pynoddy Documentation

Release

Florian Wellmann, Sam Thiele

October 11, 2015

CONTENTS

1 pynoddy **3**

1.1 How does it work? 3
1.2 Installation 3
1.3 Documentation 3
1.4 Tutorial 3
1.5 Dependencies 3
1.6 License 4
1.7 What is Noddy? 4
1.8 References 4

2 Simulation of a Noddy history and visualisation of output **5**

2.1 Compute the model 5
2.2 Loading Noddy output files 6
2.3 Plotting sections through the model 6
2.4 Export model to VTK 6

3 Change Noddy input file and recompute model **9**

3.1 Get basic information on the model 10
3.2 Change model cube size and recompute model 10
3.3 Estimating computation time for a high-resolution model 11
3.4 Simple convergence study 14

4 Geological events in pynoddy: organisation and adpatiation **17**

4.1 Loading events from a Noddy history 17
4.2 Changing aspects of geological events 18
4.3 Changing the order of geological events 19
4.4 Determining the stratigraphic difference between two models 20

5 Creating a model from scratch **23**

5.1 Defining a stratigraphy 23
5.2 Add a fault event 24
5.3 Complete Model Set-up 26

6 Read and Visualise Geophysical Potential-Fields **29**

6.1 Read history file from Virtual Explorer 29
6.2 Visualise calculated geophysical fields 31
6.3 Change history and compare gravity 31
6.4 Figure with all results 36

7 Reproducible Experiments with pynoddy **39**

7.1 Defining an experiment 39
7.2 Setting up the experiment class 40
7.3 Loading an example model from the Virtual Explorer Atlas 40

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

This example shows how the module pynoddy.history can be used to compute the model, and how simple visualisations can be generated with pynoddy.output.

```
from IPython.core.display import HTML
css_file = 'pynoddy.css'
HTML(open(css_file, "r").read())
```

```
%matplotlib inline
```

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

# Now import pynoddy
import pynoddy
reload(pynoddy)
import pynoddy.output
import pynoddy.history

# 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!!
print subprocess.Popen(['noddy.exe', history, output_name, 'BLOCK'],
                       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 substituted 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.output.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 124 cells of width 100 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('y', figsize = (5,3))
```

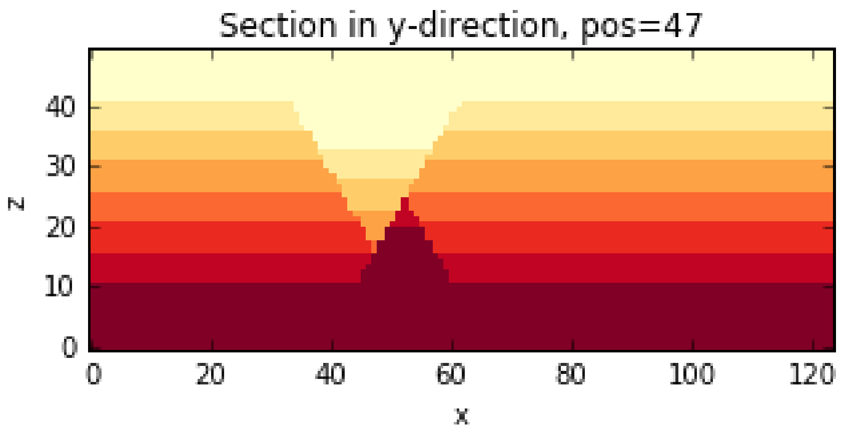

Fig. 2.1: png

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()
```

The exported VTK file can be visualised in any VTK viewer, for example in the (free) viewer Paraview (www.paraview.org). An example visualisation of the model in 3-D is presented in the figure below.

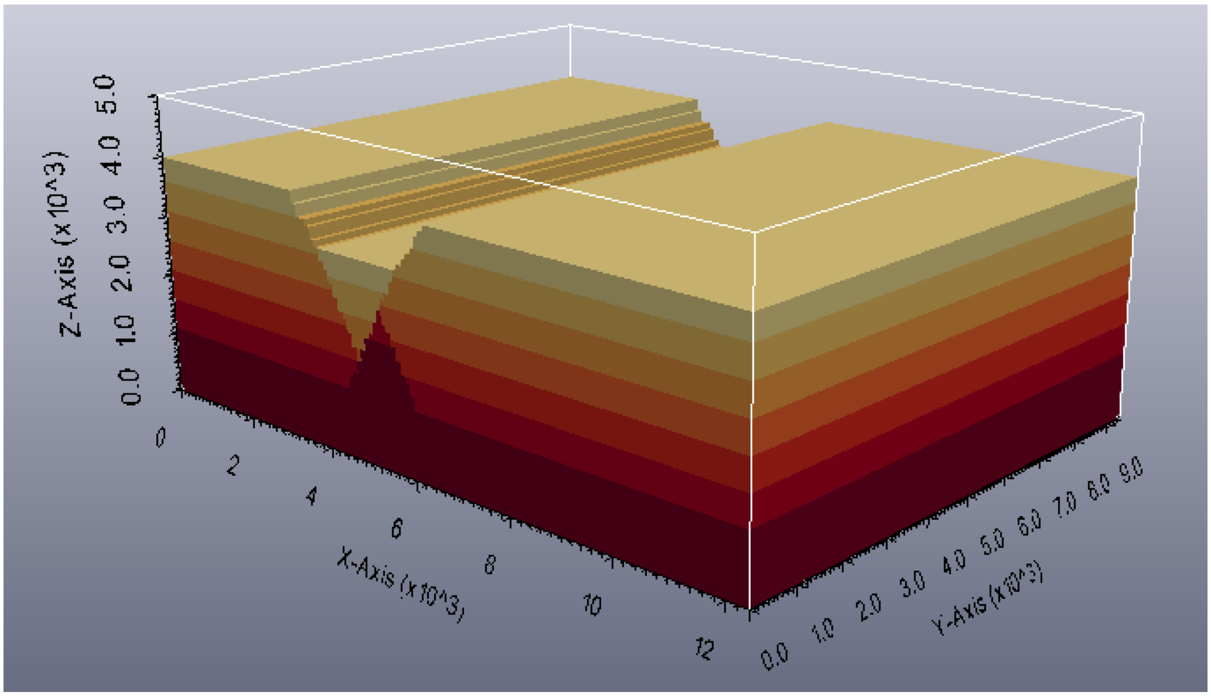

Fig. 2.2: 3-D Visualisation generated with Paraview (top layer transparent)

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.

```
from IPython.core.display import HTML
css_file = 'pynoddy.css'
HTML(open(css_file, "r").read())
```

```
cd ../docs/notebooks/
```

```
/Users/flow/git/pynoddy/docs/notebooks
```

```
%matplotlib inline
```

```
import sys, os
import matplotlib.pyplot as plt
import numpy as np
# adjust some settings for matplotlib
from matplotlib import rcParams
# print rcParams
rcParams['font.size'] = 15
# determine path of repository to set paths corretly below
repo_path = os.path.realpath('../..')
import pynoddy
import pynoddy.history
import pynoddy.output
```

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 at 0x103ac2a50>,
 2: <pynoddy.events.Fault at 0x103ac2a90>,
 3: <pynoddy.events.Fault at 0x103ac2ad0>}
```

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,
 '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
NH1 = pynoddy.history.NoddyHistory(history)
pynoddy.compute_model(history, output_name)
NO1 = pynoddy.output.NoddyOutput(output_name)
```

```
# 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)
```

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  582800 cells with a blocksize 100 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('y', position=0, ax = ax1, colorbar=False, title="Low resolution")
NO2.plot_section('y', position=1, ax = ax2, colorbar=False, title="High resolution")

plt.show()
```

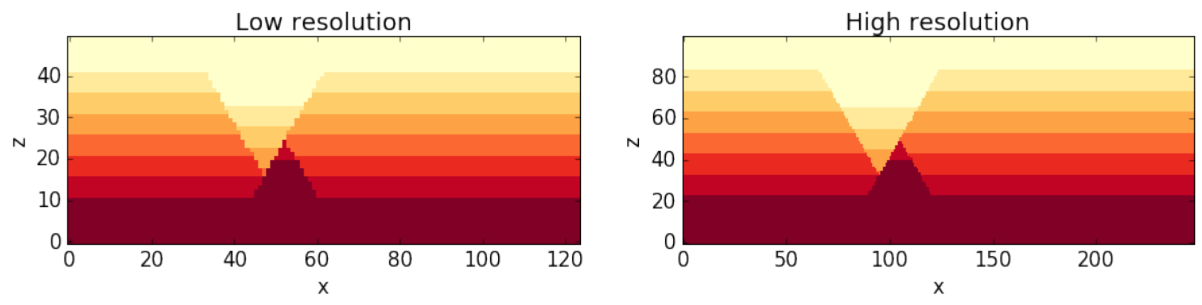

Fig. 3.1: png

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:  0.73 seconds
Simulation time for high-resolution model:  5.78 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

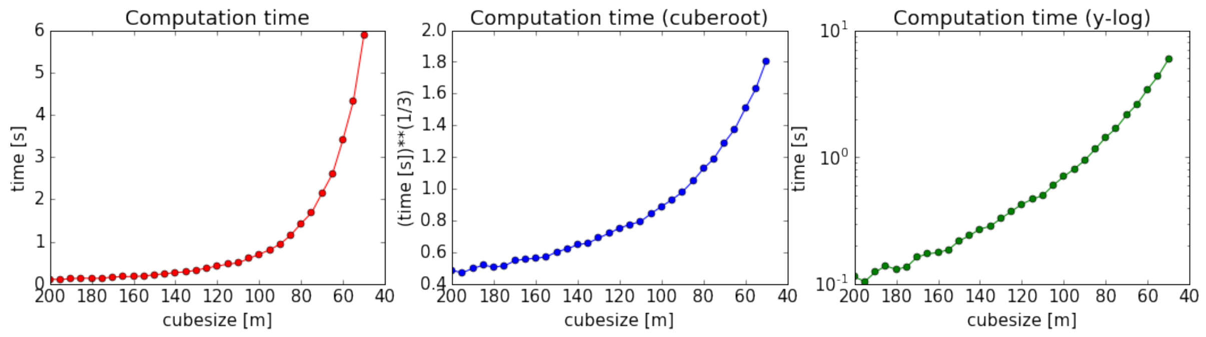

Fig. 3.2: png

exponent (as the line is obviously not straight), so something like:

$$f(x) = a + (b \log_{10}(x))^{-c}$$

Let's try to fit the curve with `scipy.optimize.curve_fit`:

```
# 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), p0 = [-1, 0.5, 2])
popt
```

```
array([ -0.05618538,   0.50990774,  12.45183398])
```

Interesting, it looks like Noody scales with something like:

$$f(x) = (0.5 \log_{10}(x))^{-12}$$

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: 12.4 seconds
```

Now let's check the actual simulation time:

```
NH1.change_cube_size(cube_size)
NH1.write_history(tmp_history)
start_time = time.time()
```

