pyGeoPressure Documentation

Yu Hao

Jul 31, 2020

GETTING STARTED:

pyGeoPressure is an open source Python package for pore pressure prediction using well log data and seismic velocity data.

CHAPTER

ONE

FEATURES

1. Overburden (or Lithostatic) Pressure Calculation
2. Eaton’s method and Parameter Optimization
3. Bowers’ method and Parameter Optimization
4. Multivariate method and Parameter Optimization

CHAPTER

TWO

CONTRIBUTE

- Source Code: https://github.com/whimian/pyGeoPressure
- Issue Tracker: https://github.com/whimian/pyGeoPressure/issues

CHAPTER

THREE

LICENSE

The project is licensed under the MIT license, see the file 'MIT' for details.

CHAPTER

FOUR

DOCUMENTATION STRUCTURE

- **Getting Started** (*thorough introduction and installation instructions*)
- *Tutorials* (*walkthrough of main features using example survey*)
- **How-to** (*topic guides*)
- **References** (*inner workings*)

CHAPTER

FIVE

CONTENTS

5.1 Introduction

Pore pressure (geopressure) is of great importance in different stages of oil and gas (hydrocarbon) exploration and development. Predicted regional pressure data can help with:

1. well planning
2. Preventing hazards like kicks and blowouts.
3. building geomechanical model
4. anaylizing hydrocarbon distribution

Pore pressure prediction is to use geophyical and petrophysical properties (like velocity, resistivity) measured or calculated to evaluate pore pressure underground instead of measuring pressure directly which is expensive and can only be done after a well is drilled. Usually pore pressure prediction is performed with well logging data after exploration wells are drilled and cemented, and with seismic velocity data for regional pore pressure prediction.

`pyGeoPressure` is an open source python package designed for pore pressure prediction with both well log data and seismic velocity data. Though lightweighted, `pyGeoPressure` is able to perform whole workflow from data management to pressure prediction.

The main features of `pyGeoPressure` are:

1. Overburden (or Lithostatic) Pressure Calculation (Tutorials: *OBP calculation for well*, *OBP calculation for seismic*)
2. Eaton's method and Parameter Optimization (Tutorials: *Eaton for well*, *Eaton for seismic*)
3. Bowers' method and Parameter Optimization (Tutorials: *Bowers for well*, *Bowers for seismic*)
4. Multivariate method and Parameter Optimization (Tutorials: *Multivariate for well*)

Aside from main prediction features, `pyGeoPressrue` provides other functionalities to facilitate the workflow:

- Survey definition
- Data Management
- Well log data processing
- Generating figures

5.2 Installation

5.2.1 Denpendencies

`pyGeoPressure` supports both Python 2.7 and Python 3.6 and some of mainly dependent packages are:

- NumPy
- SciPy
- matplotlib
- Jupyter
- segyio

5.2.2 Installing Python

The recommended way to intall Python is to use conda package manager from Anaconda Inc. You may download and install Miniconda from https://conda.io/miniconda which contains both Python and conda package manager.

5.2.3 Installing pyGeoPressure

`pyGeoPressure` is recommended to be installed in a seperate python environment which can be easily created with `conda`. So first create a new environment with `conda`. The new environment should have `pip` installed.

```
conda update conda
conda create -n ENV python=3.6 pip
```

or

```
conda update conda
conda create -n ENV python=2.7 pip
```

if using Python 2.7.

Install from `pyPI`

`pyGeoPressure` is on `PyPI`, so run the following command to install `pyGeoPressure` from `pypi`.

```
pip install pygeopressure
```

Install from `github repo`

Install latest develop branch from github:

```
pip install -e git://github.com/whimian/pyGeoPressure.git@develop
```

Alternatively, if you don't have `git` installed, you can download the repo from Github, unzip, `cd` to that directory and run:

```
pip install pyGeoPressure
```

5.2.4 For Developers

Clone the github repo:

```
git clone https://github.com/whimian/pyGeoPressure.git
```

Setup the development environment with `conda`:

```
conda env create --file test/test_env_2.yml
```

or

```
conda env create --file test/test_env_3.yml
```

The testing framework used is `pytest`. To run all tests, just run the following code at project directory:

```
pytest --cov
```

5.3 Overview

Note: The following tutorials are created with a set of jupyter notebooks, users may download these notebooks (Download) and the example survey (Download), and run these notebooks locally.

5.3.1 OBP calculation for well

OBP calculation include the following step:

1. Extrapolate density log to the surface
2. Calculate Overburden Pressure

- Calculate Hydrostatic Pressrue (*)

[2]:
```
from __future__ import print_function, division, unicode_literals
%matplotlib inline
import matplotlib.pyplot as plt

plt.style.use(['seaborn-paper', 'seaborn-whitegrid'])
plt.rcParams['font.sans-serif']=['SimHei']
plt.rcParams['axes.unicode_minus']=False
```

(continues on next page)

(continued from previous page)

```
import numpy as np

import pygeopressure as ppp
```

1. Extrapolate density log to the surface

Create survey with the example survey `CUG`:

```
[18]: # set to the directory on your computer
SURVEY_FOLDER = "M:/CUG_depth"

survey = ppp.Survey(SURVEY_FOLDER)
```

Retrieve well `CUG1`:

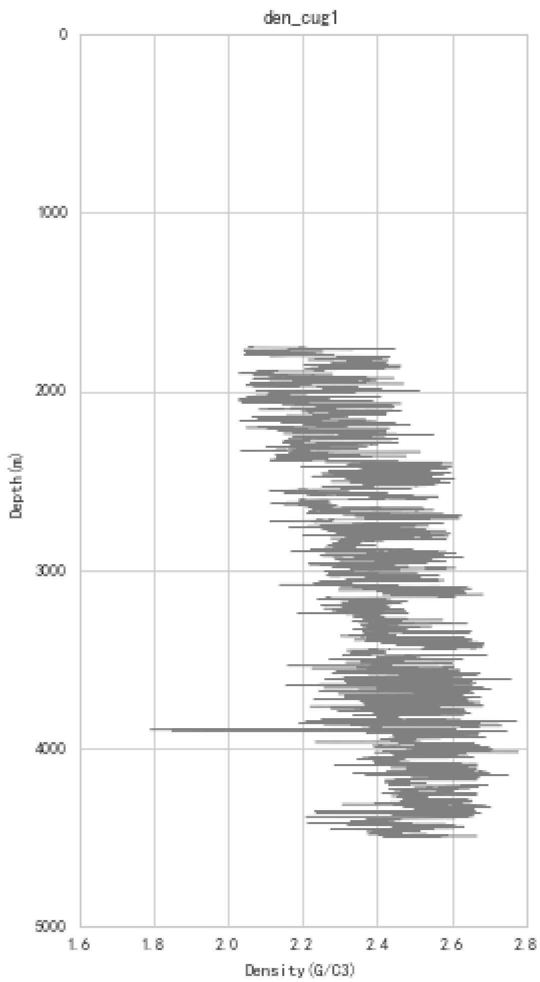
```
[4]: well_cug1 = survey.wells['CUG1']
```

Get density log:

```
[5]: den_log = well_cug1.get_log("Density")
```

View density log:

```
[6]: fig_den, ax_den = plt.subplots()
ax_den.invert_yaxis()

den_log.plot(ax_den)

# set style
ax_den.set(ylim=(5000,0), aspect=(1.2/5000)*2)
fig_den.set_figheight(8)
fig_den.show()
```



Find optimized coefficients for Traugott equation:

```
[7]: a, b = ppp.optimize_traugott(
         den_log, 2000, 3000, kb=well_cug1.kelly_bushing, wd=well_cug1.water_depth)
```

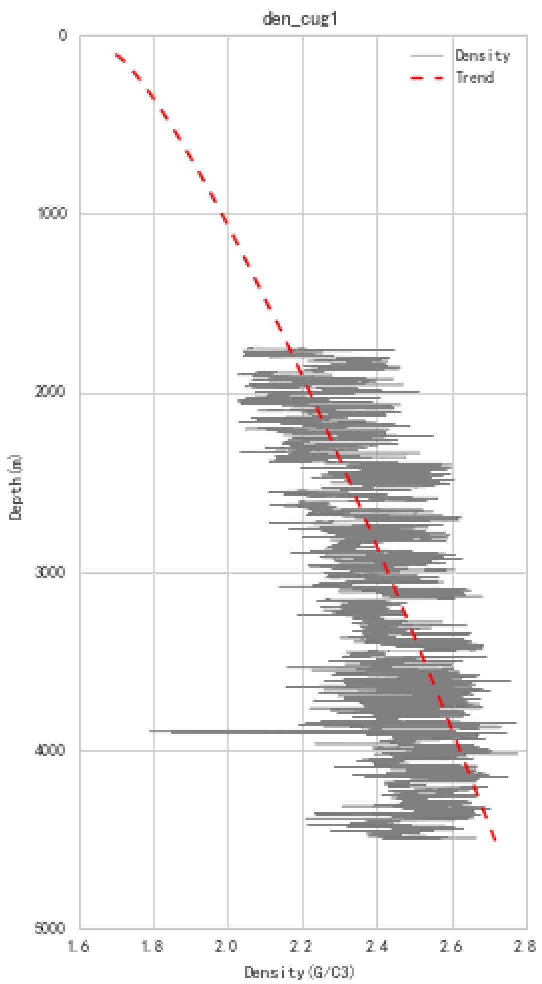
View fitted density trend:

```
[8]: fig_den, ax_den = plt.subplots()
     ax_den.invert_yaxis()
     # draw density log
     den_log.plot(ax_den, label='Density')
     # draw fitted density trend line
     den_trend = ppp.traugott_trend(
         np.array(den_log.depth), a, b,
         kb=well_cug1.kelly_bushing, wd=well_cug1.water_depth)

     ax_den.plot(den_trend, den_log.depth,
                 color='r', linestyle='--', zorder=2, label='Trend')

     # set style
     ax_den.set(ylim=(5000,0), aspect=(1.2/5000)*2)
     ax_den.legend()
```

(continues on next page)

(continued from previous page)

```
fig_den.set_figheight(8)
fig_den.show()
```



Since we will extrapolate density to mudline (sea bottom), density values of the inverval from mudline to kelly bushing will be NaN (See figure above).

Also, the actual variation of rock density underground doesnot have such high frequency as density logging data, so we need to perform some filtering and smoothing of the original signal.

Density log processing (filtering and smoothing):

[9]:
```
den_log_filter = ppp.upscale_log(den_log, freq=20)

den_log_filter_smooth = ppp.smooth_log(den_log_filter, window=1501)
```

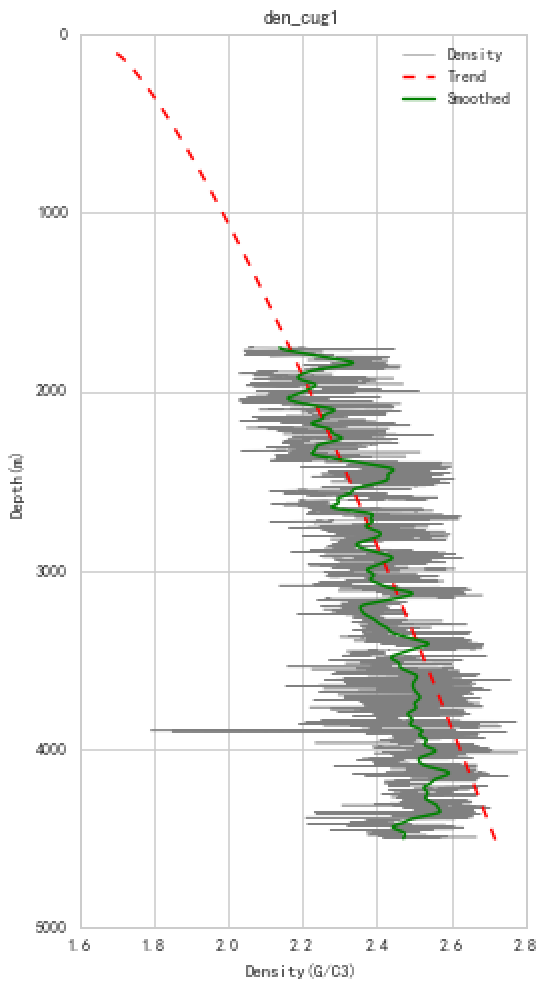
View processed log data:

[10]:
```
fig_den, ax_den = plt.subplots()
ax_den.invert_yaxis()
# draw density log
den_log.plot(ax_den, label='Density')
# draw fitted density trend line
```

(continues on next page)

(continued from previous page)

```
ax_den.plot(den_trend, den_log.depth,
            color='r', linestyle='--', zorder=2, label='Trend')
# draw processed density log
ax_den.plot(den_log_filter_smooth.data, den_log_filter_smooth.depth,
            color='g', zorder=3, label='Smoothed')

# set style
ax_den.set(ylim=(5000,0), aspect=(1.2/5000)*2)
ax_den.legend()
fig_den.set_figheight(8)
fig_den.show()
```



Extrapolate processed density log with fitted trend:

```
[11]: extra_log = ppp.extrapolate_log_traugott(
          den_log_filter_smooth, a, b,
          kb=well_cug1.kelly_bushing, wd=well_cug1.water_depth)
```

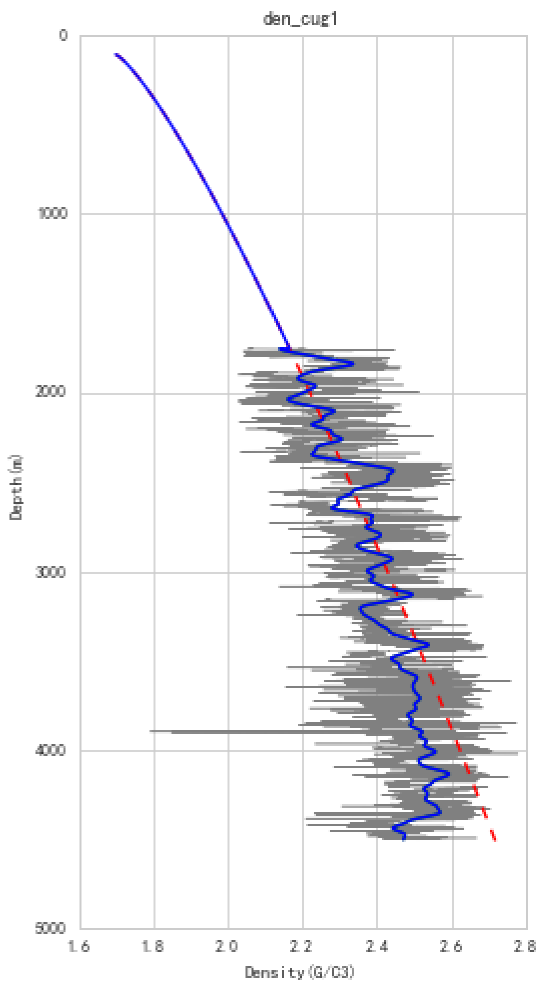
View extraploted density log:

```
[12]: fig_den, ax_den = plt.subplots()
      ax_den.invert_yaxis()
```

(continues on next page)

(continued from previous page)

```
# draw density log
den_log.plot(ax_den, label='Density')
# draw trend line
ax_den.plot(den_trend, den_log.depth,
            color='r', linestyle='--', zorder=2, label='Trend')
# draw processed density log
ax_den.plot(den_log_filter_smooth.data, den_log_filter_smooth.depth,
            color='g', zorder=3, label='Smoothed')
# draw extrapolated density
ax_den.plot(extra_log.data, extra_log.depth,
            color='b', zorder=4, label='Extrapolated')

# set style
ax_den.set(ylim=(5000,0), aspect=(1.2/5000)*2)
fig_den.set_figheight(8)
fig_den.show()
```



The extrapolated log (blue line in the figure above) is used for calculation of Overburden Pressure.

