pygeomod Documentation

Release 0.2

Florian Wellmann

August 08, 2014

CONTENTS

Contents:

Contents

CHAPTER

ONE

PYGEOMOD

1.1 Python wrapper for Geomodeller

pygeomod is a Python package with several modules to manipulate and control geological models created with Geomodeller. It contains methods to automate model computation and export (for structured and unstructured grids), to change input files, and to adapt exported and imported data sets around geological models.

The current version of the repository can be accessed on github:

https://github.com/flohorovicic/pygeomod

No guarantee for any working order :-) However, if you have any questions, feel free to email me!

1.2 Getting Started

The best way to get started (after installation, that is) is to go through the IPython tutorial and example notebooks. I will try to keep them as self-consistent as possible (with all data sets provided in the package, as well).

1.3 Documentation

The best way to view the documentation online is through the ReadTheDocs page:

http://pygeomod.readthedocs.org/en/latest/

The documentation is also available in PDF and HTML versions as part of the repository: HTML: https://github.com/flohorovicic/pygeomod/tree/master/doc/*build/html/index.html PDF: https://github.com/flohorovicic/pygeomod/tree/master/doc/*build/latex/pygeomod.pdf

1.4 Installation

Simplest way (if you have pip installed): download from Python Package Index and install:

pip install pygeomod

For more information on pip, see http://www.pip-installer.org/en/latest/quickstart.html

NOTE: to be on the safe side, it is a good idea to install experiemntal Python packages into a virtual environment first:

https://python-packaging-user-guide.readthedocs.org/en/latest/tutorial.html

Alternatively, you can download the distribution (either from PyPI or from github) and simply run python setup.py install

1.5 License

pygeomodeller is free software and published under an MIT License (basically, you can do anything you want with the code, as long as you provide attribution back to me and don't hold me liable).

If you use this work in any form in a publication, please cite our article as reference:

J. Florian Wellmann, Stefan Finsterle, Adrian Croucher, Integrating Structural Geological Data into the Inverse Modelling Framework of iTOUGH2, Computers & Geosciences, Available online 31 October 2013, ISSN 0098-3004, http://dx.doi.org/10.1016/j.cageo.2013.10.014.

The manuscript is available online at: http://www.sciencedirect.com/science/article/pii/S0098300413002781

Disclaimer: the code is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Have fun!

CHAPTER

TWO

ANALYSIS AND MODIFICATION OF EXPORTED 3D STRUCTURAL DATA

All structural data from an entire GeoModeller project can be exported into ASCII files using the function in the GUI:

Export -> 3D Structural Data

This method generates files for defined geological parameters:

- "Points" (i.e. formation contact points) and
- "Foliations" (i.e. orientations/ potential field gradients).

Exported parameters include all those defined in sections as well as 3D data points.

The `struct_data` package contains methods to check, visualise, and extract/modify parts of these exported data sets, for example to import them into a different Geomodeller project. Several of these methods will be explained and tested in this notebook.

The export function generates files with the endings `_Foliations.csv` and `_Points.csv`. Classes for both of these data sets are available to handle them accordingly.

```
# import module
import sys
# as long as not in Pythonpath, we have to set directory:
sys.path.append(r'/Users/flow/git/pygeomod')
import struct_data
```

```
%pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

```
WARNING: pylab import has clobbered these variables: ['norm']
%matplotlib prevents importing * from pylab and numpy
```

2.1 Point data sets

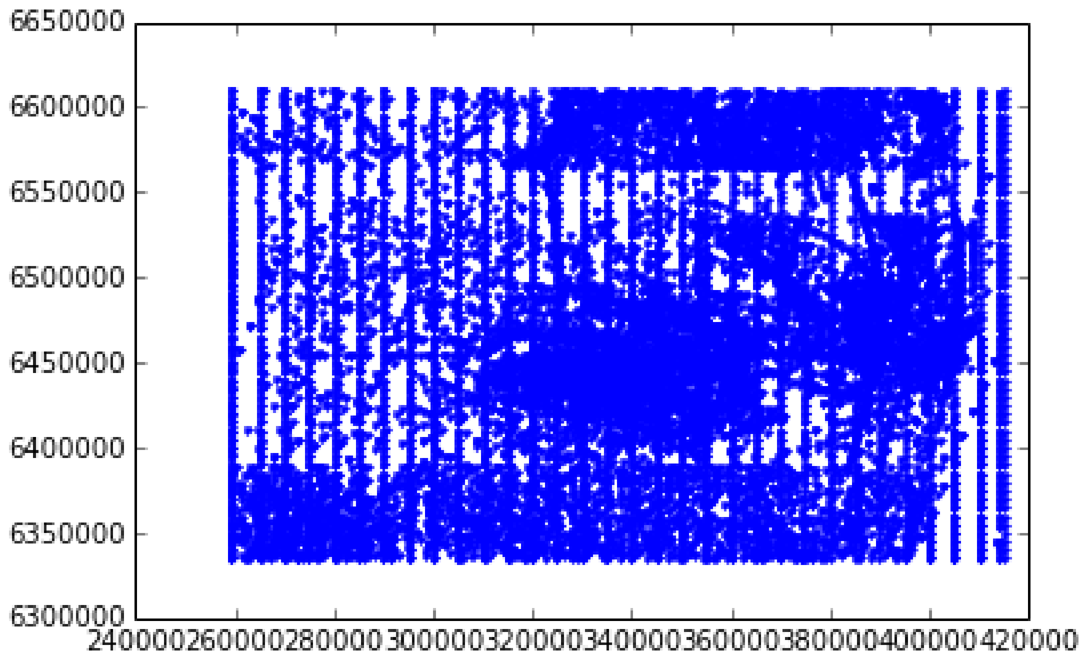
We will first start with reading and analysing the point data sets:

```
# only required during testing stage
reload(struct_data)
# use example data
pts = struct_data.Struct3DPoints(filename = r'../data/wt_Points.csv')
```

```
pts.get_formation_names()
pts.formation_names
```

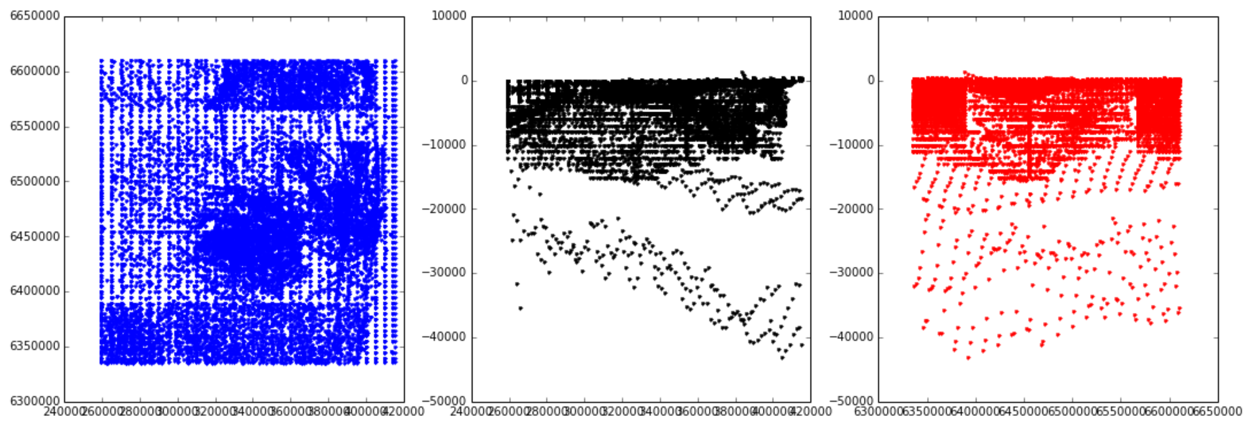
```
array(['Basement', 'CPB_Fault_01', 'CPB_Fault_02', 'CPB_Fault_03',
       'CPB_Fault_04', 'CPB_Fault_05', 'CPB_Fault_06', 'CPB_Fault_07',
       'CPB_Fault_08', 'CPB_Fault_09', 'CPB_Fault_10',
       'CPB_Fault_11_Serpentine', 'CPB_Fault_12_Darling', 'CPB_Fault_13',
       'CPB_Fault_14', 'CPB_Fault_15', 'CPB_Fault_16',
       'CPB_Fault_Transverse_02', 'CPB_Fault_Transverse_03',
       'CPB_Fault_Transverse_04', 'CPB_Fault_Transverse_05',
       'Cattamarra_Coal_Measures', 'Eneabba_Fm', 'Gage_Ss',
       'Kockatea_Shale', 'Late_Triassic', 'Leederville_Fm', 'Lesueur_Ss',
       'Lower_crust', 'Moho', 'Neocomian_Unc', 'Permian', 'Sea_level',
       'South_Perth_Sh', 'Topo', 'Yarragadee_Fm', 'Yilgarn'],
      dtype='|S32')
```

```
pts.plot_plane()
```



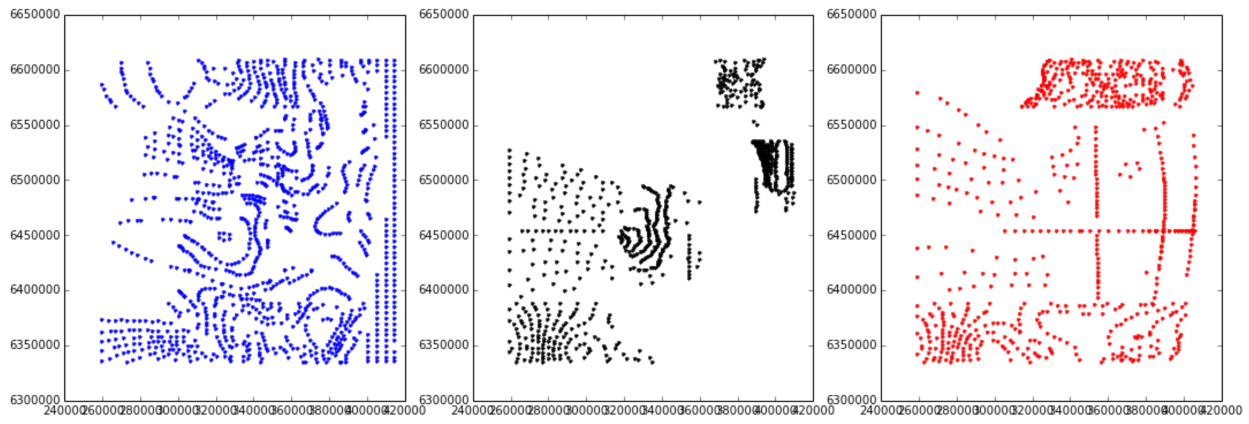
Or, a bit more complex plotting options:

```
fig = plt.figure(figsize = (18,6))
ax1 = fig.add_subplot(131)
ax2 = fig.add_subplot(132)
ax3 = fig.add_subplot(133)
pts.plot_plane(ax = ax1)
pts.plot_plane(plane = ('x','z'), ax = ax2, color = 'k')
pts.plot_plane(plane = ('y','z'), ax = ax3, color = 'r')
```



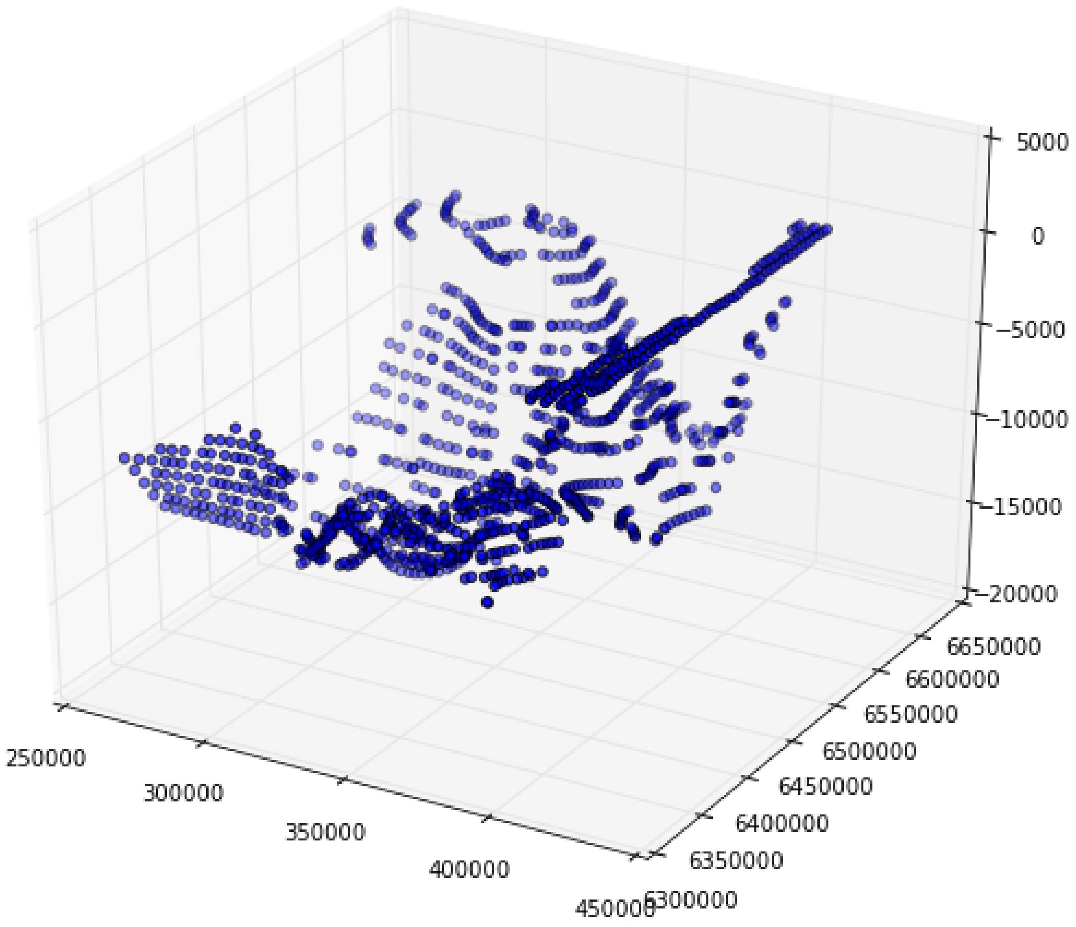
We can also create plots of points for specified formations only:

```
fig = plt.figure(figsize = (18,6))
ax1 = fig.add_subplot(131)
ax2 = fig.add_subplot(132)
ax3 = fig.add_subplot(133)
pts.plot_plane(ax = ax1, formation_names = ['Basement'])
pts.plot_plane(ax = ax2, formation_names = ['Yarragadee_Fm'], color = 'k')
pts.plot_plane(ax = ax3, formation_names = ['Lesueur_Ss'], color = 'r')
```



It is also possible to create 3-D perspective plots (not many options yet, but a lot more could be included):