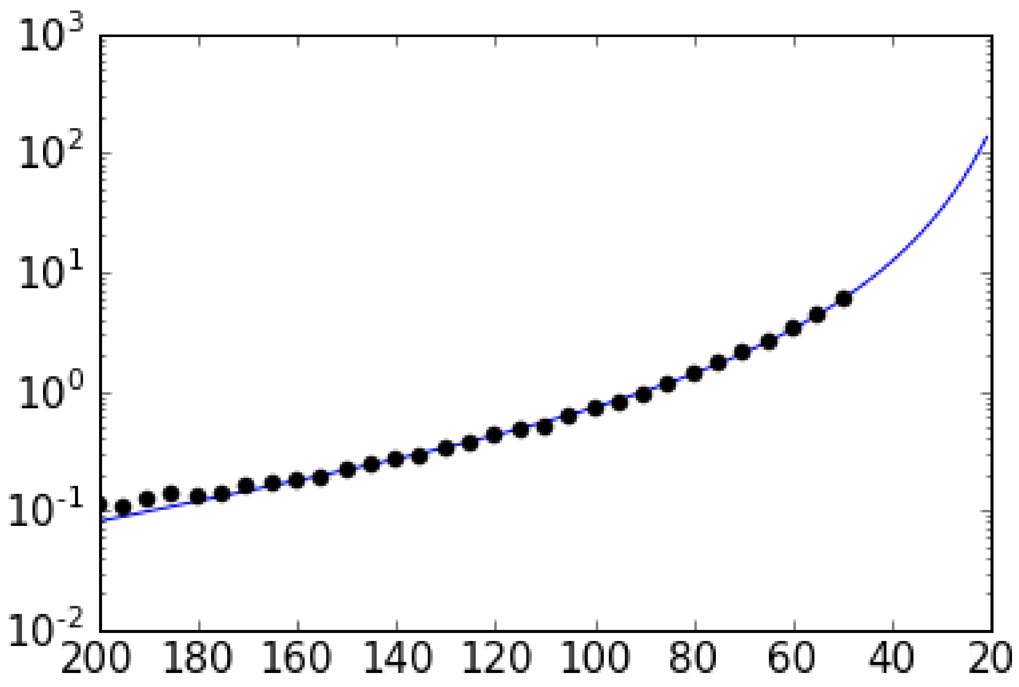

Fig. 3.3: png

```
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: 11.6 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

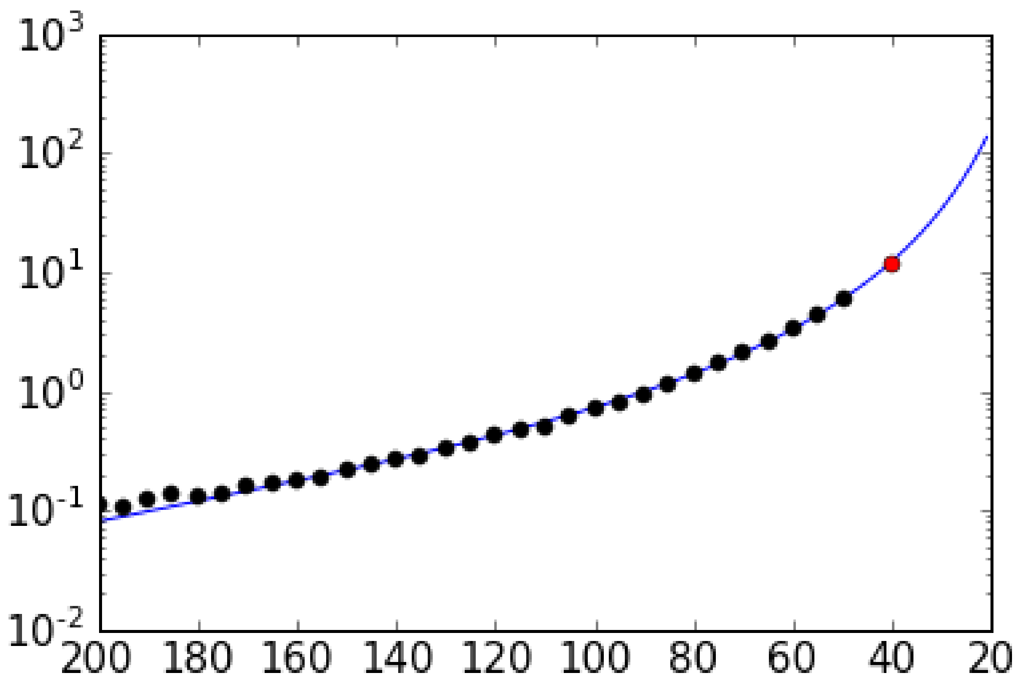

Fig. 3.4: png

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 0x107eb7250>
```

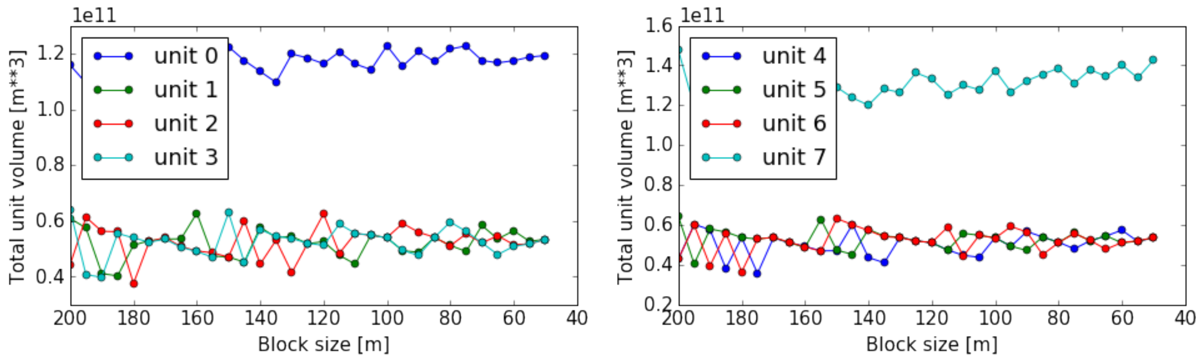

Fig. 3.5: png

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.

```
from IPython.core.display import HTML
css_file = 'pynoddy.css'
HTML(open(css_file, "r").read())
```

```
%matplotlib inline
```

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

import pynoddy
import pynoddy.history
import pynoddy.events
import pynoddy.output
reload(pynoddy)
```

```
<module 'pynoddy' from '/Users/flow/git/pynoddy/pynoddy/__init__.pyc'>
```

```
# 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)
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 at 0x10cf2b410>,
 2: <pynoddy.events.Fault at 0x10cf2b450>,
 3: <pynoddy.events.Fault at 0x10cf2b490>}
```

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,
 '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. Parameters can directly be updated in the properties dictionary:

```
H1 = pynoddy.history.NoddyHistory(history_ori)
# get the original dip of the fault
dip_ori = H1.events[3].properties['Dip']

# add 10 degrees to dip
add_dip = -10
dip_new = dip_ori + add_dip

# and assign back to properties dictionary:
H1.events[3].properties['Dip'] = dip_new
# H1.events[2].properties['Dip'] = dip_new1
```

```
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('y', position=0, ax = ax1, colorbar=False, title="Dip = %.0f" % dip_ori, savefig
NO2.plot_section('y', position=1, ax = ax2, colorbar=False, title="Dip = %.0f" % dip_new)
plt.show()
```

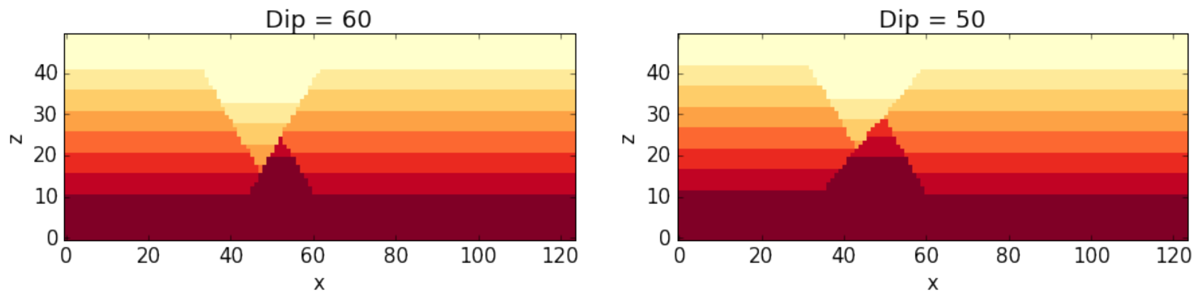

Fig. 4.1: png

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:

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

```
# 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
```

We now swap the position of two events in the kinematic history. For this purpose, a high-level function can directly be used:

```
# 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('y', ax = ax1, colorbar=False, title="Model 1")
NO2.plot_section('y', ax = ax2, colorbar=False, title="Model 2")

plt.show()
```

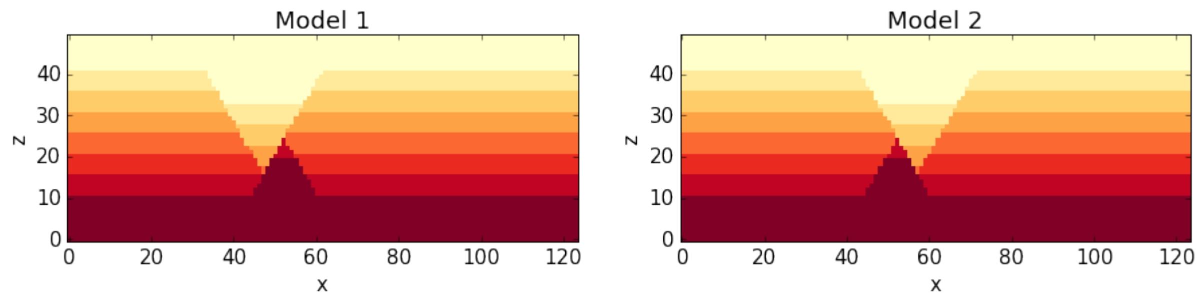

Fig. 4.2: png

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(figsize = (5,3))
ax = fig.add_subplot(111)
ax.imshow(diff[:,10,:].transpose(), interpolation='nearest',
          cmap = "RdBu", origin = 'lower left')
```

```
<matplotlib.image.AxesImage at 0x10cf3be10>
```

(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...)

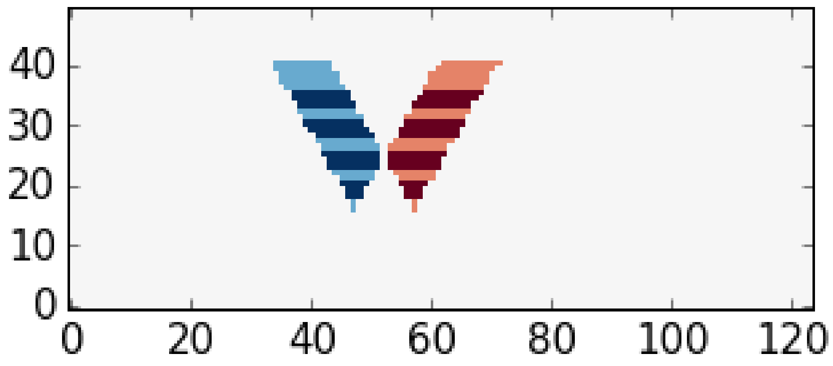

Fig. 4.3: png

Again, we may want to visualise results in 3-D. We can use the `export_to_vtk`-function as before, but now assing the data array to be exported as the calulcated differnce field:

```
NO1.export_to_vtk(vtk_filename = "model_diff", data = diff)
```

A 3-D view of the difference plot is presented below.

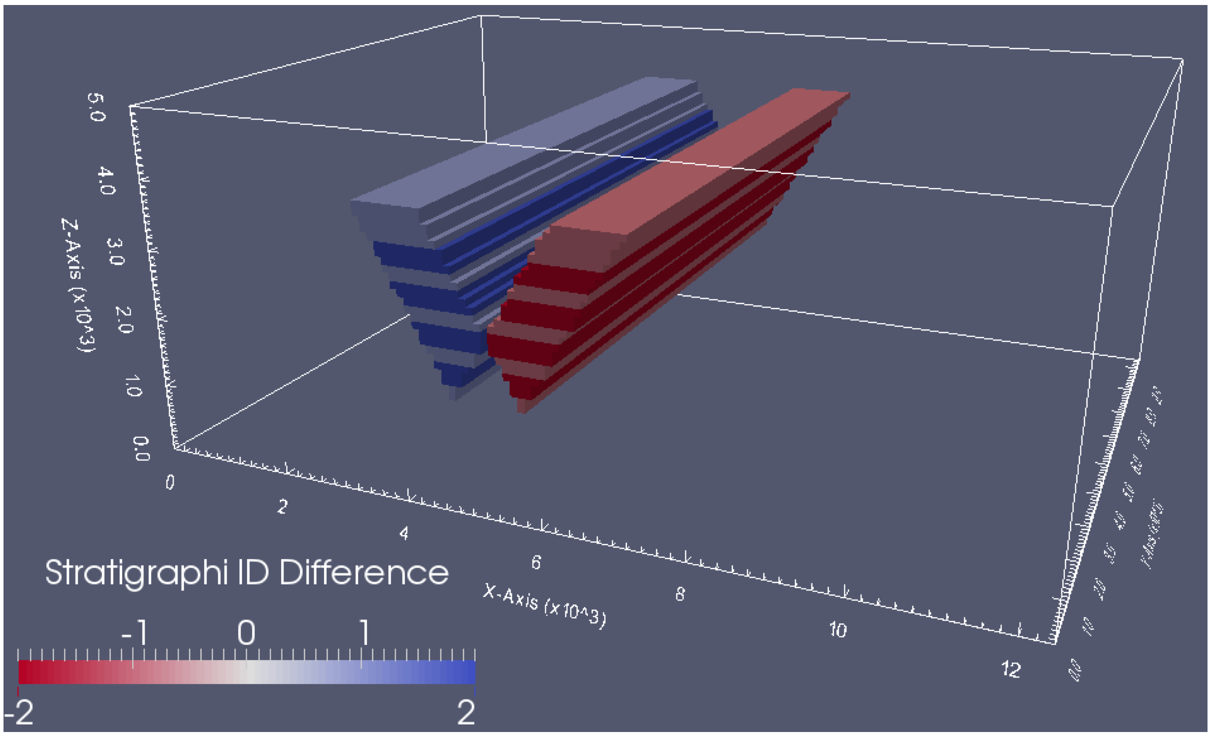

Fig. 4.4: 3-D visualisation of stratigraphic id difference

CHAPTER

FIVE

CREATING A MODEL FROM SCRATCH

We describe here how to generate a simple history file for computation with Noddy using the functionality of pynoddy. If possible, it is advisable to generate the history files with the Windows GUI for Noddy as this method provides, to date, a simpler and more complete interface to the entire functionality.

For completeness, pynoddy contains the functionality to generate simple models, for example to automate the model construction process, or to enable the model construction for users who are not running Windows. Some simple examlpes are shown in the following.

```
from matplotlib import rc_params
```

```
from IPython.core.display import HTML
css_file = 'pynoddy.css'
HTML(open(css_file, "r").read())
```

```
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
repo_path = os.path.realpath('../..')
import pynoddy.history
```

```
%matplotlib inline
```

```
rcParams.update({'font.size': 20})
```

5.1 Defining a stratigraphy

We start with the definition of a (base) stratigraphy for the model.

```
# Combined: model generation and output vis to test:
history = "simple_model.his"
output_name = "simple_out"
reload(pynoddy.history)
reload(pynoddy.events)

# create pynoddy object
nm = pynoddy.history.NoddyHistory()
# add stratigraphy
strati_options = {'num_layers' : 8,
                  'layer_names' : ['layer 1', 'layer 2', 'layer 3',
                                   'layer 4', 'layer 5', 'layer 6',
                                   'layer 7', 'layer 8'],
                  'layer_thickness' : [1500, 500, 500, 500, 500, 500, 500, 500]}
nm.add_event('stratigraphy', strati_options )
nm.write_history(history)
```

```
# Compute the model
reload(pynoddy)
pynoddy.compute_model(history, output_name)
```

```
''
```

```
# Plot output
import pynoddy.output
reload(pynoddy.output)
nout = pynoddy.output.NoddyOutput(output_name)
nout.plot_section('y', layer_labels = strati_options['layer_names'][::-1],
                  colorbar = True, title="",
                  savefig = False, fig_filename = "ex01_strati.eps")
```

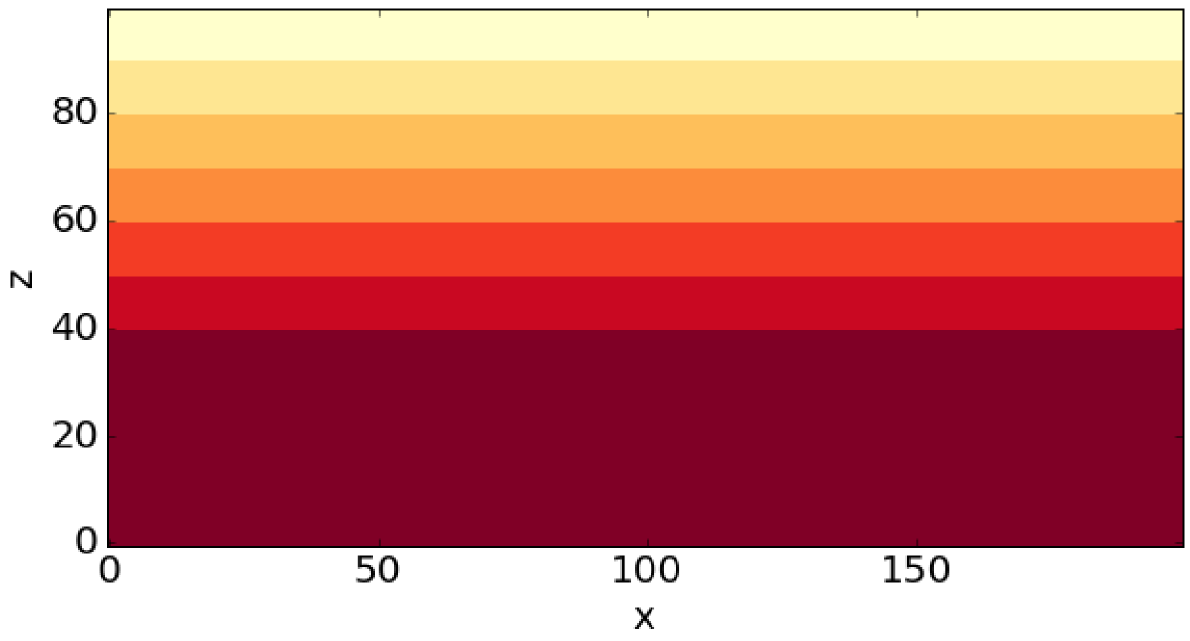

Fig. 5.1: png

5.2 Add a fault event

As a next step, let's now add the faults to the model.

```
reload(pynoddy.history)
reload(pynoddy.events)
nm = pynoddy.history.NoddyHistory()
# add stratigraphy
strati_options = {'num_layers' : 8,
                  'layer_names' : ['layer 1', 'layer 2', 'layer 3', 'layer 4', 'layer 5', 'layer
                  'layer_thickness' : [1500, 500, 500, 500, 500, 500, 500, 500]}
nm.add_event('stratigraphy', strati_options )




# The following options define the fault geometry:
fault_options = {'name' : 'Fault_E',
                 'pos' : (6000, 0, 5000),
                 'dip_dir' : 270,
                 'dip' : 60,
                 'slip' : 1000}

nm.add_event('fault', fault_options)
```

```
nm.events
```

```
{1: <pynoddy.events.Stratigraphy at 0x1073fc590>,
 2: <pynoddy.events.Fault at 0x107565fd0>}
```

```
nm.write_history(history)
```

```
# Compute the model
pynoddy.compute_model(history, output_name)
```

```
''
```

```
# Plot output
reload(pynoddy.output)
nout = pynoddy.output.NoddyOutput(output_name)
nout.plot_section('y', layer_labels = strati_options['layer_names'][::-1],
                  colorbar = True, title = "",
                  savefig = False, fig_filename = "ex01_fault_E.eps")
```

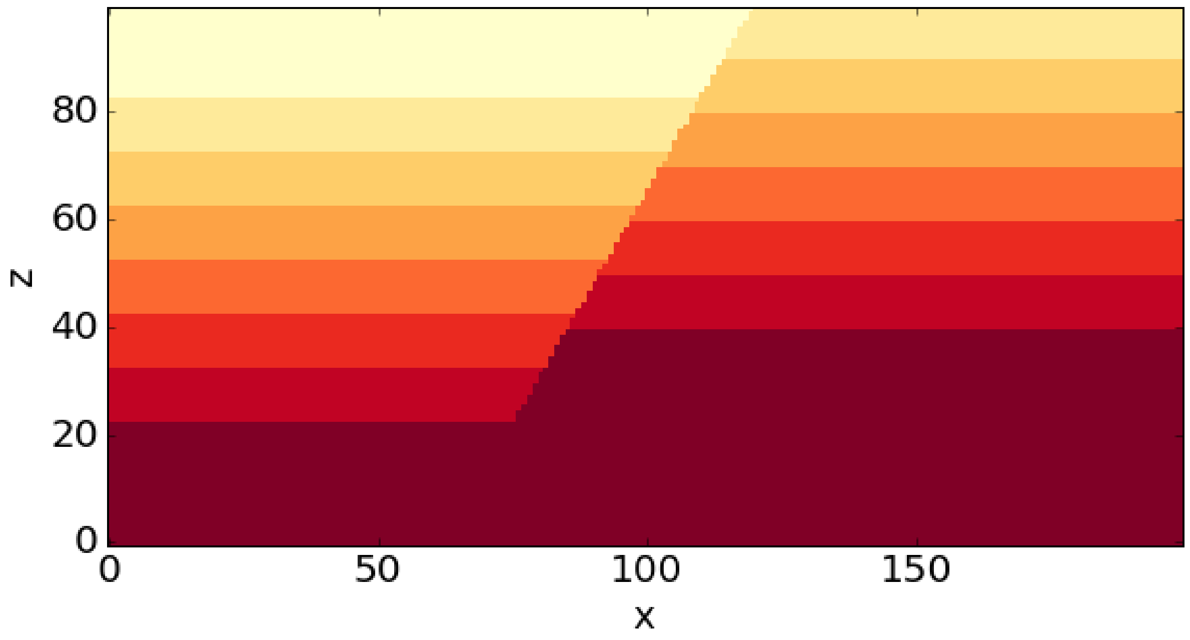

Fig. 5.2: png

```
# The following options define the fault geometry:
fault_options = {'name' : 'Fault_1',
                 'pos' : (5500, 3500, 0),
                 'dip_dir' : 270,
                 'dip' : 60,
                 'slip' : 1000}

nm.add_event('fault', fault_options)
nm.write_history(history)
```

```
# Compute the model
pynoddy.compute_model(history, output_name)
```

```
''
```

```
# Plot output
reload(pynoddy.output)
nout = pynoddy.output.NoddyOutput(output_name)
nout.plot_section('y', layer_labels = strati_options['layer_names'][::-1], colorbar = True)
```

