pynoddy Documentation

Release

Florian Wellmann

March 29, 2014

CONTENTS

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 priviledges 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!!

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.

These notebooks are also included in this documentation as non-interactive versions.

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

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] 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 sedimentary pile
4. Finally, add a sequence of late faults affecting the entire system.

The result could look something like this:

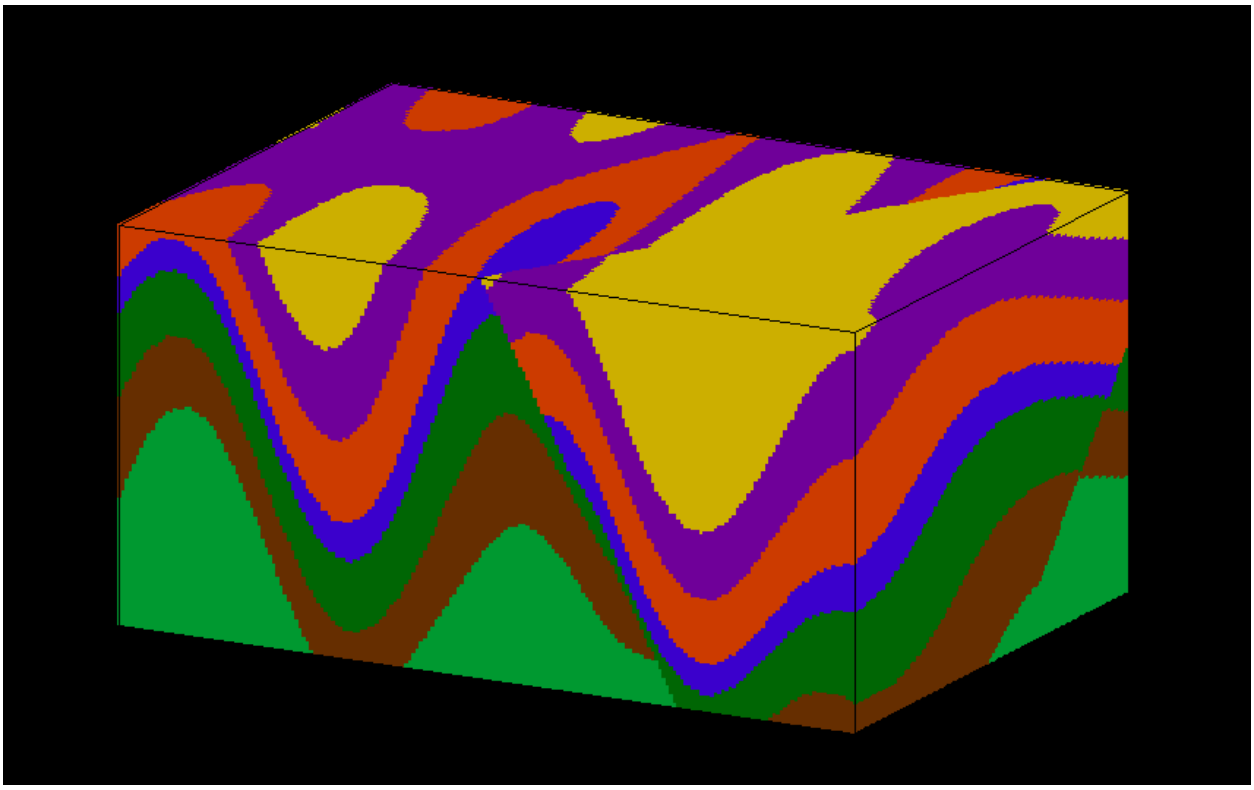
The software runs on Windows only, but the source files (written in C) are available for download to generate a command line version of the modelling step alone:

https://github.com/markjessell/functionNoddy

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

1.8 References

[1] 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:

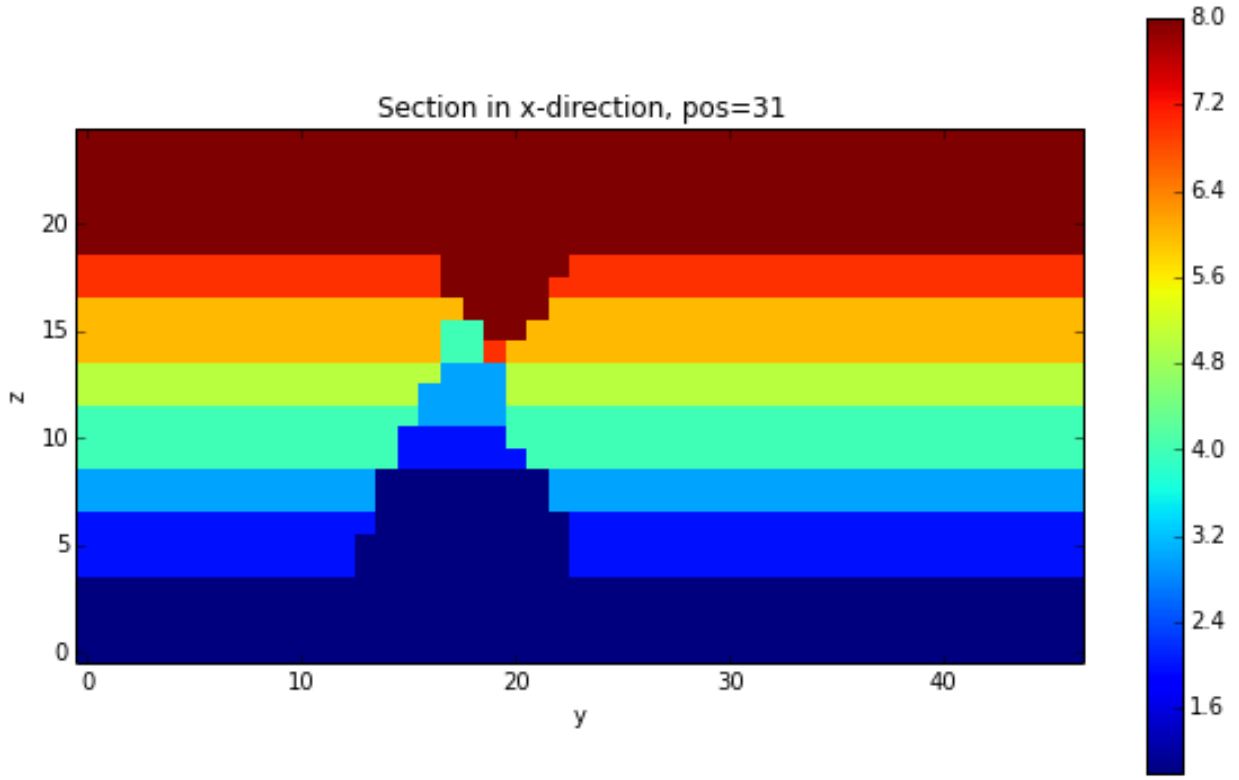
```
print("The model has an extent of %.0f m in x-direction, with %d cells of width %.0f m" %
      (N1.extent_x, N1.nx, N1.delx))
```

```
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)
```

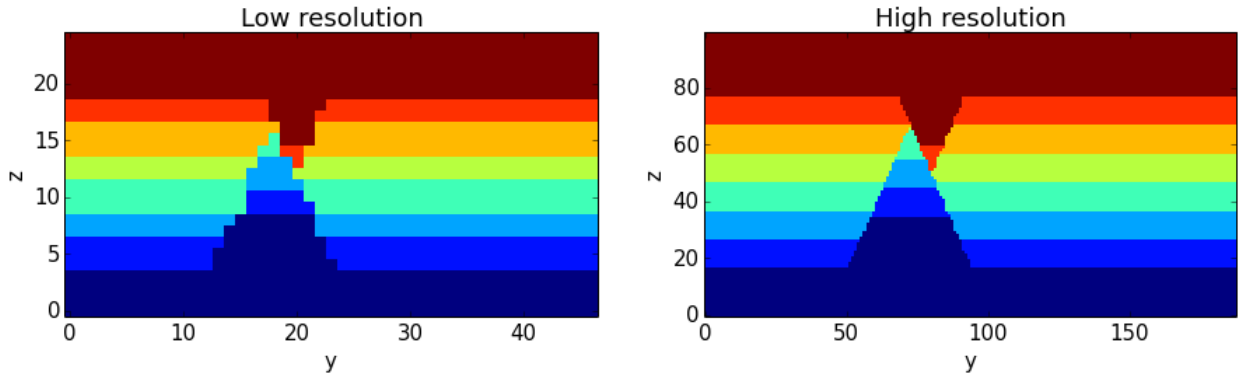
The different cell sizes are also represented in the output files:

```
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))
```