```
pts.plot_3D(formation_names = ['Basement'])
```



```
pts.formation_names
```

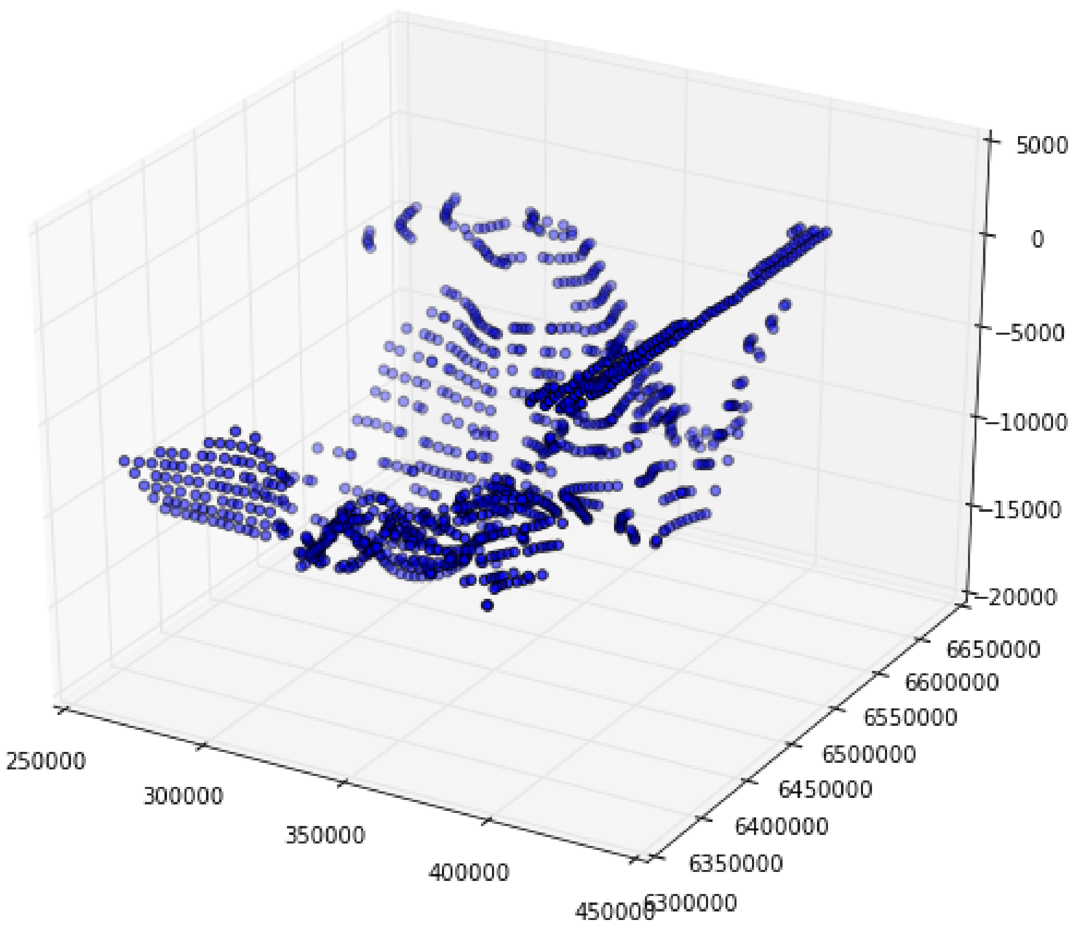
```
array(['Basement', 'CPB_Fault_01', 'CPB_Fault_02', 'CPB_Fault_03',
       'CPB_Fault_04', 'CPB_Fault_05', 'CPB_Fault_06', 'CPB_Fault_07',
       'CPB_Fault_08', 'CPB_Fault_09', 'CPB_Fault_10',
       'CPB_Fault_11_Serpentine', 'CPB_Fault_12_Darling', 'CPB_Fault_13',
       'CPB_Fault_14', 'CPB_Fault_15', 'CPB_Fault_16',
       'CPB_Fault_Transverse_02', 'CPB_Fault_Transverse_03',
       'CPB_Fault_Transverse_04', 'CPB_Fault_Transverse_05',
       'Cattamarra_Coal_Measures', 'Eneabba_Fm', 'Gage_Ss',
       'Kockatea_Shale', 'Late_Triassic', 'Leederville_Fm', 'Lesueur_Ss',
       'Lower_crust', 'Moho', 'Neocomian_Unc', 'Permian', 'Sea_level',
       'South_Perth_Sh', 'Topo', 'Yarragadee_Fm', 'Yilgarn'],
      dtype='|S32')
```

Generate subset for defined formations

It is often required to generate a subset of the points, either for specified formations only (although that could also be done during the input in GeoModeller...), or for a defined range/ extent. These options are possible with the package, and selections can be stored to new files:

```
# only required during testing stage
reload(struct_data)
# use example data
```

```
pts = struct_data.Struct3DPoints(filename = r'../data/wt_Points.csv')
pts_subset = pts.create_formation_subset(['Basement'])

pts_subset.len

1209

pts_subset.plot_3D()
```



Generating a subset for a subvolume

It is also possible to extract data in only a specified range. The range is defined with keywords, e.g. `from_x = xx`, `to_x = xx`. All other ranges (not stated) are simply kept as before.

Here an example:

```
# only required during testing stage
reload(struct_data)
# use example data
pts = struct_data.Struct3DPoints(filename = r'../data/wt_Points.csv')

pts.xmin, pts.xmax, pts.ymin, pts.ymax
```

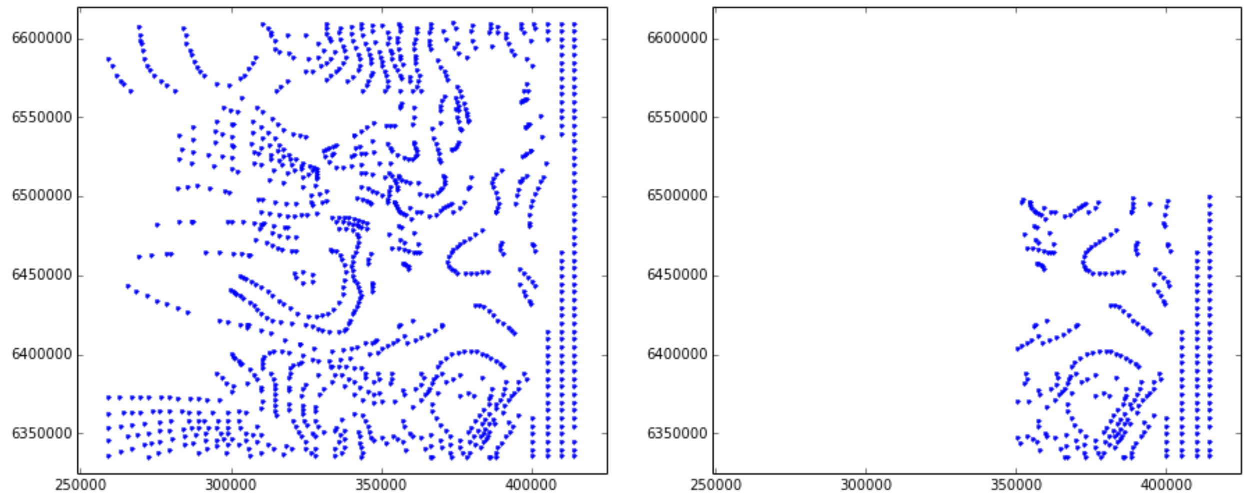
```
(259000.0, 415000.0, 6335020.0, 6610000.0)
```

```
pts_subset = pts.extract_range(from_x = 350000., to_y = 6500000)
```

Now let's compare the two sets in a plot:

```
fig = plt.figure(figsize = (15,6))
ax1 = fig.add_subplot(121)
ax2 = fig.add_subplot(122)
pts.plot_plane(ax = ax1, formation_names = ['Basement'])
pts_subset.plot_plane(ax = ax2, formation_names = ['Basement'])
# set ranges to compare plots
ax1.set_xlim((pts.xmin-10000, pts.xmax+10000))
ax1.set_ylim((pts.ymin-10000, pts.ymax+10000))
ax2.set_xlim((pts.xmin-10000, pts.xmax+10000))
ax2.set_ylim((pts.ymin-10000, pts.ymax+10000))
```

```
(6325020.0, 6620000.0)
```



Thin data

In many cases, a lot of data has been digitised for specific locations. This can lead to problems with the kriging interpolation in Geomodeller. A simple function to thin data is implemented here to avoid this problem. The method is really simple and can be quite compute intense (for smalll grids), so check what you are doing!

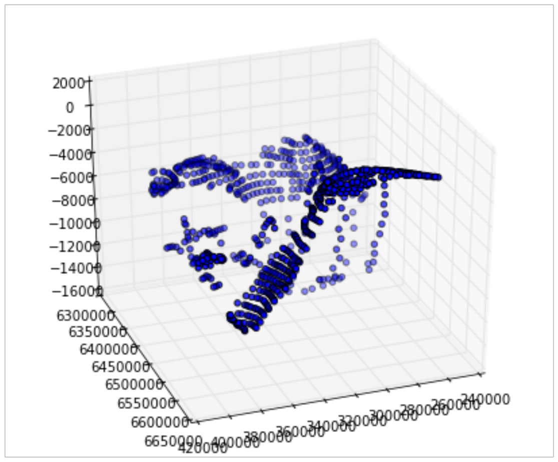
Note: If spatial data is imported, Geomodeller actually has a functionality for a type of "thinning" - if you work with this data (e.g. digitised from MapInfo), then this might be the function to use...

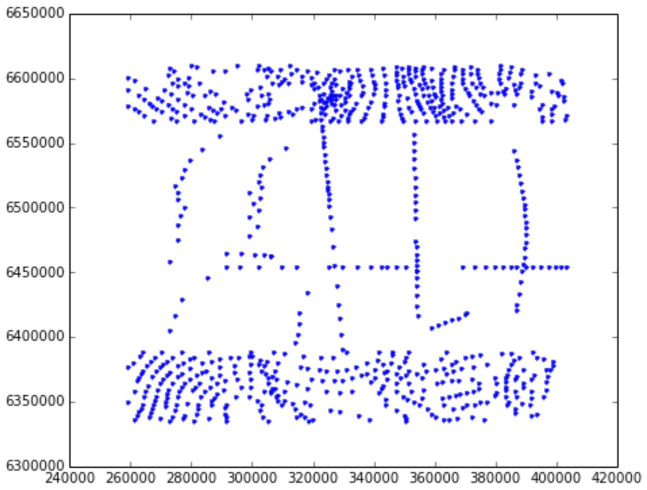
The thinning is performed for **the entire set** and **not aware of formations** (so far...) and works on a grid/ raster base for defined number of cells in each axis direction of `nx, ny, nz`. The best procedure is therefore to first create a subset for one formation, and then to perform the thinning for this subset:

```
# only required during testing stage
reload(struct_data)
# use example data
pts = struct_data.Struct3DPoints(filename = r'../data/wt_Points.csv')
pts_sub1 = pts.create_formation_subset('Permian')
```

Let's have a look at one formation, the Permian. We can clearly see the dense data in the overlapping regions and the highly defined traces through the model:

```
fig = plt.figure(figsize = (16,6))
ax1 = fig.add_subplot(121, projection='3d')
ax2 = fig.add_subplot(122)
ax1.view_init(elev=30, azim=70)
pts_sub1.plot_3D(ax = ax1)
pts_sub1.plot_plane(ax = ax2)
```



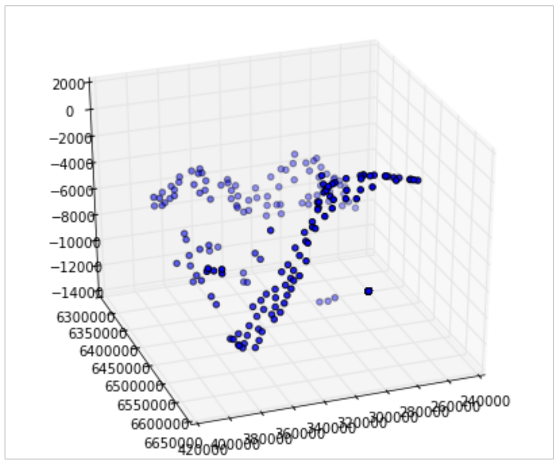


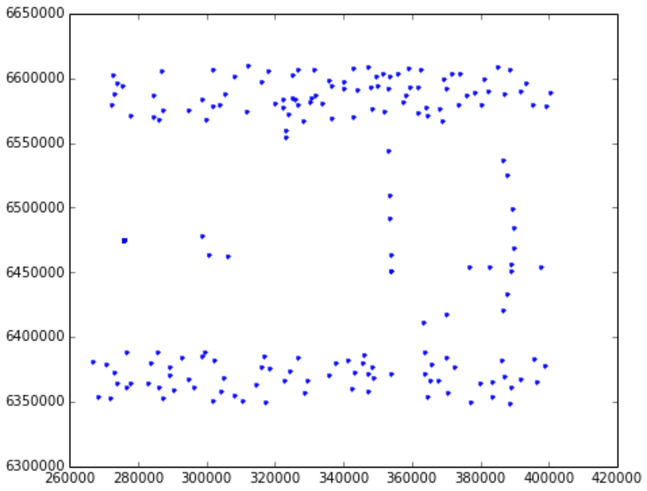
We want to thin this now with a grid which is `n` times smaller than the extent in each direction:

```
nx = 20
ny = 20
nz = 20

pts_sub2 = pts_sub1.thin(nx, ny, nz)

pts_sub1.len, pts_sub2.len

(810, 319)
```

And compare those to the set above in a plot:

```
fig = plt.figure(figsize = (16,6))
ax1 = fig.add_subplot(121, projection='3d')
ax2 = fig.add_subplot(122)
ax1.view_init(elev=30, azim=70)
pts_sub2.plot_3D(ax = ax1)
pts_sub2.plot_plane(ax = ax2)
```





Nice, it seems to work (surprised myself...)

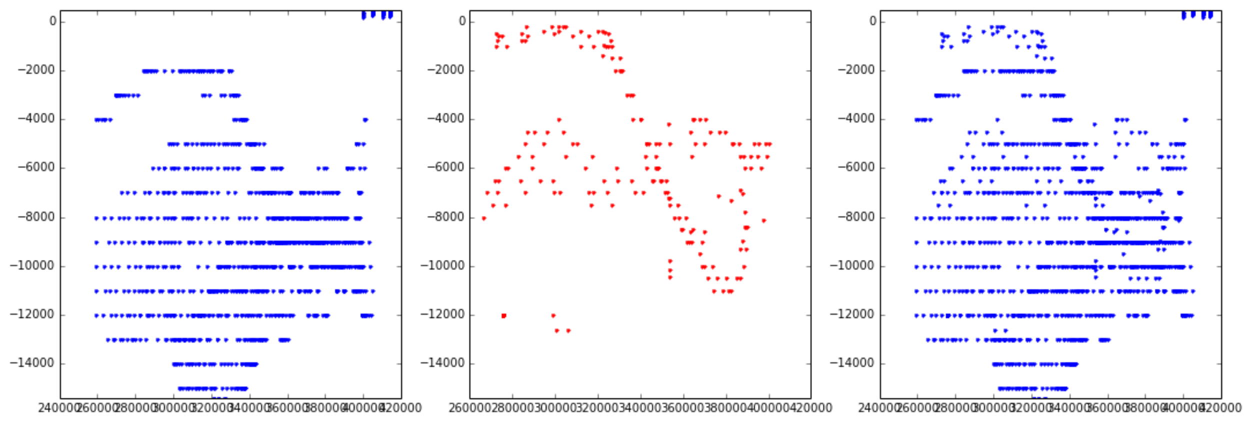
Of course, it's up to the user to evaluate if the thinning is creating "wrong" Geomodels afterwards... good luck!

Combine two point sets

It might be important to combine datasets, for example after thinning two different geological formations. As an example, we will combine the thinned Permian data set from above with the (complete) set for the Basement:

```
# only required during testing stage
reload(struct_data)
# use example data
pts = struct_data.Struct3DPoints(filename = r'../data/wt_Points.csv')
pts_sub3 = pts.create_formation_subset('Basement')
# create a copy for comparison:
pts_sub3b = pts.create_formation_subset('Basement')

pts_sub3.combine_with(pts_sub2)

# create a plot of all three point sets:
fig = plt.figure(figsize = (18,6))
ax1 = fig.add_subplot(131)
ax2 = fig.add_subplot(132)
ax3 = fig.add_subplot(133)
pts_sub3b.plot_plane(('x','z'), ax = ax1)
pts_sub2.plot_plane(('x','z'), ax = ax2, color='r')
pts_sub3.plot_plane(('x','z'), ax = ax3)
ax1.set_ylim((pts_sub3.zmin, pts_sub3.zmax))
ax2.set_ylim((pts_sub3.zmin, pts_sub3.zmax))
ax3.set_ylim((pts_sub3.zmin, pts_sub3.zmax))
```

```
(-15400.0, 454.0)
```



Note: This function changes the pointset in place and does not return a new one (as for thinning, subsetting, etc.)

Remove formation from point set

It is also possible to remove one or multiple formations from the set. This functionality can be used in combination with thinning and combining to thin only the data set for one formation, and then combine it back into the original set:

```
# only required during testing stage
reload(struct_data)
# use example data
pts = struct_data.Struct3DPoints(filename = r'../data/wt_Points.csv')

print("Original length of point set: %d" % pts.len)


# formation to perform thinning:
formation = 'Permian'

# first step: extract formation
pts_perm = pts.create_formation_subset(formation)

# now remove from original
pts.remove_formations(formation)

# perform thinning
nx = ny = nz = 20
pts_perm_thinned = pts_perm.thin(nx, ny, nz)

# and now combine back with original:
pts.combine_with(pts_perm_thinned)

print("New length of point set (after thinning): %d" % pts.len)
```

```
Original length of point set: 14184
New length of point set (after thinning): 13693
```

Save new set to file

Of course, we want to save this new and adapted data set to a file! This is also simply possible with:

```
# only required during testing stage
reload(struct_data)
# use example data
pts = struct_data.Struct3DPoints(filename = r'../data/wt_Points.csv')
```