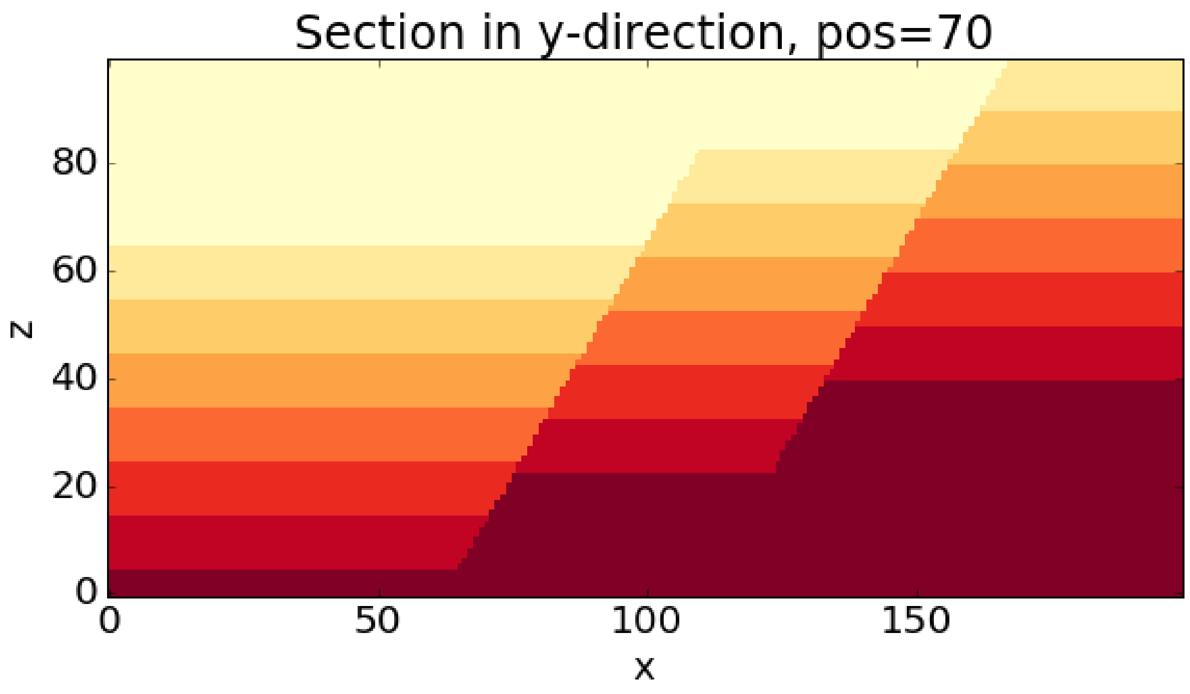

Fig. 5.3: png

```
nm1 = pynoddy.history.NoddyHistory(history)
```

```
nm1.get_extent()
```

```
(10000.0, 7000.0, 5000.0)
```

5.3 Complete Model Set-up

And here now, combining all the previous steps, the entire model set-up with base stratigraphy and two faults:

```
reload(pynoddy.history)
reload(pynoddy.events)
nm = pynoddy.history.NoddyHistory()
# add stratigraphy
strati_options = {'num_layers' : 8,
                  'layer_names' : ['layer 1', 'layer 2', 'layer 3',
                                   'layer 4', 'layer 5', 'layer 6',
                                   'layer 7', 'layer 8'],
                  'layer_thickness' : [1500, 500, 500, 500, 500,
                                       500, 500, 500]}
nm.add_event('stratigraphy', strati_options )
```

```
# The following options define the fault geometry:
fault_options = {'name' : 'Fault_W',
                 'pos' : (4000, 3500, 5000),
                 'dip_dir' : 90,
                 'dip' : 60,
                 'slip' : 1000}

nm.add_event('fault', fault_options)
# The following options define the fault geometry:
fault_options = {'name' : 'Fault_E',
                 'pos' : (6000, 3500, 5000),
                 'dip_dir' : 270,
                 'dip' : 60,
                 'slip' : 1000}

nm.add_event('fault', fault_options)
nm.write_history(history)
```

```
# Change cube size
nm1 = pynoddy.history.NoddyHistory(history)
nm1.change_cube_size(50)
nm1.write_history(history)
```

```
# Compute the model
pynoddy.compute_model(history, output_name)
```

```
''
```

```
# Plot output
reload(pynoddy.output)
nout = pynoddy.output.NoddyOutput(output_name)
nout.plot_section('y', layer_labels = strati_options['layer_names'][::-1],
                  colorbar = True, title="",
                  savefig = True, fig_filename = "ex01_faults_combined.eps",
                  cmap = 'YlOrRd') # note: YlOrRd colourmap should be suitable for colorblindness
```

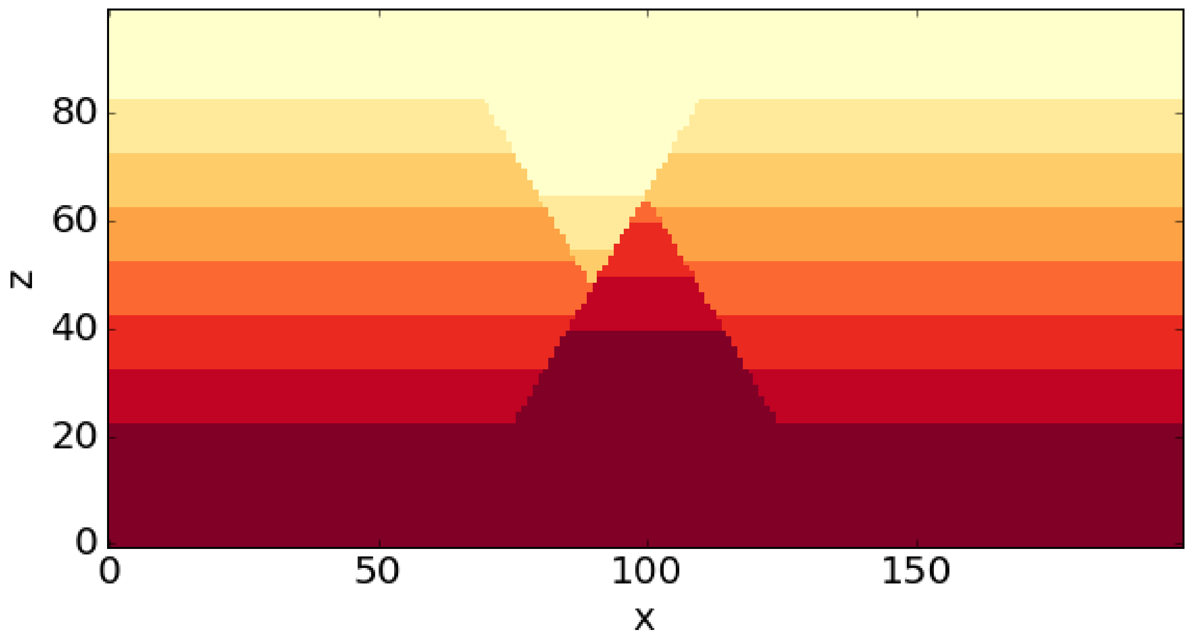

Fig. 5.4: png

CHAPTER

SIX

READ AND VISUALISE GEOPHYSICAL POTENTIAL-FIELDS

Geophysical potential fields (gravity and magnetics) can be calculated directly from the generated kinematic model. A wide range of options also exists to consider effects of geological events on the relevant rock properties. We will here use pynoddy to simply and quickly test the effect of changing geological structures on the calculated geophysical response.

```
%matplotlib inline
```

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

```
import matplotlib.pyplot as plt
import numpy as np
```

```
from IPython.core.display import HTML
css_file = 'pynoddy.css'
HTML(open(css_file, "r").read())
```

6.1 Read history file from Virtual Explorer

Many Noddy models are available on the site of the Virtual Explorer in the Structural Geophysics Atlas. We will download and use one of these models here as the base model.

We start with the history file of a “Fold and Thrust Belt” setting stored on:

`http://virtualexplorer.com.au/special/noddyatlas/ch3/ch3_5/his/nfold_thrust.his`

The file can directly be downloaded and opened with pynoddy:

```
import pynoddy.history
reload(pynoddy.history)

his = pynoddy.history.NoddyHistory(url = \
            "http://tectonique.net/asg/ch3/ch3_5/his/fold_thrust.his")

his.determine_model_stratigraphy()
```

```
his.change_cube_size(50)
```

```
# Save to (local) file to compute and visualise model
history_name = "fold_thrust.his"
```

```
his.write_history(history_name)
# his = pynoddy.history.NoddyHistory(history_name)
```

```
output = "fold_thrust_out"
pynoddy.compute_model(history_name, output)
```

```
''
```

```
import pynoddy.output
# load and visualise model
h_out = pynoddy.output.NoddyOutput(output)
```

```
# his.determine_model_stratigraphy()
h_out.plot_section('x',
                   layer_labels = his.model_stratigraphy,
                   colorbar_orientation = 'horizontal',
                   colorbar=False,
                   title = '',
#                   savefig=True, fig_filename = 'fold_thrust_NS_section.eps',
                   cmap = 'YlOrRd')
```

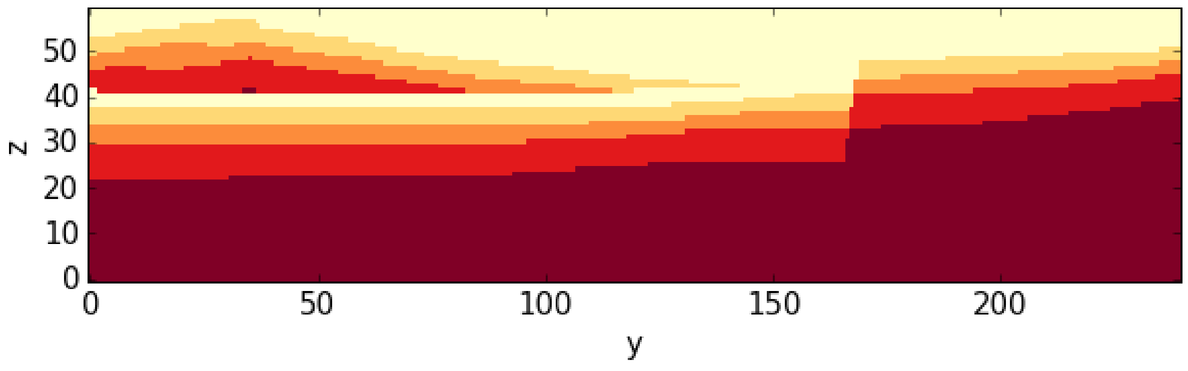

Fig. 6.1: png

```
h_out.plot_section('y', layer_labels = his.model_stratigraphy,
                   colorbar_orientation = 'horizontal', title = '', cmap = 'YlOrRd',
#                   savefig=True, fig_filename = 'fold_thrust_EW_section.eps',
                   ve=1.5)
```

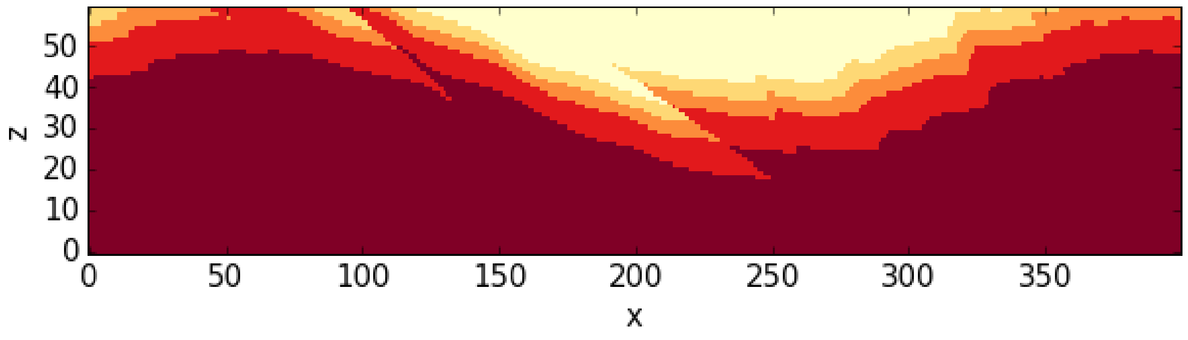

Fig. 6.2: png

```
h_out.export_to_vtk(vtk_filename = "fold_thrust")
```

6.2 Visualise calculated geophysical fields

The first step is to recompute the model with the generation of the geophysical responses

```
pynoddy.compute_model(history_name, output, sim_type = 'GEOPHYSICS')
```

```
''
```

We now get two files for the caluclated fields: '.grv' for gravity, and '.mag' for the magnetic field. We can extract the information of these files for visualisation and further processing in python:

```
reload(pynoddy.output)
geophys = pynoddy.output.NoddyGeophysics(output)
```

```
fig = plt.figure(figsize = (5,5))
ax = fig.add_subplot(111)
# imshow(geophys.grv_data, cmap = 'jet')
# define contour levels
levels = np.arange(322,344,1)
cf = ax.contourf(geophys.grv_data, levels, cmap = 'gray', vmin = 324, vmax = 342)
cbar = plt.colorbar(cf, orientation = 'horizontal')
# print levels
```

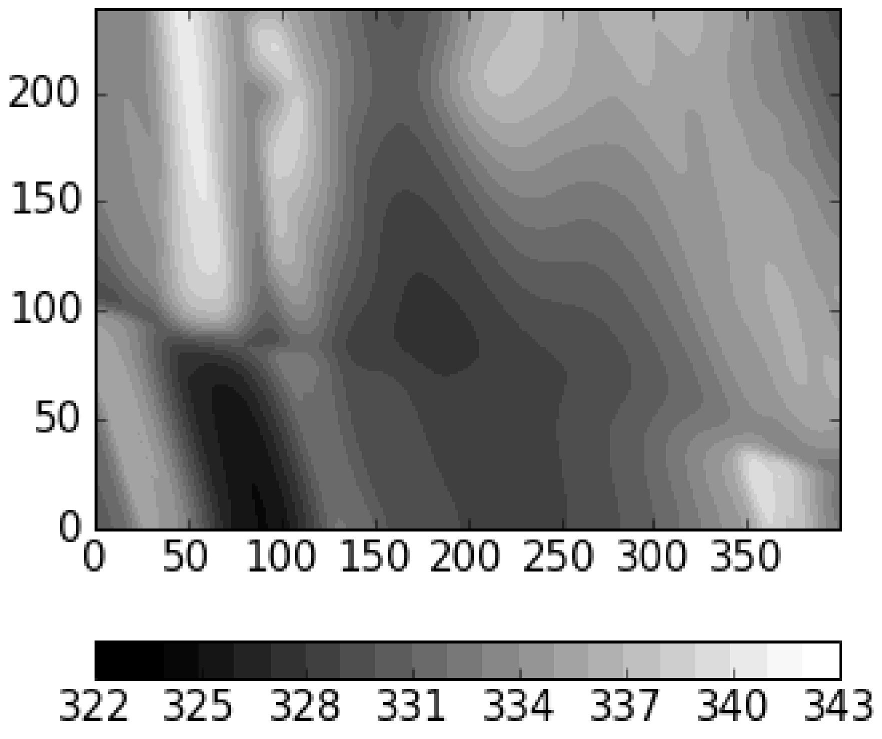

Fig. 6.3: png

6.3 Change history and compare gravity

As a next step, we will now change aspects of the geological history (paramtereised in as parameters of the kinematic events) and calculate the effect on the gravity. Then, we will compare the changed gravity field to the original field.

Let's have a look at the properties of the defined faults in the original model:

```
for i in range(4):
    print("\nEvent %d" % (i+2))
    print "Event type:\t" + his.events[i+2].event_type
    print "Fault slip:\t%.1f" % his.events[i+2].properties['Slip']
    print "Fault dip:\t%.1f" % his.events[i+2].properties['Dip']
    print "Dip direction:\t%.1f" % his.events[i+2].properties['Dip Direction']
```

```
Event 2
Event type: FAULT
Fault slip: -5000.0
Fault dip:  0.0
Dip direction:  90.0

Event 3
Event type: FAULT
Fault slip: -3000.0
Fault dip:  0.0
Dip direction:  90.0

Event 4
Event type: FAULT
Fault slip: -3000.0
Fault dip:  0.0
Dip direction:  90.0

Event 5
Event type: FAULT
Fault slip: 12000.0
Fault dip:  80.0
Dip direction:  170.0
```

```
reload(pynoddy.history)
reload(pynoddy.events)
his2 = pynoddy.history.NoddyHistory("fold_thrust.his")

print his2.events[6].properties
```

```
{'Dip': 130.0, 'Cylindricity': 0.0, 'Wavelength': 12000.0, 'Amplitude': 1000.0, 'Pitch': 0.0, 'Y'
```

As a simple test, we are changing the fault slip for all the faults and simply add 1000 m to all defined slips. In order to not mess up the original model, we are creating a copy of the history object first:

```
import copy
his = pynoddy.history.NoddyHistory(history_name)
his.all_events_end += 1
his_changed = copy.deepcopy(his)

# change parameters of kinematic events
slip_change = 2000.
wavelength_change = 2000.
# his_changed.events[3].properties['Slip'] += slip_change
# his_changed.events[5].properties['Slip'] += slip_change
# change fold wavelength
his_changed.events[6].properties['Wavelength'] += wavelength_change
his_changed.events[6].properties['X'] += wavelength_change/2.
```

We now write the adjusted history back to a new history file and then calculate the updated gravity field:

```
his_changed.write_history('fold_thrust_changed.his')
```

```
# %%timeit
# recompute block model
pynoddy.compute_model('fold_thrust_changed.his', 'fold_thrust_changed_out')
```

```
''
```

```
# %%timeit
# recompute geophysical response
pynoddy.compute_model('fold_thrust_changed.his', 'fold_thrust_changed_out',
                      sim_type = 'GEOPHYSICS')
```

```
''
```

```
# load changed block model
geo_changed = pynoddy.output.NoddyOutput('fold_thrust_changed_out')
# load output and visualise geophysical field
geophys_changed = pynoddy.output.NoddyGeophysics('fold_thrust_changed_out')
```

```
fig = plt.figure(figsize = (5,5))
ax = fig.add_subplot(111)
# imshow(geophys_changed.grv_data, cmap = 'jet')
cf = ax.contourf(geophys_changed.grv_data, levels, cmap = 'gray', vmin = 324, vmax = 342)
cbar = plt.colorbar(cf, orientation = 'horizontal')
```

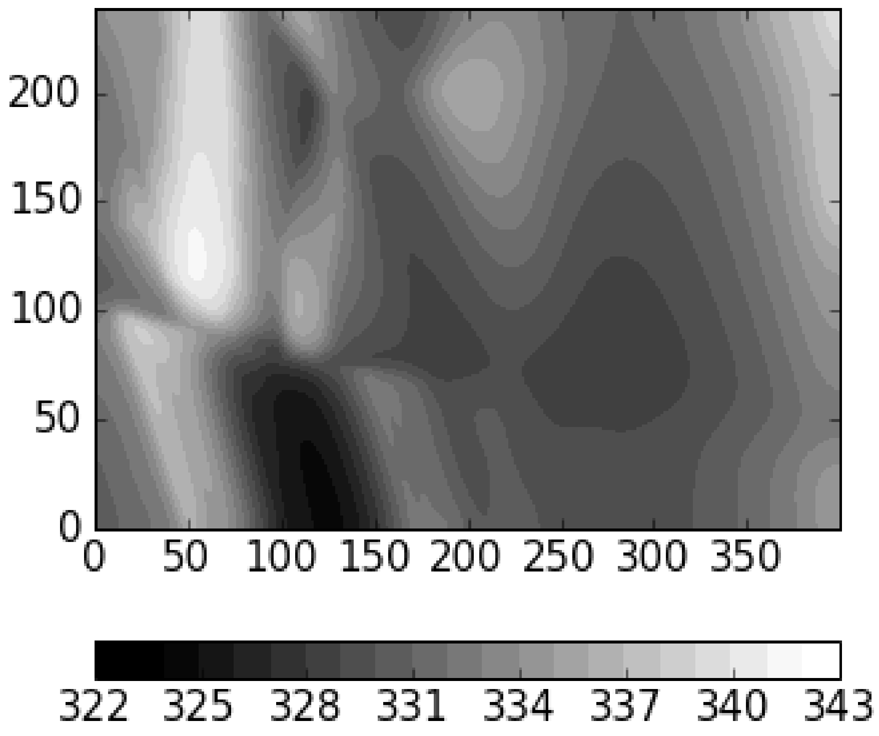

Fig. 6.4: png

```
fig = plt.figure(figsize = (5,5))
ax = fig.add_subplot(111)
# imshow(geophys.grv_data - geophys_changed.grv_data, cmap = 'jet')
maxval = np.ceil(np.max(np.abs(geophys.grv_data - geophys_changed.grv_data)))
# comp_levels = np.arange(-maxval,1.01 * maxval, 0.05 * maxval)
cf = ax.contourf(geophys.grv_data - geophys_changed.grv_data, 20,
                 cmap = 'spectral')
cbar = plt.colorbar(cf, orientation = 'horizontal')
```

```
# compare sections through model
geo_changed.plot_section('y', colorbar = False)
h_out.plot_section('y', colorbar = False)
```

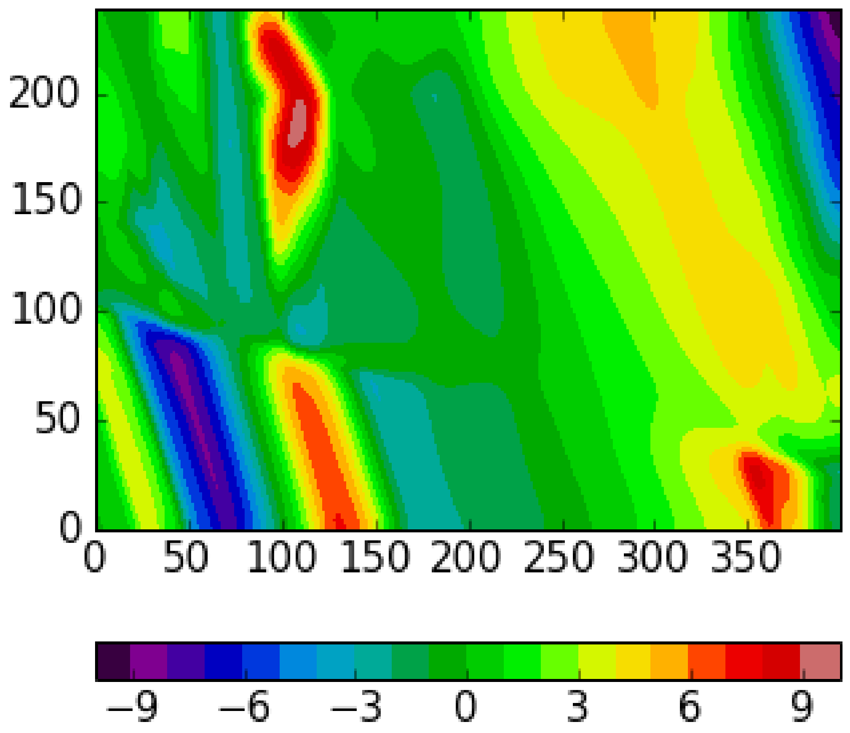

Fig. 6.5: png

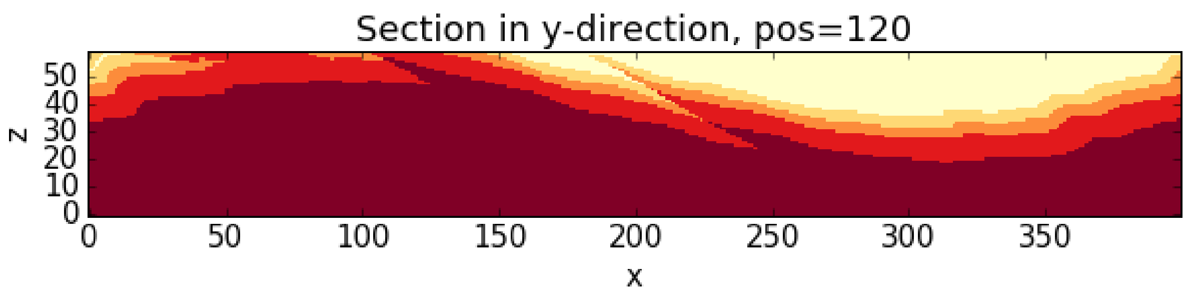

Fig. 6.6: png

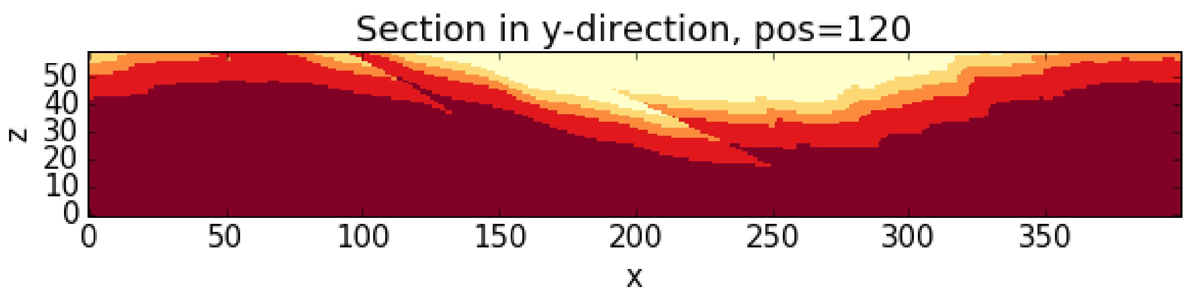

Fig. 6.7: png

```
for i in range(4):
    print("Event %d" % (i+2))
    print his.events[i+2].properties['Slip']
    print his.events[i+2].properties['Dip']
    print his.events[i+2].properties['Dip Direction']
```

```
Event 2
-5000.0
0.0
90.0
Event 3
-3000.0
0.0
90.0
Event 4
-3000.0
0.0
90.0
Event 5
12000.0
80.0
170.0
```

```
# recompute the geology blocks for comparison:
pynoddy.compute_model('fold_thrust_changed.his', 'fold_thrust_changed_out')
```

```
''
```

```
geology_changed = pynoddy.output.NoddyOutput('fold_thrust_changed_out')
```

```
geology_changed.plot_section('x',
#                     layer_labels = his.model_stratigraphy,
                      colorbar_orientation = 'horizontal',
                      colorbar=False,
                      title = '',
#                     savefig=True, fig_filename = 'fold_thrust_NS_section.eps',
                      cmap = 'YlOrRd')
```

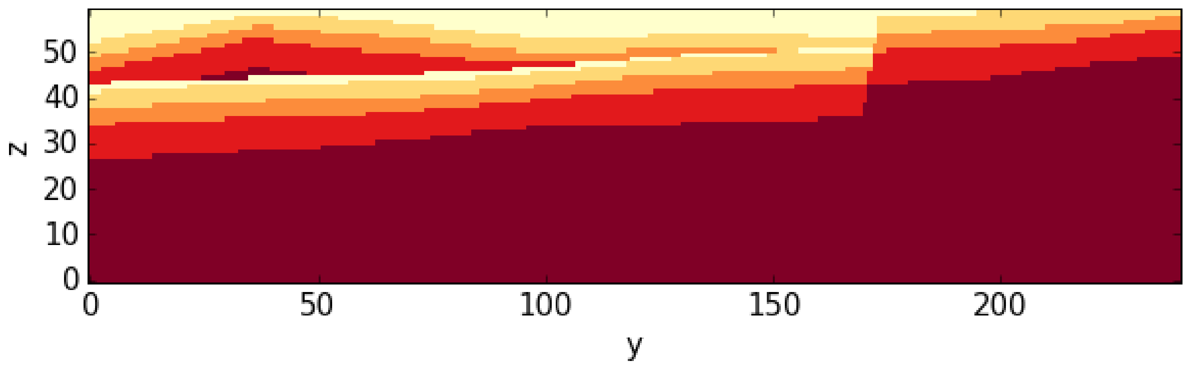

Fig. 6.8: png

```
geology_changed.plot_section('y',
                              # layer_labels = his.model_stratigraphy,
                      colorbar_orientation = 'horizontal', title = '', cmap = 'YlOrRd',
#                     savefig=True, fig_filename = 'fold_thrust_EW_section.eps',
                      ve=1.5)
```

```
# Calculate block difference and export as VTK for 3-D visualisation:
import copy
```

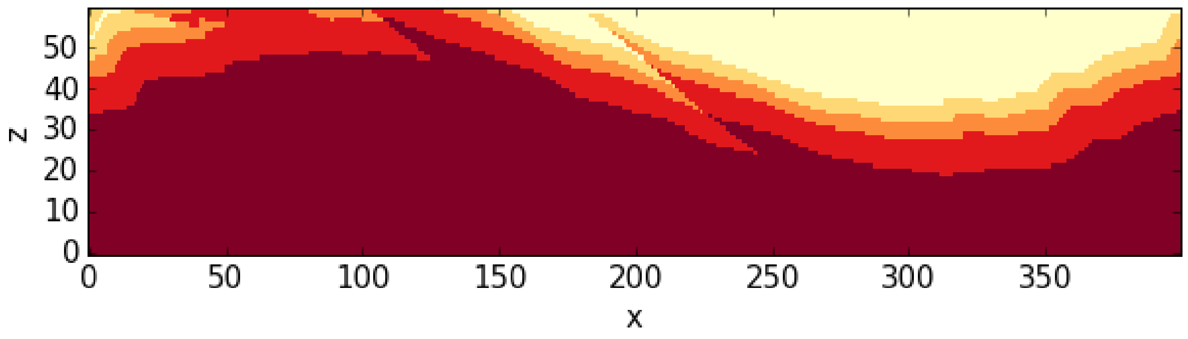

Fig. 6.9: png

```
diff_model = copy.deepcopy(geology_changed)
diff_model.block -= h_out.block
```

```
diff_model.export_to_vtk(vtk_filename = "diff_model_fold_thrust_belt")
```

6.4 Figure with all results

We now create a figure with the gravity field of the original and the changed model, as well as a difference plot to highlight areas with significant changes. This example also shows how additional equations can easily be combined with pynoddy classes.

```
fig = plt.figure(figsize=(20,8))
ax1 = fig.add_subplot(131)
# original plot
levels = np.arange(322,344,1)
cf1 = ax1.contourf(geophys.grv_data, levels, cmap = 'gray', vmin = 324, vmax = 342)
# cbar1 = ax1.colorbar(cf1, orientation = 'horizontal')
fig.colorbar(cf1, orientation='horizontal')
ax1.set_title('Gravity of original model')

ax2 = fig.add_subplot(132)




cf2 = ax2.contourf(geophys_changed.grv_data, levels, cmap = 'gray', vmin = 324, vmax = 342)
ax2.set_title('Gravity of changed model')
fig.colorbar(cf2, orientation='horizontal')

ax3 = fig.add_subplot(133)


comp_levels = np.arange(-10.,10.1,0.25)
cf3 = ax3.contourf(geophys.grv_data - geophys_changed.grv_data, comp_levels, cmap = 'RdBu_r')
ax3.set_title('Gravity difference')

fig.colorbar(cf3, orientation='horizontal')

plt.savefig("grav_ori_changed_compared.eps")
```

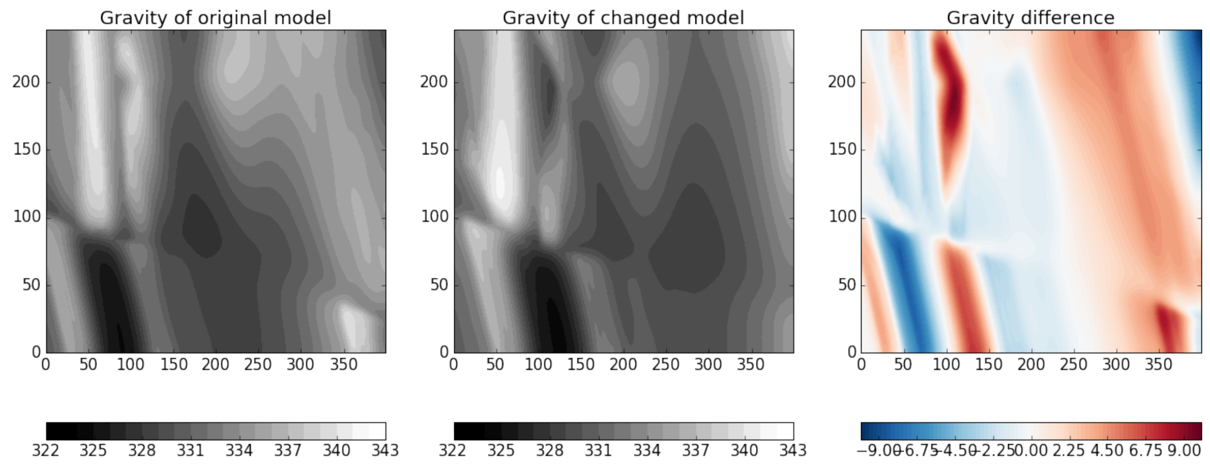
Gravity of original model
Gravity of changed model
Gravity difference
0
50
100
150
200
250
300
350
322
325
328
331
334
337
340
343
−9.00
−6.75
−4.50
−2.25
0.00
2.25
4.50
6.75
9.00

