pynoddy Documentation

Release

Florian Wellmann

August 16, 2015

Contents:

CHAPTER

ONE

PYNODDY

pynoddy is a python package to write, change, and analyse kinematic geological modelling simulations performed with Noddy (see below for more infomration on Noddy).

1.1 How does it work?

At this stage, pynoddy provides wrapper modules for existing Noddy history (.his) and result files (.g00, etc.). It is

1.2 Installation

To install pynoddy simply run:

```
python setup.py install
```

Note:

- sufficient privileges are required (i.e. run in sudo with MacOSX/ Linux and set permissions on Windows)

Important: the Noddy executable has to be in a directory defined in the PATH variable!!

Important: the topology executable has to be in a directory defined in the PATH variable!!

1.3 Documentation

1.4 Tutorial

A tutorial starting with simple examples for changing the geological history and visualisation of output, as well as the implementation of stochastic simulations and uncertainty visualisation are available as interactive ipython notebooks.

1.5 Dependencies

pynoddy depends on several standard Python packages that should be shipped with any standard distribution (and are easy to install, otherwise):

. numpy . matplotlib . pickle

The uncertainty anaysis, quantification, and visualisation methods based on information theory are implemented in the python package pygeoinfo. This package is available on github and part of the python package index. It is automatically installed with the setup script provided with this package. For more information, please see:

(todo: include package info!)

In addition, to export model results for full 3-D visualisation with VTK, the pyevtk package is used, available on bitbucket:

https://bitbucket.org/pauloh/pyevtk/src/9c19e3a54d1e?at=v0.1.0

The package is automatically downloaded and installed when running python setup.py install.

1.6 License

pynoddy is free software and published under a MIT license (see license file included in the repository). Please attribute the work when you use it, feel free to change and adapt it otherwise!

1.7 What is Noddy?

Noddy itself is a kinematic modelling program written by Mark Jessell [1][2] to simulate the effect of subsequent geological events (folding, unconformities, faulting, etc.) on a primary sedimentary pile. A typical example would be:

1. Create a sedimentary pile with defined thicknesses for multiple formations
2. Add a folding event (for example simple sinoidal folding, but complex methods are possible!)
3. Add an unconformity and, above it, a new stratigraphy
4. Finally, add a sequence of late faults affecting the entire system.

The result could look something like this:

The software runs on Windows only, but the source files (written in C) are available for download to generate a command line version of the modelling step alone:

https://github.com/flohorovicic/pynoddy

It has been tested and compiled on MacOSX, Windows and Linux.

1.8 References

[1] Mark W. Jessell. Noddy, an interactive map creation package. Unpublished MSc Thesis, University of London. 1981. [2] Mark W. Jessell, Rick K. Valenta, Structural geophysics: Integrated structural and geophysical modelling, In: Declan G. De Paor, Editor(s), Computer Methods in the Geosciences, Pergamon, 1996, Volume 15, Pages 303-324, ISSN 1874-561X, ISBN 9780080424309, http://dx.doi.org/10.1016/S1874-561X(96)80027-7.

CHAPTER

TWO

SIMULATION OF A NODDY HISTORY AND VISUALISATION OF OUTPUT

Examples of how the module can be used to run Noddy simulations and visualise the output.

```
# Basic settings
import sys, os
import subprocess

# Now import pynoddy
import pynoddy

# determine path of repository to set paths corretly below

repo_path = os.path.realpath('../..')
```

2.1 Compute the model

The simplest way to perform the Noddy simulation through Python is simply to call the executable. One way that should be fairly platform independent is to use Python's own subprocess module:

```
# Change to sandbox directory to store results
os.chdir(os.path.join(repo_path, 'sandbox'))

# Path to exmaple directory in this repository
example_directory = os.path.join(repo_path,'examples')
# Compute noddy model for history file
history_file = 'simple_two_faults.his'
history = os.path.join(example_directory, history_file)
output_name = 'noddy_out'
# call Noddy

# NOTE: Make sure that the noddy executable is accessible in the system!!
sys
print subprocess.Popen(['noddy', history, output_name],
                       shell=False, stderr=subprocess.PIPE,
                       stdout=subprocess.PIPE).stdout.read()
#
```

For convenience, the model computation is wrapped into a Python function in pynoddy:

```
pynoddy.compute_model(history, output_name)
```

Note: The Noddy call from Python is, to date, calling Noddy through the subprocess function. In a future implementation, this call could be subsituted with a full wrapper for the C-functions written in Python. Therefore, using

the member function compute_model is not only easier, but also the more "future-proof" way to compute the Noddy model.

2.2 Loading Noddy output files

Noddy simulations produce a variety of different output files, depending on the type of simulation. The basic output is the geological model. Additional output files can contain geophysical responses, etc.

Loading the output files is simplified with a class class container that reads all relevant information and provides simple methods for plotting, model analysis, and export. To load the output information into a Python object:

```
N1 = pynoddy.NoddyOutput(output_name)
```

The object contains the calculated geology blocks and some additional information on grid spacing, model extent, etc. For example:

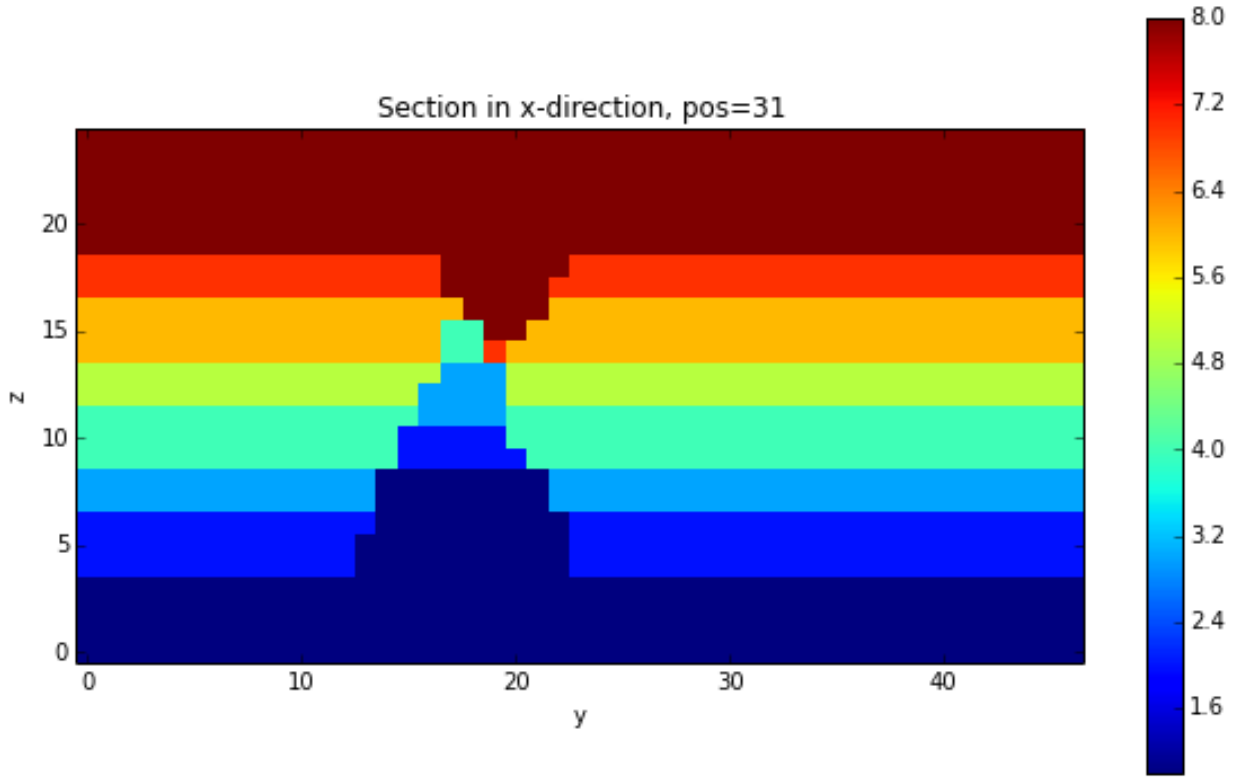
```
print("The model has an extent of %.0f m in x-direction, with %d cells of width %.0f m" %
      (N1.extent_x, N1.nx, N1.delx))
```

```
The model has an extent of 12400 m in x-direction, with 62 cells of width 200 m
```

2.3 Plotting sections through the model

The NoddyOutput class has some basic methods for the visualisation of the generated models. To plot sections through the model:

```
N1.plot_section('x')
```



2.4 Export model to VTK

A simple possibility to visualise the modeled results in 3-D is to export the model to a VTK file and then to visualise it with a VTK viewer, for example Paraview. To export the model, simply use:

```
N1.export_to_vtk()
```

CHAPTER

THREE

CHANGE NODDY INPUT FILE AND RECOMPUTE MODEL

In this section, we will briefly present possibilities to access the properties defined in the Noddy history input file and show how simple adjustments can be performed, for example changing the cube size to obtain a model with a higher resolution.