```
# extract defined range and only Basement points:
pts_subset = pts.create_formation_subset(['Basement'])
pts_subset_1 = pts_subset.extract_range(from_x = 350000., to_y = 6500000)

pts_subset_1.save("new_Points.csv")
```

Change formation names

Sometime it is required to adjust the name of a formation or unit in the point set, for example for combination with another point set. This operation is possible with a dictionary for one or more formations with a mapping old -> new name:

```
# only required during testing stage
reload(struct_data)
# use example data
pts = struct_data.Struct3DPoints(filename = r'../data/wt_Points.csv')

print pts.formation_names

rename_dict = {'Basement' : 'Basement_new', 'CPB_Fault_01' : 'Euler'}

pts.rename_formations(rename_dict)

print pts.formation_names
```

```
['Basement' 'CPB_Fault_01' 'CPB_Fault_02' 'CPB_Fault_03' 'CPB_Fault_04'
 'CPB_Fault_05' 'CPB_Fault_06' 'CPB_Fault_07' 'CPB_Fault_08' 'CPB_Fault_09'
 'CPB_Fault_10' 'CPB_Fault_11_Serpentine' 'CPB_Fault_12_Darling'
 'CPB_Fault_13' 'CPB_Fault_14' 'CPB_Fault_15' 'CPB_Fault_16'
 'CPB_Fault_Transverse_02' 'CPB_Fault_Transverse_03'
 'CPB_Fault_Transverse_04' 'CPB_Fault_Transverse_05'
 'Cattamarra_Coal_Measures' 'Eneabba_Fm' 'Gage_Ss' 'Kockatea_Shale'
 'Late_Triassic' 'Leederville_Fm' 'Lesueur_Ss' 'Lower_crust' 'Moho'
 'Neocomian_Unc' 'Permian' 'Sea_level' 'South_Perth_Sh' 'Topo'
 'Yarragadee_Fm' 'Yilgarn']
Change name from CPB_Fault_01 to Euler
Change name from Basement to Basement_new
['Basement_new' 'CPB_Fault_02' 'CPB_Fault_03' 'CPB_Fault_04' 'CPB_Fault_05'
 'CPB_Fault_06' 'CPB_Fault_07' 'CPB_Fault_08' 'CPB_Fault_09' 'CPB_Fault_10'
 'CPB_Fault_11_Serpentine' 'CPB_Fault_12_Darling' 'CPB_Fault_13'
 'CPB_Fault_14' 'CPB_Fault_15' 'CPB_Fault_16' 'CPB_Fault_Transverse_02'
 'CPB_Fault_Transverse_03' 'CPB_Fault_Transverse_04'
 'CPB_Fault_Transverse_05' 'Cattamarra_Coal_Measures' 'Eneabba_Fm' 'Euler'
 'Gage_Ss' 'Kockatea_Shale' 'Late_Triassic' 'Leederville_Fm' 'Lesueur_Ss'
 'Lower_crust' 'Moho' 'Neocomian_Unc' 'Permian' 'Sea_level'
 'South_Perth_Sh' 'Topo' 'Yarragadee_Fm' 'Yilgarn']
```

More ideas

The methods could be used to check different geological interpretations quickly, i.e. different data sets for Moho structures, etc.: Create one model, load different data sets and compute the models.

It would also be possible to use the method directly to reproduce the “data thinning” example from Martin Putz’ 2001 paper - might be interesting as an indication for interpolation stability.

In extension: it should be possible to perform a kind of “bootstrapping” method to test interpolation uncertainty with respect to data density!

I didn’t check, but it might be possible to load 3D structural data through the API (although not sure, as a lot of checks are performed in the GUI). If this is possible, then all of the previous methods could easily be automated and/or included in model validity and uncertainty estimation steps!

And with a bit of coding:

It should be relatively simple to add some more functionality, for example to create a simple data density plot, as a “zero order” estimation of model uncertainty with respect to available data (i.e. no data, high uncertainty).

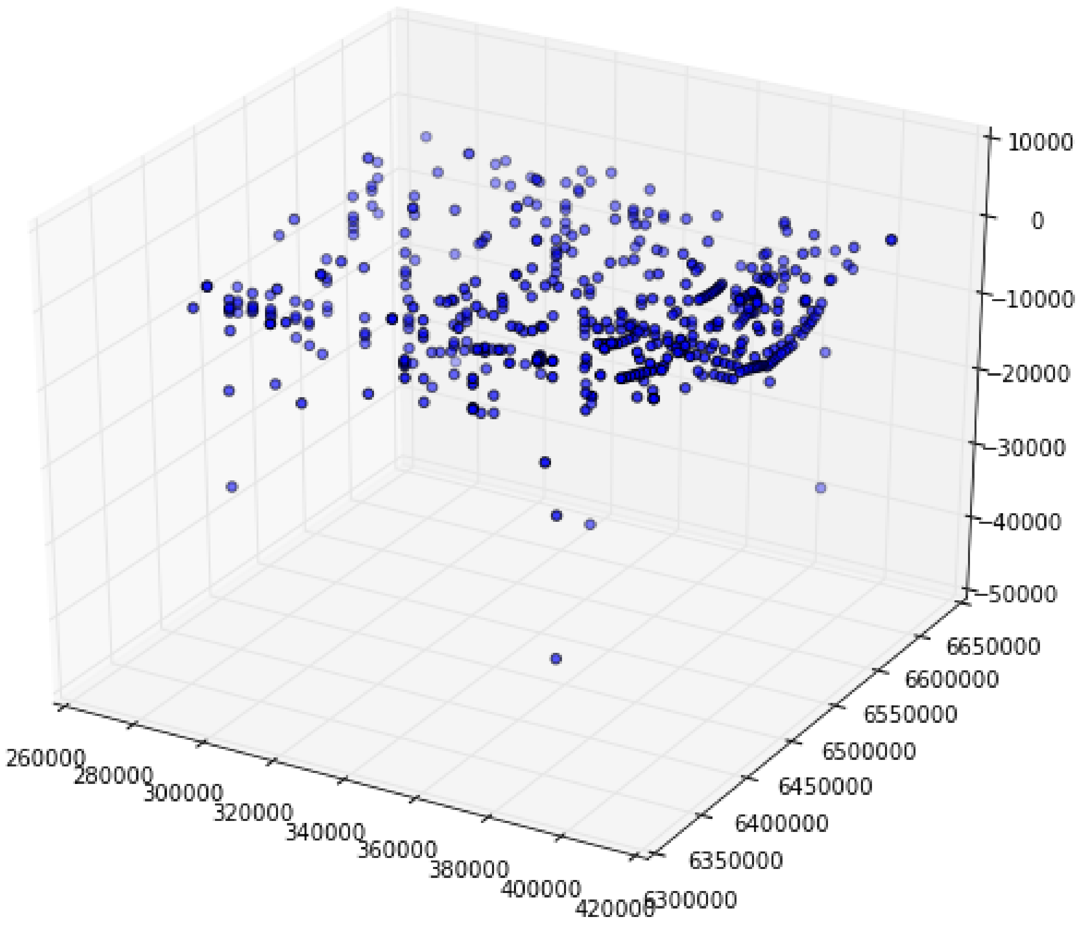
CHAPTER THREE

ORIENTATION DATA SETS

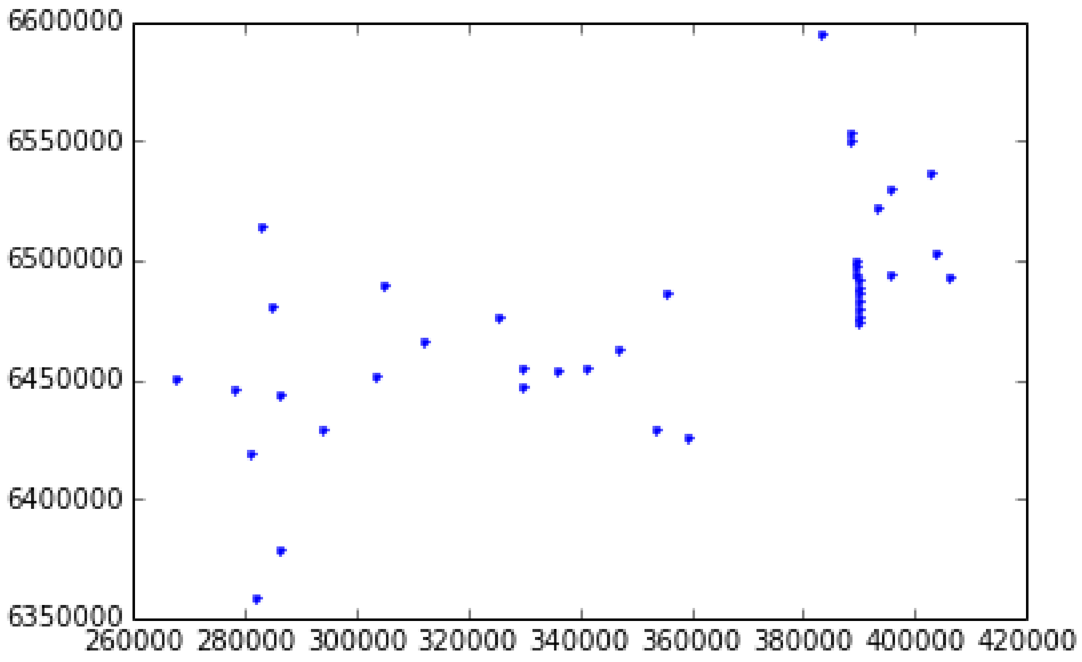
The functionality for orientation data sets is very similar to the point data sets as the main functionality is really for sorting and adapting parameters according to location and formation, and not so much for actual operations that affect the vectorial information (althoguh it might be interesting to include a vector-specific upscaling/ averaging, etc. - but this is not implemented to date).

Loading a data set is exactly as before, and the plotting commands now show the location of orientation data:

```
# only required during testing stage
reload(struct_data)
# use example data
fol = struct_data.Struct3DFoliations(filename = r'../data/wt_Foliations.csv')

fol.plot_3D()
```



```
fol.plot_plane(formation_names = 'Yarragadee_Fm')
```


6600000
6550000
6500000
6450000
6400000
6350000
260000
280000
300000
320000
340000
360000
380000
400000
420000

CHAPTER

FOUR

CREATING AN IRREGULAR MESH FROM GEOMODELLER FOR SHEMAT SIMULATIONS

Regular meshes can be exported directly from within the Geomodeller GUI. However, in many cases, a more flexible solution is required, for example to:

- update a mesh automatically (without using the GUI), or
- create an irregular mesh with refined regions

These steps can easily be performed with a set of Python scripts and C programs that access Geomodellers funcionality through the API.

The main funcionality required here is combined in the Python package `pygeomod`. Two main packages are required: `geogrid.py` is the most recent development and contains a (relatively general) class definition for rectangular grids in general, with the link to Geomodeller in particular. The package ´geomodeller_xml_obj.py‘ contains methods to access and modify information stored in the Geomodeller xml Project-files. This functionality can be used, for example, to change geological input parameters (e.g. dip of a fault) directly from the Python script.

A note on installation:

The most tricky part is to get the API properly installed, all libraries linked, and compiled on a system. On esim39, the required library path settings are defined in

`adjust_to_jni.sh`

Another important point (for now, should be fixed at some stage...) is that the shared object `libgeomod.so` has to be located in the current directory... time to write a proper make file, but to date that’s the stage the project is in.

We will first start here with an example for the generation of an rectilinear refined mesh for a simulation with SHEMAT.

```
# first step: import standard libraries and set pylab for plotting functionalities
%pylab inline
import numpy as np
import matplotlib.pyplot as plt
import sys, os

Welcome to pylab, a matplotlib-based Python environment [backend: module://IPython.zmq.pylab.backend_
For more information, type 'help(pylab)'.

# Add path to pygeomod and import module (note: this is only required because it can't be installed p
sys.path.append(r'/home/jni/git/tmp/pygeomod_tmp')
import geogrid
```

4.1 Creating a regular grid

The geogrid module contains a variety of methods to generate grids. In combinbation wtih Geomodeller, the easiest thing to do is to create a regular mesh from a Geomodeller project:

```
# Define path to geomodeller model file:
geomodel = r'/home/jni/git/tmp/geomuce/gemuce_tmp/examples/simple_three_layer/simple_three_layer.xml'

reload(geogrid) # only required for development stage - can be removed afterwards
# Now: define a GeoGrid object:
G1 = geogrid.GeoGrid()
# and set the boundaries/ model extent according to the Geomodeller model:
G1.get_dimensions_from_geomodeller_xml_project(geomodel)

# and create a regular grid for a defined number of cells in each direction:
nx = 25
ny = 2
nz = 25
G1.define_regular_grid(nx, ny, nz)

# ...and, finally, update the grid properties on the base of the Geomodeller model:
G1.update_from_geomodeller_project(geomodel)
```

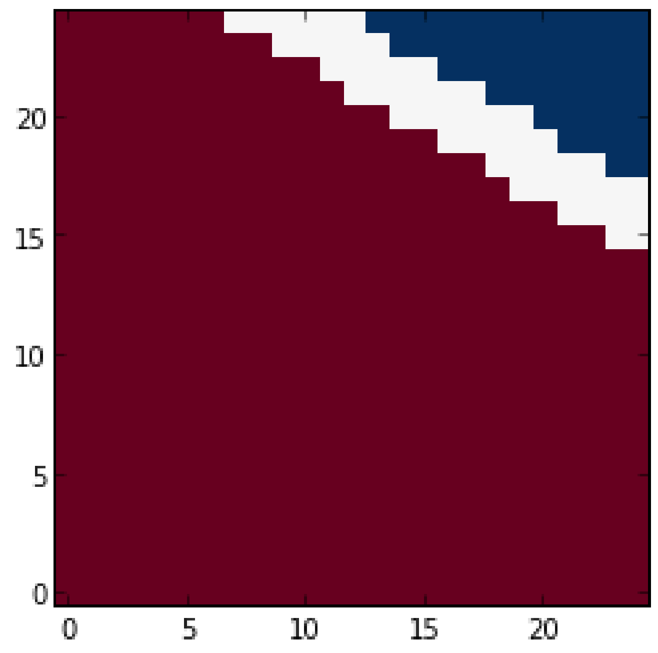
The grid is stored in the object variable `G1.grid` as a numpy array.