2. Calculation of Overburden Pressure

[13]:
```
obp_log = ppp.obp_well(extra_log,
         kb=well_cug1.kelly_bushing, wd=well_cug1.water_depth,
         rho_w=1.01)
```

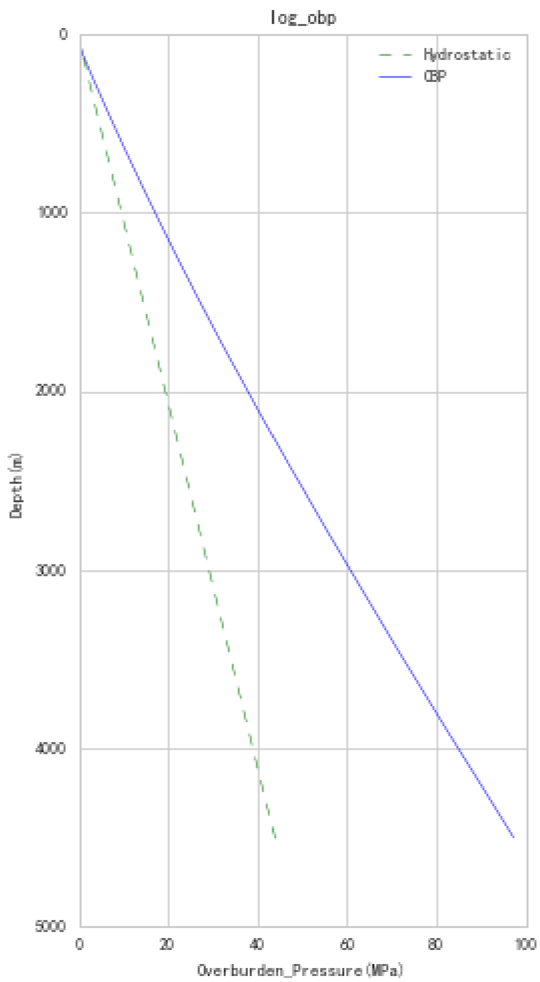
* Calculation of Hydrostatic Pressure

Since parameters used for hydrostatic pressrue calcualtion like kelly busshing and water depth are store in Well, so we add a shortcut for hydrostatic pressure calculation in Well.

[14]:
```
# hydro_log = ppp.hydrostatic_well(
#     obp_log.depth, kb=well_cug1.kelly_bushing, wd=well_cug1.water_depth,
#     rho_f=1., rho_w=1.)

hydro_log = well_cug1.hydro_log()
```

View calcualted overburden pressure:

[15]:
```
fig_obp, ax_obp = plt.subplots()
ax_obp.invert_yaxis()

hydro_log.plot(ax_obp, color='g', linestyle='--', label='Hydrostatic')

obp_log.plot(ax_obp, color='b', label='OBP')

# set style
ax_obp.set(ylim=(5000,0), aspect=(100/5000)*2)
ax_obp.legend()
fig_obp.set_figheight(8)
fig_obp.show()
```



Save calculated Overburden Pressure:

```
[16]: # well_cug1.add_log("Overbuden_Pressure")
```

optional, calculated overburden pressure has already been saved, so users don't need to run these notebooks in specific order.

5.3.2 Eaton method with well log

Pore pressure prediction with Eaton's method using well log data.

Steps:

1. Calculate Velocity Normal Compaction Trend
2. Optimize for Eaton's exponent n
3. Predict pore pressure using Eaton's method

```
[2]: from __future__ import print_function, division, unicode_literals
%matplotlib inline
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
plt.style.use(['seaborn-paper', 'seaborn-whitegrid'])
plt.rcParams['font.sans-serif']=['SimHei']
plt.rcParams['axes.unicode_minus']=False

import numpy as np

import pygeopressure as ppp
```

1. Calculate Velocity Normal Compaction Trend

Create survey with the example survey CUG:

```
[3]: # set to the directory on your computer
SURVEY_FOLDER = "C:/Users/yuhao/Desktop/CUG_depth"

survey = ppp.Survey(SURVEY_FOLDER)
```

Retrieve well CUG1:

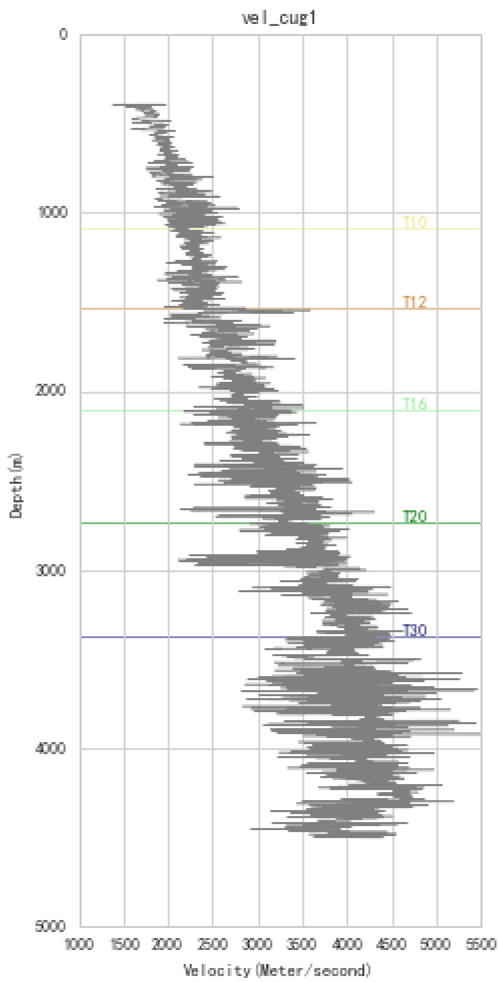
```
[4]: well_cug1 = survey.wells['CUG1']
```

Get velocity log:

```
[5]: vel_log = well_cug1.get_log("Velocity")
```

View velocity log:

```
[6]: fig_vel, ax_vel = plt.subplots()
ax_vel.invert_yaxis()
vel_log.plot(ax_vel)
well_cug1.plot_horizons(ax_vel)

# set fig style
ax_vel.set(ylim=(5000,0), aspect=(5000/4600)*2)
ax_vel.set_aspect(2)
fig_vel.set_figheight(8)
```



Optimize for NCT coefficients a, b:

`well.params['horizon']['T20']` returns the depth of horizon T20.

[7]:
```
a, b = ppp.optimize_nct(
    vel_log=well_cug1.get_log("Velocity"),
    fit_start=well_cug1.params['horizon']["T16"],
    fit_stop=well_cug1.params['horizon']["T20"])
```

And use a, b to calculate normal velocity trend

[8]:
```
from pygeopressure.velocity.extrapolate import normal_log
nct_log = normal_log(vel_log, a=a, b=b)
```

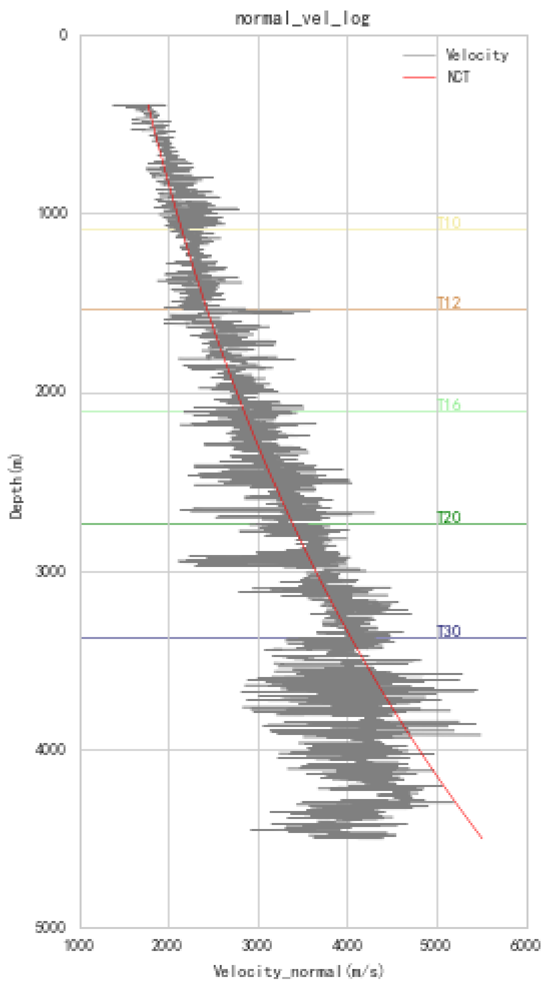
View fitted NCT:

[9]:
```
fig_vel, ax_vel = plt.subplots()
ax_vel.invert_yaxis()
# plot velocity
vel_log.plot(ax_vel, label='Velocity')
# plot horizon
well_cug1.plot_horizons(ax_vel)
# plot fitted nct
```

(continues on next page)

(continued from previous page)

```
nct_log.plot(ax_vel, color='r', zorder=2, label='NCT')

# set fig style
ax_vel.set(ylim=(5000,0), aspect=(5000/4600)*2)
ax_vel.set_aspect(2)
ax_vel.legend()
fig_vel.set_figheight(8)
```



Save fitted nct:

```
[10]: # well_cug1.params['nct'] = {"a": a, "b": b}

      # well_cug1.save_params()
```

2. Optimize for Eaton's exponent n

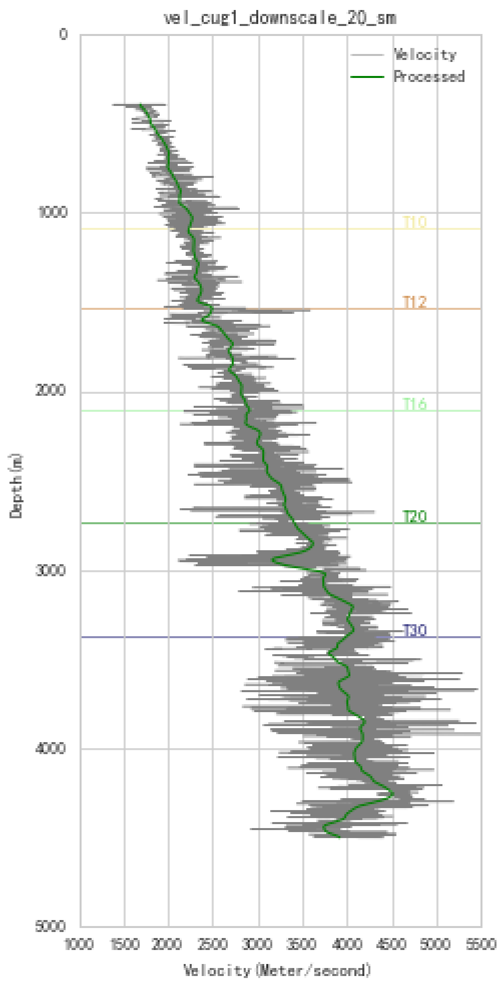
First, we need to preprocess velocity.

Velocity log processing (filtering and smoothing):

[11]:
```
vel_log_filter = ppp.upscale_log(vel_log, freq=20)

vel_log_filter_smooth = ppp.smooth_log(vel_log_filter, window=1501)
```

Veiw processed velocity:

[12]:
```
fig_vel, ax_vel = plt.subplots()
ax_vel.invert_yaxis()
# plot velocity
vel_log.plot(ax_vel, label='Velocity')
# plot horizon
well_cug1.plot_horizons(ax_vel)
# plot processed velocity
vel_log_filter_smooth.plot(ax_vel, color='g', zorder=2, label='Processed',␣
↪linewidth=1)

# set fig style
ax_vel.set(ylim=(5000,0), aspect=(5000/4600)*2)
ax_vel.set_aspect(2)
ax_vel.legend()
fig_vel.set_figheight(8)
```



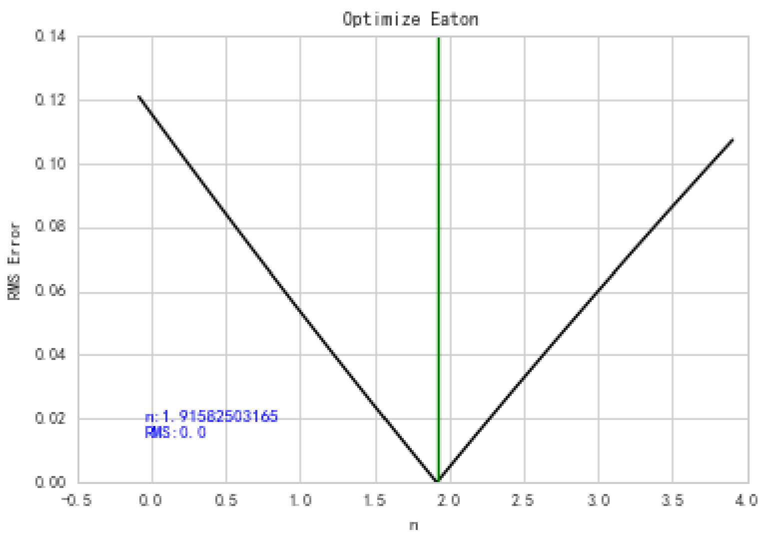
We will use the processed velocity data for pressure prediction.

Optimize Eaton's exponential `n`:

[13]:
```
n = ppp.optimize_eaton(
    well=well_cug1,
    vel_log=vel_log_filter_smooth,
    obp_log="Overburden_Pressure",
    a=a, b=b)
```

See the RMS error variation with `n`:

[14]:
```
from pygeopressure.basic.plots import plot_eaton_error

fig_err, ax_err = plt.subplots()

plot_eaton_error(
    ax=ax_err,
    well=well_cug1,
    vel_log=vel_log_filter_smooth,
    obp_log="Overburden_Pressure",
    a=a, b=b)
```



Save optimized n:

```
[15]: # well_cug1.params['nct'] = {"a": a, "b": b}

# well_cug1.save_params()
```

3.Predict pore pressure using Eaton's method

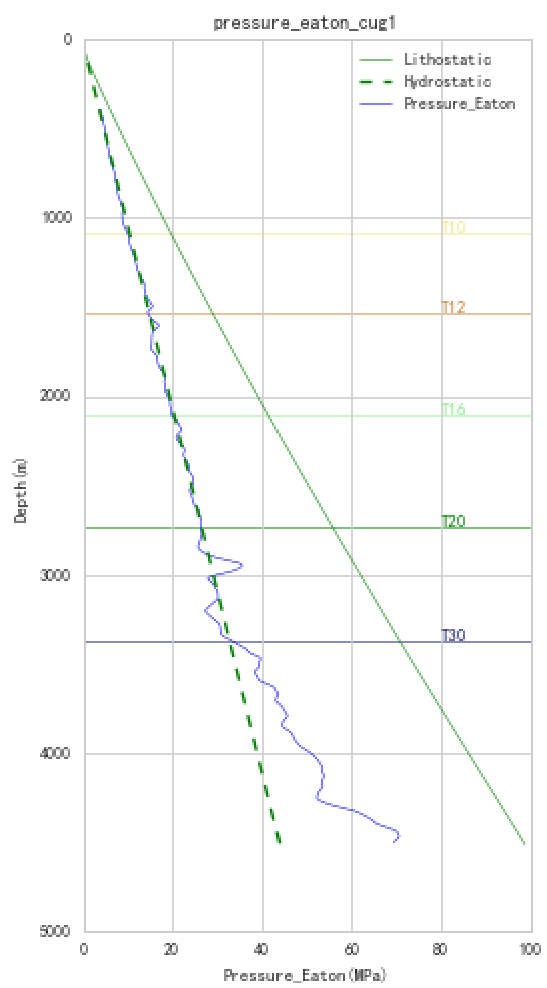
Calculate pore pressure using Eaton's method requires velocity, Eaton's exponential, normal velocity, hydrostatic pressure and overburden pressure.

`Well.eaton()` will try to read saved data, users only need to specify them when they are different from the saved ones.

```
[16]: pres_eaton_log = well_cug1.eaton(vel_log_filter_smooth, n=n)
```

View predicted pressure:

```
[17]: fig_pres, ax_pres = plt.subplots()
ax_pres.invert_yaxis()

well_cug1.get_log("Overburden_Pressure").plot(ax_pres, 'g', label='Lithostatic')
ax_pres.plot(well_cug1.hydrostatic, well_cug1.depth, 'g', linestyle='--', label=
↪"Hydrostatic")
pres_eaton_log.plot(ax_pres, color='blue', label='Pressure_Eaton')
well_cug1.plot_horizons(ax_pres)

# set figure and axis size
ax_pres.set_aspect(2/50)
ax_pres.legend()
fig_pres.set_figheight(8)
```



5.3.3 Bowers method with well log

Pore pressure prediction with Bowers' method using well log data.

Predicton of geopressure using Bowers' model needs the following steps:

1. determine Bowers loading equation coefficients A and B
2. determine Bowers unloading equation coefficients V_{max} and U
3. Pressure Prediction

[2]:
```
from __future__ import print_function, division, unicode_literals
%matplotlib inline
import matplotlib.pyplot as plt

plt.style.use(['seaborn-paper', 'seaborn-whitegrid'])
plt.rcParams['font.sans-serif']=['SimHei']
plt.rcParams['axes.unicode_minus']=False

import numpy as np
```
(continues on next page)

(continued from previous page)

```
import pygeopressure as ppp
```

1. determine Bowers loading equation coefficients A and B

Create survey with the example survey CUG:

```
[3]: # set to the directory on your computer
SURVEY_FOLDER = "M:/CUG_depth"

survey = ppp.Survey(SURVEY_FOLDER)
```

Retrieve well CUG1:

```
[4]: well_cug1 = survey.wells['CUG1']
```

Get velocity log:

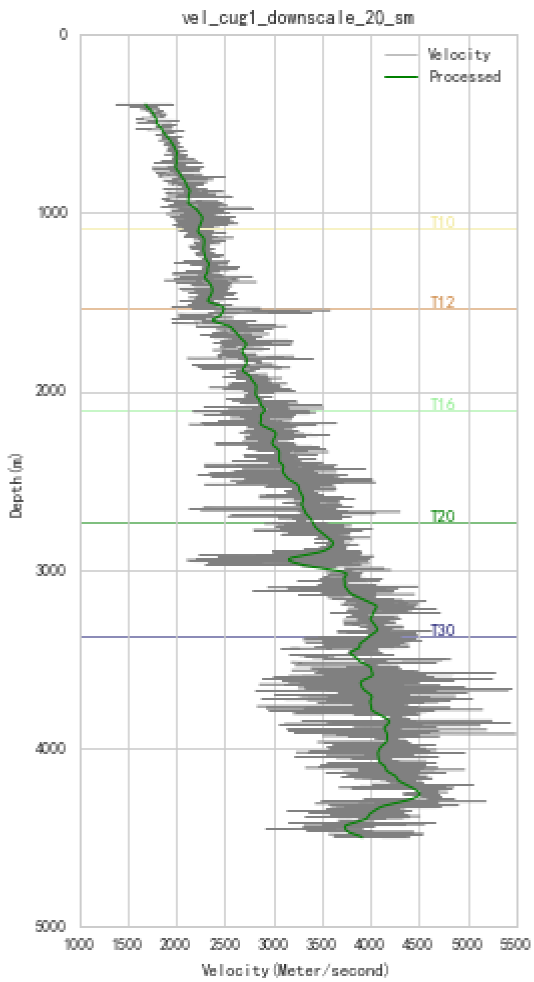
```
[5]: vel_log = well_cug1.get_log("Velocity")
```

Preprocessing velocity data

```
[6]: vel_log_filter = ppp.upscale_log(vel_log, freq=20)

vel_log_filter_smooth = ppp.smooth_log(vel_log_filter, window=1501)
```

View velocity and processed velocity

```
[7]: fig_vel, ax_vel = plt.subplots()
ax_vel.invert_yaxis()
# plot velocity
vel_log.plot(ax_vel, label='Velocity')
# plot horizon
well_cug1.plot_horizons(ax_vel)
# plot processed velocity
vel_log_filter_smooth.plot(ax_vel, color='g', zorder=2, label='Processed',␣
↪linewidth=1)

# set fig style
ax_vel.set(ylim=(5000,0), aspect=(4600/5000)*2)
ax_vel.legend()
fig_vel.set_figheight(8)
```



Optimize for Bowers' loading equation coefficients A, B:

[8]:
```
a, b, err = ppp.optimize_bowers_virgin(
    well=well_cug1,
    vel_log=vel_log_filter_smooth,
    obp_log='Overburden_Pressure',
    upper='T12',
    lower='T20',
    pres_log='loading',
    mode='both')
```

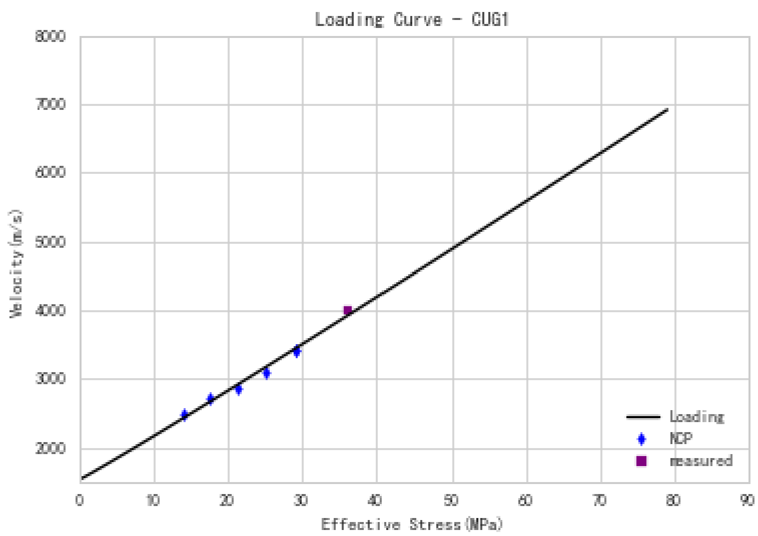
Plot optimized virgin curve:

[9]:
```
fig_bowers, ax_bowers = plt.subplots()

ppp.plot_bowers_vrigin(
    ax=ax_bowers,
    well=well_cug1,
    a=a,
    b=b,
    vel_log=vel_log_filter_smooth,
    obp_log='Overburden_Pressure',
```

(continues on next page)

(continued from previous page)

```
    upper='T12',
    lower='T20',
    pres_log='loading',
    mode='both')
```



2. determine Bowers unloading equation coefficients V_{max} and U

After manually select paramter U, optimze for parameter U:

[10]:
```
u = ppp.optimize_bowers_unloading(
    well=well_cug1,
    vel_log=vel_log_filter_smooth,
    obp_log='Overburden_Pressure',
    a=a,
    b=b,
    vmax=4600,
    pres_log='unloading')
```

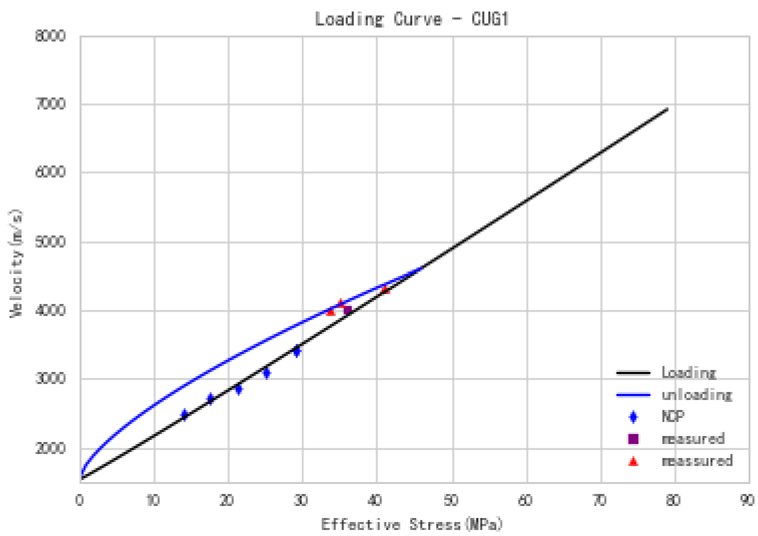
Draw unloading curve and virgin curve together with optimized parameters:

[11]:
```
fig_bowers, ax_bowers = plt.subplots()
# draw virgin(loading) curve
ppp.plot_bowers_vrigin(
    ax=ax_bowers,
    well=well_cug1,
    a=a,
    b=b,
    vel_log=vel_log_filter_smooth,
    obp_log='Overburden_Pressure',
    upper='T12',
    lower='T20',
    pres_log='loading',
    mode='both')

# draw unloading curve
ppp.plot_bowers_unloading(
```

(continues on next page)

(continued from previous page)

```
    ax=ax_bowers,
    a=a,
    b=b,
    vmax=4600,
    u=u,
    well=well_cug1,
    vel_log=vel_log_filter_smooth,
    obp_log='Overburden_Pressure',
    pres_log='unloading')
```



3. Pressure Prediction with Bowers model

predict pressure with coefficients calculated above:

[12]:
```
pres_log = well_cug1.bowers(
    vel_log=vel_log_filter_smooth, a=a, b=b, u=u)
```

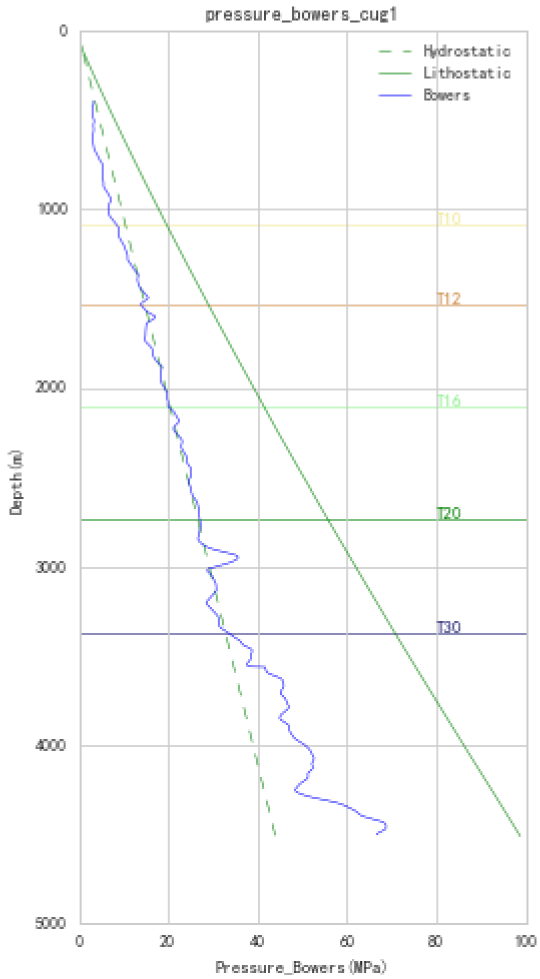
View Bowers Pressure Results:

[13]:
```
fig_pres, ax_pres = plt.subplots()
ax_pres.invert_yaxis()
# plot hydrostatic
well_cug1.hydro_log().plot(ax_pres, linestyle='--', color='green', label='Hydrostatic
↪')
# plot OBP
well_cug1.get_log("Overburden_Pressure").plot(ax_pres, color='green', label=
↪'Lithostatic')
# plot pressure
pres_log.plot(ax_pres, label='Bowers', color='blue')
# plot horizon
well_cug1.plot_horizons(ax_pres)


# set fig style
ax_pres.set(ylim=(5000,0), aspect=(100/5000)*2)
```

(continues on next page)

(continued from previous page)

```
ax_pres.legend()
fig_pres.set_figheight(8)
```



5.3.4 Multivairate Model

```
[2]: from __future__ import print_function, division, unicode_literals
%matplotlib inline
import matplotlib.pyplot as plt

plt.style.use(['seaborn-paper', 'seaborn-whitegrid'])
plt.rcParams['font.sans-serif']=['SimHei']
plt.rcParams['axes.unicode_minus']=False

import numpy as np

import pygeopressure as ppp
```

1. Calculate optimized multivariate model coefficients

Create survey with the example survey CUG:

[3]:
```
# set to the directory on your computer
SURVEY_FOLDER = "M:/CUG_depth"

survey = ppp.Survey(SURVEY_FOLDER)
```

Retrieve well CUG1:

[4]:
```
well_cug1 = survey.wells['CUG1']
```

Get Velocity, Shale volume and Porosity logs:

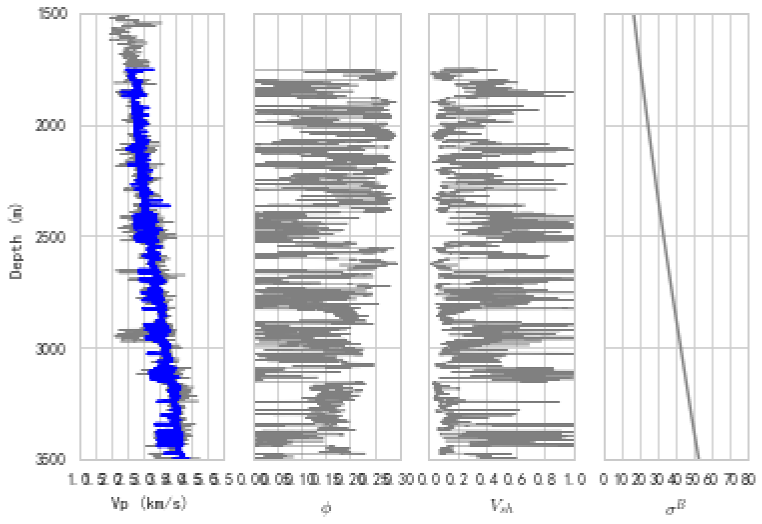
[5]:
```
vel_log = well_cug1.get_log("Velocity")
por_log = well_cug1.get_log("Porosity")
vsh_log = well_cug1.get_log("Shale_Volume")

obp_log = well_cug1.get_log("Overburden_Pressure")
```

Calculate optimized multivariate model parameters:

[6]:
```
a0, a1, a2, a3 = ppp.optimize_multivaraite(
    well=well_cug1,
    obp_log=obp_log,
    vel_log=vel_log,
    por_log=por_log,
    vsh_log=vsh_log,
    B=well_cug1.params['bowers']['B'],
    upper=1500, lower=3500)
```

View velocity, porosity, shale volume and effecive pressure used for optimization, and Velocity predicted by the optimized model (blue line):

[7]:
```
fig, axes = plt.subplots(ncols=4, nrows=1, sharey=True)
axes[0].invert_yaxis()

ppp.plot_multivariate(
    axes,
    well_cug1,
    vel_log, por_log, vsh_log, obp_log, 1500, 3500, a0, a1, a2, a3,
    well_cug1.params['bowers']['B'])
```



2. Pressure Prediction with multivaraite model

Multivaraite pressure prediction:

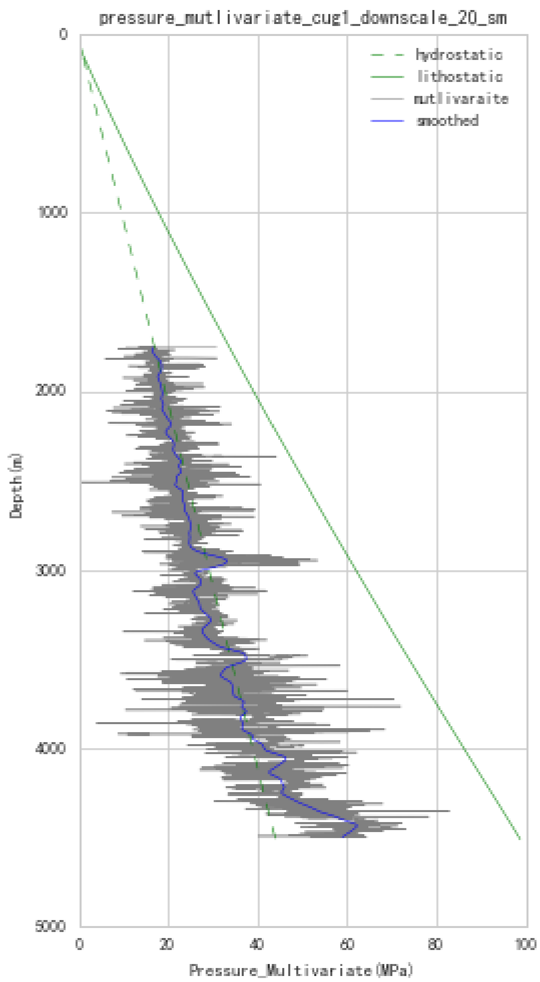
[8]:
```
pres_log = well_cug1.multivariate(vel_log, por_log, vsh_log)
```

Post-process predicted pressure:

[9]:
```
pres_log_filter = ppp.upscale_log(pres_log, freq=20)

pres_log_filter_smooth = ppp.smooth_log(pres_log_filter, window=1501)
```

View predicted pressure:

[10]:
```
fig_pres, ax_pres = plt.subplots()
ax_pres.invert_yaxis()

well_cug1.hydro_log().plot(ax_pres, color='green', linestyle='--',
                           zorder=2, label='hydrostatic')

well_cug1.get_log("Overburden_Pressure").plot(ax_pres, color='g',
                                              label='lithostatic')

pres_log.plot(ax_pres, label='mutlivaraite', zorder=1)

pres_log_filter_smooth.plot(ax_pres, label='smoothed', zorder=5, color='b')

ax_pres.set(xlim=[0,100], ylim=[5000,0], aspect=(100/5000)*2)
ax_pres.legend()
fig_pres.set(figheight=8)
fig_pres.show()
```



5.3.5 Overburden Pressure Calculation (Seismic)

Overburden Pressure Calculation involves:

1. estimation of density data
2. calculation of OBP

[ ]:
```
from __future__ import print_function, division, unicode_literals
%matplotlib inline
import matplotlib.pyplot as plt

plt.style.use(['seaborn-paper', 'seaborn-whitegrid'])
plt.rcParams['font.sans-serif']=['SimHei']
plt.rcParams['axes.unicode_minus']=False

import numpy as np

import pygeopressure as ppp
```

1. Estimation of density data

Create survey with the example survey CUG:

[2]:
```
# set to the directory on your computer
SURVEY_FOLDER = "C:/Users/yuhao/Desktop/CUG_depth"

survey = ppp.Survey(Path(SURVEY_FOLDER))
```

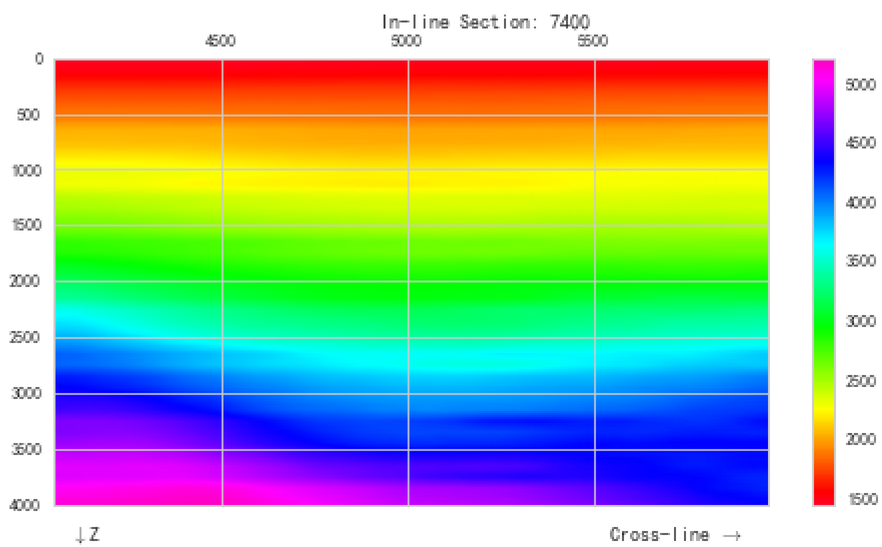
Retrieve Velocity data:

[3]:
```
vel_cube = survey.seismics['velocity']
```

View Velocity cube section:

[4]:
```
fig_vel, ax_vel = plt.subplots()

im = vel_cube.plot(ppp.InlineIndex(7400), ax_vel, kind='img', cm='gist_rainbow')
fig_vel.colorbar(im)
fig_vel.set(figwidth=8)
```

[4]:
```
[None]
```



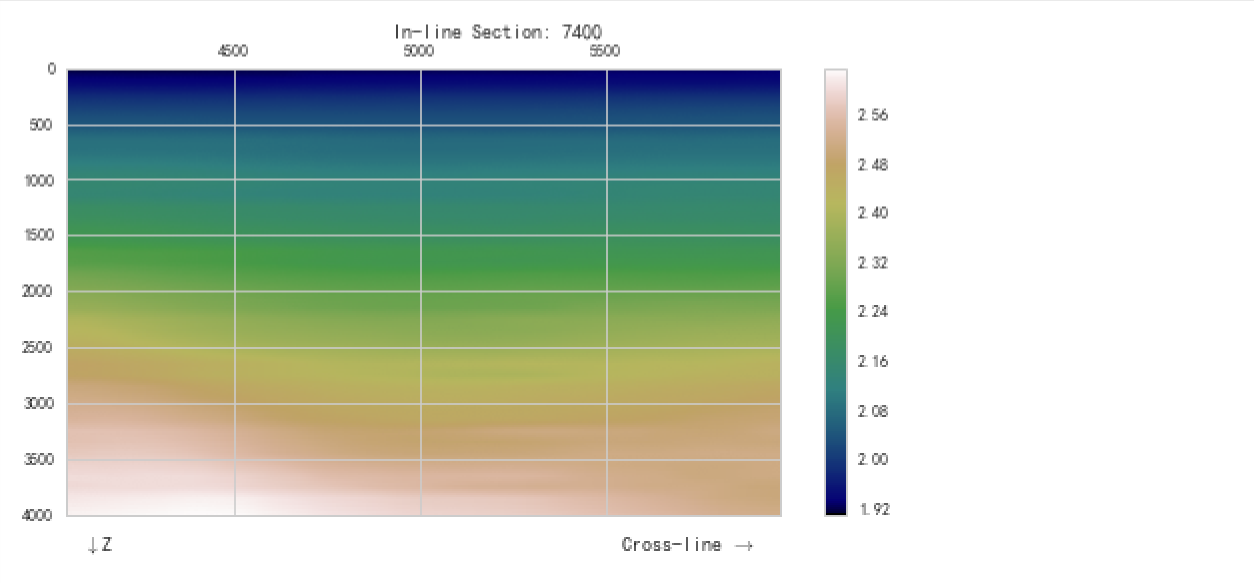
Caculate density using Gardner equation from velocity:

[5]:
```
den_cube = ppp.gardner_seis("den_from_vel", vel_cube)
```

View 2D section of computed density cube:

[11]:
```
fig_den, ax_den = plt.subplots()

im = den_cube.plot(ppp.InlineIndex(7400), ax_den, kind='img', cm='gist_earth')
fig_den.colorbar(im)
fig_den.set(figwidth=8)
```

[11]:
```
[None]
```



2. Calculation of Overburden Pressure

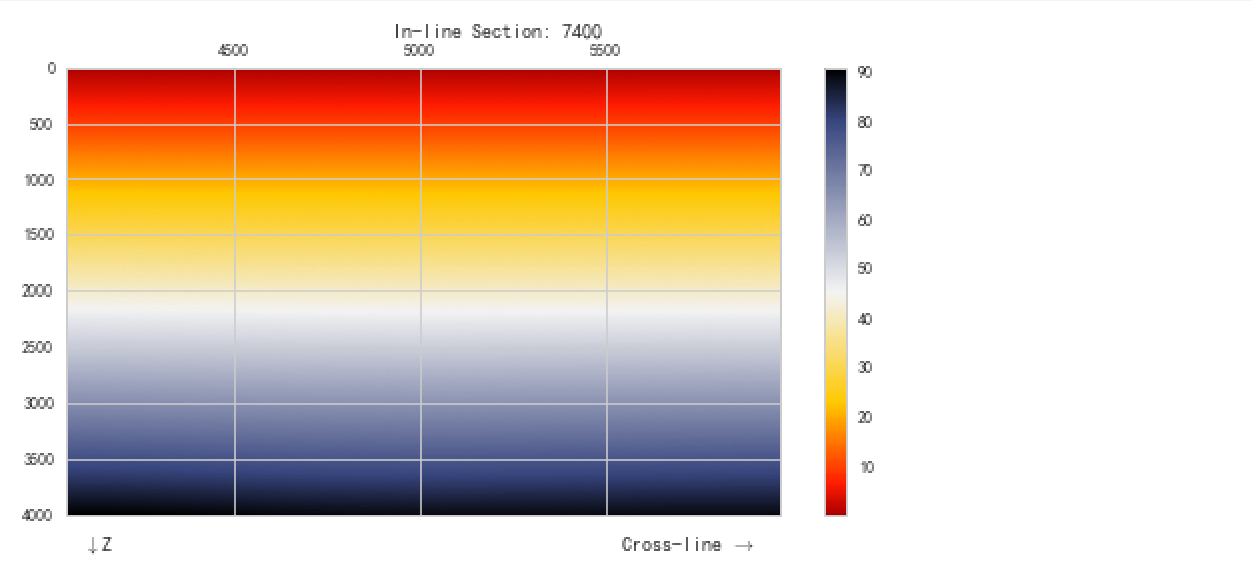
[9]:
```
obp_cube = ppp.obp_seis("obp_new", den_cube)
```

View calculated OBP section:

Here use a colormap defined in OpenDtect.

[12]:
```
from pygeopressure.basic.vawt import opendtect_seismic_colormap

fig_obp, ax_obp = plt.subplots()

im = obp_cube.plot(ppp.InlineIndex(7400), ax_obp, kind='img', cm=opendtect_seismic_
↪colormap())

fig_obp.colorbar(im)
fig_obp.set(figwidth=8)
```

[12]:
```
[None]
```



5.3.6 Eaton Method with Seismic Velocity Data

[2]:
```
from __future__ import print_function, division, unicode_literals
%matplotlib inline
import matplotlib.pyplot as plt

plt.style.use(['seaborn-paper', 'seaborn-whitegrid'])
plt.rcParams['font.sans-serif']=['SimHei']
plt.rcParams['axes.unicode_minus']=False

import numpy as np

import pygeopressure as ppp
```

Create survey CUG:

[3]:
```
# set to the directory on your computer
SURVEY_FOLDER = "M:/CUG_depth"

survey = ppp.Survey(SURVEY_FOLDER)
```

Retrieve well CUG1:

[4]:
```
well_cug1 = survey.wells['CUG1']
```

Get a, b from well CUG1:

[5]:
```
a = well_cug1.params['nct']["a"]
b = well_cug1.params['nct']["b"]
```

Get n from well CUG1:

[6]:
```
n = well_cug1.params['n']
```

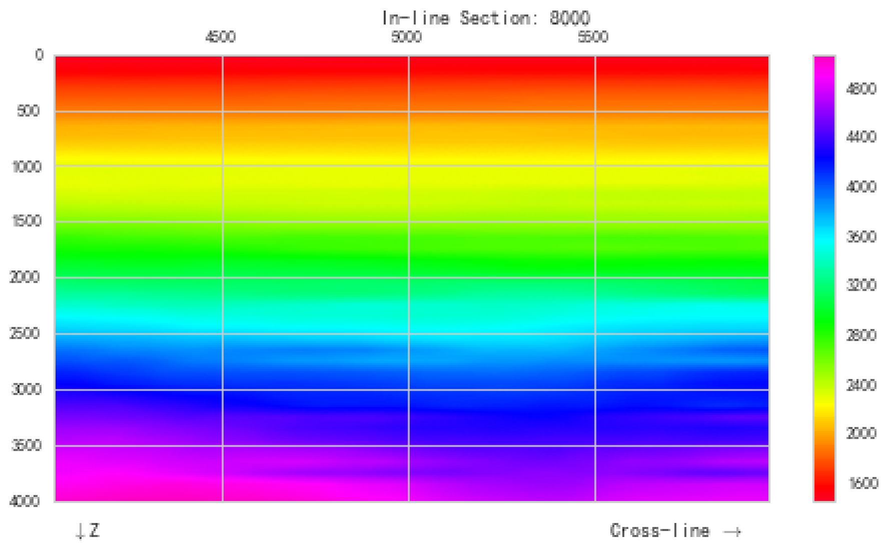
Retrieve seismic data:

[7]:
```
vel_cube = survey.seismics['velocity']
obp_cube = survey.seismics['obp_new']
```

View velocity section:

[8]:
```
fig_vel, ax_vel = plt.subplots()

im = vel_cube.plot(
    ppp.InlineIndex(8000), ax_vel, kind='img', cm='gist_rainbow')
fig_vel.colorbar(im)
fig_vel.set(figwidth=8)
```

[8]:
```
[None]
```



Pressure Prediction with Eaton method:

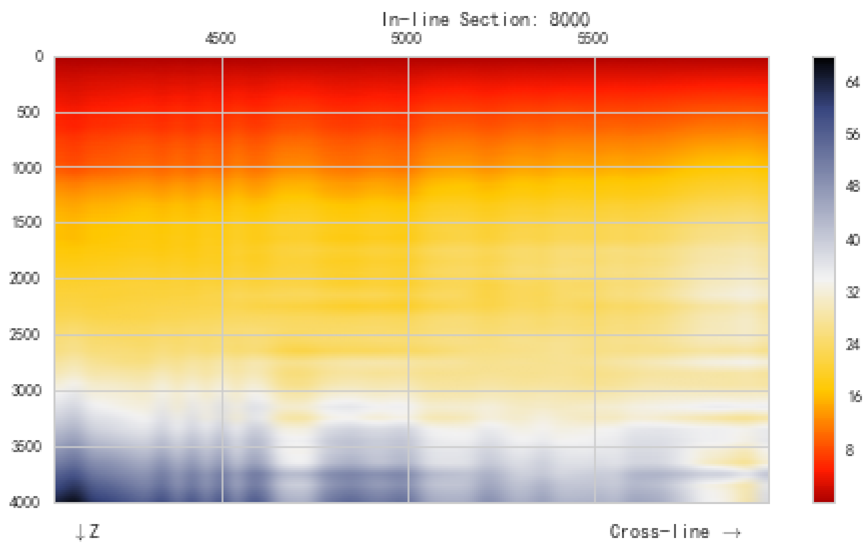
[9]:
```
eaton_cube = ppp.eaton_seis(
    "eaton_new", obp_cube, vel_cube, n=3,
    upper=survey.horizons['T16'], lower=survey.horizons['T20'])
```

`eaton_seis` function will automatically optimize the coefficients of Normal Compaction Trend, `a` and `b`.

View calculated pressure:

[10]:
```
from pygeopressure.basic.vawt import opendtect_seismic_colormap

fig_pres, ax_pres = plt.subplots()

im = eaton_cube.plot(
    ppp.InlineIndex(8000), ax_pres,
    kind='img', cm=opendtect_seismic_colormap())

fig_pres.colorbar(im)
fig_pres.set(figwidth=8)
```

[10]:
```
[None]
```



5.3.7 Bowers method with seismic velocity

Pore pressure prediction with Bowers' method using well log data.

[3]:
```
from __future__ import print_function, division, unicode_literals
%matplotlib inline
import matplotlib.pyplot as plt

plt.style.use(['seaborn-paper', 'seaborn-whitegrid'])
plt.rcParams['font.sans-serif']=['SimHei']
plt.rcParams['axes.unicode_minus']=False

import numpy as np

import pygeopressure as ppp
```

Create survey CUG:

[5]:
```
# set to the directory on your computer
SURVEY_FOLDER = "M:/CUG_depth"

survey = ppp.Survey(SURVEY_FOLDER)
```

Retrieve well CUG1:

[6]:
```
well_cug1 = survey.wells['CUG1']
```

Get Bowers coefficients A, B from well CUG1:

[7]:
```
a = well_cug1.params['bowers']["A"]
b = well_cug1.params['bowers']['B']
```

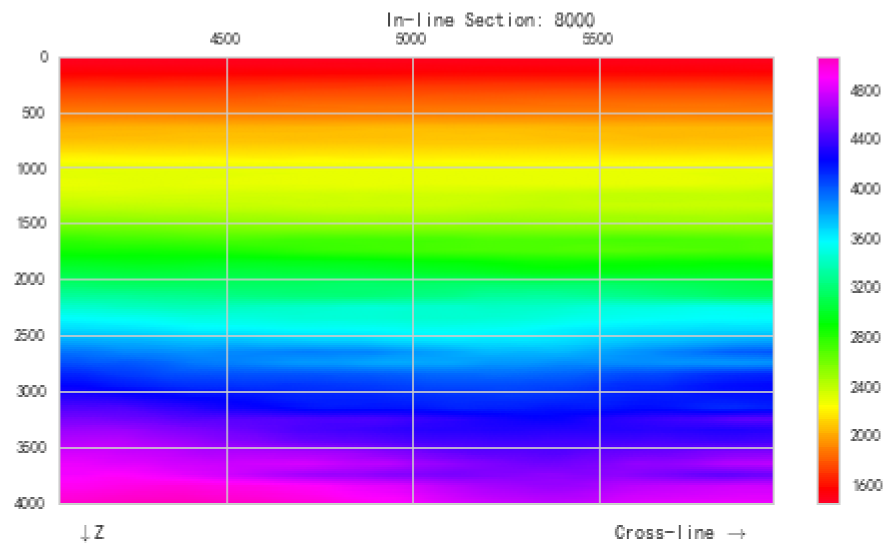
Retrieve seismic data:

[8]:
```
vel_cube = survey.seismics['velocity']
obp_cube = survey.seismics['obp_new']
```

View velocity section:

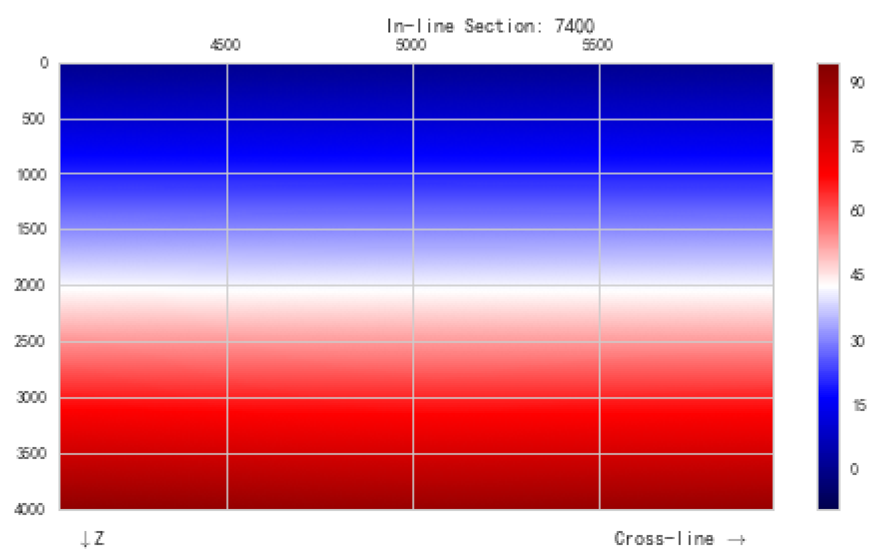
[9]:
```
fig_vel, ax_vel = plt.subplots()

im = vel_cube.plot(
    ppp.InlineIndex(8000), ax_vel, kind='img', cm='gist_rainbow')
fig_vel.colorbar(im)
fig_vel.set(figwidth=8)
```

[9]:
```
[None]
```



[10]:
```
fig, ax = plt.subplots()
im = obp_cube.plot(ppp.InlineIndex(7400), ax, kind='img')
fig.colorbar(im)
fig.set(figwidth=8)
```

[10]:
```
[None]
```



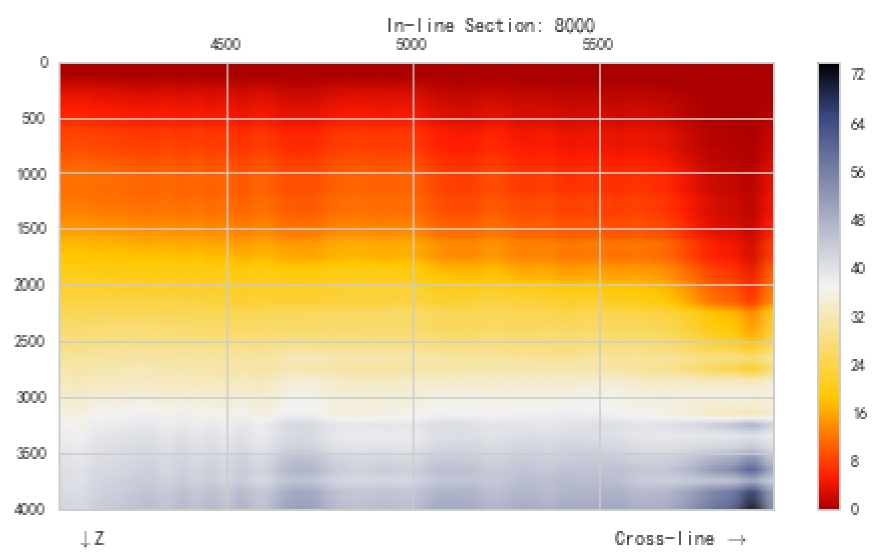
Pressure Prediction with Bowers method:

[11]:
```
bowers_cube = ppp.bowers_seis(
    "bowers_new", obp_cube, vel_cube,
    upper=survey.horizons['T16'], lower=survey.horizons['T20'],
    mode='optimize')
```

View predicted pressure section:

[12]:
```
from pygeopressure.basic.vawt import opendtect_seismic_colormap

fig_pres, ax_pres = plt.subplots()

im = bowers_cube.plot(
    ppp.InlineIndex(8000), ax_pres,
    kind='img', cm=opendtect_seismic_colormap())

fig_pres.colorbar(im)
fig_pres.set(figwidth=8)
```

[12]:
```
[None]
```



5.4 Survey Setup

A geophysical survey is a data set measured and recorded with reference to a particular area of the Earth's surface[1]. Survey is the basic management unit for projects in pyGeoPressure. It holds both survey geometry and references to seismic and well data associated with the survey area.

In `pyGeoPressure`, a `Survey` object is initialized with a survey folder

```
import pygeopressure as ppp

survey = ppp.Survey('path/to/survey/folder')
```

So to setup a survey is to build the survey folder.

[1] http://www.glossary.oilfield.slb.com/en/Terms/s/survey.aspx

5.4.1 Survey Folder Structure

In pyGeoPressure, all information and data are stored in a survey folder with the following structure:

```
+---EXAMPLE_SURVEY
|   .survey
|   |
|   +---Seismics
|   |       velocity.seis
|   |       density.seis
|   |       pressure.seis
|   |
|   +---Surfaces
|   |       T20.hor
|   |       T16.hor
|   |
|   \---Wellinfo
|           .CUG1
|           .CUG2
|           well_data.h5
|
```

Within the survey directory named `EXAMPLE_SURVEY`, there are three sub-folders `Seismics`, `Surfaces` and `Wellinfo`. At the root of the survey folder is a survey definition file `.survey`. The definition file defines the geometry of the survey, the folder structure defines its association with the data.

5.4.2 .survey

First and foremost, there is a `.survey` file, which stores geometry definition of the whole geophysical survey and auxiliary information.

Geometry Definition

Survey Geometry defines:

1. Survey Area extent
2. Inline/Crossline Coordinates, along which the survey are conducted.
3. X/Y Coordinates, real world Coordinates
4. Relations between them

In pyGeoPressure, survey geometry is defined using a method I personally dubbed "Three points" method. Given the inline/crossline number and X/Y coordinates of three points on the survey grid, we are able to solve the linear equaions for transformation between inline/crossline coordination and X/Y coordination.

Information in `.survey` file of the example survey are

```
{
    "name": "CUG_depth",
    "point_A": [6400, 4100, 701319, 3274887],
    "point_B": [6400, 4180, 702185, 3274387],
    "point_C": [6440, 4180, 702685, 3275253],
    "inline_range": [6400, 7000, 20],
    "crline_range": [4100, 6020, 40],
    "z_range": [0, 5000, 4, "m"]
}
```

Of the three points selected, point A and point B share the same inline, and point B and point C share the same crossline.

In addition to coordinations of three points, the extent and step of inline, crossline ,z coordinates and unit of z are also needed to fully define the extent of the survey.

Seismics

Within `Seismics` folder, each file written in JSON with extention `.seis` represents a seismic data cube. These files contain file path to the actual SEG-Y file storing seismic data, and the type of property (`Property_Type`) and wether data is in depth scale or not (`inDepth`). So the `Seismics` folder doesn't need to store large SEG-Y files, it just holds references to them. In-line/cross-line range and Z range are also written in these files.

Surfaces

Surfaces like seimic horizons are stored in `Surfaces` folder. Surface files ending with `.hor` are tsv files storing inline number, crossline number and depth values defining the geometry of a 3D geologic surface.

Wellinfo

Well information is stored in `Wellinfo`. Each file with file name with extention `.well` is a well information file, it stores well position information like coordination, kelly bushing and interpretation information like interpretated layers, fitted coefficients. It also holds a pointer to where the log curve data is stored. By default, well log curve data are stored in `well_data.h5`, but users can point to other storage files.

5.4.3 Create New Survey

A helper function `create_survey_directory` can facilitate users to build survey folder structure:

```
import pygeopressure as ppp

ppp.create_survey_directory(ROOTDIR, SURVEY_NAME)
```

It will create a survey folder named `SURVEY_NAME` in `ROOTDIR` with three sub-folders for each kind of data and a `.survey` file.

5.5 Add Well

Adding a new well is acheived by add a `.well` file to `Wellinfo` folder in survey directory(see *Survey Setup*).

A minimal `.well` should contain the following information: 1. "well_name" 2. "loc" - X/Y coordination of the well 3. "KB" - kelly bushing elevation 4. "WD" - water depth 5. "TD" - total depth of the wellbore 6. "hdf_file" - storage file path, if only the file name is provided, pyGeoPressure will assume it is in the `Wellinfo` folder.

```
{
    "well_name": "CUG1",
    "loc": [
        707838,
        3274780
```

(continues on next page)

(continued from previous page)

```
    ],
    "KB": 23,
    "WD": 85,
    "TD": 5000,
    "hdf_file": "well_data.h5"
}
```

More information can be stored in well information file. Please check out the `CUG1.well` in the example surey.

5.6 Import Well Log Curve Data from file

First, say we have a new well `CUG3`. After writing the `CUG3.well` file, and save it to `Wellinfo` folder. We initialize the survey.

[2]:
```
survey = ppp.Survey(Path(SURVEY_FOLDER))
```

```
'No well named cug3'
```

A meassage shows "no well named `cug3`", it's because there is no data stored in `well_data.h5` file. (In `pyGeoPressure`, well log data is stored in hdf5 file.)

But we can still get its information:

[38]:
```
survey.wells
```

[38]:
```
{u'CUG1': <pygeopressure.basic.well.Well at 0x11031588>,
 u'CUG3': <pygeopressure.basic.well.Well at 0x10a0a8d0>}
```

[5]:
```
cug3 = survey.wells['CUG3']
```

We provides two methods for importing well log curve data:

5.6.1 0. Read las/pseudo-las file

First, we need to read data from file. `pyGeoPressure` provides a class `LasData` for reading las file and pseudo-las file.

[7]:
```
las_data = ppp.LasData(las_file="C:/Users/yuhao/Desktop/CUG_depth/log_curves.las")
```

get las file type with:

[11]:
```
las_data.file_type
```

[11]:
```
'pseudo-las'
```

Read pseudo-las file with:

[12]:
```
las_data.read_pseudo_las()
```

5.6.2 1. at runtime

At runtime, well log curve data are stored in pandas dataframe. Each `Well` object has a `dataframe` attribute. To import well log curve to `Well`, users can directly set a new dataframe read by `LasData` to `Well.dataframe`.

```
[14]: cug3 = survey.wells['CUG3']
```

Set the data_frame of `LasData` to `Well`:

```
[19]: cug3.data_frame = las_data.data_frame
```

```
[20]: cug3.logs
```

```
[20]: [u'Shale_Volume',
       u'Density',
       u'Density_filter20_sm1500',
       u'Velocity',
       u'Velocity_filter20_sm1500',
       u'Overburden_Pressure',
       u'Porosity']
```

Or part of the read dataframe:

```
[25]: cug3.data_frame = las_data.data_frame[['Depth(m)', 'Shale_Volume(Fraction)',
      ↪'Density(G/C3)']]
```

```
[26]: cug3.logs
```

```
[26]: [u'Shale_Volume', u'Density']
```

After importing logs, users should call ``save_well_logs()`` to save them to storage file.

5.6.3 2. edit `.hd5` file

In `pyGeoPressure`, well log curve data is stored in hdf5 files (I call it storage file) on disk. To import well log curves, users can directly manage hdf5 file.

`pyGeoPressure` provides class `WellStorage` to manage hdf5 files.

```
[27]: storage = ppp.WellStorage(hdf5_file='C:/Users/yuhao/Desktop/CUG_depth/Wellinfo/well_
      ↪data.h5')
```

```
[28]: storage.add_well(well_name='cug3', well_data_frame=las_data.data_frame)
```

```
[29]: storage.wells
```

```
[29]: ['cug1', 'cug2', 'cug3']
```

When we reinitialize the survey, we will see that the data has been imported into Well `CUG3`.

```
[34]: survey = ppp.Survey(Path(SURVEY_FOLDER))
```

```
[35]: cug3 = survey.wells['CUG3']
```

```
[36]: cug3.logs
```

```
[36]: [u'Shale_Volume',
 u'Density',
 u'Density_filter20_sm1500',
 u'Velocity',
 u'Velocity_filter20_sm1500',
 u'Overburden_Pressure',
 u'Porosity']
```

5.7 Adding Seismic Cube and Surface

5.7.1 Add Seismic Cube

Seismic cube is added by creating a new `.seis` file in `Seismics` folder.

The content of *velocity.seis* file in our example survey are:

```
{
    "path": "velocity.sgy",
    "inline_range": [200, 650, 2],
    "z_range": [400, 1100, 4],
    "crline_range": [700, 1200, 2],
    "inDepth": true,
    "Property_Type": "Velocity"
}
```

Note that if path is relative, pygeopressure will look for segy file in the Seismics folder.

5.7.2 Add Horizon

Horizons can added by placing the horizon data file with extention `.hor` in `Surfaces` folder.

Horizon files are tsv(Tab Seperated Value) files with three columns each stores inline, crossline and Z value.

It header should be:

```
inline  crline  z
```

5.8 Data Types

Three basic Data types in `pyGeoPressure` are `Well` for well, `Log` for well log and `SeiSEGY` for seismic data.

5.8.1 Well

class pygeopressure.basic.well.**Well**(*json_file*, *hdf_path=None*)
A class representing a well with information and log curve data.

Initializer:

Well.**__init__**(*json_file*, *hdf_path=None*)

Parameters

- **json_file** (*str*) – path to parameter file
- **hdf_path** (*str, optional*) – path to hdf5 file used to override the one written in json_file

Properties

Well.**depth**()
depth values of the well

Returns

Return type numpy.ndarray

Well.**logs**()
logs stored in this well

Returns

Return type list

Well.**unit_dict**()
properties and their units

Well.**hydrostatic**()
Hydrostatic Pressure

Returns

Return type numpy.ndarray

Well.**lithostatic**()
Overburden Pressure (Lithostatic)

Returns

Return type numpy.ndarray

Well.**hydro_log**()

Returns Hydrostatic Pressure

Return type *Log*

Well.**normal_velocity**()
Normal Velocity calculated using NCT stored in well

Returns

Return type numpy.ndarray

log curve data manipulation

`Well.`**`get_log`**(*logs*, *ref=None*)

Retreive one or several logs in well

Parameters

- **logs** (*str or list str*) – names of logs to be retrieved
- **ref** (*{'sea', 'kb'}*) – depth reference, 'sea' references to sea level, 'kb' references to Kelly Bushing

Returns one or a list of Log objects

Return type *Log*

`Well.`**`add_log`**(*log*, *name=None*, *unit=None*)

Add new Log to current well

Parameters

- **log** (*Log*) – log to be added
- **name** (*str, optional*) – name for the newly added log, None, use log.name
- **unit** (*str, optional*) – unit for the newly added log, None, use log.unit

`Well.`**`drop_log`**(*log_name*)

delete a Log in current Well

Parameters **log_name** (*str*) – name of the log to be deleted

`Well.`**`rename_log`**(*log_name*, *new_log_name*)

Parameters

- **log_name** (*str*) – log name to be replaced
- **new_log_name** (*str*)

`Well.`**`update_log`**(*log_name*, *log*)

Update well log already in current well with a new Log

Parameters

- **log_name** (*str*) – name of the log to be replaced in current well
- **log** (*Log*) – Log to replace

`Well.`**`to_las`**(*file_path*, *logs_to_export=None*, *full_las=False*, *null_value=1e+30*)

Export logs to LAS or pseudo-LAS file

Parameters

- **file_path** (*str*) – output file path
- **logs_to_export** (*list of str*) – Log names to be exported, None export all logs
- **full_las** (*bool*) – True, export LAS header; False export only data hence psuedo-LAS
- **null_value** (*scalar*) – Null Value representation in output file.

`Well.`**`save_well_logs`**()

Save current well logs to file

Get Meassured pyGeoPressure

`Well.get_pressure`(*pres_key*, *ref=None*, *hydrodynamic=0*, *coef=False*)

Get Pressure Values or Pressure Coefficients

Parameters

- **pres_key** (*str*) – Pressure data name
- **ref** (*{'sea', 'kb'}*) – depth reference, 'sea' references to sea level, 'kb' references to Kelly Bushing
- **hydrodynamic** (*float*) – return Pressure at depth deeper than this value
- **coef** (*bool*) – True - get pressure coefficient else get pressure value

Returns Log object containing Pressure or Pressure coefficients

Return type *Log*

`Well.get_pressure_normal`()

return pressure points within normally pressured zone.

Returns Log object containing normally pressured measurements

Return type *Log*

Pressure Prediction

`Well.eaton`(*vel_log*, *obp_log=None*, *n=None*, *a=None*, *b=None*)

Predict pore pressure using Eaton method

Parameters

- **vel_log** (*Log*) – velocity log
- **obp_log** (*Log*) – overburden pressure log
- **n** (*scalar*) – Eaton exponent

Returns a Log object containing calculated pressure.

Return type *Log*

`Well.bowers`(*vel_log*, *obp_log=None*, *a=None*, *b=None*, *u=None*, *vmax=None*, *start_depth=None*, *buf=20*, *end_depth=None*, *end_buffer=10*)

Predict pore pressure using Eaton method

Parameters

- **vel_log** (*Log*) – velocity log
- **obp_log** (*Log*) – overburden pressure log
- **a, b, u** (*float*) – bowers model coefficients

Returns a Log object containing calculated pressure.

Return type *Log*

`Well.multivariate`(*vel_log*, *por_log*, *vsh_log*, *obp_log=None*, *a0=None*, *a1=None*, *a2=None*, *a3=None*, *b=None*)

Other

Well.**plot_horizons**(*ax*, *color_dict=None*)
Plot horizons stored in well

Well.**save_params**()
Save edited parameters to well information file

5.8.2 Log

class pygeopressure.basic.well_log.**Log**(*file_name=None*, *log_name='unk'*)
class for well log data

Initializer:

Log.**__init__**(*file_name=None*, *log_name='unk'*)

Parameters

- **file_name** (*str*) – pseudo las file path
- **log_name** (*str*) – log name to create

Alternative initializer:

classmethod Log.**from_scratch**(*depth*, *data*, *name=None*, *units=None*, *descr=None*, *prop_type=None*)

Data interfaces:

Log.**depth**()
depth data of the log

Log.**data**()
property data of the log

Log.**start**()
start depth of available property data

Log.**stop**()
end depth of available property data

Log.**start_idx**()
start index of available property data

Log.**stop_idx**()
end index of available property data

Log.**top**()
top depth of this log

Log.**bottom**()
bottom depth of this log

Plot:

`Log.plot(ax=None, color='gray', linewidth=0.5, linestyle='-', label=None, zorder=1)`
Plot log curve

Parameters **ax** (*matplotlib.axes._subplots.AxesSubplot*) – axis object to plot on, a new axis will be created if not provided

Returns

Return type matplotlib.axes._subplots.AxesSubplot

Others:

`Log.to_las(file_name)`
Save as pseudo-las file

`Log.get_data(depth)`
get data at certain depth

`Log.get_depth_idx(d)`
return index of depth

`Log.get_resampled(rate)`
return resampled log

5.8.3 SeiSEGY

Initializers:

The default initializer takes a segy file path:

The alternative initilizer `from_json` takes a info file in json.

Iterators:

Data interface:

Plots Data Sections:

Others:

Note: Internally, `pyGeoPressrue` interacts with SEGY file utlizing segyio.

5.9 API

5.9.1 pygeopressure package

Subpackages

pygeopressure.basic package

Submodules

pygeopressure.basic.horizon module

class Horizon for accessing horizon

Created on Fri July 20 2017

class pygeopressure.basic.horizon.**Horizon**(*data_file*)
Bases: object

Horizon using excel file as input

Parameters **data_file** (*str*) – path to excel data file

get_cdp(*cdp*)
Get value for a CDP point on the horizon.

