be instantiated from the DFN class and the models' size set equal to 10m x 30m x 30m. A seed value to the pseudo-random number generation system should also be selected. To add a line mapping, two points representing the extremes ends of the line are required. In the current version of the library, a volume mapping automatically encompasses the whole model region. The commands below show the DFN object creation with the inclusion of three fracture sets as well as a line and a volume mapping:

```
>>> dfn = DFN()
>>> dfn.set_RegionMaxCorner([10,30,30])
>>> dfn.set_RandomSeed(100)
>>> dfn.add_FractureSet()
Fracture Set number 0 added!
>>> dfn.add_FractureSet()
```

```
Fracture Set number 1 added!
>>> dfn.add_FractureSet()
Fracture Set number 2 added!
>>> dfn.add_LineMapping([10,30,0],
    [0.766582,4.631392,24.30794])
Line mapping added!
>>> dfn.add_VolumeMapping()
Volume mapping added!
```

With the mapping elements defined, it is possible to generate the DFN by means of Python 'while loops'. Fractures are generated and added to each fracture set up to the moment their intensity values satisfy the calibrated ones, shown in the table above. It is indicated below how this can be achieved in this example:

```
>>> while (dfn.linesMapping[0].get_P10(0) < 1.2):
...dfn.fractureSets[0].add_BaecherFracture(110, 48, 50000,"det"
    , 200, 0)
>>> while (dfn.volumesMapping[0].get_P32(1) < 1.24):
...dfn.fractureSets[1].add_BaecherFracture(180, 71, 23,"log",
    0.725, 0.52)
>>> while (dfn.volumesMapping[0].get_P32(2) < 2.12):
...dfn.fractureSets[2].add_BaecherFracture(258, 65, 30,"log",
    1.02, 0.445)
```

For the application of 'add_BaecherFracture' function, six parameters were necessary. It is worth explaining them one by one:

1. Dip direction.
2. Dip angle.
3. Fisher distribution parameter.
4. Probabilistic distribution for persistence value. Can be either 'det' for deterministic, 'log' for log-normal and 'exp' for exponential.
5. Mean persistence value (for deterministic, it is the persistence value adopted).
6. Standard deviation of the persistence value (for deterministic and exponential distribution, this value is not used).

Once the loops start, thousands of fractures will be added. In the end of the process, the stochastic DFN can be exported to a VTK file with the following command for visualization in Paraview. In case necessary, the DFN of each fracture set can be exported to a VTK file separately and the commands for achieving this are also shown below. Fig. 6 shows the resulting DFN model.

```
>>> dfn.export_DFNVtk("dfn")
DFN Vtk exported!
>>> dfn.fractureSets[0].export_FractureSetVtk("fracSet0")
>>> dfn.fractureSets[1].export_FractureSetVtk("fracSet1")
>>> dfn.fractureSets[2].export_FractureSetVtk("fracSet2")
```

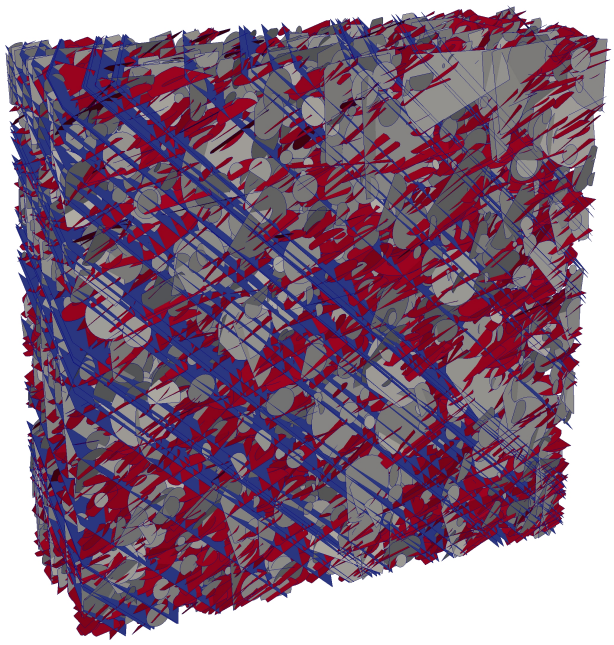

Figure 6: Constructed stochastic DFN for the Monte Seco tunnel rock mass.

After the stochastic DFN has been constructed, it is possible to start the block system generation process. First, an object from the Generator class is instantiated and some basic settings are defined. These settings are related to the minimum permissible value for the inscribed sphere radius of a block as well as the maximum allowable aspect ratio, calculated as the ratio of the bounding sphere radius to the inscribed sphere radius. Both values should be selected carefully: while overly tolerable constraints (e.g. minimum inscribed sphere radius equal to zero and maximum aspect radius equal to infinite) may lead to errors during block system generation process, tight constraints can deeply affect the generated blocks' geometry, making them no longer representative. Once these values have been selected, the generation process can occur based on the previously constructed DFN. The Python commands for performing these actions are shown below:

```
>>> generator = Generator()
>>> generator.set_MinInscribedSphereRadius(0.05)
>>> generator.set_MaxAspectRatio(30)
>>> generator.generate_RockMass(dfn)
>>> generator.export_BlocksVtk("blocksBeforeExcavation")
```

It may take around 20 minutes for the generation process to be concluded. However, the total time depends on the hardware configuration. Once the generation is over, it is then possible to visualize the generated block system and analyze the results. Fig. 7 shows the rock blocks obtained from the constructed DFN before the Monte Seco tunnel excavation.

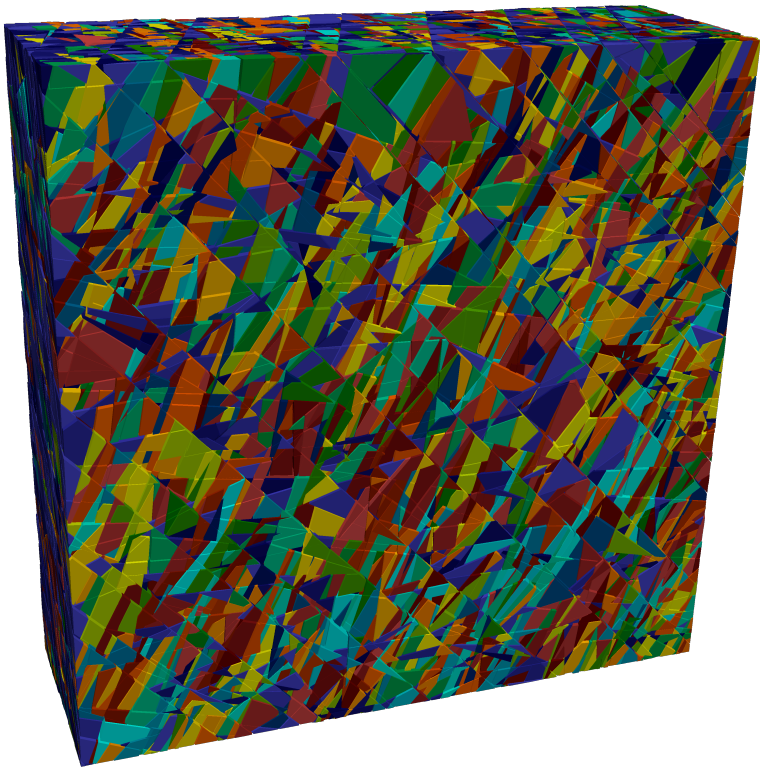

Figure 7: Block system generated before tunnel excavation, with approximately 130,000 rock blocks. Blocks are slightly shrunk to facilitate their visualization.

In order to analyze the resulting block system, the library provides for the creation of three different plots: block volume distribution, block shape diagram and block shape distribution. These plots were proposed in the work of Kalenchuck et al. [5] and the reader is referred to this publication for further information. It is recommended that these plots be generated and visualized before any excavation is performed, otherwise cut blocks, which are no longer representative of the constructed DFN, would be analyzed as well. As stated before, the 'plotTools.py' Python script is used for generating

the plots based on the Matplotlib package [6] and the commands that should be used are:

```
>>> plotTools.blockVolumeDistribution(generator.get_Volumes(
    False))
>>> plotTools.blockShapeDiagram(generator.get_AlphaValues(False
    ), generator.get_BetaValues(False), generator.get_Volumes(
    False), 0.05)
>>> plotTools.BlockShapeDistribution(generator.get_AlphaValues(
    False), generator.get_BetaValues(False), generator.
    get_Volumes(False))
>>> plotTools.showPlots()
```

In the commands above, 'False' refers to the question of whether boundary blocks (blocks cut by the model borders) should be included in the analysis or not. It is recommended that these blocks are not included. In the 'plotTools.blockShapeDiagram' command, the last value of 0.05 means that only the alpha and beta values of 5% of all rock blocks will be included in the diagram. This is done so as to impede the generation of a convoluted diagram due to the large number of points that would be inserted in case 100% of the values were considered.