```
type(G1.grid)
```

```
numpy.ndarray
```

So the grid can directly be used to create slices, plots, further caluclations, etc. However, a lot of functionality is alread implemented in the geogrid package. For example, slice plots through the model can simply be generated with:

```
G1.plot_section('y', colorbar=False, cmap='RdBu') # more plotting options possible, generally follow
```



It is also possible to export the model directly to VTK - however, this requires an installation of the pyevtk package which is not installed on esim for now:

```
G1.export_to_vtk()
```

```
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)

<ipython-input-71-972ad06a1420> in <module>()
----> 1 G1.export_to_vtk()


/home/jni/git/tmp/pygeomod_tmp/geogrid.py in export_to_vtk(self, vtk_filename, real_coords, **kwds)
    327         grid = kwds.get("grid", self.grid)
    328         var_name = kwds.get("var_name", "Geology")
--> 329         from evtk.hl import gridToVTK
    330         # define coordinates
    331         x = np.zeros(self.nx + 1)


ImportError: No module named evtk.hl
```

4.2 Rectilinear grids

Creating a rectilinear grid requires only that the cell spacings are explicitly defined. Everything else is exactly the same as before. Note that it is (at the moment) your responsibility to assing proper spacings - if you go beyond the bounds of the Geomodel, the function will not crash, but return the standard Geomodeller "out" value (usually the number of stratigraphic units + 1).

One way to create meshes in the correct range is, of course, to use the extent of the Geomodel, determined with the function:

```
reload(geogrid) # only required for development stage - can be removed afterwards
# Now: define a GeoGrid object:
G1 = geogrid.GeoGrid()
# and set the boundaries/ model extent according to the Geomodeller model:
G1.get_dimensions_from_geomodeller_xml_project(geomodel)
```

```
# The extent of the Geomodeller model can be obtained with:
G1.xmin, G1.xmax
```

```
(0, 1000)
```

```
# and the extent with:
G1.extent_x
```

```
1000
```

Let's be a bit fancy and create the horizontal (x,y) grid with a core region of high refinement and increasing mesh sizes towards the boundary. First, we define the geometry:

```
core_region = 100 # m
# define cell width in core region:
cell_width_core = 25 # m
del_core = np.ones(int(core_region / cell_width_core)) * cell_width_core
# and the number of cells in the boundary regions (the innermost cell has the size of the core cells
n_boundary = 10
```

```
# now determine the boundary width on both sides of the core region:
width_boundary_x = (G1.extent_x - core_region) / 2.
width_boundary_y = (G1.extent_y - core_region) / 2.
```

A little helper function in the `geogrid` package can be used to determine an optimal cell increase factor for the boundary cells for a given width an a number of cells, and a fixed inner cell width which we take as the width of the core cells for a neat transition:

```
dx_boundary = geogrid.optimial_cell_increase(cell_width_core, n_boundary, width_boundary_x)
dy_boundary = geogrid.optimial_cell_increase(cell_width_core, n_boundary, width_boundary_y)
```

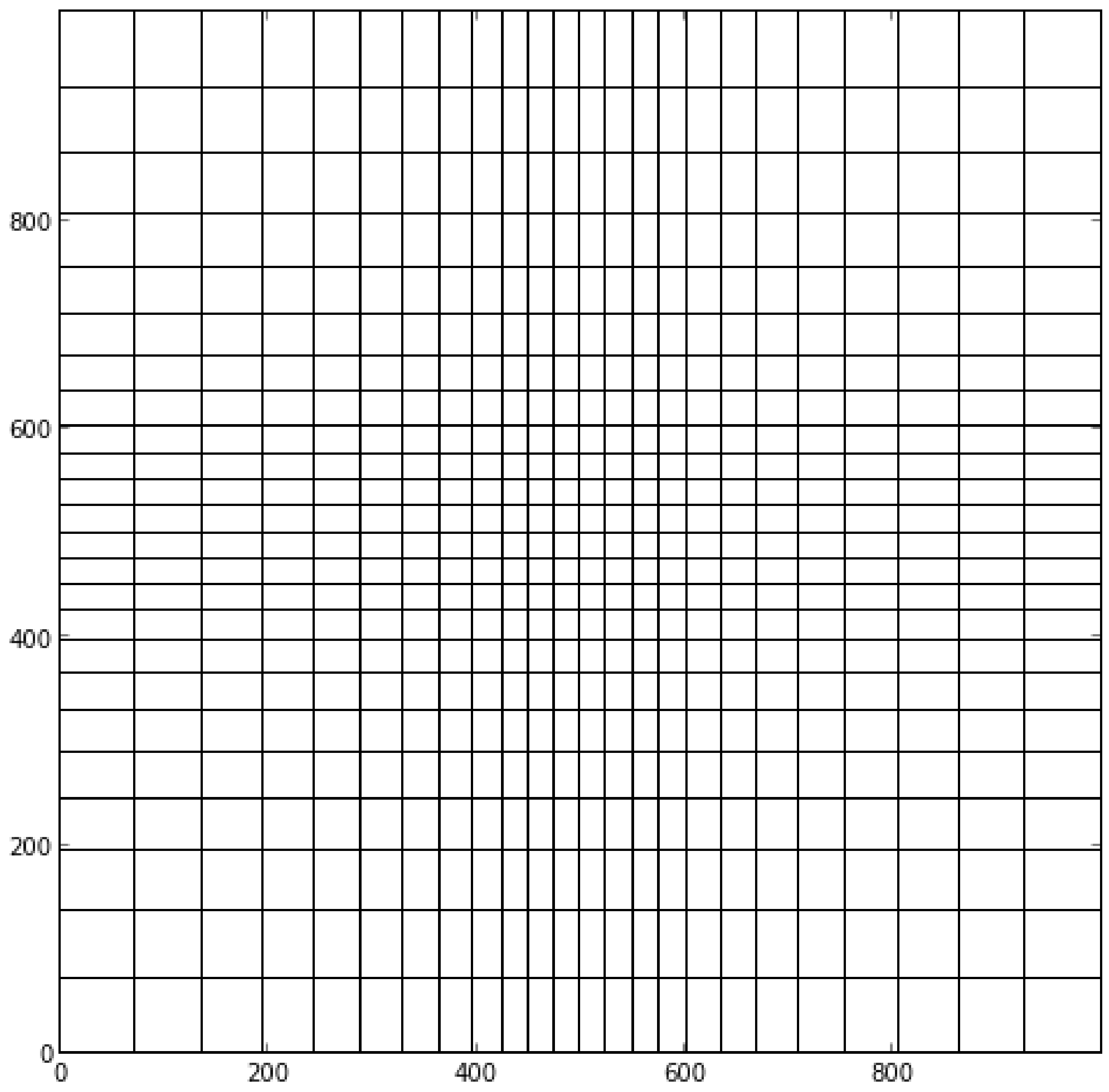
We now simply combine the boundary and core cells for the complete discretisation array:

```
delx = np.concatenate((dx_boundary[::-1], del_core, dx_boundary)) # first array reversed from large
dely = np.concatenate((dy_boundary[::-1], del_core, dy_boundary))
```

A plot of the grid:

```
fig = plt.figure(figsize = (8,8))
ax = fig.add_subplot(111)
for dx in np.cumsum(delx):
    ax.axvline(dx, color = 'k')
for dy in np.cumsum(dely):
    ax.axhline(dy, color = 'k')

ax.set_xlim((0,sum(delx)))
ax.set_ylim((0,sum(dely)))
```

```
(0, 999.99999999999864)
```



In z-direction we will create a regular mesh:

```
nz = 20
delz = np.ones(nz) * G1.extent_z / nz
```

Ok, back to the geogrid package: we now assign the cell discretisation arrays to the geogrid object and populate the grid with geology ids determined from the Geomodeller model:

```
G1.define_irregular_grid(delx, dely, delz)
G1.update_from_geomodeller_project(geomodel)

G1.grid

array([[[ 1.,  1.,  1., ...,  1.,  1.,  1.],
        [ 1.,  1.,  1., ...,  1.,  1.,  1.],
        [ 1.,  1.,  1., ...,  1.,  1.,  1.],
        ...,
        [ 1.,  1.,  1., ...,  1.,  1.,  1.],
        [ 1.,  1.,  1., ...,  1.,  1.,  1.],
        [ 1.,  1.,  1., ...,  1.,  1.,  1.]],

       [[ 1.,  1.,  1., ...,  1.,  1.,  1.],
        [ 1.,  1.,  1., ...,  1.,  1.,  1.],
        [ 1.,  1.,  1., ...,  1.,  1.,  1.],
        ...,
        [ 1.,  1.,  1., ...,  1.,  1.,  1.],
        [ 1.,  1.,  1., ...,  1.,  1.,  1.],
        [ 1.,  1.,  1., ...,  1.,  1.,  1.]],

       [[ 1.,  1.,  1., ...,  1.,  1.,  1.],
        [ 1.,  1.,  1., ...,  1.,  1.,  1.],
        [ 1.,  1.,  1., ...,  1.,  1.,  1.],
        ...,
        [ 1.,  1.,  1., ...,  1.,  1.,  1.],
        [ 1.,  1.,  1., ...,  1.,  1.,  1.],
        [ 1.,  1.,  1., ...,  1.,  1.,  1.]],

       ...,
       [[ 1.,  1.,  1., ...,  3.,  3.,  3.],
        [ 1.,  1.,  1., ...,  3.,  3.,  3.],
        [ 1.,  1.,  1., ...,  3.,  3.,  3.],
        ...,
        [ 1.,  1.,  1., ...,  3.,  3.,  3.],
        [ 1.,  1.,  1., ...,  3.,  3.,  3.],
        [ 1.,  1.,  1., ...,  3.,  3.,  3.]],

       [[ 1.,  1.,  1., ...,  3.,  3.,  3.],
        [ 1.,  1.,  1., ...,  3.,  3.,  3.],
        [ 1.,  1.,  1., ...,  3.,  3.,  3.],
        ...,
        [ 1.,  1.,  1., ...,  3.,  3.,  3.],
        [ 1.,  1.,  1., ...,  3.,  3.,  3.],
        [ 1.,  1.,  1., ...,  3.,  3.,  3.]],

       [[ 1.,  1.,  1., ...,  3.,  3.,  3.],
        [ 1.,  1.,  1., ...,  3.,  3.,  3.],
        [ 1.,  1.,  1., ...,  3.,  3.,  3.],
        ...,
        [ 1.,  1.,  1., ...,  3.,  3.,  3.],
        [ 1.,  1.,  1., ...,  3.,  3.,  3.],
        [ 1.,  1.,  1., ...,  3.,  3.,  3.]]])
```

The simple plotting functions don't work for irregular/ rectilinear grids at to date (as imshow can only plot regular grids). Export to VTK would work, in principle.

What we can do, however, is create a SHEMAT nml file (for the old SHEMAT version) directly from the grid:

```
sys.path.append(r'/home/jni/git/tmp/PySHEMAT/PySHEMAT-master')
import PySHEMAT

S1 = PySHEMAT.Shemat_file(from_geogrid = G1, nml_filename = 'updated_model.nml')
```

```
create empty file
```

CHAPTER

FIVE

PYGEOMOD

5.1 pygeomod package

5.1.1 Submodules

5.1.2 pygeomod.geogrid module

Module with classes and methods to analyse and process exported geomodel grids

Created on 21/03/2014

@author: Florian Wellmann (some parts originally developed by Erik Schaeffer)

class `pygeomod.geogrid.`**`GeoGrid`**`(`***kwds*`)`
Object definition for exported geomodel grids

`adjust_gridshape``()`
Reshape numpy array to reflect model dimensions

`define_irregular_grid``(`*delx*, *dely*, *delz*`)`
Set irregular grid according to delimter arrays in each direction

`define_regular_grid``(`*nx*, *ny*, *nz*`)`
Define a regular grid from defined project boundaries and given discretisations

`determine_cell_centers``()`
Determine cell centers for all coordinate directions in "real-world" coordinates

`determine_cell_volumes``()`
Determine cell volumes for each cell (e.g. for total formation volume calculation)

`determine_geology_ids``()`
Determine all ids assigned to cells in the grid

`determine_id_volumes``()`
Determine the total volume of each unit id in the grid

(for example for cell discretisation studies, etc.

`determine_indicator_grids``()`
Determine indicator grids for all geological units

`export_to_vtk``(`*vtk_filename='geo_grid'*, *real_coords=True*, ***kwds*`)`
Export grid to VTK for visualisation

Arguments:

- *vtk_filename* = string : vtk filename (obviously...)

- *real_coords* = bool : model extent in "real world" coordinates

Optional Keywords:

- *grid* = numpy grid : grid to save to vtk (default: self.grid)
- *var_name* = string : name of variable to plot (default: Geology)

Note: requires pyevtk, available at: https://bitbucket.org/pauloh/pyevtk

get_dimensions_from_geomodeller_xml_project(*xml_filename*)
Get grid dimensions from Geomodeller project

Arguments:

- *xml_filename* = string: filename of Geomodeller XML file

get_name_mapping_from_dict(*unit_name_dict*)
Get the name mapping directly from a dictionary

Arguments:

- *unit_name_dict* = dict with "name" : unit_id (int) pairs

get_name_mapping_from_file(*filename*)
Get the mapping between unit_ids in the model and real geological names from a csv file (e.g. the SHEMAT property file)

Arguments:

- *filename* = string : filename of csv file with id, name entries

load_delxyz(*delxyz_filename*)
Load grid discretisation from file

load_dimensions(*dimensions_filename*)
Load project dimensions from file

load_grid()
Load exported grid, discretisation and dimensions from file

plot_section(*direction*, *cell_pos='center'*, ***kwds*)
Plot a section through the model in a given coordinate direction

Arguments:

- *direction* = 'x', 'y', 'z' : coordinate direction for section position
- *cell_pos* = int/'center','min','max' : cell position, can be given as

value of cell id, or as 'center' (default), 'min', 'max' for simplicity

Optional Keywords:

- *cmap* = mpl.colormap : define colormap for plot (default: jet)
- *colorbar* = bool: attach colorbar (default: True)
- *rescale* = bool: rescale color bar to range of visible slice (default: False)
- *ve* = float : vertical exageration (for plots in x,y-direction)
- *figsize* = (x,y) : figsize settings for plot
- ***ax* = matplotlib.axis** [add plot to this axis (default: new axis)] if axis is defined, the axis is returned and the plot not shown Note: if ax is passed, colorbar is False per default!
- *savefig* = bool : save figure to file (default: show)

- *fig_filename* = string : filename to save figure

print_unit_names_volumes()
Formatted output to STDOUT of unit names (or ids, if names are note defined) and calculated volumes

remap_ids(*mapping_dictionary*)
Remap geological unit ids to new ids as defined in mapping dictionary

Arguments:

- *mapping_dictionary* = dict : {1 : 1, 2 : 3, ...} : e.g.: retain

id 1, but map id 2 to 3 (note: if id not specified, it will be retained)

set_delxyz(*delxyz*)
Set delx, dely, delz arrays explicitly and update additional attributes

Arguments:

- *delxyz* = (delx-array, dely-array, delz-array): arrays with cell dimensions

set_dimensions(***kwds*)
Set model dimensions, if no argument provided: xmin = 0, max = sum(delx) and accordingly for y,z

Optional keywords:

- *dim* = (xmin, xmax, ymin, ymax, zmin, zmax) : set dimensions explicitly

update_from_geomodeller_project(*xml_filename*)
Update grid properties directly from Geomodeller project

Arguments:

- *xml_filename* = string: filename of Geomodeller XML file

pygeomod.geogrid.**combine_grids**(*G1*, *G2*, *direction*, *merge_type='keep_first'*, ***kwds*)
Combine two grids along one axis

..Note: this implementation assumes (for now) that the overlap is perfectly matching, i.e. grid cell sizes identical and at equal positions, or that they are perfectly adjacent!

Arguments:

- G1, G2 = GeoGrid : grids to be combined
- direction = 'x', 'y', 'z': direction in which grids are combined
- **merge_type = method to combine grid:** 'keep_first' : keep elements of first grid (default) 'keep_second' : keep elements of second grid 'random' : randomly choose an element to retain

..Note: all other dimensions must be matching perfectly!!

Optional keywords:

- *overlap_analysis* = bool : perform a detailed analysis of the overlapping area, including

mismatch. Also returns a second item, a GeoGrid with information on mismatch!

Returns:

- *G_comb* = GeoGrid with combined grid
- *G_overlap* = Geogrid with analysis of overlap (of overlap_analysis=True)

pygeomod.geogrid.**optimial_cell_increase**(*starting_cell_width*, *n_cells*, *width*)
Determine an array with optimal cell width for a defined starting cell width, total number of cells, and total width

Basically, this function optimised a factor between two cells to obtain a total width

Arguments:

- *starting_cell_width* = float : width of starting/ inner cell
- *n_cells* = int : total number of cells
- *total_width* = float : total width (sum over all elements in array)

Returns: del_array : numpy.ndarray with cell discretisations

Note: optmisation with scipy.optimize - better (analytical?) methods might exist but I can't think of them at the moment

5.1.3 pygeomod.geomodeller_xml_obj module

Class definition for GeoModeller XML-Files This version includes drillholes Specific methods are defined for the uncertainty analysis (in combination with Uncertainty_Obj module)

3. (a) Florian Wellmann, 2009-2013

class `pygeomod.geomodeller_xml_obj.`**`GeomodellerClass`**

Wrapper for GeoModeller XML-datafiles to perform all kinds of data manipulation and analysis on low level basis, e.g.: - Uncertainty Simulation - TWT to depth conversion - Data analysis, e.g. Auto-documentation

`add_to_project_name`(*s*)

add string s to project name, e.g. Number of uncertainty run

`append_drillhole_data`(*element*, *data_list*)

append data in list as drillhole data (i.e. new Row elements); element should be one of the drillhole file elements, i.e. self.drillholes["survey"], self.drillholes["collar"] or self.drillholes["geology"] list should be on correct format

`change_foliation`(*element*, ***args*)

change foliation data, argument one or more of: azimuth, dip, normalpolarity = true/false, x_coord, y_coord" or: add_dip, add_azimuth, add_x_coord, add_y_coord to add values to existing values! use if_name = and if_provenance = to add conditions!

`change_foliation_polarity`(*element*)

change polarity of foliation element

`change_formation_point_pos`(*element*, ***args*)

change position of formation point in section element arguments: x_coord, y_coord : set to this coordinates add_x_coord, add_y_coord : add values to existing coordinates use if_name = and if_provenance = to add conditions!

`create_TOUGH_formation_names`(***kwds*)

create formation names that are compatible with format required by TOUGH, i.e. String of length 5 returns and stores as dictionary with Geomodeller Names as key and TOUGH names as entry (self.tough_formation_names) simply cuts original formation name to a name length of 5; if cut name already exists: create new name, three first string values followed by two integers that are subsequently increased optional keywods: out = string : set out formation to this name (for TOUGH2: no leading spaces allowed! set to 5 chars!)

`create_fault_dict`()

create dictionary for fault elements with names as keys

`create_formation_dict`()

create dictionary for formation elements with formation names as keys

create_sections_dict()
: create dictionary for section elements with section names as keys (for easier use...)

deepcopy_tree()
: create a deep copy of original tree to restore later, e.g. for uncertainty evaluation

delete_drillhole_data(*element*)
: delete all drillhole data of an element (i.e. delete all Row elements; element should be one of self.drillholes[”survey”], self.drillholes[”collar”] or self.drillholes[”geology”]

get_drillhole_data(*element*)
: get drillhole data from result element as list element should be one of the drillhole file elements, i.e. self.drillholes[”survey”], self.drillholes[”collar”] or self.drillholes[”geology”] also performs some type conversion, etc. based on type of element (e.g. in GeologyTable: set from and to as float values!) and removes leading and tailing “”

get_drillhole_elements()
: get drillhole elements and store in dictionary

get_drillholes_old()
: get drillhole tables as elements

get_fault_parameters()
: get fault parameters out of

get_faults()
: get fault elements out of rootelement and safe as local list

get_folation_polarity(*foliation_element*)
: get polarity of foliation element; return true if Normal Polarity

get_foliation_azimuth(*foliation_element*)
: get dip of foliation element

get_foliation_dip(*foliation_element*)
: get dip of foliation element

get_foliations(*section_element*)
: get all foliation data elements from a for section

get_formation_data(*section_element*)
: not used any more! use get_formation_point_data(section_element) instead

get_formation_names()
: get formation names and return as list

get_formation_parameters()
: read formation parameters; physical properties, density, th. cond etc... store in dict

get_formation_point_data(*section_element*)
: get all formation point data elements from a for section

get_formations()
: get formation elements out of rootelement and safe as local list

get_interface_name(*interface_element*)
: get name of interface, i.e. the formation

get_model_extent()
: get extent of model returns (x_min, x_max, y_min, y_max, z_min, z_max) and saves extent in self.x_min, self.x_max, etc.

get_model_range()
get model range from model extent, e.g. for automatic mesh generation

get_name(*section_element*)
get the name of any section element (if defined)

get_points_in_sections()
Create dictionary of all points (with obs-id) in all sections

get_pole_points(*element*)

get_provenance(*element*)
get provenance of an element, return as string if not defined: returns None

get_provenance_table()
get provenance table and return as dictionary with provenance rank as key

get_provenances()
get provenance table and return as dictionary with provenance rank as key deprecated, use get_provenance_table() instead!

get_section_names()
get all section names out of local variable self.sections

get_sections()
get sections out of rootelement, safe array with section elements in local variable

get_stratigraphy_list(***kwds*)
get project stratigraphy and return as list; lowermost formation: 1 for GeoModeller dll access (this ist the formation number that is returned with the GetComputedLithologyXYZ function in the geomodeller dll optional keywords: out = string : set 'out' formation to this name (might be necessary for TOUGH2 simulation!)

load_deepcopy_tree(*deepcopy_tree*)
load tree information from deepcopied tree into object

load_geomodeller_file(*xml_file*)

plot_points(*element*)

reload_geomodeller_file(*deepcopy_tree*)
restore original tree root from deep copy of orignial tree deep copy can be created (not automatically to save memory!) with self.deepcopy_tree()

set_drillhole_data(*element*, *data_list*)
set data in list as drillhole data (i.e. delete all existing Row elements and create new ones); element should be one of the drillhole file elements, i.e. self.drillholes["survey"], self.drillholes["collar"] or self.drillholes["geology"] list should be on correct format

set_provenance_table(*provenance_dict*)
create provenance table from dictionary; can be used to extend existing provenance table as a first step to assign uncertainty values attention: old provenance table is deleted!

twt_to_depth(*sec_element*, *formula*, ***args*)
Convert all data within a section from twt to depth (including orientation data!! Input: section element with data points, conversion function as string with 't' as placeholder for twt-time, e.g. '2 * (-t) ** 2 + 18 * (-t)' ATTENTION: check if t negative optional arguments: change_dip (boolean) : change dip angle in foliation data according to first derivative of twt-to-depth formula create_plot (boolean) : create summary plot with twt and converted depth for conversion formula control

write_xml(*save_dir*)
Write elementtree to xml-file

`pygeomod.geomodeller_xml_obj.`**`beta`**`(`*a*, *b*, *size=None*`)`

The Beta distribution over `[0, 1]`.

The Beta distribution is a special case of the Dirichlet distribution, and is related to the Gamma distribution. It has the probability distribution function

$$f(x;a,b) = \frac{1}{B(\alpha,\beta)} x^{\alpha-1}(1-x)^{\beta-1},$$

where the normalisation, B, is the beta function,

$$B(\alpha,\beta) = \int_0^1 t^{\alpha-1}(1-t)^{\beta-1} dt.$$

It is often seen in Bayesian inference and order statistics.

a [float] Alpha, non-negative.

b [float] Beta, non-negative.

size [tuple of ints, optional] The number of samples to draw. The output is packed according to the size given.

out [ndarray] Array of the given shape, containing values drawn from a Beta distribution.

`pygeomod.geomodeller_xml_obj.`**`binomial`**`(`*n*, *p*, *size=None*`)`

Draw samples from a binomial distribution.

Samples are drawn from a Binomial distribution with specified parameters, n trials and p probability of success where n an integer >= 0 and p is in the interval [0,1]. (n may be input as a float, but it is truncated to an integer in use)

n [float (but truncated to an integer)] parameter, >= 0.

p [float] parameter, >= 0 and <=1.

size [{tuple, int}] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

samples [{ndarray, scalar}] where the values are all integers in [0, n].

scipy.stats.distributions.binom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Binomial distribution is

$$P(N) = \binom{n}{N} p^N (1-p)^{n-N},$$

where n is the number of trials, p is the probability of success, and N is the number of successes.

When estimating the standard error of a proportion in a population by using a random sample, the normal distribution works well unless the product p*n <=5, where p = population proportion estimate, and n = number of samples, in which case the binomial distribution is used instead. For example, a sample of 15 people shows 4 who are left handed, and 11 who are right handed. Then p = 4/15 = 27%. 0.27*15 = 4, so the binomial distribution should be used in this case.

Draw samples from the distribution:

```
>>> n, p = 10, .5 # number of trials, probability of each trial
>>> s = np.random.binomial(n, p, 1000)
# result of flipping a coin 10 times, tested 1000 times.
```

A real world example. A company drills 9 wild-cat oil exploration wells, each with an estimated probability of success of 0.1. All nine wells fail. What is the probability of that happening?

Let's do 20,000 trials of the model, and count the number that generate zero positive results.

```
>>> sum(np.random.binomial(9,0.1,20000)==0)/20000.
answer = 0.38885, or 38%.
```

`pygeomod.geomodeller_xml_obj.`**`chisquare`**(*df*, *size=None*)

Draw samples from a chi-square distribution.

When *df* independent random variables, each with standard normal distributions (mean 0, variance 1), are squared and summed, the resulting distribution is chi-square (see Notes). This distribution is often used in hypothesis testing.

df [int] Number of degrees of freedom.

size [tuple of ints, int, optional] Size of the returned array. By default, a scalar is returned.

output [ndarray] Samples drawn from the distribution, packed in a *size*-shaped array.

ValueError When *df* <= 0 or when an inappropriate *size* (e.g. `size=-1`) is given.

The variable obtained by summing the squares of *df* independent, standard normally distributed random variables:

$$Q = \sum_{i=0}^{\mathtt{df}} X_i^2$$

is chi-square distributed, denoted

$$Q \sim \chi_k^2.$$

The probability density function of the chi-squared distribution is

$$p(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2-1} e^{-x/2},$$

where Γ is the gamma function,

$$\Gamma(x) = \int_0^{-\infty} t^{x-1} e^{-t} dt.$$

NIST/SEMATECH e-Handbook of Statistical Methods

```
>>> np.random.chisquare(2,4)
array([ 1.89920014,  9.00867716,  3.13710533,  5.62318272])
```

`pygeomod.geomodeller_xml_obj.`**`exponential`**(*scale=1.0*, *size=None*)

Exponential distribution.

Its probability density function is

$$f(x; \frac{1}{\beta}) = \frac{1}{\beta} \exp(-\frac{x}{\beta}),$$

for `x > 0` and 0 elsewhere. β is the scale parameter, which is the inverse of the rate parameter $\lambda = 1/\beta$. The rate parameter is an alternative, widely used parameterization of the exponential distribution [3]_.

The exponential distribution is a continuous analogue of the geometric distribution. It describes many common situations, such as the size of raindrops measured over many rainstorms [1]_, or the time between page requests to Wikipedia [2]_.

scale [float] The scale parameter, $\beta = 1/\lambda$.

size [tuple of ints] Number of samples to draw. The output is shaped according to *size*.

`pygeomod.geomodeller_xml_obj.`**`f`**(*dfnum*, *dfden*, *size=None*)

Draw samples from a F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters should be greater than zero.

The random variate of the F distribution (also known as the Fisher distribution) is a continuous probability distribution that arises in ANOVA tests, and is the ratio of two chi-square variates.

dfnum [float] Degrees of freedom in numerator. Should be greater than zero.

dfden [float] Degrees of freedom in denominator. Should be greater than zero.

size [{tuple, int}, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. By default only one sample is returned.

samples [{ndarray, scalar}] Samples from the Fisher distribution.

scipy.stats.distributions.f [probability density function,] distribution or cumulative density function, etc.

The F statistic is used to compare in-group variances to between-group variances. Calculating the distribution depends on the sampling, and so it is a function of the respective degrees of freedom in the problem. The variable *dfnum* is the number of samples minus one, the between-groups degrees of freedom, while *dfden* is the within-groups degrees of freedom, the sum of the number of samples in each group minus the number of groups.

An example from Glantz[1], pp 47-40. Two groups, children of diabetics (25 people) and children from people without diabetes (25 controls). Fasting blood glucose was measured, case group had a mean value of 86.1, controls had a mean value of 82.2. Standard deviations were 2.09 and 2.49 respectively. Are these data consistent with the null hypothesis that the parents diabetic status does not affect their children's blood glucose levels? Calculating the F statistic from the data gives a value of 36.01.

Draw samples from the distribution:

```
>>> dfnum = 1. # between group degrees of freedom
>>> dfden = 48. # within groups degrees of freedom
>>> s = np.random.f(dfnum, dfden, 1000)
```

The lower bound for the top 1% of the samples is :

```
>>> sort(s)[-10]
7.61988120985
```

So there is about a 1% chance that the F statistic will exceed 7.62, the measured value is 36, so the null hypothesis is rejected at the 1% level.

`pygeomod.geomodeller_xml_obj.`**`gamma`**(*shape*, *scale=1.0*, *size=None*)

Draw samples from a Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, *shape* (sometimes designated "k") and *scale* (sometimes designated "theta"), where both parameters are > 0.

shape [scalar > 0] The shape of the gamma distribution.

scale [scalar > 0, optional] The scale of the gamma distribution. Default is equal to 1.

size [shape_tuple, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

out [ndarray, float] Returns one sample unless *size* parameter is specified.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1}\frac{e^{-x/\theta}}{\theta^k\Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 2. # mean and dispersion
>>> s = np.random.gamma(shape, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> y = bins**(shape-1)*(np.exp(-bins/scale) /
...                      (sps.gamma(shape)*scale**shape))
>>> plt.plot(bins, y, linewidth=2, color='r')
>>> plt.show()
```

`pygeomod.geomodeller_xml_obj.`**`geometric`**(*p*, *size=None*)

Draw samples from the geometric distribution.

Bernoulli trials are experiments with one of two outcomes: success or failure (an example of such an experiment is flipping a coin). The geometric distribution models the number of trials that must be run in order to achieve success. It is therefore supported on the positive integers, `k = 1, 2, ...`.

The probability mass function of the geometric distribution is

$$f(k) = (1-p)^{k-1}p$$

where p is the probability of success of an individual trial.

p [float] The probability of success of an individual trial.

size [tuple of ints] Number of values to draw from the distribution. The output is shaped according to *size*.

out [ndarray] Samples from the geometric distribution, shaped according to *size*.

Draw ten thousand values from the geometric distribution, with the probability of an individual success equal to 0.35:

```
>>> z = np.random.geometric(p=0.35, size=10000)
```

How many trials succeeded after a single run?

```
>>> (z == 1).sum() / 10000.
0.34889999999999999 #random
```

`pygeomod.geomodeller_xml_obj.`**`get_state`**`()`

Return a tuple representing the internal state of the generator.

For more details, see *set_state*.

out [tuple(str, ndarray of 624 uints, int, int, float)] The returned tuple has the following items:

1. the string 'MT19937'.
2. a 1-D array of 624 unsigned integer keys.
3. an integer `pos`.
4. an integer `has_gauss`.
5. a float `cached_gaussian`.

set_state

set_state and *get_state* are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

`pygeomod.geomodeller_xml_obj.`**`gumbel`**`(`*loc=0.0*, *scale=1.0*, *size=None*`)`

Gumbel distribution.

Draw samples from a Gumbel distribution with specified location and scale. For more information on the Gumbel distribution, see Notes and References below.

loc [float] The location of the mode of the distribution.

scale [float] The scale parameter of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

out [ndarray] The samples

scipy.stats.gumbel_l scipy.stats.gumbel_r scipy.stats.genextreme

probability density function, distribution, or cumulative density function, etc. for each of the above

weibull

The Gumbel (or Smallest Extreme Value (SEV) or the Smallest Extreme Value Type I) distribution is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. The Gumbel is a special case of the Extreme Value Type I distribution for maximums from distributions with "exponential-like" tails.

The probability density for the Gumbel distribution is

$$p(x) = \frac{e^{-(x - \mu)/ \beta}}{\beta} e^{ -e^{-(x - \mu)/ \beta}},$$

where μ is the mode, a location parameter, and β is the scale parameter.

The Gumbel (named for German mathematician Emil Julius Gumbel) was used very early in the hydrology literature, for modeling the occurrence of flood events. It is also used for modeling maximum wind speed and rainfall rates. It is a "fat-tailed" distribution - the probability of an event in the tail of the distribution is larger than if one used a Gaussian, hence the surprisingly frequent occurrence of 100-year floods. Floods were initially modeled as a Gaussian process, which underestimated the frequency of extreme events.

It is one of a class of extreme value distributions, the Generalized Extreme Value (GEV) distributions, which also includes the Weibull and Frechet.

The function has a mean of $\mu + 0.57721\beta$ and a variance of $\frac{\pi^2}{6}\beta^2$.

Gumbel, E. J., *Statistics of Extremes*, New York: Columbia University Press, 1958.

Reiss, R.-D. and Thomas, M., *Statistical Analysis of Extreme Values from Insurance, Finance, Hydrology and Other Fields*, Basel: Birkhauser Verlag, 2001.

Draw samples from the distribution:

```
>>> mu, beta = 0, 0.1 # location and scale
>>> s = np.random.gumbel(mu, beta, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, normed=True)
>>> plt.plot(bins, (1/beta)*np.exp(-(bins - mu)/beta)
...          * np.exp( -np.exp( -(bins - mu) /beta) ),
...          linewidth=2, color='r')
>>> plt.show()
```

Show how an extreme value distribution can arise from a Gaussian process and compare to a Gaussian:

```
>>> means = []
>>> maxima = []
>>> for i in range(0,1000) :
...    a = np.random.normal(mu, beta, 1000)
...    means.append(a.mean())
...    maxima.append(a.max())
>>> count, bins, ignored = plt.hist(maxima, 30, normed=True)
>>> beta = np.std(maxima)*np.pi/np.sqrt(6)
>>> mu = np.mean(maxima) - 0.57721*beta
>>> plt.plot(bins, (1/beta)*np.exp(-(bins - mu)/beta)
...          * np.exp(-np.exp(-(bins - mu)/beta)),
...          linewidth=2, color='r')
>>> plt.plot(bins, 1/(beta * np.sqrt(2 * np.pi))
...          * np.exp(-(bins - mu)**2 / (2 * beta**2)),
...          linewidth=2, color='g')
>>> plt.show()
```

`pygeomod.geomodeller_xml_obj.`**`hypergeometric`**(*ngood*, *nbad*, *nsample*, *size=None*)

Draw samples from a Hypergeometric distribution.