Fig. 6.10: png

CHAPTER

SEVEN

REPRODUCIBLE EXPERIMENTS WITH PYNODDY

All `pynoddy` experiments can be defined in a Python script, and if all settings are appropriate, then this script can be re-run to obtain a reproduction of the results. However, it is often more convenient to encapsulate all elements of an experiment within one class. We show here how this is done in the `pynoddy.experiment.Experiment` class and how this class can be used to define simple reproducible experiments with kinematic models.

```
from IPython.core.display import HTML
css_file = 'pynoddy.css'
HTML(open(css_file, "r").read())
```

```
%matplotlib inline
```

```
# here the usual imports. If any of the imports fails, make sure that pynoddy is installed
# properly, ideally with 'python setup.py develop' or 'python setup.py install'
import sys, os
import matplotlib.pyplot as plt
import numpy as np
# adjust some settings for matplotlib
from matplotlib import rcParams
# print rcParams
rcParams['font.size'] = 15
# determine path of repository to set paths corretly below
repo_path = os.path.realpath('../..')
import pynoddy.history
import pynoddy.experiment
reload(pynoddy.experiment)
rcParams.update({'font.size': 15})
```

7.1 Defining an experiment

We are considering the following scenario: we defined a kinematic model of a prospective geological unit at depth. As we know that the estimates of the (kinematic) model parameters contain a high degree of uncertainty, we would like to represent this uncertainty with the model.

Our approach is here to perform a randomised uncertainty propagation analysis with a Monte Carlo sampling method. Results should be presented in several figures (2-D slice plots and a VTK representation in 3-D).

To perform this analysis, we need to perform the following steps (see main paper for more details):

1. Define kinematic model parameters and construct the initial (base) model;
2. Assign probability distributions (and possible parameter correlations) to relevant uncertain input parameters;
3. Generate a set of n random realisations, repeating the following steps:
    (a) Draw a randomised input parameter set from the parameter distribu- tion;
    (b) Generate a model with this parameter set;
    (c) Analyse the generated model and store results;

4. Finally: perform postprocessing, generate figures of results

It would be possible to write a Python script to perform all of these steps in one go. However, we will here take another path and use the implementation in a Pynoddy Experiment class. Initially, this requires more work and a careful definition of the experiment - but, finally, it will enable a higher level of flexibility, extensibility, and reproducibility.

7.2 Setting up the experiment class

We use an experiment class that is pre-defined in the pynoddy.experiment module and inherits many base functions from the Experiment-class definition.

7.3 Loading an example model from the Virtual Explorer Atlas

As in the last example, we will use a model from the Virtual Explorer Atlas as an examlpe model for this simulation. We use a model for a fold interference structure. A discretised 3-D version of this model (from the Virtual Explorer website) is presented in the figure below. The model represents a fold interference pattern of "Type 1" according to the definition of Ramsey (1967).

Instead of loading the model into a history object, we are now directly creating an experiment object for the type of uncertainty analysis:

```
reload(pynoddy.history)
reload(pynoddy.experiment)

from pynoddy.experiment import UncertaintyAnalysis
reload(UncertaintyAnalysis)
# model_url = 'http://virtualexplorer.com.au/special/noddyatlas/ch3/ch3_7/his/typeb.his'
model_url = 'http://tectonique.net/asg/ch3/ch3_7/his/typeb.his'
ue = UncertaintyAnalysis.UncertaintyAnalysis(url = model_url)
# ue = pynoddy.experiment.UncertaintyAnalysis(history = "typeb_tmp.his")
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-4-0906eece45d0> in <module>()
      6 # model_url = 'http://virtualexplorer.com.au/special/noddyatlas/ch3/ch3_7/his/typeb.his'
      7 model_url = 'http://tectonique.net/asg/ch3/ch3_7/his/typeb.his'
----> 8 ue = UncertaintyAnalysis.UncertaintyAnalysis(url = model_url)
      9 # ue = pynoddy.experiment.UncertaintyAnalysis(history = "typeb_tmp.his")


TypeError: __init__() got an unexpected keyword argument 'url'
```

For simpler visualisation in this notebook, we will analyse the following steps in a section view of the model.

We consider a section in y-direction through the model:

```
ue.write_history("typeb_tmp3.his")
```

```
ue.write_history("typeb_tmp2.his")
```

```
ue.change_cube_size(100)
ue.plot_section('y')
```

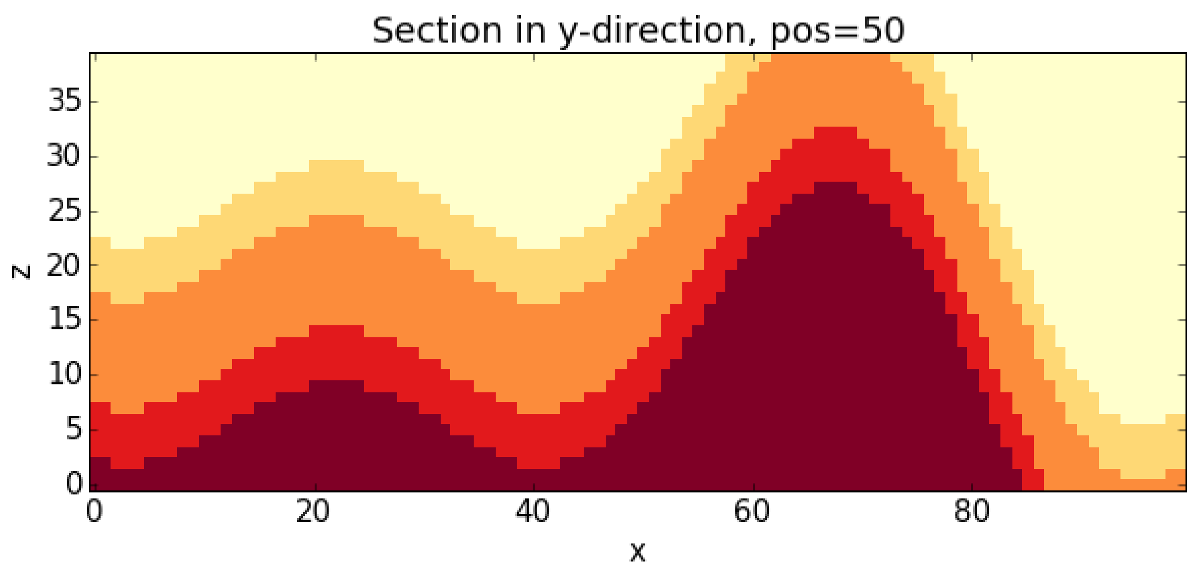

Fig. 7.1: png

Before we start to draw random realisations of the model, we should first store the base state of the model for later reference. This is simply possibel with the freeze() method which stores the current state of the model as the "base-state":

```
ue.freeze()
```

We now intialise the random generator. We can directly assign a random seed to simplify reproducibility (note that this is not *essential*, as it would be for the definition in a script function: the random state is preserved within the model and could be retrieved at a later stage, as well!):

```
ue.set_random_seed(12345)
```

The next step is to define probability distributions to the relevant event parameters. Let's first look at the different events:

```
ue.info(events_only = True)
```

```
This model consists of 3 events:
     (1) - STRATIGRAPHY
     (2) - FOLD
     (3) - FOLD
```

```
ev2 = ue.events[2]
```

```
ev2.properties
```

```
{'Amplitude': 1250.0,
 'Cylindricity': 0.0,
 'Dip': 90.0,
 'Dip Direction': 90.0,
 'Pitch': 0.0,
 'Single Fold': 'FALSE',
 'Type': 'Sine',
 'Wavelength': 5000.0,
 'X': 1000.0,
 'Y': 0.0,
 'Z': 0.0}
```

Next, we define the probability distributions for the uncertain input parameters:

```
param_stats = [{'event' : 2,
                'parameter': 'Amplitude',
                'stdev': 100.0,
                'type': 'normal'},
                {'event' : 2,
                'parameter': 'Wavelength',
                'stdev': 500.0,
                'type': 'normal'},
                {'event' : 2,
                'parameter': 'X',
                'stdev': 500.0,
                'type': 'normal'}]

ue.set_parameter_statistics(param_stats)
```

```
resolution = 100
ue.change_cube_size(resolution)
tmp = ue.get_section('y')
prob_4 = np.zeros_like(tmp.block[:,:,:])
n_draws = 10


for i in range(n_draws):
    ue.random_draw()
    tmp = ue.get_section('y', resolution = resolution)
    prob_4 += (tmp.block[:,:,:] == 4)

# Normalise
prob_4 = prob_4 / float(n_draws)
```

```
fig = plt.figure(figsize = (12,8))
ax = fig.add_subplot(111)
ax.imshow(prob_4.transpose()[:,0,:],
           origin = 'lower left',
           interpolation = 'none')
plt.title("Estimated probability of unit 4")
plt.xlabel("x (E-W)")
plt.ylabel("z")
```

```
<matplotlib.text.Text at 0x10c0b6a50>
```

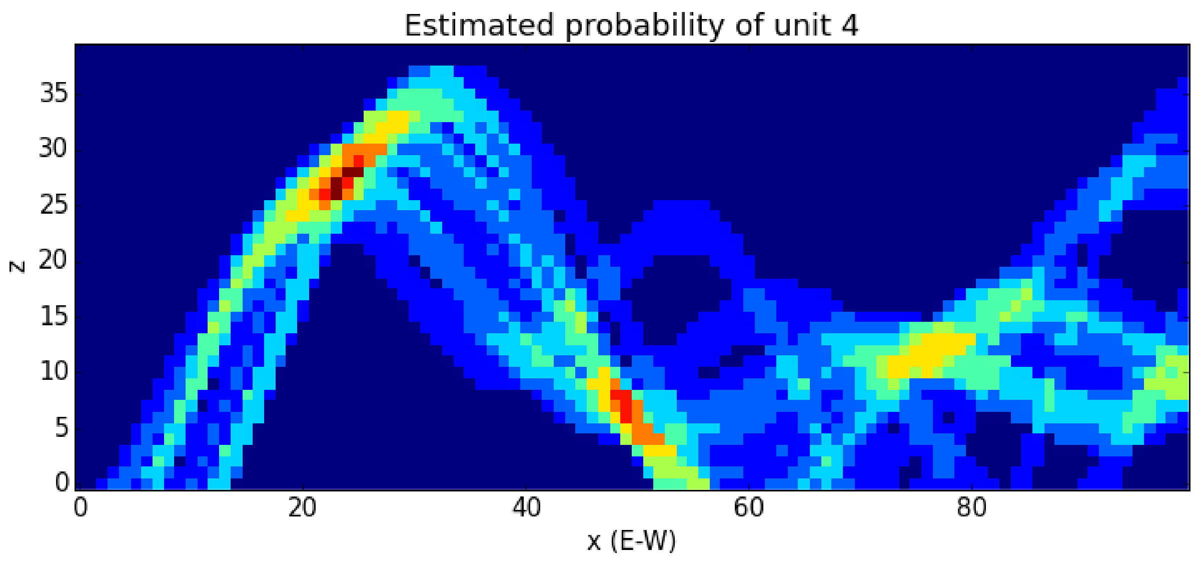

Fig. 7.2: png

```
reload(pynoddy.history)
reload(pynoddy.output)
reload(pynoddy.experiment)
model_url = 'http://virtualexplorer.com.au/special/noddyatlas/ch3/ch3_7/his/typeb.his'
# ue = pynoddy.experiment.UncertaintyAnalysis(url = model_url)
ue = pynoddy.experiment.UncertaintyAnalysis(history = "typeb_tmp.his")
ue.change_cube_size(100)
# tmp = ue.get_section('y')
# tmp.plot_section('y', position = 0, data = prob_4, cmap = 'jet')
```

```
STRATIGRAPHY
FOLD
FOLD
```

```
ue.plot_section('y')
```