Also outlined here is the way that events are stored in the history file as single objects. For more information on accessing and changing the events themselves, please be patient until we get to the next section.

```
import sys, os
import matplotlib.pyplot as plt
# adjust some settings for matplotlib
from matplotlib import rcParams
# print rcParams
rcParams['font.size'] = 15
# determine path of repository to set paths corretly below
os.chdir(r'/Users/Florian/git/pynoddy/docs/notebooks/')
repo_path = os.path.realpath('../..')
import pynoddy
```

First step: load the history file into a Python object:

```
# Change to sandbox directory to store results
os.chdir(os.path.join(repo_path, 'sandbox'))
# Path to exmaple directory in this repository
example_directory = os.path.join(repo_path,'examples')
# Compute noddy model for history file
history_file = 'simple_two_faults.his'
history = os.path.join(example_directory, history_file)
output_name = 'noddy_out'
H1 = pynoddy.history.NoddyHistory(history)
```

Technical note: the `NoddyHistory` class can be accessed on the level of pynoddy (as it is imported in the `__init__.py` module) with the shortcut:

```
H1 = pynoddy.NoddyHistory(history)
```

I am using the long version `pynoddy.history.NoddyHistory` here to ensure that the correct package is loaded with the `reload()` function. If you don't make changes to any of the pynoddy files, this is not required. So for any practical cases, the shortcuts are absolutely fine!

3.1 Get basic information on the model

The history file contains the entire information on the Noddy model. Some information can be accessed through the NoddyHistory object (and more will be added soon!), for example the total number of events:

```
print("The history contains %d events" % H1.n_events)
```

```
The history contains 3 events
```

Events are implemented as objects, the classes are defined in `H1.events`. All events are accessible in a list on the level of the history object:

```
H1.events
```

```
{1: <pynoddy.events.Stratigraphy instance at 0x10a6c3bd8>,
 2: <pynoddy.events.Fault instance at 0x10a6c3c20>,
 3: <pynoddy.events.Fault instance at 0x10a6c3cf8>}
```

The properties of an event are stored in the event objects themselves. To date, only a subset of the properties (deemed as relevant for the purpose of pynoddy so far) are parsed. The .his file contains a lot more information! If access to this information is required, adjustments in pynoddy.events have to be made.

For example, the properties of a fault object are:

```
H1.events[2].properties
# print H1.events[5].properties.keys()
```

```
{'Amplitude': 2000.0,
 'Blue': 254.0,
 'Color Name': 'Custom Colour 8',
 'Cyl Index': 0.0,
 'Dip': 60.0,
 'Dip Direction': 90.0,
 'Event #2': 'FAULT',
 'Geometry': 'Translation',
 'Green': 0.0,
 'Movement': 'Hanging Wall',
 'Pitch': 90.0,
 'Profile Pitch': 90.0,
 'Radius': 1000.0,
 'Red': 0.0,
 'Rotation': 30.0,
 'Slip': 1000.0,
 'X': 5500.0,
 'XAxis': 2000.0,
 'Y': 3968.0,
 'YAxis': 2000.0,
 'Z': 0.0,
 'ZAxis': 2000.0}
```

3.2 Change model cube size and recompute model

The Noddy model itself is, once computed, a continuous model in 3-D space. However, for most visualisations and further calculations (e.g. geophysics), a discretised version is suitable. The discretisation (or block size) can be adapted in the history file. The according pynoddy function is change_cube_size.

A simple example to change the cube size and write a new history file:

```
# We will first recompute the model and store results in an output file for comparison
reload(pynoddy.history)
reload(pynoddy.output)
NH1 = pynoddy.history.NoddyHistory(history)
```

```
pynoddy.compute_model(history, output_name)
NO1 = pynoddy.output.NoddyOutput(output_name)
```

```
(62, 47, 25)
```

```
# Now: change cubsize, write to new file and recompute
NH1.change_cube_size(50)
# Save model to a new history file and recompute (Note: may take a while to compute now)
new_history = "fault_model_changed_cubesize.his"
new_output_name = "noddy_out_changed_cube"
NH1.write_history(new_history)
pynoddy.compute_model(new_history, new_output_name)
NO2 = pynoddy.output.NoddyOutput(new_output_name)
```

```
(248, 188, 100)
```

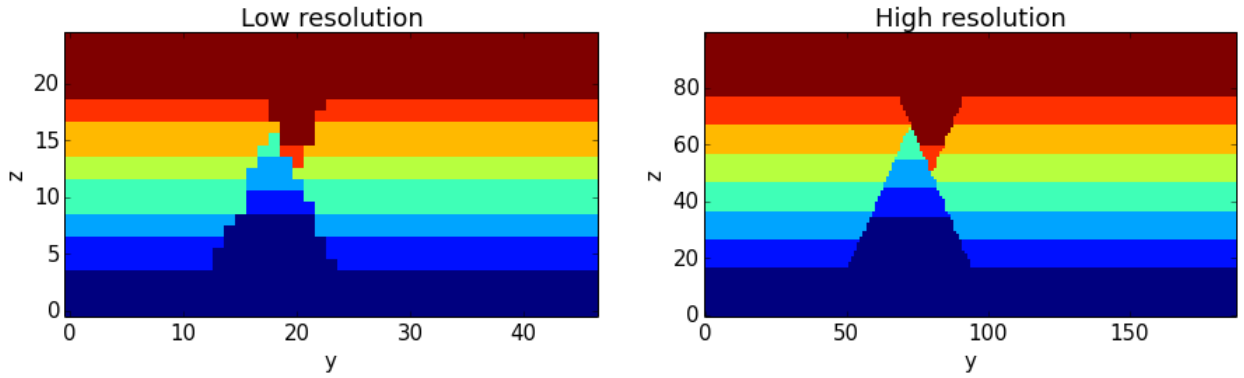
The different cell sizes are also represented in the output files:

```
print("Model 1 contains a total of %7d cells with a blocksize %.0f m" %
      (NO1.n_total, NO1.delx))
print("Model 2 contains a total of %7d cells with a blocksize %.0f m" %
      (NO2.n_total, NO2.delx))
```

```
Model 1 contains a total of   72850 cells with a blocksize 200 m
Model 2 contains a total of 4662400 cells with a blocksize 50 m
```

We can compare the effect of the different model discretisations in section plots, created with the plot_section method described before. Let's get a bit more fancy here and use the functionality to pass axes to the plot_section method, and to create one figure as direct comparison:

```
# create basic figure layout
fig = plt.figure(figsize = (15,5))
ax1 = fig.add_subplot(121)
ax2 = fig.add_subplot(122)
NO1.plot_section('x', position=0, ax = ax1, colorbar=False, title="Low resolution")
NO2.plot_section('x', position=1, ax = ax2, colorbar=False, title="High resolution")

plt.show()
```



Note: the following two subsections contain some slighly advanced examples on how to use the possibility to adjust cell sizes through scripts directly to autmote processes that are infeasible using the GUI version of Noddy - as a 'peek preview' of the automation for uncertainty estimation that follows in a later section. Feel free to skip those two sections if you are only interested in the basic features so far.

3.3 Estimating computation time for a high-resolution model

You surely realised (if you ran these examples in an actual interactive ipython notebook) that the computation of the high-resolution model takes siginificantly longer than the low-resolution model. In a practical case, this can be very important.

```
# We use here simply the time() function to evaulate the simualtion time.
# This is not the best possible way to do it, but probably the simplest.
import time
start_time = time.time()
pynoddy.compute_model(history, output_name)
end_time = time.time()

print("Simulation time for low-resolution model: %5.2f seconds" % (end_time - start_time))

start_time = time.time()
pynoddy.compute_model(new_history, new_output_name)
end_time = time.time()

print("Simulation time for high-resolution model: %5.2f seconds" % (end_time - start_time))
```

```
Simulation time for low-resolution model:  0.08 seconds
Simulation time for high-resolution model: 32.43 seconds
```

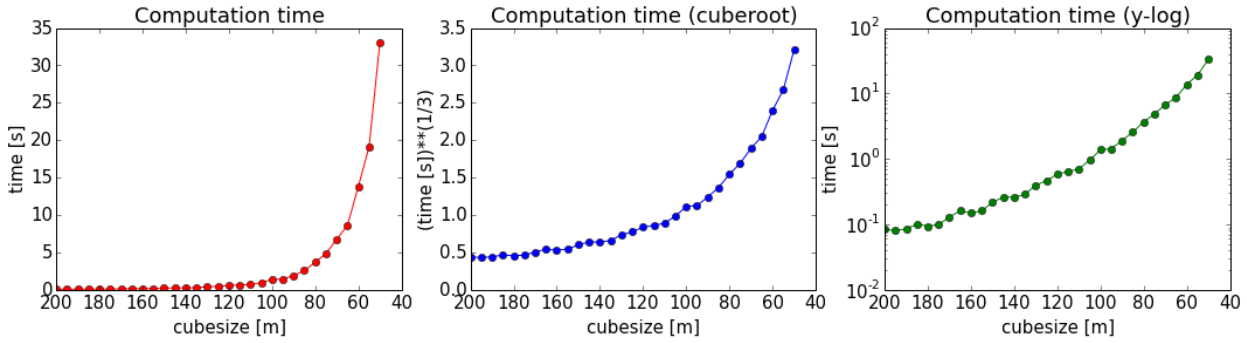
For an estimation of required computing time for a given discretisation, let's evaulate the time for a couple of steps, plot, and extrapolate:

```
# perform computation for a range of cube sizes
cube_sizes = np.arange(200,49,-5)
times = []
NH1 = pynoddy.history.NoddyHistory(history)
tmp_history = "tmp_history"
tmp_output = "tmp_output"
for cube_size in cube_sizes:
    NH1.change_cube_size(cube_size)
    NH1.write_history(tmp_history)
    start_time = time.time()
    pynoddy.compute_model(tmp_history, tmp_output)
    end_time = time.time()
    times.append(end_time - start_time)
times = np.array(times)
```

```
# create plot
fig = plt.figure(figsize=(18,4))
ax1 = fig.add_subplot(131)
ax2 = fig.add_subplot(132)
ax3 = fig.add_subplot(133)

ax1.plot(cube_sizes, np.array(times), 'ro-')
ax1.set_xlabel('cubesize [m]')
ax1.set_ylabel('time [s]')
ax1.set_title('Computation time')
ax1.set_xlim(ax1.get_xlim()[::-1])

ax2.plot(cube_sizes, times**(1/3.), 'bo-')
ax2.set_xlabel('cubesize [m]')
ax2.set_ylabel('(time [s])**(1/3)')
ax2.set_title('Computation time (cuberoot)')
```

```
ax2.set_xlim(ax2.get_xlim()[::-1])

ax3.semilogy(cube_sizes, times, 'go-')
ax3.set_xlabel('cubesize [m]')
ax3.set_ylabel('time [s]')
ax3.set_title('Computation time (y-log)')
ax3.set_xlim(ax3.get_xlim()[::-1])
```

```
(200.0, 40.0)
```



It is actually quite interesting that the computation time does not scale with cubesize to the power of three (as could be expected, given that we have a mesh in three dimensions). Or am I missing something?