Samples are drawn from a Hypergeometric distribution with specified parameters, ngood (ways to make a good selection), nbad (ways to make a bad selection), and nsample = number of items sampled, which is less than or equal to the sum ngood + nbad.

ngood [int or array_like] Number of ways to make a good selection. Must be nonnegative.

nbad [int or array_like] Number of ways to make a bad selection. Must be nonnegative.

nsample [int or array_like] Number of items sampled. Must be at least 1 and at most `ngood + nbad`.

size [int or tuple of int] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

samples [ndarray or scalar] The values are all integers in [0, n].

scipy.stats.distributions.hypergeom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Hypergeometric distribution is

$$P(x) = \frac{\binom{m}{n}\binom{N-m}{n-x}}{\binom{N}{n}},$$

where $0 \le x \le m$ and $n + m - N \le x \le n$

for P(x) the probability of x successes, n = ngood, m = nbad, and N = number of samples.

Consider an urn with black and white marbles in it, ngood of them black and nbad are white. If you draw nsample balls without replacement, then the Hypergeometric distribution describes the distribution of black balls in the drawn sample.

Note that this distribution is very similar to the Binomial distribution, except that in this case, samples are drawn without replacement, whereas in the Binomial case samples are drawn with replacement (or the sample space is infinite). As the sample space becomes large, this distribution approaches the Binomial.

Draw samples from the distribution:

```
>>> ngood, nbad, nsamp = 100, 2, 10
# number of good, number of bad, and number of samples
>>> s = np.random.hypergeometric(ngood, nbad, nsamp, 1000)
>>> hist(s)
#   note that it is very unlikely to grab both bad items
```

Suppose you have an urn with 15 white and 15 black marbles. If you pull 15 marbles at random, how likely is it that 12 or more of them are one color?

```
>>> s = np.random.hypergeometric(15, 15, 15, 100000)
>>> sum(s>=12)/100000. + sum(s<=3)/100000.
#   answer = 0.003 ... pretty unlikely!
```

pygeomod.geomodeller_xml_obj.**laplace**(*loc=0.0*, *scale=1.0*, *size=None*)

Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).

The Laplace distribution is similar to the Gaussian/normal distribution, but is sharper at the peak and has fatter tails. It represents the difference between two independent, identically distributed exponential random variables.

loc [float] The position, μ, of the distribution peak.

scale [float] λ, the exponential decay.

It has the probability density function

$$f(x; \mu, \lambda) = \frac{1}{2\lambda} \exp\left(-\frac{|x - \mu|}{\lambda}\right).$$

The first law of Laplace, from 1774, states that the frequency of an error can be expressed as an exponential function of the absolute magnitude of the error, which leads to the Laplace distribution. For many problems in Economics and Health sciences, this distribution seems to model the data better than the standard Gaussian distribution

Draw samples from the distribution

```
>>> loc, scale = 0., 1.
>>> s = np.random.laplace(loc, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, normed=True)
>>> x = np.arange(-8., 8., .01)
>>> pdf = np.exp(-abs(x-loc/scale))/(2.*scale)
>>> plt.plot(x, pdf)
```

Plot Gaussian for comparison:

```
>>> g = (1/(scale * np.sqrt(2 * np.pi)) *
...      np.exp( - (x - loc)**2 / (2 * scale**2) ))
>>> plt.plot(x,g)
```

`pygeomod.geomodeller_xml_obj.`**`logistic`**(*loc=0.0*, *scale=1.0*, *size=None*)

Draw samples from a Logistic distribution.

Samples are drawn from a Logistic distribution with specified parameters, loc (location or mean, also median), and scale (>0).

loc : float

scale : float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

samples [{ndarray, scalar}] where the values are all integers in [0, n].

scipy.stats.distributions.logistic [probability density function,] distribution or cumulative density function, etc.

The probability density for the Logistic distribution is

$$P(x) = P(x) = \frac{e^{-(x-\mu)/s}}{s(1+e^{-(x-\mu)/s})^2},$$

where μ = location and s = scale.

The Logistic distribution is used in Extreme Value problems where it can act as a mixture of Gumbel distributions, in Epidemiology, and by the World Chess Federation (FIDE) where it is used in the Elo ranking system, assuming the performance of each player is a logistically distributed random variable.

Draw samples from the distribution:

```
>>> loc, scale = 10, 1
>>> s = np.random.logistic(loc, scale, 10000)
>>> count, bins, ignored = plt.hist(s, bins=50)
```

# plot against distribution

```
>>> def logist(x, loc, scale):
...     return exp((loc-x)/scale)/(scale*(1+exp((loc-x)/scale))**2)
>>> plt.plot(bins, logist(bins, loc, scale)*count.max()/\
... logist(bins, loc, scale).max())
>>> plt.show()
```

`pygeomod.geomodeller_xml_obj.`**`lognormal`**(*mean=0.0*, *sigma=1.0*, *size=None*)

Return samples drawn from a log-normal distribution.

Draw samples from a log-normal distribution with specified mean, standard deviation, and array shape. Note that the mean and standard deviation are not the values for the distribution itself, but of the underlying normal distribution it is derived from.

mean [float] Mean value of the underlying normal distribution

sigma [float, > 0.] Standard deviation of the underlying normal distribution

size [tuple of ints] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

samples [ndarray or float] The desired samples. An array of the same shape as *size* if given, if *size* is None a float is returned.

scipy.stats.lognorm [probability density function, distribution,] cumulative density function, etc.

A variable *x* has a log-normal distribution if *log(x)* is normally distributed. The probability density function for the log-normal distribution is:

$$p(x) = \frac{1}{\sigma x \sqrt{2\pi}} e^{(-\frac{(ln(x)-\mu)^2}{2\sigma^2})}$$

where μ is the mean and σ is the standard deviation of the normally distributed logarithm of the variable. A log-normal distribution results if a random variable is the *product* of a large number of independent, identically-distributed variables in the same way that a normal distribution results if the variable is the *sum* of a large number of independent, identically-distributed variables.

Limpert, E., Stahel, W. A., and Abbt, M., "Log-normal Distributions across the Sciences: Keys and Clues," *BioScience*, Vol. 51, No. 5, May, 2001. http://stat.ethz.ch/~stahel/lognormal/bioscience.pdf

Reiss, R.D. and Thomas, M., *Statistical Analysis of Extreme Values*, Basel: Birkhauser Verlag, 2001, pp. 31-32.

Draw samples from the distribution:

```
>>> mu, sigma = 3., 1. # mean and standard deviation
>>> s = np.random.lognormal(mu, sigma, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='mid')

>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
...        / (x * sigma * np.sqrt(2 * np.pi)))

>>> plt.plot(x, pdf, linewidth=2, color='r')
>>> plt.axis('tight')
>>> plt.show()
```

Demonstrate that taking the products of random samples from a uniform distribution can be fit well by a log-normal probability density function.

```
>>> # Generate a thousand samples: each is the product of 100 random
>>> # values, drawn from a normal distribution.
>>> b = []
>>> for i in range(1000):
...    a = 10. + np.random.random(100)
...    b.append(np.product(a))

>>> b = np.array(b) / np.min(b) # scale values to be positive
>>> count, bins, ignored = plt.hist(b, 100, normed=True, align='center')
>>> sigma = np.std(np.log(b))
>>> mu = np.mean(np.log(b))
>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
...        / (x * sigma * np.sqrt(2 * np.pi)))

>>> plt.plot(x, pdf, color='r', linewidth=2)
>>> plt.show()
```

`pygeomod.geomodeller_xml_obj.`**`logseries`**`(`*p*, *size=None*`)`

Draw samples from a Logarithmic Series distribution.

Samples are drawn from a Log Series distribution with specified parameter, p (probability, 0 < p < 1).

loc : float

scale : float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

samples [{ndarray, scalar}] where the values are all integers in [0, n].

scipy.stats.distributions.logser [probability density function,] distribution or cumulative density function, etc.

The probability density for the Log Series distribution is

$$P(k) = \frac{-p^k}{k \ln(1-p)},$$

where p = probability.

The Log Series distribution is frequently used to represent species richness and occurrence, first proposed by Fisher, Corbet, and Williams in 1943 [2]. It may also be used to model the numbers of occupants seen in cars [3].

Draw samples from the distribution:

```
>>> a = .6
>>> s = np.random.logseries(a, 10000)
>>> count, bins, ignored = plt.hist(s)
```

# plot against distribution

```
>>> def logseries(k, p):
...     return -p**k/(k*log(1-p))
>>> plt.plot(bins, logseries(bins, a)*count.max()/
             logseries(bins, a).max(), 'r')
>>> plt.show()
```

`pygeomod.geomodeller_xml_obj.`**`multinomial`**`(`*n*, *pvals*, *size=None*`)`

Draw samples from a multinomial distribution.

The multinomial distribution is a multivariate generalisation of the binomial distribution. Take an experiment with one of `p` possible outcomes. An example of such an experiment is throwing a dice, where the outcome can be 1 through 6. Each sample drawn from the distribution represents *n* such experiments. Its values, `X_i = [X_0, X_1, ..., X_p]`, represent the number of times the outcome was `i`.

n [int] Number of experiments.

pvals [sequence of floats, length p] Probabilities of each of the `p` different outcomes. These should sum to 1 (however, the last element is always assumed to account for the remaining probability, as long as `sum(pvals[:-1]) <= 1)`.

size [tuple of ints] Given a *size* of `(M, N, K)`, then `M*N*K` samples are drawn, and the output shape becomes `(M, N, K, p)`, since each sample has shape `(p,)`.

Throw a dice 20 times:

```
>>> np.random.multinomial(20, [1/6.]*6, size=1)
array([[4, 1, 7, 5, 2, 1]])
```

It landed 4 times on 1, once on 2, etc.

Now, throw the dice 20 times, and 20 times again:

```
>>> np.random.multinomial(20, [1/6.]*6, size=2)
array([[3, 4, 3, 3, 4, 3],
       [2, 4, 3, 4, 0, 7]])
```

For the first run, we threw 3 times 1, 4 times 2, etc. For the second, we threw 2 times 1, 4 times 2, etc.

A loaded dice is more likely to land on number 6:

```
>>> np.random.multinomial(100, [1/7.]*5)
array([13, 16, 13, 16, 42])
```

`pygeomod.geomodeller_xml_obj.`**`multivariate_normal`**(*mean*, *cov*[, *size*])

Draw random samples from a multivariate normal distribution.

The multivariate normal, multinormal or Gaussian distribution is a generalization of the one-dimensional normal distribution to higher dimensions. Such a distribution is specified by its mean and covariance matrix. These parameters are analogous to the mean (average or "center") and variance (standard deviation, or "width," squared) of the one-dimensional normal distribution.

mean [1-D array_like, of length N] Mean of the N-dimensional distribution.

cov [2-D array_like, of shape (N, N)] Covariance matrix of the distribution. Must be symmetric and positive semi-definite for "physically meaningful" results.

size [int or tuple of ints, optional] Given a shape of, for example, `(m,n,k)`, `m*n*k` samples are generated, and packed in an *m*-by-*n*-by-*k* arrangement. Because each sample is *N*-dimensional, the output shape is `(m,n,k,N)`. If no shape is specified, a single (*N*-D) sample is returned.

out [ndarray] The drawn samples, of shape *size*, if that was provided. If not, the shape is `(N,)`.

In other words, each entry `out[i,j,...,:]` is an N-dimensional value drawn from the distribution.

The mean is a coordinate in N-dimensional space, which represents the location where samples are most likely to be generated. This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.

Covariance indicates the level to which two variables vary together. From the multivariate normal distribution, we draw N-dimensional samples, $X = [x_1, x_2, ...x_N]$. The covariance matrix element C_{ij} is the covariance of x_i and x_j. The element C_{ii} is the variance of x_i (i.e. its "spread").

Instead of specifying the full covariance matrix, popular approximations include:

- Spherical covariance (*cov* is a multiple of the identity matrix)
- Diagonal covariance (*cov* has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```
>>> mean = [0,0]
>>> cov = [[1,0],[0,100]] # diagonal covariance, points lie on x or y-axis
>>> import matplotlib.pyplot as plt
>>> x,y = np.random.multivariate_normal(mean,cov,5000).T
>>> plt.plot(x,y,'x'); plt.axis('equal'); plt.show()
```

Note that the covariance matrix must be non-negative definite.

Papoulis, A., *Probability, Random Variables, and Stochastic Processes*, 3rd ed., New York: McGraw-Hill, 1991.

Duda, R. O., Hart, P. E., and Stork, D. G., *Pattern Classification*, 2nd ed., New York: Wiley, 2001.

```
>>> mean = (1,2)
>>> cov = [[1,0],[1,0]]
>>> x = np.random.multivariate_normal(mean,cov,(3,3))
>>> x.shape
(3, 3, 2)
```

The following is probably true, given that 0.6 is roughly twice the standard deviation:

```
>>> print list( (x[0,0,:] - mean) < 0.6 )
[True, True]
```

pygeomod.geomodeller_xml_obj.**negative_binomial**(*n*, *p*, *size=None*)

Draw samples from a negative_binomial distribution.

Samples are drawn from a negative_Binomial distribution with specified parameters, *n* trials and *p* probability of success where *n* is an integer > 0 and *p* is in the interval [0, 1].

n [int] Parameter, > 0.

p [float] Parameter, >= 0 and <=1.

size [int or tuple of ints] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

samples [int or ndarray of ints] Drawn samples.

The probability density for the Negative Binomial distribution is

$$P(N;n,p) = \binom{N+n-1}{n-1}p^{n}(1-p)^{N},$$

where $n-1$ is the number of successes, p is the probability of success, and $N+n-1$ is the number of trials.

The negative binomial distribution gives the probability of n-1 successes and N failures in N+n-1 trials, and success on the (N+n)th trial.

If one throws a die repeatedly until the third time a "1" appears, then the probability distribution of the number of non-"1"s that appear before the third "1" is a negative binomial distribution.

Draw samples from the distribution:

A real world example. A company drills wild-cat oil exploration wells, each with an estimated probability of success of 0.1. What is the probability of having one success for each successive well, that is what is the probability of a single success after drilling 5 wells, after 6 wells, etc.?

```
>>> s = np.random.negative_binomial(1, 0.1, 100000)
>>> for i in range(1, 11):
...     probability = sum(s<i) / 100000.
...     print i, "wells drilled, probability of one success =", probability
```

pygeomod.geomodeller_xml_obj.**noncentral_chisquare**(*df*, *nonc*, *size=None*)

Draw samples from a noncentral chi-square distribution.

The noncentral χ^2 distribution is a generalisation of the χ^2 distribution.

df [int] Degrees of freedom, should be >= 1.

nonc [float] Non-centrality, should be > 0.

size [int or tuple of ints] Shape of the output.

The probability density function for the noncentral Chi-square distribution is

$$P(x; df, nonc) = \sum_{i=0}^{\infty} \frac{e^{-nonc/2}(nonc/2)^i}{i!} P_{Y_{df+2i}}(x),$$

where Y_q is the Chi-square with q degrees of freedom.

In Delhi (2007), it is noted that the noncentral chi-square is useful in bombing and coverage problems, the probability of killing the point target given by the noncentral chi-squared distribution.

Draw values from the distribution and plot the histogram

```
>>> import matplotlib.pyplot as plt
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
...                   bins=200, normed=True)
>>> plt.show()
```

Draw values from a noncentral chisquare with very small noncentrality, and compare to a chisquare.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, .0000001, 100000),
...                   bins=np.arange(0., 25, .1), normed=True)
>>> values2 = plt.hist(np.random.chisquare(3, 100000),
...                    bins=np.arange(0., 25, .1), normed=True)
>>> plt.plot(values[1][0:-1], values[0]-values2[0], 'ob')
>>> plt.show()
```

Demonstrate how large values of non-centrality lead to a more symmetric distribution.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
...                   bins=200, normed=True)
>>> plt.show()
```

`pygeomod.geomodeller_xml_obj.`**`noncentral_f`**`(`*dfnum*, *dfden*, *nonc*, *size=None*`)`

Draw samples from the noncentral F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters > 1. *nonc* is the non-centrality parameter.

dfnum [int] Parameter, should be > 1.

dfden [int] Parameter, should be > 1.

nonc [float] Parameter, should be >= 0.

size [int or tuple of ints] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

samples [scalar or ndarray] Drawn samples.

When calculating the power of an experiment (power = probability of rejecting the null hypothesis when a specific alternative is true) the non-central F statistic becomes important. When the null hypothesis is true, the F statistic follows a central F distribution. When the null hypothesis is not true, then it follows a non-central F statistic.

Weisstein, Eric W. "Noncentral F-Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/NoncentralF-Distribution.html

Wikipedia, "Noncentral F distribution", http://en.wikipedia.org/wiki/Noncentral_F-distribution

In a study, testing for a specific alternative to the null hypothesis requires use of the Noncentral F distribution. We need to calculate the area in the tail of the distribution that exceeds the value of the F distribution for the null hypothesis. We'll plot the two probability distributions for comparison.