Fig. 8 shows the block volume distribution, Fig. 9 presents the block shape distribution, and Fig. 10 portrays the block shape diagram obtained from the generated block system.

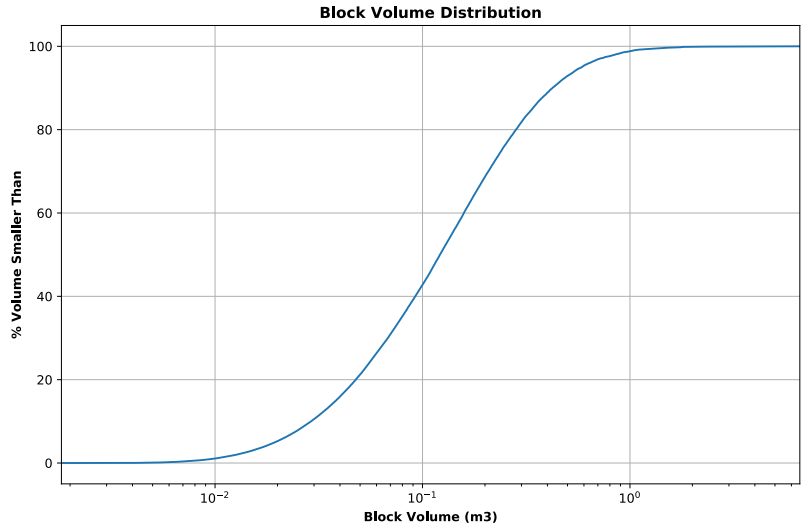

Figure 8: Block volume distribution.

Concluding, it is desired to perform the tunnel excavation in the block system generated. For this goal, the library offers a class named 'ExcavationElement', which defines auxiliary fictitious planes used during the block system generation

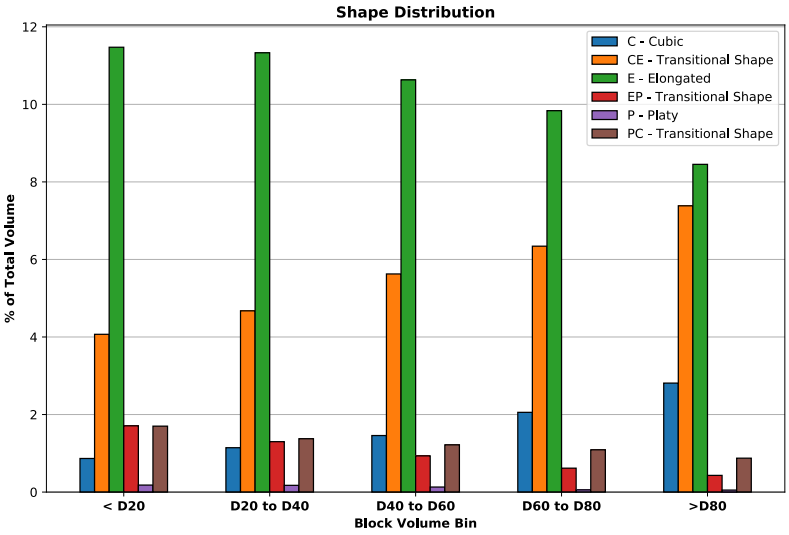

Figure 9: Block shape distribution.

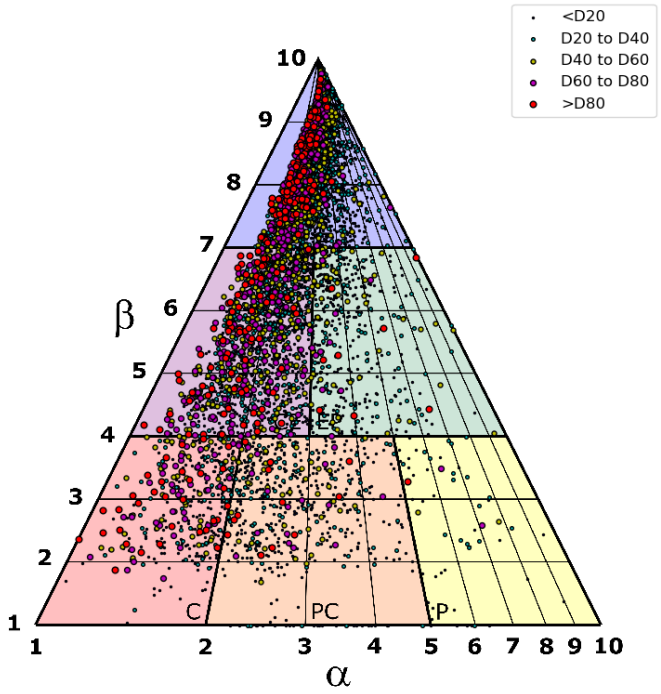

Figure 10: Block shape diagram.

for delimiting the excavation region. The excavation elements are triangles with vertices and connectivity defined by an external file with extension 'OBJ'. This files should be written with the following configuration: first, all vertices are inserted, followed by the triangles' connectivity. Every line with vertice data should begin with the 'v' letter and every line with connectivity information should start with the 'f' letter. This example comes with the 'tunnel.obj' file, which should be studied and understood.

The tunnel excavation is performed and the block system VTK file generated by the following commands. The block system including the tunnel excavation is presented in Fig. 11.

```
>>> generator.import_ExcavationElementsObj("tunnel")
>>> generator.export_ExcavationElementsVtk("tunnelvtk")
>>> generator.excavate_RockMass()
>>> generator.export_BlocksVtk("blocksAfterExcavation")
```

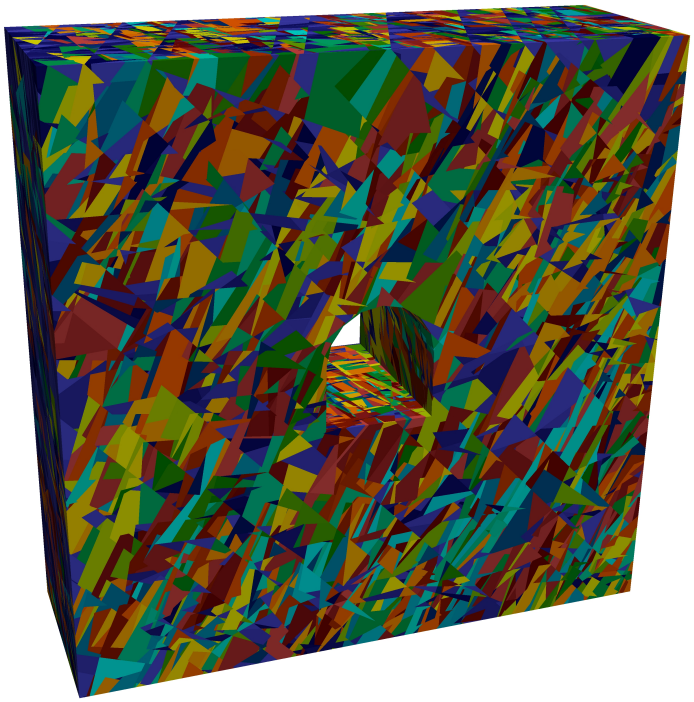

Figure 11: 3D block system generated from the constructed DFN with a tunnel excavation.

2.3 Open Pit Slope Model

In this third example, the fractured rock mass of an open pit slope will be generated. This example shows that $UnBlocks^{gen}$ is capable of handling complex excavation geometries.

Fig. 12 shows the open pit excavation region in darker gray color and Fig. 13 the block system generated considering the excavation. Of note, the observant user may realize that, apparently, "extra" blocks are generated near the benches. This is due to the fictitious excavation planes that are used to assist in adjusting the model geometry to the excavation process. However, the blocks' ID number define the block that should be clumped after the excavation. In this way, make sure that blocks with the same ID are checked when determining the blocks real geometry, in case they are used in other software packages (e.g. DEM or DDA simulation tools).

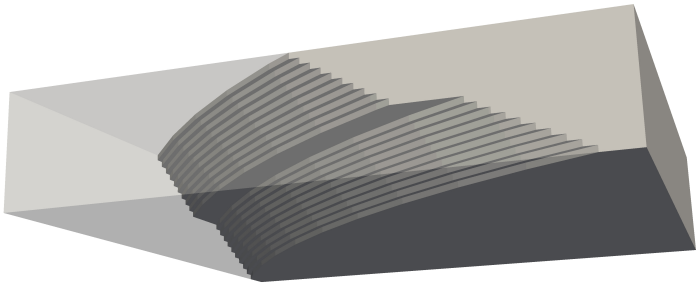

Figure 12: Excavation region being represented by the darker gray color.

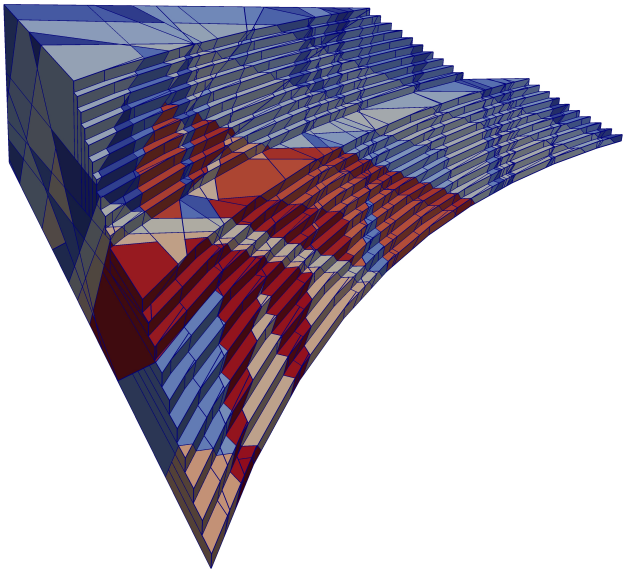

Figure 13: 3D block system generated from the constructed DFN with an open pit excavation.

The Python commands for this example can be found in the examples folder.

3 Library Python Commands

3.1 Discrete Fracture Network construction

The following commands are available from a DFN object:

- **add_FractureSet()** - Add a new fracture set to the DFN model.
- **add_LineMapping([x1,y1,z1], [x2,y2,z2])** - Add a new line mapping connecting point 1 to point 2.
- **add_CircularMapping([x1,y1,z1], dipDirection, dipAngle, radius)** - Add a new circular area mapping with center in point 1 and dip direction, dip angle and radius as specified.
- **add_QuadrilateralMapping([x1,y1,z1], [x2,y2,z2], [x3,y3,z3], [x4,y4,z4])** - Add a new quadrilateral area mapping delimited by 4 points.
- **add_VolumeMapping()** - Add a volumetric mapping encompassing the whole model volume.
- **set_RegionMaxCorner([x,y,z])** - Set the size of the model region ([100,100,100] by default).
- **set_NumberOfBorderPoints(number)** - Set the number of points to represent circular entities (26 by default).
- **set_RandomSeed(seed)** - Set the seed for the pseudo-random number generator.
- **export_DFNVtk("fileName")** - Export the entire DFN model to a VTK file.
- **export_RegionVtk("fileName")** - Export the model region to a VTK file.

3.1.1 Fracture Sets container

The Fracture Sets container within a DFN object can be accessed by means of the 'fractureSets' vector. The following commands are available from a Fracture Set object:

- **add_CircularFracture([x1,y1,z1], dipDirection, dipAngle, radius)** - Add a new circular fracture with center in point 1 and dip direction, dip angle and radius as specified.
- **add_TriangularFracture([x1,y1,z1], [x2,y2,z2], [x3,y3,z3])** - Add a new triangular fracture defined by 3 points.

- **add_BaecherFracture(meanDipDirection, meanDipAngle, fisherConstant, "sizeDistribution", meanFractureSize, stdFractureSize)** - Add a new circular stochastic fracture with location following a Poisson process, orientation being defined by the Fisher distribution and persistence (i.e. fracture radius) following either a deterministic value, exponential distribution or log-normal distribution. The distribution is selected by setting "sizeDistribution" equal to "det", "exp" or "log". 'stdFractureSize' value is not used by both deterministic and exponentially distributed persistence.
- **export_FractureSetVtk("fileName")** - Export the fracture set DFN to a VTK file.

3.1.2 Mapping containers

There are three mapping containers within a DFN object: lines, surfaces and volumes mapping containers. The lines mapping container can be accessed by means of the 'linesMapping' vector; the surfaces mapping container by means of the 'surfacesMapping' vector; and the volumes mapping container by means of the 'volumesMapping' vector.

The following commands are available from a Line Mapping object:

- **get_P10(fractureSetId)** - Get the P10 value for the fractures within fracture set id specified.
- **export_LineMappingVtk("fileName")** - Export the line mapping to a VTK file.

The following commands are available from a Surface Mapping object:

- **get_P20(fractureSetId)** - Get the P20 value for the fractures within fracture set id specified.
- **get_P21(fractureSetId)** - Get the P21 value for the fractures within fracture set id specified.
- **export_SurfaceMappingVtk("fileName")** - Export the surface mapping to a VTK file.

The following commands are available from a Volume Mapping object:

- **get_P30(fractureSetId)** - Get the P30 value for the fractures within fracture set id specified.
- **get_P32(fractureSetId)** - Get the P32 value for the fractures within fracture set id specified.
- **export_VolumeMappingVtk("fileName")** - Export the volume mapping to a VTK file.

3.2 3D Block System generation

The following commands are available from a Generator object:

- **set_MaxAspectRatio(value)** - Set the maximum aspect ratio permissible for a rock block.
- **set_MinInscribedSphereRadius(value)** - Set the minimum inscribed sphere radius permissible for a rock block.
- **generate_RockMass(dfn)** - Generate the 3D blocks system from a DFN object.
- **import_ExcavationElementsObj("fileName")** - Import the watertight triangular mesh representing the excavation region.
- **excavate_RockMass()** - Excavate the generated block system in the region defined by the imported triangular mesh.
- **export_ExcavationElementsVtk("fileName")** - Export the watertight triangular mesh representing the excavation region to a VTK file.
- **export_BlocksVtk("fileName")** - Export the 3D block system to a VTK file.
- **get_Volumes(bool)** - Get a Python list of blocks volume, used in conjunction with 'plotTools.py' (bool set to 'True' for considering border blocks or 'False' for disconsidering them).
- **get_AlphaValues(bool)** - Get a Python list of blocks alpha value, used in conjunction with 'plotTools.py' (bool set to 'True' for considering border blocks or 'False' for disconsidering them).
- **get_BetaValues(bool)** - Get a Python list of blocks beta value, used in conjunction with 'plotTools.py' (bool set to 'True' for considering border blocks or 'False' for disconsidering them).

3.2.1 Blocks container

The Blocks container within a Generator object can be accessed by means of the 'blocks' vector. The following commands are available from a Block object:

- **get_Volume()** - Get the volume of the block.
- **get_Alpha()** - Get the alpha value of the block.
- **get_Beta()** - Get the beta value of the block.

- **get_Order()** - Get the order of the block.
- **get_AspectRatio()** - Get the aspect ratio value of the block.
- **get_InscribedSphereRadius()** - Get the inscribed sphere radius value of the block.
- **export_BlockVtk("fileName")** - Export the block geometry and information to a VTK file.

3.3 Blocks' shape and size analysis

The following commands are available from the 'plotTools.py' Python script:

- **blockVolumeDistribution(volumes)** - Generate the block volume distribution. 'volumes' is provided by a Generator object.
- **blockShapeDiagram(alphaValues, betaValues, volumes, percentage)** - Generate the block shape diagram. 'alphaValues', 'betaValues' and 'volumes' are provided by a Generator object. 'percentage' refers to the percentage amount of results that will be plotted in the diagram from the total number of results available.
- **BlockShapeDistribution(alphaValues, betaValues, volumes)** - Generate the block shape distribution. 'alphaValues', 'betaValues' and 'volumes' are provided by a Generator object.
- **showPlots()** - Plot the generated distributions and diagram.

4 About the License

5 Acknowledging $UnBlocks^{gen}$

In case $UnBlocks^{gen}$ was used for a scientific research, please provide the citation to the following SoftwareX journal paper: (recently submitted to SoftwareX)

References

[1] U. Ayachit, "The paraview guide: A parallel visualization application," 2015.

[2] C. Boon, G. Houlsby, and S. Utili, "A new rock slicing method based on linear programming," *Computers and Geotechnics*, vol. 65, pp. 12 – 29, 2015.

[3] L. L. Rasmussen, P. P. Cacciari, M. M. Futai, M. M. de Farias, and A. P. de Assis, "Efficient 3d probabilistic stability analysis of rock tunnels using a lattice model and cloud computing," *Tunnelling and Underground Space Technology*, vol. 85, pp. 282 – 293, 2019.

[4] W. S. Dershowitz and H. Herda, "Interpretation of fracture spacing and intensity," in *33rd US Symp. Rock Mech.*, (Sante Fe, USA), p. 757, 1992.

[5] K. S. Kalenchuk, M. S. Diederichs, and S. McKinnon, "Characterizing block geometry in jointed rockmasses," *International Journal of Rock Mechanics and Mining Sciences*, vol. 43, no. 8, pp. 1212 – 1225, 2006.

[6] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.