Parameters **cdp** (*tuple of int (inline, crossline)*)

pygeopressure.basic.indexes module

class for survey index definition

created on Jun 10th 2017

class pygeopressure.basic.indexes.**CdpIndex**(*cdp*)
Bases: *pygeopressure.basic.indexes.SurveyIndex*

class pygeopressure.basic.indexes.**CrlineIndex**(*value*)
Bases: *pygeopressure.basic.indexes.SurveyIndex*

class pygeopressure.basic.indexes.**DepthIndex**(*value*)
Bases: *pygeopressure.basic.indexes.SurveyIndex*

class pygeopressure.basic.indexes.**InlineIndex**(*value*)
Bases: *pygeopressure.basic.indexes.SurveyIndex*

class pygeopressure.basic.indexes.**SurveyIndex**(*value*)
Bases: object

pygeopressure.basic.las module

an interface for interacting with Las file

Created on Thu May 10 2018

class pygeopressure.basic.las.**LasData**(*las_file*)

Bases: object

Class for reading LAS and pseudo-LAS file data

null_values could be set to more values in order to deal with messy files

property data_frame

property file_type

find_logs()

property logs

read_las()

read_pseudo_las()

property units

pygeopressure.basic.log_tools module

pygeopressure.basic.optimizer module

pygeopressure.basic.plots module

a Well class utilizing pandas DataFrame and hdf5 storage

Created on May 27 2018

class pygeopressure.basic.plots.**LoadingPlot**(*ax*, *obp_logs*, *vel_logs*, *pres_logs*, *well_names*)

Bases: object

Parameters **json_file** (*str*) – path to parameter file

check_error(*obp_log*, *vel_log*, *pres_log*)

error_sigma()

fit()

plot()

pygeopressure.basic.plots.**plot_bowers_unloading**(*ax*, *a*, *b*, *u*, *vmax*, *well*, *vel_log*, *obp_log*, *pres_log='unloading'*)

plot bowers unloading plot

pygeopressure.basic.plots.**plot_bowers_vrigin**(*ax*, *a*, *b*, *well*, *vel_log*, *obp_log*, *upper*, *lower*, *pres_log='loading'*, *mode='nc'*, *nnc=5*)

pygeopressure.basic.plots.**plot_eaton_error**(*ax*, *well*, *vel_log*, *obp_log*, *a*, *b*, *pres_log='loading'*)

pygeopressure.basic.plots.**plot_multivariate**(*axes*, *well*, *vel_log*, *por_log*, *vsh_log*, *obp_log*, *upper*, *lower*, *a0*, *a1*, *a2*, *a3*, *B*)

pygeopressure.basic.seisegy module

pygeopressure.basic.survey module

pygeopressure.basic.survey_setting module

A survey setting class

Created on Sat Jan 20 2018

class pygeopressure.basic.survey_setting.**SurveySetting**(*threepoints*)

Bases: object

class to hold survey settings and compute additional coordination property

static angle(*x*, *y*)

Return angle from 0 to pi

x : tuple y : tuple

azimuth_and_invertedAxis()

Determine azimuth (Crossline axis direction from Coordination North) and Inline axis is positive to the right (invertedAxis=False) or to the left (invertedAxis=True)

coord_2_line(*coordinate*)

draw_survey_line(*ax*)

four_corner_on_canvas(*canvas_width*, *canvas_height*, *scale_factor=0.8*)

get the coordinaiton of four corners of survey area on canvas

line_2_coord(*inline*, *crline*)

pygeopressure.basic.threepoints module

Created on Feb. 14th 2018

exception pygeopressure.basic.threepoints.**Invalid_threepoints_Exception**(*message=None*)

Bases: Exception

exception pygeopressure.basic.threepoints.**Not_threepoints_v1_Exception**(*message=None*)

Bases: Exception

exception pygeopressure.basic.threepoints.**Not_threepoints_v2_Exception**(*message=None*)

Bases: Exception

class pygeopressure.basic.threepoints.**ThreePoints**(*json_file=None*)

Bases: object

inline, crossline and z coordinates of three points in survey

pygeopressure.basic.utils module

some utilities

`pygeopressure.basic.utils.`**`methdispatch`**(*func*)

`pygeopressure.basic.utils.`**`nmse`**(*measure*, *predict*)
Normalized Root-Mean-Square Error

with RMS(y - y*) as nominator, and MEAN(y) as denominator

`pygeopressure.basic.utils.`**`pick_sparse`**(*a_array*, *n*)
Pick n equally spaced samples from array

Parameters

- **a_array** (*1-d ndarray*)
- **n** (*int*) – number of samples to pick

`pygeopressure.basic.utils.`**`rmse`**(*measure*, *predict*)
Relative Root-Mean-Square Error

with RMS(y - y*) as nominator, and RMS(y) as denominator

`pygeopressure.basic.utils.`**`split_sequence`**(*sequence*, *length*)
Split a sequence into fragments with certain length

pygeopressure.basic.vawt module

Created on Thu Apr 26 2017

`class` `pygeopressure.basic.vawt.`**`Wiggles`**(*data*, *wiggleInterval=10*, *overlap=1*, *posFill='black'*, *negFill=None*, *lineColor='black'*, *rescale=True*, *extent=None*, *ax=None*)

Bases: `object`

`wiggle`(*values*)
Plot a trace in VAWT(Variable Area Wiggle Trace)

`wiggles`()
2-D Wiggle Trace Variable Amplitude Plot

`pygeopressure.basic.vawt.`**`img`**(*data*, *extent*, *ax*, *cm='seismic'*, *ptype='seis'*)

`pygeopressure.basic.vawt.`**`opendtect_seismic_colormap`**()

`pygeopressure.basic.vawt.`**`wiggle`**(*values*, *origin=0*, *posFill='black'*, *negFill=None*, *lineColor='black'*, *resampleRatio=10*, *rescale=False*, *zmin=0*, *zmax=None*, *ax=None*)
Plot a trace in VAWT(Variable Area Wiggle Trace)

Parameters

- **x** (*input data (1D numpy array)*)
- **origin** (*(default, 0) value to fill above or below (float)*)
- **posFill** (*(default, black)*) – color to fill positive wiggles with (string or None)
- **negFill** (*(default, None)*) – color to fill negative wiggles with (string or None)
- **lineColor** (*(default, black)*) – color of wiggle trace (string or None)

- **resampleRatio** (*(default, 10)*) – factor to resample traces by before plotting (1 = raw data) (float)
- **rescale** (*(default, False)*) – If True, rescale "x" to be between -1 and 1
- **zmin** (*(default, 0)*) – The minimum z to use for plotting
- **zmax** (*(default, len(x))*) – The maximum z to use for plotting
- **ax** (*(default, current axis)*) – The matplotlib axis to plot onto

Returns

Return type Plot

`pygeopressure.basic.vawt.`**`wiggles`**(*data*, *wiggleInterval=10*, *overlap=5*, *posFill='black'*, *negFill=None*, *lineColor='black'*, *rescale=True*, *extent=None*, *ax=None*)

2-D Wiggle Trace Variable Amplitude Plot

Parameters

- **x** (*input data (2D numpy array)*)
- **wiggleInterval** (*(default, 10) Plot 'wiggles' every wiggleInterval traces*)
- **overlap** (*(default, 0.7) amount to overlap 'wiggles' by (1.0 = scaled*) – to wiggleInterval)
- **posFill** (*(default, black) color to fill positive wiggles with (string*) – or None)
- **negFill** (*(default, None) color to fill negative wiggles with (string*) – or None)
- **lineColor** (*(default, black) color of wiggle trace (string or None)*)
- **resampleRatio** (*(default, 10) factor to resample traces by before*) – plotting (1 = raw data) (float)
- **extent** (*(default, (0, nx, 0, ny)) The extent to use for the plot.*) – A 4-tuple of (xmin, xmax, ymin, ymax)
- **ax** (*(default, current axis) The matplotlib axis to plot onto.*)
- **Output** – a matplotlib plot on the current axes

pygeopressure.basic.well module

a Well class utilizing pandas DataFrame and hdf5 storage

Created on Tue Dec 27 2016

class `pygeopressure.basic.well.`**`Well`**(*json_file*, *hdf_path=None*)

Bases: `object`

A class representing a well with information and log curve data.

`add_log`(*log*, *name=None*, *unit=None*)

Add new Log to current well

Parameters

- **log** (*Log*) – log to be added
- **name** (*str, optional*) – name for the newly added log, None, use log.name
- **unit** (*str, optional*) – unit for the newly added log, None, use log.unit

bowers (*vel_log*, *obp_log=None*, *a=None*, *b=None*, *u=None*, *vmax=None*, *start_depth=None*, *buf=20*, *end_depth=None*, *end_buffer=10*)
Predict pore pressure using Eaton method

Parameters

- **vel_log** (*Log*) – velocity log
- **obp_log** (*Log*) – overburden pressure log
- **a, b, u** (*float*) – bowers model coefficients

Returns a Log object containing calculated pressure.

Return type *Log*

property depth
depth values of the well

Returns

Return type numpy.ndarray

drop_log (*log_name*)
delete a Log in current Well

Parameters **log_name** (*str*) – name of the log to be deleted

eaton (*vel_log*, *obp_log=None*, *n=None*, *a=None*, *b=None*)
Predict pore pressure using Eaton method

Parameters

- **vel_log** (*Log*) – velocity log
- **obp_log** (*Log*) – overburden pressure log
- **n** (*scalar*) – Eaton exponent

Returns a Log object containing calculated pressure.

Return type *Log*

get_log (*logs*, *ref=None*)
Retreive one or several logs in well

Parameters

- **logs** (*str or list str*) – names of logs to be retrieved
- **ref** (*{'sea', 'kb'}*) – depth reference, 'sea' references to sea level, 'kb' references to Kelly Bushing

Returns one or a list of Log objects

Return type *Log*

get_pressure (*pres_key*, *ref=None*, *hydrodynamic=0*, *coef=False*)
Get Pressure Values or Pressure Coefficients

Parameters

- **pres_key** (*str*) – Pressure data name
- **ref** (*{'sea', 'kb'}*) – depth reference, 'sea' references to sea level, 'kb' references to Kelly Bushing
- **hydrodynamic** (*float*) – return Pressure at depth deeper than this value

- **coef** (*bool*) – True - get pressure coefficient else get pressure value

Returns Log object containing Pressure or Pressure coefficients

Return type *Log*

get_pressure_normal()
return pressure points within normally pressured zone.

Returns Log object containing normally pressured measurements

Return type *Log*

hydro_log()

Returns Hydrostatic Pressure

Return type *Log*

property hydrostatic
Hydrostatic Pressure

Returns

Return type numpy.ndarray

property lithostatic
Overburden Pressure (Lithostatic)

Returns

Return type numpy.ndarray

property logs
logs stored in this well

Returns

Return type list

multivariate(*vel_log*, *por_log*, *vsh_log*, *obp_log=None*, *a0=None*, *a1=None*, *a2=None*, *a3=None*, *b=None*)

property normal_velocity
Normal Velocity calculated using NCT stored in well

Returns

Return type numpy.ndarray

plot_horizons(*ax*, *color_dict=None*)
Plot horizons stored in well

rename_log(*log_name*, *new_log_name*)

Parameters

- **log_name** (*str*) – log name to be replaced
- **new_log_name** (*str*)

save_params()
Save edited parameters to well information file

save_well_logs()
Save current well logs to file

to_las(*file_path*, *logs_to_export=None*, *full_las=False*, *null_value=1e+30*)
Export logs to LAS or pseudo-LAS file

Parameters

- **file_path** (*str*) – output file path
- **logs_to_export** (*list of str*) – Log names to be exported, None export all logs
- **full_las** (*bool*) – True, export LAS header; False export only data hence psuedo-LAS
- **null_value** (*scalar*) – Null Value representation in output file.

property unit_dict
properties and their units

update_log (*log_name*, *log*)
Update well log already in current well with a new Log

Parameters

- **log_name** (*str*) – name of the log to be replaced in current well
- **log** (*Log*) – Log to replace

pygeopressure.basic.well_log module

class Log for well log data

Created on Fri Apr 18 2017

class pygeopressure.basic.well_log.**Log** (*file_name=None*, *log_name='unk'*)
Bases: object

class for well log data

property bottom
bottom depth of this log

property data
property data of the log

property depth
depth data of the log

classmethod from_scratch (*depth*, *data*, *name=None*, *units=None*, *descr=None*, *prop_type=None*)

get_data (*depth*)
get data at certain depth

get_depth_idx (*d*)
return index of depth

get_resampled (*rate*)
return resampled log

plot (*ax=None*, *color='gray'*, *linewidth=0.5*, *linestyle='-'*, *label=None*, *zorder=1*)
Plot log curve

Parameters **ax** (*matplotlib.axes._subplots.AxesSubplot*) – axis object to plot on, a new axis will be created if not provided

Returns

Return type matplotlib.axes._subplots.AxesSubplot

property start
start depth of available property data

property start_idx
start index of available property data

property stop
end depth of available property data

property stop_idx
end index of available property data

to_las (*file_name*)
Save as pseudo-las file

property top
top depth of this log

pygeopressure.basic.well_storage module

an interface to a hdf5 storage file

Created on Thu May 10 2018

class pygeopressure.basic.well_storage.**WellStorage** (*hdf5_file=None*)
Bases: object

interface to hdf5 file storing well logs

this class is designed to accept only LasData.data_frame as input data

add_well (*well_name, well_data_frame*)

get_well_data (*well_name*)

logs_into_well (*well_name, logs_data_frame*)

remove_well (*well_name*)

update_well (*well_name, well_data_frame*)

property wells

Module contents

pygeopressure.pressure package

Submodules

pygeopressure.pressure.bowers module

Routines to calculate pore pressure

pygeopressure.pressure.bowers.**bowers** (*v, obp, u, start_idx, a, b, vmax, end_idx=None*)
Compute pressure using Bowers equation.

Parameters

- **v** (*1-d ndarray*) – velocity array whose unit is m/s.