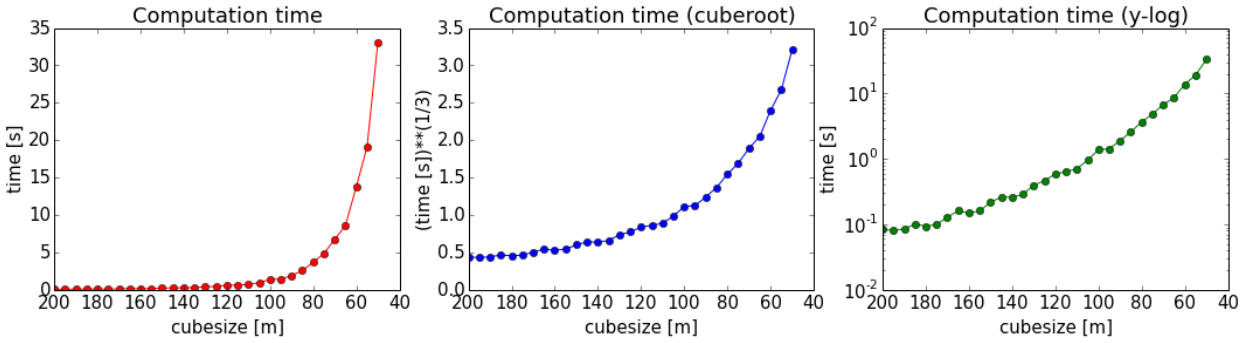
```
Simulation time for low-resolution model:  0.08 seconds
Simulation time for high-resolution model: 32.43 seconds
```