Anyway, just because we can: let's assume that the scaling is somehow exponential and try to fit a model for a time prediction. Given the last plot, it looks like we could fit a logarithmic model with probably an additional exponent (as the line is obviously not straight), so something like:

```
# perform curve fitting with scipy.optimize
import scipy.optimize
# define function to be fit
def func(x,a,b,c):
    return a + (b*np.log10(x))**(-c)

popt, pcov = scipy.optimize.curve_fit(func, cube_sizes, np.array(times))
popt
```

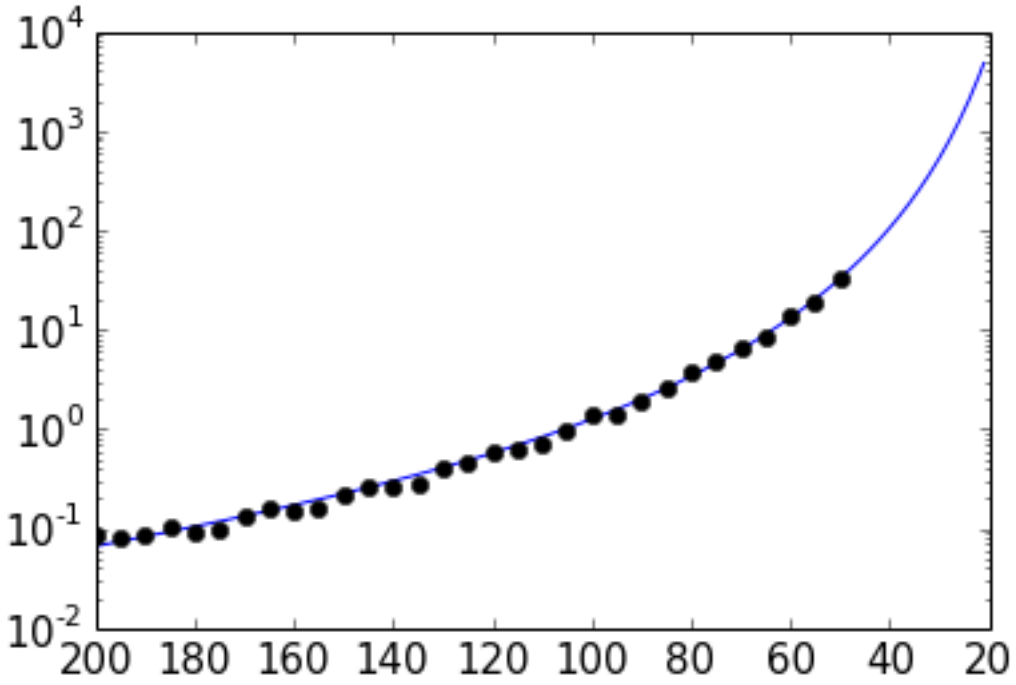
```
array([ -1.15814515e-02,   4.94030524e-01,   1.99015465e+01])
```

Interesting, it looks like Noody scales with something like:

Note: if you understand more about computational complexity than me, it might not be that interesting to you at all - if this is the case, please contact me and tell me why this result could be expected...

```
a,b,c = popt
cube_range = np.arange(200,20,-1)
times_eval = func(cube_range, a, b, c)
fig = plt.figure()
ax = fig.add_subplot(111)
ax.semilogy(cube_range, times_eval, '-')
ax.semilogy(cube_sizes, times, 'ko')
# reverse x-axis
ax.set_xlim(ax.get_xlim()[::-1])
```

```
(200.0, 20.0)
```



Not too bad... let's evaluate the time for a cube size of 40 m:

```
cube_size = 40 # m
time_est = func(cube_size, a, b, c)
print("Estimated time for a cube size of %d m: %.1f seconds" % (cube_size, time_est))
```

```
Estimated time for a cube size of 40 m: 105.0 seconds
```

Now let's check the actual simulation time:

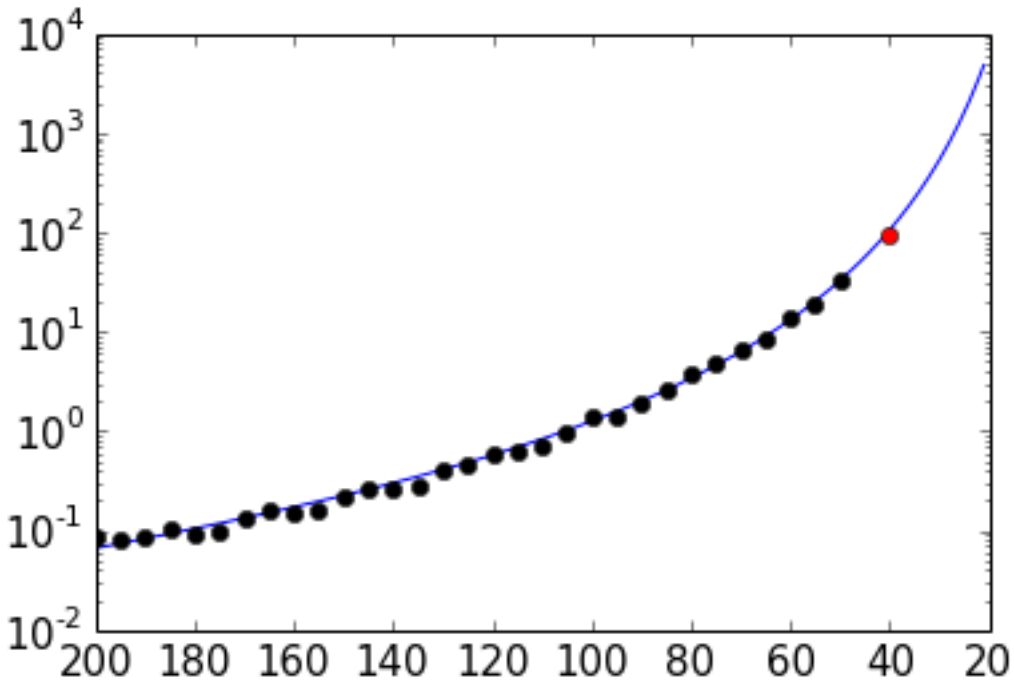
```
NH1.change_cube_size(cube_size)
NH1.write_history(tmp_history)
start_time = time.time()
pynoddy.compute_model(tmp_history, tmp_output)
end_time = time.time()
time_comp = end_time - start_time

print("Actual computation time for a cube size of %d m: %.1f seconds" % (cube_size, time_comp))
```

```
Actual computation time for a cube size of 40 m: 94.4 seconds
```

Not too bad, probably in the range of the inherent variability... and if we check it in the plot:

```
fig = plt.figure()
ax = fig.add_subplot(111)
ax.semilogy(cube_range, times_eval, '-')
ax.semilogy(cube_sizes, times, 'ko')
ax.semilogy(cube_size, time_comp, 'ro')
# reverse x-axis
ax.set_xlim(ax.get_xlim()[::-1])
```

```
(200.0, 20.0)
```



Anyway, the point of this excercise was not a precise evaluation of Noddy's computational complexity, but to provide a simple means of evaluating computation time for a high resolution model, using the flexibility of writing simple scripts using pynoddy, and a couple of additional python modules.

For a realistic case, it should, of course, be sufficient to determine the time based on a lot less computed points. If you like, test it with your favourite model and tell me if it proved useful (or not)!

3.4 Simple convergence study

So: why would we want to run a high-resolution model, anyway? Well, of course, it produces nicer pictures - but on a scientific level, that's completely irrelevant (haha, not true - so nice if it would be...).

Anyway, if we want to use the model in a scientific study, for example to evaluate volume of specific units, or to estimate the geological topology (Mark is working on this topic with some cool ideas - example to be implemented here, "soon"), we want to know if the resolution of the model is actually high enough to produce meaningful results.

As a simple example of the evaluation of model resolution, we will here inlcude a volume convergence study, i.e. we will estimate at which level of increasing model resolution the estimated block volumes do not change anymore.