- **obp** (*1-d ndarray*) – Overburden pressure whose unit is Pa.
- **v0** (*float, optional*) – the velocity of unconsolidated regolith whose unit is m/s.
- **a** (*float, optional*) – coefficient a
- **b** (*float, optional*) – coefficient b

Notes

$$P = S - \left[\frac{(V - V_0)}{a}\right]^{\frac{1}{b}}$$

[3]

`pygeopressure.pressure.bowers.`**`bowers_varu`**(*v*, *obp*, *u*, *start_idx*, *a*, *b*, *vmax*, *buf=20*, *end_idx=None*, *end_buffer=10*)

Bowers Method with buffer zone above unloading zone

Parameters

- **v** (*1-d ndarray*) – velocity array whose unit is m/s.
- **obp** (*1-d ndarray*) – Overburden pressure whose unit is Pa.
- **u** (*float*) – coefficient u
- **start_idx** (*int*) – index of start of fluid expansion
- **a** (*float, optional*) – coefficient a
- **b** (*float, optional*) – coefficient b
- **vmax** (*float*)
- **buf** (*int, optional*) – len of buffer interval, buf should be smaller than start_idx
- **end_idx** (*int*) – end of fluid expasion
- **end_buffer** (*int*) – len of end buffer interval

`pygeopressure.pressure.bowers.`**`invert_unloading`**(*v*, *a*, *b*, *u*, *v_max*)

invert of Unloading curve in Bowers's method.

`pygeopressure.pressure.bowers.`**`invert_virgin`**(*v*, *a*, *b*)

invert of virgin curve.

`pygeopressure.pressure.bowers.`**`power_bowers`**(*sigma_vc_ratio*, *u*)

`pygeopressure.pressure.bowers.`**`unloading_curve`**(*sigma*, *a*, *b*, *u*, *v_max*)

Unloading curve in Bowers's method.

`pygeopressure.pressure.bowers.`**`virgin_curve`**(*sigma*, *a*, *b*)

Virgin curve in Bowers' method.

[3] Bowers, G. L. (1994). Pore pressure estimation from velocity data: accounting from overpressure mechanisms besides undercompaction: Proceedings of the IADC/SPE drilling conference, Dallas, 1994, (IADC/SPE), 1994, pp 515–530. In International Journal of Rock Mechanics and Mining Sciences & Geomechanics Abstracts (Vol. 31, p. 276). Pergamon.

pygeopressure.pressure.bowers_seis module

pygeopressure.pressure.eaton module

Routines for eaton pressure prediction

Created on Sep 20 2018

`pygeopressure.pressure.eaton.`**`eaton`**(*v*, *vn*, *hydrostatic*, *lithostatic*, *n=3*)

Compute pore pressure using Eaton equation.

Parameters

- **v** (*1-d ndarray*) – velocity array whose unit is m/s.
- **vn** (*1-d ndarray*) – normal velocity array whose unit is m/s.
- **hydrostatic** (*1-d ndarray*) – hydrostatic pressure in mPa
- **lithostatic** (*1-d ndarray*) – Overburden pressure whose unit is mPa.
- **v0** (*float, optional*) – the velocity of unconsolidated regolith whose unit is ft/s.
- **n** (*float, optional*) – eaton exponent

Returns

Return type ndarray

Notes

$$P = S - \sigma_n \left(\frac{V}{V_n}\right)^n$$

[4]

`pygeopressure.pressure.eaton.`**`power_eaton`**(*v_ratio*, *n*)

Notes

$$\frac{\sigma}{\sigma_n} = \left(\frac{V}{V_n}\right)^n$$

`pygeopressure.pressure.eaton.`**`sigma_eaton`**(*es_norm*, *v_ratio*, *n*)

calculate effective pressure with the ratio of velocity and normal velocity

[4] Eaton, B. A., & others. (1975). The equation for geopressure prediction from well logs. In Fall Meeting of the Society of Petroleum Engineers of AIME. Society of Petroleum Engineers.

Notes

$$\sigma = \sigma_n \left(\frac{V}{V_n} \right)^n$$

pygeopressure.pressure.eaton_seis module

pygeopressure.pressure.hydrostatic module

Function to calculate hydrostatic pressure

Created on Fri Nov 11 2016

`pygeopressure.pressure.hydrostatic.`**`hydrostatic_pressure`**(*depth*, *kelly_bushing=0*, *depth_w=0*, *rho_f=1.0*, *rho_w=1.0*)

Parameters

- **depth** (*scalar or 1-d ndarray*) – measured depth, unit: meter
- **rho_f** (*scalar*) – density of pore fluid, g/cm3
- **kelly_bushing** (*scalar*) – kelly bushing elevation, in meter
- **depth_w** (*scalar*) – sea water depth
- **rho_w** (*scalar*) – sea water density

Returns **pressure** – unit: mPa

Return type scalar or 1-d ndarray

`pygeopressure.pressure.hydrostatic.`**`hydrostatic_trace`**(*depth*, *rho=1.01*, *g=9.8*, *shift=0*)

`pygeopressure.pressure.hydrostatic.`**`hydrostatic_well`**(*depth*, *kb=0*, *wd=0*, *rho_f=1.0*, *rho_w=1.0*)

Returns Hydrostatic pressure as a Log

Return type *Log*

pygeopressure.pressure.multivariate module

Routines for multivariate pressure prediction

Created on Sep 20 2018

`pygeopressure.pressure.multivariate.`**`effective_stress_multivariate`**(*vel*, *phi*, *vsh*, *a_0*, *a_1*, *a_2*, *a_3*, *B*, *U*, *vmax*, *start_idx*, *end_idx=None*) `pygeopressure.pressure.multivariate.`**`effective_stress_multivariate_varu`**`(`*vel, phi, vsh, a_0, a_1, a_2, a_3, B, U, vmax, start_idx, buf=20, end_idx=None, end_buffer=10*`)`

`pygeopressure.pressure.multivariate.`**`invert_multivariate_unloading`**`(`*vel, phi, vsh, a_0, a_1, a_2, a_3, B, U, vmax*`)`

Calculate effective stress using multivariate unloading curve

`pygeopressure.pressure.multivariate.`**`invert_multivariate_virgin`**`(`*vel, phi, vsh, a_0, a_1, a_2, a_3, B*`)`

Calculate effective stress using multivariate virgin curve

Parameters

- **vel** (*1-d ndarray*) – velocity array whose unit is m/s.
- **phi** (*1-d ndarray*) – porosity array
- **vsh** (*1-d ndarray*) – shale volume
- **a_0, a_1, a_2, a_3** (*scalar*) – coefficients

Returns **sigma**

Return type 1-d ndarray

`pygeopressure.pressure.multivariate.`**`multivariate_unloading`**`(`*sigma, phi, vsh, a_0, a_1, a_2, a_3, B, U, vmax*`)`

Calculate velocity using multivariate unloading curve

`pygeopressure.pressure.multivariate.`**`multivariate_virgin`**`(`*sigma, phi, vsh, a_0, a_1, a_2, a_3, B*`)`

Calculate velocity using multivariate virgin curve

Parameters

- **sigma** (*1-d ndarray*) – effective pressure
- **phi** (*1-d ndarray*) – effective porosity
- **vsh** (*1-d ndarray*) – shale volume
- **a_0, a_1, a_2, a_3** (*float*) – coefficients of equation
- **B** (*float*) – effective pressure exponential

Returns **out** – velocity array

Return type 1-d ndarray

Notes

$$V = a_0 + a_1\phi + a_2V_{sh} + a_3\sigma^B$$

[5]

pygeopressure.pressure.multivariate.**pressure_multivariate**(*obp*, *vel*, *phi*, *vsh*, *a_0*, *a_1*, *a_2*, *a_3*, *B*, *U*, *vmax*, *start_idx*, *end_idx=None*)

Pressure Prediction using multivariate model

pygeopressure.pressure.multivariate.**pressure_multivariate_varu**(*obp*, *vel*, *phi*, *vsh*, *a_0*, *a_1*, *a_2*, *a_3*, *B*, *U*, *vmax*, *start_idx*, *buf=20*, *end_idx=None*, *end_buffer=10*)

Pressure Prediction using multivariate model

pygeopressure.pressure.obp module

pygeopressure.pressure.utils module

Module contents

pygeopressure.velocity package

Submodules

pygeopressure.velocity.conversion module

Routines performing velocity type conversion

pygeopressure.velocity.conversion.**avg2int**(*twt*, *v_avg*)

Parameters

- **twt** (*1-d ndarray*)
- **v_avg** (*1-d ndarray*)

Returns **v_int**

Return type 1-d ndarray

pygeopressure.velocity.conversion.**int2avg**(*twt*, *v_int*)

Parameters

- **twt** (*1-d ndarray*)
- **v_int** (*1-d ndarray*)

Returns **v_avg**

[5] Sayers, C., Smit, T., van Eden, C., Wervelman, R., Bachmann, B., Fitts, T., et al. (2003). Use of reflection tomography to predict pore pressure in overpressured reservoir sands. In submitted for presentation at the SEG 2003 annual meeting.

Return type 1-d ndarray

Notes

$$V_{int}[i](t_i - t_{i-1}) = V_{avg}[i]t_i - \ V_{avg}[i-1]t_{i-1}$$

`pygeopressure.velocity.conversion.`**`int2rms`**(*twt*, *v_int*)

Parameters

- **twt** (*1-d ndarray*)
- **v_int** (*1-d ndarray*)

Returns **v_rms**

Return type 1-d ndarray

`pygeopressure.velocity.conversion.`**`rms2int`**(*twt*, *v_rms*)

Convert rms velocity to interval velocity

Parameters

- **twt** (*1-d ndarray*) – input two-way-time array, in ms
- **rms** (*1-d nadarray*) – rms velocity array, in m/s

Returns **v_int** – interval velocity array with the same length of twt and rms

Return type 1-d ndarray

Notes

This routine uses Dix equation to comput inverval velocity.

$$V_{int}[i]^2 = \frac{V_{rms}[i]^2 t_i - V_{rms}[i-1]^2 \ t_{i-1}}{t_i - t_{i-1}}$$

twt and rms should be of the same length of more than 2.

Examples

```
>>> a = np.arange(10)
>>> twt = np.arange(10)
>>> rms2int(twt, a)
array([  0.        ,   1.        ,   2.64575131,   4.35889894,
         6.08276253,   7.81024968,   9.53939201,  11.26942767,
        13.        ,  14.73091986])
```

`pygeopressure.velocity.conversion.`**`twt2depth`**(*twt*, *v_avg*, *prop_2_convert*, *stepDepth=4*, *startDepth=None*, *endDepth=None*)

Parameters

- **twt** (*1-d ndarray*)
- **v_avg** (*1-d ndarray*)
- **prop_2_convert** (*1-d ndarray*)
- **stepDepth** (*scalar*)

- **startDpeth (optional)** (*scalar*)
- **endDepth (optional)** (*scalar*)

Returns

- **newDepth** (*1-d ndarray*) – new depth array
- **new_prop_2_convert** (*1-d ndarray*) – average velocity in depth domain

pygeopressure.velocity.extrapolate module

Functions relating velocity trend extrapolation

pygeopressure.velocity.extrapolate.**normal**(*x*, *a*, *b*)

Extrapolate velocity using normal trend.

Parameters

- **x** (*1-d ndarray*) – depth to convert
- **a, b** (*scalar*) – coefficents

Returns **out** – esitmated velocity

Return type 1-d ndarray

Notes

$$\log dt_{Normal} = a - bz$$

is transformed to

$$v = e^{bz-a}$$

Note that the exponential relation is unphysical especially in depth bellow the interval within which the equation is calibrated.

References

pygeopressure.velocity.extrapolate.**normal_dt**(*x*, *a*, *b*)

normal trend of transit time

Parameters **x** (*1-d ndarray*) – depth to convert

pygeopressure.velocity.extrapolate.**normal_log**(*vel_log*, *a*, *b*)

Returns normal velocity log

Return type *Log*

pygeopressure.velocity.extrapolate.**set_v0**(*v*)

set global variable v0 for slotnick()

pygeopressure.velocity.extrapolate.**slotnick**(*x*, *k*)

Relation between velocity and depth

Parameters

- **x** (*1-d ndarray*) – Depth to convert
- **k** (*scalar*) – velocity gradient

Notes

typical values of velocity gradient k falls in the range 0.6-1.0s-1

References

pygeopressure.velocity.interpolation module

2-d interpolation routines

`pygeopressure.velocity.interpolation.`**`interp_DW`**(*array2d*)

2-D distance-weighted interpolation

Parameters **array2d** (*ndarray*) – 2-D ndarray void values being singaled by np.nan

Examples

```
>>> a = np.array([[2, 2, 2], [2, np.nan, 2], [2, 2, 2]])
>>> b = interp_DW(a)
```

`pygeopressure.velocity.interpolation.`**`spline_1d`**(*twt*, *vel*, *step*, *startTwt=None*, *endTwt=None*, *method='cubic'*)

pygeopressure.velocity.smoothing module

2-d smoothing

`pygeopressure.velocity.smoothing.`**`smooth`**(*x*, *window_len=11*, *window='hanning'*)

Smooth the data using a window with requested size.

This method is based on the convolution of a scaled window with the signal. The signal is prepared by introducing reflected copies of the signal (with the window size) in both ends so that transient parts are minimized in the begining and end part of the output signal.

Parameters

- **x** (*ndarray*) – the input signal
- **window_len** (*scalar*) – the dimension of the smoothing window; should be an odd integer.
- **window** (*scalar*) – the type of window from 'flat', 'hanning', 'hamming', 'bartlett', 'blackman' flat window will produce a moving average smoothing.

Returns **y** – the smoothed signal

Return type ndarray

Examples

```
>>> t=linspace(-2,2,0.1)
>>> x=sin(t)+randn(len(t))*0.1
>>> y=smooth(x)
```

See also:

`numpy.hanning()`, `numpy.hamming()`, `numpy.bartlett()`, `numpy.blackman()`, `numpy.convolve()`

`TODO()` the window parameter could be the window itself if an array instead of a string

Notes

length(output) != length(input), to correct this: return y[(window_len/2-1):-(window_len/2)] instead of just y.

`pygeopressure.velocity.smoothing.`**`smooth_2d`**(*m*)

`pygeopressure.velocity.smoothing.`**`smooth_trace`**(*trace_data*, *window=120*)

Module contents

Module contents

PYTHON MODULE INDEX

p

INDEX

T

U

V

W