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
```

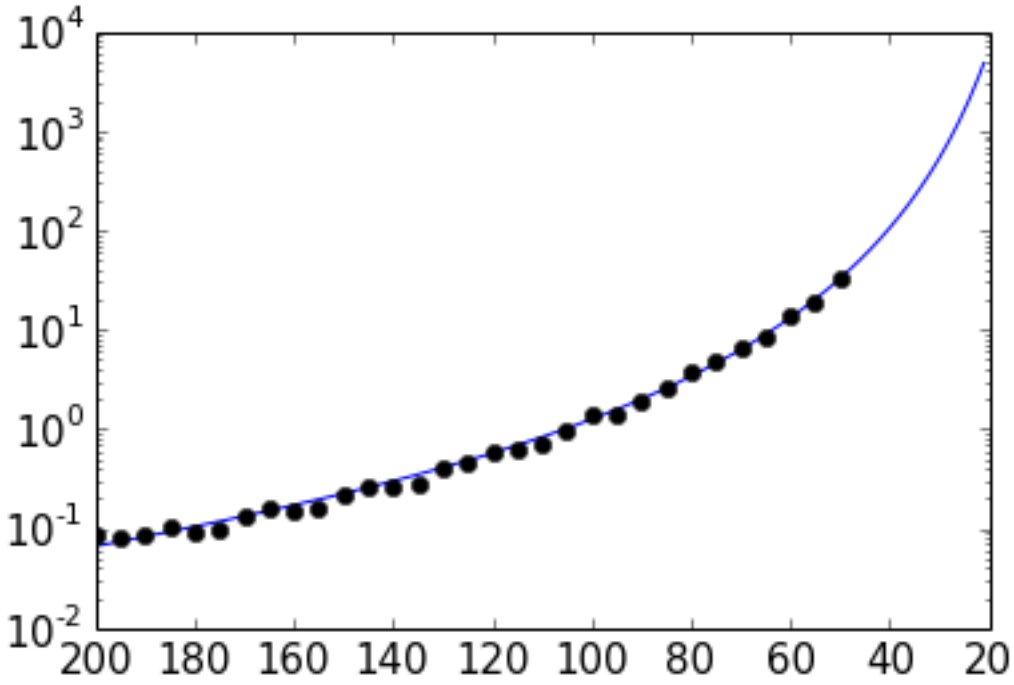
```
array([ -1.15814515e-02,   4.94030524e-01,   1.99015465e+01])
```

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:

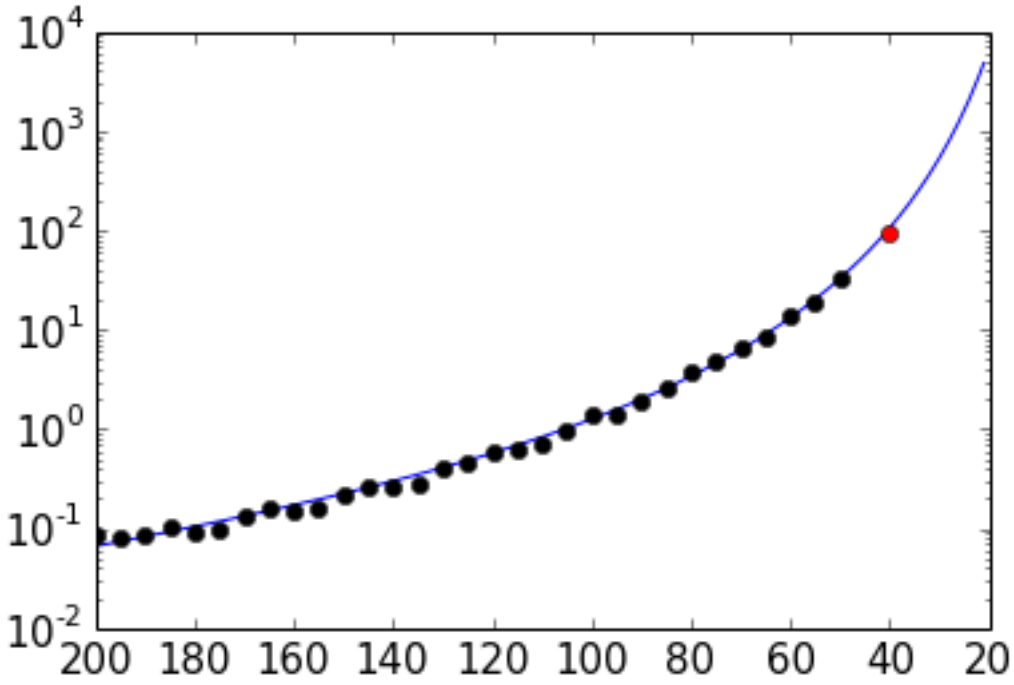
```
NH1.change_cube_size(cube_size)
NH1.write_history(tmp_history)
start_time = time.time()
pynoddy.compute_model(tmp_history, tmp_output)
end_time = time.time()
time_comp = end_time - start_time

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...).

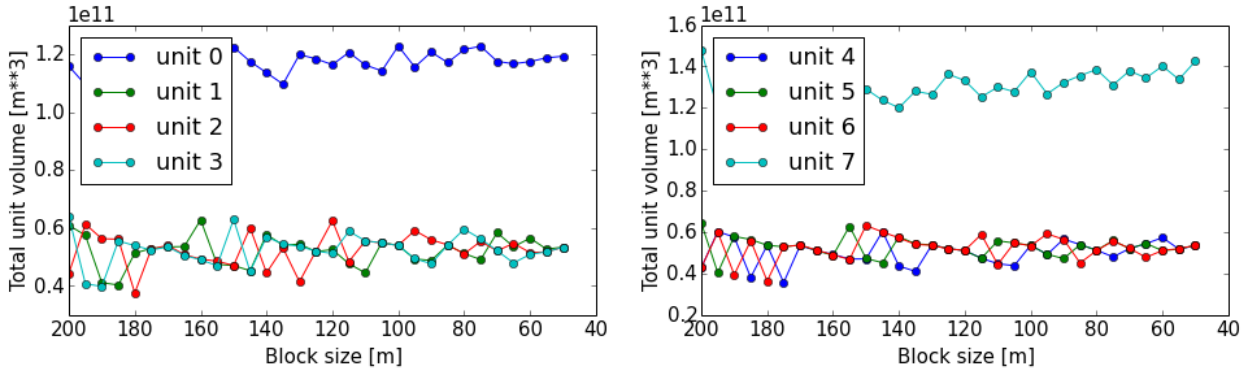
Anyway, if we want to use the model in a scientific study, for example to evaluate volume of specific units, or to estimate the geological topology (Mark is working on this topic with some cool ideas - example to be implemented here, "soon"), we want to know if the resolution of the model is actually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