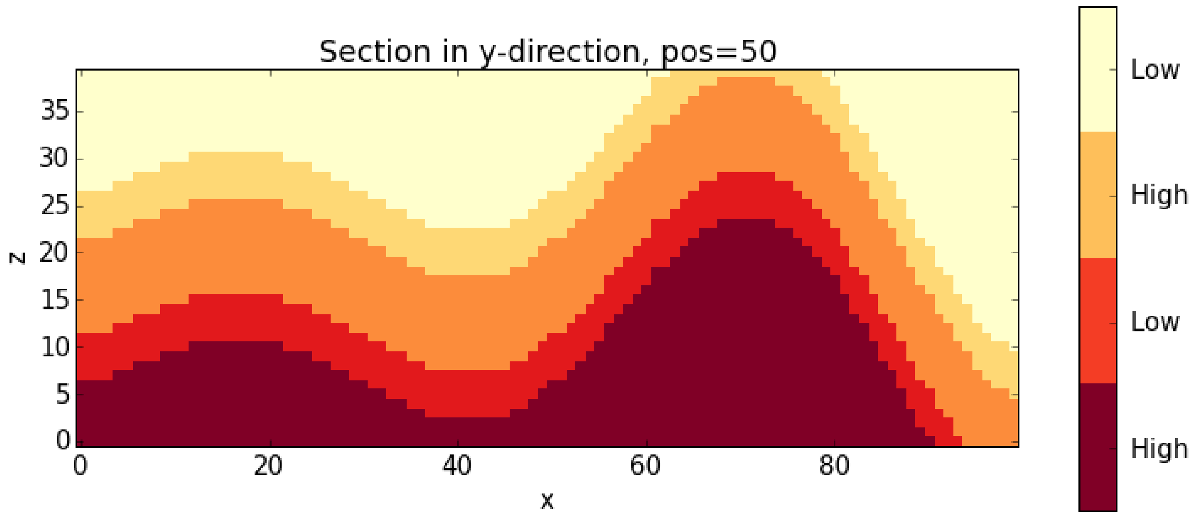

Fig. 7.3: png

```
ue.export_to_vtk(vtk_filename = "typeb")
```

```
ue.export_to_vtk(vtk_filename = "prob4", data = prob_4)
```

```
pwd
```

```
u'/Users/flow/git/pynoddy/docs/notebooks'
```

```
block = ue.get_section('y')
```

```
tmp = np.zeros_like(block.block)
tmp += (block.block[:,0,:] == 2)
```

```
plt.imshow(tmp[:,0,:].transpose())
```

```
<matplotlib.image.AxesImage at 0x10e9e5750>
```

```
block.plot_section('y')
```

```
plt.imshow(block.block[:,0,:].transpose(), origin = 'lower left', interpolation = 'none')
plt.colorbar(orientation = "horizontal")
```

```
<matplotlib.colorbar.Colorbar instance at 0x10eb97b48>
```

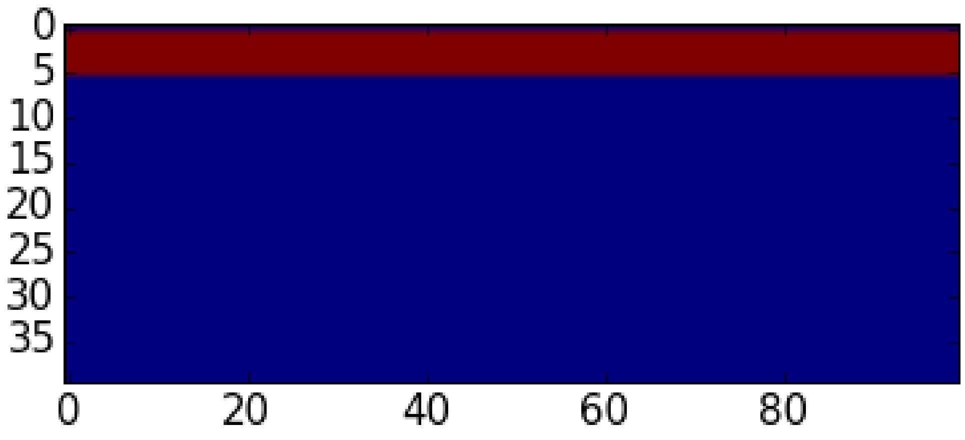

Fig. 7.4: png

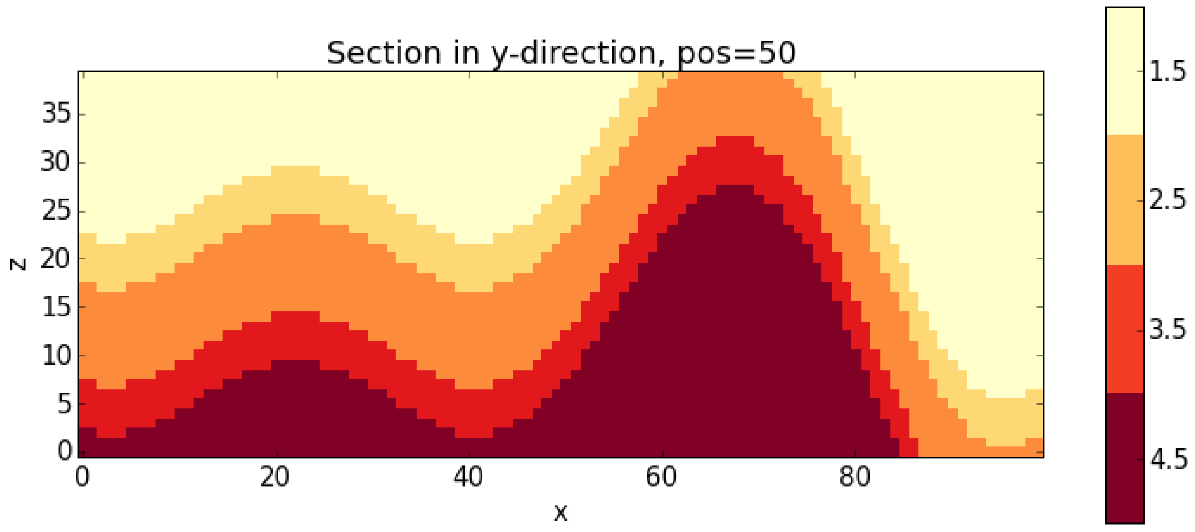

Fig. 7.5: png

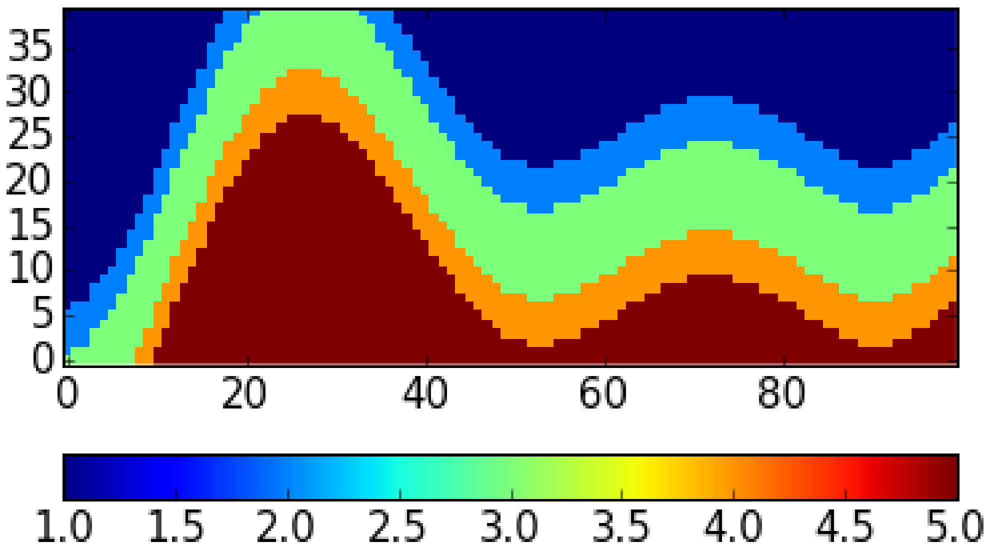

Fig. 7.6: png

```
# filter out unit 4:
test = np.zeros_like(block.block[:,0,:])
test += (block.block[:,0,:] == 4)
plt.imshow(test.transpose(), origin = 'lower left', interpolation = 'none')
```

```
<matplotlib.image.AxesImage at 0x10ed5cc90>
```

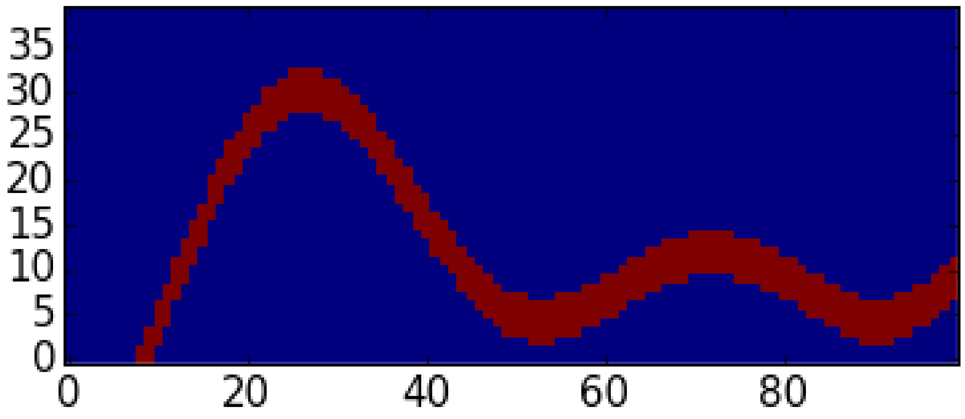

Fig. 7.7: png

```
ue.write_history("typeb_tmp.his")
```

```
ue.events[2].event_lines[-1]
```

```
'\tName\t= Fold\n'
```

```
for i,line in enumerate(ue.history_lines):
    if 'BlockOptions' in line:
        print ue.history_lines[i-1] == '\n'
```

```
True
```

```
list.insert??
```

```
a = ['a', 'b', 'c']
```

```
print a[2]
a.insert(2,'2')
```

```
c
```

```
a
```

```
['a', 'b', '2', 'c']
```