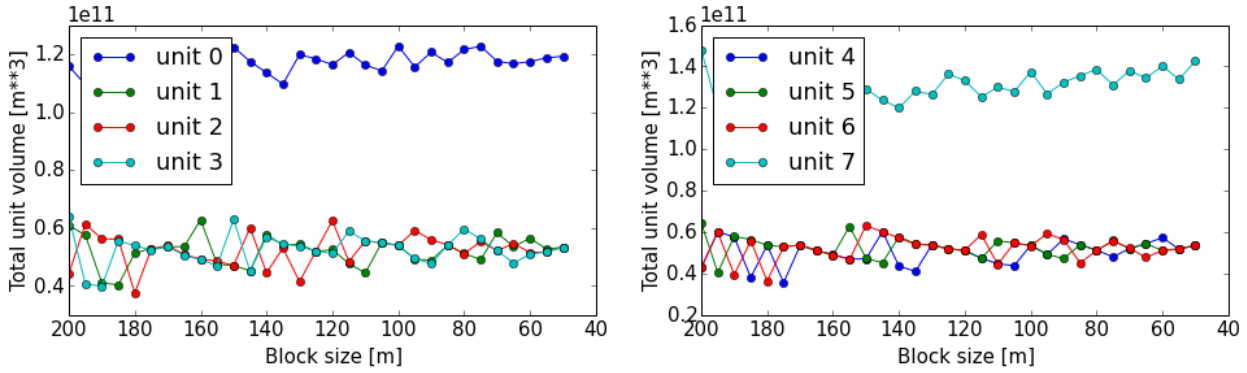
The entire procedure is very similar to the computational time evaluation above, only that we now also analyse the output and determine the rock volumes of each defined geological unit:

```
# perform computation for a range of cube sizes
reload(pynoddy.output)
cube_sizes = np.arange(200,49,-5)
all_volumes = []
N_tmp = pynoddy.history.NoddyHistory(history)
tmp_history = "tmp_history"
tmp_output = "tmp_output"
for cube_size in cube_sizes:
    # adjust cube size
    N_tmp.change_cube_size(cube_size)
```

```
    N_tmp.write_history(tmp_history)
    pynoddy.compute_model(tmp_history, tmp_output)
    # open simulated model and determine volumes
    O_tmp = pynoddy.output.NoddyOutput(tmp_output)
    O_tmp.determine_unit_volumes()
    all_volumes.append(O_tmp.unit_volumes)
```

```
all_volumes = np.array(all_volumes)
fig = plt.figure(figsize=(16,4))
ax1 = fig.add_subplot(121)
ax2 = fig.add_subplot(122)

# separate into two plots for better visibility:
for i in range(np.shape(all_volumes)[1]):
    if i < 4:
        ax1.plot(cube_sizes, all_volumes[:,i], 'o-', label='unit %d' %i)
    else:
        ax2.plot(cube_sizes, all_volumes[:,i], 'o-', label='unit %d' %i)

ax1.legend(loc=2)
ax2.legend(loc=2)
# reverse axes
ax1.set_xlim(ax1.get_xlim()[::-1])
ax2.set_xlim(ax2.get_xlim()[::-1])

ax1.set_xlabel("Block size [m]")
ax1.set_ylabel("Total unit volume [m**3]")
ax2.set_xlabel("Block size [m]")
ax2.set_ylabel("Total unit volume [m**3]")
```

```
<matplotlib.text.Text at 0x1103cc810>
```



It looks like the volumes would start to converge from about a block size of 100 m. The example model is pretty small and simple, probably not the best example for this study. Try it out with your own, highly complex, favourite pet model :-)

CHAPTER

FOUR

GEOLOGICAL EVENTS IN PYNODDY: ORGANISATION AND ADPATIATION

We will here describe how the single geological events of a Noddy history are organised within pynoddy. We will then evaluate in some more detail how aspects of events can be adapted and their effect evaluated.

4.1 Loading events from a Noddy history

In the current set-up of pynoddy, we always start with a pre-defined Noddy history loaded from a file, and then change aspects of the history and the single events. The first step is therefore to load the history file and to extract the single geological events. This is done automatically as default when loading the history file into the History object:

```
import sys, os
import matplotlib.pyplot as plt
# adjust some settings for matplotlib
from matplotlib import rcParams
# print rcParams
rcParams['font.size'] = 15
# determine path of repository to set paths corretly below
os.chdir(r'/Users/Florian/git/pynoddy/docs/notebooks/')# some basic module imports
repo_path = os.path.realpath('../..')

import pynoddy
```

```
# Change to sandbox directory to store results
os.chdir(os.path.join(repo_path, 'sandbox'))

# Path to exmaple directory in this repository
example_directory = os.path.join(repo_path,'examples')
# Compute noddy model for history file
history = 'simple_two_faults.his'
history_ori = os.path.join(example_directory, history_file)
output_name = 'noddy_out'
reload(pynoddy.history)
reload(pynoddy.events)
H1 = pynoddy.history.NoddyHistory(history_ori)
# Before we do anything else, let's actually define the cube size here to
# adjust the resolution for all subsequent examples
H1.change_cube_size(100)
# compute model - note: not strictly required, here just to ensure changed cube size
H1.write_history(history)
pynoddy.compute_model(history, output_name)
```

Events are stored in the object dictionary "events" (who would have thought), where the key corresponds to the position in the timeline:

```
H1.events
```

```
{1: <pynoddy.events.Stratigraphy instance at 0x1046863b0>,
 2: <pynoddy.events.Fault instance at 0x1046863f8>,
 3: <pynoddy.events.Fault instance at 0x104686950>}
```

We can see here that three events are defined in the history. Events are organised as objects themselves, containing all the relevant properties and information about the events. For example, the second fault event is defined as:

```
H1.events[3].properties
```

```
{'Amplitude': 2000.0,
 'Blue': 0.0,
 'Color Name': 'Custom Colour 5',
 'Cyl Index': 0.0,
 'Dip': 60.0,
 'Dip Direction': 270.0,
 'Event #3': 'FAULT',
 'Geometry': 'Translation',
 'Green': 0.0,
 'Movement': 'Hanging Wall',
 'Pitch': 90.0,
 'Profile Pitch': 90.0,
 'Radius': 1000.0,
 'Red': 254.0,
 'Rotation': 30.0,
 'Slip': 1000.0,
 'X': 5500.0,
 'XAxis': 2000.0,
 'Y': 7000.0,
 'YAxis': 2000.0,
 'Z': 5000.0,
 'ZAxis': 2000.0}
```

4.2 Changing aspects of geological events

So what we now want to do, of course, is to change aspects of these events and to evaluate the effect on the resulting geological model.

Changes are best

```
reload(pynoddy.history)
reload(pynoddy.events)
H1 = pynoddy.history.NoddyHistory(history)
# get the original dip of the fault
dip_ori = H1.events[3].properties['Dip']
# dip_ori1 = H1.events[2].properties['Dip']
# add 10 degrees to dip
add_dip = -10
dip_new = dip_ori + add_dip
# dip_new1 = dip_ori1 + add_dip

# and assign back to properties dictionary:
```

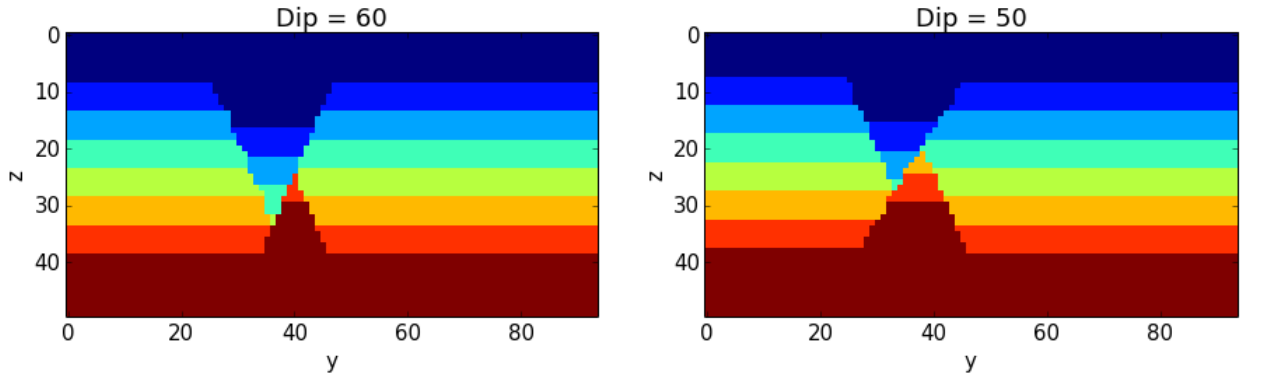
```
H1.events[3].properties['Dip'] = dip_new
# H1.events[2].properties['Dip'] = dip_new1
```

```
H1.events[3]
```

```
<pynoddy.events.Fault instance at 0x10437e950>
```

What is left now is to write the model back to a new history file, to recompute the model, and then visualise the output, as before, to compare the results:

```
reload(pynoddy.output)
new_history = "dip_changed"
new_output = "dip_changed_out"
H1.write_history(new_history)
pynoddy.compute_model(new_history, new_output)
# load output from both models
NO1 = pynoddy.output.NoddyOutput(output_name)
NO2 = pynoddy.output.NoddyOutput(new_output)

# create basic figure layout
fig = plt.figure(figsize = (15,5))
ax1 = fig.add_subplot(121)
ax2 = fig.add_subplot(122)
NO1.plot_section('x', position=0, ax = ax1, colorbar=False, title="Dip = %.0f" % dip_ori)
NO2.plot_section('x', position=1, ax = ax2, colorbar=False, title="Dip = %.0f" % dip_new)

plt.show()
```



4.3 Changing the order of geological events

The geological history is parameterised as single events in a timeline. Changing the order of events can be performed with two basic methods:

1. Swapping two events with a simple command
2. Adjusting the entire timeline with a complete remapping of events

The first method is probably the most useful to test how a simple change in the order of events will effect the final geological model. We will use it here with our example to test how the model would change if the timing of the faults is swapped.

The method to swap two geological events is defined on the level of the history object:

```
reload(pynoddy.history)
reload(pynoddy.events)
H1 = pynoddy.history.NoddyHistory(history)
H1.change_cube_size(100)
# compute model - note: not strictly required, here just to ensure changed cube size

H1.write_history(history)
pynoddy.compute_model(history, output_name)
```

```
# The names of the two fault events defined in the history file are:
print H1.events[2].name
print H1.events[3].name
```

```
Fault2
Fault1
```

```
# Now: swap the events:
H1.swap_events(2,3)
```

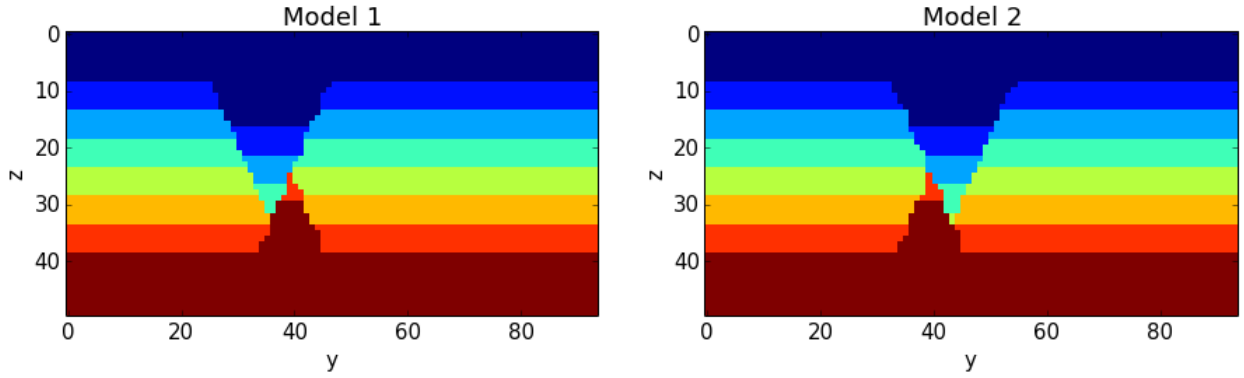
```
# And let's check if this is correctly relfected in the events order now:
print H1.events[2].name
print H1.events[3].name
```

```
Fault1
Fault2
```

Now let's create a new history file and evaluate the effect of the changed order in a cross section view:

```
new_history = "faults_changed_order.his"
new_output = "faults_out"
H1.write_history(new_history)
pynoddy.compute_model(new_history, new_output)
```

```
reload(pynoddy.output)
# Load and compare both models
NO1 = pynoddy.output.NoddyOutput(output_name)
NO2 = pynoddy.output.NoddyOutput(new_output)
# create basic figure layout
fig = plt.figure(figsize = (15,5))
ax1 = fig.add_subplot(121)
ax2 = fig.add_subplot(122)
NO1.plot_section('x', ax = ax1, colorbar=False, title="Model 1")
NO2.plot_section('x', ax = ax2, colorbar=False, title="Model 2")

plt.show()
```



4.4 Determining the stratigraphic difference between two models

Just as another quick example of a possible application of pynoddy to evaluate aspects that are not simply possible with, for example, the GUI version of Noddy itself. In the last example with the changed order of the faults, we might be interested to determine where in space this change had an effect. We can test this quite simply using the `NoddyOutput` objects.

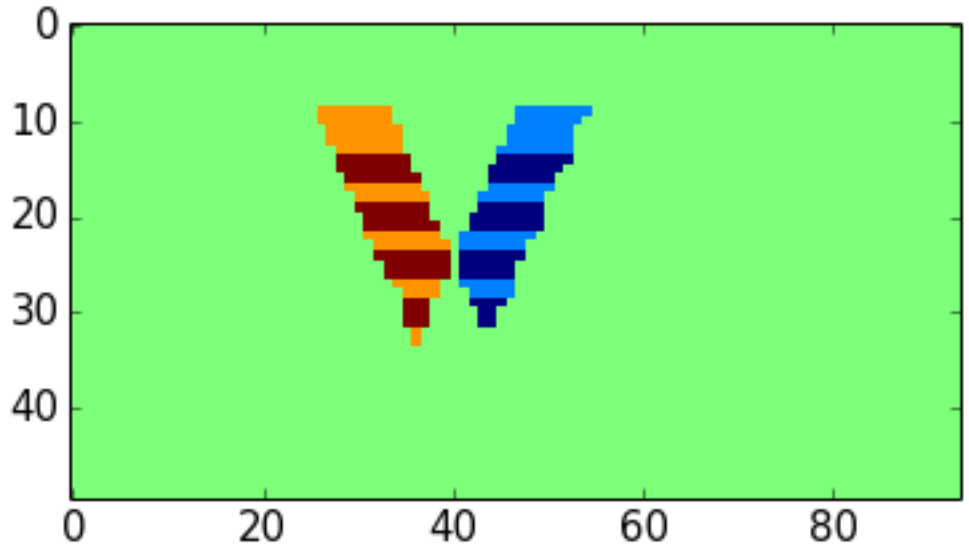
The geology data is stored in the `NoddyOutput.block` attribute. To evaluate the difference between two models, we can therefore simply compute:

```
diff = (NO2.block - NO1.block)
```

And create a simple visualisation of the difference in a slice plot with:

```
fig = plt.figure()
ax = fig.add_subplot(111)
ax.imshow(diff[10,:,:].transpose(), interpolation='nearest')
```

```
<matplotlib.image.AxesImage at 0x104651b90>
```



(Adding a meaningful title and axis labels to the plot is left to the reader as simple excercise :-) Future versions of pynoddy might provide an automatic implementation for this step...)

CHAPTER

FIVE

PYNODDY MODULES, CLASSES AND FUNCTIONS

5.1 Basic modules (low-level access)

The modules in this section provide low-level access to the functionality in Noddy. Basically, these modules provide parsers for Noddy input and output files and class definitions for suitable Noddy elements.

5.1.1 Main module

Package initialization file for pynoddy

`pynoddy.`**`compute_model`**(*history*, *output_name*, ***kwds*)

`pynoddy.`**`compute_topology`**(*rootname*, ***kwds*)

5.1.2 History file parser: pynoddy.history

Noddy history file wrapper Created on 24/03/2014

@author: Florian Wellmann

class `pynoddy.history.`**`NoddyHistory`**(*history=None*, ***kwds*)

Bases: `object`

Class container for Noddy history files

`add_event`(*event_type*, *event_options*, ***kwds*)

Add an event type to history

Arguments:

- *event_type* = string : type of event, legal options to date are:

'stratigraphy', 'fault', 'fold', 'unconformity' - *event_options* = list : required options to create event (event dependent)

Optional keywords:

- *event_num* = int : event number (default: implicitly defined with increasing counter)

`change_cube_size`(*cube_size*, ***kwds*)

Change the model cube size (isotropic)

Arguments:

- *cube_size* = float : new model cube size

change_event_params(*changes_dict*)
Change multiple event parameters according to settings in changes_dict

Arguments:

- *changes_dict* = dictionary : entries define relative changes for (multiple) parameters

Per default, the values in the dictionary are added to the event parameters.

copy_events()
Create a copy of the current event state

create_footer_from_template()
Create model footer (with all settings) from template

create_new_history()
Methods to create a Noddy model

determine_events(***kwds*)
Determine events and save line numbers

Note:

Parsing of the history file is based on a fixed Noddy output order. If this is, for some reason (e.g. in a changed version of Noddy) not the case, then this parsing might fail!

Optional Keywords:

- verbose = True if this function is should write to the print bufffer, otherwise False. Default is true.

---

determine_model_stratigraphy()
Determine stratigraphy of entire model from all events

get_cube_size(***kwds*)
Determine cube size for model export **Optional Args**

-type: choose geology or geophysics cube size to return. Should be either 'Geology' (default) or 'Geophysics'

get_date_saved()
Determine the last savepoint of the file

get_drillhole_data(*x*, *y*, ***kwds*)
Get geology values along 1-D profile at position x,y with a 1 m resolution

The following steps are performed: 1. creates a copy of the entire object, 2. sets values of origin, extent and geology cube size, 3. saves model to a temporary file, 4. runs Noddy on that file 5. opens and analyses output 6. deletes temporary files

Note: this method only works if write access to current directory is enabled and noddy can be executed!

Arguments:

- *x* = float: x-position of drillhole
- *y* = float: y-position of drillhole

Optional Arguments:

- *z_min* = float : minimum depth of drillhole (default: model range)
- *z_max* = float : maximum depth of drillhole (default: model range)
- *resolution* = float : resolution along profile (default: 1 m)

get_ev_counter()
Event counter for implicit and continuous definition of events

get_event_param(*event_number*, *name*)
Returns the value of a given parameter for a given event.

Arguments:

- *event_number* = the event to get a parameter for (integer)
- *name* = the name of the parameter to retreive (string)

Returns

- Returns the value of the request parameter, or None if it does not exists.

get_event_params(*event_number*)
Returns the parameter dictionary for a given event.

Arguments:

- *event_number* = the event to get a parameter for (integer)

Returns

- Returns the parameter dictionary for the requested event

get_extent()
Get model extent and return and store in local variables

Returns: (extent_x, extent_y, extent_z)

get_filename()
Determine model filename from history file/ header

get_footer_lines()
Get the footer lines from self.history_lines

The footer contains everything below events (all settings, etc.)

get_info_string(***kwds*)
Get model information as string

Optional keywords:

- *events_only* = bool : only information on events

get_origin()
Get coordinates of model origin and return and store in local variables

Returns: (origin_x, origin_y, origin_z)

info(***kwds*)
Print out model information

Optional keywords:

- *events_only* = bool : only information on events

load_history(*history*)
Load Noddy history

Arguments:

- *history* = string : Name of Noddy history file

load_history_from_url(*url*)
Directly load a Noddy history from a URL

This method is useful to load a model from the Structural Geophysics Atlas on the pages of the Virtual Explorer. See: http://virtualexplorer.com.au/special/noddyatlas/index.html

Arguments:

- *url* : url of history file

reorder_events(*reorder_dict*)
Reorder events accoring to assignment in reorder_dict

Arguments:

- *reorder_dict* = dict : for example {1 : 2, 2 : 3, 3 : 1}

set_event_params(*params_dict*)
set multiple event parameters according to settings in params_dict

Arguments:

- *params_dict* = dictionary : entries to set (multiple) parameters

set_extent(*extent_x*, *extent_y*, *extent_z*)
Set model extent and update local variables

Arguments:

- *extent_x* = float : extent in x-direction
- *extent_y* = float : extent in y-direction
- *extent_z* = float : extent in z-direction

set_origin(*origin_x*, *origin_y*, *origin_z*)
Set coordinates of model origin and update local variables

Arguments:

- *origin_x* = float : x-location of model origin
- *origin_y* = float : y-location of model origin
- *origin_z* = float : z-location of model origin

swap_events(*event_num_1*, *event_num_2*)
Swap two geological events in the timeline

Arguments:

- *event_num_1/2* = int : number of events to be swapped ("order")

update_all_event_properties()
Update properties of all events - in case changes were made

update_event_numbers()
Update event numbers in 'Event #' line in noddy history file

write_history(*filename*)
Write history to new file

Arguments:

- *filename* = string : filename of new history file

Hint: Just love it how easy it is to 'write history' with Noddy ;-)

5.1.3 Output file parser: pynoddy.output

Noddy output file analysis Created on 24/03/2014

@author: Florian Wellmann, Sam Thiele

class pynoddy.output.**NoddyGeophysics**(*output_name*)
Bases: object

Definition to read, analyse, and visualise calculated geophysical responses

read_gravity()
Read calculated gravity response

read_magnetics()
Read caluclated magnetic field response

class pynoddy.output.**NoddyOutput**(*output_name*)
Bases: object

Class definition for Noddy output analysis

compare_dimensions_to(*other*)
Compare model dimensions to another model

determine_unit_volumes()
Determine volumes of geological units in the discretized block model

export_to_vtk(***kwds*)
Export model to VTK

Export the geology blocks to VTK for visualisation of the entire 3-D model in an external VTK viewer, e.g. Paraview.

..Note:: Requires pyevtk, available for free on: https://github.com/firedrakeproject/firedrake/tree/master/python/evtk

Optional keywords:

- *vtk_filename* = string : filename of VTK file (default: output_name)
- *data* = np.array : data array to export to VKT (default: entire block model)

get_section_lines(*direction='y'*, *position='center'*, ***kwds*)
Create and returns a list of lines representing a section block through the model

Arguments:

- *direction* = 'x', 'y', 'z' : coordinate direction of section plot (default: 'y')
- ***position* = int or 'center'** [cell position of section as integer value] or identifier (default: 'center')

Returns: A tuple of lists of dictionaries.... ie: ([dictionary of x coordinates, with lithology pairs as keys, separated by an underscore],

[dictionary of y coordinates, with lithology pairs as keys, separated by an underscore], [dictionary of z coordinates, with lithology pairs as keys, separated by an underscore], [dictionary of colours, with lithologies as keys])

For example: get_section_lines()[0]["1_2"] returns a list of all the x coordinates from the contact between lithology 1 and lithology 2. Note that the smaller lithology index always comes first in the code.

get_section_voxels(*direction='y'*, *position='center'*, ***kwds*)
Create and returns section block through the model

Arguments:

- *direction* = 'x', 'y', 'z' : coordinate direction of section plot (default: 'y')
- ***position* = int or 'center'** [cell position of section as integer value] or identifier (default: 'center')

Optional Keywords:

- *data* = np.array : data to plot, if different to block data itself
- *litho_filter* = a list of lithologies to draw. All others will be ignored.

get_surface_grid(*lithoID*, ***kwds*)

Returns a grid of lines that define a grid on the specified surface. Note that this cannot handle layers that are repeated in the z direction...

Arguments:

- ***lithoID* - the top surface of this lithology will be calculated. If a list is passed,** the top surface of each lithology in the list is calculated.

Keywords:

- *res* - the resolution to sample at. Default is 2 (ie. every second voxel is sampled).

Returns: a tuple containing lists of tuples of x, y and z coordinate dictionaries and colour dictionaries, one containing the east-west lines and one the north-south lines: ((x,y,z,c),(x,y,z,c)). THe dictionary keys are the lithoID's passed in the lithoID parameter.

load_geology()

Load block geology ids from .g12 output file

load_model_info()

Load information about model discretisation from .g00 file

plot_section(*direction='y'*, *position='center'*, ***kwds*)

Create a section block through the model

Arguments:

- *direction* = 'x', 'y', 'z' : coordinate direction of section plot (default: 'y')
- ***position* = int or 'center'** [cell position of section as integer value] or identifier (default: 'center')

Optional Keywords:

- *ax* = matplotlib.axis : append plot to axis (default: create new plot)
- *figsize* = (x,y) : matplotlib figsize
- *colorbar* = bool : plot colorbar (default: True)
- ***colorbar_orientation* = 'horizontal' or 'vertical'** [orientation of colorbar] (default: 'vertical')
- *title* = string : plot title
- *savefig* = bool : save figure to file (default: show directly on screen)
- *cmap* = matplotlib.cmap : colormap (default: YlOrRd)
- *fig_filename* = string : figure filename
- *ve* = float : vertical exaggeration
- *layer_labels* = list of strings: labels for each unit in plot
- *layers_from* = noddy history file : get labels automatically from history file

- *data* = np.array : data to plot, if different to block data itself
- *litho_filter* = a list of lithologies to draw. All others will be ignored.

set_basename(*name*)
Set model basename

class pynoddy.output.**NoddyTopology**(*noddy_model*, ***kwds*)
Bases: object

Definition to read, analyse, and visualise calculated voxel topology

calculate_difference(*G2*, *data=False*)
Calculates the difference between this NoddyTopology and another NoddyTopology or networkX graph

Arguments

- *G2* = a valid NoddyTopology object or NetworkX graph that this topology is to be compared with

Returns A tuple containing: - The number of different edges - a list of these edges

calculate_overlap(*G2*)
Calculates the overlap between this NoddyTopology and another NoddyTopology or networkX graph

Arguments

- *G2* = a valid NoddyTopology object or NetworkX graph that this topology is to be compared with

Returns

- The number of overlapping edges
- A list of these edges

static calculate_unique_topologies(*topology_list*, ***kwds*)
Calculates the number of unique topologies in a list of NoddyTopologies

Arguments:

- *topology_list* = The list of NoddyTopologies to search through.

Optional Keywords:

- *output* = A File or list to write cumulative observed topologies distribution. Default is None (nothing written).
- ***ids* = A list to write the unique topology id's for each topology in the provided topology_list (in that** order). Default is None.
- *frequency* = A list to write frequency counts to.

Returns:

- Returns a list of unique topologies.

collapse_stratigraphy()
Collapses all stratigraphic edges in this network to produce a network that only contains structurally bound rock volumes. Essentially this is a network built only with Topology codes and ignoring lithology

Returns

- a new NoddyTopology object containing the collapsed graph. The original object is not modified.

collapse_topology(*verbose=False*)
Collapses all topology codes down to the last (most recent) difference. Information regarding specific model topology is generalised, eg. lithology A has a fault and stratigrappic contact with B (regardless of how many different faults are involved).

Note that this function has not been properly tested, and i'm not exactly sure what it does...

Optional Arguments:

- *verbose* = True if this function should write to the print buffer. Default is False.

Returns

- a new NoddyTopology object containing the collapsed graph. The original object is not modified.

static `combine_topologies`(*topology_list*)
Combines a list of topology networks into a weighted 'super-network'. This is designed for estimating the likelyhood of a given edge occuring using a series of networks generated in a Monte-Carlo type analysis.

Arguments

- *topology_list* = A list of networkX graphs or NoddyTopology objects to build supernetwork from.

Returns

- A NetworkX graph object containing all edges from the input graphs and weighted ('weight' parameter) according to their observed frequency.

`draw_3d_network`(***kwds*)
Draws a 3D network using matplotlib.

Optional Keywords:

- ***show* = If True, the 3D network is displayed immediatly on-screen in an** interactive mayavi viewer. Default is True.
- *output* = If defined an image of the network is saved to this location.
- *node_size* = The size of the nodes. Default is 40.
- *geology* = a NoddyOutput object to draw with the network
- *res* = resolution to draw geology at. Default is 4 (ie 1/4 of all voxels are drawn)
- ***horizons* = a list of geology surfaces to draw. Default is nothing (none drawn). Slow!** See NoddyOutput.get_surface_grid for details.
- *sections* = draw geology sections. Default is True.

`draw_adjacency_matrix`(***kwds*)
Draws an adjacency matrix representing this topology object.

Keywords:

- *path* = The path to save this image to. If not provided, the image is drawn to the screen
- *dpi* = The resolution to save this image. Default is 300
- *size* = The size of the image to save (in inches). This value will be used as the width and the height

`draw_difference_matrix`(*G2*, ***kwds*)
Draws an adjacency matrix containing the difference between this topology and the provided topology

Arguments:

- *G2* = A different NoddyTopology or NetworkX Graph to compare to

Optional Keywords:

- *path* = The path to save this image to. If not provided, the image is drawn to the screen
- *dpi* = The resolution to save this image. Default is 300

- *size* = The size of the image to save (in inches). This value will be used as the width and the height

static draw_graph_matrix(*G*, ***kwds*)

Draws an adjacency matrix representing the specified graph object. Equivalent to NoddyTopology.draw_matrix_image() but for a networkX graph object.

Keywords:

- *path* = The path to save this image to. If not provided, the image is drawn to the screen
- *dpi* = The resolution to save this image. Default is 300
- *size* = The size of the image to save (in inches). This value will be used as the width and the height

draw_network_hive(***kwds*)

Draws a network hive plot (see https://github.com/ericmjl/hiveplot). The axes of the hive are: node lithology, edge age & edge area.

ie. the top axis lists the nodes in stratigraphic order. The second axis lists edges in structural age & thrid axis lists edges by surface area.

Nodes are joined to edge-nodes by lines on the graph if they are topologically linked (ie. if an edge has that node as an end point).

Optional Keywords - *path* = the path to save this figure - *dpi* = the resolution of the figure - *bg* = the background color. Default is black.

draw_network_image(*outputname=''*, ***kwds*)

Draws a network diagram of this NoddyTopology to the specified image

Arguments

- *outputname* = the path of the image being written. If left as '' the image is written to the same directory as the basename.

Optional Keywords

- *dimension* = '2D' for a 2D network diagram or '3D' for a 3D network diagram. Default is '2D'.
- *axis* = the axis to view on for 3D network diagrams
- *perspective* = True to use perspective projection, or False for orthographic projection. Default is False.
- *node_size* = The size that nodes are drawn. Default is 1500.
- ***layout* = The layout algorithm used in 2D. Options are 'spring_layout' (default), 'shell_layout', 'circular_lay**
  and 'spectral_layout'.
- *verbose* = True if this function is allowed to write to the print buffer, otherwise false. Default is true

filter_node_volumes(*min_volume=50*)

Removes all nodes with volumes less than the specified size

Arguments:

- *min_volume* = the threshold volume. Nodes with smaller volumes are deleted.

Returns

- returns the number of deleted nodes

find_first_match(*known*)
Identical to is_unique, except that the index of the first match is returned if this matches, otherwise -1 is returned. **Arguments**:

-*known* = a list of valid NoddyTopology objects or NetworkX graphs to compare with.

Returns:

- Returns the index of the first matching topology object, or -1

find_matching(*known*)
Finds the first matching NoddyTopology (or NetworkX graph) in the specified list

Arguments: -*known* = a list of valid NoddyTopology objects or NetworkX graphs to compare with.

Returns:

- Returns the first matching object (jaccard coefficient = 1), or otherwise None

is_unique(*known*)
Returns True if the topology of this model is different (ie. forms a different network) to a list of models.

Arguments: -*known* = a list of valid NoddyTopology objects or NetworkX graphs to compare with.

Returns:

- Returns true if this topology is unique, otherwise false

jaccard_coefficient(*G2*)
Calculates the Jaccard Coefficient (ratio between the intersection & union) of the graph representing this NOddyTopology and G2.

Arguments

- *G2* = a valid NoddyTopology object or NetworkX graph that this topology is to be compared with

Returns

- The jaccard_coefficient

loadNetwork()
Loads the topology network into a NetworkX datastructure

read_adjacency_matrix()
DEPRECIATED Read max number of lithologies aross all models

read_properties()

write_summary_file(*path*, *append=True*)
Writes summary information about this network to a file

Optional Arguments

- ***append* = True if summary information should be appended to the file. If so the file is written as a csv spread**
  Default is true. If False is passed, a single, detailed summary is written for this network.

5.1.4 Additional useful classes

pynoddy.events

Module for reading and manipulating geological events Created on Mar 26, 2014

@author: Florian Wellmann

class pynoddy.events.**Dyke**(***kwds*)
Bases: *pynoddy.events.Event*

Dyke event

parse_event_lines(*lines*)
Read specific event lines from history file **Arguments**:

•*lines* = list of lines : lines with event information (as stored in .his file)

class pynoddy.events.**Event**(***kwds*)
Bases: object

Main class container for geological events

Include here all elements that events have in common (position, etc. - possibly even things like color and other aspects that are defined in the history... Parse for equal settings and include here!)

set_event_lines(*lines*)
Explicitly define event lines

set_event_number(*num*)
Set number in 'Event #' line to num

update_properties(***kwds*)
Update properties (required if self.properties assignment changed!)

class pynoddy.events.**Fault**(***kwds*)
Bases: *pynoddy.events.Event*

Fault event

parse_event_lines(*lines*)
Read specific event lines from history file

Arguments:

- *lines* = list of lines : lines with event information (as stored in .his file)

class pynoddy.events.**Fold**(***kwds*)
Bases: *pynoddy.events.Event*

Folding event

parse_event_lines(*lines*)
Read specific event lines from history file

Arguments:

- *lines* = list of lines : lines with event information (as stored in .his file)

class pynoddy.events.**Plug**(***kwds*)
Bases: *pynoddy.events.Event*

class pynoddy.events.**Strain**(***kwds*)
Bases: *pynoddy.events.Event*

Strain event

parse_event_lines(*lines*)
Read specific event lines from history file **Arguments**:

•*lines* = list of lines : lines with event information (as stored in .his file)

class pynoddy.events.**Stratigraphy**(***kwds*)
Bases: *pynoddy.events.Event*

Sedimentary pile with defined stratigraphy

parse_event_lines(*lines*)
Read specific event lines from history file

Arguments:

- *lines* = list of lines : lines with event information (as stored in .his file)

class pynoddy.events.**Tilt**(**kwds*)
Bases: *pynoddy.events.Event*

Tilt event

parse_event_lines(*lines*)
Read specific event lines from history file

Arguments:

- *lines* = list of lines : lines with event information (as stored in .his file)

class pynoddy.events.**Unconformity**(**kwds*)
Bases: *pynoddy.events.Event*

Unconformity event

change_height(*val*)
Change the vertical position (i.e. height) of the entire stratigraphic pile above the unconformity

---

Note: This is not identical to changing only the 'Z' property as the height of all layers has to be adjusted for (geological) consistency

---

Arguments:

- *val* = float : value added to current z-values

parse_event_lines(*lines*)
Read specific event lines from history file

Arguments:

- *lines* = list of lines : lines with event information (as stored in .his file)

5.2 Modules for Kinematic experiments

The modules described in this section are designed to provide a high-level access to the kinematic modelling functionality in Noddy. The modules encapsulate the required aspects of complete experiments, including input file generation, adaptation of parameters, random number generation, model computation, and postprocessing.

5.2.1 Base classes for pynoddy experiments

The base class for any type of experiments is defined in the pynoddy.experiment module. Base class from which PyNoddy experiments should inherit.

Much basic functionality (random perturbation, plotting etc. is defined here).

Thought: perhaps drawing functions etc. should be moved into NoddyOutput class?

@author: flohorovicic, samthiele

class pynoddy.experiment.**Experiment**(*history=None*, ***kwds*)
Bases: *pynoddy.history.NoddyHistory*, *pynoddy.output.NoddyOutput*

Noddy experiment container, inheriting from both noddy history and output methods

export_to_vtk(***kwds*)
Export model to VTK

Export the geology blocks to VTK for visualisation of the entire 3-D model in an external VTK viewer, e.g. Paraview.

..Note:: Requires pyevtk, available for free on: https://github.com/firedrakeproject/firedrake/tree/master/python/evtk

Optional keywords:

- *vtk_filename* = string : filename of VTK file (default: output_name)
- *data* = np.array : data array to export to VKT (default: entire block model)
- *recompute* = bool : recompute the block model (default: True)
- *model_type* = 'current', 'base' : model type (base "freezed" model can be plotted for comparison)

..Note:: If data is defined, the model is not recomputed and the data from this array is plotted

freeze(***kwds*)
Freeze the current model state: store the event settings for later comparison

get_sampling_line_data(*xyz_from*, *xyz_to*)
Get computed model along a line, for example as a drillhole position

Arguments:

- *xyz_from* = [x, y, z] : list of float values for starting position
- *xyz_to* = [x, y, z] : list of float values for starting position

get_section(*direction='y'*, *position='center'*, ***kwds*)
Get geological section of the model (re-computed at required resolution) as noddy object

Arguments:

- *direction* = 'x', 'y', 'z' : coordinate direction of section plot (default: 'y')
- ***position* = int or 'center'** [cell position of section as integer value] or identifier (default: 'center')

Optional arguments:

- *resolution* = float : set resolution for section (default: self.cube_size)
- *model_type* = 'current', 'base' : model type (base "freezed" model can be plotted for comparison)
- *compute_output* = bool : provide output from command line call (default: True)

get_up_to_date()
Get model state

is_up_to_date
Model state

load_parameter_file(*filename*, ***kwds*)
Load parameter statistics from external csv file

The csv file should contain a header row with the relevant keywords identifying columns. In order to be read in correctly, the header should contain the labels:

•'event' : event id

- 'parameter' : Noddy parameter ('Dip', 'Dip Direction', etc.)
- 'mean' : mean parameter value
- 'type' = 'normal', 'vonmises' or 'uniform'.

In addition, it is necessary to define PDF type and parameters. For now, the following settings are supported: - '+-' = Defines the 2.5th and 97.5th percentiles of the distribution,

similar to a 95% confidence interval.

- 'stdev' = standard deviation. Only works if type='normal'.
- 'min' = The minimum value of a uniform distribution (if type='uniform')
- 'max' = The maximum value of a uniform distribution (if type='uniform')

Arguments:

- *filename* = string : filename

Optional arguments:

- *delim* = string : delimiter (default: ',' or ';', both checked)

plot_section(*direction='y'*, *position='center'*, ***kwds*)

Extended version of plot_section method from pynoddy.output class

Arguments:

- *direction* = 'x', 'y', 'z' : coordinate direction of section plot (default: 'y')
- ***position* = int or 'center'** [cell position of section as integer value] or identifier (default: 'center')

Optional Keywords:

- *ax* = matplotlib.axis : append plot to axis (default: create new plot)
- *figsize* = (x,y) : matplotlib figsize
- *colorbar* = bool : plot colorbar (default: True)
- ***colorbar_orientation* = 'horizontal' or 'vertical'** [orientation of colorbar] (default: 'vertical')
- *title* = string : plot title
- *savefig* = bool : save figure to file (default: show directly on screen)
- *cmap* = matplotlib.cmap : colormap (default: YlOrRd)
- *fig_filename* = string : figure filename
- *ve* = float : vertical exaggeration
- *layer_labels* = list of strings: labels for each unit in plot
- *layers_from* = noddy history file : get labels automatically from history file
- *resolution* = float : set resolution for section (default: self.cube_size)
- *model_type* = 'current', 'base' : model type (base "freezed" model can be plotted for comparison)
- *data* = np.array : data to plot, if different to block data itself

random_draw(***kwds*)
Perform a random draw for parameter distributions as defined, and calculate model

This method is based on the model "base-state", and not the current state (as opposed to the self.random_perturbation() method).

Optional Keywords:

- *verbose* = bool: print out parameter changes as they happen (default: False)
- *store_params* = bool : store random parameter set (default: True)

random_perturbation(***kwds*)
Perform a random perturbation of the model according to parameter statistics defined in self.param_stats.

Note that by default, this function is identical to random_draw. If model_type is set to 'current', then parameters are varied according using the *current values* as distribution means - this allows 'random walk' away from the initial model state, which is usually not desired.

Optional arguments:

- *store_params* = bool : store random parameter set (default: True)
- *verbose* = bool: print out parameter changes as they happen (default: False)
- ***model_type* = 'base', 'current'** [perturb on basis of current model,] or use base model (default: 'base' model)

reset_base()
Set events back to base model (stored in self.base_events)

reset_random_seed()
Reset random seed to defined value (stored in self.seed, set with self.set_random_seed)

set_parameter_statistics(*param_stats*)
Define parameter statistics for uncertainty simulation and sensitivity analysis

param_stats = list : list with relevant statistics defined for event parameters list is organised as: param_stats[event_id][parameter_name][stats_type] = value

Example: param_stats[2]["Dip"]["min"] = 200.

Possible statistics are:

- min = float : minimum bound
- max = float : maximum bound
- type = 'normal', 'uniform' : distribution type
- stdev = float : standard deviation (if normal distribution)

set_random_seed(*random_seed*)
Set random seed for reproducible experiments

Arguments:

- *random_seed* = int (or array-like) : define seed

set_up_to_date()
Set boolean variable for valid object

shuffle_event_order(*event_ids*)
Randomly shuffle order of events

Arguments:

- *event_ids* = [list of event ids] : event ids to be randomly shuffeled

update()
Update model computation

write_parameter_changes(*filepath*)

5.2.2 MonteCarlo class

This class provides the basic functionality to perform MonteCarlo error propagation experiments with Noddy. Created on Monday Jul 13 10:09:55 2015

@author: Sam Thiele

class pynoddy.experiment.MonteCarlo.**MonteCarlo**(*history*, *parameters*, *base_name='out'*)
Bases: *pynoddy.experiment.Experiment*

Perform Monte Carlo simulations on a model using defined input statistics

static clean(*path*, *basename=None*, ***kwds*)

Deletes files and folders created during Monte Carlo simulations

Arguments

- *path* = The directory to search. Subdirectories are included in the search.
- *basename* = The basename of files to delete. If left as None (default) all files are deleted.

Optional Kewords

- *delete_noddy_working_files* = If True, noddy working files are deleled. Default is True.
- *delete_noddy_history_files* = If True, noddy history files are deleted. Default is True.
- *delete_topology_files* = If True, topology files are deleted. Default is True.

cleanup(***kwds*)
Deletes files and folders created during Monte Carlo simulations

Optional Kewords

- *delete_noddy_working_files* = If True, noddy working files are deleled. Default is True.
- *delete_noddy_history_files* = If True, noddy history files are deleted. Default is True.
- *delete_topology_files* = If True, topology files are deleted. Default is True.

generate_model_instances(*path*, *count*, ***kwds*)
Generates the specified of randomly varied Noddy models.

Arguments:

- *path* = The directory that Noddy models should be generated in
- *count* = The number of random variations to generate

Optional Kewords:

- ***threads* = The number of seperate threads to run when generating noddy models. Note that RAM is** often a limiting factor (at this point every thread requires at least ~1Gb of ram).

- ***sim_type* = The type of simulation to run. This can be any of: 'BLOCK', 'GEOPHYSICS', 'SURFACES',** 'BLOCK_GEOPHYS', 'TOPOLOGY', 'BLOCK_SURFACES', 'ALL'. Default is 'BLOCK'.

- ***write_changes* = A file (path) to write the parameters used in each model realisation to (minus the extension).** The default is None (no file written).
- *verbose* = True if this function sends output to the print buffer. Default is True.
- *seed* = The random seed to use in this experiment. If not specified, threads are seeded with PID * TID * time (*nodeID).

static `generate_models_from_existing_histories` (*path*, ***kwds*)

Processes all existing his files in the given directory

Arguments:

- *path* = The directory that will be searched for .his files

Optional Kewords:

- ***threads* = The number of seperate threads to run when generating noddy models. For optimum** performance this should equal the number of logical cores - 1, unless RAM is a limiting factor (at this point every thread requires at least 2Gb of ram).
- ***sim_type* = The type of simulation to run. This can be any of: 'BLOCK', 'GEOPHYSICS', 'SURFACES',** 'BLOCK_GEOPHYS', 'TOPOLOGY', 'BLOCK_SURFACES', 'ALL'. Default is 'BLOCK'.
- ***force_recalculate* = Forces the recalculation of existing noddy files. Default is False, hence this** function will not run history files that are already associated with Noddy data files.
- *verbose* = True if this function sends output to the print buffer. Default is True.

static `load_noddy_realisations` (*path*, ***args*)

Loads all model realisations and returns them as an array of NoddyOutput objects

Arguments:

- ***path* = The root directory that models should be loaded from. All models with the same base_name** as this class will be loaded (including subdirectoriess)

Optional Arguments:

- *verbose* = True if this function should write debug information to the print buffer. Default is True.

Returns:

- a list of NoddyOutput objects

static `load_topology_realisations` (*path*, ***args*)

Loads all model topology realisations and returns them as an array of NoddyTopology objects

Arguments:

- ***path* = The root directory that models should be loaded from. All models with the same base_name** as this class will be loaded (including subdirectoriess)

Optional Arguments:

- ***load_attributes* = True if nodes and edges in the topology network should be attributed with properties such a** and surface area and lithology colour. Default is True.
- *verbose* = True if this function should write debug information to the print buffer. Default is True.

Returns:

- a list of NoddyTopology objects

5.2.3 SensitivityAnalysis class

class pynoddy.experiment.SensitivityAnalysis.**SensitivityAnalysis**(*history=None*, ***kwds*)

Bases: *pynoddy.experiment.Experiment*

Sensitivity analysis experiments for kinematic models

Sensitivity analysis with methods from the SALib package: https://github.com/jdherman/SALib

add_sampling_line(*x*, *y*, ***kwds*)

Define a vertical sampling line, for example as a drillhole at position (x,y)

As default, the entire length for the model extent is exported. Ohter depth ranges can be defined with optional keywords.

Arguments:

- *x* = float: x-position of drillhole
- *y* = float: y-position of drillhole

Optional keywords:

- *z_min* = float : minimum z-value (default: model origin)
- *z_max* = float : maximum z-value (default: surface)
- *label* = string : add a label to line (e.g. drillhole name, location, etc.)

create_params_file(***kwds*)

Create params file from defined parameter statistics for SALib analysis

Note: parameter statistics have to be defined in self.param_stats dictionary (use self.set_parameter_statistics)

Optional keywords:

- *filename* = string : name of parameter file (default: params_file_tmp.txt)

determine_distances(***kwds*)

Determine distances for a given parameter sets, based on defined sampling lines

Optional keywords:

- *param_values* = list of parameter values (as, for example, created by SALib methods)
- *resolution* = float : model resolution to calculate distance at sampling lines

distance(***kwds*)

Calculate distance between current state and base model

The standard distance is calculated as L1 norm of relative stratigraphic difference along sampling lines.

Optional keywords:

- *norm* = 'L1', 'L2' : norm to calculate distance
- *resolution* = float : model resolution to calculate distance at sampling lines

get_model_lines(***kwds*)

Get base model along the defined sampling lines

Optional keywords:

- *model_type* = 'base', 'current' : model type (select base to get freezed model)

- *resolution* = float : model resolution to calculate distance at sampling lines

perform_analsis(*n=10*, ***kwds*)
Perform Sobol sensitivity analysis with SALib methods

Arguments:

- *n* = int : number of sobol iterations (default: 10)

Optional keywords:

- *calc_second_order* = bool : second order stats (default: True)

plot_distances(***kwds*)
Create diagnostic plot of calculated distances

Optional keywords:

- *savefig* = bool : save figure to file (default: show)
- *fig_filename* = string : figure filename (default: distances.png)

plot_sensitivity_matrix(***kwds*)
Create a plot of the sensitivity matrix

Optional keywords:

- *savefig* = bool : save figure to file (default: show)
- *fig_filename* = string : figure filename (default: distances.png)

p