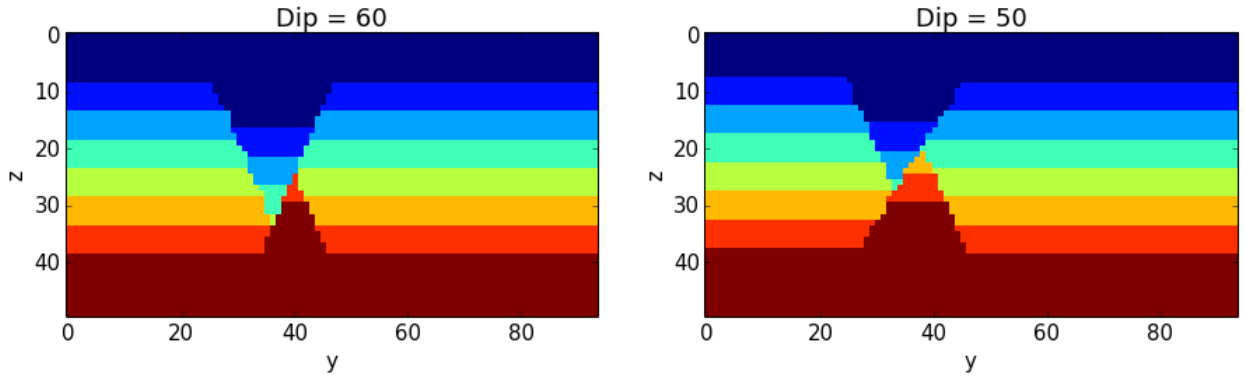
```
reload(pynoddy.history)
reload(pynoddy.events)
H1 = pynoddy.history.NoddyHistory(history)
# get the original dip of the fault
dip_ori = H1.events[3].properties['Dip']
# dip_ori1 = H1.events[2].properties['Dip']
# add 10 degrees to dip
add_dip = -10
dip_new = dip_ori + add_dip
# dip_new1 = dip_ori1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

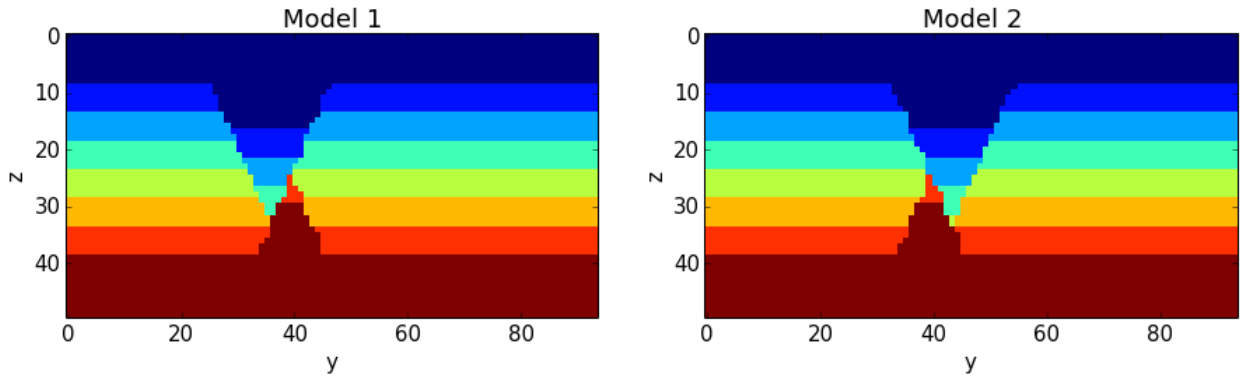
The first method is probably the most useful to test how a simple change in the order of events will effect the final geological model. We will use it here with our example to test how the model would change if the timing of the faults is swapped.

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.

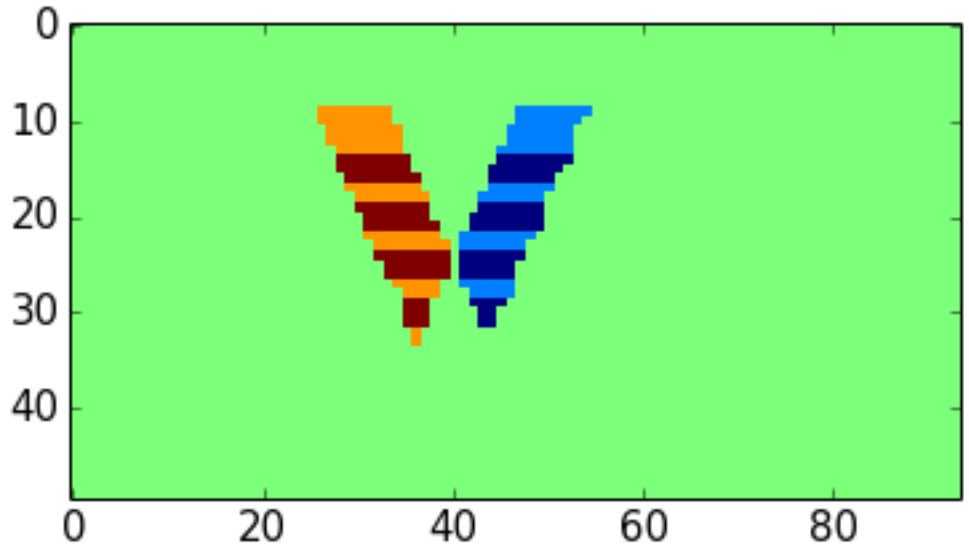
The geology data is stored in the `NoddyOutput.block` attribute. To evaluate the difference between two models, we can therefore simply compute:

```
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

STOCHASTIC EVENTS

The parameters defining the geological events are, by their very nature, highly uncertain. In addition to these uncertainties, the kinematic approach by itself is only a limiting approximation. The picture that we obtain from the kinematic forword model is therefore a very overconfident repsresentation of reality - an aspect that (hopefully) everyone using Noddy is aware of...

In order to respect the vast nature of these uncertainties, we introduce here an adapted version of the standard geological events defined in Noddy: the definition of a stochastic event:

A stochastic event is geological event in the Noddy history that has an uncertainty associated to it in such a way that, recomputing the geolgoical history will result in a different result each time (note: the definition is borrowed from the notion of stochastic events in probabilistic programming, see for example pymc).

5.1 Definition of stochastic events

We start, as before, with a pre-defined geological noddy history for simplicity:

CHAPTER

SIX

PYNODDY PACKAGE

6.1 Submodules

6.2 pynoddy.history module

Noddy history file wrapper Created on 24/03/2014

@author: Florian Wellmann

class pynoddy.history.**NoddyHistory**(*history*)

Class container for Noddy history files

change_cube_size(*cube_size*)

Change the model cube size (isotropic)

Arguments:

- *cube_size* = float : new model cube size

determine_events()

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!

---

load_history(*history*)

Load Noddy history

Arguments:

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

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 ;-)

6.3 pynoddy.output module

Noddy output file analysis Created on 24/03/2014

@author: Florian Wellmann

class pynoddy.output.**NoddyOutput**(*output_name*)

Class definition for Noddy output analysis

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)

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)
- *title* = string : plot title
- *savefig* = bool : save figure to file (default: show directly on screen)
- *cmap* = matplotlib.cmap : colormap
- *fig_filename* = string : figure filename

6.4 Module contents

Package initialization file for pynoddy

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

CHAPTER

SEVEN

INDICES AND TABLES

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

p