MiningPy

Release 0.6.3

Iain Fullelove

Jul 13, 2023

CONTENTS

Version: 0.6.3

Documentation: https://miningpy.readthedocs.io/en/latest/

Repository: https://bitbucket.org/incitron/miningpy

Stable Release:

- Anaconda Cloud Stable: https://anaconda.org/miningpy/miningpy
- PyPi Stable: https://pypi.org/project/miningpy

Nightly Release: Version Format: *version.version.version.yyyyMMddHHmm*

- Anaconda Cloud Nightly: https://anaconda.org/miningpy_nightly/miningpy
- PyPi Nightly: https://test.pypi.org/project/miningpy/

Testing Pipelines (Azure DevOps): https://dev.azure.com/Iain123/MiningPy

CHAPTER

ONE

ABOUT

MiningPy is intended to help mining engineers harness the full power of the Python ecosystem to solve routine mine planning problems. MiningPy was developed at IMC Mining. For any support or enquiries, please feel free to get in contact with the engineers at IMC Mining. This package includes tools to help with:

- **Block model manipulation:**
  - Indexing (ijk)
  - Reblocking (geometric & attribute based)
  - Rotations
  - Calculating the model framework (origin, dimensions, rotation, extents, etc…)
  - Validating the block model (missing internal blocks, checking the model is regular, etc…)
  - Creating bench reserves
  - Aggregating blocks for scheduling
  - Haulage modelling & encoding to the block model
- **Interfacing with commercial mine planning packages, such as:**
  - Maptek Vulcan
  - GEOVIA Whittle
  - COMET
  - Minemax Scheduler/Tempo
  - Datamine
- **Visualisation:**
  - Previewing block models directly in Python for fast reviewing of work
  - Previewing designs (.dxf) directly in Python
  - Exporting block models in ParaView compatible format

CHAPTER

TWO

WHY MININGPY?

There are numerous geological packages that have been written in Python, such as GemPy, PyGSLIB, and GeostatsPy. However, none of these packages directly provide any tools to handle mining engineering specific problems. MiningPy aims to provide a simple API to mining engineers that extends existing data science tools like Pandas, without having to re-invent the wheel every time they need to interface with commercial mine planning software or manipulate mining data.

CHAPTER

THREE

INSTALLATION

MiningPy is distributed using:

- conda-forge
- Anaconda Cloud
- PyPi

3.1 Conda

MiningPy can be installed using the Conda package manager. To install using *conda*, you need to add the *conda-forge* channel so that all dependencies are installed correctly:

```
conda config --add channels conda-forge
```

To install from conda-forge (after adding the conda-forge channel):

```
conda install miningpy
```

To install from Anaconda Cloud (after adding the conda-forge channel):

```
conda install -c miningpy miningpy
```

3.2 Pip

MiningPy can be installed using the Pip package manager:

```
pip install miningpy
```

CHAPTER

FOUR

EXAMPLE

```
import pandas as pd
import miningpy

blockModelData = {
    'x': [5, 5, 15],
    'y': [5, 15, 25],
    'z': [5, 5, 5],
    'tonnage': [50, 100, 50],
}

blockModel = pd.DataFrame(blockModelData)
blockModel.plot3D(
    xyz_cols=('x', 'y', 'z'),
    dims=(5, 5, 5),  # block dimensions (5m * 5m * 5m)
    col='tonnage',  # block attribute to colour by
)
```

CHAPTER

FIVE

DOCUMENTATION

Auto-generated documentation is hosted at Read The Docs.

You may also build the documentation yourself:

```
git clone https://bitbucket.org/incitron/miningpy/miningpy.git
cd miningpy/docs
make html
```

The documention can then be found in *miningpy/docs/_build/html/index.html*.

CHAPTER

SIX

SUPPORTED PLATFORMS & TESTING

6.1 Platforms

MiningPy is only tested on Microsoft Windows 10.

6.2 Testing

The package is built and tested nightly using environments based on Virtualenv and Conda (with the current base Anaconda packages).

Azure DevOps hosts and runs and the testing pipelines.

MiningPy is tested to be compatible with the following versions of Python:

- Python 3.9
- Python 3.8
- Python 3.7

Offical testing and support has been removed for the following versions of Python (although MiningPy might still work with them):

- Python 3.6

VTK is a dependency of MiningPy and there are known issues with the current Linux version of VTK published on PyPi.

The package is also automatically deployed nightly to TestPyPi, to ensure that official package releases are stable. The versioning format used on TestPyPi is: *version.version.version.yyyyMMddHHmm*.

CHAPTER

SEVEN

API REFERENCE

The MiningPy API currently includes:

- *Core API*.
- *Visualisation*.
- *Maptek Vulcan Integration*.

CHAPTER

EIGHT

AUTHOR

The creator of MiningPy is a mining engineer that primarly works in long-term strategic mine planning at IMC Mining. For any support or enquiries, please feel free to get in contact with the engineers at IMC Mining.

CHAPTER

NINE

LICENSE

MiningPy is licensed under the very liberal MIT-License.

9.1 Core API

This page describes the core functions in MiningPy

9.1.1 ijk

miningpy.core.**ijk**(*blockmodel: DataFrame*, *method: str = 'ijk'*, *indexing: int = 0*, *xyz_cols: Optional[Tuple[str, str, str]] = None*, *origin: Optional[Tuple[Union[int, float], Union[int, float], Union[int, float]]] = None*, *dims: Optional[Tuple[Union[int, float, str], Union[int, float, str], Union[int, float, str]]] = None*, *rotation: Tuple[Union[int, float], Union[int, float], Union[int, float]] = (0, 0, 0)*, *ijk_cols: Tuple[str, str, str] = ('i', 'j', 'k')*, *print_warnings: bool = True*, *inplace: bool = False*) → DataFrame

Calculate block ijk indexes from their xyz cartesian coordinates

Parameters

blockmodel: pd.DataFrame
pandas dataframe of block model

method: str, default 'ijk'
can be used to only calculate i, or j, or k

indexing: int, default 0
controls whether origin block has coordinates 0,0,0 or 1,1,1

xyz_cols: tuple of strings
names of x,y,z columns in model

origin: tuple of floats or ints
x,y,z origin of model - this is the corner of the bottom block (not the centroid)

dims: tuple of floats, ints or str
x,y,z dimension of regular parent blocks can either be a number or the columns names of the x,y,z columns in the dataframe

rotation: tuple of floats or ints, default (0,0,0)
rotation of block model grid around x,y,z axis, -180 to 180 degrees

ijk_cols: tuple of strings, default ('i', 'j', 'k')
name of the i,j,k columns added to the model

print_warnings: bool, default True
if True then will check if blocks are on a regular grid before the IJK calculation and print a warning to the user if the blocks are not regular (i.e. could get funky IJK values).

inplace: bool, default False
whether to do calculation inplace on pandas.DataFrame

Returns

pandas.DataFrame
indexed block model

Examples

```
>>> import pandas as pd
>>> import miningpy
...
>>> # block model data and framework
>>> data = {'x': [5, 5, 15],
...         'y': [5, 15, 25],
...         'z': [5, 5, 5]}
>>> xdim, ydim, zdim = 5, 5, 5  # regular block dimensions
>>> xorg, yorg, zorg = 2.5, 2.5, 2.5  # model origin (corner of first block)
...
>>> # Create block model from data
>>> blockmodel = pd.DataFrame(data)
>>> print(blockmodel)
    x   y  z
0   5   5  5
1   5  15  5
2  15  25  5
>>> # calculate i, j, k indexes
>>> blockmodel.ijk(indexing=1,  # use ijk function just like any other Pandas␣
↪function
...                 xyz_cols=('x', 'y', 'z'),  # input the x, y, z column names as␣
↪a tuple
...                 origin=(xorg, yorg, zorg),
...                 dims=(xdim, ydim, zdim),
...                 inplace=True) # can do inplace True/False like other standard␣
↪Pandas functions
>>> # print results of ijk calculation
>>> print(blockmodel)
    x   y  z  i  j  k
0   5   5  5  1  1  1
1   5  15  5  1  3  1
2  15  25  5  3  5  1
```

9.1.2 xyz

miningpy.core.**xyz**(*blockmodel: DataFrame*, *method: str = 'xyz'*, *indexing: int = 0*, *ijk_cols: Tuple[str, str, str] = ('i', 'j', 'k')*, *origin: Optional[Tuple[Union[int, float], Union[int, float], Union[int, float]]] = None*, *dims: Optional[Tuple[Union[int, float, str], Union[int, float, str], Union[int, float, str]]] = None*, *rotation: Tuple[Union[int, float], Union[int, float], Union[int, float]] = (0, 0, 0)*, *xyz_cols: Tuple[str, str, str] = ('x', 'y', 'z')*, *inplace: bool = False*) → DataFrame

Calculate xyz cartesian cooridinates of blocks from their ijk indexes

Parameters

blockmodel: pd.DataFrame
pandas dataframe of block model

method: str, default 'xyz'
can be used to only calculate i, or j, or k

indexing: int, default 0
controls whether origin block has coordinates 0,0,0 or 1,1,1

ijk_cols: tuple of strings, default ('i', 'j', 'k')
name of the i,j,k columns added to the model

origin: tuple of floats or ints
x,y,z origin of model - this is the corner of the bottom block (not the centroid)

dims: tuple of floats, ints or str
x,y,z dimension of regular parent blocks can either be a number or the columns names of the x,y,z columns in the dataframe

rotation: tuple of floats or ints, default (0,0,0)
rotation of block model grid around x,y,z axis, -180 to 180 degrees

xyz_cols: tuple of strings, default ('x', 'y', 'z')
names of x,y,z columns in model

inplace: bool, default False
whether to do calculation inplace on pandas.DataFrame

Returns

pandas.DataFrame
indexed block model

9.1.3 rotate_grid

miningpy.core.**rotate_grid**(*blockmodel: DataFrame*, *xyz_cols: Tuple[str, str, str] = ('x', 'y', 'z')*, *origin: Optional[Tuple[Union[int, float], Union[int, float], Union[int, float]]] = None*, *rotation: Tuple[Union[int, float], Union[int, float], Union[int, float]] = (0, 0, 0)*, *return_full_model: bool = True*, *derotate: bool = False*, *inplace: bool = False*) → Union[DataFrame, dict]

Rotate block model relative to cartesian grid This method uses a rotation matrix method Rotation is done using the right hand rule

Parameters

blockmodel: pd.DataFrame
pandas dataframe of block model

xyz_cols: tuple of strings, default ('x', 'y', 'z')
names of x,y,z columns in model

origin: tuple of floats or ints
x,y,z origin of model - this is the corner of the bottom block (not the centroid)

rotation: tuple of floats or ints, default (0,0,0)
rotation of block model grid around x,y,z axis, -180 to 180 degrees

return_full_model: bool, default True
whether to return the full block model or just a dict of the rotated x,y,z coordinates

derotate: bool
whether to rotate a model or derotate it back to it's normal orthogonal coordinate system this parameter exists because using the reverse angles in more than one dimension will not derotate a model

inplace: bool, default False
whether to do calculation inplace on pandas.DataFrame

Returns

pandas.DataFrame
rotated block model or dict of rotated x,y,z coordinates

9.1.4 group_weighted_average

miningpy.core.**group_weighted_average**(*blockmodel: DataFrame*, *avg_cols: Union[str, List[str]]*, *weight_col: str*, *group_cols: Optional[Union[str, List[str]]] = None*) → DataFrame

weighted average of block model attribute(s)

Parameters

blockmodel: pd.DataFrame
pandas dataframe of block model

avg_cols: str or list of str
column(s) to take the weighted average

weight_col: str
column to weight on. Example the tonnes column

group_cols: str or list of str
the columns you want to group on. Either single column or list of columns

Returns

pandas.DataFrame
block model

9.1.5 nblocks_xyz

miningpy.core.**nblocks_xyz**(*blockmodel: DataFrame*, *xyz_cols: Optional[Tuple[str, str, str]] = None*, *dims: Optional[Tuple[Union[int, float, str], Union[int, float, str], Union[int, float, str]]] = None*, *origin: Optional[Tuple[Union[int, float], Union[int, float], Union[int, float]]] = None*, *rotation: Tuple[Union[int, float], Union[int, float], Union[int, float]] = (0, 0, 0)*) → Tuple[Union[int, float], Union[int, float], Union[int, float]]

Number of blocks along the x,y,z axis. If the model is rotated, it is unrotated and then the number of blocks in the x,y,z axis is calculated.

Parameters

blockmodel: pd.DataFrame
pandas dataframe of block model

xyz_cols: tuple of strings
names of x,y,z columns in model

dims: tuple of floats, ints or str
x,y,z dimension of regular parent blocks can either be a number or the columns names of the x,y,z columns in the dataframe

origin: tuple of floats or ints
x,y,z origin of model - this is the corner of the bottom block (not the centroid)

rotation: tuple of floats or ints, default (0,0,0)
rotation of block model grid around x,y,z axis, -180 to 180 degrees

Returns

tuple of floats
Number of blocks along the x,y,z axis.

9.1.6 model_origin

miningpy.core.**model_origin**(*blockmodel: DataFrame*, *xyz_cols: Optional[Tuple[str, str, str]] = None*, *dims: Optional[Tuple[Union[int, float, str], Union[int, float, str], Union[int, float, str]]] = None*) → Tuple[float, float, float]

calculate the origin of a block model grid relative to its current xyz grid origin is the corner of the block with min xyz coordinates

Parameters

blockmodel: pd.DataFrame
pandas dataframe of block model

xyz_cols: tuple of strings
names of x,y,z columns in model

dims: tuple of floats, ints or str
x,y,z dimension of regular parent blocks can either be a number or the columns names of the x,y,z columns in the dataframe

Returns

tuple of floats
origin of a block model for each axis (x,y,z)

9.1.7 block_dims

miningpy.core.**block_dims**(*blockmodel: DataFrame*, *xyz_cols: Optional[Tuple[str, str, str]] = None*, *origin: Optional[Tuple[Union[int, float], Union[int, float], Union[int, float]]] = None*, *rotation: Tuple[Union[int, float], Union[int, float], Union[int, float]] = (0, 0, 0)*) → Tuple[float, float, float]

estimate the x, y, z dimensions of blocks if the blocks are rotated then they are unrotated first then the x, y, z dimensions are estimated

note that this function just estimates the dimensions of the blocks it may not always get the perfectly correct answer if there are alot of blocks missing in the grid (i.e. a sparse array of blocks) the estimation is less likely to be correct

Parameters

blockmodel: pd.DataFrame
pandas dataframe of block model

xyz_cols: tuple of strings
names of x,y,z columns in model

origin: tuple of floats or ints
x,y,z origin of model - this is the corner of the bottom block (not the centroid)

rotation: tuple of floats or ints, default (0,0,0)
rotation of block model grid around x,y,z axis, -180 to 180 degrees

Returns

tuple of floats (xdim, ydim, zdim)

9.1.8 check_regular

miningpy.core.**check_regular**(*blockmodel: DataFrame*, *xyz_cols: Optional[Tuple[str, str, str]] = None*, *origin: Optional[Tuple[Union[int, float], Union[int, float], Union[int, float]]] = None*, *dims: Optional[Tuple[Union[int, float, str], Union[int, float, str], Union[int, float, str]]] = None*, *rotation: Tuple[Union[int, float], Union[int, float], Union[int, float]] = (0, 0, 0)*, *tolerance: Union[int, float] = 1e-05*) → bool

check if the blocks in a block model are actually on a regular grid (including a rotated grid). note this is just an estimation of regularity, it is not perfect

Parameters

blockmodel: pd.DataFrame
pandas dataframe of block model

xyz_cols: tuple of strings
names of x,y,z columns in model

origin: tuple of floats or ints
x,y,z origin of model - this is the corner of the bottom block (not the centroid)

dims: tuple of floats, ints or str
x,y,z dimension of regular parent blocks can either be a number or the columns names of the x,y,z columns in the dataframe

rotation: tuple of floats or ints, default (0,0,0)
rotation of block model grid around x,y,z axis, -180 to 180 degrees

tolerance: float or int, default 0.00001
the difference of a blocks centroid from the point on a grid it should be located generally in the range of 0.1 to 0.000001

Returns

bool
whether block model is regular or not. True if regular.

9.1.9 check_internal_blocks_missing

miningpy.core.**check_internal_blocks_missing**(*blockmodel: DataFrame*, *xyz_cols: Optional[Tuple[str, str, str]] = None*, *dims: Optional[Tuple[Union[int, float, str], Union[int, float, str], Union[int, float, str]]] = None*, *rotation: Tuple[Union[int, float], Union[int, float], Union[int, float]] = (0, 0, 0)*, *origin: Tuple[Union[int, float], Union[int, float], Union[int, float]] = (0, 0, 0)*) → bool

check if there are missing internal blocks (not side blocks) within a regular block model

Parameters

blockmodel: pd.DataFrame
pandas dataframe of block model

xyz_cols: tuple of strings
names of x,y,z columns in model

dims: tuple of floats, ints or str
x,y,z dimension of regular parent blocks can either be a number or the columns names of the x,y,z columns in the dataframe

rotation: tuple of floats or ints, default (0,0,0)
rotation of block model grid around x,y,z axis, -180 to 180 degrees

origin: tuple of floats or ints, default (0,0,0)
ONLY NEEDED IF MODEL IS ROTATED x,y,z origin of model - this is the corner of the bottom block (not the centroid)

Returns

bool
whether block model contains missing internal blocks returns True if so

9.1.10 vulcan_csv

miningpy.core.**vulcan_csv**(*blockmodel: DataFrame*, *path: Optional[str] = None*, *var_path: Optional[str] = None*, *xyz_cols: Optional[Tuple[str, str, str]] = None*, *dims: Optional[Tuple[Union[int, float, str], Union[int, float, str], Union[int, float, str]]] = None*, *inplace: bool = False*) → DataFrame

transform pandas.Dataframe block model into Vulcan import compatible CSV format.

Parameters

blockmodel: pd.DataFrame
pandas dataframe of block model

path: str
filename for vulcan csv block model file

var_path: {optional} str
filename for csv that lists the Vulcan dtype of each column in block model (used if manually creating bdf)

xyz_cols: tuple of strings
names of x,y,z columns in model

dims: tuple of floats, ints or str
x,y,z dimension of regular parent blocks can either be a number or the columns names of the x,y,z columns in the dataframe

inplace: bool, default False
whether to do calculation inplace (i.e. add Vulcan headers inplace) or return pandas.DataFrame with Vulcan headers

Returns

pandas.DataFrame
block model in Vulcan import CSV format

9.1.11 vulcan_bdf

miningpy.core.**vulcan_bdf**(*blockmodel: DataFrame*, *path: Optional[str] = None*, *origin: Optional[Tuple[Union[int, float], Union[int, float], Union[int, float]]] = None*, *dims: Optional[Tuple[Union[int, float, str], Union[int, float, str], Union[int, float, str]]] = None*, *start_offset: Tuple[Union[int, float], Union[int, float], Union[int, float]] = (0.0, 0.0, 0.0)*, *end_offset: Optional[Tuple[Union[int, float], Union[int, float], Union[int, float]]] = None*, *format: str = 'T'*) → bool

Create a Vulcan block definition file from a vulcan block model. This script creates a BDF from a vulcan block model csv that can be imported into Vulcan. It assumes that bearing, dip and plunge are the default. values for the block model. Variables are given a default value of -99.0, a blank description and type 'double'. This script will define the parent schema. All of these values can be edited within Vulcan once the script has been run and bdf imported.

Parameters

blockmodel: pd.DataFrame
pandas dataframe of block model in Vulcan CSV import compatible format (use funtion: "vulcan_csv")

path: str
filename for vulcan bdf file

origin: tuple of floats or ints
x,y,z origin of model - this is the corner of the bottom block (not the centroid)

dims: tuple of floats, ints or str
x,y,z dimension of regular parent blocks can either be a number or the columns names of the x,y,z columns in the dataframe

start_offset: tuple of floats or ints, default (0.0, 0.0, 0.0)
minimum offset along the x,y,z axes

end_offset: tuple of floats or ints
maximum offset along the x,y,z axes

format: {'C', 'T'}, default 'T'
block model file format (classic = C, extended = T)

Returns

True if Vulcan .bdf file is exported with no errors

9.1.12 geometric_reblock

miningpy.core.**geometric_reblock**(*blockmodel: DataFrame*, *xyz_cols: Optional[Tuple[str, str, str]] = None*, *origin: Optional[Tuple[Union[int, float], Union[int, float], Union[int, float]]] = None*, *dims: Optional[Tuple[Union[int, float, str], Union[int, float, str], Union[int, float, str]]] = None*, *reblock_multiplier: Optional[Tuple[Union[int, float], Union[int, float], Union[int, float]]] = None*, *varlist_agg: Optional[dict] = None*, *min_cols: Optional[list] = None*, *max_cols: Optional[list] = None*)

Reblock a regular block model into larger or smaller blocks (aggregate or split blocks). Aggregating multiple small blocks into big blocks will be referred to as "superblocking" and splitting big blocks into smaller children will be referred to as "subblocking". This tool can be used as a tool for geometrically aggregating blocks in bench-phases.

This function utilises a `reblock_multiplier` in (x, y, z) dimensions to define the reblock. The function will either superblock or subblock but not both in the same call. The reblock_multiplier must be a multiple of the parent dimension. To superblock in one dimension and subblock in another, subblock to the smallest dimensions required and then superblock to the required size.

Weighting of blocks, is handled by argument `varlist_agg`. This dictionary must have the block model attribute to weight by as the key and the attributes to be weighted as a list in the key.

Parameters

blockmodel: pd.DataFrame
pandas dataframe of block model

xyz_cols: tuple of strings
names of x,y,z columns in model

dims: tuple of floats, ints or str
x,y,z dimension of regular parent blocks

origin: tuple of floats or ints
x,y,z origin of model - this is the corner of the bottom block (not the centroid)

reblock_multiplier: tuple of floats or ints
the scalar to change dimensions of blocks. E.g., (2, 2, 1) will double the blocks in the x and y dimension and keep the z dimension the same.

varlist_agg: dictionary
used to weight attributes. Key is the attribute to weight by, value is a list of attributes to weight. All attributes that are required to be carried through the reblock must be a key in the dictionary even if they are not weighting anything. Keys with empty lists will be divided or summed accordingly E.g., to carry through ore tonnes, volume and energy with ore grades, varlist_agg = {'ore_tonnes': ['au_grade', 'cu_grade'], 'volume':['density'], 'energy':[]}

min_cols: {optional} list of model attributes
attributes that require the min value e.g., pit_number

max_cols: {optional} list of model attributes
attributes that require the max value e.g., resource category

Returns

pandas.DataFrame
reblocked block model

Examples

```
>>> import pandas as pd
>>> import miningpy
```

```
>>> # block model example (zuck small - MineLib)
>>> url = "https://drive.google.com/uc?export=download&
↪id=1SOrYhqiu5Tg8Zjb7be4fUWhbFDTU1sEk"
```

```
>>> # read in block model from link
>>> data = pd.read_csv(url, compression='zip')
```

```
>>> # listing attributes to carry through (n.b. dropping ID column)
>>> # keys are what to weight by and values are lists of attributes to be weighted
>>> varlist_agg = {
>>> 'rock_tonnes': ['cost', 'value'],
>>> 'ore_tonnes': [],
>>> }
```

```
>>> # take the max or min value of reblock
>>> min_cols = ['final_pit']
>>> max_cols = ['period']
```

```
>>> # reblock function
>>> reblock = data.geometric_reblock(
>>> dims=(1, 1, 1),  # original dims of model
>>> xyz_cols=('x', 'y', 'z'),
>>> origin=(-0.5, -0.5, -0.5),  # bottom left corner
>>> reblock_multiplier=(2, 2, 1),  # doubling x and y dim and keeping z dim the same
>>> varlist_agg=varlist_agg,
>>> min_cols=min_cols,
>>> max_cols=max_cols,
>>> )
```

```
>>> reblock.plot3D(dims=(2, 2, 1), xyz_cols=('x', 'y', 'z'), col='value', widget=
↪'section')  # reblocked plot
>>> data.plot3D(dims=(1, 1, 1), xyz_cols=('x', 'y', 'z'), col='value', widget=
↪'section')  # original plot
```

9.1.13 index_3Dto1D

miningpy.core.**index_3Dto1D**(*blockmodel: DataFrame*, *indexing: int = 0*, *xyz_cols: Optional[Tuple[str, str, str]] = None*, *origin: Optional[Tuple[Union[int, float], Union[int, float], Union[int, float]]] = None*, *dims: Optional[Tuple[Union[int, float, str], Union[int, float, str], Union[int, float, str]]] = None*, *rotation: Tuple[Union[int, float], Union[int, float], Union[int, float]] = (0, 0, 0)*, *nblocks_xyz: Optional[Tuple[int, int, int]] = None*, *idxcol: str = 'ijk'*, *inplace: bool = False*) → DataFrame

Convert 3D array of xyz block centroids to 1D index that is reversible. Opposite of the function index_1Dto3D()

This is identical to the "ijk" parameter in Datamine block models. Note that "ijk" value from this function and from Datamine may be different, depending on which axis Datamine uses as the major indexing axis. Bot "ijk" indexing values are still valid.

Parameters

blockmodel: pd.DataFrame
pandas dataframe of block model

indexing: int, default 0
controls whether origin block has coordinates 0,0,0 or 1,1,1

xyz_cols: tuple of strings
names of x,y,z columns in model

origin: tuple of floats or ints
x,y,z origin of model - this is the corner of the bottom block (not the centroid)

dims: tuple of floats, ints or str
x,y,z dimension of regular parent blocks can either be a number or the columns names of the x,y,z columns in the dataframe

rotation: tuple of floats or ints, default (0,0,0)
rotation of block model grid around x,y,z axis, -180 to 180 degrees

nblocks_xyz: tuple of ints or None
number of blocks along the x,y,z axis. If the model is rotated, it is unrotated and then the number of blocks in the x,y,z axis is calculated. If "None" (default value) then the nx,ny,nz is automatically estimated

idxcol: str, default 'ijk'
name of the 1D index column added to the model

inplace: bool
whether to do calculation inplace on pandas.DataFrame

Returns

pandas.DataFrame
block model with 1D indexed column

9.1.14 index_1Dto3D

miningpy.core.**index_1Dto3D**(*blockmodel: DataFrame*, *indexing: int = 0*, *idxcol: str = 'ijk'*, *origin: Optional[Tuple[Union[int, float], Union[int, float], Union[int, float]]] = None*, *dims: Optional[Tuple[Union[int, float, str], Union[int, float, str], Union[int, float, str]]] = None*, *rotation: Tuple[Union[int, float], Union[int, float], Union[int, float]] = (0, 0, 0)*, *nblocks_xyz: Optional[Tuple[int, int, int]] = None*, *xyz_cols: Tuple[str, str, str] = ('x', 'y', 'z')*, *inplace: bool = False*) → DataFrame

Convert IJK index back to xyz block centroids. Opposite of the function index_3Dto1D()

This is identical to the "ijk" parameter in Datamine block models. Note that "ijk" value from this function and from Datamine may be different, depending on which axis Datamine uses as the major indexing axis. Bot "ijk" indexing values are still valid.

Parameters

blockmodel: pd.DataFrame
pandas dataframe of block model

indexing: int, default 0
controls whether origin block has coordinates 0,0,0 or 1,1,1

idxcol: str, default 'ijk'
name of the 1D index column added to the model

origin: tuple of floats or ints
x,y,z origin of model - this is the corner of the bottom block (not the centroid)

dims: tuple of floats, ints or str
x,y,z dimension of regular parent blocks can either be a number or the columns names of the x,y,z columns in the dataframe

rotation: tuple of floats or ints, default (0,0,0)
rotation of block model grid around x,y,z axis, -180 to 180 degrees

nblocks_xyz: tuple of ints or None
number of blocks along the x,y,z axis. If the model is rotated, it is unrotated and then the number of blocks in the x,y,z axis is calculated.

xyz_cols: tuple of strings, default ('x', 'y', 'z')
names of x,y,z columns in model

inplace: bool
whether to do calculation inplace on pandas.DataFrame

Returns

pandas.DataFrame
block model with 1D indexed column

9.1.15 grade_tonnage_plot

miningpy.core.**grade_tonnage_plot**(*blockmodel: DataFrame*, *grade_col: str*, *ton_col: str*, *cog_grades: Optional[List] = None*, *cog_grade_points: Optional[int] = None*, *plot_path: Optional[str] = None*, *show_plot: bool = False*)

Create and return grade-tonnage table and optional export plot as a .png and save image. Grade-Tonnage curves are a visual representation of the impact of cut-off grades on mineral reserves. Grades to plot can be specified, else grades to plot will be generated based on the range of grades in the model.

Parameters

blockmodel: pd.DataFrame
pandas dataframe of block model

grade_col: str
name of grade to plot, typically 'ore' grade

ton_col: str
name of tonnage column in the model

cog_grades: {optional} ints, floats
list of cut off grades to plot

cog_grade_points: {optional} int, default 10
number of cut off grades to plot between min and max grade

plot_path: {optional} str
path to save plot .png

show_plot: {optional} bool
if running in interactive console, show plot, default False

Returns

pandas.DataFrame
dataframe of grade-tonnage

matplotlib.pyplot
Grade-Tonnage plot

9.2 Visualisation

This page describes the visualisation functions in MiningPy

9.2.1 plot3D

miningpy.visualisation.**plot3D**(*blockmodel: DataFrame*, *xyz_cols: Tuple[str, str, str] = ('x', 'y', 'z')*, *col: Optional[str] = None*, *dims: Optional[Tuple[Union[int, float], Union[int, float], Union[int, float]]] = None*, *rotation: Tuple[Union[int, float], Union[int, float], Union[int, float]] = (0, 0, 0)*, *widget: Optional[str] = None*, *min_max: Optional[Tuple[Union[int, float], Union[int, float]]] = None*, *legend_colour: str = 'bwr'*, *window_size: Optional[Tuple[int, int]] = None*, *show_edges: bool = True*, *show_grid: bool = True*, *shadows: bool = True*, *show_plot: bool = True*) → Plotter

create activate 3D vtk plot of block model that is fully interactive

Parameters

blockmodel: pd.DataFrame
pandas dataframe of block model

xyz_cols: tuple of strings, default ('x', 'y', 'z')
names of x,y,z columns in model

col: str
attribute column to plot (e.g., tonnage, grade, etc)

dims: tuple of floats or ints
x,y,z dimension of regular parent blocks

rotation: tuple of floats or ints, default (0, 0, 0)
rotation of block model grid around x,y,z axis, -180 to 180 degrees

widget: {"slider","section"}
add widgets such as slider (cut off grade) or cross-section.

min_max: tuple of floats or ints
minimum and maximum to colour by values above/below these values will just be coloured the max/min colours

legend_colour: {optional} str, default 'bwr'
set the legend colour scale. can be any matplotlib cmap colour spectrum.

see: https://matplotlib.org/3.1.1/gallery/color/colormap_reference.html

see: https://matplotlib.org/3.1.0/tutorials/colors/colormaps.html

window_size: {optional} tuple of ints, default (1920, 1080)
size of plot window in pixels

show_edges: bool, default True
whether to show the edges of blocks or not.

show_grid: bool, default True
add x,y,z grid to see coordinates on plot.

shadows: bool, default True
whether to model shadows with a light source from the users perspective. if False, it is like the block model has been lit up with lights from all angles.

show_plot: bool, default True
whether to open active window or just return pyvistaqt.plotting.BackgroundPlotter object to .show() later.

Returns

pyvistaqt.plotting.BackgroundPlotter object & active window of block model 3D plot

9.2.2 export_html

miningpy.visualisation.**export_html**(*blockmodel: DataFrame*, *path: Optional[str] = None*, *xyz_cols: Tuple[str, str, str] = ('x', 'y', 'z')*, *dims: Optional[Tuple[Union[int, float], Union[int, float], Union[int, float]]] = None*, *rotation: Tuple[Union[int, float], Union[int, float], Union[int, float]] = (0, 0, 0)*, *cols: Optional[List[str]] = None*, *data_name: str = 'blockModel'*, *colour: Tuple[float] = (0.666667, 1, 0.498039)*, *split_by: Optional[str] = None*, *colour_range: Tuple[str] = ('blue', 'red')*) → bool

exports blocks and attributes of block model and embeds the data in a paraview glance html app to visualise and distribute

Parameters

blockmodel: pd.DataFrame
pandas dataframe of block model

path: str
filename for html file

xyz_cols: tuple of strings, default (‘x’, ‘y’, ‘z’)
names of x,y,z columns in model

dims: tuple of floats or ints
x,y,z dimension of regular parent blocks

rotation: tuple of floats or ints, default (0, 0, 0)
rotation of block model grid around x,y,z axis, -180 to 180 degrees

cols: list of strings
columns of attributes to visualise using vtk. If None then exports all columns

data_name: str, default ‘blockModel’
base name used for dataset in Paraview Glance if the split_by column is specified then the unique values in the split_by colum are appended to the base name

colour: tuple of floats, default (0.666667, 1, 0.498039)
default solid colouring of blocks if a column is chosen to split by then this colouring is not considered.

split_by: str
column that is used to split up the block model into components in the Paraview Glance app HTML file. The maximum number of unique values in this column is 256.

colour_range: tuple of two strings, default (‘blue’, ‘red’)
colouring range of values in the split_by column if no split_by then colour_range is ignored. accepted colours are: ‘red’, ‘blue’, ‘green’, ‘white’, ‘black’ default is (‘blue’, ‘red’). i.e. colour blue to red

Returns

True if .html file is exported with no errors

9.2.3 plot3D_dxf

miningpy.visualisation.**plot3D_dxf**(*path: Optional[str] = None*, *colour: Tuple[float] = (0.666667, 1, 0.498039)*, *show_wireframe: bool = False*, *show_grid: bool = False*, *cross_section: bool = False*, *show_plot: bool = True*) → Plotter

create activate 3D vtk plot of dxf that is fully interactive the dxf has to either be strings or a triangulation

Parameters

path: str
path to dxf file to visualise

colour: tuple of floats, default (0.666667, 1, 0.498039)
default solid colouring of the triangulation

show_wireframe: bool, default False
whether to show the edges/lines of a wireframe or not

show_grid: bool, default False
add x,y,z grid to see coordinates on plot

cross_section: bool, default False
add widget cross-section to plot

show_plot: bool, default True
whether to open active window or just return pyvista.Plotter object to .show() later

Returns

pyvista.Plotter object & active window of dxf visualisation

9.2.4 export_dxf_html

miningpy.visualisation.**export_dxf_html**(*path: Optional[str] = None*, *output: Optional[str] = None*, *data_name: str = 'DXF'*, *colour: Tuple[float] = (0.666667, 1, 0.498039)*) → bool

exports dxf file and embeds the data in a paraview glance html app to visualise and distribute

Parameters

path: str
path of input dxf file

output: str
path of .vtp file to export

data_name: str, default 'DXF'
base name used for dataset in Paraview Glance

colour: tuple of floats, default (0.666667, 1, 0.498039)
default solid colouring of the triangulation - RGB ranges 0 to 1

Returns

True if .html file is exported with no errors

9.2.5 dxf2vtk

miningpy.visualisation.**dxf2vtk**(*path: Optional[str] = None*, *output: Optional[str] = None*, *colour: Tuple[float] = (0.666667, 1, 0.498039)*) → PolyData

save dxf to .vtp (vtk polydata) file format so that it can be opened in paraview for external viewing

Parameters

path: str
path of input dxf file

output: str
path of html file to export

colour: tuple of floats, default (0.666667, 1, 0.498039)
default solid colouring of the triangulation

Returns

exports .vtp file and returns pyvista.PolyData object

9.2.6 blocks2vtk

miningpy.visualisation.**blocks2vtk**(*blockmodel: DataFrame*, *path: Optional[str] = None*, *xyz_cols: Tuple[str, str, str] = ('x', 'y', 'z')*, *dims: Optional[Tuple[Union[int, float], Union[int, float], Union[int, float]]] = None*, *rotation: Tuple[Union[int, float], Union[int, float], Union[int, float]] = (0, 0, 0)*, *cols: Optional[List[str]] = None*, *output_file: bool = True*) → vtkUnstructuredGrid

exports blocks and attributes of block model to a vtk file to visualise in paraview

Parameters

blockmodel: pd.DataFrame
pandas dataframe of block model

path: str
filename for vtk file

xyz_cols: tuple of strings, default ('x', 'y', 'z')
names of x,y,z columns in model

dims: tuple of floats or ints
x,y,z dimension of regular parent blocks

rotation: tuple of floats or ints, default (0, 0, 0)
rotation of block model grid around x,y,z axis, -180 to 180 degrees

cols: list of str
columns of attributes to visualise using vtk. If None then exports all columns

output_file: bool, default True
whether to output .vtu (vtk unstructured grid file) or to just return vtu object to user

Returns

vtkUnstructredGrid object if .vtu file is exported with no errors

9.2.7 blocks2dxf

miningpy.visualisation.**blocks2dxf**(*blockmodel: DataFrame*, *path: Optional[str] = None*, *dxf_split: Optional[str] = None*, *facetype: str = '3DFACE'*, *xyz_cols: Tuple[str, str, str] = ('x', 'y', 'z')*, *dims: Optional[Tuple[Union[int, float], Union[int, float], Union[int, float]]] = None*, *rotation: Tuple[Union[int, float], Union[int, float], Union[int, float]] = (0, 0, 0)*) → bool

exports blocks and attributes of block model to a vtk file to visualise in paraview

Parameters

blockmodel: pd.DataFrame
pandas dataframe of block model

path: str
filename for dxf files. If multiple dxfs produced, this will be used as the file suffix

dxf_split: str
column to split dxf files by. for example, could be the year mined column from minemax if None then one dxf is made of every block in blockmodel

facetype: {'3DFACE', 'MESH', None}, default '3DFACE'
type of face for the blocks 3DFACE will create standard dxf faces which are understood by most software MESH is a newer type which requires less space but might not work well None will create no face (could be useful when we add line drawing functionality in the function)

xyz_cols: tuple of strings, default ('x', 'y', 'z')
names of x,y,z columns in model

dims: tuple of floats or ints
x,y,z dimension of regular parent blocks

rotation: tuple of floats or ints, default (0, 0, 0)
rotation of block model grid around x,y,z axis, -180 to 180 degrees

Returns

True if .dxf file(s) are exported with no errors

9.2.8 face_position_dxf

miningpy.visualisation.**face_position_dxf**()

9.3 Maptek Vulcan Integration

This page describes the functions built into MiningPy for integration with Maptek Vulcan

9.3.1 vulcan_dir_to_path

miningpy.vulcan.**vulcan_dir_to_path**(*parent_path*)

adds Vulcan installation directory to path so that the correct executables can be found.

9.3.2 bdf_from_bmf

miningpy.vulcan.**bdf_from_bmf**(*input_bmf_path*, *output_bdf_path*)

Use this to create a new block model definition file (.bdf) from an existing block model.

9.3.3 create_bdf

miningpy.vulcan.**create_bdf**(*output_bdf_path*)

Create a basic regular Vulcan bdf that defines a block model framework. Used in conjunction with the csv_to_bmf / pandas_to_bmf functions when creating a bmf file.

9.3.4 bmf_to_csv

miningpy.vulcan.**bmf_to_csv**(*input_bmf_path*, *output_csv_path*, *bounding_triangulation=None*, *mask_variable=[None, None]*, *test_condition=None*)

Use this to export block values to a nominated CSV file. The resulting CSV file can be edited through software packages that are familiar with the CSV format, such as Microsoft Excel and Microsoft Access. The resulting file will be named after the original block model and stored in the same directory.

9.3.5 csv_to_bmf

miningpy.vulcan.**csv_to_bmf**(*input_bmf_path*)

9.3.6 pandas_to_bmf

miningpy.vulcan.**pandas_to_bmf**(*input_bmf_path*)

9.3.7 bmf_to_pandas

miningpy.vulcan.**bmf_to_pandas**(*input_bmf_path*, *remove_header_rows=True*, *bounding_triangulation=None*, *mask_variable=[None, None]*, *test_condition=None*)

Read a regular block model file directly to a pandas dataframe in a single function

9.3.8 mine_block_model

miningpy.vulcan.**mine_block_model**(*input_bmf_path*)

Use this to report on the proportion of a block that falls in a nominated triangulation(s). The 'mined-out' value, which can be reported as a percentage or fraction, will be written to a specified block model variable.

9.3.9 set_variable

miningpy.vulcan.**set_variable**(*input_bmf_path*, *variable*, *value*)

Use this to set a numeric block model variable. i.e. for resetting to 0 before flagging, etc…

9.3.10 create_variable

miningpy.vulcan.**create_variable**(*input_bmf_path*, *variable*, *value*)

Use this to create a numeric block model variable. Data type may be "float" or "int".

9.3.11 strings_to_dxf

miningpy.vulcan.**strings_to_dxf**()

Use this to export the design data contained in a Vulcan design database into a nominated drawing file (.dwg and .dxf).

9.3.12 dxf_to_strings

miningpy.vulcan.**dxf_to_strings**()

Use this to export the design data contained in a Vulcan design database into a nominated drawing file (.dwg and .dxf).

9.4 Block Model Interactive

The interactive HTML app above can be created using the *export_html* function:

```
import pandas as pd
import miningpy

# block model example (zuck small - MineLib)
url = "https://drive.google.com/uc?export=download&id=1SOrYhqiu5Tg8Zjb7be4fUWhbFDTU1sEk"

# read in block model from link
blockModel = pd.read_csv(url, compression='zip')

blockModel.export_html(
    path='blockmodel.html',
```

(continues on next page)

(continued from previous page)

```
    xyz_cols=('x', 'y', 'z'),
    dims=(1, 1, 1),  # block dimensions (5m * 5m * 5m)
)
```

9.5 DXF Triangulation Interactive

The interactive HTML app above can be created using the *export_dxf_html* function:

```
import requests
import hashlib
import os
import miningpy

# dxf wireframe example
url = "https://drive.google.com/uc?export=download&id=1RyaNDSV4K_LrjoIiJrFZ4KAbzj7iySuh"

dxf_raw = requests.get(url).text
dxf_raw = dxf_raw.replace("\r\n", "\n")

temp_file = 'examples/' + hashlib.md5(dxf_raw.encode(encoding='UTF-8')).hexdigest()

# create temporary dxf file for ezdxf
with open(temp_file, 'w') as file:
    file.write(dxf_raw)

# read temporary dxf file and plot using MiningPy function
miningpy.export_dxf_html(temp_file, output='dxf_triangulation.html')

# delete temporary dxf file that was downloaded
os.remove(temp_file)
```

9.6 DXF Design Strings Interactive

The interactive HTML app above can be created using the *export_dxf_html* function:

```
import requests
import hashlib
import os
import miningpy

# dxf design strings example
```

(continues on next page)

(continued from previous page)

```
url = "https://drive.google.com/uc?export=download&id=1B5GwGnkbdC_Q2RRKnVQ05F-RLzWiPoFt"

dxf_raw = requests.get(url).text
dxf_raw = dxf_raw.replace("\r\n", "\n")

temp_file = 'examples/' + hashlib.md5(dxf_raw.encode(encoding='UTF-8')).hexdigest()

# create temporary dxf file for ezdxf
with open(temp_file, 'w') as file:
    file.write(dxf_raw)

# read temporary dxf file and plot using MiningPy function
miningpy.export_dxf_html(temp_file, output='dxf_strings.html')

# delete temporary dxf file that was downloaded
os.remove(temp_file)
```

INDEX