```
>>> dfnum = 3 # between group deg of freedom
>>> dfden = 20 # within groups degrees of freedom
>>> nonc = 3.0
>>> nc_vals = np.random.noncentral_f(dfnum, dfden, nonc, 1000000)
>>> NF = np.histogram(nc_vals, bins=50, normed=True)
>>> c_vals = np.random.f(dfnum, dfden, 1000000)
>>> F = np.histogram(c_vals, bins=50, normed=True)
>>> plt.plot(F[1][1:], F[0])
>>> plt.plot(NF[1][1:], NF[0])
>>> plt.show()
```

pygeomod.geomodeller_xml_obj.**normal**(*loc=0.0*, *scale=1.0*, *size=None*)

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently [2]_, is often called the bell curve because of its characteristic shape (see the example below).

The normal distributions occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution [2]_.

loc [float] Mean ("centre") of the distribution.

scale [float] Standard deviation (spread or "width") of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

scipy.stats.distributions.norm [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gaussian distribution is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ is the mean and σ the standard deviation. The square of the standard deviation, σ^2, is called the variance.

The function has its peak at the mean, and its "spread" increases with the standard deviation (the function reaches 0.607 times its maximum at $x+\sigma$ and $x-\sigma$ [2]_). This implies that *numpy.random.normal* is more likely to return samples lying close to the mean, rather than those far away.

Draw samples from the distribution:

```
>>> mu, sigma = 0, 0.1 # mean and standard deviation
>>> s = np.random.normal(mu, sigma, 1000)
```

Verify the mean and the variance:

```
>>> abs(mu - np.mean(s)) < 0.01
True
```

```
>>> abs(sigma - np.std(s, ddof=1)) < 0.01
True
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, normed=True)
>>> plt.plot(bins, 1/(sigma * np.sqrt(2 * np.pi)) *
...                np.exp( - (bins - mu)**2 / (2 * sigma**2) ),
...          linewidth=2, color='r')
>>> plt.show()
```

pygeomod.geomodeller_xml_obj.**pareto**(*a*, *size=None*)

Draw samples from a Pareto II or Lomax distribution with specified shape.

The Lomax or Pareto II distribution is a shifted Pareto distribution. The classical Pareto distribution can be obtained from the Lomax distribution by adding the location parameter m, see below. The smallest value of the Lomax distribution is zero while for the classical Pareto distribution it is m, where the standard Pareto distribution has location m=1. Lomax can also be considered as a simplified version of the Generalized Pareto distribution (available in SciPy), with the scale set to one and the location set to zero.

The Pareto distribution must be greater than zero, and is unbounded above. It is also known as the "80-20 rule". In this distribution, 80 percent of the weights are in the lowest 20 percent of the range, while the other 20 percent fill the remaining 80 percent of the range.

shape [float, > 0.] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

scipy.stats.distributions.lomax.pdf [probability density function,] distribution or cumulative density function, etc.

scipy.stats.distributions.genpareto.pdf [probability density function,] distribution or cumulative density function, etc.

The probability density for the Pareto distribution is

$$p(x) = \frac{am^a}{x^{a+1}}$$

where a is the shape and m the location

The Pareto distribution, named after the Italian economist Vilfredo Pareto, is a power law probability distribution useful in many real world problems. Outside the field of economics it is generally referred to as the Bradford distribution. Pareto developed the distribution to describe the distribution of wealth in an economy. It has also found use in insurance, web page access statistics, oil field sizes, and many other problems, including the download frequency for projects in Sourceforge [1]. It is one of the so-called "fat-tailed" distributions.

Draw samples from the distribution:

```
>>> a, m = 3., 1. # shape and mode
>>> s = np.random.pareto(a, 1000) + m
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='center')
>>> fit = a*m**a/bins**(a+1)
>>> plt.plot(bins, max(count)*fit/max(fit),linewidth=2, color='r')
>>> plt.show()
```

pygeomod.geomodeller_xml_obj.**permutation**(*x*)

Randomly permute a sequence, or return a permuted range.

If *x* is a multi-dimensional array, it is only shuffled along its first index.

x [int or array_like] If *x* is an integer, randomly permute `np.arange(x)`. If *x* is an array, make a copy and shuffle the elements randomly.

out [ndarray] Permuted sequence or array range.

```
>>> np.random.permutation(10)
array([1, 7, 4, 3, 0, 9, 2, 5, 8, 6])

>>> np.random.permutation([1, 4, 9, 12, 15])
array([15,  1,  9,  4, 12])

>>> arr = np.arange(9).reshape((3, 3))
>>> np.random.permutation(arr)
array([[6, 7, 8],
       [0, 1, 2],
       [3, 4, 5]])
```

pygeomod.geomodeller_xml_obj.**poisson**(*lam=1.0*, *size=None*)

Draw samples from a Poisson distribution.

The Poisson distribution is the limit of the Binomial distribution for large N.

lam [float] Expectation of interval, should be >= 0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

The Poisson distribution

$$f(k; \lambda)=\frac{\lambda^k e^{-\lambda}}{k!}$$

For events with an expected separation λ the Poisson distribution $f(k; \lambda)$ describes the probability of k events occurring within the observed interval λ.

Because the output is limited to the range of the C long type, a ValueError is raised when *lam* is within 10 sigma of the maximum representable value.

Draw samples from the distribution:

```
>>> import numpy as np
>>> s = np.random.poisson(5, 10000)
```

Display histogram of the sample:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 14, normed=True)
>>> plt.show()
```

pygeomod.geomodeller_xml_obj.**power**(*a*, *size=None*)

Draws samples in [0, 1] from a power distribution with positive exponent a - 1.

Also known as the power function distribution.

a [float] parameter, > 0

size [tuple of ints]

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

samples [{ndarray, scalar}] The returned samples lie in [0, 1].

ValueError If a<1.

The probability density function is

$$P(x; a) = ax^{a-1}, 0 \le x \le 1, a > 0.$$

The power function distribution is just the inverse of the Pareto distribution. It may also be seen as a special case of the Beta distribution.

It is used, for example, in modeling the over-reporting of insurance claims.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> samples = 1000
>>> s = np.random.power(a, samples)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, bins=30)
>>> x = np.linspace(0, 1, 100)
>>> y = a*x**(a-1.)
>>> normed_y = samples*np.diff(bins)[0]*y
>>> plt.plot(x, normed_y)
>>> plt.show()
```

Compare the power function distribution to the inverse of the Pareto.

```
>>> from scipy import stats
>>> rvs = np.random.power(5, 1000000)
>>> rvsp = np.random.pareto(5, 1000000)
>>> xx = np.linspace(0,1,100)
>>> powpdf = stats.powerlaw.pdf(xx,5)

>>> plt.figure()
>>> plt.hist(rvs, bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('np.random.power(5)')

>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of 1 + np.random.pareto(5)')

>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of stats.pareto(5)')
```

`pygeomod.geomodeller_xml_obj.`**`rand`**(*d0, d1, ..., dn*)

Random values in a given shape.

Create an array of the given shape and propagate it with random samples from a uniform distribution over `[0, 1)`.

d0, d1, ..., dn [int, optional] The dimensions of the returned array, should all be positive. If no argument is given a single Python float is returned.

out [ndarray, shape `(d0, d1, ..., dn)`] Random values.

random

This is a convenience function. If you want an interface that takes a shape-tuple as the first argument, refer to np.random.random_sample .

```
>>> np.random.rand(3,2)
array([[ 0.14022471,  0.96360618],  #random
       [ 0.37601032,  0.25528411],  #random
       [ 0.49313049,  0.94909878]]) #random
```

`pygeomod.geomodeller_xml_obj.`**`randint`**(*low*, *high=None*, *size=None*)

Return random integers from *low* (inclusive) to *high* (exclusive).

Return random integers from the "discrete uniform" distribution in the "half-open" interval [*low*, *high*). If *high* is None (the default), then results are from [0, *low*).

low [int] Lowest (signed) integer to be drawn from the distribution (unless `high=None`, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if `high=None`).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random.random_integers [similar to *randint*, only for the closed] interval [*low*, *high*], and 1 is the lowest value if *high* is omitted. In particular, this other one is the one to use to generate uniformly distributed discrete non-integers.

```
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0])
>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

```
>>> np.random.randint(5, size=(2, 4))
array([[4, 0, 2, 1],
       [3, 2, 2, 0]])
```

`pygeomod.geomodeller_xml_obj.`**`randn`**(*d0*, *d1*, ..., *dn*)

Return a sample (or samples) from the "standard normal" distribution.

If positive, int_like or int-convertible arguments are provided, *randn* generates an array of shape `(d0, d1, ..., dn)`, filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1 (if any of the d_i are floats, they are first converted to integers by truncation). A single float randomly sampled from the distribution is returned if no argument is provided.

This is a convenience function. If you want an interface that takes a tuple as the first argument, use *numpy.random.standard_normal* instead.

d0, d1, ..., dn [int, optional] The dimensions of the returned array, should be all positive. If no argument is given a single Python float is returned.

Z [ndarray or float] A `(d0, d1, ..., dn)`-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

random.standard_normal : Similar, but takes a tuple as its argument.

For random samples from $N(\mu, \sigma^2)$, use:

`sigma * np.random.randn(...) + mu`

```
>>> np.random.randn()
2.1923875335537315 #random
```

Two-by-four array of samples from N(3, 6.25):

```
>>> 2.5 * np.random.randn(2, 4) + 3
array([[-4.49401501,  4.00950034, -1.81814867,  7.29718677],  #random
       [ 0.39924804,  4.68456316,  4.99394529,  4.84057254]]) #random
```

`pygeomod.geomodeller_xml_obj.`**`random`**`()`

random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample $Unif[a, b), b > a$ multiply the output of *random_sample* by *(b-a)* and add *a*:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless `size=None`, in which case a single float is returned).

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```
>>> 5 * np.random.random_sample((3, 2)) - 5
array([[-3.99149989, -0.52338984],
       [-2.99091858, -0.79479508],
       [-1.23204345, -1.75224494]])
```

`pygeomod.geomodeller_xml_obj.`**`random_integers`**`(`*low*, *high=None*, *size=None*`)`

Return random integers between *low* and *high*, inclusive.

Return random integers from the "discrete uniform" distribution in the closed interval [*low*, *high*]. If *high* is None (the default), then results are from [1, *low*].

low [int] Lowest (signed) integer to be drawn from the distribution (unless `high=None`, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, the largest (signed) integer to be drawn from the distribution (see above for behavior if `high=None`).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random.randint [Similar to *random_integers*, only for the half-open] interval [*low*, *high*), and 0 is the lowest value if *high* is omitted.

To sample from N evenly spaced floating-point numbers between a and b, use:

```
a + (b - a) * (np.random.random_integers(N) - 1) / (N - 1.)
```

```
>>> np.random.random_integers(5)
4
>>> type(np.random.random_integers(5))
<type 'int'>
>>> np.random.random_integers(5, size=(3.,2.))
array([[5, 4],
       [3, 3],
       [4, 5]])
```

Choose five random numbers from the set of five evenly-spaced numbers between 0 and 2.5, inclusive (*i.e.*, from the set $0, 5/8, 10/8, 15/8, 20/8$):

```
>>> 2.5 * (np.random.random_integers(5, size=(5,)) - 1) / 4.
array([ 0.625,  1.25 ,  0.625,  0.625,  2.5  ])
```

Roll two six sided dice 1000 times and sum the results:

```
>>> d1 = np.random.random_integers(1, 6, 1000)
>>> d2 = np.random.random_integers(1, 6, 1000)
>>> dsums = d1 + d2
```

Display results as a histogram:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(dsums, 11, normed=True)
>>> plt.show()
```

`pygeomod.geomodeller_xml_obj.`**`random_sample`**`(`*size=None*`)`

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample $Unif[a, b), b > a$ multiply the output of *random_sample* by *(b-a)* and add *a*:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless `size=None`, in which case a single float is returned).

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```
>>> 5 * np.random.random_sample((3, 2)) - 5
array([[-3.99149989, -0.52338984],
       [-2.99091858, -0.79479508],
       [-1.23204345, -1.75224494]])
```

pygeomod.geomodeller_xml_obj.**ranf**()

random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample $Unif[a, b), b > a$ multiply the output of *random_sample* by *(b-a)* and add *a*:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless `size=None`, in which case a single float is returned).

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```
>>> 5 * np.random.random_sample((3, 2)) - 5
array([[-3.99149989, -0.52338984],
       [-2.99091858, -0.79479508],
       [-1.23204345, -1.75224494]])
```

pygeomod.geomodeller_xml_obj.**rayleigh**(*scale=1.0*, *size=None*)

Draw samples from a Rayleigh distribution.

The χ and Weibull distributions are generalizations of the Rayleigh.

scale [scalar] Scale, also equals the mode. Should be >= 0.

size [int or tuple of ints, optional] Shape of the output. Default is None, in which case a single value is returned.

The probability density function for the Rayleigh distribution is

$$P(x; scale) = \frac{x}{scale^2} e^{\frac{-x^2}{2 \cdot scale^2}}$$

The Rayleigh distribution arises if the wind speed and wind direction are both gaussian variables, then the vector wind velocity forms a Rayleigh distribution. The Rayleigh distribution is used to model the expected output from wind turbines.

Draw values from the distribution and plot the histogram

```
>>> values = hist(np.random.rayleigh(3, 100000), bins=200, normed=True)
```

Wave heights tend to follow a Rayleigh distribution. If the mean wave height is 1 meter, what fraction of waves are likely to be larger than 3 meters?

```
>>> meanvalue = 1
>>> modevalue = np.sqrt(2 / np.pi) * meanvalue
>>> s = np.random.rayleigh(modevalue, 1000000)
```

The percentage of waves larger than 3 meters is:

```
>>> 100.*sum(s>3)/1000000.
0.087300000000000003
```

`pygeomod.geomodeller_xml_obj.`**`sample`**`()`

random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample $Unif[a, b), b > a$ multiply the output of *random_sample* by *(b-a)* and add *a*:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless `size=None`, in which case a single float is returned).

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```
>>> 5 * np.random.random_sample((3, 2)) - 5
array([[-3.99149989, -0.52338984],
       [-2.99091858, -0.79479508],
       [-1.23204345, -1.75224494]])
```

`pygeomod.geomodeller_xml_obj.`**`seed`**`(`*seed=None*`)`

Seed the generator.

This method is called when *RandomState* is initialized. It can be called again to re-seed the generator. For details, see *RandomState*.

seed [int or array_like, optional] Seed for *RandomState*.

RandomState

`pygeomod.geomodeller_xml_obj.`**`set_state`**`(`*state*`)`

Set the internal state of the generator from a tuple.

For use if one has reason to manually (re-)set the internal state of the "Mersenne Twister"[1]_ pseudo-random number generating algorithm.

state [tuple(str, ndarray of 624 uints, int, int, float)] The *state* tuple has the following items:

1. the string 'MT19937', specifying the Mersenne Twister algorithm.
2. a 1-D array of 624 unsigned integers `keys`.
3. an integer `pos`.

4. an integer `has_gauss`.
5. a float `cached_gaussian`.

out [None] Returns 'None' on success.

get_state

set_state and *get_state* are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

For backwards compatibility, the form (str, array of 624 uints, int) is also accepted although it is missing some information about the cached Gaussian value: `state = ('MT19937', keys, pos)`.

`pygeomod.geomodeller_xml_obj.`**`shuffle`**`(`*x*`)`

Modify a sequence in-place by shuffling its contents.

x [array_like] The array or list to be shuffled.

None

```
>>> arr = np.arange(10)
>>> np.random.shuffle(arr)
>>> arr
[1 7 5 2 9 4 3 6 0 8]
```

This function only shuffles the array along the first index of a multi-dimensional array:

```
>>> arr = np.arange(9).reshape((3, 3))
>>> np.random.shuffle(arr)
>>> arr
array([[3, 4, 5],
       [6, 7, 8],
       [0, 1, 2]])
```

`pygeomod.geomodeller_xml_obj.`**`standard_cauchy`**`(`*size=None*`)`

Standard Cauchy distribution with mode = 0.

Also known as the Lorentz distribution.

size [int or tuple of ints] Shape of the output.

samples [ndarray or scalar] The drawn samples.

The probability density function for the full Cauchy distribution is

$$P(x; x_0, \gamma) = \frac{1}{\pi \gamma \bigl[1+ (\frac{x-x_0}{\gamma})^2 \bigr] }$$

and the Standard Cauchy distribution just sets $x_0 = 0$ and $\gamma = 1$

The Cauchy distribution arises in the solution to the driven harmonic oscillator problem, and also describes spectral line broadening. It also describes the distribution of values at which a line tilted at a random angle will cut the x axis.

When studying hypothesis tests that assume normality, seeing how the tests perform on data from a Cauchy distribution is a good indicator of their sensitivity to a heavy-tailed distribution, since the Cauchy looks very much like a Gaussian distribution, but with heavier tails.

Draw samples and plot the distribution:

```
>>> s = np.random.standard_cauchy(1000000)
>>> s = s[(s>-25) & (s<25)]  # truncate distribution so it plots well
>>> plt.hist(s, bins=100)
>>> plt.show()
```

pygeomod.geomodeller_xml_obj.**standard_exponential**(*size=None*)

Draw samples from the standard exponential distribution.

standard_exponential is identical to the exponential distribution with a scale parameter of 1.

size [int or tuple of ints] Shape of the output.

out [float or ndarray] Drawn samples.

Output a 3x8000 array:

```
>>> n = np.random.standard_exponential((3, 8000))
```

pygeomod.geomodeller_xml_obj.**standard_gamma**(*shape*, *size=None*)

Draw samples from a Standard Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, shape (sometimes designated "k") and scale=1.

shape [float] Parameter, should be > 0.

size [int or tuple of ints] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

samples [ndarray or scalar] The drawn samples.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1}\frac{e^{-x/\theta}}{\theta^k\Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 1. # mean and width
>>> s = np.random.standard_gamma(shape, 1000000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> y = bins**(shape-1) * ((np.exp(-bins/scale))/ \
...                       (sps.gamma(shape) * scale**shape))
>>> plt.plot(bins, y, linewidth=2, color='r')
>>> plt.show()
```

pygeomod.geomodeller_xml_obj.**standard_normal**(*size=None*)

Returns samples from a Standard Normal distribution (mean=0, stdev=1).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

out [float or ndarray] Drawn samples.

```
>>> s = np.random.standard_normal(8000)
>>> s
array([ 0.6888893 ,  0.78096262, -0.89086505, ...,  0.49876311, #random
       -0.38672696, -0.4685006 ])                                #random
>>> s.shape
(8000,)
>>> s = np.random.standard_normal(size=(3, 4, 2))
>>> s.shape
(3, 4, 2)
```

pygeomod.geomodeller_xml_obj.**standard_t**(*df*, *size=None*)

Standard Student's t distribution with df degrees of freedom.

A special case of the hyperbolic distribution. As *df* gets large, the result resembles that of the standard normal distribution (*standard_normal*).

df [int] Degrees of freedom, should be > 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn samples.

The probability density function for the t distribution is

$$P(x, df) = \frac{\Gamma(\frac{df+1}{2})}{\sqrt{\pi df}\Gamma(\frac{df}{2})}\Bigl(1+\frac{x^2}{df} \Bigr)^{-(df+1)/2}$$

The t test is based on an assumption that the data come from a Normal distribution. The t test provides a way to test whether the sample mean (that is the mean calculated from the data) is a good estimate of the true mean.

The derivation of the t-distribution was forst published in 1908 by William Gisset while working for the Guinness Brewery in Dublin. Due to proprietary issues, he had to publish under a pseudonym, and so he used the name Student.

From Dalgaard page 83 [1]_, suppose the daily energy intake for 11 women in Kj is:

```
>>> intake = np.array([5260., 5470, 5640, 6180, 6390, 6515, 6805, 7515, \
...                    7515, 8230, 8770])
```

Does their energy intake deviate systematically from the recommended value of 7725 kJ?

We have 10 degrees of freedom, so is the sample mean within 95% of the recommended value?

```
>>> s = np.random.standard_t(10, size=100000)
>>> np.mean(intake)
6753.636363636364
>>> intake.std(ddof=1)
1142.1232221373727
```

Calculate the t statistic, setting the ddof parameter to the unbiased value so the divisor in the standard deviation will be degrees of freedom, N-1.

```
>>> t = (np.mean(intake)-7725)/(intake.std(ddof=1)/np.sqrt(len(intake)))
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(s, bins=100, normed=True)
```

For a one-sided t-test, how far out in the distribution does the t statistic appear?

```
>>> >>> np.sum(s<t) / float(len(s))
0.0090699999999999999  #random
```

So the p-value is about 0.009, which says the null hypothesis has a probability of about 99% of being true.

`pygeomod.geomodeller_xml_obj.`**`triangular`**(*left*, *mode*, *right*, *size=None*)

Draw samples from the triangular distribution.

The triangular distribution is a continuous probability distribution with lower limit left, peak at mode, and upper limit right. Unlike the other distributions, these parameters directly define the shape of the pdf.

left [scalar] Lower limit.

mode [scalar] The value where the peak of the distribution occurs. The value should fulfill the condition `left <= mode <= right`.

right [scalar] Upper limit, should be larger than *left*.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] The returned samples all lie in the interval [left, right].

The probability density function for the Triangular distribution is

$$P(x;l,m,r) = \begin{cases} \frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \leq x \leq m, \\ \frac{2(m-x)}{(r-l)(r-m)} & \text{for } m \leq x \leq r, \\ 0 & \text{otherwise.} \end{cases}$$

The triangular distribution is often used in ill-defined problems where the underlying distribution is not known, but some knowledge of the limits and mode exists. Often it is used in simulations.

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.triangular(-3, 0, 8, 100000), bins=200,
...              normed=True)
>>> plt.show()
```

`pygeomod.geomodeller_xml_obj.`**`uniform`**(*low=0.0*, *high=1.0*, *size=1*)

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval `[low, high)` (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by *uniform*.

low [float, optional] Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

high [float] Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.

size [int or tuple of ints, optional] Shape of output. If the given size is, for example, (m,n,k), m*n*k samples are generated. If no shape is specified, a single sample is returned.

out [ndarray] Drawn samples, with shape *size*.

randint : Discrete uniform distribution, yielding integers. random_integers : Discrete uniform distribution over the closed

interval `[low, high]`.

random_sample : Floats uniformly distributed over `[0, 1)`. random : Alias for *random_sample*. rand : Convenience function that accepts dimensions as input, e.g.,

> `rand(2,2)` would generate a 2-by-2 array of floats, uniformly distributed over `[0, 1)`.

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b - a}$$

anywhere within the interval `[a, b)`, and zero elsewhere.

Draw samples from the distribution:

```
>>> s = np.random.uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, normed=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```

`pygeomod.geomodeller_xml_obj.`**`vonmises`**`(`*mu*, *kappa*, *size=None*`)`

Draw samples from a von Mises distribution.

Samples are drawn from a von Mises distribution with specified mode (mu) and dispersion (kappa), on the interval [-pi, pi].

The von Mises distribution (also known as the circular normal distribution) is a continuous probability distribution on the unit circle. It may be thought of as the circular analogue of the normal distribution.

mu [float] Mode ("center") of the distribution.

kappa [float] Dispersion of the distribution, has to be >=0.

size [int or tuple of int] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

samples [scalar or ndarray] The returned samples, which are in the interval [-pi, pi].

scipy.stats.distributions.vonmises [probability density function,] distribution, or cumulative density function, etc.

The probability density for the von Mises distribution is

$$p(x) = \frac{e^{\kappa cos(x-\mu)}}{2\pi I_0(\kappa)},$$

where μ is the mode and κ the dispersion, and $I_0(\kappa)$ is the modified Bessel function of order 0.

The von Mises is named for Richard Edler von Mises, who was born in Austria-Hungary, in what is now the Ukraine. He fled to the United States in 1939 and became a professor at Harvard. He worked in probability theory, aerodynamics, fluid mechanics, and philosophy of science.

Abramowitz, M. and Stegun, I. A. (ed.), *Handbook of Mathematical Functions*, New York: Dover, 1965.

von Mises, R., *Mathematical Theory of Probability and Statistics*, New York: Academic Press, 1964.

Draw samples from the distribution:

```
>>> mu, kappa = 0.0, 4.0 # mean and dispersion
>>> s = np.random.vonmises(mu, kappa, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> x = np.arange(-np.pi, np.pi, 2*np.pi/50.)
>>> y = -np.exp(kappa*np.cos(x-mu))/(2*np.pi*sps.jn(0,kappa))
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

`pygeomod.geomodeller_xml_obj.`**`wald`**`(`*mean*, *scale*, *size=None*`)`

Draw samples from a Wald, or Inverse Gaussian, distribution.

As the scale approaches infinity, the distribution becomes more like a Gaussian.

Some references claim that the Wald is an Inverse Gaussian with mean=1, but this is by no means universal.

The Inverse Gaussian distribution was first studied in relationship to Brownian motion. In 1956 M.C.K. Tweedie used the name Inverse Gaussian because there is an inverse relationship between the time to cover a unit distance and distance covered in unit time.

mean [scalar] Distribution mean, should be > 0.

scale [scalar] Scale parameter, should be >= 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn sample, all greater than zero.

The probability density function for the Wald distribution is

$$P(x; mean, scale) = \sqrt{\frac{scale}{2\pi x^3}} e^{\frac{-scale(x-mean)^2}{2\cdot mean^2 x}}$$

As noted above the Inverse Gaussian distribution first arise from attempts to model Brownian Motion. It is also a competitor to the Weibull for use in reliability modeling and modeling stock returns and interest rate processes.

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.wald(3, 2, 100000), bins=200, normed=True)
>>> plt.show()
```

`pygeomod.geomodeller_xml_obj.`**`weibull`**`(`*a*, *size=None*`)`

Weibull distribution.

Draw samples from a 1-parameter Weibull distribution with the given shape parameter *a*.

$$X = (-ln(U))^{1/a}$$

Here, U is drawn from the uniform distribution over (0,1].

The more common 2-parameter Weibull, including a scale parameter λ is just $X = \lambda(-ln(U))^{1/a}$.

a [float] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

scipy.stats.distributions.weibull_max scipy.stats.distributions.weibull_min scipy.stats.distributions.genextreme gumbel

The Weibull (or Type III asymptotic extreme value distribution for smallest values, SEV Type III, or Rosin-Rammler distribution) is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. This class includes the Gumbel and Frechet distributions.

The probability density for the Weibull distribution is

$$p(x) = \frac{a}{\lambda}(\frac{x}{\lambda})^{a-1}e^{-(x/\lambda)^a},$$

where a is the shape and λ the scale.

The function has its peak (the mode) at $\lambda(\frac{a-1}{a})^{1/a}$.

When `a = 1`, the Weibull distribution reduces to the exponential distribution.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> s = np.random.weibull(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> x = np.arange(1,100.)/50.
>>> def weib(x,n,a):
...     return (a / n) * (x / n)**(a - 1) * np.exp(-(x / n)**a)

>>> count, bins, ignored = plt.hist(np.random.weibull(5.,1000))
>>> x = np.arange(1,100.)/50.
>>> scale = count.max()/weib(x, 1., 5.).max()
>>> plt.plot(x, weib(x, 1., 5.)*scale)
>>> plt.show()
```

`pygeomod.geomodeller_xml_obj.`**`zipf`**`(`*a*, *size=None*`)`

Draw samples from a Zipf distribution.

Samples are drawn from a Zipf distribution with specified parameter *a* > 1.

The Zipf distribution (also known as the zeta distribution) is a continuous probability distribution that satisfies Zipf's law: the frequency of an item is inversely proportional to its rank in a frequency table.

a [float > 1] Distribution parameter.

size [int or tuple of int, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn; a single integer is equivalent in its result to providing a mono-tuple, i.e., a 1-D array of length *size* is returned. The default is None, in which case a single scalar is returned.

samples [scalar or ndarray] The returned samples are greater than or equal to one.

scipy.stats.distributions.zipf [probability density function,] distribution, or cumulative density function, etc.

The probability density for the Zipf distribution is

$$p(x) = \frac{x^{-a}}{\zeta(a)},$$

where ζ is the Riemann Zeta function.

It is named for the American linguist George Kingsley Zipf, who noted that the frequency of any word in a sample of a language is inversely proportional to its rank in the frequency table.

Zipf, G. K., *Selected Studies of the Principle of Relative Frequency in Language*, Cambridge, MA: Harvard Univ. Press, 1932.

Draw samples from the distribution:

```
>>> a = 2. # parameter
>>> s = np.random.zipf(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
Truncate s values at 50 so plot is interesting
>>> count, bins, ignored = plt.hist(s[s<50], 50, normed=True)
>>> x = np.arange(1., 50.)
>>> y = x**(-a)/sps.zetac(a)
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

5.1.4 pygeomod.struct_data module

Analysis and modification of structural data exported from GeoModeller

All structural data from an entire GeoModeller project can be exported into ASCII files using the function in the GUI:

Export -> 3D Structural Data

This method generates files for defined geological parameters: "Points" (i.e. formation contact points) and "Foliations" (i.e. orientations/ potential field gradients).

Exported parameters include all those defined in sections as well as 3D data points.

This package contains methods to check, visualise, and extract/modify parts of these exported data sets, for example to import them into a different Geomodeller project.

class `pygeomod.struct_data.`**`Struct3DFoliations`**`(`***kwds*`)`

Bases: `pygeomod.struct_data.Struct3DPoints`

Class container for foliations (i.e. orientations) exported from GeoModeller

Mainly based on Struct3DPoints as must required functionality for location of elements - some functions overwritten, e.g. save and parse to read orientation data, as well!

However, further methods might be added or adapted in the future, for example: - downsampling according to (eigen)vector methods, e.g. the work from the Monash guys, etc. - ploting of orientations in 2-D and 3-D

`parse``()`

Parse filename and load data into numpy record

The point information is stored in a purpose defined numpy record self.points

`save``(`*filename*`)`

Save points set to file

Arguments:

- *filename* = string : name of new file

class `pygeomod.struct_data.`**`Struct3DPoints`**`(`***kwds*`)`

Class container for 3D structural points data sets

`combine_with`(*pts_set*)
Combine this point set with another point set

Arguments:

- *pts_set* = Struct3DPoints : points set to combine

`create_formation_subset`(*formation_names*)
Create a subset (as another Struct3DPoints object) with specified formations only

Arguments:

- *formation_names* : list of formation names

Returns: Struct3DPoints object with subset of points

`extract_range`(***kwds*)
Extract subset for defined ranges

Pass ranges as keywords: from_x, to_x, from_y, to_y, from_z, to_z All not defined ranges are simply kept as before

Returns: pts_subset : Struct3DPoints data subset

`get_formation_names`()
Get names of all formations that have a point in this data set and store in:

self.formation_names

`get_range`()
Update min, max for all coordinate axes and store in self.xmin, self.xmax, ...

`parse`()
Parse filename and load data into numpy record

The point information is stored in a purpose defined numpy record self.points

`plot_3D`(***kwds*)
Create a plot of points in 3-D

Optional keywords:

- *ax* = matplotlib axis object: if provided, plot is attached to this axis
- *formation_names* = list of formations : plot only points for specific formations

`plot_plane`(*plane=('x', 'y'), **kwds*)
Create 2-D plots for point distribution

Arguments:

- *plane* = tuple of plane axes directions, e.g. ('x','y') (default)

Optional Keywords:

- *ax* = matplotlib axis object: if provided, plot is attached to this axis
- *formation_names* = list of formations : plot only points for specific formations

`remove_formations`(*formation_names*)
Remove points for specified formations from the point set

This function can be useful, for example, to remove one formation, perform a thinning operation, and then add it back in with the *combine_with* function.

Arguments:

- *formation_names* = list of formations to be removed (or a single string to remove only one formation)

rename_formations(*rename_dict*)
Rename formation according to assignments in dictionary

Mapping in dictionary is of the form: old_name_1 : new_name_1, old_name_2 : new_name_2, ...

save(*filename*)
Save points set to file

Arguments:

- *filename* = string : name of new file

thin(*nx*, *ny*, *nz*, ***kwds*)
Thin data for one formations on grid with defined number of cells and store as subset

Arguments:

- *nx*, *ny*, *nz* = int : number of cells in each direction for thinning grid

The thinning is performed on a raster and not 'formation-aware', following this simple procedure:

1.Iterate through grid

(2) If multiple points for formation in this cell: thin (3a) If thin: Select one point in cell at random and keep this one! (3b) else: if one point in raneg, keep it!

Note: Thinning is performed for all formations, so make sure to create a subset for a single formation first!

Returns: pts_subset = Struct3DPoints : subset with thinned data for formation

5.1.5 Module contents

Module initialisation for pygeomod Created on 21/03/2014

@author: Florian

CHAPTER

SIX

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX