Fatiando a Terra

Geophysical modeling and inversion

User and developer guide

Release 0.1

Leonardo Uieda

March 20, 2013

CONTENTS

An open source toolkit for geophysical modeling and inversion

Fatiando provides an easy and flexible way to perform common tasks like: generating synthetic data, forward modeling, inversion, 3D visualization, and more! All from inside the powerful Python (http://www.python.org) language.

For more information visit the official site (http://www.fatiando.org).

The **source code** of Fatiando is hosted on GitHub (https://github.com/leouieda/fatiando).

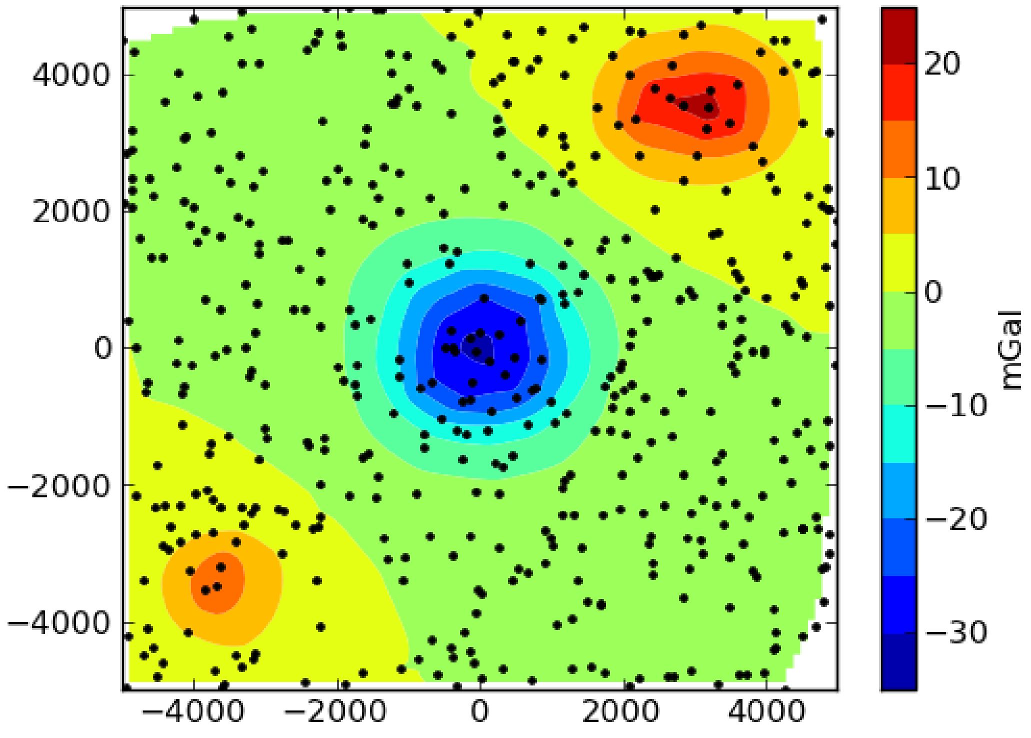
License: Fatiando is licensed under the **BSD license**. This means that it can be reused and remixed with few restrictions. See the *license text* for more information.

The best **place to start** learning about Fatiando is the *Cookbook*! There, you'll find many sample scripts with common tasks that can help you get started.

As an **example**, this is how easy it is to create synthetic noise-corrupted gravity data on random points from a 3D prism model:

```
>>> from fatiando.mesher import Prism
>>> from fatiando.vis import mpl
>>> from fatiando import gridder, utils, gravmag
>>> # Create the prism model
>>> prisms = [
...     Prism(-4000, -3000, -4000, -3000, 0, 2000, {'density':1000}),
...     Prism(-1000, 1000, -1000, 1000, 0, 2000, {'density':-1000}),
...     Prism(2000, 4000, 3000, 4000, 0, 2000, {'density':1000})]
>>> # Generate 500 random observation points at 100m height
>>> xp, yp, zp = gridder.scatter((-5000, 5000, -5000, 5000), 500, z=-100)
>>> # Calculate their gravitational effect and contaminate it with 0.1 mGal
>>> # gaussian noise
>>> gz = utils.contaminate(gravmag.prism.gz(xp, yp, zp, prisms), 0.1)
>>> # Plot the result
>>> mpl.contourf(xp, yp, gz, (100, 100), 12, interp=True)
>>> cb = mpl.colorbar()
>>> cb.set_label('mGal')
>>> mpl.plot(xp, yp, '.k')
>>> mpl.show()
```

which results in something like this:



For those of you not so interested in potential fields, there is a new module *fatiando.seismic.wavefd* for **2D finite difference simulations of seismic waves**!

CHAPTER

ONE

CONTRIBUTORS

A (hopefully) updated list of people who contributed to Fatiando a Terra:

- Leonardo Uieda (http://fatiando.org/people/uieda) - Observatório Nacional, Brazil
- Vanderlei Coelho de Oliveira Junior (http://fatiando.org/people/oliveira-jr) - Observatório Nacional, Brazil
- José Fernando Caparica Junior (http://fatiando.org/people/caparicajr) - LENEP, Brazil

CHAPTER TWO

LICENSE

CHAPTER

THREE

CHANGELOG

3.1 Version 0.1

Release date: NOT RELEASED

Changes:

- Change license to BSD (see the *license text*).
- The API is now fully accessible by only importing `fatiando`
- Added a *Cookbook* section to the documentation with all the sample scripts from the cookbook folder.
- Implemented "Robust 3D gravity gradient inversion by planting anomalous densities" by Uieda and Barbosa (2012) in *fatiando.gravmag.harvester*
- Added harvester command line program that runs this new inversion
- Added magnetic total field anomaly function to *fatiando.gravmag.prism*
- Added *fatiando.vis.myv.savefig3d* to save a Mayavi scene
- Added *fatiando.vis.myv.polyprisms* 3D plotter function for PolygonalPrism
- Added *fatiando.vis.myv.points3d* 3D plotter function for points
- Added gravity gradient tensor components and magnetic total field anomaly to *fatiando.gravmag.polyprism*
- Added option to control the line width to *prisms* and *polyprisms* in *fatiando.vis.myv*
- Added module *fatiando.gravmag.tensor* for processing gradient tensor data. Includes eigenvalues and eigenvectors, tensor invariants, center of mass estimation, etc.
- Added module *fatiando.gravmag.imaging* with imaging methods for potential fields
- Added module *fatiando.gravmag.euler* with Euler deconvolution methods for potential field data
- Added module *fatiando.seismic.wavefd* with 2D Finite Difference simulations of elastic seismic waves
- Added unit conversion functions to *fatiando.utils*
- Added tesseroids forward modeling *fatiando.gravmag.tesseroid*, meshing and plotting with Mayavi
- New *fatiando.io* module to fetch models and data from the web and convert them to useful formats (for now supports the CRUST2.0 global curstal model)
- If building inplace or packaging, the setup script puts the Mercurial changeset hash in a file. Then *fatiando.logger.header* loads the hash from file and put a "Unknown" if it can't read. This way importing fatiando won't fail if the there is no changeset information available.
- *fatiando.mesher.PrismMesh.dump*: takes a mesh file, a property file and a property name. Saves the output to these files.
- Transformed all geometric elements (like Prism, Polygon, etc) into classes
- Ported all C extensions to Python + Numpy. This way compiling is not a prerequisite to installing

- Using Cython (http://www.cython.org) for optional extension modules. If they exist, they are loaded to replace the Python + Numpy versions. This all happens at runtime.
- Move all physical constants used in `fatiando` to module *fatiando.constants*
- Data modules hidden inside functions in *fatiando.gravmag.basin2d*
- Functions in *fatiando.gravmag.basin2d* spit out Polygons instead of the vertices estimated. Now you don't have to build the polygons by hand.

Bug fixes:

CHAPTER

FOUR

INSTALLING FATIANDO

Note: If you have any trouble installing please write to the mailing list (https://groups.google.com/forum/#!forum/fatiando) or to Leonardo Uieda (http://fatiando.org/people/uieda/). This will help us make Fatiando better!

4.1 Install the dependencies

Fatiando requires the following packages:

- numpy (http://numpy.scipy.org/)
- scipy (http://scipy.org/)
- matplotlib (http://matplotlib.sourceforge.net/)
- PIL (http://www.pythonware.com/products/pil/)
- mayavi (http://code.enthought.com/projects/mayavi/)
- Cython (http://cython.org/): to compile faster modules in C. Needed only when installing from source (or using `pip`).

All of these can be found on most **GNU/Linux** distros. If you're on the latest (or close to) **Ubuntu**, you can run:

```
sudo apt-get install python-dev python-numpy python-matplotlib python-scipy\
python-mpltoolkits.basemap python-imaging mayavi2 cython python-pip
```

You'll need `python-dev` to build the optimized Cython modules and `python-pip` to install fatiando. All of these can also be found in the Software Center, Synaptic, etc.

Note: The '2' in `mayavi2` is not a typo. It really is called that.

On **Windows**, I recommend downloading PythonXY (http://code.google.com/p/pythonxy/). It comes with Python, all of our dependencies, plus a whole bunch of useful stuff! Trust me, it's better than installing things separately. **Warning**: If you already have Python installed, you should uninstall it before installing PythonXY. When installing PythonXY, make sure the following are selected (or go with a full install to be sure):

- numpy
- scipy
- matplotlib
- PIL
- ETS (for mayavi)
- VTK (for mayavi)

4.2 Installing on Linux

After you've installed the dependencies you can proceed to install Fatiando using pip (http://www.pip-installer.org):

```
sudo pip install fatiando
```

That's it! If you already have Fatiando installed and want to **upgrade** to a newer version, use:

```
sudo pip install fatiando --upgrade
```

To uninstall simply run:

```
sudo pip uninstall fatiando
```

If you don't have root access (no `sudo` for you), you can install Fatiando on a virtualenv (http://pypi.python.org/pypi/virtualenv). If you don't know what that means, read this (http://jontourage.com/2011/02/09/virtualenv-pip-basics/).

4.3 Installing on Windows

After you've installed PythonXY (or similar) with all the dependencies, download the latest Windows installer from PyPI (http://pypi.python.org/pypi/fatiando). Just click through the installer and you should be done!

4.4 Testing the install

Try running one of the recipes from the *Cookbook*. If you get an error message, please post to the mailing list (https://groups.google.com/forum/#!forum/fatiando).

CHAPTER

FIVE

API REFERENCE: THE FATIANDO PACKAGE

The fatiando package contains all the subpackages and modules required for most tasks.

Modules for each geophysical method are group in subpackages:

- gravmag: Gravity and magnetics (i.e., potential fields)
- seismic: Seismics and seismology
- geothermal: Geothermal heat transfer modeling

Modules for gridding, meshing, visualization, user interface, input/output etc:

- mesher: Mesh generation and definition of geometric elements
- gridder: Grid generation and operations (e.g., interpolation)
- vis: Plotting utilities for 2D (using matplotlib) and 3D (using mayavi)
- io: Input/Output of models, data sets, etc (fetch from web repositories)
- gui: Graphical user interfaces (still very primitive)
- utils: Miscelaneous utilities, like mathematical functions, unit conversion, etc
- logger: A simpler interface to the Python logging module for log files
- constants: Physical constants and unit conversions

Also included is the fatiando.inversion package with utilities for implementing inverse problems. There you'll find common regularizing functions, linear inverse problem solvers, and non-linear gradient solvers. This package is generaly only used from inside Fatiando itself, not when using Fatiando in scripts. For usage examples, see the source of modules fatiando.seismic.epic2d and fatiando.gravmag.basin2d.

See the documentation for each module to find out more about what they do and how to use them.

5.1 Gravity and magnetics (fatiando.gravmag)

Gravity and magnetics forward modeling, inversion, transformations and utilities.

Forward modeling

The forward modeling modules provide ways to calculate the gravitational and magnetic field of various types of geometric objects:

- prism: 3D right rectangular prisms
- polyprism: 3D prisms with polygonal horizontal cross-sections

- sphere: Spheres in Cartesian coordinates
- tesseroid: Tesseroids (spherical prisms) for modeling in spherical coordinates
- talwani: 2D bodies with polygonal vertical cross-sections
- half_sph_shell: Gravity fields of half a spherical shell. Useful for benchmarking and testing.

Inversion

The inversion modules use the forward modeling models and the fatiando.inversion package to solve potential field inverse problems:

- basin2d: 2D inversion of the shape of sedimentary basins and other outcropping bodies
- harvester: 3D inversion of compact bodies by planting anomalous densities
- euler: 3D Euler deconvolution methods to estimate source location

Processing

The processing modules offer tools to prepare potential field data before or after modeling.

- transform: Space domain potential field transformations, like upward continuation
- fourier: Potential field transformations using the FFT
- imaging: Imaging methods for potential fields for estimating physical property distributions
- tensor: Utilities for operating on the gradient tensor

5.1.1 3D inversion by planting anomalous densities (`fatiando.gravmag.harvester`)

3D potential field inversion by planting anomalous densities.

Implements the method of Uieda and Barbosa (2012a) with improvements by Uieda and Barbosa (2012b).

A "heuristic" inversion for compact 3D geologic bodies. Performs the inversion by iteratively growing the estimate around user-specified "seeds". Supports various kinds of data (gravity, gravity tensor).

The inversion is performed by function harvest. The required information, such as observed data, seeds, and regularization, are passed to the function through classes Seed and Potential, Gz, Gxx, etc.

See the *Cookbook* for some example applications to synthetic data.

Functions

- harvest: Performs the inversion
- sow: Creates the seeds from a set of (x, y, z) points and physical properties
- loadseeds: Loads from a JSON file a set of (x, y, z) points and physical properties that specify the seeds. Pass output to sow

Data types

- Potential: gravitational potential
- Gz: vertical component of gravitational acceleration (i.e., gravity anomaly)
- Gxx: North-North component of the gravity gradient tensor
- Gxy: North-East component of the gravity gradient tensor
- Gxz: North-vertical component of the gravity gradient tensor
- Gyy: East-East component of the gravity gradient tensor
- Gyz: East-vertical component of the gravity gradient tensor

- `Gzz`: vertical-vertical component of the gravity gradient tensor

References

Uieda, L., and V. C. F. Barbosa (2012a), Robust 3D gravity gradient inversion by planting anomalous densities, Geophysics, 77(4), G55-G66, doi:10.1190/geo2011-0388.1 [pdf (http://www.mendeley.com/download/public/1406731/4823610241/45dec08fa03c4d5950ecdaef8d7532767a57a1a8/dl.pdf)]

Uieda, L., and V. C. F. Barbosa (2012b), Use of the "shape-of-anomaly" data misfit in 3D inversion by planting anomalous densities, SEG Technical Program Expanded Abstracts, 1-6, doi:10.1190/segam2012-0383.1 [pdf (http://www.mendeley.com/download/public/1406731/4932659461/67606df295d428a7f729a74cf80b7ed4aa37553b/dl.pdf)]

---

class `fatiando.gravmag.harvester.`**`Data`** (*x*, *y*, *z*, *data*, *meshtype*)

Bases: `object`

A container for some potential field data.

Know about its data, observation positions, nature of the mesh, and how to calculate the effect of a single cell.

class `fatiando.gravmag.harvester.`**`Gxx`** (*x*, *y*, *z*, *data*, *meshtype='prism'*)

Bases: `fatiando.gravmag.harvester.Potential`

A container for data of the xx (north-north) component of the gravity gradient tensor.

Coordinate system used: x->North y->East z->Down

Parameters:

- **x, y, z** [1D arrays] Arrays with the x, y, z coordinates of the data points
- **data** [1D array] The values of the data at the observation points

class `fatiando.gravmag.harvester.`**`Gxy`** (*x*, *y*, *z*, *data*, *meshtype='prism'*)

Bases: `fatiando.gravmag.harvester.Potential`

A container for data of the xy (north-east) component of the gravity gradient tensor.

Coordinate system used: x->North y->East z->Down

Parameters:

- **x, y, z** [1D arrays] Arrays with the x, y, z coordinates of the data points
- **data** [1D array] The values of the data at the observation points

class `fatiando.gravmag.harvester.`**`Gxz`** (*x*, *y*, *z*, *data*, *meshtype='prism'*)

Bases: `fatiando.gravmag.harvester.Potential`

A container for data of the xz (north-vertical) component of the gravity gradient tensor.

Coordinate system used: x->North y->East z->Down

Parameters:

- **x, y, z** [1D arrays] Arrays with the x, y, z coordinates of the data points
- **data** [1D array] The values of the data at the observation points

class `fatiando.gravmag.harvester.`**`Gyy`** (*x*, *y*, *z*, *data*, *meshtype='prism'*)

Bases: `fatiando.gravmag.harvester.Potential`

A container for data of the yy (east-east) component of the gravity gradient tensor.

Coordinate system used: x->North y->East z->Down

Parameters:

- **x, y, z** [1D arrays] Arrays with the x, y, z coordinates of the data points
- **data** [1D array] The values of the data at the observation points

class `fatiando.gravmag.harvester.`**`Gyz`** (*x*, *y*, *z*, *data*, *meshtype='prism'*)
Bases: `fatiando.gravmag.harvester.Potential`

A container for data of the yz (east-vertical) component of the gravity gradient tensor.

Coordinate system used: x->North y->East z->Down

Parameters:

- **x, y, z** [1D arrays] Arrays with the x, y, z coordinates of the data points
- **data** [1D array] The values of the data at the observation points

class `fatiando.gravmag.harvester.`**`Gz`** (*x*, *y*, *z*, *data*, *meshtype='prism'*)
Bases: `fatiando.gravmag.harvester.Potential`

A container for data of the gravity anomaly.

Coordinate system used: x->North y->East z->Down

Parameters:

- **x, y, z** [1D arrays] Arrays with the x, y, z coordinates of the data points
- **data** [1D array] The values of the data at the observation points

class `fatiando.gravmag.harvester.`**`Gzz`** (*x*, *y*, *z*, *data*, *meshtype='prism'*)
Bases: `fatiando.gravmag.harvester.Potential`

A container for data of the zz (vertical-vertical) component of the gravity gradient tensor.

Coordinate system used: x->North y->East z->Down

Parameters:

- **x, y, z** [1D arrays] Arrays with the x, y, z coordinates of the data points
- **data** [1D array] The values of the data at the observation points

class `fatiando.gravmag.harvester.`**`Neighbor`** (*i*, *props*, *seed*, *distance*, *effect*)
Bases: `object`

A neighbor.

class `fatiando.gravmag.harvester.`**`Potential`** (*x*, *y*, *z*, *data*, *meshtype='prism'*)
Bases: `fatiando.gravmag.harvester.Data`

A container for data of the gravitational potential.

Coordinate system used: x->North y->East z->Down

Parameters:

- **x, y, z** [1D arrays] Arrays with the x, y, z coordinates of the data points
- **data** [1D array] The values of the data at the observation points

class `fatiando.gravmag.harvester.`**`Seed`** (*i*, *props*)
Bases: `object`

A seed.

class `fatiando.gravmag.harvester.`**`TotalField`** (*x*, *y*, *z*, *data*, *inc*, *dec*, *meshtype='prism'*)
Bases: `fatiando.gravmag.harvester.Potential`

A container for data of the total field magnetic anomaly.

Coordinate system used: x->North y->East z->Down

Parameters:

- **x, y, z** [1D arrays] Arrays with the x, y, z coordinates of the data points
- **data** [1D array] The values of the data at the observation points

- **inc, dec** [floats] The inclination and declination of the inducing field

`fatiando.gravmag.harvester.`**`harvest`**(*data*, *seeds*, *mesh*, *compactness*, *threshold*)

Run the inversion algorithm and produce an estimate physical property distribution (density and/or magnetization).

Paramters:

- **data** [list of data (e.g., `Gz`)] The data that will be inverted. Data used must match the physical properties given to the seeds (e.g., gravity data requires seeds to have `'density'` prop)
- **seeds** [list of `Seed`] Lits of seeds used to start the growth process of the inversion. Use `sow` to generate seeds.
- **mesh** [`fatiando.mesher.PrismMesh`] The mesh used in the inversion. Will estimate the physical property distribution on this mesh
- **compactness** [float] The compactness regularing parameter (i.e., how much should the estimate be consentrated around the seeds). Must be positive. To find a good value for this, start with a small value (like 0.001), run the inversion and increase the value until desired compactness is achieved.
- **threshold** [float] Control how much the solution can grow (usually downward). In order for estimate to grow by the accretion of 1 prism, this prism must decrease the data misfit measure by *threshold* decimal percent. Depends on the size of the cells in the *mesh* and the distance from a cell to the observations. Use values between 0.001 and 0.000001. If cells are small and *threshold* is large (0.001), the seeds won't grow. If cells are large and *threshold* is small (0.000001), the seeds will grow too much.

Returns:

- **estimate, predicted_data** [a dict and a list] *estimate* is a dict like:

```
{'physical_property':array, ...}
```

estimate contains the estimates physical properties. The properties present in *estimate* are the ones given to the seeds. Include the properties in the *mesh* using:

```
mesh.addprop('density', estimate['density'])
```

This way you can plot the estimate using `fatiando.vis.myv`.

predicted_data is a list of numpy arrays with the predicted (model) data. The list is in the same order as *data*. To plot a map of the fit for visual inspection and a histogram of the residuals:

```python
from fatiando.vis import mpl
mpl.figure()
# Plot the observed and predicted data as contours for visual
# inspection
mpl.subplot(1, 2, 1)
mpl.axis('scaled')
mpl.title('Observed and predicted data')
levels = mpl.contourf(x, y, gz, (ny, nx), 10)
mpl.colorbar()
# Assuming gz is the only data used
mpl.contour(x, y, predicted[0], (ny, nx), levels)
# Plot a histogram of the residuals
residuals = gz - predicted[0]
mpl.subplot(1, 2, 2)
mpl.title('Residuals')
mpl.hist(residuals, bins=10)
mpl.show()
# It's also good to see the mean and standard deviation of the
# residuals
print "Residuals mean:", residuals.mean()
print "Residuals stddev:", residuals.std()
```

`fatiando.gravmag.harvester.`**`loadseeds`**(*fname*)

Load a set of seed locations and physical properties from a file.

The output can then be used with the `sow` function.

The seed file should be formatted as:

```
[
    [x1, y1, z1, {"density":dens1}],
    [x2, y2, z2, {"density":dens2, "magnetization":mag2}],
    [x3, y3, z3, {"magnetization":mag3, "inclination":inc3,
                  "declination":dec3}],
    ...
]
```

x, y, z are the coordinates of the seed and the dict (`{'density':2670}`) are its physical properties.

Warning: Must use `"`, not `'`, in the physical property names!

Each seed can have different kinds of physical properties. If inclination and declination are not given, will use the inc and dec of the inducing field (i.e., no remanent magnetization).

The techie among you will recognize that the seed file is in JSON format.

Remember: the coordinate system is x->North, y->East, and z->Down

Parameters:

- **fname** [str or file] Open file object or filename string

Returns:

- **[[x1, y1, z1, props1], [x2, y2, z2, props2], ...]** (x, y, z) are the points where the seeds will be placed and *props* is dict with the values of the physical properties of each, seed.

Example:

```
>>> from StringIO import StringIO
>>> file = StringIO(
...     '[[1, 2, 3, {"density":4, "magnetization":5}],' +
...     ' [6, 7, 8, {"magnetization":-1}]]')
>>> seeds = loadseeds(file)
>>> for s in seeds:
...     print s
[1, 2, 3, {u'magnetization': 5, u'density': 4}]
[6, 7, 8, {u'magnetization': -1}]
```

`fatiando.gravmag.harvester.`**`sow`**(*locations*, *mesh*)

Create the seeds given a list of (x,y,z) coordinates and physical properties.

Removes seeds that would fall on the same location with overlapping physical properties.

Parameters:

- **locations** [list] The locations and physical properties of the seeds. Should be a list like:

```
[
    [x1, y1, z1, {"density":dens1}],
    [x2, y2, z2, {"density":dens2, "magnetization":mag2}],
    [x3, y3, z3, {"magnetization":mag3, "inclination":inc3,
                  "declination":dec3}],
    ...
]
```

- **mesh** [`fatiando.mesher.PrismMesh`] The mesh that will be used in the inversion.

Returns:

- **seeds** [list of `Seed`] The seeds that can be passed to `harvest`

5.1.2 Forward modeling with 3D right rectangular prisms (`fatiando.gravmag.prism`)

Calculate the potential fields of the 3D right rectangular prism.

Gravity

The gravitational fields are calculated using the forumla of Nagy et al. (2000)

- `potential`
- `gx`
- `gy`
- `gz`
- `gxx`
- `gxy`
- `gxz`
- `gyy`
- `gyz`
- `gzz`

Magnetic

The Total Field anomaly is calculated using the formula of Bhattacharyya (1964).

- `tf`

References

Bhattacharyya, B. K. (1964), Magnetic anomalies due to prism-shaped bodies with arbitrary polarization, Geophysics, 29(4), 517, doi: 10.1190/1.1439386.

Nagy, D., G. Papp, and J. Benedek (2000), The gravitational potential and its derivatives for the prism: Journal of Geodesy, 74, 552–560, doi: 10.1007/s001900000116.

Note: This is a Python + Numpy implementation of the potential field effects of right rectangular prisms. There is a Cython implementation in _cprism.pyx It will be loaded automatically if it is compiled.

`fatiando.gravmag._prism.`**`potential`**(*xp*, *yp*, *zp*, *prisms*, *dens=None*)

Calculates the gravitational potential.

Note: The coordinate system of the input parameters is to be x -> North, y -> East and z -> Down.

Note: All input values in **SI** units(!) and output in **mGal**!

Parameters:

- **xp, yp, zp** [arrays] Arrays with the x, y, and z coordinates of the computation points.
- **prisms** [list of `Prism`] The density model used to calculate the gravitational effect. Prisms must have the property `'density'`. Prisms that don't have this property will be ignored in the computations. Elements of *prisms* that are None will also be ignored. *prisms* can also be a `PrismMesh`.

- **dens** [float or None] If not None, will use this value instead of the `'density'` property of the prisms. Use this, e.g., for sensitivity matrix building.

  > **Warning:** Uses this value for **all** prisms! Not only the ones that have `'density'` as a property.

Returns:

- **res** [array] The field calculated on xp, yp, zp

`fatiando.gravmag._prism.`**`gx`**(*xp*, *yp*, *zp*, *prisms*, *dens=None*)

Calculates the g_x gravity acceleration component.

Note: The coordinate system of the input parameters is to be x -> North, y -> East and z -> Down.

Note: All input values in **SI** units(!) and output in **mGal**!

Parameters:

- **xp, yp, zp** [arrays] Arrays with the x, y, and z coordinates of the computation points.
- **prisms** [list of `Prism`] The density model used to calculate the gravitational effect. Prisms must have the property `'density'`. Prisms that don't have this property will be ignored in the computations. Elements of *prisms* that are None will also be ignored. *prisms* can also be a `PrismMesh`.
- **dens** [float or None] If not None, will use this value instead of the `'density'` property of the prisms. Use this, e.g., for sensitivity matrix building.

  > **Warning:** Uses this value for **all** prisms! Not only the ones that have `'density'` as a property.

Returns:

- **res** [array] The field calculated on xp, yp, zp

`fatiando.gravmag._prism.`**`gy`**(*xp*, *yp*, *zp*, *prisms*, *dens=None*)

Calculates the g_y gravity acceleration component.

Note: The coordinate system of the input parameters is to be x -> North, y -> East and z -> Down.

Note: All input values in **SI** units(!) and output in **mGal**!

Parameters:

- **xp, yp, zp** [arrays] Arrays with the x, y, and z coordinates of the computation points.
- **prisms** [list of `Prism`] The density model used to calculate the gravitational effect. Prisms must have the property `'density'`. Prisms that don't have this property will be ignored in the computations. Elements of *prisms* that are None will also be ignored. *prisms* can also be a `PrismMesh`.
- **dens** [float or None] If not None, will use this value instead of the `'density'` property of the prisms. Use this, e.g., for sensitivity matrix building.

  > **Warning:** Uses this value for **all** prisms! Not only the ones that have `'density'` as a property.

Returns:

- **res** [array] The field calculated on xp, yp, zp

`fatiando.gravmag._prism.`**`gz`** (*xp*, *yp*, *zp*, *prisms*, *dens=None*)
Calculates the g_z gravity acceleration component.

Note: The coordinate system of the input parameters is to be x -> North, y -> East and z -> Down.

Note: All input values in **SI** units(!) and output in **mGal**!

Parameters:

- **xp, yp, zp** [arrays] Arrays with the x, y, and z coordinates of the computation points.
- **prisms** [list of `Prism`] The density model used to calculate the gravitational effect. Prisms must have the property `'density'`. Prisms that don't have this property will be ignored in the computations. Elements of *prisms* that are None will also be ignored. *prisms* can also be a `PrismMesh`.
- **dens** [float or None] If not None, will use this value instead of the `'density'` property of the prisms. Use this, e.g., for sensitivity matrix building.

  Warning: Uses this value for **all** prisms! Not only the ones that have `'density'` as a property.

Returns:

- **res** [array] The field calculated on xp, yp, zp

`fatiando.gravmag._prism.`**`gxx`** (*xp*, *yp*, *zp*, *prisms*, *dens=None*)
Calculates the g_{xx} gravity gradient tensor component.

Note: The coordinate system of the input parameters is to be x -> North, y -> East and z -> Down.

Note: All input values in **SI** units(!) and output in **mGal**!

Parameters:

- **xp, yp, zp** [arrays] Arrays with the x, y, and z coordinates of the computation points.
- **prisms** [list of `Prism`] The density model used to calculate the gravitational effect. Prisms must have the property `'density'`. Prisms that don't have this property will be ignored in the computations. Elements of *prisms* that are None will also be ignored. *prisms* can also be a `PrismMesh`.
- **dens** [float or None] If not None, will use this value instead of the `'density'` property of the prisms. Use this, e.g., for sensitivity matrix building.

  Warning: Uses this value for **all** prisms! Not only the ones that have `'density'` as a property.

Returns:

- **res** [array] The field calculated on xp, yp, zp

`fatiando.gravmag._prism.`**`gxy`** (*xp*, *yp*, *zp*, *prisms*, *dens=None*)
Calculates the g_{xy} gravity gradient tensor component.

Note: The coordinate system of the input parameters is to be x -> North, y -> East and z -> Down.

Note: All input values in **SI** units(!) and output in **mGal**!

Parameters:

- **xp, yp, zp** [arrays] Arrays with the x, y, and z coordinates of the computation points.
- **prisms** [list of `Prism`] The density model used to calculate the gravitational effect. Prisms must have the property `'density'`. Prisms that don't have this property will be ignored in the computations. Elements of *prisms* that are None will also be ignored. *prisms* can also be a `PrismMesh`.
- **dens** [float or None] If not None, will use this value instead of the `'density'` property of the prisms. Use this, e.g., for sensitivity matrix building.

  Warning: Uses this value for **all** prisms! Not only the ones that have `'density'` as a property.

Returns:

- **res** [array] The field calculated on xp, yp, zp

`fatiando.gravmag._prism.`**`gxz`**(*xp*, *yp*, *zp*, *prisms*, *dens=None*)

Calculates the g_{xz} gravity gradient tensor component.

Note: The coordinate system of the input parameters is to be x -> North, y -> East and z -> Down.

Note: All input values in **SI** units(!) and output in **mGal**!

Parameters:

- **xp, yp, zp** [arrays] Arrays with the x, y, and z coordinates of the computation points.
- **prisms** [list of `Prism`] The density model used to calculate the gravitational effect. Prisms must have the property `'density'`. Prisms that don't have this property will be ignored in the computations. Elements of *prisms* that are None will also be ignored. *prisms* can also be a `PrismMesh`.
- **dens** [float or None] If not None, will use this value instead of the `'density'` property of the prisms. Use this, e.g., for sensitivity matrix building.

  Warning: Uses this value for **all** prisms! Not only the ones that have `'density'` as a property.

Returns:

- **res** [array] The field calculated on xp, yp, zp

`fatiando.gravmag._prism.`**`gyy`**(*xp*, *yp*, *zp*, *prisms*, *dens=None*)

Calculates the g_{yy} gravity gradient tensor component.

Note: The coordinate system of the input parameters is to be x -> North, y -> East and z -> Down.

Note: All input values in **SI** units(!) and output in **mGal**!

Parameters:

- **xp, yp, zp** [arrays] Arrays with the x, y, and z coordinates of the computation points.
- **prisms** [list of `Prism`] The density model used to calculate the gravitational effect. Prisms must have the property `'density'`. Prisms that don't have this property will be ignored in the computations. Elements of *prisms* that are None will also be ignored. *prisms* can also be a `PrismMesh`.
- **dens** [float or None] If not None, will use this value instead of the `'density'` property of the prisms. Use this, e.g., for sensitivity matrix building.

> **Warning:** Uses this value for **all** prisms! Not only the ones that have `'density'` as a property.

Returns:

- **res** [array] The field calculated on xp, yp, zp

`fatiando.gravmag._prism.`**`gyz`**(*xp*, *yp*, *zp*, *prisms*, *dens=None*)
Calculates the g_{yz} gravity gradient tensor component.

> **Note:** The coordinate system of the input parameters is to be x -> North, y -> East and z -> Down.

> **Note:** All input values in **SI** units(!) and output in **mGal**!

Parameters:

- **xp, yp, zp** [arrays] Arrays with the x, y, and z coordinates of the computation points.
- **prisms** [list of `Prism`] The density model used to calculate the gravitational effect. Prisms must have the property `'density'`. Prisms that don't have this property will be ignored in the computations. Elements of *prisms* that are None will also be ignored. *prisms* can also be a `PrismMesh`.
- **dens** [float or None] If not None, will use this value instead of the `'density'` property of the prisms. Use this, e.g., for sensitivity matrix building.

> **Warning:** Uses this value for **all** prisms! Not only the ones that have `'density'` as a property.

Returns:

- **res** [array] The field calculated on xp, yp, zp

`fatiando.gravmag._prism.`**`gzz`**(*xp*, *yp*, *zp*, *prisms*, *dens=None*)
Calculates the g_{zz} gravity gradient tensor component.

> **Note:** The coordinate system of the input parameters is to be x -> North, y -> East and z -> Down.

> **Note:** All input values in **SI** units(!) and output in **mGal**!

Parameters:

- **xp, yp, zp** [arrays] Arrays with the x, y, and z coordinates of the computation points.
- **prisms** [list of `Prism`] The density model used to calculate the gravitational effect. Prisms must have the property `'density'`. Prisms that don't have this property will be ignored in the computations. Elements of *prisms* that are None will also be ignored. *prisms* can also be a `PrismMesh`.
- **dens** [float or None] If not None, will use this value instead of the `'density'` property of the prisms. Use this, e.g., for sensitivity matrix building.

> **Warning:** Uses this value for **all** prisms! Not only the ones that have `'density'` as a property.

Returns:

- **res** [array] The field calculated on xp, yp, zp

`fatiando.gravmag._prism.`**`tf`**(*xp*, *yp*, *zp*, *prisms*, *inc*, *dec*, *pmag=None*, *pinc=None*, *pdec=None*)
Calculate the total-field anomaly of prisms.

Note: Input units are SI. Output is in nT

Note: The coordinate system of the input parameters is to be x -> North, y -> East and z -> Down.

Parameters:

- **xp, yp, zp** [arrays] Arrays with the x, y, and z coordinates of the computation points.
- **prisms** [list of `Prism`] The model used to calculate the total field anomaly. Prisms must have the physical property `'magnetization'` will be ignored. If the physical properties `'inclination'` and `'declination'` are not present, will use the values of *inc* and *dec* instead (regional field). *prisms* can also be a `PrismMesh`.
- **inc** [float] The inclination of the regional field (in degrees)
- **dec** [float] The declination of the regional field (in degrees)
- **pmag** [float or None] If not None, will use this value instead of the `'magnetization'` property of the prisms. Use this, e.g., for sensitivity matrix building.
- **pinc** [float or None] If not None, will use this value instead of the `'inclination'` property of the prisms. Use this, e.g., for sensitivity matrix building.
- **pdec** [float or None] If not None, will use this value instead of the `'declination'` property of the prisms. Use this, e.g., for sensitivity matrix building.

Returns:

- **res** [array] The field calculated on xp, yp, zp

5.1.3 Forward modeling with 3D polygonal prisms (`fatiando.gravmag.polyprism`)

Calculate the potential fields of the 3D prism with polygonal crossection using the formula of Plouff (1976).

Gravity

First and second derivatives of the gravitational potential:

- `gz`
- `gxx`
- `gxy`
- `gxz`
- `gyy`
- `gyz`
- `gzz`

Magnetic

The Total Field magnetic anomaly:

- `tf`

References

Plouff, D. , 1976, Gravity and magnetic fields of polygonal prisms and applications to magnetic terrain corrections, Geophysics, 41(4), 727-741.

`fatiando.gravmag.polyprism.`**`gxx`**(*xp*, *yp*, *zp*, *prisms*)

Calculates the g_{xx} gravity gradient tensor component.

Note: The coordinate system of the input parameters is to be x -> North, y -> East and z -> Down.

Note: All input values in SI units and output in Eotvos!

Parameters:

- **xp, yp, zp** [arrays] The x, y, and z coordinates of the computation points.
- **prisms** [list of `fatiando.mesher.PolygonalPrism`] The model used to calculate the field. Prisms must have the physical property `'density'` will be ignored.

Returns:

- **res** [array] The effect calculated on the computation points.

`fatiando.gravmag.polyprism.`**`gxy`**(*xp*, *yp*, *zp*, *prisms*)

Calculates the g_{xy} gravity gradient tensor component.

Note: The coordinate system of the input parameters is to be x -> North, y -> East and z -> Down.

Note: All input values in SI units and output in Eotvos!

Parameters:

- **xp, yp, zp** [arrays] The x, y, and z coordinates of the computation points.
- **prisms** [list of `fatiando.mesher.PolygonalPrism`] The model used to calculate the field. Prisms must have the physical property `'density'` will be ignored.

Returns:

- **res** [array] The effect calculated on the computation points.

`fatiando.gravmag.polyprism.`**`gxz`**(*xp*, *yp*, *zp*, *prisms*)

Calculates the g_{xz} gravity gradient tensor component.

Note: The coordinate system of the input parameters is to be x -> North, y -> East and z -> Down.

Note: All input values in SI units and output in Eotvos!

Parameters:

- **xp, yp, zp** [arrays] The x, y, and z coordinates of the computation points.
- **prisms** [list of `fatiando.mesher.PolygonalPrism`] The model used to calculate the field. Prisms must have the physical property `'density'` will be ignored.

Returns:

- **res** [array] The effect calculated on the computation points.

`fatiando.gravmag.polyprism.`**`gyy`**(*xp*, *yp*, *zp*, *prisms*)

Calculates the g_{yy} gravity gradient tensor component.

Note: The coordinate system of the input parameters is to be x -> North, y -> East and z -> Down.

Note: All input values in SI units and output in Eotvos!

Parameters:

- **xp, yp, zp** [arrays] The x, y, and z coordinates of the computation points.
- **prisms** [list of `fatiando.mesher.PolygonalPrism`] The model used to calculate the field. Prisms must have the physical property `'density'` will be ignored.

Returns:

- **res** [array] The effect calculated on the computation points.

`fatiando.gravmag.polyprism.`**`gyz`**(*xp*, *yp*, *zp*, *prisms*)

Calculates the g_{yz} gravity gradient tensor component.

Note: The coordinate system of the input parameters is to be x -> North, y -> East and z -> Down.

Note: All input values in SI units and output in Eotvos!

Parameters:

- **xp, yp, zp** [arrays] The x, y, and z coordinates of the computation points.
- **prisms** [list of `fatiando.mesher.PolygonalPrism`] The model used to calculate the field. Prisms must have the physical property `'density'` will be ignored.

Returns:

- **res** [array] The effect calculated on the computation points.

`fatiando.gravmag.polyprism.`**`gz`**(*xp*, *yp*, *zp*, *prisms*)

Calculates the g_z gravity acceleration component.

Note: The coordinate system of the input parameters is to be x -> North, y -> East and z -> Down.

Note: All input values in SI units and output in mGal!

Parameters:

- **xp, yp, zp** [arrays] The x, y, and z coordinates of the computation points.
- **prisms** [list of `fatiando.mesher.PolygonalPrism`] The model used to calculate the field. Prisms must have the physical property `'density'` will be ignored.

Returns:

- **res** [array] The effect calculated on the computation points.

`fatiando.gravmag.polyprism.`**`gzz`**(*xp*, *yp*, *zp*, *prisms*)

Calculates the g_{zz} gravity gradient tensor component.

Note: The coordinate system of the input parameters is to be x -> North, y -> East and z -> Down.

Note: All input values in SI units and output in Eotvos!

Parameters:

- **xp, yp, zp** [arrays] The x, y, and z coordinates of the computation points.

- **prisms** [list of `fatiando.mesher.PolygonalPrism`] The model used to calculate the field. Prisms must have the physical property `'density'` will be ignored.

Returns:

- **res** [array] The effect calculated on the computation points.

`fatiando.gravmag.polyprism.`**`tf`**(*xp*, *yp*, *zp*, *prisms*, *inc*, *dec*)
Calculate the total-field anomaly of polygonal prisms.

Note: The coordinate system of the input parameters is to be x -> North, y -> East and z -> Down.

Note: Input units are SI. Output is in nT

Parameters:

- **xp, yp, zp** [arrays] Arrays with the x, y, and z coordinates of the computation points.
- **prisms** [list of `fatiando.mesher.PolygonalPrism`] The model used to calculate the total field anomaly. Prisms must have the physical property `'magnetization'` will be ignored. If the physical properties `'inclination'` and `'declination'` are not present, will use the values of *inc* and *dec* instead (regional field).
- **inc** [float] The inclination of the regional field (in degrees)
- **dec** [float] The declination of the regional field (in degrees)

Returns:

- **res** [array] The field calculated on xp, yp, zp

5.1.4 Forward modeling with spheres (`fatiando.gravmag.sphere`)

Calculate the potential fields of a homogeneous sphere.

Magnetic

Calculates the total field anomaly. Uses the formula in Blakely (1995).

- `tf`: calculates the total-field anomaly

Remember that:

The magnetization $\mathbf{M}$ and the dipole moment $\mathbf{m}$ are related with the volume V:

$$\mathbf{M} = \frac{\mathbf{m}}{V}.$$

The total-field anomaly is:

$$\Delta T = |\mathbf{T}| - |\mathbf{F}|,$$

where $\mathbf{T}$ is the measured field and $\mathbf{F}$ is a reference (regional) field. The forward modeling functions `bx`, `by`, and `bz` calculate the 3 components of the field perturbation $\Delta\mathbf{F}$

$$\Delta\mathbf{F} = \mathbf{T} - \mathbf{F}.$$

Then the total-field anomaly caused by the sphere is

$$\Delta T \approx \hat{\mathbf{F}} \cdot \Delta\mathbf{F}.$$

Gravity

References

`fatiando.gravmag.sphere.`**`gz`**(*xp*, *yp*, *zp*, *spheres*)

Calculates the g_z gravity acceleration component.

Note: The coordinate system of the input parameters is to be x -> North, y -> East and z -> Down.

Note: All input values in SI and output in mGal!

Parameters:

- **xp, yp, zp** [arrays] The x, y, and z coordinates where the field will be calculated
- **spheres** [list of `fatiando.mesher.Sphere`] The spheres. Spheres must have the property `'density'`. Those without will be ignored.

Returns:

- **res** [array] The field calculated on xp, yp, zp

`fatiando.gravmag.sphere.`**`tf`**(*xp*, *yp*, *zp*, *spheres*, *inc*, *dec*)

Calculate the total-field anomaly of spheres.

Note: Input units are SI. Output is in nT

Parameters:

- **xp, yp, zp** [arrays] The x, y, and z coordinates where the anomaly will be calculated
- **spheres** [list of `fatiando.mesher.Sphere`] The spheres. Spheres must have the properties `'magnetization'`, `'inclination'` and `'declination'`. If `'inclination'` and `'declination'` are not present, will use the values of *inc* and *dec* instead. Those without `'magnetization'` will be ignored.
- **inc** [float] The inclination of the regional field (in degrees)
- **dec** [float] The declination of the regional field (in degrees)

Returns:

- **tf** [array] The total-field anomaly

5.1.5 Forward modeling with tesseroids (`fatiando.gravmag.tesseroid`)

Calculates the potential fields of a tesseroid.

`fatiando.gravmag.tesseroid.`**`gx`**(*lons*, *lats*, *heights*, *tesseroids*, *dens=None*, *ratio=1.0*)

Calculate the x (North) component of the gravitational attraction due to a tesseroid model.

`fatiando.gravmag.tesseroid.`**`gxx`**(*lons*, *lats*, *heights*, *tesseroids*, *dens=None*, *ratio=3*)

Calculate the xx (North-North) component of the gravity gradient tensor due to a tesseroid model.

`fatiando.gravmag.tesseroid.`**`gxy`**(*lons*, *lats*, *heights*, *tesseroids*, *dens=None*, *ratio=3*)

Calculate the xy (North-East) component of the gravity gradient tensor due to a tesseroid model.

`fatiando.gravmag.tesseroid.`**`gxz`**(*lons*, *lats*, *heights*, *tesseroids*, *dens=None*, *ratio=3*)

Calculate the xz (North-radial) component of the gravity gradient tensor due to a tesseroid model.

`fatiando.gravmag.tesseroid.`**`gy`**(*lons*, *lats*, *heights*, *tesseroids*, *dens=None*, *ratio=1.0*)

Calculate the y (East) component of the gravitational attraction due to a tesseroid model.

`fatiando.gravmag.tesseroid.`**`gyy`**(*lons*, *lats*, *heights*, *tesseroids*, *dens=None*, *ratio=3*)

Calculate the yy (East-East) component of the gravity gradient tensor due to a tesseroid model.

`fatiando.gravmag.tesseroid.`**`gyz`**(*lons*, *lats*, *heights*, *tesseroids*, *dens=None*, *ratio=3*)

Calculate the yz (East-radial) component of the gravity gradient tensor due to a tesseroid model.

`fatiando.gravmag.tesseroid.`**`gz`**(*lons*, *lats*, *heights*, *tesseroids*, *dens=None*, *ratio=1.0*)
Calculate the z (radial) component of the gravitational attraction due to a tesseroid model.

`fatiando.gravmag.tesseroid.`**`gzz`**(*lons*, *lats*, *heights*, *tesseroids*, *dens=None*, *ratio=3*)
Calculate the zz (radial-radial) component of the gravity gradient tensor due to a tesseroid model.

`fatiando.gravmag.tesseroid.`**`potential`**(*lons*, *lats*, *heights*, *tesseroids*, *dens=None*, *ratio=1.0*)
Calculate the gravitational potential due to a tesseroid model.

5.1.6 Forward modeling with 2D polygons (`fatiando.gravmag.talwani`)

Calculate the gravitational attraction of a 2D body with polygonal vertical cross-section using the formula of Talwani et al. (1959)

Use the `Polygon` object to create polygons.

Warning: the vertices must be given clockwise! If not, the result will have an inverted sign.

Components

- `gz`

References

Talwani, M., J. L. Worzel, and M. Landisman (1959), Rapid Gravity Computations for Two-Dimensional Bodies with Application to the Mendocino Submarine Fracture Zone, J. Geophys. Res., 64(1), 49-59, doi:10.1029/JZ064i001p00049.

`fatiando.gravmag.talwani.`**`gz`**(*xp*, *zp*, *polygons*)
Calculates the g_z gravity acceleration component.

Note: The coordinate system of the input parameters is z -> **DOWN**.

Note: All input values in **SI** units(!) and output in **mGal**!

Parameters:

- **xp, zp** [arrays] The x and z coordinates of the computation points.
- **polygons** [list of `Polygon`] The density model used. Polygons must have the property `'density'`. Polygons that don't have this property will be ignored in the computations. Elements of *polygons* that are None will also be ignored.

  Note: The y coordinate of the polygons is used as z!

Returns:

- **gz** [array] The g_z component calculated on the computation points

5.1.7 Forward modeling of half a spherical shell (`fatiando.gravmag.half_sph_shell`)

Forward modeling of the gravitational effects of half a spherical shell.

Can only calculate the effects in a point located at the pole at different heights.

Components, gx, gy, gxy, gxz, and gyz are all equal to zero (0).

Functions:

- `potential`: the gravitational potential
- `gz`: the vertical component of the gravitational attraction
- `gxx`: the xx (North-North) component of the gravity gradient tensor
- `gyy`: the yy (East-East) component of the gravity gradient tensor
- `gzz`: the zz (radial-radial) component of the gravity gradient tensor

5.1.8 Inversion for the relief of 2D basins (`fatiando.gravmag.basin2d`)

Estimate the basement relief of two-dimensional basins from potential field data.

POLYGONAL PARAMETRIZATION

- `triangular`
- `trapezoidal`

Uses 2D bodies with a polygonal cross-section to parameterize the basin relief. Potential fields are calculated using the `fatiando.gravmag.talwani` module.

Warning: Vertices of polygons must always be in clockwise order!

Triangular basin

Use when the basin can be approximated by a 2D body with **triangular** vertical cross-section. The triangle is assumed to have 2 known vertices at the surface (the edges of the basin) and one unknown vertice in the subsurface. The inversion will then estimate the (x, z) coordinates of the unknown vertice.

Example using synthetic data:

```
>>> import numpy
>>> import fatiando as ft
>>> # Make a triangular basin model (will estimate the last point)
>>> verts = [(10000, 1), (90000, 1), (50000, 5000)]
>>> left, middle, right = verts
>>> model = ft.mesher.Polygon(verts, {'density':500})
>>> # Generate the synthetic gz profile
>>> xs = numpy.arange(0, 100000, 10000)
>>> zs = numpy.zeros_like(xs)
>>> gz = ft.gravmag.talwani.gz(xs, zs, [model])
>>> # Estimate the coordinates of the last point using Levenberg-Marquardt
>>> solver = ft.inversion.gradient.levmarq(initial=(10000, 1000))
>>> p, residuals = ft.gravmag.basin2d.triangular(xs, zs, gz, [left, middle],
...     500, solver)
>>> print '%.1f, %.1f' % (p.vertices[-1][0], p.vertices[-1][1])
50000.0, 5000.0
```

Same example but this time using `iterate=True` to view the steps of the algorithm:

```
>>> import numpy
>>> import fatiando as ft
>>> # Make a triangular basin model (will estimate the last point)
>>> verts = [(10000, 1), (90000, 1), (50000, 5000)]
>>> left, middle, right = verts
>>> model = ft.mesher.Polygon(verts, {'density':500})
>>> # Generate the synthetic gz profile
>>> xs = numpy.arange(0, 100000, 10000)
>>> zs = numpy.zeros_like(xs)
>>> gz = ft.gravmag.talwani.gz(xs, zs, [model])
>>> # Estimate the coordinates of the last point using Levenberg-Marquardt
>>> solver = ft.inversion.gradient.levmarq(initial=(70000, 2000))
>>> iterator = ft.gravmag.basin2d.triangular(xs, zs, gz, [left, middle], 500,
...                                          solver, iterate=True)
>>> for p, residuals in iterator:
...     print '%.4f, %.4f' % (p[0], p[1])
70000.0000, 2000.0000
69999.8803, 2005.4746
69998.6825, 2059.0979
69986.4671, 2502.6963
69843.9902, 3960.5022
67972.7679, 4728.4970
59022.3186, 4820.1359
50714.4178, 4952.5628
50001.0118, 4999.4345
50000.0006, 4999.9999
```

Trapezoidal basin

Use when the basin can be approximated by a 2D body with **trapezoidal** vertical cross-section. The trapezoid is assumed to have 2 known vertices at the surface (the edges of the basin) and two unknown vertice in the subsurface. We assume that the x coordinates of the unknown vertices are the same as the x coordinates of the known vertices (i.e., the unknown vertices are directly under the known vertices). The inversion will then estimate the z coordinates of the unknown vertices.

Example of inverting for the z coordinates of the unknown vertices:

```
>>> import numpy
>>> import fatiando as ft
>>> # Make a trapezoidal basin model (will estimate the last two point)
>>> verts = [(10000, 1), (90000, 1), (90000, 5000), (10000, 3000)]
>>> model = ft.mesher.Polygon(verts, {'density':500})
>>> # Generate the synthetic gz profile
>>> xs = numpy.arange(0, 100000, 10000)
>>> zs = numpy.zeros_like(xs)
>>> gz = ft.gravmag.talwani.gz(xs, zs, [model])
>>> # Estimate the coordinates of the two z coords using Levenberg-Marquardt
>>> solver = ft.inversion.gradient.levmarq(initial=(1000, 500))
>>> p, residuals = ft.gravmag.basin2d.trapezoidal(xs, zs, gz, verts[0:2], 500,
...                                               solver)
>>> print '%.1f, %.1f' % (p.vertices[-2][1], p.vertices[-1][1])
5000.0, 3000.0
```

Same example but this time using `iterate=True` to view the steps of the algorithm:

```
>>> import numpy
>>> import fatiando as ft
>>> # Make a trapezoidal basin model (will estimate the last two point)
>>> verts = [(10000, 5), (90000, 10), (90000, 5000), (10000, 3000)]
>>> model = ft.mesher.Polygon(verts, {'density':500})
>>> # Generate the synthetic gz profile
>>> xs = numpy.arange(0, 100000, 10000)
>>> zs = numpy.zeros_like(xs)
>>> gz = ft.gravmag.talwani.gz(xs, zs, [model])
>>> # Estimate the coordinates of the two z coords using Levenberg-Marquardt
>>> solver = ft.inversion.gradient.levmarq(initial=(1000, 500))
>>> iterator = ft.gravmag.basin2d.trapezoidal(xs, zs, gz, verts[0:2], 500,
...                                           solver, iterate=True)
>>> for p, residuals in iterator:
...     print '%.4f, %.4f' % (p[0], p[1])
1000.0000, 500.0000
1010.4375, 509.4191
1111.6975, 600.5546
1888.0846, 1281.9163
3926.6071, 2780.5317
```

```
4903.8174, 3040.3444
4998.6977, 3001.0087
4999.9980, 3000.0017
5000.0000, 2999.9999
```

---

class `fatiando.gravmag.basin2d.`**`TrapezoidalGzDM`**(*xp*, *zp*, *data*, *verts*, *density*, *delta=1.0*)

Bases: `fatiando.inversion.datamodule.DataModule`

Data module for the inversion to estimate the relief of a trapezoidal basin.

Packs the necessary data and interpretative model information.

The forward modeling is done using `talwani`. Derivatives are calculated using a 2-point finite difference approximation. The Hessian matrix is calculated using a Gauss-Newton approximation.

Parameters:

- **xp, zp** [array] Arrays with the x and z coordinates of the profile data points
- **data** [array] The profile gravity anomaly data
- **verts** [list of lists] List of the [x, z] coordinates of the two know vertices.

  > **Warning:** Very important that the vertices in the list be ordered from left to right! Otherwise the forward model will give results with an inverted sign and terrible things may happen!

- **density** [float] Density contrast of the basin
- **delta** [float] Interval used to calculate the approximate derivatives

> **Warning:** It is very important that the vertices in the list be ordered clockwise! Otherwise the forward model will give results with an inverted sign and terrible things may happen!

class `fatiando.gravmag.basin2d.`**`TriangularGzDM`**(*xp*, *zp*, *data*, *verts*, *density*, *delta=1.0*)

Bases: `fatiando.inversion.datamodule.DataModule`

Data module for the inversion to estimate the relief of a triangular basin.

Packs the necessary gravity anomaly data and interpretative model information.

The forward modeling is done using `talwani`. Derivatives are calculated using a 2-point finite difference approximation. The Hessian matrix is calculated using a Gauss-Newton approximation.

Parameters:

- **xp, zp** [array] Arrays with the x and z coordinates of the profile data points
- **data** [array] The profile gravity anomaly data
- **verts** [list of lists] List of the [x, z] coordinates of the two know vertices.

  > **Warning:** Very important that the vertices in the list be ordered from left to right! Otherwise the forward model will give results with an inverted sign and terrible things may happen!

- **density** [float] Density contrast of the basin
- **delta** [float] Interval used to calculate the approximate derivatives

> **Warning:** It is very important that the vertices in the list be ordered clockwise! Otherwise the forward model will give results with an inverted sign and terrible things may happen!

`fatiando.gravmag.basin2d.`**`trapezoidal`**(*xp*, *zp*, *data*, *verts*, *density*, *solver*, *iterate=False*)

Estimate basement relief of a triangular basin. The basin is modeled as a triangle with two known vertices at the surface. The parameters estimated are the x and z coordinates of the third vertice.

Parameters:

- **xp, zp** [array] Arrays with the x and z coordinates of the profile data points
- **data** [array] The profile gravity anomaly data
- **verts** [list of lists] List of the [x, z] coordinates of the two know vertices.

  Warning: Very important that the vertices in the list be ordered from left to right! Otherwise the forward model will give results with an inverted sign and terrible things may happen!

- **density** [float] Density contrast of the basin
- **solver** [function] A non-linear inverse problem solver generated by a factory function from the `fatiando.inversion.gradient` package.
- **iterate** [True or False] If True, will yield the current estimate at each iteration yielded by *solver*. In Python terms, `iterate=True` transforms this function into a generator function.

Returns:

- results : list = [estimate, residuals]:
  - **estimate** [array or `fatiando.mesher.Polygon`] If `iterate==False`, will return a Polygon, else will yield the estimated [z1, z2] coordinates of the bottom vertices.
  - **residuals** [array] The residuals of the inversion (difference between measured and predicted data)

`fatiando.gravmag.basin2d.`**`triangular`**(*xp*, *zp*, *data*, *verts*, *density*, *solver*, *iterate=False*)

Estimate basement relief of a triangular basin. The basin is modeled as a triangle with two known vertices at the surface. The parameters estimated are the x and z coordinates of the third vertice.

Parameters:

- **xp, zp** [array] Arrays with the x and z coordinates of the profile data points
- **data** [array] The profile gravity anomaly data
- **verts** [list of lists] List of the [x, z] coordinates of the two know vertices.

  Warning: Very important that the vertices in the list be ordered from left to right! Otherwise the forward model will give results with an inverted sign and terrible things may happen!

- **density** [float] Density contrast of the basin
- **solver** [function] A non-linear inverse problem solver generated by a factory function from the `fatiando.inversion.gradient` package.
- **iterate** [True or False] If True, will yield the current estimate at each iteration yielded by *solver*. In Python terms, `iterate=True` transforms this function into a generator function.

Returns:

- results : list = [estimate, residuals]:
  - **estimate** [array or `fatiando.mesher.Polygon`] If `iterate==False`, will return a Polygon, else will yield the estimated [x, z] coordinates of the missing vertice
  - **residuals** [array] The residuals of the inversion (difference between measured and predicted data)

5.1.9 Processing and transformations using the FFT (`fatiando.gravmag.fourier`)

Potential field processing using the Fast Fourier Transform

Note: Requires gridded data to work!

Derivatives

- `derivx`: Calculate the n-th order derivative of a potential field in the x-direction
- `derivy`: Calculate the n-th order derivative of a potential field in the y-direction
- `derivz`: Calculate the n-th order derivative of a potential field in the z-direction

---

`fatiando.gravmag.fourier.`**`derivx`**(*x*, *y*, *data*, *shape*, *order=1*)

Calculate the derivative of a potential field in the x direction.

Warning: If the data is not in SI units, the derivative will be in strange units! I strongly recommend converting the data to SI **before** calculating the derivative (use one of the unit conversion functions of `fatiando.utils`). This way the derivative will be in SI units and can be easily converted to what unit you want.

Parameters:

- **x, y** [1D-arrays] The x and y coordinates of the grid points
- **data** [1D-array] The potential field at the grid points
- **shape** [tuple = (ny, nx)] The shape of the grid
- **order** [int] The order of the derivative

Returns:

- **deriv** [1D-array] The derivative

`fatiando.gravmag.fourier.`**`derivy`**(*x*, *y*, *data*, *shape*, *order=1*)

Calculate the derivative of a potential field in the y direction.

Warning: If the data is not in SI units, the derivative will be in strange units! I strongly recommend converting the data to SI **before** calculating the derivative (use one of the unit conversion functions of `fatiando.utils`). This way the derivative will be in SI units and can be easily converted to what unit you want.

Parameters:

- **x, y** [1D-arrays] The x and y coordinates of the grid points
- **data** [1D-array] The potential field at the grid points
- **shape** [tuple = (ny, nx)] The shape of the grid
- **order** [int] The order of the derivative

Returns:

- **deriv** [1D-array] The derivative

`fatiando.gravmag.fourier.`**`derivz`**(*x*, *y*, *data*, *shape*, *order=1*)

Calculate the derivative of a potential field in the z direction.

Warning: If the data is not in SI units, the derivative will be in strange units! I strongly recommend converting the data to SI **before** calculating the derivative (use one of the unit conversion functions of `fatiando.utils`). This way the derivative will be in SI units and can be easily converted to what unit you want.

Parameters:

- **x, y** [1D-arrays] The x and y coordinates of the grid points
- **data** [1D-array] The potential field at the grid points
- **shape** [tuple = (ny, nx)] The shape of the grid
- **order** [int] The order of the derivative

Returns:

- **deriv** [1D-array] The derivative

5.1.10 Imaging methods for potential fields (`fatiando.gravmag.imaging`)

Imaging methods for potential fields.

Implements some of the methods described in Fedi and Pilkington (2012). Most methods convert the observed data (gravity, magnetic, etc) into a physical property distribution (density, magnetization, etc). Most methods require gridded data to work.

- `geninv`: The Generalized Inverse solver in the frequency domain (Cribb, 1976)
- `sandwich`: Sandwich model (Pedersen, 1991). Uses depth weighting as in Pilkington (1997)
- `migrate`: 3D potential field migration (Zhdanov et al., 2011). Actually uses the formula of Fedi and Pilkington (2012), which are comprehensible.

Warning: Most of these methods provide estimates of physical property values that are completely out of scale (mostly due to depth weighting). Therefore, I don't recommend using the actual values of the physical properties for anything other than finding an approximate location for the sources.

Note: If you want the estimate physical property values in SI units, you must pass the data also in SI units! Use the unit conversion functions in `fatiando.utils`

References

Cribb, J. (1976), Application of the generalized linear inverse to the inversion of static potential data, Geophysics, 41(6), 1365, doi:10.1190/1.1440686

Fedi, M., and M. Pilkington (2012), Understanding imaging methods for potential field data, Geophysics, 77(1), G13, doi:10.1190/geo2011-0078.1

Pedersen, L. B. (1991), Relations between potential fields and some equivalent sources, Geophysics, 56(7), 961, doi:10.1190/1.1443129

Pilkington, M. (1997), 3-D magnetic imaging using conjugate gradients, Geophysics, 62(4), 1132, doi:10.1190/1.1444214

Zhdanov, M. S., X. Liu, G. A. Wilson, and L. Wan (2011), Potential field migration for rapid imaging of gravity gradiometry data, Geophysical Prospecting, 59(6), 1052-1071, doi:10.1111/j.1365-2478.2011.01005.x

`fatiando.gravmag.imaging.`**`geninv`**(*x*, *y*, *z*, *data*, *shape*, *zmin*, *zmax*, *nlayers*)

Generalized Inverse imaging in the frequency domain (Cribb, 1976).

Calculates a physical property distribution given potential field data on a **regular grid**.

Note: Only works on **gravity** data for now.

Note: The data **must** be leveled, i.e., on the same height!

Note: The coordinate system adopted is x->North, y->East, and z->Down

Warning: The Generalized Inverse does **not** use depth weights. This means that the solution will tend to be concentrated on the surface!

Parameters:

- **x, y** [1D-arrays] The x and y coordinates of the grid points
- **z** [float or 1D-array] The z coordinate of the grid points
- **data** [1D-array] The potential field at the grid points
- **shape** [tuple = (ny, nx)] The shape of the grid
- **zmin, zmax** [float] The top and bottom, respectively, of the region where the physical property distribution is calculated
- **nlayers** [int] The number of layers used to divide the region where the physical property distribution is calculated

Returns:

- **mesh** [`fatiando.mesher.PrismMesh`] The estimated physical property distribution set in a prism mesh (for easy 3D plotting)

`fatiando.gravmag.imaging.`**`migrate`**(*x*, *y*, *z*, *gz*, *zmin*, *zmax*, *meshshape*, *power=0.5*, *scale=1*)

3D potential field migration (Zhdanov et al., 2011).

Actually uses the formula of Fedi and Pilkington (2012), which are comprehensible.

Note: Only works on **gravity** data for now.

Note: The data **do not** need to be leveled or on a regular grid.

Note: The coordinate system adopted is x->North, y->East, and z->Down

Parameters:

- **x, y** [1D-arrays] The x and y coordinates of the grid points
- **z** [float or 1D-array] The z coordinate of the grid points
- **gz** [1D-array] The gravity anomaly data at the grid points
- **zmin, zmax** [float] The top and bottom, respectively, of the region where the physical property distribution is calculated
- **meshshape** [tuple = (nz, ny, nx)] Number of prisms in the output mesh in the x, y, and z directions, respectively
- **power** [float] The power law used for the depth weighting. This controls what depth the bulk of the solution will be.

- **scale** [float] A scale factor for the depth weights. Simply changes the scale of the physical property values.

Returns:

- **mesh** [`fatiando.mesher.PrismMesh`] The estimated physical property distribution set in a prism mesh (for easy 3D plotting)

`fatiando.gravmag.imaging.`**`sandwich`**(*x*, *y*, *z*, *data*, *shape*, *zmin*, *zmax*, *nlayers*, *power=0.5*)

Sandwich model (Pedersen, 1991).

Calculates a physical property distribution given potential field data on a **regular grid**. Uses depth weights.

Note: Only works on **gravity** data for now.

Note: The data **must** be leveled, i.e., on the same height!

Note: The coordinate system adopted is x->North, y->East, and z->Down

---

Parameters:

- **x, y** [1D-arrays] The x and y coordinates of the grid points
- **z** [float or 1D-array] The z coordinate of the grid points
- **data** [1D-array] The potential field at the grid points
- **shape** [tuple = (ny, nx)] The shape of the grid
- **zmin, zmax** [float] The top and bottom, respectively, of the region where the physical property distribution is calculated
- **nlayers** [int] The number of layers used to divide the region where the physical property distribution is calculated
- **power** [float] The power law used for the depth weighting. This controls what depth the bulk of the solution will be.

Returns:

- **mesh** [`fatiando.mesher.PrismMesh`] The estimated physical property distribution set in a prism mesh (for easy 3D plotting)

5.1.11 Utilities for operating on the gradient tensor (`fatiando.gravmag.tensor`)

Utilities for operating on the gradient tensor of potential fields.

Functions

- `invariants`: Calculates the first (I_1), second (I_2), and dimensionless (I) invariants
- `eigen`: Calculates the eigenvalues and eigenvectors of the an array of gradient tensor measurements
- `center_of_mass`: Estimate the center of mass of sources from the first eigenvector using the method of Beiki and Pedersen (2010)

Theory

Following Pedersen and Rasmussen (1990), the characteristic polynomail of the gravity gradient tensor $\mathbf{\Gamma}$ is

$$\lambda^3 + I_1\lambda - I_2 = 0$$

where λ is an eigen value and I_1 and I_2 are the two invariants. The dimensionless invariant I is

$$I = -\frac{(I_2/2)^2}{(I_1/3)^3}$$

The invariant I indicates the dimensionality of the source. $I = 0$ for 2 dimensional bodies and $I = 1$ for a monopole.

References

Beiki, M., and L. B. Pedersen (2010), Eigenvector analysis of gravity gradient tensor to locate geologic bodies, Geophysics, 75(6), I37, doi:10.1190/1.3484098

Pedersen, L. B., and T. M. Rasmussen (1990), The gradient tensor of potential field anomalies: Some implications on data collection and data processing of maps, Geophysics, 55(12), 1558, doi:10.1190/1.1442807

`fatiando.gravmag.tensor.`**`center_of_mass`**`(`*x, y, z, eigvec1, windows=1, wcenter=None, wmin=None, wmax=None*`)`

Estimates the center of mass of a source using the method of Beiki and Pedersen (2010).

Uses an expanding window to get the best estimate and deal with multiple sources.

Parameters:

- **x, y, z** [arrays] The x, y, and z coordinates of the observation points
- **eigvec1** [array (shape = (N, 3) where N is the number of observations)] The first eigenvector of the gravity gradient tensor at each observation point
- **windows** [int] The number of expanding windows to use
- **wcenter** [list = [x, y]] The [x, y] coordinates of the center of the expanding windows. Will default to the middle of the data area if None
- **wmin, wmax** [float] Minimum and maximum size of the expanding windows. Will default to 10% data area and 100% data area, respectively, if None

Returns:

- **[xo, yo, zo], sigma** [float] xo, yo, zo are the coordinates of the estimated center of mass. sigma is the estimated standard deviation of the distances between the estimated center of mass and the lines passing through the observation point in the direction of the eigenvector

Example:

```
>>> import fatiando as ft
>>> # Generate synthetic data using a prism
>>> prism = ft.mesher.Prism(-200,0,-100,100,0,200,{'density':1000})
>>> x, y, z = ft.gridder.regular((-500,500,-500,500), (20,20), z=-100)
>>> tensor = [ft.gravmag.prism.gxx(x, y, z, [prism]),
...           ft.gravmag.prism.gxy(x, y, z, [prism]),
...           ft.gravmag.prism.gxz(x, y, z, [prism]),
...           ft.gravmag.prism.gyy(x, y, z, [prism]),
...           ft.gravmag.prism.gyz(x, y, z, [prism]),
...           ft.gravmag.prism.gzz(x, y, z, [prism])]
>>> # Get the eigenvector
>>> eigenvals, eigenvecs = ft.gravmag.tensor.eigen(tensor)
>>> # Now estimate the center of mass
>>> cm, sigma = ft.gravmag.tensor.center_of_mass(x, y, z, eigenvecs[0])
>>> xo, yo, zo = cm
>>> print "%.2lf, %.2lf, %.2lf" % (xo, yo, zo)
-100.05, 0.00, 99.86
```

`fatiando.gravmag.tensor.`**`eigen`**`(`*tensor*`)`

Calculates the eigenvalues and eigenvectors of the gradient tensor.

Note: The coordinate system used is x->North, y->East, z->Down

Parameters:

- **tensor** [list] A list of arrays with the 6 components of the gradient tensor measured on a set of points. The order of the list should be: [gxx, gxy, gxz, gyy, gyz, gzz]

Returns:

- **result** [list = [[eigval1, eigval2, eigval3], [eigvec1, eigvec2, eigvec3]]] The eigenvalues and eigenvectors at each observation point.
  - **eigval1,2,3** [array] The first, second, and third eigenvalues
  - **eigvec1,2,3** [array (shape = (N, 3) where N is the number of points)] The first, second, and third eigenvectors

Example:

```
>>> tensor = [[2], [0], [0], [3], [0], [1]]
>>> eigenvals, eigenvecs = eigen(tensor)
>>> print eigenvals[0], eigenvecs[0]
[ 3.] [[ 0.  1.  0.]]
>>> print eigenvals[1], eigenvecs[1]
[ 2.] [[ 1.  0.  0.]]
>>> print eigenvals[2], eigenvecs[2]
[ 1.] [[ 0.  0.  1.]]
```

`fatiando.gravmag.tensor.`**`invariants`**(*tensor*)

Calculates the first, second, and dimensionless invariants of the gradient tensor.

Note: The coordinate system used is x->North, y->East, z->Down

Parameters:

- **tensor** [list] A list of arrays with the 6 components of the gradient tensor measured on a set of points. The order of the list should be: [gxx, gxy, gxz, gyy, gyz, gzz]

Returns:

- **invariants** [list = [I_1, I_2, I]] The invariants calculated for each point

5.1.12 3D Euler deconvolution methods (`fatiando.gravmag.euler`)

Euler deconvolution methods for potential fields.

- `expanding_window`: Run a given solver for the Euler deconvolution on an expanding window and return the best estimate
- `classic`: The classic solution to Euler's equation for potential fields

`fatiando.gravmag.euler.`**`classic`**(*xp*, *yp*, *zp*, *field*, *xderiv*, *yderiv*, *zderiv*, *index*)

Classic 3D Euler deconvolution of potential field data.

Works on any potential field that satisfies Euler's homogeneity equation.

Parameters:

- **xp, yp, zp** [arrays] The x, y, and z coordinates of the observation points
- **field** [array] The potential field measured at the observation points
- **xderiv, yderiv, zderiv** [arrays] The x-, y-, and z-derivatives of the potential field measured (calculated) at the observation points

- **index** [float] The structural index of the source

Returns:

- **results** [dict] The results in a dictionary:

```
{'point':[x, y, z], # The estimated coordinates of source
 'baselevel':baselevel, # The estimated baselevel
 'mean error':mean_error, # The mean error of the estimated location
 'uncertainty':[sx, sy, sz] # The uncertainty in x, y, and z
}
```

Note: The uncertainty estimate is not very reliable.

`fatiando.gravmag.euler.`**`expanding_window`**(*xp*, *yp*, *zp*, *field*, *xderiv*, *yderiv*, *zderiv*, *index*, *euler*, *center*, *minsize*, *maxsize*, *nwindows=20*)

Perform the Euler deconvolution on windows of growing size and return the best estimate.

Parameters:

- **xp, yp, zp** [arrays] The x, y, and z coordinates of the observation points of the **whole** data set
- **field** [array] The potential field measured at the observation points of the **whole** data set
- **xderiv, yderiv, zderiv** [arrays] The x-, y-, and z-derivatives of the potential field measured (calculated) at the observation points of the **whole** data set
- **index** [float] The structural index of the source
- **euler** [function] The Euler deconvolution solver function (like `fatiando.gravmag.euler.classic`)
- **center** [[x, y]] The coordinates of the center of the expanding window
- **minsize, maxsize** [floats] The minimum and maximum size of the expanding window
- **nwindows** [int] Number of windows between minsize and maxsize

Returns:

- **results** [dict] The best results in a dictionary:

```
{'point':[x, y, z], # The estimated coordinates of source
 'baselevel':baselevel, # The estimated baselevel
 'mean error':mean_error, # The mean error of the estimated location
 'uncertainty':[sx, sy, sz] # The uncertainty in x, y, and z
}
```

5.1.13 Space domain processing (`fatiando.gravmag.transform`)

Space domain potential field transformations, like upward continuation, derivatives and total mass.

Transformations

- `upcontinue`: Upward continuation of the vertical component of gravity g_z using numerical integration

`fatiando.gravmag.transform.`**`upcontinue`**(*gz*, *height*, *xp*, *yp*, *dims*)

Upward continue g_z data using numerical integration of the analytical formula:

$$g_z(x, y, z) = \frac{z - z_0}{2\pi} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} g_z(x', y', z_0) \frac{1}{[(x - x')^2 + (y - y')^2 + (z - z_0)^2]^{\frac{3}{2}}} dx' dy'$$

Note: Data needs to be on a regular grid!

Note: Units are SI for all coordinates and mGal for g_z

Note: be aware of coordinate systems! The *x*, *y*, *z* coordinates are: x -> North, y -> East and z -> **DOWN**.

Parameters:

- **gz** [array] The gravity values on the grid points
- **height** [float] How much higher to move the gravity field (should be POSITIVE!)
- **xp, yp** [arrays] The x and y coordinates of the grid points
- **dims** [list = [dy, dx]] The grid spacing in the y and x directions

Returns:

- **gzcont** [array] The upward continued g_z

5.2 Seismics and seismology (`fatiando.seismic`)

Various tools for seismic and seismology, like direct modeling, inversion (tomography), epicenter determination, etc.

Forward modeling and inversion

- `ttime2d`: 2D seismic ray travel-time modeling
- `epic2d`: 2D epicenter determination
- `profile`: Modeling and inversion of seismic profiling
- `wavefd`: Finite difference solution of the 2D elastic wave equation

Tomography

- `srtomo`: 2D straight-ray tomography

5.2.1 2D seismic ray travel-time modeling (`fatiando.seismic.ttime2d`)

Calculate travel-times of seismic waves in 2D.

- `straight`: Calculate the travel-time of a straight ray through a mesh of square cells

`fatiando.seismic.ttime2d.`**`straight`**(*cells*, *prop*, *srcs*, *recs*, *velocity=None*, *par=False*)

Calculate the travel times inside a mesh of square cells between source and receiver pairs assuming the rays are straight lines (no refraction or reflection).

Note: Don't care about the units as long they are compatible.

For a homogeneous model, *cells* can be a list with only one big cell.

Parameters:

- **cells** [list of `fatiando.mesher.Square`] The velocity model to use to trace the straight rays. Cells must have the physical property given in parameter *prop*. This will be used as the velocity of each cell. (*cells* can also be a `SquareMesh`)
- **prop** [str] Which physical property of the cells to use as velocity. Normaly one would choose `'vp'` or `'vs'`
- **srcs** [list fo lists] List with [x, y] coordinate pairs of the wave sources.
- **recs** [list fo lists] List with [x, y] coordinate pairs of the receivers sources
- **velocity** [float or None] If not None, will use this value instead of the prop of cells as the velocity. Useful when building sensitivity matrices (use velocity = 1).
- **par** [True or False] If True, will run the calculations in parallel using all the cores available. Not recommended for Jacobian matrix building!

srcs and *recs* are lists of source-receiver pairs. Each source in *srcs* is associated with the corresponding receiver in *recs* for a given travel time.

For example:

```
>>> # One source was recorded at 3 receivers.
>>> # The medium is homogeneous and can be
>>> # represented by a single Square
>>> from fatiando.mesher import Square
>>> cells = [Square([0, 10, 0, 10], {'vp':2})]
>>> src = (5, 0)
>>> srcs = [src, src, src]
>>> recs = [(0, 0), (5, 10), (10, 0)]
>>> print straight(cells, 'vp', srcs, recs)
[ 2.5  5.   2.5]
```

Returns:

- **times** [array] The total times each ray took to get from a source to a receiver (in compatible units with *prop*)

5.2.2 2D epicenter determination (`fatiando.seismic.epic2d`)

Epicenter determination in 2D, i.e., assuming a flat Earth.

Homogeneous Earth

- `homogeneous`

Estimates the (x, y) cartesian coordinates of the epicenter based on travel-time residuals between S and P waves, assuming a homogeneous velocity distribution.

Example using synthetic data:

```
>>> import fatiando as ft
>>> # Generate synthetic travel-time residuals
>>> area = (0, 10, 0, 10)
>>> vp = 2
>>> vs = 1
>>> model = [ft.mesher.Square(area, props={'vp':vp, 'vs':vs})]
>>> # The true source (epicenter)
>>> src = (5, 5)
>>> recs = [(5, 0), (5, 10), (10, 0)]
>>> srcs = [src, src, src]
>>> ptime = ft.seismic.ttime2d.straight(model, 'vp', srcs, recs)
>>> stime = ft.seismic.ttime2d.straight(model, 'vs', srcs, recs)
>>> ttres = stime - ptime
>>> # Solve using Newton's
>>> # Generate synthetic travel-time residuals method
>>> solver = ft.inversion.gradient.newton(initial=(1, 1), tol=10**(-3),
...     maxit=1000)
>>> # Estimate the epicenter
>>> p, residuals = ft.seismic.epic2d.homogeneous(ttres, recs, vp, vs, solver)
>>> print "(%.4f, %.4f)" % (p[0], p[1])
(5.0000, 5.0000)
```

Example using `iterate = True` to step through the solver algorithm:

```
>>> import fatiando as ft
>>> # Generate synthetic travel-time residuals
>>> area = (0, 10, 0, 10)
>>> vp = 2
>>> vs = 1
>>> model = [ft.mesher.Square(area, props={'vp':vp, 'vs':vs})]
>>> # The true source (epicenter)
>>> src = (5, 5)
>>> recs = [(5, 0), (5, 10), (10, 0)]
>>> srcs = [src, src, src]
>>> ptime = ft.seismic.ttime2d.straight(model, 'vp', srcs, recs)
>>> stime = ft.seismic.ttime2d.straight(model, 'vs', srcs, recs)
>>> ttres = stime - ptime
>>> # Solve using Newton's
>>> # Generate synthetic travel-time residuals method
>>> solver = ft.inversion.gradient.newton(initial=(1, 1), tol=10**(-3),
...     maxit=5)
>>> # Show the steps to estimate the epicenter
>>> steps = ft.seismic.epic2d.homogeneous(ttres, recs, vp, vs, solver,
...     iterate=True)
>>> for p, r in steps:
...     print "(%.4f, %.4f)" % (p[0], p[1])
(1.0000, 1.0000)
(2.4157, 5.8424)
(4.3279, 4.7485)
(4.9465, 4.9998)
(4.9998, 5.0000)
(5.0000, 5.0000)
```

class `fatiando.seismic.epic2d.`**`TTRFlat`**(*ttres*, *recs*, *vp*, *vs*)

Bases: `fatiando.inversion.datamodule.DataModule`

Data module for epicenter estimation using travel-time residuals between S and P waves, assuming a homogeneous Earth.

The travel-time residual measured by the ith receiver is a function of the (x, y) coordinates of the epicenter:

$$t_{S_i} - t_{P_i} = \Delta t_i(x, y) = \left(\frac{1}{V_S} - \frac{1}{V_P}\right)\sqrt{(x_i - x)^2 + (y_i - y)^2}$$

The elements G_{i1} and G_{i2} of the Jacobian matrix for this data type are

$$G_{i1}(x, y) = -\left(\frac{1}{V_S} - \frac{1}{V_P}\right)\frac{x_i - x}{\sqrt{(x_i - x)^2 + (y_i - y)^2}}$$

$$G_{i2}(x, y) = -\left(\frac{1}{V_S} - \frac{1}{V_P}\right)\frac{y_i - y}{\sqrt{(x_i - x)^2 + (y_i - y)^2}}$$

The Hessian matrix is approximated by $2\bar{\bar{G}}^T\bar{\bar{G}}$ (Gauss-Newton method).

Parameters:

- **ttres** [array] Travel-time residuals between S and P waves

- **recs** [list of lists] List with the (x, y) coordinates of the receivers
- **vp** [float] Assumed velocity of P waves
- **vs** [float] Assumed velocity of S waves

`fatiando.seismic.epic2d.`**`homogeneous`**(*ttres*, *recs*, *vp*, *vs*, *solver*, *damping=0.0*, *equality=0.0*, *ref={}*, *iterate=False*)

Estimate the (x, y) coordinates of the epicenter of an event using travel-time residuals between P and S waves and assuming a homogeneous Earth

Parameters:

- **ttres** [array] Array with the travel-time residuals between S and P waves
- **recs** [list of lists] List with the (x, y) coordinates of the receivers
- **vp** [float] Assumed velocity of P waves
- **vs** [float] Assumed velocity of S waves
- **solver** [function] A non-linear inverse problem solver generated by a factory function from a `inversion` inverse problem solver module.
- **damping** [float] Damping regularizing parameter (i.e., how much damping to apply). Must be a positive scalar.
- **equality** [float] Positive scalar regularizing parameter for Equality constraints. How much to impose the reference values on the coordinates of the epicenter. A high value means strong constraints, a small value means loose constraints. (see `Equality`).
- **ref** [dict] The reference values for the x or y coordinates (or both). For example, to impose that x be close to 8, use `ref = {'x':8}`. To impose a constraint on both x and y, use `ref = {'x':8, 'y':12}`.
- **iterate** [True or False] If True, will yield the current estimate at each iteration yielded by *solver*. In Python terms, `iterate=True` transforms this function into a generator function.

Returns:

- **results** [list = [estimate, residuals]] The estimated (x, y) coordinates and the residuals (difference between measured and predicted travel-time residuals)

`fatiando.seismic.epic2d.`**`mapgoal`**(*xs*, *ys*, *ttres*, *recs*, *vp*, *vs*, *damping=0.0*, *equality=0.0*, *ref={}*)

Make a map of the goal function for a given set of inversion parameters.

The goal function is define as:

$$\Gamma(\bar{p}) = \phi(\bar{p}) + \sum_{r=1}^{R} \mu_r \theta_r(\bar{p})$$

where $\phi(\bar{p})$ is the data-misfit function, $\theta_r(\bar{p})$ is the rth of the R regularizing function used, and μ_r is the rth regularizing parameter.

Parameters:

- **xs, ys** [arrays] Lists of x and y values where the goal function will be calculated
- **ttres** [array] Array with the travel-time residuals between S and P waves
- **recs** [list of lists] List with the (x, y) coordinates of the receivers
- **vp** [float] Assumed velocity of P waves
- **vs** [float] Assumed velocity of S waves
- **damping** [float] Damping regularizing parameter (i.e., how much damping to apply). Must be a positive scalar.

- **equality** [float] Positive scalar regularizing parameter for Equality constraints. How much to impose the reference values on the coordinates of the epicenter. A high value means strong constraints, a small value means loose constraints. (see `Equality`).
- **ref** [dict] The reference values for the x or y coordinates (or both). For example, to impose that x be close to 8, use `ref = {'x':8}`. To impose a constraint on both x and y, use `ref = {'x':8, 'y':12}`.

Returns:

- **goals** [array] Array with the goal function values

5.2.3 Seismic profiling (`fatiando.seismic.profile`)

Forward modeling and inversion of seismic profiling data.

Vertical seismic profiling

- `vertical`
- `ivertical`

Model and invert vertical seismic profiling data. In this kind of profiling, the wave source is located at the surface on top of the well. The travel-times of first arrivals is then measured at different depths along the well. The ith travel-time t_i measured at depth z_i is a function of the wave velocity v_j and distance d_{ij} that it traveled in each layer

$$t_i(z_i) = \sum_{j=1}^{M} \frac{d_{ij}}{v_j}$$

The distance d_{ij} is smaller or equal to the thickness of the layer s_j. Notice that $d_{ij} = 0$ if the jth layer is bellow z_i, $d_{ij} = s_j$ if the jth layer is above z_i, and $d_{ij} < s_j$ if z_i is inside the jth layer.

To generate synthetic seismic profiling data, use `vertical` like so:

```
>>> import fatiando as ft
>>> # Make the synthetic 4 layer model
>>> thicks = [10, 20, 10, 30]
>>> vels = [2, 4, 10, 5]
>>> # Make an array with the z_i
>>> zs = [10, 30, 40, 70]
>>> # Calculate the travel-times
>>> for t in ft.seismic.profile.vertical(thicks, vels, zs):
...     print '%.1f' % (t),
5.0 10.0 11.0 17.0
```

To make t_i linear with respect to v_j, we can use *slowness* w_j instead of velocity

$$t_i(z_i) = \sum_{j=1}^{M} d_{ij} w_j$$

This allows us to easily invert for the slowness of each layer, given their thickness. Here's an example of using `ivertical` to do this on some synthetic data:

```
>>> import numpy
>>> import fatiando as ft
>>> # Make the synthetic 4 layer model
>>> thicks = [10, 20, 10, 30]
>>> vels = [2, 4, 10, 8]
>>> # Make an array with the z_i
>>> zs = numpy.arange(5, sum(thicks), 1)
>>> # Calculate the travel-times
>>> tts = ft.seismic.profile.vertical(thicks, vels, zs)
>>> # Make a linear solver and solve for the slowness
>>> solver = ft.inversion.linear.overdet(nparams=len(thicks))
>>> p, residuals = ft.seismic.profile.ivertical(tts, zs, thicks, solver)
>>> for slow in p:
...     print '%.1f' % (1./slow),
2.0 4.0 10.0 8.0
```

class `fatiando.seismic.profile.`**`VerticalSlownessDM`**(*traveltimes*, *zp*, *thickness*)

Bases: `fatiando.inversion.datamodule.DataModule`

Data module for a vertical seismic profile first-arrival travel-time data. Assumes that only the slowness of the layers are parameters in the inversion.

In this case, the inverse problem in linear. The element G_{ij} of the Jacobian (sensitivity) matrix is given by

$$G_{ij} = d_{ij}$$

where d_{ij} is the distance that the ith first-arrival traveled inside the jth layer.

Uses `fatiando.seismic.ttime2d.straight` for forward modeling to build the Jacobian matrix.

Parameters:

- **traveltimes** [list] The first-arrival travel-times calculated at the measurement stations
- **zp** [list] The depths of the measurement stations (seismometers)
- **thickness** [list] The thickness of each layer in order of increasing depth

`fatiando.seismic.profile.`**`ivertical`**(*traveltimes*, *zp*, *thickness*, *solver=None*, *damping=0.0*, *smooth=0.0*, *sharp=0.0*, *beta=1e-10*, *iterate=False*)

Invert first-arrival travel-time data for the slowness of each layer.

Parameters:

- **traveltimes** [array] The first-arrival travel-times calculated at the measurement stations
- **zp** [list] The depths of the measurement stations (seismometers)
- **thickness** [list] The thickness of each layer in order of increasing depth
- **solver** [function or None] A linear or non-linear inverse problem solver generated by a factory function from a module of package `inversion`. If None, will use a default solver function.
- **damping** [float] Damping regularizing parameter (i.e., how much damping to apply). Must be a positive scalar.
- **smooth** [float] Smoothness regularizing parameter (i.e., how much smoothness to apply). Must be a positive scalar.
- **sharp** [float] Sharpness (total variation) regularizing parameter (i.e., how much sharpness to apply). Must be a positive scalar.

  Note: Total variation regularization doesn't seem to work for this problem

- **beta** [float] Total variation parameter. See `TotalVariation` for details
- **iterate** [True or False] If True, will yield the current estimate at each iteration yielded by *solver*. In Python terms, `iterate=True` transforms this function into a generator function.

Returns:

- results : list = [slowness, residuals]:
  - **slowness** [array] The slowness of each layer
  - **residuals** [array] The inversion residuals (observed travel-times minus predicted travel-times by the slowness estimate)

`fatiando.seismic.profile.`**`vertical`**(*thickness*, *velocity*, *zp*)

Calculates the first-arrival travel-times for given a layered model. Simulates a vertical seismic profile.

The source is assumed to be at z = 0. The z-axis is positive downward.

Parameters:

- **thickness** [list] The thickness of each layer in order of increasing depth
- **velocity** [list] The velocity of each layer in order of increasing depth
- **zp** [list] The depths of the measurement stations (seismometers)

Returns:

- **travel_times** [array] The first-arrival travel-times calculated at the measurement stations.

5.2.4 Straigh-ray 2D tomography with SrTomo (`fatiando.seismic.srtomo`)

SrTomo: Straight-ray 2D travel-time tomography (i.e., does not consider reflection or refraction)

Functions

- `run`: Run the tomography on a given data set
- `slowness2vel`: Safely convert slowness to velocity (avoids zero division)

Examples

Using simple synthetic data:

```
>>> import fatiando as ft
>>> # One source was recorded at 3 receivers.
>>> # The medium has 2 velocities: 2 and 5
>>> model = [ft.mesher.Square([0, 10, 0, 5], {'vp':2}),
...          ft.mesher.Square([0, 10, 5, 10], {'vp':5})]
>>> src = (5, 0)
>>> srcs = [src, src, src]
>>> recs = [(0, 0), (5, 10), (10, 0)]
>>> # Calculate the synthetic travel-times
>>> ttimes = ft.seismic.ttime2d.straight(model, 'vp', srcs, recs)
>>> print ttimes
[ 2.5  3.5  2.5]
>>> # Run the tomography to calculate the 2 velocities
>>> mesh = ft.mesher.SquareMesh((0, 10, 0, 10), shape=(2, 1))
>>> # Run the tomography
>>> estimate, residuals = ft.seismic.srtomo.run(ttimes, srcs, recs, mesh)
>>> # Actually returns slowness instead of velocity
>>> for velocity in slowness2vel(estimate):
...     print '%.4f' % (velocity),
2.0000 5.0000
>>> for v in residuals:
...     print '%.4f' % (v),
0.0000 0.0000 0.0000
```

Again, using simple synthetic data but this time use Newton's method to solve:

```
>>> import fatiando as ft
>>> # One source was recorded at 3 receivers.
>>> # The medium has 2 velocities: 2 and 5
>>> model = [ft.mesher.Square([0, 10, 0, 5], {'vp':2}),
...          ft.mesher.Square([0, 10, 5, 10], {'vp':5})]
>>> src = (5, 0)
>>> srcs = [src, src, src]
>>> recs = [(0, 0), (5, 10), (10, 0)]
>>> # Calculate the synthetic travel-times
>>> ttimes = ft.seismic.ttime2d.straight(model, 'vp', srcs, recs)
>>> print ttimes
[ 2.5  3.5  2.5]
>>> # Run the tomography to calculate the 2 velocities
>>> mesh = ft.mesher.SquareMesh((0, 10, 0, 10), shape=(2, 1))
>>> # Will use Newton's method to solve this
>>> solver = ft.inversion.gradient.newton(initial=[0, 0], maxit=5)
>>> estimate, residuals = ft.seismic.srtomo.run(ttimes, srcs, recs, mesh,
...                                              solver)
>>> # Actually returns slowness instead of velocity
>>> for velocity in slowness2vel(estimate):
...     print '%.4f' % (velocity),
2.0000 5.0000
>>> for v in residuals:
...     print '%.4f' % (v),
0.0000 0.0000 0.0000
```

Note: A simple way to plot the results is to use the `addprop` method of the mesh and then pass the mesh to `fatiando.vis.map.squaremesh`.

class `fatiando.seismic.srtomo.`**`TravelTime`**(*ttimes*, *srcs*, *recs*, *mesh*, *sparse=False*)

Bases: `fatiando.inversion.datamodule.DataModule`

Data module for the 2D travel-time straight-ray tomography problem.

Bundles together the travel-time data, source and receiver positions and uses this to calculate gradients, Hessians and other inverse problem related quantities.

Parameters:

- **ttimes** [array] Array with the travel-times of the straight seismic rays.
- **srcs** [list of lists] List of the [x, y] positions of the sources.
- **recs** [list of lists] List of the [x, y] positions of the receivers.
- **mesh** [`SquareMesh` or compatible] The mesh where the inversion (tomography) will take place.
- **sparse** [True or False] Wether or not to use sparse matrices from scipy

The ith travel-time is the time between the ith element in *srcs* and the ith element in *recs*.

For example:

```
>>> from fatiando.mesher import SquareMesh
>>> # One source
>>> src = (5, 0)
>>> # was recorded at 3 receivers
>>> recs = [(0, 0), (5, 10), (10, 0)]
>>> # Resulting in Vp travel times
>>> ttimes = [2.5, 5., 2.5]
>>> # If running the tomography on a mesh
>>> mesh = SquareMesh(bounds=(0, 10, 0, 10), shape=(10, 10))
>>> # what TravelTime expects is
>>> srcs = [src, src, src]
>>> dm = TravelTime(ttimes, srcs, recs, mesh)
```

`fatiando.seismic.srtomo.`**`run`**(*ttimes*, *srcs*, *recs*, *mesh*, *solver=None*, *sparse=False*, *damping=0.0*, *smooth=0.0*, *sharp=0.0*, *beta=1e-05*)

Perform a 2D straight-ray travel-time tomography. Estimates the slowness (1/velocity) of cells in mesh (because slowness is linear and easier)

Regularization is usually **not** optional. At least some amount of damping is required.

Parameters:

- **ttimes** [array] The travel-times of the straight seismic rays.
- **srcs** [list of lists] List of the [x, y] positions of the sources.
- **recs** [list of lists] List of the [x, y] positions of the receivers.
- **mesh** [`SquareMesh` or compatible] The mesh where the inversion (tomography) will take place.
- **solver** [function] A linear or non-linear inverse problem solver generated by a factory function from a module of package `fatiando.inversion`. If None, will use the default solver.
- **sparse** [True or False] If True, will use sparse matrices from *scipy.sparse*.

  Note: If you provided a solver function, don't forget to turn on sparcity in the inversion solver module **BEFORE** creating the solver function! The usual way of doing this is by calling the `use_sparse` function. Ex: `fatiando.inversion.gradient.use_sparse()`

  Warning: Jacobian matrix building using sparse matrices isn't very optimized. It will be slow but won't overflow the memory.

- **damping** [float] Damping regularizing parameter (i.e., how much damping to apply). Must be a positive scalar.
- **smooth** [float] Smoothness regularizing parameter (i.e., how much smoothness to apply). Must be a positive scalar.
- **sharp** [float] Sharpness (total variation) regularizing parameter (i.e., how much sharpness to apply). Must be a positive scalar.
- **beta** [float] Total variation parameter. See `fatiando.inversion.regularizer.TotalVariation` for details

Returns:

- results : list = [slowness, residuals]:
  - **slowness** [array] The slowness of each cell in *mesh*
  - **residuals** [array] The inversion residuals (observed travel-times minus predicted travel-times by the slowness estimate)

`fatiando.seismic.srtomo.`**`slowness2vel`**`(`*slowness*, *tol=1e-08*`)`

Safely convert slowness to velocity.

Almost 0 slowness is mapped to 0 velocity.

Parameters:

- **slowness** [array] The slowness values
- **tol** [float] Slowness < tol will be set to 0 velocity

Returns:

- **velocity** [array] The converted velocities

5.2.5 Finite difference solution of the 2D wave equation (`fatiando.seismic.wavefd`)

Finite difference solution of the 2D wave equation for isotropic media.

Warning: Due to the high computational demmand of these simulations, the pure Python time stepping functions are **very** slow! I strongly recommend using the optimized Cython time stepping module.

Simulates both elastic and acoustic waves:

- `elastic_psv`: Simulates the coupled P and SV elastic waves
- `elastic_sh`: Simulates SH elastic waves

Sources

- `MexHatSource`: Mexican hat wavelet source
- `SinSqrSource`: Sine squared source

Auxiliary function

- `lame`: Calculate the Lame constants from P and S wave velocities and density

Theory

We start with the wave equation for elastic isotropic media

$$(\lambda + \mu)\nabla(\nabla \cdot \mathbf{u}) + \mu\nabla^2\mathbf{u} - \rho\partial_t^2\mathbf{u} = -\mathbf{f}$$

where $\mathbf{u} = (u_x, u_y, y_z)$ is the particle movement vector, ρ is the density, λ and μ are the Lame constants, and $\mathbf{f}$ is the source vector.

In the 2D approximation, we assume all derivatives in the y direction are zero and consider only x and z coordinates (though u_y remains). The three equations in the vector equation above can be separated into two groups:

$$\mu\left(\partial_x^2 u_y + \partial_z^2 u_y\right) - \rho\partial_t^2 u_y = -f_y$$

and

$$(\lambda + 2\mu)\partial_x^2 u_x + \mu\partial_z^2 u_x + (\lambda + \mu)\partial_x\partial_z u_z - \rho\partial_t^2 u_x = -f_x$$

$$(\lambda + 2\mu)\partial_z^2 u_z + \mu\partial_x^2 u_z + (\lambda + \mu)\partial_x\partial_z u_x - \rho\partial_t^2 u_z = -f_z$$

The first equation depends only on u_y and represents SH waves. The other two depend on u_x and u_z and are coupled. They represent P and SV waves.

I'll use an explicit finite difference solution for these equations. I'll use a second order approximation for the time derivative and a fourth order approximation for the spacial derivatives.

The finite difference solution for SH waves is:

$$\begin{aligned}
u_y[i,j]_{t+1} =& 2u_y[i,j]_t - u_y[i,j]_{t-1} + \frac{\Delta t^2}{\rho[i,j]}\Bigg[f_y[i,j]_t \\
&+ \mu[i,j]\left(\frac{-u_y[i,j+2]_t + 16u_y[i,j+1]_t - 30u_y[i,j]_t + 16u_y[i,j-1]_t - u_y[i,j-2]_t}{12\Delta x^2}\right) \\
&+ \mu[i,j]\left(\frac{-u_y[i+2,j]_t + 16u_y[i+1,j]_t - 30u_y[i,j]_t + 16u_y[i-1,j]_t - u_y[i-2,j]_t}{12\Delta z^2}\right)\Bigg]
\end{aligned}$$

where $[i,j]_t$ is the quantity at the grid node i,j at a time t. In this formulation, i denotes z coordinates and j x coordinates.

The solution for P and SV waves is:

$$\begin{aligned}
u_x[i,j]_{t+1} =& 2u_x[i,j]_t - u_x[i,j]_{t-1} \\
&+ \frac{\Delta t^2}{\rho[i,j]}\Bigg\{f_x[i,j]_t + (\lambda[i,j] + 2\mu[i,j])\left(\frac{u_x[i,j+1]_t - 2u_x[i,j]_t + u_x[i,j-1]_t}{\Delta x^2}\right) \\
&+ \mu[i,j]\left(\frac{u_x[i+1,j]_t - 2u_x[i,j]_t + u_x[i-1,j]_t}{\Delta z^2}\right) \\
&+ (\lambda[i,j] + \mu[i,j])\left(\frac{u_z[i,j]_t - u_z[i-1,j]_t - u_z[i,j-1]_t + u_z[i-1,j-1]_t}{\Delta x \Delta z}\right)\Bigg\}
\end{aligned}$$

$$\begin{aligned}u_z[i,j]_{t+1} =& 2u_z[i,j]_t - u_z[i,j]_{t-1} \\ &+ \frac{\Delta t^2}{\rho[i,j]}\Bigg\{ f_z[i,j]_t + (\lambda[i,j] + 2\mu[i,j])\left(\frac{u_z[i+1,j]_t - 2u_z[i,j]_t + u_z[i-1,j]_t}{\Delta z^2}\right) \\ &+ \mu[i,j]\left(\frac{u_z[i,j+1]_t - 2u_z[i,j]_t + u_z[i,j-1]_t}{\Delta x^2}\right) \\ &+ (\lambda[i,j] + \mu[i,j])\left(\frac{u_x[i,j]_t - u_x[i-1,j]_t - u_x[i,j-1]_t + u_x[i-1,j-1]_t}{\Delta x \Delta z}\right)\Bigg\}\end{aligned}$$

---

class `fatiando.seismic.wavefd.`**`MexHatSource`**(*i*, *j*, *amp*, *wlength*, *delay=0*)

Bases: `object`

A wave source that vibrates as a mexicam hat (Ricker) wavelet.

$$\psi(t) = A\frac{2}{\sqrt{3\sigma\pi^{\frac{1}{4}}}}\left(1 - \frac{t^2}{\sigma^2}\right)\exp\left(\frac{-t^2}{2\sigma^2}\right)$$

Parameters:

- **i, j** [int] The i,j coordinates of the source in the target finite difference grid. i is the index for z, j for x
- **amp** [float] The amplitude of the source (A)
- **wlength** [float] The "wave length" (σ)
- **delay** [float] The delay before the source starts

Note: If you want the source to start with amplitude close to 0, use `delay = 3.5*wlength`.

`coords`()

Get the i,j coordinates of the source in the finite difference grid.

Returns:

- **(i,j)** [tuple] The i,j coordinates

class `fatiando.seismic.wavefd.`**`SinSqrSource`**(*i*, *j*, *amp*, *wlength*, *delay=0*)

Bases: `fatiando.seismic.wavefd.MexHatSource`

A wave source that vibrates as a sine squared function.

$$\psi(t) = A\sin\left(t\frac{2\pi}{T}\right)^2$$

Parameters:

- **i, j** [int] The i,j coordinates of the source in the target finite difference grid. i is the index for z, j for x
- **amp** [float] The amplitude of the source (A)
- **wlength** [float] The wave length (T)
- **delay** [float] The delay before the source starts

Note: If you want the source to start with amplitude close to 0, use `delay = 3.5*wlength`.

`fatiando.seismic.wavefd.`**`elastic_psv`**(*spacing*, *shape*, *pvel*, *svel*, *dens*, *deltat*, *iterations*, *xsources*, *zsources*, *padding=1.0*, *partition=(1, 1)*)

Simulate P and SV waves using an explicit finite differences scheme.

Parameters:

- **spacing** [(dz, dx)] The node spacing of the finite differences grid
- **shape** [(nz, nx)] The number of nodes in the grid in the z and x directions
- **svel** [2D-array (shape = *shape*)] The S wave velocity at all the grid nodes
- **dens** [2D-array (shape = *shape*)] The value of the density at all the grid nodes
- **deltat** [float] The time interval between iterations
- **iterations** [int] Number of time steps to take
- **xsources** [list] A list of the sources of waves for the particle movement in the x direction (see `MexHatSource` for an example source)
- **zsources** [list] A list of the sources of waves for the particle movement in the z direction
- **padding** [float] The decimal percentage of padding to use in the grid to avoid reflections at the borders
- **partition** [tuple = (sz, sx)] How to split the grid for parallel computation. Number of parts in z and x, respectively

Yields:

- **ux, uz** [2D-arrays] The particle movement in the x and z direction at each time step

`fatiando.seismic.wavefd.`**`elastic_sh`**(*spacing*, *shape*, *svel*, *dens*, *deltat*, *iterations*, *sources*, *padding=1.0*, *partition=(1, 1)*)

Simulate SH waves using an explicit finite differences scheme.

Parameters:

- **spacing** [(dz, dx)] The node spacing of the finite differences grid
- **shape** [(nz, nx)] The number of nodes in the grid in the z and x directions
- **svel** [2D-array (shape = *shape*)] The S wave velocity at all the grid nodes
- **dens** [2D-array (shape = *shape*)] The value of the density at all the grid nodes
- **deltat** [float] The time interval between iterations
- **iterations** [int] Number of time steps to take
- **sources** [list] A list of the sources of waves (see `MexHatSource` for an example source)
- **padding** [float] The decimal percentage of padding to use in the grid to avoid reflections at the borders
- **partition** [tuple = (sz, sx)] How to split the grid for parallel computation. Number of parts in z and x, respectively

Yields:

- **uy** [2D-array] The particle movement in the y direction at each time step

`fatiando.seismic.wavefd.`**`lame`**(*pvel*, *svel*, *dens*)

Calculate the Lame constants λ and μ from the P and S wave velocities (α and β) and the density (ρ).

$$\mu = \beta^2 \rho$$

$$\lambda = \alpha^2 \rho - 2\mu$$

Parameters:

- **pvel** [float or array] The P wave velocity
- **svel** [float or array] The S wave velocity
- **dens** [float or array] The density

Returns:

- **[lambda, mu]** [floats or arrays] The Lame constants

Examples:

```
>>> print lame(2000, 1000, 2700)
(5400000000, 2700000000)
>>> import numpy as np
>>> pv = np.array([2000, 3000])
>>> sv = np.array([1000, 1700])
>>> dens = np.array([2700, 3100])
>>> lamb, mu = lame(pv, sv, dens)
>>> print lamb
[5400000000 9982000000]
>>> print mu
[2700000000 8959000000]
```

5.3 Geothermal heat (`fatiando.geothermal`)

Modeling and inversion for geothermal heat transfer.

Climate signal

Modeling and inversion of temperature residuals measured in wells due to temperature perturbations in the surface.

- `climsig`

5.3.1 Climate change signal in well temperature logs (`fatiando.geothermal.climsig`)

Modeling and inversion of temperature residuals measured in wells due to temperature perturbations in the surface.

Perturbations can be of two kinds:

Abrupt

- `abrupt`
- `iabrupt`

Assumes that the temperature perturbation was abrupt. The residual temperature at a depth z_i in the well at a time t after the perturbation is given by

$$T_i(z_i) = A\left[1 - \mathrm{erf}\left(\frac{z_i}{\sqrt{4\lambda t}}\right)\right]$$

where A is the amplitude of the perturbation, λ is the thermal diffusivity of the medium, and erf is the error function.

Example of inverting for the amplitude and time since the perturbation using synthetic data:

```
>>> import numpy
>>> import fatiando as ft
>>> # Generate the sythetic data along a well
>>> zp = numpy.arange(0, 100, 1)
>>> amp = 2
>>> age = 100 # Uses years to avoid overflows
>>> temp = ft.geothermal.climsig.abrupt(amp, age, zp)
>>> # Run the inversion for the amplitude and time
>>> p, residuals = ft.geothermal.climsig.iabrupt(temp, zp)
>>> print "amp: %.2f  age: %.2f" % (p[0], p[1])
amp: 2.00  age: 100.00
```

Linear

- linear
- ilinear

Assumes that the temperature perturbation was linear with time. The residual temperature at a depth z_i in the well at a time t after the perturbation was started is given by

$$T_i(z_i) = A\left[\left(1 + 2\frac{z_i^2}{4\lambda t}\right)\text{erfc}\left(\frac{z_i}{\sqrt{4\lambda t}}\right) - \frac{2}{\sqrt{\pi}}\left(\frac{z_i}{\sqrt{4\lambda t}}\right)\exp\left(-\frac{z_i^2}{4\lambda t}\right)\right]$$

where A is the amplitude of the perturbation, λ is the thermal diffusivity of the medium, and erf is the error function.

Example of inverting for the amplitude and time since the perturbation using synthetic data:

```
>>> import numpy
>>> import fatiando as ft
>>> # Generate the sythetic data along a well
>>> zp = numpy.arange(0, 100, 1)
>>> amp = 3.45
>>> age = 52.5 # Uses years to avoid overflows
>>> temp = ft.geothermal.climsig.linear(amp, age, zp)
>>> # Run the inversion for the amplitude and time
>>> p, residuals = ft.geothermal.climsig.ilinear(temp, zp)
>>> print "amp: %.2f  age: %.2f" % (p[0], p[1])
amp: 3.45  age: 52.50
```

---

class `fatiando.geothermal.climsig.`**`AbruptDM`**(*temp*, *zp*, *diffus=31.5576*)

Bases: `fatiando.inversion.datamodule.DataModule`

Data module for a single abrupt temperature perturbation.

Packs the necessary data for the inversion.

Derivatives with respect to the amplitude and age are calculated using the formula

$$\frac{\partial T_i}{\partial A} = 1 - \text{erf}\left(\frac{z_i}{\sqrt{4\lambda t}}\right)$$

and

$$\frac{\partial T_i}{\partial t} = \frac{A}{t\sqrt{\pi}}\left(\frac{z_i}{\sqrt{4\lambda t}}\right)\exp\left[-\left(\frac{z_i}{\sqrt{4\lambda t}}\right)^2\right]$$

The Hessian matrix is calculated using a Gauss-Newton approximation.

Parameters:

- **temp** [array] The temperature profile
- **zp** [array] Depths along the profile
- **diffus** [float] Thermal diffusivity of the medium (in m^2/year)

class `fatiando.geothermal.climsig.`**`LinearDM`**(*temp*, *zp*, *diffus=31.5576*)

Bases: `fatiando.inversion.datamodule.DataModule`

Data module for a single linear temperature perturbation.

Packs the necessary data for the inversion.

Derivatives with respect to the age are calculated using a 2-point finite difference approximation. Derivatives with respect to amplitude are calculate using the formula

$$\frac{\partial T_i}{\partial A} = \left(1 + 2\frac{z_i^2}{4\lambda t}\right)\text{erfc}\left(\frac{z_i}{\sqrt{4\lambda t}}\right) - \frac{2}{\sqrt{\pi}}\left(\frac{z_i}{\sqrt{4\lambda t}}\right)\exp\left(-\frac{z_i^2}{4\lambda t}\right)$$

The Hessian matrix is calculated using a Gauss-Newton approximation.

Parameters:

- **temp** [array] The temperature profile
- **zp** [array] The depths along the profile
- **diffus** [float] Thermal diffusivity of the medium (in m^2/year)

`fatiando.geothermal.climsig.`**`abrupt`**(*amp*, *age*, *zp*, *diffus=31.5576*)

Calculate the residual temperature profile in depth due to an abrupt temperature perturbation.

Parameters:

- **amp** [float] Amplitude of the perturbation (in C)
- **age** [float] Time since the perturbation occured (in years)
- **zp** [array] Arry with the depths of computation points along the well (in meters)
- **diffus** [float] Thermal diffusivity of the medium (in m^2/year)

See the default values for the thermal diffusivity in `fatiando.constants`.

Returns

- **temp** [array] The residual temperatures measured along the well

`fatiando.geothermal.climsig.`**`iabrupt`**(*temp*, *zp*, *solver=None*, *diffus=31.5576*, *iterate=False*)

Invert the residual temperature profile to estimate the amplitude and age of an abrupt temperature perturbation.

Parameters:

- **temp** [array] The temperature profile
- **zp** [array] The depths along the profile
- **solver** [function or None] A non-linear inverse problem solver generated by a factory function from a `fatiando.inversion` inverse problem solver module. If None, will use the default solver.
- **diffus** [float] Thermal diffusivity of the medium (in m^2/year)
- **iterate** [True or False] If True, will yield the current estimate at each iteration yielded by *solver*. In Python terms, `iterate=True` transforms this function into a generator function.

See the default values for the thermal diffusivity in `fatiando.constants`.

Returns:

- **results** [list = [p, residuals]] The estimated paramter vector `p = [amp, age]` and the residuals (fit) produced by the inversion. The residuals are the observed data minus the data predicted by the estimated parameters.

`fatiando.geothermal.climsig.`**`ilinear`**(*temp*, *zp*, *solver=None*, *diffus=31.5576*, *iterate=False*)

Invert the residual temperature profile to estimate the amplitude and age of a linear temperature perturbation.

Parameters:

- **temp** [array] Array with the temperature profile
- **zp** [array] Array with the depths along the profile
- **solver** [function] A non-linear inverse problem solver generated by a factory function from a `inversion` inverse problem solver module.If None, will use the default solver.
- **diffus** [float] Thermal diffusivity of the medium (in m^2/year)
- **iterate** [True or False] If True, will yield the current estimate at each iteration yielded by *solver*. In Python terms, `iterate=True` transforms this function into a generator function.

See the default values for the thermal diffusivity in `fatiando.constants`.

Returns:

- **results** [list = [p, residuals]] The estimated paramter vector `p = [amp, age]` and the residuals (fit) produced by the inversion. The residuals are the observed data minus the data predicted by the estimated parameters.

`fatiando.geothermal.climsig.`**`linear`**(*amp*, *age*, *zp*, *diffus=31.5576*)

Calculate the residual temperature profile in depth due to a linear temperature perturbation.

Parameters:

- **amp** [float] Amplitude of the perturbation (in C)
- **age** [float] Time since the perturbation occured (in years)
- **zp** [array] The depths of computation points along the well (in meters)
- **diffus** [float] Thermal diffusivity of the medium (in m^2/year)

See the default values for the thermal diffusivity in `fatiando.constants`.

Returns

- **temp** [array] The residual temperatures measured along the well

5.4 Meshing (`fatiando.mesher`)

Generate and operate on various kinds of meshes and geometric elements

Geometric elements

- `Polygon`
- `Square`
- `Prism`
- `PolygonalPrism`
- `Sphere`
- `Tesseroid`

Meshes

- `SquareMesh`
- `PrismMesh`
- `PrismRelief`
- `TesseroidMesh`

Utility functions

- `extract`: Extract the values of a physicalr property from the cells in a list
- `vfilter`: Remove cells whose physical property value falls outside a given range
- `vremove`: Remove the cells with a given physical property value

---

class `fatiando.mesher.`**`GeometricElement`**(*props*)

Bases: `object`

Base class for all geometric elements.

`addprop`(*prop*, *value*)

Add a physical property to this geometric element.

If it already has the property, the given value will overwrite the existing one.

Parameters:

- **prop** [str] Name of the physical property.
- **value** [float] The value of this physical property.

class `fatiando.mesher.`**`Polygon`**`(`*vertices*, *props=None*`)`

Bases: `fatiando.mesher.GeometricElement`

Create a polygon object.

Note: Most applications require the vertices to be **clockwise**!

Parameters:

- **vertices** [list of lists] List of [x, y] pairs with the coordinates of the vertices.
- **props** [dict] Physical properties assigned to the polygon. Ex: `props={'density':10, 'susceptibility':10000}`

class `fatiando.mesher.`**`PolygonalPrism`**`(`*vertices*, *z1*, *z2*, *props=None*`)`

Bases: `fatiando.mesher.GeometricElement`

Create a 3D prism with polygonal crossection.

Note: The coordinate system used is x -> North, y -> East and z -> Down

Note: *vertices* must be **CLOCKWISE** or will give inverse result.

Parameters:

- **vertices** [list of lists] Coordinates of the vertices. A list of `[x, y]` pairs.
- **z1, z2** [float] Top and bottom of the prism
- **props** [dict] Physical properties assigned to the prism. Ex: `props={'density':10, 'magnetization':10000}`

Examples:

```
>>> verts = [[1, 1], [1, 2], [2, 2], [2, 1]]
>>> p = PolygonalPrism(verts, 0, 3, props={'temperature':25})
>>> p.props['temperature']
25
>>> print p.x
[ 1.  1.  2.  2.]
>>> print p.y
[ 1.  2.  2.  1.]
>>> print p.z1, p.z2
0.0 3.0
>>> p.addprop('density', 2670)
>>> print p.props['density']
2670
```

`topolygon``()`

Get the polygon describing the prism viewed from above.

Returns:

- **polygon** [`fatiando.mesher.Polygon`] The polygon

Example:

```
>>> verts = [[1, 1], [1, 2], [2, 2], [2, 1]]
>>> p = PolygonalPrism(verts, 0, 100)
>>> poly = p.topolygon()
>>> print poly.x
```

```
[ 1.  1.  2.  2.]
>>> print poly.y
[ 1.  2.  2.  1.]
```

class `fatiando.mesher.`**`Prism`**(*x1*, *x2*, *y1*, *y2*, *z1*, *z2*, *props=None*)
Bases: `fatiando.mesher.GeometricElement`

Create a 3D right rectangular prism.

Note: The coordinate system used is x -> North, y -> East and z -> Down

Parameters:

- **x1, x2** [float] South and north borders of the prism
- **y1, y2** [float] West and east borders of the prism
- **z1, z2** [float] Top and bottom of the prism
- **props** [dict] Physical properties assigned to the prism. Ex: `props={'density':10, 'magnetization':10000}`

Examples:

```
>>> from fatiando.mesher import Prism
>>> p = Prism(1, 2, 3, 4, 5, 6, {'density':200})
>>> p.props['density']
200
>>> print p.get_bounds()
[1.0, 2.0, 3.0, 4.0, 5.0, 6.0]
>>> print p
x1:1 | x2:2 | y1:3 | y2:4 | z1:5 | z2:6 | density:200
>>> p = Prism(1, 2, 3, 4, 5, 6)
>>> print p
x1:1 | x2:2 | y1:3 | y2:4 | z1:5 | z2:6
>>> p.addprop('density', 2670)
>>> print p
x1:1 | x2:2 | y1:3 | y2:4 | z1:5 | z2:6 | density:2670
```

`center`()
Return the coordinates of the center of the prism.

Returns:

- **coords** [list = [xc, yc, zc]] Coordinates of the center

Example:

```
>>> prism = Prism(1, 2, 1, 3, 0, 2)
>>> print prism.center()
[1.5, 2.0, 1.0]
```

`get_bounds`()
Get the bounding box of the prism (i.e., the borders of the prism).

Returns:

- **bounds** [list] `[x1, x2, y1, y2, z1, z2]`, the bounds of the prism

Examples:

```
>>> p = Prism(1, 2, 3, 4, 5, 6)
>>> print p.get_bounds()
[1.0, 2.0, 3.0, 4.0, 5.0, 6.0]
```

class `fatiando.mesher.`**`PrismMesh`**(*bounds*, *shape*, *props=None*)
Bases: `object`

Generate a 3D regular mesh of right rectangular prisms.

Prisms are ordered as follows: first layers (z coordinate), then EW rows (y) and finaly x coordinate (NS).

Note: Remember that the coordinate system is x->North, y->East and z->Down

Ex: in a mesh with shape `(3,3,3)` the 15th element (index 14) has z index 1 (second layer), y index 1 (second row), and x index 2 (third element in the column).

`PrismMesh` can used as list of prisms. It acts as an iteratior (so you can loop over prisms). It also has a `__getitem__` method to access individual elements in the mesh. In practice, `PrismMesh` should be able to be passed to any function that asks for a list of prisms, like `fatiando.gravmag.prism.gz`.

To make the mesh incorporate a topography, use `carvetopo`

Parameters:

- **bounds** [list = [xmin, xmax, ymin, ymax, zmin, zmax]] Boundaries of the mesh.
- **shape** [tuple = (nz, ny, nx)] Number of prisms in the x, y, and z directions.
- **props** [dict] Physical properties of each prism in the mesh. Each key should be the name of a physical property. The corresponding value should be a list with the values of that particular property on each prism of the mesh.

Examples:

```
>>> from fatiando.mesher import PrismMesh
>>> mesh = PrismMesh((0,1,0,2,0,3),(1,2,2))
>>> for p in mesh:
...     print p
x1:0 | x2:0.5 | y1:0 | y2:1 | z1:0 | z2:3
x1:0.5 | x2:1 | y1:0 | y2:1 | z1:0 | z2:3
x1:0 | x2:0.5 | y1:1 | y2:2 | z1:0 | z2:3
x1:0.5 | x2:1 | y1:1 | y2:2 | z1:0 | z2:3
>>> print mesh[0]
x1:0 | x2:0.5 | y1:0 | y2:1 | z1:0 | z2:3
>>> print mesh[-1]
x1:0.5 | x2:1 | y1:1 | y2:2 | z1:0 | z2:3
```

One with physical properties:

```
>>> props = {'density':[2670.0, 1000.0]}
>>> mesh = PrismMesh((0, 2, 0, 4, 0, 3), (1, 1, 2), props=props)
>>> for p in mesh:
...     print p
x1:0 | x2:1 | y1:0 | y2:4 | z1:0 | z2:3 | density:2670
x1:1 | x2:2 | y1:0 | y2:4 | z1:0 | z2:3 | density:1000
```

You can use `get_xs` (and similar methods for y and z) to get the x coordinates os the prisms in the mesh:

```
>>> mesh = PrismMesh((0, 2, 0, 4, 0, 3), (1, 1, 2))
>>> print mesh.get_xs()
[ 0.  1.  2.]
>>> print mesh.get_ys()
[ 0.  4.]
>>> print mesh.get_zs()
[ 0.  3.]
```

addprop (*prop*, *values*)

Add physical property values to the cells in the mesh.

Different physical properties of the mesh are stored in a dictionary.

Parameters:

- **prop** [str] Name of the physical property.

- **values** [list or array] Value of this physical property in each prism of the mesh. For the ordering of prisms in the mesh see `PrismMesh`

`carvetopo`(*x*, *y*, *height*)

Mask (remove) prisms from the mesh that are above the topography.

Accessing the ith prism will return None if it was masked (above the topography). Also mask prisms outside of the topography grid provided. The topography height information does not need to be on a regular grid, it will be interpolated.

Parameters:

- **x, y** [lists] x and y coordinates of the grid points
- **height** [list or array] Array with the height of the topography

`celltype`

alias of `Prism`

`dump`(*meshfile*, *propfile*, *prop*)

Dump the mesh to a file in the format required by UBC-GIF program MeshTools3D.

Parameters:

- **meshfile** [str or file] Output file to save the mesh. Can be a file name or an open file.
- **propfile** [str or file] Output file to save the physical properties *prop*. Can be a file name or an open file.
- **prop** [str] The name of the physical property in the mesh that will be saved to *propfile*.

Note: Uses -10000000 as the dummy value for plotting topography

Examples:

```
>>> from StringIO import StringIO
>>> meshfile = StringIO()
>>> densfile = StringIO()
>>> mesh = PrismMesh((0, 10, 0, 20, 0, 5), (1, 2, 2))
>>> mesh.addprop('density', [1, 2, 3, 4])
>>> mesh.dump(meshfile, densfile, 'density')
>>> print meshfile.getvalue().strip()
2 2 1
0 0 0
2*10
2*5
1*5
>>> print densfile.getvalue().strip()
1.0000
3.0000
2.0000
4.0000
```

`get_layer`(*i*)

Return the set of prisms corresponding to the ith layer of the mesh.

Parameters:

- **i** [int] The index of the layer

Returns:

- **prisms** [list of `Prism`] The prisms in the ith layer

Examples:

```
>>> mesh = PrismMesh((0, 2, 0, 2, 0, 2), (2, 2, 2))
>>> layer = mesh.get_layer(0)
>>> for p in layer:
...     print p
x1:0 | x2:1 | y1:0 | y2:1 | z1:0 | z2:1
x1:1 | x2:2 | y1:0 | y2:1 | z1:0 | z2:1
x1:0 | x2:1 | y1:1 | y2:2 | z1:0 | z2:1
x1:1 | x2:2 | y1:1 | y2:2 | z1:0 | z2:1
>>> layer = mesh.get_layer(1)
>>> for p in layer:
...     print p
x1:0 | x2:1 | y1:0 | y2:1 | z1:1 | z2:2
x1:1 | x2:2 | y1:0 | y2:1 | z1:1 | z2:2
x1:0 | x2:1 | y1:1 | y2:2 | z1:1 | z2:2
x1:1 | x2:2 | y1:1 | y2:2 | z1:1 | z2:2
```

get_xs()
Return an array with the x coordinates of the prisms in mesh.

get_ys()
Return an array with the y coordinates of the prisms in mesh.

get_zs()
Return an array with the z coordinates of the prisms in mesh.

class fatiando.mesher.**PrismRelief**(*ref*, *dims*, *nodes*)
Bases: object

Generate a 3D model of a relief (topography) using prisms.

Use to generate: * topographic model * basin model * Moho model * etc

PrismRelief can used as list of prisms. It acts as an iteratior (so you can loop over prisms). It also has a __getitem__ method to access individual elements in the mesh. In practice, PrismRelief should be able to be passed to any function that asks for a list of prisms, like fatiando.gravmag.prism.gz.

Parameters:

- **ref** [float]

  Reference level. Prisms will have:

  - bottom on zref and top on z if z > zref;
  - bottom on z and top on zref otherwise.
- **dims** [tuple = (dy, dx)] Dimensions of the prisms in the y and x directions
- **nodes** [list of lists = [x, y, z]] Coordinates of the center of the top face of each prism.x, y, and z are lists with the x, y and z coordinates on a regular grid.

addprop(*prop*, *values*)
Add physical property values to the prisms.

Warning: If the z value of any point in the relief is bellow the reference level, its corresponding prism will have the physical property value with oposite sign than was assigned to it.

Parameters:

- **prop** [str] Name of the physical property.
- **values** [list] List or array with the value of this physical property in each prism of the relief.

class fatiando.mesher.**Sphere**(*x*, *y*, *z*, *radius*, *props=None*)
Bases: fatiando.mesher.GeometricElement

Create a sphere.

Note: The coordinate system used is x -> North, y -> East and z -> Down

Parameters:

- **x, y, z** [float] The coordinates of the center of the sphere
- **props** [dict] Physical properties assigned to the prism. Ex: `props={'density':10, 'magnetization':10000}`

Examples:

```
>>> s = Sphere(1, 2, 3, 10, {'magnetization':200})
>>> s.props['magnetization']
200
>>> s.addprop('density', 20)
>>> print s.props['density']
20
>>> print s
x:1 | y:2 | z:3 | radius:10 | density:20 | magnetization:200
>>> s = Sphere(1, 2, 3, 4)
>>> print s
x:1 | y:2 | z:3 | radius:4
>>> s.addprop('density', 2670)
>>> print s
x:1 | y:2 | z:3 | radius:4 | density:2670
```

class `fatiando.mesher.`**`Square`**(*bounds*, *props=None*)

Bases: `fatiando.mesher.GeometricElement`

Create a square object.

Parameters:

- **bounds** [list = [x1, x2, y1, y2]] Coordinates of the top right and bottom left corners of the square
- **props** [dict] Physical properties assigned to the square. Ex: `props={'density':10, 'slowness':10000}`

Example:

```
>>> sq = Square([0, 1, 2, 4], {'density':750})
>>> print sq
x1:0 | x2:1 | y1:2 | y2:4 | density:750
>>> sq.addprop('magnetization', 100)
>>> print sq
x1:0 | x2:1 | y1:2 | y2:4 | density:750 | magnetization:100
```

`topolygon`()

Convert this square into a polygon representation.

Vertices are ordered clockwise considering that x is North.

Returns:

- **polygon** [`Polygon`] The polygon equivalente of the square

Example:

```
>>> square = Square((0, 1, 0, 3), {'vp':1000})
>>> poly = square.topolygon()
>>> print poly.x
[ 0.  1.  1.  0.]
>>> print poly.y
[ 0.  0.  3.  3.]
>>> print poly.props['vp']
1000
```

class `fatiando.mesher.`**`SquareMesh`**(*bounds*, *shape*, *props=None*)

Bases: `object`

Generate a 2D regular mesh of squares.

For all purposes, `SquareMesh` can be used as a list of `Square`. The order of the squares in the list is: x directions varies first, then y.

Parameters:

- **bounds** [list = [x1, x2, y1, y2]] Boundaries of the mesh
- **shape** [tuple = (ny, nx)] Number of squares in the y and x dimension, respectively
- **props** [dict] Physical properties of each square in the mesh. Each key should be the name of a physical property. The corresponding value should be a list with the values of that particular property on each square of the mesh.

Examples:

```
>>> mesh = SquareMesh((0, 4, 0, 6), (2, 2))
>>> for s in mesh:
...     print s
x1:0 | x2:2 | y1:0 | y2:3
x1:2 | x2:4 | y1:0 | y2:3
x1:0 | x2:2 | y1:3 | y2:6
x1:2 | x2:4 | y1:3 | y2:6
>>> print mesh[1]
x1:2 | x2:4 | y1:0 | y2:3
>>> print mesh[-1]
x1:2 | x2:4 | y1:3 | y2:6
```

With physical properties:

```
>>> mesh = SquareMesh((0, 4, 0, 6), (2, 1), {'slowness':[3.4, 8.6]})
>>> for s in mesh:
...     print s
x1:0 | x2:4 | y1:0 | y2:3 | slowness:3.4
x1:0 | x2:4 | y1:3 | y2:6 | slowness:8.6
```

Or:

```
>>> mesh = SquareMesh((0, 4, 0, 6), (2, 1))
>>> mesh.addprop('slowness', [3.4, 8.6])
>>> for s in mesh:
...     print s
x1:0 | x2:4 | y1:0 | y2:3 | slowness:3.4
x1:0 | x2:4 | y1:3 | y2:6 | slowness:8.6
```

`addprop`(*prop*, *values*)

Add physical property values to the cells in the mesh.

Different physical properties of the mesh are stored in a dictionary.

Parameters:

- **prop** [str] Name of the physical property
- **values** [list or array] The value of this physical property in each square of the mesh. For the ordering of squares in the mesh see `SquareMesh`

`get_xs`()

Get a list of the x coordinates of the corners of the cells in the mesh.

If the mesh has nx cells, get_xs() will return nx + 1 values.

`get_ys`()

Get a list of the y coordinates of the corners of the cells in the mesh.

If the mesh has ny cells, get_ys() will return ny + 1 values.

img2prop (*fname*, *vmin*, *vmax*, *prop*)
Load the physical property value from an image file.

The image is converted to gray scale and the gray intensity of each pixel is used to set the value of the physical property of the cells in the mesh. Gray intensity values are scaled to the range `[vmin, vmax]`.

If the shape of image (number of pixels in y and x) is different from the shape of the mesh, the image will be interpolated to match the shape of the mesh.

Parameters:

- **fname** [str] Name of the image file
- **vmax, vmin** [float] Range of physical property values (used to convert the gray scale to physical property values)
- **prop** [str] Name of the physical property

class `fatiando.mesher.`**Tesseroid**(*w*, *e*, *s*, *n*, *top*, *bottom*, *props=None*)
Bases: `fatiando.mesher.GeometricElement`

Create a tesseroid (spherical prism).

Parameters:

- **w, e** [float] West and east borders of the tesseroid in decimal degrees
- **s, n** [float] South and north borders of the tesseroid in decimal degrees
- **top, bottom** [float] Bottom and top of the tesseroid with respect to the mean earth radius in meters. Ex: if the top is 100 meters above the mean earth radius, `top=100`, if 100 meters bellow `top=-100`.
- **props** [dict] Physical properties assigned to the tesseroid. Ex: `props={'density':10, 'magnetization':10000}`

Examples:

```
>>> from fatiando.mesher import Tesseroid
>>> t = Tesseroid(1, 2, 3, 4, 6, 5, {'density':200})
>>> t.props['density']
200
>>> print t.get_bounds()
[1.0, 2.0, 3.0, 4.0, 6.0, 5.0]
>>> print t
w:1 | e:2 | s:3 | n:4 | top:6 | bottom:5 | density:200
>>> t = Tesseroid(1, 2, 3, 4, 6, 5)
>>> print t
w:1 | e:2 | s:3 | n:4 | top:6 | bottom:5
>>> t.addprop('density', 2670)
>>> print t
w:1 | e:2 | s:3 | n:4 | top:6 | bottom:5 | density:2670
```

get_bounds()
Get the bounding box of the tesseroid (i.e., the borders).

Returns:

- **bounds** [list] `[w, e, s, n, top, bottom]`, the bounds of the tesseroid

Examples:

```
>>> t = Tesseroid(1, 2, 3, 4, 6, 5)
>>> print t.get_bounds()
[1.0, 2.0, 3.0, 4.0, 6.0, 5.0]
```

class `fatiando.mesher.`**`TesseroidMesh`**(*bounds*, *shape*, *props=None*)

Bases: `fatiando.mesher.PrismMesh`

Generate a 3D regular mesh of tesseroids.

Tesseroids are ordered as follows: first layers (height coordinate), then N-S rows and finaly E-W.

Ex: in a mesh with shape `(3,3,3)` the 15th element (index 14) has height index 1 (second layer), y index 1 (second row), and x index 2 (third element in the column).

This class can used as list of tesseroids. It acts as an iteratior (so you can loop over tesseroids). It also has a `__getitem__` method to access individual elements in the mesh. In practice, it should be able to be passed to any function that asks for a list of tesseroids, like `fatiando.gravmag.tesseroid.gz`.

To make the mesh incorporate a topography, use `carvetopo`

Parameters:

- **bounds** [list = [w, e, s, n, top, bottom]] Boundaries of the mesh. `w, e, s, n` in degrees, `top` and `bottom` are heights (positive upward) and in meters.
- **shape** [tuple = (nr, nlat, nlon)] Number of tesseroids in the radial, latitude, and longitude directions.
- **props** [dict] Physical properties of each tesseroid in the mesh. Each key should be the name of a physical property. The corresponding value should be a list with the values of that particular property on each tesseroid of the mesh.

`celltype`

alias of `Tesseroid`

`fatiando.mesher.`**`extract`**(*prop*, *prisms*)

Extract the values of a physical property from the cells in a list.

If a cell is *None* or doesn't have the physical property, a value of *None* will be put in it's place.

Parameters:

- **prop** [str] The name of the physical property to extract
- **cells** [list] A list of cells (e.g., `Prism`, `PolygonalPrism`, etc)

Returns:

- **values** [array] The extracted values

Examples:

```
>>> cells = [Prism(1, 2, 3, 4, 5, 6, {'foo':1}),
...          Prism(1, 2, 3, 4, 5, 6, {'foo':10}),
...          None,
...          Prism(1, 2, 3, 4, 5, 6, {'bar':2000})]
>>> print extract('foo', cells)
[1, 10, None, None]
```

`fatiando.mesher.`**`vfilter`**(*vmin*, *vmax*, *prop*, *cells*)

Remove cells whose physical property value falls outside a given range.

If a cell is *None* or doesn't have the physical property, it will be not be included in the result.

Parameters:

- **vmin** [float] Minimum value
- **vmax** [float] Maximum value
- **prop** [str] The name of the physical property used to filter
- **cells** [list] A list of cells (e.g., `Prism`, `PolygonalPrism`, etc)

Returns:

- **filtered** [list] The cells that fall within the desired range

Examples:

```
>>> cells = [Prism(1, 2, 3, 4, 5, 6, {'foo':1}),
...          Prism(1, 2, 3, 4, 5, 6, {'foo':20}),
...          Prism(1, 2, 3, 4, 5, 6, {'foo':3}),
...          None,
...          Prism(1, 2, 3, 4, 5, 6, {'foo':4}),
...          Prism(1, 2, 3, 4, 5, 6, {'foo':200}),
...          Prism(1, 2, 3, 4, 5, 6, {'bar':1000})]
>>> for cell in vfilter(0, 10, 'foo', cells):
...     print cell
x1:1 | x2:2 | y1:3 | y2:4 | z1:5 | z2:6 | foo:1
x1:1 | x2:2 | y1:3 | y2:4 | z1:5 | z2:6 | foo:3
x1:1 | x2:2 | y1:3 | y2:4 | z1:5 | z2:6 | foo:4
```

`fatiando.mesher.`**`vremove`**(*value*, *prop*, *cells*)

Remove the cells with a given physical property value.

If a cell is *None* it will be not be included in the result.

If a cell doesn't have the physical property, it will be included in the result.

Parameters:

- **value** [float] The value of the physical property to remove
- **prop** [str] The name of the physicaRemove cells whose physical property value falls outside a given rangel property
- **cells** [list] A list of cells (e.g., `Prism`, `PolygonalPrism`, etc)

Returns:

- **removed** [list] A list of cells that have *prop* != *value*

Examples:

```
>>> cells = [Prism(1, 2, 3, 4, 5, 6, {'foo':1}),
...          Prism(1, 2, 3, 4, 5, 6, {'foo':20}),
...          Prism(1, 2, 3, 4, 5, 6, {'foo':3}),
...          None,
...          Prism(1, 2, 3, 4, 5, 6, {'foo':1}),
...          Prism(1, 2, 3, 4, 5, 6, {'foo':200}),
...          Prism(1, 2, 3, 4, 5, 6, {'bar':1000})]
>>> for cell in vremove(1, 'foo', cells):
...     print cell
x1:1 | x2:2 | y1:3 | y2:4 | z1:5 | z2:6 | foo:20
x1:1 | x2:2 | y1:3 | y2:4 | z1:5 | z2:6 | foo:3
x1:1 | x2:2 | y1:3 | y2:4 | z1:5 | z2:6 | foo:200
x1:1 | x2:2 | y1:3 | y2:4 | z1:5 | z2:6 | bar:1000
```

5.5 Gridding (`fatiando.gridder`)

Create and operate on grids and profiles.

Grid generation

- `regular`
- `scatter`

Grid I/O

Grid operations

- `cut`

- `interp`

Misc

- `spacing`

`fatiando.gridder.`**`cut`** (*x*, *y*, *scalars*, *area*)

Remove a subsection of the grid.

Parameters:

- **x, y** Arrays with the x and y coordinates of the data points.
- **scalars** List of arrays with the scalar values assigned to the grid points.
- **area** `(x1, x2, y1, y2)`: Borders of the subsection

Returns:

- **`[subx, suby, subscalars]`** Arrays with x and y coordinates and scalar values of the subsection.

`fatiando.gridder.`**`interp`** (*x*, *y*, *v*, *shape*, *area=None*, *algorithm='nn'*)

Interpolate data onto a regular grid.

Warning: Doesn't extrapolate. Will return a masked array in the extrapolated areas.

Parameters:

- **x, y** [1D arrays] Arrays with the x and y coordinates of the data points.
- **v** [1D array] Array with the scalar value assigned to the data points.
- **shape** [tuple = (ny, nx)] Shape of the interpolated regular grid, ie (ny, nx).
- **area** [tuple = (x1, x2, y1, y2)] The are where the data will be interpolated. If None, then will get the area from *x* and *y*.
- **algorithm** [string] Interpolation algorithm. Either `'nn'` for natural neighbor or `'linear'` for linear interpolation. (see numpy.griddata)

Returns:

- **`[X, Y, V]`** Three 2D arrays with the interpolated x, y, and v

`fatiando.gridder.`**`regular`** (*area*, *shape*, *z=None*)

Create a regular grid. Order of the output grid is x varies first, then y.

Parameters:

- **area** `(x1, x2, y1, y2)`: Borders of the grid
- **shape** Shape of the regular grid, ie `(ny, nx)`.
- **z** Optional. z coordinate of the grid points. If given, will return an array with the value *z*.

Returns:

- **`[xcoords, ycoords]`** Numpy arrays with the x and y coordinates of the grid points
- **`[xcoords, ycoords, zcoords]`** If *z* given. Numpy arrays with the x, y, and z coordinates of the grid points

`fatiando.gridder.`**`scatter`** (*area*, *n*, *z=None*)

Create an irregular grid with a random scattering of points.

Parameters:

- **area** `(x1, x2, y1, y2)`: Borders of the grid
- **n** Number of points

- **z** Optional. z coordinate of the points. If given, will return an array with the value *z*.

Returns:

- `[xcoords, ycoords]` Numpy arrays with the x and y coordinates of the points
- `[xcoords, ycoords, zcoords]` If *z* given. Arrays with the x, y, and z coordinates of the points

`fatiando.gridder.`**`spacing`**(*area*, *shape*)

Returns the spacing between grid nodes

Parameters:

- **area** `(x1, x2, y1, y2)`: Borders of the grid
- **shape** Shape of the regular grid, ie `(ny, nx)`.

Returns:

- `[dy, dx]` Spacing the y and x directions

5.6 Visualization (`fatiando.vis`)

Plotting utilities.

Wrappers to facilitate common plotting tasks using powerful third-party libraries.

- `mpl`: 2D plotting using matplotlib (http://matplotlib.sourceforge.net/)
- `myv`: 3D plotting using Mayavi (http://code.enthought.com/projects/mayavi/)

5.6.1 Plot 2D objects, maps and grids with matplotlib (`fatiando.vis.mpl`)

Wrappers for `matplotlib` functions to facilitate plotting grids, 2D objects, etc.

This module loads all functions from `matplotlib.pyplot`, adds new functions and overwrites some others (like `contour`, `pcolor`, etc).

Grids

- `contour`
- `contourf`
- `pcolor`

Grids are automatically reshaped and interpolated if desired or necessary.

2D objects

- `points`
- `paths`
- `square`
- `squaremesh`
- `polygon`
- `layers`

Interactive

- `draw_polygon`
- `draw_layers`

- `pick_points`

Basemap (map projections)

- `basemap`
- `draw_geolines`
- `draw_countries`
- `draw_coastlines`

Auxiliary

- `set_area`
- `m2km`

`fatiando.vis.mpl.`**`basemap`**(*area*, *projection*, *resolution='c'*)

Make a basemap to use when plotting with map projections.

Uses the matplotlib basemap toolkit.

Parameters:

- **area** [list] `[west, east, south, north]`, i.e., the area of the data that is going to be plotted
- **projection** [str] The name of the projection you want to use. Choose from:
  - 'ortho': Orthographic
  - 'geos': Geostationary
  - 'robin': Robinson
  - 'cass': Cassini
  - 'merc': Mercator
  - 'poly': Polyconic
  - 'lcc': Lambert Conformal
  - 'stere': Stereographic
- **resolution** [str] The resolution for the coastlines. Can be 'c' for crude, 'l' for low, 'i' for intermediate, 'h' for high

Returns:

- **basemap** [mpl_toolkits.basemap.Basemap] The basemap

`fatiando.vis.mpl.`**`contour`**(*x*, *y*, *v*, *shape*, *levels*, *interp=False*, *color='k'*, *label=None*, *clabel=True*, *style='solid'*, *linewidth=1.0*, *basemap=None*)

Make a contour plot of the data.

Parameters:

- **x, y** [array] Arrays with the x and y coordinates of the grid points. If the data is on a regular grid, then assume x varies first (ie, inner loop), then y.
- **v** [array] The scalar value assigned to the grid points.
- **shape** [tuple = (ny, nx)] Shape of the regular grid. If interpolation is not False, then will use *shape* to grid the data.
- **levels** [int or list] Number of contours to use or a list with the contour values.
- **interp** [True or False] Wether or not to interpolate before trying to plot. If data is not on regular grid, set to True!
- **color** [str] Color of the contour lines.

- **label** [str] String with the label of the contour that would show in a legend.
- **clabel** [True or False] Wether or not to print the numerical value of the contour lines
- **style** [str] The style of the contour lines. Can be `'dashed'`, `'solid'` or `'mixed'` (solid lines for positive contours and dashed for negative)
- **linewidth** [float] Width of the contour lines
- **basemap** [mpl_toolkits.basemap.Basemap] If not None, will use this basemap for plotting with a map projection (see `basemap` for creating basemaps)

Returns:

- **levels** [list] List with the values of the contour levels

`fatiando.vis.mpl.`**`contourf`**(*x*, *y*, *v*, *shape*, *levels*, *interp=False*, *cmap=<Mock object at 0x5684110>*, *basemap=None*)

Make a filled contour plot of the data.

Parameters:

- **x, y** [array] Arrays with the x and y coordinates of the grid points. If the data is on a regular grid, then assume x varies first (ie, inner loop), then y.
- **v** [array] The scalar value assigned to the grid points.
- **shape** [tuple = (ny, nx)] Shape of the regular grid. If interpolation is not False, then will use *shape* to grid the data.
- **levels** [int or list] Number of contours to use or a list with the contour values.
- **interp** [True or False] Wether or not to interpolate before trying to plot. If data is not on regular grid, set to True!
- **cmap** [colormap] Color map to be used. (see pyplot.cm module)
- **basemap** [mpl_toolkits.basemap.Basemap] If not None, will use this basemap for plotting with a map projection (see `basemap` for creating basemaps)

Returns:

- **levels** [list] List with the values of the contour levels

`fatiando.vis.mpl.`**`draw_coastlines`**(*basemap*, *linewidth=1*, *style='solid'*)

Draw the coastlines using the given basemap.

Parameters:

- **basemap** [mpl_toolkits.basemap.Basemap] The basemap used for plotting (see `basemap`)
- **linewidth** [float] The width of the lines
- **style** [str] The style of the lines. Can be: 'solid', 'dashed', 'dashdot' or 'dotted'

`fatiando.vis.mpl.`**`draw_countries`**(*basemap*, *linewidth=1*, *style='dashed'*)

Draw the country borders using the given basemap.

Parameters:

- **basemap** [mpl_toolkits.basemap.Basemap] The basemap used for plotting (see `basemap`)
- **linewidth** [float] The width of the lines
- **style** [str] The style of the lines. Can be: 'solid', 'dashed', 'dashdot' or 'dotted'

`fatiando.vis.mpl.`**`draw_geolines`**(*area*, *dlon*, *dlat*, *basemap*, *linewidth=1*)

Draw the parallels and meridians on a basemap plot.

Parameters:

- **area** [list] `[west, east, south, north]`, i.e., the area where the lines will be plotted

- **dlon, dlat** [float] The spacing between the lines in the longitude and latitude directions, respectively (in decimal degrees)
- **basemap** [mpl_toolkits.basemap.Basemap] The basemap used for plotting (see basemap)
- **linewidth** [float] The width of the lines

`fatiando.vis.mpl.`**`draw_layers`**(*area*, *axes*, *style='-'*, *marker='o'*, *color='k'*, *width=2*)

Draw series of horizontal layers by clicking with the mouse.

The y-axis is assumed to be depth, the x-axis is the physical property of each layer.

INSTRUCTIONS:

- Click to make a new layer;
- Press 'e' to erase the last layer;
- Close the figure window to finish;

Parameters:

- **area** [list = [x1, x2, y1, y2]] Borders of the area containing the polygon
- **axes** [matplotlib Axes] The figure to use for drawing the polygon. To get an Axes instace, just do:

  ```
  from matplotlib import pyplot
  axes = pyplot.figure().add_subplot(1,1,1)
  ```

  You can plot things to `axes` before calling this function so that they'll appear on the background.
- **style** [str] Line style (as in matplotlib.pyplot.plot)
- **marker** [str] Style of the point markers (as in matplotlib.pyplot.plot)
- **color** [str] Line color (as in matplotlib.pyplot.plot)
- **width** [float] The line width (as in matplotlib.pyplot.plot)

Returns:

- layers : list = [thickness, values]
  - **thickness** [list] The thickness of each layer, in order of increasing depth
  - **values** [list] The physical property value of each layer, in the same order

`fatiando.vis.mpl.`**`draw_polygon`**(*area*, *axes*, *style='-'*, *marker='o'*, *color='k'*, *width=2*, *alpha=0.5*, *xy2ne=False*)

Draw a polygon by clicking with the mouse.

INSTRUCTIONS:

- Left click to pick the edges of the polygon;
- Draw edges CLOCKWISE;
- Press 'e' to erase the last edge;
- Right click to close the polygon;
- Close the figure window to finish;

Parameters:

- **area** [list = [x1, x2, y1, y2]] Borders of the area containing the polygon
- **axes** [matplotlib Axes] The figure to use for drawing the polygon. To get an Axes instace, just do:

  ```
  from matplotlib import pyplot
  axes = pyplot.figure().add_subplot(1,1,1)
  ```

  You can plot things to `axes` before calling this function so that they'll appear on the background.

- **style** [str] Line style (as in matplotlib.pyplot.plot)
- **marker** [str] Style of the point markers (as in matplotlib.pyplot.plot)
- **color** [str] Line color (as in matplotlib.pyplot.plot)
- **width** [float] The line width (as in matplotlib.pyplot.plot)
- **alpha** [float] Transparency of the fill of the polygon. 0 for transparent, 1 for opaque (fills the polygon once done drawing)
- **xy2ne** [True or False] If True, will exchange the x and y axis so that x points north. Use this when drawing on a map viewed from above. If the y-axis of the plot is supposed to be z (depth), then use `xy2ne=False`.

Returns:

- **edges** [list of lists] List of `[x, y]` pairs with the edges of the polygon

`fatiando.vis.mpl.`**`layers`**(*thickness*, *values*, *style='-k'*, *z0=0.0*, *linewidth=1*, *label=None*, ***kwargs*)

Plot a series of layers and values associated to each layer.

Parameters:

- **thickness** [list] The thickness of each layer in order of increasing depth
- **values** [list] The value associated with each layer in order of increasing depth
- **style** [str] String with the color and line style (as in matplotlib.pyplot.plot)
- **z0** [float] The depth of the top of the first layer
- **linewidth** [float] Line width
- **label** [str] label associated with the square.

Returns:

- **axes** [`matplitlib.axes`] The axes element of the plot

`fatiando.vis.mpl.`**`m2km`**(*axis=None*)

Convert the x and y tick labels from meters to kilometers.

Parameters:

- **axis** [matplotlib axis instance] The plot.

Tip: Use `fatiando.vis.gca()` to get the current axis. Or the value returned by `fatiando.vis.subplot` or `matplotlib.pyplot.subplot`.

`fatiando.vis.mpl.`**`paths`**(*pts1*, *pts2*, *style='-k'*, *linewidth=1*, *label=None*)

Plot paths between the two sets of points.

Parameters:

- **pts1** [list of lists] List of (x, y) pairs with the coordinates of the points
- **pts2** [list of lists] List of (x, y) pairs with the coordinates of the points
- **style** [str] String with the color and line style (as in matplotlib.pyplot.plot)
- **linewidth** [float] The width of the lines representing the paths
- **label** [str] If not None, then the string that will show in the legend

`fatiando.vis.mpl.`**`pcolor`**(*x*, *y*, *v*, *shape*, *interp=False*, *cmap=<Mock object at 0x5684150>*, *vmin=None*, *vmax=None*, *basemap=None*)

Make a pseudo-color plot of the data.

Parameters:

- **x, y** [array] Arrays with the x and y coordinates of the grid points. If the data is on a regular grid, then assume x varies first (ie, inner loop), then y.
- **v** [array] The scalar value assigned to the grid points.
- **shape** [tuple = (ny, nx)] Shape of the regular grid. If interpolation is not False, then will use *shape* to grid the data.
- **interp** [True or False] Wether or not to interpolate before trying to plot. If data is not on regular grid, set to True!
- **cmap** [colormap] Color map to be used. (see pyplot.cm module)
- **vmin, vmax** Saturation values of the colorbar.
- **basemap** [mpl_toolkits.basemap.Basemap] If not None, will use this basemap for plotting with a map projection (see `basemap` for creating basemaps)

Returns:

- **axes** [`matplitlib.axes`] The axes element of the plot

`fatiando.vis.mpl.`**`pick_points`**(*area*, *axes*, *marker='o'*, *color='k'*, *size=8*, *xy2ne=False*)

Get the coordinates of points by clicking with the mouse.

INSTRUCTIONS:

- Left click to pick the points;
- Press 'e' to erase the last point picked;
- Close the figure window to finish;

Parameters:

- **area** [list = [x1, x2, y1, y2]] Borders of the area containing the points
- **axes** [matplotlib Axes] The figure to use for drawing the polygon. To get an Axes instace, just do:

  ```
  from matplotlib import pyplot
  axes = pyplot.figure().add_subplot(1,1,1)
  ```

  You can plot things to `axes` before calling this function so that they'll appear on the background.
- **marker** [str] Style of the point markers (as in matplotlib.pyplot.plot)
- **color** [str] Line color (as in matplotlib.pyplot.plot)
- **size** [float] Marker size (as in matplotlib.pyplot.plot)
- **xy2ne** [True or False] If True, will exchange the x and y axis so that x points north. Use this when drawing on a map viewed from above. If the y-axis of the plot is supposed to be z (depth), then use `xy2ne=False`.

Returns:

- **points** [list of lists] List of `[x, y]` coordinates of the points

`fatiando.vis.mpl.`**`points`**(*pts*, *style='.k'*, *size=10*, *label=None*)

Plot a list of points.

Parameters:

- **pts** [list of lists] List of [x, y] pairs with the coordinates of the points
- **style** [str] String with the color and line style (as in matplotlib.pyplot.plot)
- **size** [int] Size of the plotted points
- **label** [str] If not None, then the string that will show in the legend

Returns:

- **axes** [`matplitlib.axes`] The axes element of the plot

`fatiando.vis.mpl.`**`polygon`**(*polygon*, *style='-k'*, *linewidth=1*, *fill=None*, *alpha=1.0*, *label=None*, *xy2ne=False*)

Plot a polygon.

Parameters:

- **polygon** [`fatiando.mesher.Polygon`] The polygon
- **style** [str] Color and line style string (as in matplotlib.pyplot.plot)
- **linewidth** [float] Line width
- **fill** [str] A color string used to fill the polygon. If None, the polygon is not filled
- **alpha** [float] Transparency of the fill (1 >= alpha >= 0). 0 is transparent and 1 is opaque
- **label** [str] String with the label identifying the polygon in the legend
- **xy2ne** [True or False] If True, will exchange the x and y axis so that the x coordinates of the polygon are north. Use this when drawing on a map viewed from above. If the y-axis of the plot is supposed to be z (depth), then use `xy2ne=False`.

Returns:

- **lines** [matplotlib Line object] Line corresponding to the polygon plotted

`fatiando.vis.mpl.`**`set_area`**(*area*)

Set the area of a Matplolib plot using xlim and ylim.

Parameters:

- **area** [list = [x1, x2, y1, y2]] Coordinates of the top right and bottom left corners of the area

`fatiando.vis.mpl.`**`square`**(*area*, *style='-k'*, *linewidth=1*, *fill=None*, *alpha=1.0*, *label=None*)

Plot a square.

Parameters:

- **area** [list = [x1, x2, y1, y2]] Borders of the square
- **style** [str] String with the color and line style (as in matplotlib.pyplot.plot)
- **linewidth** [float] Line width
- **fill** [str] A color string used to fill the square. If None, the square is not filled
- **alpha** [float] Transparency of the fill (1 >= alpha >= 0). 0 is transparent and 1 is opaque
- **label** [str] label associated with the square.

Returns:

- **axes** [`matplitlib.axes`] The axes element of the plot

`fatiando.vis.mpl.`**`squaremesh`**(*mesh*, *prop*, *cmap=<Mock object at 0x56840d0>*, *vmin=None*, *vmax=None*)

Make a pseudo-color plot of a mesh of squares

Parameters:

- **mesh** [`fatiando.mesher.SquareMesh` or compatible] The mesh (a compatible mesh must implement the methods `get_xs` and `get_ys`)
- **prop** [str] The physical property of the squares to use as the color scale.
- **cmap** [colormap] Color map to be used. (see pyplot.cm module)
- **vmin, vmax** [float] Saturation values of the colorbar.

Returns:

- **axes** [`matplitlib.axes`] The axes element of the plot

5.6.2 Plot 3D objects in Mayavi2 (`fatiando.vis.myv`)

Wrappers for calls to Mayavi2's *mlab* module for plotting `fatiando.mesher` objects and automating common tasks.

Objects

- `prisms`
- `polyprisms`
- `points`
- `tesseroids`

Misc objects

- `outline`
- `axes`
- `wall_north`
- `wall_south`
- `wall_east`
- `wall_west`
- `wall_top`
- `wall_bottom`
- `earth`
- `core`
- `continents`
- `meridians`
- `parallels`

Helpers

- `figure`
- `title`
- `show`
- `savefig`

`fatiando.vis.myv.`**`axes`**(*plot*, *nlabels=5*, *extent=None*, *ranges=None*, *color=(0, 0, 0)*, *width=2*, *fmt='%-#.2f'*)

Add an Axes module to a Mayavi2 plot or dataset.

Parameters:

- **plot** Either the plot (as returned by one of the plotting functions of this module) or a TVTK dataset.
- **nlabels** [int] Number of labels on the axes
- **extent** [list = [xmin, xmax, ymin, ymax, zmin, zmax]] Default if the objects extent.
- **ranges** [list = [xmin, xmax, ymin, ymax, zmin, zmax]] What will be display in the axes labels. Default is *extent*
- **color** [tuple = (r, g, b)] RGB of the color of the axes and text
- **width** [float] Line width
- **fmt** [str] Label number format

Returns:

- **axes** [Mayavi axes instace] The axes object in the pipeline

`fatiando.vis.myv.`**`continents`**(*color=(0, 0, 0)*, *linewidth=1*, *resolution=2*, *opacity=1*, *radius=6378137.0*)

Plot the outline of the continents.

Parameters:

- **color** [tuple] RGB color of the lines. Default = black
- **linewidth** [float] The width of the continent lines
- **resolution** [float] The data_source.on_ratio parameter that controls the resolution of the continents
- **opacity** [float] The opacity of the lines. Must be between 0 and 1
- **radius** [float] The radius of the sphere where the continents will be plotted. Defaults to the mean Earth radius

Returns:

- **continents** [Mayavi surface] The Mayavi surface element of the continents

`fatiando.vis.myv.`**`core`**(*inner=False*, *color=(1, 0, 0)*, *opacity=1*)

Draw a sphere representing the Earth's core.

Parameters:

- **inner** [True or False] If True, will use the radius of the inner core, else the outer core.
- **color** [tuple] RGB color of the sphere. Defaults to red.
- **opacity** [float] The opacity of the sphere. Must be between 0 and 1

Returns:

- **sphere** [Mayavi surface] The Mayavi surface element of the sphere

`fatiando.vis.myv.`**`earth`**(*color=(0.4, 0.5, 1.0)*, *opacity=1*)

Draw a sphere representing the Earth.

Parameters:

- **color** [tuple] RGB color of the sphere. Defaults to ocean blue.
- **opacity** [float] The opacity of the sphere. Must be between 0 and 1

Returns:

- **sphere** [Mayavi surface] The Mayavi surface element of the sphere

`fatiando.vis.myv.`**`figure`**(*size=None*, *zdown=True*)

Create a default figure in Mayavi with white background

Parameters:

- **size** [tuple = (dx, dy)] The size of the figure. If `None` will use the default size.
- **zdown** [True or False] If True, will turn the figure upside-down to make the z-axis point down

Return:

- **fig** [Mayavi figure object] The figure

`fatiando.vis.myv.`**`meridians`**(*longitudes*, *color=(0, 0, 0)*, *linewidth=1*, *opacity=1*)

Draw meridians on the Earth.

Parameters:

- **longitudes** [list] The longitudes where the meridians will be drawn.
- **color** [tuple] RGB color of the lines. Defaults to black.

- **linewidth** [float] The width of the lines
- **opacity** [float] The opacity of the lines. Must be between 0 and 1

Returns:

- **lines** [Mayavi surface] The Mayavi surface element of the lines

`fatiando.vis.myv.`**`outline`**(*extent=None*, *color=(0, 0, 0)*, *width=2*)

Create a default outline in Mayavi2.

Parameters:

- **extent** [list = [xmin, xmax, ymin, ymax, zmin, zmax]] Default if the objects extent.
- **color** [tuple = (r, g, b)] RGB of the color of the axes and text
- **width** [float] Line width

Returns:

- **outline** [Mayavi outline instace] The outline in the pipeline

`fatiando.vis.myv.`**`parallels`**(*latitudes*, *color=(0, 0, 0)*, *linewidth=1*, *opacity=1*)

Draw parallels on the Earth.

Parameters:

- **latitudes** [list] The latitudes where the parallels will be drawn.
- **color** [tuple] RGB color of the lines. Defaults to black.
- **linewidth** [float] The width of the lines
- **opacity** [float] The opacity of the lines. Must be between 0 and 1

Returns:

- **lines** [list] List of the Mayavi surface elements of each line

`fatiando.vis.myv.`**`points`**(*points*, *color=(0, 0, 0)*, *opacity=1*, *size=200.0*)

Plot a series of 3D points.

Note: Still doesn't plot points with physical properties.

Parameters:

- **points** [list] The list of points to plot. Each point is an [x, y, z] list with the x, y, and z coordinates of the point
- **color** [tuple = (r, g, b)] RGB of the color of the points
- **opacity** [float] Decimal percentage of opacity
- **size** [float] The size of the points (relative to their spacing)

`fatiando.vis.myv.`**`polyprisms`**(*prisms*, *prop=None*, *style='surface'*, *opacity=1*, *edges=True*, *vmin=None*, *vmax=None*, *cmap='blue-red'*, *linewidth=0.1*)

Plot a list of 3D polygonal prisms using Mayavi2.

Will not plot a value None in *prisms*.

Parameters:

- **prisms** [list of `fatiando.mesher.PolygonalPrism`] The prisms
- **prop** [str or None] The physical property of the prisms to use as the color scale. If a prism doesn't have *prop*, or if it is None, then it will not be plotted
- **style** [str] Either `'surface'` for solid prisms or `'wireframe'` for just the contour
- **opacity** [float] Decimal percentage of opacity

- **edges** [True or False] Wether or not to display the edges of the prisms in black lines. Will ignore this if `style='wireframe'`
- **vmin, vmax** [float] Min and max values for the color scale. If *None* will default to the min and max of *prop* in the prisms.
- **cmap** [Mayavi colormap] Color map to use. See the 'Colors and Legends' menu on the Mayavi2 GUI for valid color maps.
- **linewidth** [float] The width of the lines (edges) of the prisms.

Returns:

- **surface** the last element on the pipeline

`fatiando.vis.myv.`**`prisms`**(*prisms*, *prop=None*, *style='surface'*, *opacity=1*, *edges=True*, *vmin=None*, *vmax=None*, *cmap='blue-red'*, *linewidth=0.1*)

Plot a list of 3D right rectangular prisms using Mayavi2.

Will not plot a value None in *prisms*

Parameters:

- **prisms** [list of `fatiando.mesher.Prism`] The prisms
- **prop** [str or None] The physical property of the prisms to use as the color scale. If a prism doesn't have *prop*, or if it is None, then it will not be plotted
- **style** [str] Either `'surface'` for solid prisms or `'wireframe'` for just the contour
- **opacity** [float] Decimal percentage of opacity
- **edges** [True or False] Wether or not to display the edges of the prisms in black lines. Will ignore this if `style='wireframe'`
- **vmin, vmax** [float] Min and max values for the color scale. If *None* will default to the min and max of *prop* in the prisms.
- **cmap** [Mayavi colormap] Color map to use. See the 'Colors and Legends' menu on the Mayavi2 GUI for valid color maps.
- **linewidth** [float] The width of the lines (edges) of the prisms.

Returns:

- **surface** the last element on the pipeline

`fatiando.vis.myv.`**`savefig`**(*fname*, *magnification=None*)

Save a snapshot the current Mayavi figure to a file.

Parameters:

- **fname** [str] The name of the file. The format is deduced from the extension.
- **magnification** [int or None] If not None, then the scaling between the pixels on the screen, and the pixels in the file saved.

`fatiando.vis.myv.`**`show`**()

Show the 3D plot of Mayavi2.

Enters a loop until the window is closed.

`fatiando.vis.myv.`**`tesseroids`**(*tesseroids*, *prop=None*, *style='surface'*, *opacity=1*, *edges=True*, *vmin=None*, *vmax=None*, *cmap='blue-red'*, *linewidth=0.1*)

Plot a list of tesseroids using Mayavi2.

Will not plot a value None in *tesseroids*

Parameters:

- **tesseroids** [list of `fatiando.mesher.Tesseroid`] The prisms

- **prop** [str or None] The physical property of the tesseroids to use as the color scale. If a tesseroid doesn't have *prop*, or if it is None, then it will not be plotted
- **style** [str] Either `'surface'` for solid tesseroids or `'wireframe'` for just the contour
- **opacity** [float] Decimal percentage of opacity
- **edges** [True or False] Wether or not to display the edges of the tesseroids in black lines. Will ignore this if `style='wireframe'`
- **vmin, vmax** [float] Min and max values for the color scale. If *None* will default to the min and max of *prop*.
- **cmap** [Mayavi colormap] Color map to use. See the 'Colors and Legends' menu on the Mayavi2 GUI for valid color maps.
- **linewidth** [float] The width of the lines (edges).

Returns:

- **surface** the last element on the pipeline

`fatiando.vis.myv.`**`title`**`(`*text*, *color=(0, 0, 0)*, *size=0.3*, *height=1*`)`

Draw a title on a Mayavi figure.

Warning: Must be called **after** you've plotted something (e.g., prisms) to the figure. This is a bug.

Parameters:

- **text** [str] The title
- **color** [tuple = (r, g, b)] RGB of the color of the text
- **size** [float] The size of the text
- **height** [float] The height where the title will be placed on the screen

`fatiando.vis.myv.`**`wall_bottom`**`(`*bounds*, *color=(0, 0, 0)*, *opacity=0.1*`)`

Draw a 3D wall in Mayavi2 on the Bottom side.

Note: Remember that x->North, y->East and z->Down

Parameters:

- **bounds** [list = [xmin, xmax, ymin, ymax, zmin, zmax]] The extent of the region where the wall is placed
- **color** [tuple = (r, g, b)] RGB of the color of the wall
- **opacity** [float] Decimal percentage of opacity

Tip: You can use `add_axes` to create and *axes* variable and get the bounds as `axes.axes.bounds`

`fatiando.vis.myv.`**`wall_east`**`(`*bounds*, *color=(0, 0, 0)*, *opacity=0.1*`)`

Draw a 3D wall in Mayavi2 on the East side.

Note: Remember that x->North, y->East and z->Down

Parameters:

- **bounds** [list = [xmin, xmax, ymin, ymax, zmin, zmax]] The extent of the region where the wall is placed
- **color** [tuple = (r, g, b)] RGB of the color of the wall
- **opacity** [float] Decimal percentage of opacity

Tip: You can use `add_axes` to create and *axes* variable and get the bounds as `axes.axes.bounds`

`fatiando.vis.myv.`**`wall_north`**(*bounds*, *color=(0, 0, 0)*, *opacity=0.1*)

Draw a 3D wall in Mayavi2 on the North side.

Note: Remember that x->North, y->East and z->Down

Parameters:

- **bounds** [list = [xmin, xmax, ymin, ymax, zmin, zmax]] The extent of the region where the wall is placed
- **color** [tuple = (r, g, b)] RGB of the color of the wall
- **opacity** [float] Decimal percentage of opacity

Tip: You can use `add_axes` to create and *axes* variable and get the bounds as `axes.axes.bounds`

`fatiando.vis.myv.`**`wall_south`**(*bounds*, *color=(0, 0, 0)*, *opacity=0.1*)

Draw a 3D wall in Mayavi2 on the South side.

Note: Remember that x->North, y->East and z->Down

Parameters:

- **bounds** [list = [xmin, xmax, ymin, ymax, zmin, zmax]] The extent of the region where the wall is placed
- **color** [tuple = (r, g, b)] RGB of the color of the wall
- **opacity** [float] Decimal percentage of opacity

Tip: You can use `add_axes` to create and *axes* variable and get the bounds as `axes.axes.bounds`

`fatiando.vis.myv.`**`wall_top`**(*bounds*, *color=(0, 0, 0)*, *opacity=0.1*)

Draw a 3D wall in Mayavi2 on the Top side.

Note: Remember that x->North, y->East and z->Down

Parameters:

- **bounds** [list = [xmin, xmax, ymin, ymax, zmin, zmax]] The extent of the region where the wall is placed
- **color** [tuple = (r, g, b)] RGB of the color of the wall
- **opacity** [float] Decimal percentage of opacity

Tip: You can use `add_axes` to create and *axes* variable and get the bounds as `axes.axes.bounds`

`fatiando.vis.myv.`**`wall_west`**(*bounds*, *color=(0, 0, 0)*, *opacity=0.1*)

Draw a 3D wall in Mayavi2 on the West side.

Note: Remember that x->North, y->East and z->Down

Parameters:

- **bounds** [list = [xmin, xmax, ymin, ymax, zmin, zmax]] The extent of the region where the wall is placed
- **color** [tuple = (r, g, b)] RGB of the color of the wall
- **opacity** [float] Decimal percentage of opacity

Tip: You can use `add_axes` to create and *axes* variable and get the bounds as `axes.axes.bounds`

5.7 Input/Output (`fatiando.io`)

Input/Output utilities for grids, models, etc

CRUST2.0

Load and convert the CRUST2.0 global crustal model (http://igppweb.ucsd.edu/ gabi/rem.html) (Bassin et al., 2000).

- `fetch_crust2`: Download the .tar.gz archive with the model from the website
- `crust2_to_tesseroids`: Convert the CRUST2.0 model to tesseroids

References

Bassin, C., Laske, G. and Masters, G., The Current Limits of Resolution for Surface Wave Tomography in North America, EOS Trans AGU, 81, F897, 2000.

`fatiando.io.`**`crust2_to_tesseroids`**(*fname*)

Convert the CRUST2.0 model to tesseroids.

Opens the .tar.gz archive and converts the model to `fatiando.mesher.Tesseroid`. Each tesseroid will have its `props` set to the apropriate Vp, Vs and density.

The CRUST2.0 model includes 7 layers: ice, water, soft sediments, hard sediments, upper crust, middle curst and lower crust. It also includes the mantle bellow the Moho. The mantle portion is not included in this conversion because there is no way to place a bottom on it.

Parameters:

- **fname** [str] Name of the model .tar.gz archive (see `fetch_crust2`)

Returns:

- **model** [list of `fatiando.mesher.Tesseroid`] The converted model

`fatiando.io.`**`fetch_crust2`**(*fname='crust2.tar.gz'*)

Download the CRUST2.0 model from http://igppweb.ucsd.edu/~gabi/crust2.html

Parameters:

- **fname** [str] The name that the archive file will be saved when downloaded

Returns:

- **fname** [str] The downloaded file name

5.8 Graphical user interfaces (`fatiando.gui`)

Modules for graphical user interfaces (GUI)

- `simple`: Simple GUIs using the interactive capabilities of `matplotlib`

5.8.1 Simple GUIs using `matplotlib` (`fatiando.gui.simple`)

Simple GUIs using the interactive capabilities of `matplotlib`

Interactive gravimetric modeling

- `Moulder`
- `BasinTrap`
- `BasinTri`

Interactive modeling of layered media

- `Lasagne`

---

class `fatiando.gui.simple.`**`BasinTrap`**(*area*, *nodes*, *xp*, *zp*, *gz=None*)

Bases: `fatiando.gui.simple.Moulder`

Interactive gravity modeling using a trapezoidal model.

The trapezoid has two surface nodes with fixed position. The bottom two have fixed x coordinates but movable z. The x coordinates for the bottom nodes are the same as the ones for the surface nodes. The user can then model by controling the depths of the two bottom nodes.

Example:

```
# Define the area of modeling
area = (0, 1000, 0, 1000)
# Where the gravity effect is calculated
xp = range(0, 1000, 10)
zp = [0]*len(xp)
# Where the two surface nodes are. Use depth = 1 because direct modeling
# doesn't like it when the model and computation points coincide
nodes = [[100, 1], [900, 1]]
# Create the application
app = BasinTrap(area, nodes, xp, zp)
# Run it (close the window to finish)
app.run()
# and save the calculated gravity anomaly profile
app.savedata("mydata.txt")
```

Parameters:

- **area** [list = [xmin, xmax, zmin, zmax]] Are of the subsuface to use for modeling. Remember, z is positive downward.
- **nodes** [list of lists = [[x1, z1], [x2, z2]]] x and z coordinates of the two top nodes. Must be in clockwise order!
- **xp, zp** [array] Arrays with the x and z coordinates of the computation points
- **gz** [array] The observed gravity values at the computation points. Will be plotted as black points together with the modeled (predicted) data. If None, will ignore this.

class `fatiando.gui.simple.`**`BasinTri`**(*area*, *nodes*, *xp*, *zp*, *gz=None*)

Bases: `fatiando.gui.simple.Moulder`

Interactive gravity modeling using a triangular model.

The triangle has two surface nodes with fixed positions. The user can then model by controling the bottom node.

Example:

```
# Define the area of modeling
area = (0, 1000, 0, 1000)
# Where the gravity effect is calculated
xp = range(0, 1000, 10)
zp = [0]*len(xp)
# Where the two surface nodes are. Use depth = 1 because direct modeling
# doesn't like it when the model and computation points coincide
nodes = [[100, 1], [900, 1]]
# Create the application
app = BasinTri(area, nodes, xp, zp)
# Run it (close the window to finish)
app.run()
# and save the calculated gravity anomaly profile
app.savedata("mydata.txt")
```

Parameters:

- **area** [list = [xmin, xmax, zmin, zmax]] Are of the subsuface to use for modeling. Remember, z is positive downward.
- **nodes** [list of lists = [[x1, z1], [x2, z2]]] x and z coordinates of the two top nodes. Must be in clockwise order!
- **xp, zp** [array] Arrays with the x and z coordinates of the computation points
- **gz** [array] The observed gravity values at the computation points. Will be plotted as black points together with the modeled (predicted) data. If None, will ignore this.

class `fatiando.gui.simple.`**`Lasagne`**(*thickness*, *zp*, *vmin*, *vmax*, *tts=None*)

Interactive modeling of vertical seismic profiling for 1D layered media.

The wave source is assumed to be on the surface of a vertical borehole. The receivers are at given depths. What is measured is the travel-time of first arrivals.

Assumes that the thickness of the layers are known. The user then only needs to choose the velocities.

Example:

```
# Define the thickness of the layers
thickness = [10, 20, 5, 10]
# Define the measuring points along the well
zp = range(1, sum(thickness), 1)
# Define the velocity range
vmin, vmax = 1, 10000
# Run the application
app = Lasagne(thickness, zp, vmin, vmax)
app.run()
# Save the modeled data
app.savedata("mydata.txt")
```

Parameters:

- **thickness** [list] The thickness of each layer in order of increasing depth
- **zp** [list] The depths of the measurement stations (seismometers)
- **vmin, vmax** [float] Range of velocities to allow
- **tts** [array] The observed travel-time values at the measurement stations. Will be plotted as black points together with the modeled (predicted) data. If None, will ignore this.

class `fatiando.gui.simple.`**`Moulder`**(*area*, *xp*, *zp*, *gz=None*)

Interactive potential field direct modeling in 2D using polygons.

Uses module `talwani` for computations.

For the moment only works for the gravity anomaly.

To run this in a script, use:

```
# Define the area of modeling
area = (0, 1000, 0, 1000)
# Where the gravity effect is calculated
xp = range(0, 1000, 10)
zp = [0]*len(xp)
# Create the application
app = Moulder(area, xp, zp)
# Run it (close the window to finish)
app.run()
# and save the calculated gravity anomaly profile
app.savedata("mydata.txt")
```

Parameters:

- **area** [list = [xmin, xmax, zmin, zmax]] Are of the subsuface to use for modeling. Remember, z is positive downward
- **xp, zp** [array] Arrays with the x and z coordinates of the computation points
- **gz** [array] The observed gravity values at the computation points. Will be plotted as black points together with the modeled (predicted) data. If None, will ignore this.

"The truth is out there"

5.9 Miscellaneous Utilities (`fatiando.utils`)

Miscellaneous utility functions and classes.

Mathematical functions

- `normal`
- `gaussian`
- `gaussian2d`

Point scatter generation

- `random_points`
- `circular_points`
- `connect_points`

Unit conversion

- `si2mgal`
- `mgal2si`
- `si2eotvos`
- `eotvos2si`
- `si2nt`
- `nt2si`

Coordinate system conversions

- `sph2cart`

Others

- `contaminate`: Contaminate a vector with pseudo-random Gaussian noise
- `dircos`: Get the 3 coordinates of a unit vector

- `vecmean`: Take the mean array out of a list of arrays
- `vecstd` Take the standard deviation array out of a list of arrays
- `SparseList`: Store only non-zero elements on an immutable list
- `sec2hms`: Convert seconds to hours, minutes, and seconds
- `sec2year`: Convert seconds to Julian years
- `year2sec`: Convert Julian years to seconds

---

class `fatiando.utils.`**`SparseList`**(*size*, *elements=None*)

Bases: `object`

Store only non-zero elements on an immutable list.

Can iterate over and access elements just like if it were a list.

Parameters:

- **size** [int] Size of the list.
- **elements** [dict] Dictionary used to initialize the list. Keys are the index of the elements and values are their respective values.

Example:

```
>>> l = SparseList(5)
>>> l[3] = 42.0
>>> print len(l)
5
>>> print l[1], l[3]
0.0 42.0
>>> l[1] += 3.0
>>> for i in l:
...     print i,
0.0 3.0 0.0 42.0 0.0
>>> l2 = SparseList(4, elements={1:3.2, 3:2.8})
>>> for i in l2:
...     print i,
0.0 3.2 0.0 2.8
```

`fatiando.utils.`**`circular_points`**(*area*, *n*, *random=False*)

Generate a set of n points positioned in a circular array.

The diameter of the circle is equal to the smallest dimension of the area

Parameters:

- **area** [list = [x1, x2, y1, y2]] Area inside of which the points are contained
- **n** [int] Number of points
- **random** [True or False] If True, positions of the points on the circle will be chosen at random

Result:

- **points** [list] List of (x, y) coordinates of the points

`fatiando.utils.`**`connect_points`**(*pts1*, *pts2*)

Connects each point in the first list with all points in the second. If the first list has N points and the second has M, the result are 2 lists with N*M points each, representing the connections.

Parameters:

- **pts1** [list] List of (x, y) coordinates of the points.
- **pts2** [list] List of (x, y) coordinates of the points.

Returns:

- **results** [lists of lists = [connect1, connect2]] 2 lists with the connected points

`fatiando.utils.`**`contaminate`**(*data*, *stddev*, *percent=False*, *return_stddev=False*)

Add pseudorandom gaussian noise to an array.

Noise added is normally distributed.

Parameters:

- **data** [list or array] Data to contaminate
- **stddev** [float] Standard deviation of the Gaussian noise that will be added to *data*
- **percent** [True or False] If `True`, will consider *stddev* as a decimal percentage and the standard deviation of the Gaussian noise will be this percentage of the maximum absolute value of *data*
- **return_stddev** [True or False] If `True`, will return also the standard deviation used to contaminate *data*

Returns:

if *return_stddev* is `False`:

- **contam** [array] The contaminated data array

else:

- **results** [list = [contam, stddev]] The contaminated data array and the standard deviation used to contaminate it.

`fatiando.utils.`**`dircos`**(*inc*, *dec*)

Returns the 3 coordinates of a unit vector given its inclination and declination.

Note: Coordinate system is assumed to be x->North, y->East, z->Down. Inclination is positive down and declination is measured with respect to x (North).

Parameter:

- **inc** [float] The inclination of the vector (in degrees)
- **dec** [float] The declination of the vector (in degrees)

Returns:

- **vect** [list = [x, y, z]] The unit vector

`fatiando.utils.`**`eotvos2si`**(*value*)

Convert a value from Eotvos to SI units.

Parameters:

- **value** [number or array] The value in Eotvos

Returns:

- **value** [number or array] The value in SI

`fatiando.utils.`**`gaussian`**(*x*, *mean*, *std*)

Non-normalized Gaussian function

$$G(x, \bar{x}, \sigma) = \exp\left(-\frac{(x-\bar{x})^2}{\sigma^2}\right)$$

Parameters:

- **x** [float or array] Values at which to calculate the Gaussian function
- **mean** [float] The mean of the distribution $\bar{x}$
- **std** [float] The standard deviation of the distribution σ

Returns:

- **gauss** [array] Gaussian function evaluated at *x*

`fatiando.utils.`**`gaussian2d`**(*x*, *y*, *sigma_x*, *sigma_y*, *x0=0*, *y0=0*, *angle=0.0*)

Non-normalized 2D Gaussian function

Parameters:

- **x, y** [float or arrays] Coordinates at which to calculate the Gaussian function
- **sigma_x, sigma_y** [float] Standard deviation in the x and y directions
- **x0, y0** [float] Coordinates of the center of the distribution
- **angle** [float] Rotation angle of the gaussian measure from the x axis (north) growing positive to the east (positive y axis)

Returns:

- **gauss** [array] Gaussian function evaluated at *x*, *y*

`fatiando.utils.`**`mgal2si`**(*value*)

Convert a value from mGal to SI units.

Parameters:

- **value** [number or array] The value in mGal

Returns:

- **value** [number or array] The value in SI

`fatiando.utils.`**`normal`**(*x*, *mean*, *std*)

Normal distribution.

$$N(x, \bar{x}, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\bar{x})^2}{\sigma^2}\right)$$

Parameters:

- **x** [float or array] Value at which to calculate the normal distribution
- **mean** [float] The mean of the distribution $\bar{x}$
- **std** [float] The standard deviation of the distribution σ

Returns:

- **normal** [array] Normal distribution evaluated at *x*

`fatiando.utils.`**`nt2si`**(*value*)

Convert a value from nanoTesla to SI units.

Parameters:

- **value** [number or array] The value in nanoTesla

Returns:

- **value** [number or array] The value in SI

`fatiando.utils.`**`random_points`**(*area*, *n*)

Generate a set of n random points.

Parameters:

- **area** [list = [x1, x2, y1, y2]] Area inside of which the points are contained
- **n** [int] Number of points

Result:

- **points** [list] List of (x, y) coordinates of the points `fatiando.utils.`**`sec2hms`**`(`*seconds*`)`

Convert seconds into a string with hours, minutes and seconds.

Parameters:

- **seconds** [float] Time in seconds

Returns:

- **time** [str] String in the format `'%dh %dm %2.5fs'`

Example:

```
>>> print sec2hms(62.2)
0h 1m 2.20000s
>>> print sec2hms(3862.12345678)
1h 4m 22.12346s
```

`fatiando.utils.`**`sec2year`**`(`*seconds*`)`

Convert seconds into decimal Julian years.

Julian years have 365.25 days.

Parameters:

- **seconds** [float] Time in seconds

Returns:

- **years** [float] Time in years

Example:

```
>>> print sec2year(31557600)
1.0
```

`fatiando.utils.`**`si2eotvos`**`(`*value*`)`

Convert a value from SI units to Eotvos.

Parameters:

- **value** [number or array] The value in SI

Returns:

- **value** [number or array] The value in Eotvos

`fatiando.utils.`**`si2mgal`**`(`*value*`)`

Convert a value from SI units to mGal.

Parameters:

- **value** [number or array] The value in SI

Returns:

- **value** [number or array] The value in mGal

`fatiando.utils.`**`si2nt`**`(`*value*`)`

Convert a value from SI units to nanoTesla.

Parameters:

- **value** [number or array] The value in SI

Returns:

- **value** [number or array] The value in nanoTesla

`fatiando.utils.`**`sph2cart`**`(`*lon*, *lat*, *height*`)`

Convert spherical coordinates to Cartesian geocentric coordinates.

Parameters:

- **lon, lat, height** [floats] Spherical coordinates. lon and lat in degrees, height in meters. height is the height above mean Earth radius.

Returns:

- **x, y, z** [floats] Converted Cartesian coordinates

`fatiando.utils.`**`vecmean`**`(`*arrays*`)`

Take the mean array out of a list of arrays.

Parameter:

- **arrays** [list] List of arrays

Returns:

- **mean** [array] The mean of each element in the arrays

Example:

```
>>> print vecmean([[1, 1, 2], [2, 3, 5]])
[ 1.5  2.   3.5]
```

`fatiando.utils.`**`vecstd`**`(`*arrays*`)`

Take the standard deviation array out of a list of arrays.

Parameter:

- **arrays** [list] List of arrays

Returns:

- **std** [array] Standard deviation of each element in the arrays

Example:

```
>>> print vecstd([[1, 1, 2], [2, 3, 5]])
[ 0.5  1.   1.5]
```

`fatiando.utils.`**`year2sec`**`(`*years*`)`

Convert decimal Julian years into seconds.

Julian years have 365.25 days.

Parameters:

- **years** [float] Time in years

Returns:

- **seconds** [float] Time in seconds

Example:

```
>>> print year2sec(1)
31557600.0
```

5.10 Logging (`fatiando.logger`)

Logging utilities for fatiando.

This module is basically a wrapper around Pythons *logging* module.

Functions

- `get`: Create a logger and enable logging using the default settings
- `tofile`: Enable logging to a file
- `header`: Generate a header message string with the current version, changeset information and date

- dummy: Get a logger without any handlers (for use in modules)

Usage

In a module, use a logger without any handlers so that it only logs if a script wants to log:

```
>>> # in fatiando.package.module.py
>>> import fatiando.logger
>>> def myfunc():
...     log = fatiando.logger.dummy('fatiando.package.module.myfunc')
...     log.info("Only logs if a script calls fatiando.logger.get")
```

Then it can be used in a script:

```
>>> myfunc()
>>> # Nothing happens
>>> import sys
>>> # Get a logger to stdout
>>> log = fatiando.logger.get(stream=sys.stdout)
>>> myfunc()
Only logs if a script calls fatiando.logger.get
>>> log.info("This is an info msg printed to stdout from the script")
This is an info msg printed to stdout from the script
>>> log.debug("This is a debug msg NOT printed")
>>> # Uncomment bellow to log to a file as well!
>>> # log = fatiando.logger.tofile(log, 'mylogfile.log')
>>> log.warning('Warning printed to both stdout and log file')
Warning printed to both stdout and log file
>>> log.error('and this is an Error message.')
and this is an Error message.
```

Note: Importing this module assigns a *logging.NullHandler* to the base logger of *fatiando*, whose name is `'fatiando'`. This way, log messages are only printed if a script calls `fatiando.logger.get` or assigns a Handler to it.

`fatiando.logger.`**`dummy`**(*name='fatiando'*)

Get a logger without any handlers.

For use inside modules.

Parameters:

- **name** [str] Name of the logger. Use the module name as *name* (including the full package hierarchy)

Returns:

- **log** [*logging.Logger*] A logger without any handlers so that it only prints when `fatiando.logger.get` or `fatiando.logger.tofile` are called

Examples:

```
>>> # in fatiando.mymod.py
>>> import fatiando.logger
>>> def myfunc():
...     log = fatiando.logger.dummy('fatiando.mymod.myfunc')
...     log.info("Not logged unless a script wants it to")
>>> myfunc()
>>>
```

`fatiando.logger.`**`get`**(*level=20, stream=<open file '<stderr>', mode 'w' at 0x2b3c0cf99270>*)

Create a logger using the default settings for Fatiando.

Parameters:

- **level** [int] Default to *logging.INFO*. See *logging* module

- **stream** [file] A stream to log to. Default to *sys.stderr*

Returns:

- **log** [*logging.Logger*] A logger with the level and name set

`fatiando.logger.`**`header`**(*comment=''*)

Generate a header message with the current version, changeset information and date.

Parameters:

- **comment** [str] Character inserted at the beginning of each line. Use this to make a message that can be inserted in source code files as comments.

Returns:

- **msg** [str] The header message

`fatiando.logger.`**`tofile`**(*logger*, *fname*, *level=10*)

Enable logging to a file.

Will enable file logging to the given *logger*.

Parameters:

- **logger** [*logging.Logger*] A logger, as returned by `fatiando.logger.get`
- **fname** [str] Log file name
- **level** [int] The logging level. Default to *logging.DEBUG*. See *logging* module

Returns:

- **log** [*logging.Logger*] *logger* with added FileHandler

Examples:

```
>>> import logging
>>> # Need to mock the FileHandler so that it works with StringIO
>>> from StringIO import StringIO
>>> f = StringIO()
>>> logging.FileHandler = lambda f, mode: logging.StreamHandler(f)
>>> # Now for the actual logger example!
>>> import sys
>>> import fatiando.logger
>>> log = fatiando.logger.tofile(fatiando.logger.get(stream=sys.stdout), f,
...                              level=logging.DEBUG)
>>> log.debug("logged to file but not stdout!")
>>> print f.getvalue().strip()
DEBUG:fatiando: logged to file but not stdout!
```

5.11 Physical constants and unit conversions (`fatiando.constants`)

Holds all physical constants and unit conversions used in `fatiando`.

All modules should import the constants from here!

All constants should be in SI, unless otherwise stated!

`fatiando.constants.`**`CM`** **= 1e-07**

Proportionality constant used in the magnetic method in henry/m (SI)

`fatiando.constants.`**`G`** **= 6.673e-11**

The gravitational constant in $m^3 kg^{-1} s^{-1}$

`fatiando.constants.`**`MEAN_EARTH_RADIUS`** **= 6378137.0**
The mean earth radius in meters

`fatiando.constants.`**`SI2EOTVOS`** **= 1000000000.0**
Conversion factor from SI units to Eotvos: $1/s^2 = 10^9\ Eotvos$

`fatiando.constants.`**`SI2MGAL`** **= 100000.0**
Conversion factor from SI units to mGal: $1\ m/s^2 = 10^5\ mGal$

`fatiando.constants.`**`T2NT`** **= 1000000000.0**
Conversion factor from tesla to nanotesla

`fatiando.constants.`**`THERMAL_DIFFUSIVITY`** **= 1e-06**
The default thermal diffusivity in m^2/s

`fatiando.constants.`**`THERMAL_DIFFUSIVITY_YEAR`** **= 31.5576**
The default thermal diffusivity but in $m^2/year$

5.12 Inverse problem solving tools (`fatiando.inversion`)

Everything you need to solve inverse problems!

The main components of this package are:

Regularizing functions

- `regularizer`

This module has many classes that implement a range of regularizing functions. These classes know how to calculate the Hessian matrix, gradient vector, and value of the regularizing function for a given parameter vector. Most of these classes are generic and can be applied to any inverse problem, though some are designed for a specific parametrization (2D or 3D grids, for example).

The regularizer classes can be passed to any inverse problem solver (bellow).

See the `Regularizer` class for the general structure the solvers expect from regularizer classes.

Inverse problem solvers

- `linear`
- `gradient`
- `heuristic`

These modules have factory functions that generate generic inverse problem solvers. Instead of solving the inverse problem, they return a solver function with they solver parameters (step size, maximum iterations, etc) already set. The solver functions are actually Python generators (http://wiki.python.org/moin/Generators). Generators are special Python functions that have a `yield` statement instead of a `return`. The difference is that generators should be used in `for` loops and return one iteration of the solving process per loop iteration.

The solver functions receive only two parameters: a list of data modules (see `datamodule`) and a list of regularizers (see `regularizer`). Data modules know how to calculate things like the Hessian matrix and gradient vector for a given data set and parametrization (specific to a given inverse problem). Regularizers know how to calculate the same things but for a given regularizing function and parametrization. This way, the user can combine any number of data sets and regularizers as he/she wants. So we can program the solvers and regularizers once and use them in any inverse problem!

A typical factory function for an iterative solver looks like:

```
def factory(initial, step, maxit):
    # Define the solver generator. The parameters need by this specific
    # solver are passed to it as optional parameters.
    # dms is a list of data modules and regs is a list of regularizers
    def solver(dms, regs, initial=initial, step=step, maxit=maxit):
        # Initialize the solver parameters
        ...
        for it in xrange(maxit):
            # Do an iteration and find an estimate p
            ...
            # Now comes the cool part! Spit out a changeset with the result
            # of this iteration in a dictionary. goals is a list of the
            # goal function values until this iteration. misfits is the same
            # but for the data-misfit function. residuals is a list of the
            # residual vectors of each data module
            yield {'estimate':p, 'goals':goals, 'misfits':misfits,
                   'residuals':residuals}
    # The factory function returns the solver function (Python magic) which
    # can be passed to a particular inverse problem.
    return solver
```

Example of using a solver

Lets say I have a module that solves a particular inverse problem called `myinvprob.py`. This module would look something like this:

```
# myinvprob.py
from fatiando import inversion

class MyDataModule(inversion.datamodule.DataModule):
    """
    My personal data module. Implements the methods in the DataModule class
    for this specific problem.
    """
    def __init__(self, data):
        ...
    ...

# Make a solver for this inverse problem using damping regularization
def solve(data, solver, damping=0):
    """
    Damping solver.

    Parameters:
    ...
    * solver
        A solver generator produced by a factory function

    """
    dms = [MyDataModule(data)]
    regs = [inversion.regularizer.Damping(damping)]
    # Now, run all iterations until the solver generator stops
    for chset in solver(dms, regs):
        continue
    # collect the results from the last changeset
    estimate = chset['estimate']
    # and return
    return estimate

# You can also make an generator solver for this problem. This way the user
# can run through the iterations to see how they progress
def iterate(data, solver, damping=0):
    ...
    # Define data modules and regularizers the same way
    ...
    # But this time, yield the estimate at each iteration
    for chset in solver(dms, regs):
        yield chset['estimate']
```

These solvers can then be called from scripts, like so:

```
# My script
from fatiando import myinvprob
from fatiando.inversion import gradient

# Load the data or generate synthetic data and pick an initial estimate
...

# Solve the problem using Newton's method
solver = gradient.newton(initial, maxit=100, tol=10**(-5))

# Solve it all in one go
estimate = myinvprob.solve(data, solver, damping=0.1)

# or run through the iterations, plotting each step
for estimate in myinvprob.iterate(data, solver, damping=0.1):
    # Plot estimate
    ...
```

Real usage examples

Some modules that use the `inversion` API:

- `fatiando.seismic.profile`
- `fatiando.seismic.srtomo`
- `fatiando.seismic.epic2d`
- `fatiando.geothermal.climsig`
- `fatiando.gravmag.basin2d`

5.12.1 Solvers for linear problems (`fatiando.inversion.linear`)

Factory functions for generic linear inverse problem solvers.

- `overdet`
- `underdet`

The factory functions produce the actual solver functions. These solver functions are Python generator functions that yield only once. This might seem unnecessary but it is done so that the linear solvers are compatible with the non-linear solvers (e.g., `gradient` and `heuristic`).

This module uses dense matrices (`numpy` arrays) by default. If you want to enable the use of sparse matrices from `scipy.sparse`, call function `fatiando.inversion.linear.use_sparse` **before** creating any solver functions!

In the case of linear problems, solver functions have the general format:

```
def solver(dms, regs):
    # Start-up
    ...
    # Calculate the estimate
    ...
    # yield the results
    yield {'estimate':p, 'misfits':[misfit], 'goals':[goal],
           'residuals':residuals}
```

Parameters:

- **dms** [list] List of data modules. Data modules should be child-classes of the `DataModule` class.

- **regs** [list] List of regularizers. Regularizers should be child-classes of the `Regularizer` class.

> **Note:** The regularizing functions must also be linear!

Yields:

- **changeset** [dict] A dictionary with the final solution: `changeset = {'estimate':p, 'misfits':[misfit], 'goals':[goal], 'residuals':residuals}`
  - **p** [array] The parameter vector.
  - **misfit** [list] The data-misfit function value
  - **goal** [list] The goal function value
  - **residuals** [list] List with the residual vectors from each data module at this iteration

`fatiando.inversion.linear.`**`overdet`**`(`*nparams*`)`

Factory function for a linear least-squares solver to an overdetermined problem (more data than parameters).

The problem at hand is finding the vector $\bar{p}$ that produces a predicted data vector $\bar{d}$ as close as possible to an observed data vector $\bar{d}^o$.

In linear problems, the relation between the parameters and the predicted data are expressed through the linear system

$$\bar{\bar{G}}\bar{p} = \bar{d}$$

where $\bar{\bar{G}}$ is the Jacobian (or sensitivity) matrix. In the **over determined** case, matrix $\bar{\bar{G}}$ has more lines than columns, i.e., more equations than unknowns.

The least-squares estimate of $\bar{p}$ can be found by minimizing the goal function

$$\Gamma(\bar{p}) = \bar{r}^T\bar{r} + \sum_{k=1}^{L}\mu_k\theta_k(\bar{p})$$

where $\bar{r} = \bar{d}^o - \bar{d}$ is the residual vector, $\theta_k(\bar{p})$ are regularizing functions, and μ_k are regularizing parameters (positive scalars).

The mininum of the goal function can be calculated by solving the linear system

$$\bar{\bar{H}}\hat{\bar{p}} = -\bar{g}$$

where $\bar{\bar{H}}$ is the Hessian matrix of the goal function, $\bar{g}$ is the gradient vector of the goal function, and $\hat{\bar{p}}$ is the estimated parameter vector.

The Hessian of the goal function is given by

$$\bar{\bar{H}} = 2\bar{\bar{G}}^T\bar{\bar{G}} + \sum_{k=1}^{L}\mu_k\bar{\bar{H}}_k$$

where $\bar{\bar{H}}_k$ are the Hessian matrices of the regularizing functions.

The gradient vector of the goal function is given by

$$\bar{g} = -2\bar{\bar{G}}^T\bar{d}^o + \sum_{k=1}^{L}\mu_k\bar{g}_k$$

where $\bar{g}_k$ are the gradient vectors of the regularizing functions.

Parameters:

- **nparams** [int] The number of parameters in vector $\bar{p}$ (usually something like `mesh.size`)

Returns

- **solver** [function] A Python generator function that solves an linear overdetermined inverse problem using the parameters given above.

`fatiando.inversion.linear.use_sparse()`

Configure the gradient solvers to use the sparse conjugate gradient linear system solver from *scipy.sparse*.

Note: This does not make the data modules use sparse matrices! That must be implemented for each inverse problem separately.

5.12.2 Gradient solvers (`fatiando.inversion.gradient`)

Factory functions for generic inverse problem solvers using gradient optimization methods.

- `newton`
- `levmarq`
- `steepest`

This module uses dense matrices (`numpy` arrays) by default. If you want to enable the use of sparse matrices from `scipy.sparse`, call function `fatiando.inversion.gradient.use_sparse` **before** creating any solver functions!

The factory functions produce the actual solver functions. Solver functions are Python generator functions that have the general format:

```
def solver(dms, regs, **kwargs):
    # Start-up
    ...
    # yield the initial estimate
    yield {'estimate':p, 'misfits':misfits, 'goals':goals,
           'residuals':residuals}
    for i in xrange(maxit):
        # Perform an iteration
        ...
        yield {'estimate':p, 'misfits':misfits, 'goals':goals,
               'residuals':residuals}
```

Parameters:

- **dms** [list] List of data modules. Data modules should be child-classes of the `DataModule` class.
- **regs** [list] List of regularizers. Regularizers should be child-classes of the `Regularizer` class.
- **kwargs** Are how the factory functions pass the needed parameters to the solvers. Not to be altered outside the factory functions.

Yields:

- **changeset** [dict] A dictionary with the solution at the current iteration: `changeset = {'estimate':p, 'misfits':misfits, 'goals':goals, 'residuals':residuals}`
  - **p** [array] The parameter vector.
  - **misfits** [list] The data-misfit function values per iteration
  - **goals** [list] The goal function values per iteration
  - **residuals** [list] List with the residual vectors from each data module at this iteration

`fatiando.inversion.gradient.`**`levmarq`**(*initial*, *damp=1.0*, *factor=10.0*, *maxsteps=20*, *maxit=100*, *tol=1e-05*, *diag=False*)

Factory function for the non-linear inverse problem solver using the Levenberg-Marquardt algorithm.

The increment to the parameter vector $\bar{p}$ is calculated by (Kelley, 1999)

$$\Delta\bar{p} = -[\bar{\bar{H}} + \lambda\bar{\bar{I}}]^{-1}\bar{g}$$

where λ is a damping factor (step size), $\bar{\bar{H}}$ is the Hessian matrix, $\bar{\bar{I}}$ is the identity matrix, and $\bar{g}$ is the gradient vector.

Parameters:

- **initial** [array] The initial estimate of the parameters
- **damp** [float] The initial damping factor (λ)
- **factor** [float] The increment/decrement to the damping factor at each iteration. Should be `factor > 1`
- **maxsteps** [int] The maximum number os times to try to take a step before giving up
- **maxit** [int] Maximum number of iterations
- **tol** [float] Relative tolerance for decreasing the goal function to before terminating
- **diag** [True or False] If True, will use the diagonal of the Hessian matrix instead of the identity matrix. Only use this is the parameters are different physical quantities (like time and distance, for example)

Returns:

- **solver** [function] A Python generator function that solves an inverse problem using the parameters given above.

References:

Kelley, C. T., 1999, Iterative methods for optimization: Raleigh: SIAM.

`fatiando.inversion.gradient.`**`newton`**(*initial*, *maxit=100*, *tol=1e-05*)

Factory function for the non-linear inverse problem solver using Newton's method.

The increment to the parameter vector $\bar{p}$ is calculated by (Kelley, 1999)

$$\Delta\bar{p} = -\bar{\bar{H}}^{-1}\bar{g}$$

where $\bar{\bar{H}}$ is the Hessian matrix and $\bar{g}$ is the gradient vector.

Parameters:

- **initial** [array] The initial estimate of the parameters
- **maxit** [int] Maximum number of iterations
- **tol** [float] Relative tolerance for decreasing the goal function to before terminating

Returns:

- **solver** [function] A Python generator function that solves an inverse problem using the parameters given above.

References:

Kelley, C. T., 1999, Iterative methods for optimization: Raleigh: SIAM.

`fatiando.inversion.gradient.`**`steepest`**(*initial*, *step=0.1*, *maxit=500*, *tol=1e-05*, *armijo=True*, *maxsteps=20*)

Factory function for the non-linear inverse problem solver using the Steepest Descent algorithm.

The increment to the parameter vector $\bar{p}$ is calculated by (Kelley, 1999)

$$\Delta\bar{p} = -\lambda\bar{g}$$

where λ is the step size and $\bar{g}$ is the gradient vector.

Optionally, the step size can be determined thought a line search algorithm using the Armijo rule (Kelley, 1999). In this case

$$\lambda = \beta^m$$

where $1 > \beta > 0$ and $m \geq 0$ is an interger that controls the step size. The line search finds the smallest m that satisfies the condition (Armijo rule)

$$\Gamma(\bar{p} + \Delta\bar{p}) - \Gamma(\bar{p}) < \alpha\beta^m ||\bar{g}(\bar{p})||^2$$

where $\Gamma(\bar{p})$ is the goal function evaluated for parameter vector $\bar{p}$, $\alpha = 10^{-4}$, and $\bar{g}(\bar{p})$ is the gradient vector of $\Gamma(\bar{p})$.

Parameters:

- **initial** [array] The initial estimate of the parameters
- **step** [float] The step size (if using Armijo, β, else λ)
- **maxit** [int] Maximum number of iterations
- **tol** [float] Relative tolerance for decreasing the goal function to before terminating
- **armijo** [True or False] If True, will use the Armijo rule to determine the best step size at each iteration. It's highly recommended to use this.
- **maxsteps** [int] If using Armijo, the maximum number os times to try to take a step before giving up (i.e., the maximum value of m). If not using Armijo, maxsteps is ignored

Returns:

- **solver** [function] A Python generator function that solves an inverse problem using the parameters given above.

References:

Kelley, C. T., 1999, Iterative methods for optimization: Raleigh: SIAM.

`fatiando.inversion.gradient.use_sparse()`

Configure the gradient solvers to use the sparse conjugate gradient linear system solver from *scipy.sparse*.

Note: This does not make the data modules use sparse matrices! That must be implemented for each inverse problem separately.

5.12.3 Heuristic solvers (`fatiando.inversion.heuristic`)

Heuristic solvers for generic inverse problems.

Sorry, nothing yet...

5.12.4 Regularizing functions (`fatiando.inversion.regularizer`)

Base Regularizer class with the format expected by all inverse problem solvers, plus a range of regularizing functions already implemented.

Tikhonov regularization

- `Damping`
- `Smoothness1D`

- Smoothness2D

Total Variation

- TotalVariation1D
- TotalVariation2D

Equality constraint

- Equality

---

class `fatiando.inversion.regularizer.`**`Damping`**(*mu*, *nparams*, *sparse=False*)

Bases: `fatiando.inversion.regularizer.Regularizer`

Damping regularization. Also known as Tikhonov order 0, Ridge Regression, or Minimum Norm.

Requires that the estimate have its l2-norm as close as possible to zero.

This regularizing function has the form

$$\theta(\bar{p}) = \bar{p}^T\bar{p}$$

The gradient and Hessian matrix are, respectively:

$$\bar{g}(\bar{p}) = 2\bar{\bar{I}}\bar{p}$$

and

$$\bar{\bar{H}}(\bar{p}) = 2\bar{\bar{I}}$$

where $\bar{\bar{I}}$ is the identity matrix.

Parameters:

- **mu** [float] The regularizing parameter. A positve scalar that controls the tradeoff between data fitting and regularization. I.e., how much damping to apply
- **nparams** [int] Number of parameters in the inversion
- **sparse** [True or False] Wether or not to use sparce matrices from scipy

Examples:

```
>>> import numpy
>>> p = [1, 2, 2]
>>> hessian = numpy.array([[1, 0, 0], [2, 0, 0], [4, 0, 0]])
>>> damp = Damping(0.1, nparams=3)
>>> print damp.value(p)
0.9
>>> print damp.sum_hessian(hessian, p)
[[ 1.2  0.   0. ]
 [ 2.   0.2  0. ]
 [ 4.   0.   0.2]]
```

class `fatiando.inversion.regularizer.`**`Equality`**(*mu*, *reference*)

Bases: `fatiando.inversion.regularizer.Regularizer`

Equality constraints.

Imposes that some or all of the parameters be as close as possible to given reference values.

This regularizing function has the form

$$\theta(\bar{p}) = (\bar{p} - \bar{p}^a)^T\bar{\bar{A}}^T\bar{\bar{A}}(\bar{p} - \bar{p}^a)$$

Vector $\bar{p}^a$ contains the refence values and matrix $\bar{\bar{A}}$ is a diagonal matrix. The elements in the diagonal of $\bar{\bar{A}}$ are either 1 or 0. If there is a reference for parameter i, then $A_{ii} = 1$, else $A_{ii} = 0$. Since this is a bit hard to explain, I'll just give an example. Suppose there are 3 parameters and I want to impose that the second one be as close as possible to the number 26. Then,

$$\bar{p} = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix}, \quad \bar{p}^{\,a} = \begin{bmatrix} 0 \\ 26 \\ 0 \end{bmatrix} \quad \text{and} \quad \bar{\bar{A}} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

The gradient and Hessian matrix are, respectively:

$$\bar{g}(\bar{p}) = 2\bar{\bar{A}}^T \bar{\bar{A}}(\bar{p} - \bar{p}^{\,a})$$

and

$$\bar{\bar{H}}(\bar{p}) = 2\bar{\bar{A}}^T \bar{\bar{A}}$$

Parameters:

- **mu** [float] The regularizing parameter. A positve scalar that controls the tradeoff between data fitting and regularization. I.e., how much equality to impose
- **reference** [dict] Dictionay with the reference values for the parameters you with to constrain. The keys are the indexes of the parameters in the parameter vector. The respective values are the reference value for each parameter. For example, to constrain parameter 1 to be as close as possible to 3.4 and parameter 57 to be as close as possible to 43.7:

  ```
  reference = {1:3.4, 57:43.7}
  ```

class `fatiando.inversion.regularizer.`**`Regularizer`**`(`*mu*`)`

Bases: `object`

A generic regularizing function module.

Use this class as a skeleton for building custom regularizer modules, like smoothness, damping, total variation, etc.

Regularizer classes are how each inverse problem solver knows how to calculate things like:

- Value of the regularizing function
- Gradient of the regularizing function
- Hessian of the regularizing function

Not all solvers use all of the above. For examples, heuristic solvers don't require gradient and hessian calculations.

This class has templates for all of these methods so that all solvers know what to expect.

Constructor parameters common to all methods:

- **mu** [float] The regularizing parameter. A positve scalar that controls the tradeoff between data fitting and regularization.

`sum_gradient``(`*gradient*, *p=None*`)`

Sums the gradient vector of this regularizer module to *gradient* and returns the result.

Parameters:

- **gradient** [array] The old gradient vector
- **p** [array] The parameter vector

Note: Solvers for linear problems will use `p = None` so that the class knows how to calculate gradients more efficiently for these cases.

Returns:

- **new_gradient** [array] The new gradient vector

sum_hessian(*hessian*, *p=None*)
Sums the Hessian matrix of this regularizer module to *hessian* and returns the result.

Parameters:

- **hessian** [array] 2D array with the old Hessian matrix
- **p** [array] The parameter vector

Note: Solvers for linear problems will use `p = None` so that the class knows how to calculate gradients more efficiently for these cases.

Returns:

- **new_hessian** [array] 2D array with the new Hessian matrix

class `fatiando.inversion.regularizer.`**`Smoothness`**(*mu*, *nparams*, *sparse=False*)
Bases: `fatiando.inversion.regularizer.Regularizer`

Smoothness regularization for n-dimensional problems. Imposes that adjacent parameters have values as close as possible to each other. What *adjacent* means depends of the dimension of the problem. It can be spacially adjacent, or just adjacent in the parameter vector, or both.

This class provides a template for smoothness classes of a specific dimension.

Warning: DON'T USE THIS CLASS DIRECTLY! Instead, use the Smoothness*D classes.

This regularizing function has the form

$$\theta(\bar{p}) = \bar{p}^T \bar{\bar{R}}^T \bar{\bar{R}} \bar{p}$$

The gradient and Hessian matrix are, respectively:

$$\bar{g}(\bar{p}) = 2\bar{\bar{R}}^T \bar{\bar{R}} \bar{p}$$

and

$$\bar{\bar{H}}(\bar{p}) = 2\bar{\bar{R}}^T \bar{\bar{R}}$$

where $\bar{\bar{R}}$ is a finite difference matrix.

Parameters:

- **mu** [float] The regularizing parameter. A positve scalar that controls the tradeoff between data fitting and regularization. I.e., how much smoothness to apply.
- **nparams** [int] Number of parameters in the inversion
- **sparse** [True or False] Wether or not to use sparce matrices from scipy

class `fatiando.inversion.regularizer.`**`Smoothness1D`**(*mu*, *nparams*, *sparse=False*)
Bases: `fatiando.inversion.regularizer.Smoothness`

Smoothness regularization for 1D problems. Also known as Tikhonov order 1. Imposes that adjacent parameters have values as close as possible to each other. By adjacent, I mean next to each other in the parameter vector, e.g., p[2] and p[3].

For example, if there are 7 parameters, matrix $\bar{\bar{R}}$ will be

$$\bar{\bar{R}} = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix}$$

Parameters:

- **mu** [float] The regularizing parameter. A positve scalar that controls the tradeoff between data fitting and regularization. I.e., how much smoothness to apply.
- **nparams** [int] Number of parameters in the inversion
- **sparse** [True or False] Wether or not to use sparce matrices from scipy

class `fatiando.inversion.regularizer.`**`Smoothness2D`**(*mu*, *shape*, *sparse=False*)

Bases: `fatiando.inversion.regularizer.Smoothness`

Smoothness regularization for 2D problems. Also known as Tikhonov order 1.

Imposes that **spacially** adjacent parameters have values as close as possible to each other. By spacially adjacent, I mean that I assume the parameters are originaly placed on a grid and the grid is then flattened to make the parameter vector.

For example, if the parameters are on a 2 x 2 grid (for example, in 2D linear gravimetric problems), there are 4 parameters on the parameter vector

$$\text{grid} = \begin{pmatrix} p_1 & p_2 \\ p_3 & p_4 \end{pmatrix}, \quad \bar{p} = \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{pmatrix}$$

In the case of our example above, the matrix $\bar{\bar{R}}$ will be

$$\bar{\bar{R}} = \begin{bmatrix} 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

Parameters:

- **mu** [float] The regularizing parameter. A positve scalar that controls the tradeoff between data fitting and regularization. I.e., how much smoothness to apply.
- **shape** [tuple = (ny, nx)] Number of parameters in each direction of the grid
- **sparse** [True or False] Wether or not to use sparce matrices from scipy

class `fatiando.inversion.regularizer.`**`TotalVariation`**(*mu*, *nparams*, *beta=1e-10*, *sparse=False*)

Bases: `fatiando.inversion.regularizer.Regularizer`

Total variation regularization for n-dimensional problems. Imposes that adjacent parameters have values as close as possible to each other, in a **l1-norm** sense. What *adjacent* means depends of the dimension of the problem. It can be spacially adjacent, or just adjacent in the parameter vector, or both.

“in a l1-norm sense” means that, instead of smoothness, total variation imposes **sharpness** on the solution. This means that most parameters will be close to each other in value, but a few will be far apart, allowing discontinuities to appear.

This class provides a template for total variation classes of a specific dimension.

> **Warning: DON’T USE THIS CLASS DIRECTLY!** Instead, use the TotalVariation*D classes.

This regularizing function has the form (Martins et al., 2011)

$$\theta(\bar{p}) = \sum_{k=1}^{L} |v_k|$$

where v_k is the kth element of vector $\bar{v}$

$$\bar{v} = \bar{\bar{R}}\bar{p}$$

Function $\theta(\bar{p})$ is not differentiable when v_k approaches zero. We can substitute it with a more friendly version (Martins et al., 2011)

$$\theta_\beta(\bar{p}) = \sum_{k=1}^{L} \sqrt{v_k^2 + \beta}$$

β should be small and controls how close this function is to $\theta(\bar{p})$. The larger the value of β is, the closer θ_β is to the smoothness regularization.

The gradient and Hessian matrix are, respectively (Martins et al., 2011):

$$\bar{g}(\bar{p}) = \bar{\bar{R}}^T \bar{q}(\bar{p})$$

and

$$\bar{\bar{H}}(\bar{p}) = \bar{\bar{R}}^T \bar{\bar{Q}}(\bar{p}) \bar{\bar{R}}$$

where $\bar{\bar{R}}$ is a finite difference matrix, and $\bar{q}$ and $\bar{\bar{Q}}$ are

$$\bar{q}(\bar{p}) = \begin{bmatrix} \frac{v_1}{\sqrt{v_1^2+\beta}} \\ \frac{v_2}{\sqrt{v_2^2+\beta}} \\ \vdots \\ \frac{v_L}{\sqrt{v_L^2+\beta}} \end{bmatrix}$$

and

$$\bar{\bar{Q}}(\bar{p}) = \begin{bmatrix} \frac{\beta}{(v_1^2+\beta)^{\frac{3}{2}}} & 0 & \dots & 0 \\ 0 & \frac{\beta}{(v_2^2+\beta)^{\frac{3}{2}}} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{\beta}{(v_L^2+\beta)^{\frac{3}{2}}} \end{bmatrix}$$

Parameters:

- **mu** [float] The regularizing parameter. A positve scalar that controls the tradeoff between data fitting and regularization. I.e., how much sharpness to apply.
- **nparams** [int] Number of parameters in the inversion
- **sparse** [True or False] Wether or not to use sparce matrices from scipy

References:

Martins, C. M., W. A. Lima, V. C. F. Barbosa, and J. B. C. Silva, 2011, Total variation regularization for depth-to-basement estimate: Part 1 - mathematical details and applications: Geophysics, 76, I1-I12.

class `fatiando.inversion.regularizer.`**`TotalVariation1D`**(*mu*, *nparams*, *beta=1e-10*, *sparse=False*)

Bases: `fatiando.inversion.regularizer.TotalVariation`

Total variation regularization for 1D problems.

Imposes that adjacent parameters have values as close as possible to each other, in a **l1-norm** sense. By adjacent, I mean next to each other in the parameter vector, e.g., p[2] and p[3].

In other words, total variation imposes sharpness on the solution. See `TotalVariation` for more explanation on this. See `Smoothness1D` for details on matrix $\bar{\bar{R}}$.

Parameters:

- **mu** [float] The regularizing parameter. A positve scalar that controls the tradeoff between data fitting and regularization. I.e., how much sharpness to apply.
- **nparams** [int] Number of parameters in the inversion

- **sparse** [True or False] Wether or not to use sparce matrices from scipy

class `fatiando.inversion.regularizer.`**`TotalVariation2D`**(*mu*, *shape*, *beta=1e-10*, *sparse=False*)

Bases: `fatiando.inversion.regularizer.TotalVariation`

Total variation regularization for 1D problems.

Imposes that spacially adjacent parameters have values as close as possible to each other, in a **l1-norm** sense. By spacially adjacent, I mean that I assume the parameters are originaly placed on a grid and the grid is then flattened to make the parameter vector.

In other words, total variation imposes sharpness on the solution. See `TotalVariation` for more explanation on this. See `Smoothness2D` for details on matrix $\bar{\bar{R}}$.

Parameters:

- **mu** [float] The regularizing parameter. A positve scalar that controls the tradeoff between data fitting and regularization. I.e., how much smoothness to apply.
- **shape** [tuple = (ny, nx)] (ny, nx): number of parameters in each direction of the grid
- **sparse** [True or False] Wether or not to use sparce matrices from scipy

`fatiando.inversion.regularizer.`**`fdmatrix1d`**(*n*)

Make a finite difference matrix for a 1D problem.

See `Smoothness1D` for more explanation on this matrix.

Parameters:

- **n** [int] Number of elements in the parameter vector

Returns:

- **fdmat** [array] The finite difference matrix for of a 1D problem with n parameters

`fatiando.inversion.regularizer.`**`fdmatrix2d`**(*shape*, *sparse=False*)

Make a finite difference matrix for a 2D problem.

See `Smoothness2D` for more explanation on this matrix.

The diagonal derivatives are not taken into account.

Parameters:

- **shape** [tuple = (ny, nx)] (ny, nx): number of parameters in each direction of the grid representing the interpretative model.
- **sparse** [True or False] If True, will use *scipy.sparse.csr_matrix* instead of normal numpy arrays

Returns:

- **fdmat** [aray] The finite difference matrix for of a 2D problem

5.12.5 Base data module class (`fatiando.inversion.datamodule`)

Base DataModule class with the format expected by all inverse problem solvers.

See the docs for the `fatiando.inversion` package for more information on the role of the data modules.

---

class `fatiando.inversion.datamodule.`**`DataModule`**(*data*)

Bases: `object`

A generic data module.

Use this class as a skeleton for building custom data modules for a specific geophysical data set and interpretative model, like gravity anomaly for right rectangular prism models, travel time residuals for epicenter calculation, etc.

Data modules are how each inverse problem solver knows how to calculate things like:

- Predicted data
- Data-misfit function
- Gradient of the data-misfit function
- Hessian of the data-misfit function

Not all solvers use all of the above. For examples, heuristic solvers don't require gradient and hessian calculations.

This class has templates for all of these methods so that all solvers know what to expect.

Normally, all data modules should store the value of the latest residual vector calculated.

Constructor parameters common to all methods:

- **data** [array] The observed data.

`get_misfit` (*residuals*)

Returns the value of the data-misfit function for a given residual vector

Parameters:

- **residuals** [array] The residual vector

Returns:

- **misfit** [float] Scalar value of the data-misfit

`get_predicted` (*p*)

Returns the predicted data vector for a given parameter vector.

Parameters:

- **p** [array] The parameter vector

Returns:

- **pred** [array] The calculated predicted data vector

`sum_gradient` (*gradient*, *p=None*, *residuals=None*)

Sums the gradient vector of this data module to *gradient* and returns the result.

Parameters:

- **gradient** [array] The old gradient vector
- **p** [array] The parameter vector
- **residuals** [array] The residuals evaluated for parameter vector *p*

Note: Solvers for linear problems will use `p = None` and `residuals = None` so that the class knows how to calculate gradients more efficiently for these cases.

Returns:

- **new_gradient** [array] The new gradient vector

`sum_hessian` (*hessian*, *p=None*)

Sums the Hessian matrix of this data module to *hessian* and returns the result.

Parameters:

- **hessian** [array] 2D array with the old Hessian matrix
- **p** [array] The parameter vector

Note: Solvers for linear problems will use `p = None` so that the class knows how to calculate gradients more efficiently for these cases.

Returns:

- **new_hessian** [array] 2D array with the new Hessian matrix

CHAPTER

SIX

COOKBOOK

Here you'll find some recipes for doing common tasks using Fatiando, like generating synthetic data, running inversions, and plotting things.

Tip: You can download `.py` files of the recipes by clicking on the `[source code]` link bellow the recipe title.

Tip: On IPython (http://www.ipython.org), try using the `%load` magic to load a recipe from the internet directly into your session:

```
%load http://fatiando.readthedocs.org/en/latest/_static/cookbook/gravmag_mag_polyprism.py
```

6.1 Geothermal: Forward and inverse modeling of an abrupt change in temperature measured in a well

```python
"""
Geothermal: Forward and inverse modeling of an abrupt change in temperature
measured in a well
"""
import numpy
from fatiando import logger, utils
from fatiando.geothermal import climsig
from fatiando.vis import mpl

log = logger.get()
log.info(logger.header())
log.info(__doc__)

# Generating synthetic data
amp = 3
age = 54
zp = numpy.arange(0, 100, 1)
temp, error = utils.contaminate(climsig.abrupt(amp, age, zp), 0.02,
    percent=True, return_stddev=True)

# Preparing for the inversion
p, residuals = climsig.iabrupt(temp, zp)
est_amp, est_age = p

mpl.figure(figsize=(12,5))
mpl.subplot(1, 2, 1)
mpl.title("Climate signal (abrupt)")
mpl.plot(temp, zp, 'ok', label='Observed')
mpl.plot(temp - residuals, zp, '--r', linewidth=3, label='Predicted')
mpl.legend(loc='lower right', numpoints=1)
mpl.xlabel("Temperature (C)")
mpl.ylabel("Z")
mpl.ylim(100, 0)
ax = mpl.subplot(1, 2, 2)
ax2 = mpl.twinx()
mpl.title("Age and amplitude")
width = 0.3
ax.bar([1 - width], [age], width, color='b', label="True")
ax.bar([1], [est_age], width, color='r', label="Estimate")
ax2.bar([2 - width], [amp], width, color='b')
ax2.bar([2], [est_amp], width, color='r')
ax.legend(loc='upper center', numpoints=1)
ax.set_ylabel("Age (years)")
ax2.set_ylabel("Amplitude (C)")
ax.set_xticks([1, 2])
ax.set_xticklabels(['Age', 'Amplitude'])
ax.set_ylim(0, 70)
ax2.set_ylim(0, 4)
mpl.show()
```

6.2 Geothermal: Forward and inverse modeling of a linear change in temperature measured in a well

```
"""
Geothermal: Forward and inverse modeling of a linear change in temperature
measured in a well
"""
import numpy
from fatiando import logger, utils
from fatiando.geothermal import climsig
from fatiando.vis import mpl

log = logger.get()
log.info(logger.header())
log.info(__doc__)

# Generating synthetic data
amp = 5.43
age = 78.2
zp = numpy.arange(0, 100, 1)
temp, error = utils.contaminate(climsig.linear(amp, age, zp),
    0.02, percent=True, return_stddev=True)

# Preparing for the inversion
p, residuals = climsig.ilinear(temp, zp)
est_amp, est_age = p

mpl.figure(figsize=(12,5))
mpl.subplot(1, 2, 1)
mpl.title("Climate signal (linear)")
mpl.plot(temp, zp, 'ok', label='Observed')
mpl.plot(temp - residuals, zp, '--r', linewidth=3, label='Predicted')
mpl.legend(loc='lower right', numpoints=1)
mpl.xlabel("Temperature (C)")
mpl.ylabel("Z")
mpl.ylim(100, 0)
ax = mpl.subplot(1, 2, 2)
ax2 = mpl.twinx()
```

```
mpl.title("Age and amplitude")
width = 0.3
ax.bar([1 - width], [age], width, color='b', label="True")
ax.bar([1], [est_age], width, color='r', label="Estimate")
ax2.bar([2 - width], [amp], width, color='b')
ax2.bar([2], [est_amp], width, color='r')
ax.legend(loc='upper center', numpoints=1)
ax.set_ylabel("Age (years)")
ax2.set_ylabel("Amplitude (C)")
ax.set_xticks([1, 2])
ax.set_xticklabels(['Age', 'Amplitude'])
ax.set_ylim(0, 100)
ax2.set_ylim(0, 7)
mpl.show()
```

6.3 Geothermal: Climate signal: What happens when assuming a climate change is linear, when in fact it was abrupt?

```
"""
Geothermal: Climate signal: What happens when assuming a climate change is
linear, when in fact it was abrupt?
"""
import numpy
from fatiando import logger, utils
from fatiando.geothermal import climsig
from fatiando.vis import mpl

log = logger.get()
log.info(logger.header())
log.info(__doc__)

# Generating synthetic data using an ABRUPT model
amp = 3
age = 54
zp = numpy.arange(0, 100, 1)
temp, error = utils.contaminate(climsig.abrupt(amp, age, zp),
    0.02, percent=True, return_stddev=True)

# Preparing for the inversion assuming that the change was LINEAR
p, residuals = climsig.ilinear(temp, zp)
est_amp, est_age = p

mpl.figure(figsize=(12,5))
mpl.subplot(1, 2, 1)
mpl.title("Climate signal\n(true is abrupt but inverted using linear)")
mpl.plot(temp, zp, 'ok', label='Observed')
mpl.plot(temp - residuals, zp, '--r', linewidth=3, label='Predicted')
mpl.legend(loc='lower right', numpoints=1)
mpl.xlabel("Temperature (C)")
mpl.ylabel("Z")
mpl.ylim(100, 0)
ax = mpl.subplot(1, 2, 2)
ax2 = mpl.twinx()
mpl.title("Age and amplitude")
width = 0.3
ax.bar([1 - width], [age], width, color='b', label="True")
ax.bar([1], [est_age], width, color='r', label="Estimate")
ax2.bar([2 - width], [amp], width, color='b')
ax2.bar([2], [est_amp], width, color='r')
ax.legend(loc='upper center', numpoints=1)
```

```python
ax.set_ylabel("Age (years)")
ax2.set_ylabel("Amplitude (C)")
ax.set_xticks([1, 2])
ax.set_xticklabels(['Age', 'Amplitude'])
ax.set_ylim(0, 150)
ax2.set_ylim(0, 4)
mpl.show()
```

6.4 GravMag: Interactive 2D forward modeling with polygons

```python
"""
GravMag: Interactive 2D forward modeling with polygons
"""
import numpy
from fatiando.gui.simple import Moulder

area = (0, 100000, 0, 5000)
xp = numpy.arange(0, 100000, 1000)
zp = numpy.zeros_like(xp)
app = Moulder(area, xp, zp)
app.run()
```

6.5 GravMag: 2D forward modeling with polygons

```python
"""
GravMag: 2D forward modeling with polygons
"""
import numpy
from fatiando import logger, utils, mesher, gravmag, inversion
from fatiando.vis import mpl

log = logger.get()
log.info(logger.header())
log.info(__doc__)

# Notice that the last two number are switched.
# This way, the z axis in the plots points down.
area = (-5000, 5000, 5000, 0)
axes = mpl.figure().gca()
mpl.xlabel("X")
mpl.ylabel("Z")
mpl.axis('scaled')
polygons = [mesher.Polygon(mpl.draw_polygon(area, axes),
                              {'density':500})]
xp = numpy.arange(-4500, 4500, 100)
zp = numpy.zeros_like(xp)
gz = gravmag.talwani.gz(xp, zp, polygons)

mpl.figure()
mpl.axis('scaled')
mpl.subplot(2,1,1)
mpl.title(r"Gravity anomaly produced by the model")
mpl.plot(xp, gz, '-k', linewidth=2)
mpl.ylabel("mGal")
mpl.xlim(-5000, 5000)
mpl.subplot(2,1,2)
mpl.polygon(polygons[0], 'o-k', linewidth=2, fill='k', alpha=0.5)
mpl.xlabel("X")
mpl.ylabel("Z")
mpl.set_area(area)
mpl.show()
```

6.6 GravMag: Simple gravity inversion for the relief of a 2D trapezoidal basin

```
"""
GravMag: Simple gravity inversion for the relief of a 2D trapezoidal basin
"""
import numpy
from fatiando import logger, utils, mesher, gravmag, inversion
from fatiando.vis import mpl

log = logger.get()
log.info(logger.header())
log.info(__doc__)

log.info("Generating synthetic data")
verts = [(10000, 1.), (90000, 1.), (90000, 7000), (10000, 3330)]
model = mesher.Polygon(verts, {'density':-100})
xp = numpy.arange(0., 100000., 1000.)
zp = numpy.zeros_like(xp)
gz = utils.contaminate(gravmag.talwani.gz(xp, zp, [model]), 0.5)

log.info("Preparing for the inversion")
solver = inversion.gradient.levmarq(initial=(9000, 500))
estimate, residuals = gravmag.basin2d.trapezoidal(xp, zp, gz, verts[0:2], -100,
    solver)

mpl.figure()
mpl.subplot(2, 1, 1)
mpl.title("Gravity anomaly")
mpl.plot(xp, gz, 'ok', label='Observed')
mpl.plot(xp, gz - residuals, '-r', linewidth=2, label='Predicted')
mpl.legend(loc='lower left', numpoints=1)
mpl.ylabel("mGal")
mpl.xlim(0, 100000)
mpl.subplot(2, 1, 2)
mpl.polygon(estimate, 'o-r', linewidth=2, fill='r', alpha=0.3,
            label='Estimated')
mpl.polygon(model, '--k', linewidth=2, label='True')
mpl.legend(loc='lower left', numpoints=1)
mpl.xlabel("X")
mpl.ylabel("Z")
mpl.set_area((0, 100000, 10000, -500))
mpl.show()
```

6.7 GravMag: Interactive 2D forward gravity modeling of a trapezoidal basin

```
"""
GravMag: Interactive 2D forward gravity modeling of a trapezoidal basin
"""
import numpy
from fatiando.gui.simple import BasinTrap

```

```python
area = (0, 100000, 0, 5000)
xp = numpy.arange(0, 100000, 1000)
zp = numpy.zeros_like(xp)
nodes = [[20000, 1], [80000, 1]]
app = BasinTrap(area, nodes, xp, zp)
app.run()
```

6.8 GravMag: Simple gravity inversion for the relief of a 2D triangular basin

```python
"""
GravMag: Simple gravity inversion for the relief of a 2D triangular basin
"""
import numpy
from fatiando import logger, utils, mesher, gravmag, inversion
from fatiando.vis import mpl

log = logger.get()
log.info(logger.header())
log.info(__doc__)

log.info("Generating synthetic data")
verts = [(10000, 1.), (90000, 1.), (80000, 5000)]
model = mesher.Polygon(verts, {'density':-100})
xp = numpy.arange(0., 100000., 1000.)
zp = numpy.zeros_like(xp)
gz = utils.contaminate(gravmag.talwani.gz(xp, zp, [model]), 1)

log.info("Preparing for the inversion")
solver = inversion.gradient.levmarq(initial=(10000, 1000))
estimate, residuals = gravmag.basin2d.triangular(xp, zp, gz, verts[0:2], -100,
    solver)

mpl.figure()
mpl.subplot(2, 1, 1)
mpl.title("Gravity anomaly")
mpl.plot(xp, gz, 'ok', label='Observed')
mpl.plot(xp, gz - residuals, '-r', linewidth=2, label='Predicted')
mpl.legend(loc='lower left')
mpl.ylabel("mGal")
mpl.xlim(0, 100000)
mpl.subplot(2, 1, 2)
mpl.polygon(estimate, 'o-r', linewidth=2, fill='r', alpha=0.3,
                label='Estimated')
mpl.polygon(model, '--k', linewidth=2, label='True')
mpl.legend(loc='lower left', numpoints=1)
mpl.xlabel("X")
mpl.ylabel("Z")
mpl.set_area((0, 100000, 10000, -500))
mpl.show()
```

6.9 GravMag: Interactive 2D forward gravity modeling of a triangular basin

```python
"""
GravMag: Interactive 2D forward gravity modeling of a triangular basin
"""
```

```python
import numpy
from fatiando.gui.simple import BasinTri

area = (0, 100000, 0, 5000)
xp = numpy.arange(0, 100000, 1000)
zp = numpy.zeros_like(xp)
nodes = [[20000, 1], [80000, 1]]
app = BasinTri(area, nodes, xp, zp)
app.run()
```

6.10 GravMag: Classic 3D Euler deconvolution of magnetic data (single window)

```python
"""
GravMag: Classic 3D Euler deconvolution of magnetic data (single window)
"""
from fatiando import logger, mesher, gridder, utils, gravmag
from fatiando.vis import mpl, myv

log = logger.get()
log.info(logger.header())

# Make a model
bounds = [-5000, 5000, -5000, 5000, 0, 5000]
model = [mesher.Prism(-1500, -500, -500, 500, 1000, 2000, {'magnetization':2})]
# Generate some data from the model
shape = (200, 200)
area = bounds[0:4]
xp, yp, zp = gridder.regular(area, shape, z=-1)
# Add a constant baselevel
baselevel = 10
# Convert from nanoTesla to Tesla because euler and derivatives require things
# in SI
tf = (utils.nt2si(gravmag.prism.tf(xp, yp, zp, model, inc=-45, dec=0))
      + baselevel)
# Calculate the derivatives using FFT
xderiv = gravmag.fourier.derivx(xp, yp, tf, shape)
yderiv = gravmag.fourier.derivy(xp, yp, tf, shape)
zderiv = gravmag.fourier.derivz(xp, yp, tf, shape)

mpl.figure()
titles = ['Total field', 'x derivative', 'y derivative', 'z derivative']
for i, f in enumerate([tf, xderiv, yderiv, zderiv]):
    mpl.subplot(2, 2, i + 1)
    mpl.title(titles[i])
    mpl.axis('scaled')
    mpl.contourf(yp, xp, f, shape, 50)
    mpl.colorbar()
    mpl.m2km()
mpl.show()

# Run the euler deconvolution on a single window
# Structural index is 3
results = gravmag.euler.classic(xp, yp, zp, tf, xderiv, yderiv, zderiv, 3)
print "Base level used: %g" % (baselevel)
print "Estimated base level: %g" % (results['baselevel'])

myv.figure()
myv.points([results['point']], size=300.)
myv.prisms(model, prop='magnetization', opacity=0.5)
```

```
axes = myv.axes(myv.outline(extent=bounds))
myv.wall_bottom(axes.axes.bounds, opacity=0.2)
myv.wall_north(axes.axes.bounds)
myv.show()
```

6.11 GravMag: Classic 3D Euler deconvolution of magnetic data using an expanding window

```
"""
GravMag: Classic 3D Euler deconvolution of magnetic data using an
expanding window
"""
from fatiando import logger, mesher, gridder, utils, gravmag
from fatiando.vis import mpl, myv

log = logger.get()
log.info(logger.header())

# Make a model
bounds = [-5000, 5000, -5000, 5000, 0, 5000]
model = [
    mesher.Prism(-1500, -500, -1500, -500, 1000, 2000, {'magnetization':2}),
    mesher.Prism(500, 1500, 500, 2000, 1000, 2000, {'magnetization':2})]
# Generate some data from the model
shape = (100, 100)
area = bounds[0:4]
xp, yp, zp = gridder.regular(area, shape, z=-1)
# Add a constant baselevel
baselevel = 10
# Convert from nanoTesla to Tesla because euler and derivatives require things
# in SI
tf = (utils.nt2si(gravmag.prism.tf(xp, yp, zp, model, inc=-45, dec=0))
      + baselevel)
# Calculate the derivatives using FFT
xderiv = gravmag.fourier.derivx(xp, yp, tf, shape)
yderiv = gravmag.fourier.derivy(xp, yp, tf, shape)
zderiv = gravmag.fourier.derivz(xp, yp, tf, shape)

mpl.figure()
titles = ['Total field', 'x derivative', 'y derivative', 'z derivative']
for i, f in enumerate([tf, xderiv, yderiv, zderiv]):
    mpl.subplot(2, 2, i + 1)
    mpl.title(titles[i])
    mpl.axis('scaled')
    mpl.contourf(yp, xp, f, shape, 50)
    mpl.colorbar()
    mpl.m2km()
mpl.show()

# Pick the centers of the expanding windows
# The number of final solutions will be the number of points picked
mpl.figure()
mpl.suptitle('Pick the centers of the expanding windows')
mpl.axis('scaled')
mpl.contourf(yp, xp, tf, shape, 50)
mpl.colorbar()
centers = mpl.pick_points(area, mpl.gca(), xy2ne=True)

# Run the euler deconvolution on an expanding window
# Structural index is 3
```

```
index = 3
results = []
for center in centers:
    results.append(
        gravmag.euler.expanding_window(xp, yp, zp, tf, xderiv, yderiv, zderiv,
            index, gravmag.euler.classic, center, 500, 5000))
    print "Base level used: %g" % (baselevel)
    print "Estimated base level: %g" % (results[-1]['baselevel'])
    print "Estimated source location: %s" % (str(results[-1]['point']))

myv.figure()
myv.points([r['point'] for r in results], size=300.)
myv.prisms(model, opacity=0.5)
axes = myv.axes(myv.outline(bounds), ranges=[b*0.001 for b in bounds])
myv.wall_bottom(bounds)
myv.wall_north(bounds)
myv.show()
```

6.12 GravMag: Classic 3D Euler deconvolution of noisy magnetic data (single window)

```
"""
GravMag: Classic 3D Euler deconvolution of noisy magnetic data (single window)
"""
from fatiando import logger, mesher, gridder, utils, gravmag
from fatiando.vis import mpl, myv

log = logger.get()
log.info(logger.header())

# Make a model
bounds = [-5000, 5000, -5000, 5000, 0, 5000]
model = [
    mesher.Prism(-500, 500, -500, 500, 1000, 2000, {'magnetization':2})]
# Generate some data from the model
shape = (100, 100)
area = bounds[0:4]
xp, yp, zp = gridder.regular(area, shape, z=-1)
# Add a constant baselevel
baselevel = 10
# Convert from nanoTesla to Tesla because euler and derivatives require things
# in SI
tf = (utils.contaminate(
        utils.nt2si(gravmag.prism.tf(xp, yp, zp, model, inc=-45, dec=0)),
        0.005, percent=True)
      + baselevel)
# Calculate the derivatives using FFT
xderiv = gravmag.fourier.derivx(xp, yp, tf, shape)
yderiv = gravmag.fourier.derivy(xp, yp, tf, shape)
zderiv = gravmag.fourier.derivz(xp, yp, tf, shape)

mpl.figure()
titles = ['Total field', 'x derivative', 'y derivative', 'z derivative']
for i, f in enumerate([tf, xderiv, yderiv, zderiv]):
    mpl.subplot(2, 2, i + 1)
    mpl.title(titles[i])
    mpl.axis('scaled')
    mpl.contourf(yp, xp, f, shape, 50)
    mpl.colorbar()
    mpl.m2km()
```

```python
mpl.show()

# Run the euler deconvolution on a single window
# Structural index is 3
results = gravmag.euler.classic(xp, yp, zp, tf, xderiv, yderiv, zderiv, 3)
print "Base level used: %g" % (baselevel)
print "Estimated base level: %g" % (results['baselevel'])

myv.figure()
myv.points([results['point']], size=300.)
myv.prisms(model, prop='magnetization', opacity=0.5)
axes = myv.axes(myv.outline(extent=bounds))
myv.wall_bottom(axes.axes.bounds, opacity=0.2)
myv.wall_north(axes.axes.bounds)
myv.show()
```

6.13 GravMag: Calculating the derivatives of the gravity anomaly using FFT

```python
"""
GravMag: Calculating the derivatives of the gravity anomaly using FFT
"""
from fatiando import logger, mesher, gridder, utils, gravmag
from fatiando.vis import mpl

log = logger.get()
log.info(logger.header())
log.info(__doc__)

log.info("Generating synthetic data")
prisms = [mesher.Prism(-1000,1000,-1000,1000,0,2000,{'density':100})]
area = (-5000, 5000, -5000, 5000)
shape = (51, 51)
z0 = -500
xp, yp, zp = gridder.regular(area, shape, z=z0)
gz = utils.contaminate(gravmag.prism.gz(xp, yp, zp, prisms), 0.001)

log.info("Calculating the x-derivative")
# Need to convert gz to SI units so that the result can be converted to Eotvos
gxz = utils.si2eotvos(
    gravmag.fourier.derivx(xp, yp, utils.mgal2si(gz), shape))
gyz = utils.si2eotvos(
    gravmag.fourier.derivy(xp, yp, utils.mgal2si(gz), shape))
gzz = utils.si2eotvos(
    gravmag.fourier.derivz(xp, yp, utils.mgal2si(gz), shape))

log.info("Computing true values of the derivative")
gxz_true = gravmag.prism.gxz(xp, yp, zp, prisms)
gyz_true = gravmag.prism.gyz(xp, yp, zp, prisms)
gzz_true = gravmag.prism.gzz(xp, yp, zp, prisms)

log.info("Plotting")
mpl.figure()
mpl.title("Original gravity anomaly")
mpl.axis('scaled')
mpl.contourf(xp, yp, gz, shape, 15)
mpl.colorbar(shrink=0.7)
mpl.m2km()

mpl.figure(figsize=(14,10))
mpl.subplots_adjust(top=0.95, left=0.05, right=0.95)
mpl.subplot(2, 3, 1)
mpl.title("x deriv (contour) + true (color map)")
mpl.axis('scaled')
levels = mpl.contourf(xp, yp, gxz_true, shape, 12)
mpl.colorbar(shrink=0.7)
mpl.contour(xp, yp, gxz, shape, 12, color='k')
mpl.m2km()
mpl.subplot(2, 3, 2)
mpl.title("y deriv (contour) + true (color map)")
mpl.axis('scaled')
levels = mpl.contourf(xp, yp, gyz_true, shape, 12)
mpl.colorbar(shrink=0.7)
mpl.contour(xp, yp, gyz, shape, 12, color='k')
mpl.m2km()
mpl.subplot(2, 3, 3)
mpl.title("z deriv (contour) + true (color map)")
mpl.axis('scaled')
levels = mpl.contourf(xp, yp, gzz_true, shape, 8)
mpl.colorbar(shrink=0.7)
mpl.contour(xp, yp, gzz, shape, levels, color='k')
mpl.m2km()
mpl.subplot(2, 3, 4)
mpl.title("Difference x deriv")
mpl.axis('scaled')
mpl.pcolor(xp, yp, (gxz_true - gxz), shape)
mpl.colorbar(shrink=0.7)
mpl.m2km()
mpl.subplot(2, 3, 5)
mpl.title("Difference y deriv")
mpl.axis('scaled')
mpl.pcolor(xp, yp, (gyz_true - gyz), shape)
mpl.colorbar(shrink=0.7)
mpl.m2km()
mpl.subplot(2, 3, 6)
mpl.title("Difference z deriv")
mpl.axis('scaled')
mpl.pcolor(xp, yp, (gzz_true - gzz), shape)
mpl.colorbar(shrink=0.7)
mpl.m2km()
mpl.show()
```

6.14 GravMag: Forward gravity modeling using a stack of 3D polygonal prisms

```
"""
GravMag: Forward gravity modeling using a stack of 3D polygonal prisms
"""
from fatiando import logger, mesher, gridder, gravmag
from fatiando.vis import mpl, myv

log = logger.get()
log.info(logger.header())
log.info(__doc__)

log.info("Draw the polygons one by one:")
bounds = [-10000, 10000, -10000, 10000, 0, 5000]
area = bounds[:4]
depths = [0, 1000, 2000, 3000, 4000]
prisms = []
```

```python
for i in range(1, len(depths)):
    axes = mpl.figure().gca()
    mpl.axis('scaled')
    for p in prisms:
        mpl.polygon(p, '.-k', xy2ne=True)
    prisms.append(
        mesher.PolygonalPrism(
            mpl.draw_polygon(area, axes, xy2ne=True),
            depths[i - 1], depths[i], {'density':500}))
# Calculate the effect
shape = (100, 100)
xp, yp, zp = gridder.regular(area, shape, z=-1)
gz = gravmag.polyprism.gz(xp, yp, zp, prisms)
# and plot it
mpl.figure()
mpl.axis('scaled')
mpl.title("gz produced by prism model (mGal)")
mpl.contourf(yp, xp, gz, shape, 20)
mpl.colorbar()
for p in prisms:
    mpl.polygon(p, '.-k', xy2ne=True)
mpl.set_area(area)
mpl.show()
# Show the prisms
myv.figure()
myv.polyprisms(prisms, 'density')
myv.axes(myv.outline(bounds), ranges=[i*0.001 for i in bounds])
myv.show()
```

6.15 GravMag: Forward modeling of the gravitational potential and its derivatives using 3D prisms

```python
"""
GravMag: Forward modeling of the gravitational potential and its derivatives
using 3D prisms
"""
from fatiando import logger, mesher, gridder, gravmag
from fatiando.vis import mpl, myv

log = logger.get()
log.info(logger.header())
log.info(__doc__)

prisms = [mesher.Prism(-4000,-3000,-4000,-3000,0,2000,{'density':1000}),
          mesher.Prism(-1000,1000,-1000,1000,0,2000,{'density':-900}),
          mesher.Prism(2000,4000,3000,4000,0,2000,{'density':1300})]
shape = (100,100)
xp, yp, zp = gridder.regular((-5000, 5000, -5000, 5000), shape, z=-150)
log.info("Calculating fileds...")
fields = [gravmag.prism.potential(xp, yp, zp, prisms),
          gravmag.prism.gx(xp, yp, zp, prisms),
          gravmag.prism.gy(xp, yp, zp, prisms),
          gravmag.prism.gz(xp, yp, zp, prisms),
          gravmag.prism.gxx(xp, yp, zp, prisms),
          gravmag.prism.gxy(xp, yp, zp, prisms),
          gravmag.prism.gxz(xp, yp, zp, prisms),
          gravmag.prism.gyy(xp, yp, zp, prisms),
          gravmag.prism.gyz(xp, yp, zp, prisms),
          gravmag.prism.gzz(xp, yp, zp, prisms)]
log.info("Plotting...")
```

```
titles = ['potential', 'gx', 'gy', 'gz',
          'gxx', 'gxy', 'gxz', 'gyy', 'gyz', 'gzz']
mpl.figure(figsize=(8, 9))
mpl.subplots_adjust(left=0.03, right=0.95, bottom=0.05, top=0.92, hspace=0.3)
mpl.suptitle("Potential fields produced by a 3 prism model")
for i, field in enumerate(fields):
    mpl.subplot(4, 3, i + 3)
    mpl.axis('scaled')
    mpl.title(titles[i])
    levels = mpl.contourf(yp*0.001, xp*0.001, field, shape, 15)
    cb = mpl.colorbar()
    mpl.contour(yp*0.001, xp*0.001, field, shape, levels, clabel=False, linewidth=0.1)
mpl.show()

myv.figure()
myv.prisms(prisms, prop='density')
axes = myv.axes(myv.outline())
myv.wall_bottom(axes.axes.bounds, opacity=0.2)
myv.wall_north(axes.axes.bounds)
myv.show()
```

6.16 GravMag: Forward modeling of the gravity anomaly using spheres (calculate on random points)

```
"""
GravMag: Forward modeling of the gravity anomaly using spheres (calculate on
random points)
"""
from fatiando import logger, mesher, gridder, utils, gravmag
from fatiando.vis import mpl

log = logger.get()
log.info(logger.header())
log.info(__doc__)

spheres = [mesher.Sphere(0, 0, -2000, 1000, {'density':1000})]
# Create a set of points at 100m height
area = (-5000, 5000, -5000, 5000)
xp, yp, zp = gridder.scatter(area, 500, z=-100)
# Calculate the anomaly
gz = utils.contaminate(gravmag.sphere.gz(xp, yp, zp, spheres), 0.1)
# Plot
shape = (100, 100)
mpl.figure()
mpl.title("gz (mGal)")
mpl.axis('scaled')
mpl.plot(yp*0.001, xp*0.001, '.k')
mpl.contourf(yp*0.001, xp*0.001, gz, shape, 15, interp=True)
mpl.colorbar()
mpl.xlabel('East y (km)')
mpl.ylabel('North x (km)')
mpl.show()
```

6.17 GravMag: Forward modeling of the gravitational potential and its derivatives using tesseroids

```
"""
GravMag: Forward modeling of the gravitational potential and its derivatives
using tesseroids
"""
import time
from fatiando import gravmag, gridder, logger, utils
from fatiando.mesher import Tesseroid
from fatiando.vis import mpl, myv

log = logger.get()
log.info(logger.header())

model = [Tesseroid(-60, -55, -30, -27, 0, -500000, props={'density':200}),
         Tesseroid(-66, -62, -18, -12, 0, -300000, props={'density':-500})]
# Show the model before calculating
scene = myv.figure(zdown=False)
myv.tesseroids(model, 'density')
myv.continents(linewidth=2)
myv.earth(opacity=0.8)
myv.meridians(range(0, 360, 45), opacity=0.2)
myv.parallels(range(-90, 90, 45), opacity=0.2)
scene.scene.camera.position = [23175275.131412581, -16937347.013663091,
    -4924328.2822419703]
scene.scene.camera.focal_point = [0.0, 0.0, 0.0]
scene.scene.camera.view_angle = 30.0
scene.scene.camera.view_up = [0.083030001958377356, -0.17178720527713925,
    0.98162883763562181]
scene.scene.camera.clipping_range = [9229054.5133903362, 54238225.321054712]
scene.scene.camera.compute_view_plane_normal()
scene.scene.render()
myv.show()

# Create the computation grid
area = (-80, -30, -40, 10)
shape = (50, 50)
lons, lats, heights = gridder.regular(area, shape, z=250000)

log.info('Calculating...')
start = time.time()
fields = [
    gravmag.tesseroid.potential(lons, lats, heights, model),
    gravmag.tesseroid.gx(lons, lats, heights, model),
    gravmag.tesseroid.gy(lons, lats, heights, model),
    gravmag.tesseroid.gz(lons, lats, heights, model),
    gravmag.tesseroid.gxx(lons, lats, heights, model),
    gravmag.tesseroid.gxy(lons, lats, heights, model),
    gravmag.tesseroid.gxz(lons, lats, heights, model),
    gravmag.tesseroid.gyy(lons, lats, heights, model),
    gravmag.tesseroid.gyz(lons, lats, heights, model),
    gravmag.tesseroid.gzz(lons, lats, heights, model)]
print "Time it took: %s" % (utils.sec2hms(time.time() - start))

log.info('Plotting...')
titles = ['potential', 'gx', 'gy', 'gz',
          'gxx', 'gxy', 'gxz', 'gyy', 'gyz', 'gzz']
mpl.figure()
bm = mpl.basemap(area, 'ortho')
for i, field in enumerate(fields):
    mpl.subplot(4, 3, i + 3)
    mpl.title(titles[i])
    mpl.contourf(lons, lats, field, shape, 15, basemap=bm)
    bm.drawcoastlines()
    mpl.colorbar()
mpl.show()
```

6.18 GravMag: Forward modeling of the gravity anomaly using tesseroids in parallel using `multiprocessing`

```
"""
GravMag: Forward modeling of the gravity anomaly using tesseroids in parallel
using ``multiprocessing``
"""
import time
from multiprocessing import Pool
from fatiando import gravmag, gridder, logger, utils
from fatiando.mesher import Tesseroid
from fatiando.vis import mpl, myv

log = logger.get()
log.info(logger.header())

# Make a "crust" model with some thinker crust and variable density
marea = (-70, 70, -70, 70)
mshape = (200, 200)
mlons, mlats = gridder.regular(marea, mshape)
dlon, dlat = gridder.spacing(marea, mshape)
depths = (30000 +
    70000*utils.gaussian2d(mlons, mlats, 10, 10, -20, -20) +
    20000*utils.gaussian2d(mlons, mlats, 5, 5, 20, 20))
densities = (2700 +
    500*utils.gaussian2d(mlons, mlats, 40, 40, -20, -20) +
    -300*utils.gaussian2d(mlons, mlats, 20, 20, 20, 20))
model = [
    Tesseroid(lon - 0.5*dlon, lon + 0.5*dlon, lat - 0.5*dlat, lat + 0.5*dlat,
              0, -depth, props={'density':density})
    for lon, lat, depth, density in zip(mlons, mlats, depths, densities)]

# Plot the tesseroid model
myv.figure(zdown=False)
myv.tesseroids(model, 'density')
myv.continents()
myv.earth(opacity=0.7)
myv.show()

# Make the computation grid
area = (-50, 50, -50, 50)
shape = (100, 100)
lons, lats, heights = gridder.regular(area, shape, z=250000)

# Divide the model into nproc slices and calculate them in parallel
log.info('Calculating...')
def calculate(chunk):
    return gravmag.tesseroid.gz(lons, lats, heights, chunk)
start = time.time()
nproc = 8 # Model size must be divisible by nproc
chunksize = len(model)/nproc
pool = Pool(processes=nproc)
gz = sum(pool.map(calculate,
    [model[i*chunksize:(i + 1)*chunksize] for i in xrange(nproc)]))
```

```python
pool.close()
print "Time it took: %s" % (utils.sec2hms(time.time() - start))

log.info('Plotting...')
mpl.figure()
bm = mpl.basemap(area, 'ortho')
bm.bluemarble()
mpl.contourf(lons, lats, gz, shape, 35, basemap=bm)
mpl.colorbar()
mpl.show()
```

6.19 GravMag: 3D gravity inversion by planting anomalous densities using `harvester` (simple example)

```python
"""
GravMag: 3D gravity inversion by planting anomalous densities using
``harvester`` (simple example)
"""
from fatiando import logger, gridder, utils
from fatiando import gravmag as gm
from fatiando.mesher import Prism, PrismMesh, vremove
from fatiando.vis import mpl, myv

log = logger.get()
log.info(logger.header())

# Create a synthetic model
model = [Prism(250, 750, 250, 750, 200, 700, {'density':1000})]
# and generate synthetic data from it
shape = (25, 25)
bounds = [0, 1000, 0, 1000, 0, 1000]
area = bounds[0:4]
xp, yp, zp = gridder.regular(area, shape, z=-1)
noise = 0.1 # 0.1 mGal noise
gz = utils.contaminate(gm.prism.gz(xp, yp, zp, model), noise)
# plot the data
mpl.figure()
mpl.title("Synthetic gravity anomaly (mGal)")
mpl.axis('scaled')
levels = mpl.contourf(yp, xp, gz, shape, 12)
mpl.colorbar()
mpl.xlabel('Horizontal coordinate y (km)')
mpl.ylabel('Horizontal coordinate x (km)')
mpl.m2km()
mpl.show()

# Inversion setup
# Create a mesh
mesh = PrismMesh(bounds, (25, 25, 25))
# Wrap the data so that harvester can use it
data = [gm.harvester.Gz(xp, yp, zp, gz)]
# Make the seed
seeds = gm.harvester.sow([[500, 500, 450, {'density':1000}]], mesh)
# Run the inversioin
estimate, predicted = gm.harvester.harvest(data, seeds, mesh,
    compactness=0.5, threshold=0.0005)

# Put the estimated density values in the mesh
mesh.addprop('density', estimate['density'])

```

```
# Plot the adjustment
mpl.figure()
mpl.title("True: color | Inversion: contour")
mpl.axis('scaled')
levels = mpl.contourf(yp, xp, gz, shape, 12)
mpl.colorbar()
mpl.contour(yp, xp, predicted[0], shape, levels, color='k')
mpl.xlabel('Horizontal coordinate y (km)')
mpl.ylabel('Horizontal coordinate x (km)')
mpl.m2km()
residuals = gz - predicted[0]
mpl.figure()
mpl.title('Residuals: mean=%g stddev=%g' % (residuals.mean(), residuals.std()))
mpl.hist(residuals, bins=10)
mpl.xlabel('Residuals (mGal)')
mpl.ylabel('# of')
mpl.show()
# Plot the result
myv.figure()
myv.prisms(model, 'density', style='wireframe')
myv.prisms(vremove(0, 'density', mesh), 'density')
myv.axes(myv.outline(bounds), ranges=[i*0.001 for i in bounds], fmt='%.1f',
    nlabels=6)
myv.wall_bottom(bounds)
myv.wall_north(bounds)
myv.show()
```

6.20 GravMag: 3D gravity inversion by planting anomalous densities using `harvester` (more complex interactive example)

```
"""
GravMag: 3D gravity inversion by planting anomalous densities using
``harvester`` (more complex interactive example)
"""
from fatiando import logger, gridder, utils
import fatiando.gravmag as gm
from fatiando.mesher import PolygonalPrism, PrismMesh, vremove
from fatiando.vis import mpl, myv

log = logger.get()
log.info(logger.header())

# Create a synthetic model
bounds = [-10000, 10000, -10000, 10000, 0, 10000]
vertices = [[-4948.97959184, -6714.64019851],
            [-2448.97959184, -3141.43920596],
            [ 2448.97959184,   312.65508685],
            [ 6938.7755102 ,  5394.54094293],
            [ 4846.93877551,  6228.28784119],
            [ 2653.06122449,  3409.4292804 ],
            [-3520.40816327, -1434.24317618],
            [-6632.65306122, -6079.4044665 ]]
model = [PolygonalPrism(vertices, 1000, 4000, {'density':1000})]
# and generate synthetic data from it
shape = (20, 20)
area = bounds[0:4]
xp, yp, zp = gridder.regular(area, shape, z=-1)
noise = 0.1 # 0.1 mGal noise
gz = utils.contaminate(gm.polyprism.gz(xp, yp, zp, model), noise)

```

```python
# Create a mesh
mesh = PrismMesh(bounds, (25, 50, 50))
# Wrap the data so that harvester can read it
data = [gm.harvester.Gz(xp, yp, zp, gz)]
# Plot the data and pick the location of the seeds
mpl.figure()
mpl.suptitle("Pick the seeds (polygon is the true source)")
mpl.axis('scaled')
levels = mpl.contourf(yp, xp, gz, shape, 12)
mpl.colorbar()
mpl.polygon(model[0], xy2ne=True)
mpl.xlabel('Horizontal coordinate y (km)')
mpl.ylabel('Horizontal coordinate x (km)')
seedx, seedy = mpl.pick_points(area, mpl.gca(), xy2ne=True).T
# Set the right density and depth
locations = [[x, y, 1500, {'density':1000}] for x, y in zip(seedx, seedy)]
mpl.show()
# Make the seed and set the compactness regularizing parameter mu
seeds = gm.harvester.sow(locations, mesh)
# Run the inversion
estimate, predicted = gm.harvester.harvest(data, seeds, mesh,
    compactness=0.05, threshold=0.0005)
# Put the estimated density values in the mesh
mesh.addprop('density', estimate['density'])
# Plot the adjustment and the result
mpl.figure()
mpl.title("True: color | Predicted: contour")
mpl.axis('scaled')
levels = mpl.contourf(yp, xp, gz, shape, 12)
mpl.colorbar()
mpl.contour(yp, xp, predicted[0], shape, levels, color='k')
mpl.xlabel('Horizontal coordinate y (km)')
mpl.ylabel('Horizontal coordinate x (km)')
mpl.m2km()
mpl.show()
# Plot the result
myv.figure()
myv.polyprisms(model, 'density', opacity=0.6, linewidth=5)
myv.prisms(vremove(0, 'density', mesh), 'density')
myv.prisms([mesh[s.i] for s in seeds], 'density')
myv.axes(myv.outline(bounds), ranges=[i*0.001 for i in bounds], fmt='%.1f',
    nlabels=6)
myv.wall_bottom(bounds)
myv.wall_north(bounds)
myv.show()
```

6.21 GravMag: 3D gravity gradient inversion by planting anomalous densities using `harvester` (with non-targeted sources)

```python
"""
GravMag: 3D gravity gradient inversion by planting anomalous densities using
``harvester`` (with non-targeted sources)
"""
from fatiando import logger, gridder, utils
from fatiando import gravmag as gm
from fatiando.mesher import Prism, PrismMesh, vremove
from fatiando.vis import mpl, myv

log = logger.get()
log.info(logger.header())
log.info(__doc__)

# Generate a synthetic model
bounds = [0, 5000, 0, 5000, 0, 1500]
model = [Prism(500, 4500, 3000, 3500, 200, 700, {'density':1200}),
         Prism(3000, 4500, 1800, 2300, 200, 700, {'density':1200}),
         Prism(500, 1500, 500, 1500, 0, 800, {'density':600}),
         Prism(0, 800, 1800, 2300, 0, 200, {'density':600}),
         Prism(4000, 4800, 100, 900, 0, 300, {'density':600}),
         Prism(0, 2000, 4500, 5000, 0, 200, {'density':600}),
         Prism(3000, 4200, 2500, 2800, 200, 700, {'density':-1000}),
         Prism(300, 2500, 1800, 2700, 500, 1000, {'density':-1000}),
         Prism(4000, 4500, 500, 1500, 400, 1000, {'density':-1000}),
         Prism(1800, 3700, 500, 1500, 300, 1300, {'density':-1000}),
         Prism(500, 4500, 4000, 4500, 400, 1300, {'density':-1000})]
# show it
myv.figure()
myv.prisms(model, 'density')
myv.axes(myv.outline(bounds), ranges=[i*0.001 for i in bounds],
              fmt='%.1f', nlabels=6)
myv.wall_bottom(bounds)
myv.wall_north(bounds)
myv.show()
# and use it to generate some tensor data
shape = (51, 51)
area = bounds[0:4]
noise = 2
x, y, z = gridder.regular(area, shape, z=-150)
gyy = utils.contaminate(gm.prism.gyy(x, y, z, model), noise)
gyz = utils.contaminate(gm.prism.gyz(x, y, z, model), noise)
gzz = utils.contaminate(gm.prism.gzz(x, y, z, model), noise)

# Set up the inversion:
# Create a prism mesh
mesh = PrismMesh(bounds, (15, 50, 50))
# Wrap the data so that harvester can use it
data = [gm.harvester.Gyy(x, y, z, gyy),
        gm.harvester.Gyz(x, y, z, gyz),
        gm.harvester.Gzz(x, y, z, gzz)]
# and the seeds
seeds = gm.harvester.sow(
    [( 800, 3250, 600, {'density':1200}),
     (1200, 3250, 600, {'density':1200}),
     (1700, 3250, 600, {'density':1200}),
     (2100, 3250, 600, {'density':1200}),
     (2500, 3250, 600, {'density':1200}),
     (2900, 3250, 600, {'density':1200}),
     (3300, 3250, 600, {'density':1200}),
     (3700, 3250, 600, {'density':1200}),
     (4200, 3250, 600, {'density':1200}),
     (3300, 2050, 600, {'density':1200}),
     (3600, 2050, 600, {'density':1200}),
     (4000, 2050, 600, {'density':1200}),
     (4300, 2050, 600, {'density':1200})],
    mesh)
# Run the inversion and collect the results
estimate, predicted = gm.harvester.harvest(data, seeds, mesh,
    compactness=1., threshold=0.0001)

# Insert the estimated density values into the mesh
mesh.addprop('density', estimate['density'])
# and get only the prisms corresponding to our estimate
density_model = vremove(0, 'density', mesh)
```

```python
print "Accretions: %d" % (len(density_model) - len(seeds))

# Get the predicted data from the data modules
tensor = (gyy, gyz, gzz)
# plot it
for true, pred in zip(tensor, predicted):
    mpl.figure()
    mpl.title("True: color | Inversion: contour")
    mpl.axis('scaled')
    levels = mpl.contourf(y*0.001, x*0.001, true, shape, 12)
    mpl.colorbar()
    mpl.contour(y*0.001, x*0.001, pred, shape, levels, color='k')
    mpl.xlabel('Horizontal coordinate y (km)')
    mpl.ylabel('Horizontal coordinate x (km)')
mpl.show()

# Plot the inversion result
myv.figure()
myv.prisms(model, 'density', style='wireframe')
myv.prisms(density_model, 'density', vmin=0)
myv.axes(myv.outline(bounds), ranges=[i*0.001 for i in bounds], fmt='%.1f',
    nlabels=6)
myv.wall_bottom(bounds)
myv.wall_north(bounds)
myv.show()
```

6.22 GravMag: 3D gravity gradient inversion by planting anomalous densities using `harvester` (3 sources sources)

```python
"""
GravMag: 3D gravity gradient inversion by planting anomalous densities using
``harvester`` (3 sources sources)
"""
from fatiando import logger, gridder, utils
import fatiando.gravmag as gm
from fatiando.mesher import Prism, PrismMesh, vremove
from fatiando.vis import mpl, myv

log = logger.get()
log.info(logger.header())
log.info(__doc__)

# Generate a synthetic model
bounds = [0, 5000, 0, 5000, -500, 2000]
model = [Prism(600, 1200, 200, 4200, 400, 900, {'density':1500}),
         Prism(3000, 4000, 1000, 2000, 200, 800, {'density':1000}),
         Prism(2700, 3200, 3700, 4200, 0, 900, {'density':800})]
# show it
myv.figure()
myv.prisms(model, 'density')
myv.axes(myv.outline(bounds), ranges=[i*0.001 for i in bounds],
              fmt='%.1f', nlabels=6)
myv.wall_bottom(bounds)
myv.wall_north(bounds)
myv.show()
# and use it to generate some tensor data
shape = (25, 25)
area = bounds[0:4]
x, y, z = gridder.regular(area, shape, z=-650)
gxy = utils.contaminate(gm.prism.gxy(x, y, z, model), 1)
```

```
gzz = utils.contaminate(gm.prism.gzz(x, y, z, model), 1)

# Wrap the data so that harvester can use it
data = [gm.harvester.Gxy(x, y, z, gxy),
        gm.harvester.Gzz(x, y, z, gzz)]
# Create a prism mesh
mesh = PrismMesh(bounds, (20, 50, 50))
# and the seeds
seeds = gm.harvester.sow(
    [(901, 701, 750, {'density':1500}),
     (901, 1201, 750, {'density':1500}),
     (901, 1701, 750, {'density':1500}),
     (901, 2201, 750, {'density':1500}),
     (901, 2701, 750, {'density':1500}),
     (901, 3201, 750, {'density':1500}),
     (901, 3701, 750, {'density':1500}),
     (3701, 1201, 501, {'density':1000}),
     (3201, 1201, 501, {'density':1000}),
     (3701, 1701, 501, {'density':1000}),
     (3201, 1701, 501, {'density':1000}),
     (2951, 3951, 301, {'density':800}),
     (2951, 3951, 701, {'density':800})],
    mesh)
# Run the inversion and collect the results
estimate, predicted = gm.harvester.harvest(data, seeds, mesh,
    compactness=1, threshold=0.0001)
# Insert the estimated density values into the mesh
mesh.addprop('density', estimate['density'])
# and get only the prisms corresponding to our estimate
density_model = vremove(0, 'density', mesh)

# Plot the results
tensor = (gxy, gzz)
titles = ('gxy', 'gzz')
mpl.figure()
mpl.suptitle("True: color | Inversion: contour")
for i in xrange(len(tensor)):
    mpl.subplot(2, 2, i + 1)
    mpl.title(titles[i])
    mpl.axis('scaled')
    levels = mpl.contourf(y*0.001, x*0.001, tensor[i], shape, 12)
    mpl.colorbar()
    mpl.contour(y*0.001, x*0.001, predicted[i], shape, levels, color='k')
for i in xrange(len(tensor)):
    mpl.subplot(2, 2, i + 3)
    residuals = tensor[i] - predicted[i]
    mpl.title('residuals stddev = %.2f' % (residuals.std()))
    mpl.hist(residuals, bins=10)
    mpl.xlabel('Residual (Eotvos)')
mpl.show()
myv.figure()
myv.prisms(model, 'density', style='wireframe')
myv.prisms([mesh[s.i] for s in seeds], 'density')
myv.axes(myv.outline(bounds), ranges=[i*0.001 for i in bounds], fmt='%.1f',
    nlabels=6)
myv.wall_bottom(bounds)
myv.wall_north(bounds)
myv.figure()
myv.prisms(model, 'density', style='wireframe')
myv.prisms(density_model, 'density')
myv.axes(myv.outline(bounds), ranges=[i*0.001 for i in bounds], fmt='%.1f',
    nlabels=6)
myv.wall_bottom(bounds)
myv.wall_north(bounds)
myv.show()
```

6.23 GravMag: 3D gravity gradient inversion by planting anomalous densities using `harvester` (dipping example)

```
"""
GravMag: 3D gravity gradient inversion by planting anomalous densities using
``harvester`` (dipping example)
"""
from fatiando import logger, gridder, utils
from fatiando import gravmag as gm
from fatiando.mesher import Prism, PrismMesh, vremove
from fatiando.vis import mpl, myv

log = logger.get()
log.info(logger.header())

# Create a synthetic model
props = {'density':1000}
model = [Prism(400, 600, 300, 500, 200, 400, props),
         Prism(400, 600, 400, 600, 400, 600, props),
         Prism(400, 600, 500, 700, 600, 800, props)]
# and generate synthetic data from it
shape = (51, 51)
bounds = [0, 1000, 0, 1000, 0, 1000]
area = bounds[0:4]
xp, yp, zp = gridder.regular(area, shape, z=-150)
noise = 0.5
gxx = utils.contaminate(gm.prism.gxx(xp, yp, zp, model), noise)
gxy = utils.contaminate(gm.prism.gxy(xp, yp, zp, model), noise)
gxz = utils.contaminate(gm.prism.gxz(xp, yp, zp, model), noise)
gyy = utils.contaminate(gm.prism.gyy(xp, yp, zp, model), noise)
gyz = utils.contaminate(gm.prism.gyz(xp, yp, zp, model), noise)
gzz = utils.contaminate(gm.prism.gzz(xp, yp, zp, model), noise)
tensor = [gxx, gxy, gxz, gyy, gyz, gzz]
titles = ['gxx', 'gxy', 'gxz', 'gyy', 'gyz', 'gzz']
# plot the data
mpl.figure()
for i in xrange(len(tensor)):
    mpl.subplot(2, 3, i + 1)
    mpl.title(titles[i])
    mpl.axis('scaled')
    levels = mpl.contourf(yp, xp, tensor[i], shape, 30)
    mpl.colorbar()
    mpl.xlabel('y (km)')
    mpl.ylabel('x (km)')
    mpl.m2km()
mpl.show()

# Inversion setup
# Create a mesh
mesh = PrismMesh(bounds, (30, 30, 30))
# Wrap the data so that harvester can use it
data = [gm.harvester.Gxx(xp, yp, zp, gxx),
        gm.harvester.Gxy(xp, yp, zp, gxy),
        gm.harvester.Gxz(xp, yp, zp, gxz),
        gm.harvester.Gyy(xp, yp, zp, gyy),
        gm.harvester.Gyz(xp, yp, zp, gyz),
        gm.harvester.Gzz(xp, yp, zp, gzz)]
```

```
# Make the seeds
seeds = gm.harvester.sow([
    [500, 400, 210, {'density':1000}],
    [500, 550, 510, {'density':1000}]], mesh)
# Run the inversioin
estimate, predicted = gm.harvester.harvest(data, seeds, mesh,
    compactness=0.5, threshold=0.001)
# Put the estimated density values in the mesh
mesh.addprop('density', estimate['density'])

# Plot the adjustment
mpl.figure()
mpl.suptitle("True: color | Inversion: contour")
for i in xrange(len(tensor)):
    mpl.subplot(2, 3, i + 1)
    mpl.title(titles[i])
    mpl.axis('scaled')
    levels = mpl.contourf(yp, xp, tensor[i], shape, 12)
    mpl.colorbar()
    mpl.contour(yp, xp, predicted[i], shape, levels, color='k')
    mpl.xlabel('y (km)')
    mpl.ylabel('x (km)')
    mpl.m2km()
mpl.figure()
mpl.suptitle("Residuals")
for i in xrange(len(tensor)):
    residuals = tensor[i] - predicted[i]
    mpl.subplot(2, 3, i + 1)
    mpl.title(titles[i] + ': stddev=%g' % (residuals.std()))
    mpl.hist(residuals, bins=10, color='gray')
    mpl.xlabel('Residuals (Eotvos)')
mpl.show()
# Plot the result
myv.figure()
myv.prisms(model, 'density', style='wireframe')
myv.prisms([mesh[s.i] for s in seeds], 'density')
myv.axes(myv.outline(bounds), ranges=[i*0.001 for i in bounds], fmt='%.1f',
    nlabels=6)
myv.wall_bottom(bounds)
myv.wall_north(bounds)
myv.figure()
myv.prisms(model, 'density', style='wireframe')
myv.prisms(vremove(0, 'density', mesh), 'density')
myv.axes(myv.outline(bounds), ranges=[i*0.001 for i in bounds], fmt='%.1f',
    nlabels=6)
myv.wall_bottom(bounds)
myv.wall_north(bounds)
myv.show()
```

6.24 GravMag: Compare the results of different 3D potential field imaging methods (migration, generalized inverse, and sandwich model)

```
"""
GravMag: Compare the results of different 3D potential field imaging methods
(migration, generalized inverse, and sandwich model)
"""
from multiprocessing import Pool
from fatiando import logger, gridder, mesher, gravmag
```

```python
from fatiando.vis import mpl, myv

log = logger.get()
log.info(logger.header())
log.info(__doc__)

# Make some synthetic gravity data from a polygonal prism model
log.info("Draw the polygons one by one")
bounds = [-10000, 10000, -10000, 10000, 0, 10000]
area = bounds[:4]
depths = [0, 1000, 3000, 7000]
prisms = []
for i in range(1, len(depths)):
    # Plot previous prisms
    axes = mpl.figure().gca()
    mpl.axis('scaled')
    for p in prisms:
        mpl.polygon(p, '.-k', xy2ne=True)
    # Draw a new polygon
    polygon = mpl.draw_polygon(area, axes, xy2ne=True)
    # append the newly drawn one
    prisms.append(
        mesher.PolygonalPrism(polygon, depths[i - 1], depths[i],
            {'density':500}))
meshshape = (30, 30, 30)
xp, yp, zp = gridder.regular(area, meshshape[1:], z=-10)
gz = gravmag.polyprism.gz(xp, yp, zp, prisms)

# Plot the data
mpl.figure()
mpl.axis('scaled')
mpl.contourf(yp, xp, gz, meshshape[1:], 30)
mpl.colorbar()
mpl.xlabel('East (km)')
mpl.ylabel('North (km)')
mpl.m2km()
mpl.show()

# A function to the imaging methods and make the 3D plots
def run(title):
    if title == 'Migration':
        result = gravmag.imaging.migrate(xp, yp, zp, gz, bounds[-2], bounds[-1],
            meshshape, power=0.5)
    elif title == 'Generalized Inverse':
        result = gravmag.imaging.geninv(xp, yp, zp, gz, meshshape[1:],
            bounds[-2], bounds[-1], meshshape[0])
    elif title == 'Sandwich':
        result = gravmag.imaging.sandwich(xp, yp, zp, gz, meshshape[1:],
            bounds[-2], bounds[-1], meshshape[0], power=0.5)
    # Plot the results
    myv.figure()
    myv.polyprisms(prisms, 'density', style='wireframe', linewidth=2)
    myv.prisms(result, 'density', edges=False)
    axes = myv.axes(myv.outline(), ranges=[b*0.001 for b in bounds],
        fmt='%.0f')
    myv.wall_bottom(axes.axes.bounds)
    myv.wall_north(axes.axes.bounds)
    myv.title(title)
    myv.show()

titles = ['Migration', 'Generalized Inverse', 'Sandwich']
# Use a pool of workers to run each method in a different process
pool = Pool(3)
```

```python
# Use map to apply the run function to each title
pool.map(run, titles)
```

6.25 GravMag: 3D imaging using the Generalized Inverse method on synthetic gravity data (simple model)

```python
"""
GravMag: 3D imaging using the Generalized Inverse method on synthetic gravity
data (simple model)
"""
from fatiando import logger, gridder, mesher, gravmag
from fatiando.vis import mpl, myv

log = logger.get()
log.info(logger.header())
log.info(__doc__)

# Make some synthetic gravity data from a simple prism model
prisms = [mesher.Prism(-1000,1000,-3000,3000,0,5000,{'density':1000})]
shape = (25, 25)
xp, yp, zp = gridder.regular((-5000, 5000, -5000, 5000), shape, z=-10)
gz = gravmag.prism.gz(xp, yp, zp, prisms)

# Plot the data
mpl.figure()
mpl.axis('scaled')
mpl.contourf(yp, xp, gz, shape, 30)
mpl.colorbar()
mpl.xlabel('East (km)')
mpl.ylabel('North (km)')
mpl.m2km()
mpl.show()

# Run the Generalized Inverse
mesh = gravmag.imaging.geninv(xp, yp, zp, gz, shape, 0, 10000, 25)

# Plot the results
myv.figure()
myv.prisms(prisms, 'density', style='wireframe', linewidth=5)
myv.prisms(mesh, 'density', edges=False)
axes = myv.axes(myv.outline())
myv.wall_bottom(axes.axes.bounds)
myv.wall_north(axes.axes.bounds)
myv.show()
```

6.26 GravMag: 3D imaging using the Generalized Inverse method on synthetic gravity data (more complex model + noisy data)

```python
"""
GravMag: 3D imaging using the Generalized Inverse method on synthetic gravity
data (more complex model + noisy data)
"""
from fatiando import logger, gridder, mesher, gravmag, utils
from fatiando.vis import mpl, myv

log = logger.get()
log.info(logger.header())
log.info(__doc__)

# Make some synthetic gravity data from a simple prism model
prisms = [mesher.Prism(-4000,-1000,-4000,-2000,2000,5000,{'density':800}),
          mesher.Prism(-1000,1000,-1000,1000,1000,6000,{'density':-800}),
          mesher.Prism(2000,4000,3000,4000,0,4000,{'density':600})]
shape = (25, 25)
xp, yp, zp = gridder.regular((-5000, 5000, -5000, 5000), shape, z=-10)
gz = utils.contaminate(gravmag.prism.gz(xp, yp, zp, prisms), 0.1)

# Plot the data
mpl.figure()
mpl.axis('scaled')
mpl.contourf(yp, xp, gz, shape, 30)
mpl.colorbar()
mpl.xlabel('East (km)')
mpl.ylabel('North (km)')
mpl.m2km()
mpl.show()

# Run the Generalized Inverse
mesh = gravmag.imaging.geninv(xp, yp, zp, gz, shape,
    0, 10000, 25)

# Plot the results
myv.figure()
myv.prisms(prisms, 'density', style='wireframe')
myv.prisms(mesh, 'density', edges=False, linewidth=5)
axes = myv.axes(myv.outline())
myv.wall_bottom(axes.axes.bounds)
myv.wall_north(axes.axes.bounds)
myv.show()
```

6.27 GravMag: 3D imaging using the migration method on synthetic gravity data (simple model)

```python
"""
GravMag: 3D imaging using the migration method on synthetic gravity data
(simple model)
"""
from fatiando import logger, gridder, mesher, gravmag
from fatiando.vis import mpl, myv

log = logger.get()
log.info(logger.header())
log.info(__doc__)

# Make some synthetic gravity data from a simple prism model
prisms = [mesher.Prism(-1000,1000,-2000,2000,2000,4000,{'density':500})]
shape = (50, 50)
xp, yp, zp = gridder.regular((-10000, 10000, -10000, 10000), shape, z=-10)
gz = gravmag.prism.gz(xp, yp, zp, prisms)

# Plot the data
mpl.figure()
mpl.axis('scaled')
mpl.contourf(yp, xp, gz, shape, 30)
mpl.colorbar()
mpl.xlabel('East (km)')
mpl.ylabel('North (km)')
mpl.m2km()
mpl.show()

mesh = gravmag.imaging.migrate(xp, yp, zp, gz, 0, 10000, (25, 25, 25))

# Plot the results
myv.figure()
myv.prisms(prisms, 'density', style='wireframe', linewidth=2)
myv.prisms(mesh, 'density', edges=False)
axes = myv.axes(myv.outline())
myv.wall_bottom(axes.axes.bounds)
myv.wall_north(axes.axes.bounds)
myv.show()
```

6.28 GravMag: 3D imaging using the migration method on synthetic gravity data (more complex model + noisy data)

```
"""
GravMag: 3D imaging using the migration method on synthetic gravity data
(more complex model + noisy data)
"""
from fatiando import logger, gridder, mesher, gravmag, utils
from fatiando.vis import mpl, myv

log = logger.get()
log.info(logger.header())
log.info(__doc__)

# Make some synthetic gravity data from a simple prism model
prisms = [mesher.Prism(-4000,0,-4000,-2000,2000,5000,{'density':1200}),
          mesher.Prism(-1000,1000,-1000,1000,1000,7000,{'density':-800}),
          mesher.Prism(2000,4000,3000,4000,0,2000,{'density':600})]
# Calculate on a scatter of points to show that migration doesn't need gridded
# data
xp, yp, zp = gridder.scatter((-6000, 6000, -6000, 6000), 1000, z=-10)
gz = utils.contaminate(gravmag.prism.gz(xp, yp, zp, prisms), 0.1)

# Plot the data
shape = (50, 50)
mpl.figure()
mpl.axis('scaled')
mpl.contourf(yp, xp, gz, shape, 30, interp=True)
mpl.colorbar()
mpl.plot(yp, xp, '.k')
mpl.xlabel('East (km)')
mpl.ylabel('North (km)')
mpl.m2km()
mpl.show()

mesh = gravmag.imaging.migrate(xp, yp, zp, gz, 0, 10000, (30, 30, 30), power=0.8)

# Plot the results
myv.figure()
myv.prisms(prisms, 'density', style='wireframe', linewidth=2)
myv.prisms(mesh, 'density', edges=False)
axes = myv.axes(myv.outline())
myv.wall_bottom(axes.axes.bounds)
myv.wall_north(axes.axes.bounds)
myv.show()
```

6.29 GravMag: 3D imaging using the sandwich model method on synthetic gravity data (simple example)

```
"""
GravMag: 3D imaging using the sandwich model method on synthetic gravity data
(simple example)
"""
from fatiando import logger, gridder, mesher, gravmag
from fatiando.vis import mpl, myv

log = logger.get()
log.info(logger.header())
log.info(__doc__)

# Make some synthetic gravity data from a simple prism model
prisms = [mesher.Prism(-1000,1000,-2000,2000,2000,4000,{'density':500})]
shape = (25, 25)
xp, yp, zp = gridder.regular((-5000, 5000, -5000, 5000), shape, z=-10)
gz = gravmag.prism.gz(xp, yp, zp, prisms)

# Plot the data
mpl.figure()
mpl.axis('scaled')
mpl.contourf(yp, xp, gz, shape, 30)
mpl.colorbar()
mpl.xlabel('East (km)')
mpl.ylabel('North (km)')
mpl.m2km()
mpl.show()

mesh = gravmag.imaging.sandwich(xp, yp, zp, gz, shape, 0, 10000, 25)

# Plot the results
myv.figure()
myv.prisms(prisms, 'density', style='wireframe', linewidth=2)
myv.prisms(mesh, 'density', edges=False)
axes = myv.axes(myv.outline())
myv.wall_bottom(axes.axes.bounds)
myv.wall_north(axes.axes.bounds)
myv.show()
```

6.30 GravMag: 3D imaging using the sandwich model method on synthetic gravity data (more complex model)

```
"""
GravMag: 3D imaging using the sandwich model method on synthetic gravity data
(more complex model)
"""
from fatiando import logger, gridder, mesher, gravmag
from fatiando.vis import mpl, myv

log = logger.get()
log.info(logger.header())
log.info(__doc__)

# Make some synthetic gravity data from a simple prism model
prisms = [mesher.Prism(-4000,0,-4000,-2000,2000,5000,{'density':1200}),
          mesher.Prism(-1000,1000,-1000,1000,1000,7000,{'density':-300}),
          mesher.Prism(2000,4000,3000,4000,0,2000,{'density':600})]
```

```
shape = (25, 25)
xp, yp, zp = gridder.regular((-10000, 10000, -10000, 10000), shape, z=-10)
gz = gravmag.prism.gz(xp, yp, zp, prisms)

# Plot the data
mpl.figure()
mpl.axis('scaled')
mpl.contourf(yp, xp, gz, shape, 30)
mpl.colorbar()
mpl.xlabel('East (km)')
mpl.ylabel('North (km)')
mpl.m2km()
mpl.show()

mesh = gravmag.imaging.sandwich(xp, yp, zp, gz, shape, 0, 10000, 25)

# Plot the results
myv.figure()
myv.prisms(prisms, 'density', style='wireframe', linewidth=3)
myv.prisms(mesh, 'density', edges=False)
axes = myv.axes(myv.outline())
myv.wall_bottom(axes.axes.bounds)
myv.wall_north(axes.axes.bounds)
myv.show()
```

6.31 GravMag: 3D forward modeling of total-field magnetic anomaly using polygonal prisms

```
"""
GravMag: 3D forward modeling of total-field magnetic anomaly using polygonal
prisms
"""
from fatiando import logger, mesher, gridder, gravmag
from fatiando.vis import mpl, myv

log = logger.get()
log.info(logger.header())
log.info(__doc__)

log.info("Draw the polygons one by one")
bounds = [-5000, 5000, -5000, 5000, 0, 5000]
area = bounds[:4]
axis = mpl.figure().gca()
mpl.axis('scaled')
prisms = [
    mesher.PolygonalPrism(
        mpl.draw_polygon(area, axis, xy2ne=True),
        0, 2000, {'magnetization':2})]
# Calculate the effect
shape = (100, 100)
xp, yp, zp = gridder.regular(area, shape, z=-500)
tf = gravmag.polyprism.tf(xp, yp, zp, prisms, 30, -15)
# and plot it
mpl.figure()
mpl.axis('scaled')
mpl.title("Total field anomalyproduced by prism model (nT)")
mpl.contourf(yp, xp, tf, shape, 20)
mpl.colorbar()
for p in prisms:
    mpl.polygon(p, '.-k', xy2ne=True)
mpl.set_area(area)
mpl.m2km()
mpl.show()
# Show the prisms
myv.figure()
myv.polyprisms(prisms, 'magnetization')
myv.axes(myv.outline(bounds), ranges=[i*0.001 for i in bounds])
myv.wall_north(bounds)
myv.wall_bottom(bounds)
myv.show()
```

6.32 GravMag: 3D forward modeling of total-field magnetic anomaly using rectangular prisms (model with induced and remanent magnetization)

```
"""
GravMag: 3D forward modeling of total-field magnetic anomaly using rectangular
prisms (model with induced and remanent magnetization)
"""
from fatiando import logger, mesher, gridder, gravmag
from fatiando.vis import mpl, myv

log = logger.get()
log.info(logger.header())
log.info(__doc__)

bounds = [-5000, 5000, -5000, 5000, 0, 5000]
prisms = [
    mesher.Prism(-4000,-3000,-4000,-3000,0,2000,
        {'magnetization':2}),
    mesher.Prism(-1000,1000,-1000,1000,0,2000,
        {'magnetization':1}),
    # This prism has remanent magnetization because it's physical property
    # dict has inclination and declination
    mesher.Prism(2000,4000,3000,4000,0,2000,
        {'magnetization':3, 'inclination':-10, 'declination':45})]
# Create a regular grid at 100m height
shape = (200, 200)
area = bounds[:4]
xp, yp, zp = gridder.regular(area, shape, z=-500)
# Calculate the anomaly for a given regional field
tf = gravmag.prism.tf(xp, yp, zp, prisms, 30, -15)
# Plot
mpl.figure()
mpl.title("Total-field anomaly (nT)")
mpl.axis('scaled')
mpl.contourf(yp, xp, tf, shape, 15)
mpl.colorbar()
mpl.xlabel('East y (km)')
mpl.ylabel('North x (km)')
mpl.m2km()
mpl.show()
# Show the prisms
myv.figure()
myv.prisms(prisms, 'magnetization')
myv.axes(myv.outline(bounds), ranges=[i*0.001 for i in bounds])
myv.wall_north(bounds)
myv.wall_bottom(bounds)
myv.show()
```

6.33 GravMag: 3D forward modeling of total-field magnetic anomaly using spheres

```python
"""
GravMag: 3D forward modeling of total-field magnetic anomaly using spheres
"""
from fatiando import logger, mesher, gridder, gravmag
from fatiando.vis import mpl

log = logger.get()
log.info(logger.header())
log.info(__doc__)

# Create a sphere model
# Considering only induced magnetization, so I don't give the 'inclination' and
# 'declination' properties
spheres = [mesher.Sphere(0, 0, 3000, 1000, {'magnetization':1})]
# Create a regular grid at 100m height
shape = (100, 100)
area = (-5000, 5000, -5000, 5000)
xp, yp, zp = gridder.regular(area, shape, z=-100)
# Calculate the anomaly for a given regional field
tf = gravmag.sphere.tf(xp, yp, zp, spheres, 30, 0)
# Plot
mpl.figure()
mpl.title("Total-field anomaly (nT)")
mpl.axis('scaled')
mpl.contourf(yp, xp, tf, shape, 15)
mpl.colorbar()
mpl.xlabel('East y (km)')
mpl.ylabel('North x (km)')
mpl.m2km()
mpl.show()
```

6.34 GravMag: Generate synthetic gravity data on an irregular grid

```python
"""
GravMag: Generate synthetic gravity data on an irregular grid
"""
import fatiando as ft
from fatiando import logger, mesher, gridder, gravmag
from fatiando.vis import mpl

log = logger.get()
log.info(logger.header())
log.info(__doc__)

prisms = [mesher.Prism(-2000, 2000, -2000, 2000, 0, 2000, {'density':1000})]
xp, yp, zp = gridder.scatter((-5000, 5000, -5000, 5000), n=100, z=-100)
gz = gravmag.prism.gz(xp, yp, zp, prisms)

shape = (100,100)
mpl.axis('scaled')
mpl.title("gz produced by prism model on an irregular grid (mGal)")
mpl.plot(xp, yp, '.k', label='Grid points')
levels = mpl.contourf(xp, yp, gz, shape, 12, interp=True)
mpl.contour(xp, yp, gz, shape, levels, interp=True)
mpl.legend(loc='lower right', numpoints=1)
mpl.m2km()
mpl.show()
```

6.35 GravMag: Center of mass estimation using the first eigenvector of the gravity gradient tensor (simple model)

```
"""
GravMag: Center of mass estimation using the first eigenvector of the gravity
gradient tensor (simple model)
"""
from fatiando.vis import mpl, myv
from fatiando import logger, mesher, gridder, utils, gravmag

log = logger.get()
log.info(logger.header())
log.info(__doc__)

# Generate some synthetic data
prisms = [mesher.Prism(-1000,1000,-1000,1000,1000,3000,{'density':1000})]
shape = (100, 100)
xp, yp, zp = gridder.regular((-5000, 5000, -5000, 5000), shape, z=-150)
noise = 2
tensor = [utils.contaminate(gravmag.prism.gxx(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gxy(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gxz(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gyy(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gyz(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gzz(xp, yp, zp, prisms), noise)]
# Plot the data
titles = ['gxx', 'gxy', 'gxz', 'gyy', 'gyz', 'gzz']
mpl.figure()
for i, title in enumerate(titles):
    mpl.subplot(3, 2, i + 1)
    mpl.title(title)
    mpl.axis('scaled')
    levels = mpl.contourf(yp, xp, tensor[i], shape, 10)
    mpl.contour(yp, xp, tensor[i], shape, levels)
    mpl.m2km()
mpl.show()
# Get the eigenvectors from the tensor data
eigenvals, eigenvecs = gravmag.tensor.eigen(tensor)
# Use the first eigenvector to estimate the center of mass
cm, sigma = gravmag.tensor.center_of_mass(xp, yp, zp, eigenvecs[0])
# Sigma is the RMS error
print "Sigma = %g" % (sigma)
# Plot the prism and the estimated center of mass
myv.figure()
myv.points([cm], size=300.)
myv.prisms(prisms, prop='density', opacity=0.5)
axes = myv.axes(
    myv.outline(extent=[-5000, 5000, -5000, 5000, 0, 5000]))
myv.wall_bottom(axes.axes.bounds, opacity=0.2)
myv.wall_north(axes.axes.bounds)
myv.show()
```

6.36 GravMag: Center of mass estimation using the first eigenvector of the gravity gradient tensor (inverted pyramid model)

```python
"""
GravMag: Center of mass estimation using the first eigenvector of the gravity
gradient tensor (inverted pyramid model)
"""
from fatiando.vis import mpl, myv
from fatiando import logger, mesher, gridder, utils, gravmag

log = logger.get()
log.info(logger.header())
log.info(__doc__)

# Generate some synthetic data
prisms = [mesher.Prism(-2000,2000,-2000,2000,500,1000,{'density':1000}),
          mesher.Prism(-1000,1000,-1000,1000,1000,1500,{'density':1000}),
          mesher.Prism(-500,500,-500,500,1500,2000,{'density':1000})]
shape = (100, 100)
xp, yp, zp = gridder.regular((-5000, 5000, -5000, 5000), shape, z=-150)
noise = 1
tensor = [utils.contaminate(gravmag.prism.gxx(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gxy(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gxz(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gyy(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gyz(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gzz(xp, yp, zp, prisms), noise)]
# Plot the data
titles = ['gxx', 'gxy', 'gxz', 'gyy', 'gyz', 'gzz']
mpl.figure()
for i, title in enumerate(titles):
    mpl.subplot(3, 2, i + 1)
    mpl.title(title)
    mpl.axis('scaled')
    levels = mpl.contourf(yp, xp, tensor[i], shape, 10)
    mpl.contour(yp, xp, tensor[i], shape, levels)
    mpl.m2km()
mpl.show()
# Get the eigenvectors from the tensor data
eigenvals, eigenvecs = gravmag.tensor.eigen(tensor)
# Use the first eigenvector to estimate the center of mass
cm, sigma = gravmag.tensor.center_of_mass(xp, yp, zp, eigenvecs[0])
print "Sigma = %g" % (sigma)
# Plot the prism and the estimated center of mass
myv.figure()
myv.points([cm], size=300.)
myv.prisms(prisms, prop='density', opacity=0.5)
axes = myv.axes(
    myv.outline(extent=[-5000, 5000, -5000, 5000, 0, 5000]))
myv.wall_bottom(axes.axes.bounds, opacity=0.2)
myv.wall_north(axes.axes.bounds)
myv.show()
```

6.37 GravMag: Center of mass estimation using the first eigenvector of the gravity gradient tensor (pyramid model)

```python
"""
GravMag: Center of mass estimation using the first eigenvector of the gravity
gradient tensor (pyramid model)
```

```
"""
from fatiando.vis import mpl, myv
from fatiando import logger, mesher, gridder, utils, gravmag

log = logger.get()
log.info(logger.header())
log.info(__doc__)

# Generate some synthetic data
prisms = [mesher.Prism(-500,500,-500,500,500,1000,{'density':1000}),
          mesher.Prism(-1000,1000,-1000,1000,1000,1500,{'density':1000}),
          mesher.Prism(-2000,2000,-2000,2000,1500,2000,{'density':1000})]
shape = (100, 100)
xp, yp, zp = gridder.regular((-5000, 5000, -5000, 5000), shape, z=-150)
noise = 1
tensor = [utils.contaminate(gravmag.prism.gxx(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gxy(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gxz(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gyy(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gyz(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gzz(xp, yp, zp, prisms), noise)]
# Plot the data
titles = ['gxx', 'gxy', 'gxz', 'gyy', 'gyz', 'gzz']
mpl.figure()
for i, title in enumerate(titles):
    mpl.subplot(3, 2, i + 1)
    mpl.title(title)
    mpl.axis('scaled')
    levels = mpl.contourf(yp, xp, tensor[i], shape, 10)
    mpl.contour(yp, xp, tensor[i], shape, levels)
    mpl.m2km()
mpl.show()
# Get the eigenvectors from the tensor data
eigenvals, eigenvecs = gravmag.tensor.eigen(tensor)
# Use the first eigenvector to estimate the center of mass
cm, sigma = gravmag.tensor.center_of_mass(xp, yp, zp, eigenvecs[0])
print "Sigma = %g" % (sigma)
# Plot the prism and the estimated center of mass
myv.figure()
myv.points([cm], size=300.)
myv.prisms(prisms, prop='density', opacity=0.5)
axes = myv.axes(
    myv.outline(extent=[-5000, 5000, -5000, 5000, 0, 5000]))
myv.wall_bottom(axes.axes.bounds, opacity=0.2)
myv.wall_north(axes.axes.bounds)
myv.show()
```

6.38 GravMag: Center of mass estimation using the first eigenvector of the gravity gradient tensor (elongated model)

```
"""
GravMag: Center of mass estimation using the first eigenvector of the gravity
gradient tensor (elongated model)
"""
from fatiando.vis import mpl, myv
from fatiando import logger, mesher, gridder, utils, gravmag

log = logger.get()
log.info(logger.header())
log.info(__doc__)
```

```

# Generate some synthetic data
prisms = [mesher.Prism(-4000,4000,-500,500,500,1000,{'density':1000})]
shape = (100, 100)
xp, yp, zp = gridder.regular((-5000, 5000, -5000, 5000), shape, z=-150)
noise = 2
tensor = [utils.contaminate(gravmag.prism.gxx(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gxy(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gxz(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gyy(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gyz(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gzz(xp, yp, zp, prisms), noise)]
# Plot the data
titles = ['gxx', 'gxy', 'gxz', 'gyy', 'gyz', 'gzz']
mpl.figure()
for i, title in enumerate(titles):
    mpl.subplot(3, 2, i + 1)
    mpl.title(title)
    mpl.axis('scaled')
    levels = mpl.contourf(yp, xp, tensor[i], shape, 10)
    mpl.contour(yp, xp, tensor[i], shape, levels)
    mpl.m2km()
mpl.show()
# Get the eigenvectors from the tensor data
eigenvals, eigenvecs = gravmag.tensor.eigen(tensor)
# Use the first eigenvector to estimate the center of mass
cm, sigma = gravmag.tensor.center_of_mass(xp, yp, zp, eigenvecs[0])
print "Sigma = %g" % (sigma)
# Plot the prism and the estimated center of mass
myv.figure()
myv.points([cm], size=300.)
myv.prisms(prisms, prop='density', opacity=0.5)
axes = myv.axes(
    myv.outline(extent=[-5000, 5000, -5000, 5000, 0, 5000]))
myv.wall_bottom(axes.axes.bounds, opacity=0.2)
myv.wall_north(axes.axes.bounds)
myv.show()
```

6.39 GravMag: Center of mass estimation using the first eigenvector of the gravity gradient tensor (vertical elongated model)

```
"""
GravMag: Center of mass estimation using the first eigenvector of the gravity
gradient tensor (vertical elongated model)
"""
from fatiando.vis import mpl, myv
from fatiando import logger, mesher, gridder, utils, gravmag

log = logger.get()
log.info(logger.header())
log.info(__doc__)

# Generate some synthetic data
prisms = [mesher.Prism(-500,500,-500,500,000,4000,{'density':1000})]
shape = (100, 100)
xp, yp, zp = gridder.regular((-5000, 5000, -5000, 5000), shape, z=-150)
noise = 1
tensor = [utils.contaminate(gravmag.prism.gxx(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gxy(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gxz(xp, yp, zp, prisms), noise),
```

```python
            utils.contaminate(gravmag.prism.gyy(xp, yp, zp, prisms), noise),
            utils.contaminate(gravmag.prism.gyz(xp, yp, zp, prisms), noise),
            utils.contaminate(gravmag.prism.gzz(xp, yp, zp, prisms), noise)]
# Plot the data
titles = ['gxx', 'gxy', 'gxz', 'gyy', 'gyz', 'gzz']
mpl.figure()
for i, title in enumerate(titles):
    mpl.subplot(3, 2, i + 1)
    mpl.title(title)
    mpl.axis('scaled')
    levels = mpl.contourf(yp, xp, tensor[i], shape, 10)
    mpl.contour(yp, xp, tensor[i], shape, levels)
    mpl.m2km()
mpl.show()
# Get the eigenvectors from the tensor data
eigenvals, eigenvecs = gravmag.tensor.eigen(tensor)
# Use the first eigenvector to estimate the center of mass
cm, sigma = gravmag.tensor.center_of_mass(xp, yp, zp, eigenvecs[0])
print "Sigma = %g" % (sigma)
# Plot the prism and the estimated center of mass
myv.figure()
myv.points([cm], size=300.)
myv.prisms(prisms, prop='density', opacity=0.5)
axes = myv.axes(
    myv.outline(extent=[-5000, 5000, -5000, 5000, 0, 5000]))
myv.wall_bottom(axes.axes.bounds, opacity=0.2)
myv.wall_north(axes.axes.bounds)
myv.show()
```

6.40 GravMag: Center of mass estimation using the first eigenvector of the gravity gradient tensor (flat slab model)

```python
"""
GravMag: Center of mass estimation using the first eigenvector of the gravity
gradient tensor (flat slab model)
"""
from fatiando.vis import mpl, myv
from fatiando import logger, mesher, gridder, utils, gravmag

log = logger.get()
log.info(logger.header())
log.info(__doc__)

# Generate some synthetic data
prisms = [mesher.Prism(-1000,1000,-1000,1000,500,700,{'density':1000})]
shape = (100, 100)
xp, yp, zp = gridder.regular((-5000, 5000, -5000, 5000), shape, z=-150)
noise = 1
tensor = [utils.contaminate(gravmag.prism.gxx(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gxy(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gxz(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gyy(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gyz(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gzz(xp, yp, zp, prisms), noise)]
# Plot the data
titles = ['gxx', 'gxy', 'gxz', 'gyy', 'gyz', 'gzz']
mpl.figure()
for i, title in enumerate(titles):
    mpl.subplot(3, 2, i + 1)
    mpl.title(title)
    mpl.axis('scaled')
    levels = mpl.contourf(yp, xp, tensor[i], shape, 10)
    mpl.contour(yp, xp, tensor[i], shape, levels)
    mpl.m2km()
mpl.show()
# Get the eigenvectors from the tensor data
eigenvals, eigenvecs = gravmag.tensor.eigen(tensor)
# Use the first eigenvector to estimate the center of mass
cm, sigma = gravmag.tensor.center_of_mass(xp, yp, zp, eigenvecs[0])
print "Sigma = %g" % (sigma)
# Plot the prism and the estimated center of mass
myv.figure()
myv.points([cm], size=300.)
myv.prisms(prisms, prop='density', opacity=0.5)
axes = myv.axes(
    myv.outline(extent=[-5000, 5000, -5000, 5000, 0, 5000]))
myv.wall_bottom(axes.axes.bounds, opacity=0.2)
myv.wall_north(axes.axes.bounds)
myv.show()
```

6.41 GravMag: Center of mass estimation using the first eigenvector of the gravity gradient tensor (large flat slab model)

```
"""
GravMag: Center of mass estimation using the first eigenvector of the gravity
gradient tensor (large flat slab model)
"""
from fatiando.vis import mpl, myv
from fatiando import logger, mesher, gridder, utils, gravmag

log = logger.get()
log.info(logger.header())
log.info(__doc__)

# Generate some synthetic data
prisms = [mesher.Prism(-3000,3000,-3000,3000,500,1000,{'density':1000})]
shape = (100, 100)
xp, yp, zp = gridder.regular((-5000, 5000, -5000, 5000), shape, z=-150)
noise = 1
tensor = [utils.contaminate(gravmag.prism.gxx(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gxy(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gxz(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gyy(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gyz(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gzz(xp, yp, zp, prisms), noise)]
# Plot the data
titles = ['gxx', 'gxy', 'gxz', 'gyy', 'gyz', 'gzz']
mpl.figure()
for i, title in enumerate(titles):
    mpl.subplot(3, 2, i + 1)
    mpl.title(title)
    mpl.axis('scaled')
    levels = mpl.contourf(yp, xp, tensor[i], shape, 10)
    mpl.contour(yp, xp, tensor[i], shape, levels)
    mpl.m2km()
mpl.show()
# Get the eigenvectors from the tensor data
eigenvals, eigenvecs = gravmag.tensor.eigen(tensor)
# Use the first eigenvector to estimate the center of mass
cm, sigma = gravmag.tensor.center_of_mass(xp, yp, zp, eigenvecs[0])
```

```
print "Sigma = %g" % (sigma)
# Plot the prism and the estimated center of mass
# This method is not good for finding the center of mass of a model like this
myv.figure()
myv.points([cm], size=300.)
myv.prisms(prisms, prop='density', opacity=0.5)
axes = myv.axes(
    myv.outline(extent=[-5000, 5000, -5000, 5000, 0, 5000]))
myv.wall_bottom(axes.axes.bounds, opacity=0.2)
myv.wall_north(axes.axes.bounds)
myv.show()
```

6.42 GravMag: Center of mass estimation using the first eigenvector of the gravity gradient tensor (2 sources with expanding windows)

```
"""
GravMag: Center of mass estimation using the first eigenvector of the gravity
gradient tensor (2 sources with expanding windows)
"""
from fatiando import logger, mesher, gridder, utils, gravmag
from fatiando.vis import mpl, myv

log = logger.get()
log.info(logger.header())
log.info(__doc__)

# Generate some synthetic data
prisms = [mesher.Prism(-2500,-500,-1000,1000,500,2500,{'density':1000}),
          mesher.Prism(500,2500,-1000,1000,500,2500,{'density':1000})]
shape = (100, 100)
area = (-5000, 5000, -5000, 5000)
xp, yp, zp = gridder.regular(area, shape, z=-150)
noise = 2
tensor = [utils.contaminate(gravmag.prism.gxx(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gxy(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gxz(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gyy(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gyz(xp, yp, zp, prisms), noise),
          utils.contaminate(gravmag.prism.gzz(xp, yp, zp, prisms), noise)]
# Get the eigenvectors from the tensor data
eigenvals, eigenvecs = gravmag.tensor.eigen(tensor)
# Plot the data
titles = ['gxx', 'gxy', 'gxz', 'gyy', 'gyz', 'gzz']
mpl.figure()
for i, title in enumerate(titles):
    mpl.subplot(3, 2, i + 1)
    mpl.title(title)
    mpl.axis('scaled')
    levels = mpl.contourf(yp, xp, tensor[i], shape, 10)
    mpl.contour(yp, xp, tensor[i], shape, levels)
    mpl.m2km()
mpl.show()

# Pick the centers of the expanding windows
# The number of final solutions will be the number of points picked
mpl.figure()
mpl.suptitle('Pick the centers of the expanding windows')
mpl.axis('scaled')
mpl.contourf(yp, xp, tensor[-1], shape, 50)
mpl.colorbar()
centers = mpl.pick_points(area, mpl.gca(), xy2ne=True)
cms = []
for center in centers:
    # Use the first eigenvector to estimate the center of mass
    cm, sigma = gravmag.tensor.center_of_mass(xp, yp, zp, eigenvecs[0],
        windows=100, wcenter=center)
    cms.append(cm)
    print "Sigma = %g" % (sigma)

# Plot the prism and the estimated center of mass
# It won't work well because we're using only a single window
myv.figure()
myv.points(cms, size=300.)
myv.prisms(prisms, prop='density', opacity=0.5)
axes = myv.axes(myv.outline(extent=[-5000, 5000, -5000, 5000, 0, 5000]))
myv.wall_bottom(axes.axes.bounds, opacity=0.2)
myv.wall_north(axes.axes.bounds)
myv.show()
```

6.43 GravMag: Calculate the gravity gradient tensor invariants

```
"""
GravMag: Calculate the gravity gradient tensor invariants
"""
import numpy
from fatiando import logger, mesher, gridder, gravmag
from fatiando.vis import mpl

log = logger.get()
log.info(logger.header())
log.info(__doc__)

log.info("Draw the polygons one by one")
area = [-10000, 10000, -10000, 10000]
dataarea = [-5000, 5000, -5000, 5000]
prisms = []
for depth in [5000, 5000]:
    fig = mpl.figure()
    mpl.axis('scaled')
    mpl.square(dataarea)
    for p in prisms:
        mpl.polygon(p, '.-k', xy2ne=True)
    mpl.set_area(area)
    prisms.append(
        mesher.PolygonalPrism(
            mpl.draw_polygon(area, fig.gca(), xy2ne=True),
            0, depth, {'density':500}))
# Calculate the effect
shape = (100, 100)
xp, yp, zp = gridder.regular(dataarea, shape, z=-500)
tensor = [
    gravmag.polyprism.gxx(xp, yp, zp, prisms),
    gravmag.polyprism.gxy(xp, yp, zp, prisms),
    gravmag.polyprism.gxz(xp, yp, zp, prisms),
    gravmag.polyprism.gyy(xp, yp, zp, prisms),
    gravmag.polyprism.gyz(xp, yp, zp, prisms),
    gravmag.polyprism.gzz(xp, yp, zp, prisms)]
# Calculate the 3 invariants
```

```python
invariants = gravmag.tensor.invariants(tensor)
data = tensor + invariants
# and plot it
mpl.figure()
mpl.axis('scaled')
mpl.suptitle("Tensor and invariants produced by prism model (Eotvos)")
titles = ['gxx', 'gxy', 'gxz', 'gyy', 'gyz', 'gzz', 'I1', 'I2', 'I']
for i in xrange(len(data)):
    mpl.subplot(3, 3, i + 1)
    mpl.title(titles[i])
    levels = 20
    if i == 8:
        levels = numpy.linspace(0, 1, levels)
    mpl.contourf(yp, xp, data[i], shape, levels)
    mpl.colorbar()
    for p in prisms:
        mpl.polygon(p, '.-k', xy2ne=True)
    mpl.set_area(dataarea)
    mpl.m2km()
mpl.show()
```

6.44 GravMag: Generate synthetic gradient tensor data from polygonal prisms

```python
"""
GravMag: Generate synthetic gradient tensor data from polygonal prisms
"""
from fatiando import logger, mesher, gridder, gravmag
from fatiando.vis import mpl, myv

log = logger.get()
log.info(logger.header())
log.info(__doc__)

log.info("Draw the polygons one by one")
bounds = [-10000, 10000, -10000, 10000, 0, 5000]
area = bounds[:4]
axis = mpl.figure().gca()
mpl.axis('scaled')
prisms = [
    mesher.PolygonalPrism(
        mpl.draw_polygon(area, axis, xy2ne=True),
        0, 1000, {'density':500})]
# Calculate the effect
shape = (100, 100)
xp, yp, zp = gridder.regular(area, shape, z=-500)
tensor = [
    gravmag.polyprism.gxx(xp, yp, zp, prisms),
    gravmag.polyprism.gxy(xp, yp, zp, prisms),
    gravmag.polyprism.gxz(xp, yp, zp, prisms),
    gravmag.polyprism.gyy(xp, yp, zp, prisms),
    gravmag.polyprism.gyz(xp, yp, zp, prisms),
    gravmag.polyprism.gzz(xp, yp, zp, prisms)]
# and plot it
titles = ['gxx', 'gxy', 'gxz', 'gyy', 'gyz', 'gzz']
mpl.figure()
mpl.axis('scaled')
mpl.suptitle("Gravity tensor produced by prism model (Eotvos)")
for i in xrange(len(tensor)):
    mpl.subplot(3, 2, i + 1)
    mpl.title(titles[i])
    mpl.contourf(yp, xp, tensor[i], shape, 20)
    mpl.colorbar()
    for p in prisms:
        mpl.polygon(p, '.-k', xy2ne=True)
    mpl.set_area(area)
    mpl.m2km()
mpl.show()
# Show the prisms
myv.figure()
myv.polyprisms(prisms, 'density')
myv.axes(myv.outline(bounds), ranges=[i*0.001 for i in bounds])
myv.wall_north(bounds)
myv.wall_bottom(bounds)
myv.show()
```

6.45 GravMag: Generate noise-corrupted gravity gradient tensor data

```
"""
GravMag: Generate noise-corrupted gravity gradient tensor data
"""
from fatiando import logger, mesher, gridder, gravmag, utils
from fatiando.vis import mpl

log = logger.get()
log.info(logger.header())
log.info(__doc__)

prisms = [mesher.Prism(-1000,1000,-1000,1000,0,2000,{'density':1000})]
shape = (100,100)
xp, yp, zp = gridder.regular((-5000, 5000, -5000, 5000), shape, z=-200)
components = [gravmag.prism.gxx, gravmag.prism.gxy, gravmag.prism.gxz,
              gravmag.prism.gyy, gravmag.prism.gyz, gravmag.prism.gzz]
log.info("Calculate the tensor components and contaminate with 5 Eotvos noise")
ftg = [utils.contaminate(comp(xp, yp, zp, prisms), 5.0) for comp in components]

log.info("Plotting...")
mpl.figure(figsize=(14,6))
mpl.suptitle("Contaminated FTG data")
names = ['gxx', 'gxy', 'gxz', 'gyy', 'gyz', 'gzz']
for i, data in enumerate(ftg):
    mpl.subplot(2,3,i+1)
    mpl.title(names[i])
    mpl.axis('scaled')
    levels = mpl.contourf(xp*0.001, yp*0.001, data, (100,100), 12)
    mpl.colorbar()
    mpl.contour(xp*0.001, yp*0.001, data, shape, levels, clabel=False)
mpl.show()
```

6.46 GravMag: Upward continuation of noisy gz data using the analytical formula

```
"""
GravMag: Upward continuation of noisy gz data using the analytical formula
"""
from fatiando import logger, mesher, gridder, utils, gravmag
```

```python
from fatiando.vis import mpl

log = logger.get()
log.info(logger.header())
log.info(__doc__)

log.info("Generating synthetic data")
prisms = [mesher.Prism(-3000,-2000,-3000,-2000,500,2000,{'density':1000}),
          mesher.Prism(-1000,1000,-1000,1000,0,2000,{'density':-800}),
          mesher.Prism(1000,3000,2000,3000,0,1000,{'density':500})]
area = (-5000, 5000, -5000, 5000)
shape = (50, 50)
z0 = -100
xp, yp, zp = gridder.regular(area, shape, z=z0)
gz = utils.contaminate(gravmag.prism.gz(xp, yp, zp, prisms), 0.5)

# Now do the upward continuation using the analytical formula
height = 2000
dims = gridder.spacing(area, shape)
gzcont = gravmag.transform.upcontinue(gz, height, xp, yp, dims)

log.info("Computing true values at new height")
gztrue = gravmag.prism.gz(xp, yp, zp - height, prisms)

log.info("Plotting")
mpl.figure(figsize=(14,6))
mpl.subplot(1, 2, 1)
mpl.title("Original")
mpl.axis('scaled')
mpl.contourf(xp, yp, gz, shape, 15)
mpl.contour(xp, yp, gz, shape, 15)
mpl.subplot(1, 2, 2)
mpl.title("Continued + true")
mpl.axis('scaled')
levels = mpl.contour(xp, yp, gzcont, shape, 12, color='b',
    label='Continued', style='dashed')
mpl.contour(xp, yp, gztrue, shape, levels, color='r', label='True',
    style='solid')
mpl.legend()
mpl.show()
```

6.47 Gridding: Cut a section from a grid

```python
"""
Gridding: Cut a section from a grid
"""
from fatiando import logger, gridder, utils
from fatiando.vis import mpl

log = logger.get()
log.info(logger.header())
log.info(__doc__)

# Generate some synthetic data on a regular grid
x, y = gridder.regular((-10, 10, -10, 10), (100,100))
# Using a 2D Gaussian
z = utils.gaussian2d(x, y, 1, 1)
subarea = [-2, 2, -3, 3]
subx, suby, subscalar = gridder.cut(x, y, [z], subarea)

```

```python
mpl.figure(figsize=(12, 5))
mpl.subplot(1, 2, 1)
mpl.title("Whole grid")
mpl.axis('scaled')
mpl.pcolor(x, y, z, (100,100))
mpl.square(subarea, 'k', linewidth=2, label='Cut this region')
mpl.legend(loc='lower left')
mpl.subplot(1, 2, 2)
mpl.title("Cut grid")
mpl.axis('scaled')
mpl.pcolor(subx, suby, subscalar[0], (40,60), interp=True)
mpl.show()
```

6.48 Gridding: Generate and plot irregular grids (scatter)

```python
"""
Gridding: Generate and plot irregular grids (scatter)
"""
from fatiando import logger, gridder, utils
from fatiando.vis import mpl

log = logger.get()
log.info(logger.header())
log.info(__doc__)

# Generate random points
x, y = gridder.scatter((-2, 2, -2, 2), n=200)
# And calculate a 2D Gaussian on these points
z = utils.gaussian2d(x, y, 1, 1)

mpl.axis('scaled')
mpl.title("Irregular grid")
mpl.plot(x, y, '.k', label='Grid points')
# Make a filled contour plot and tell the function to automatically interpolate
# the data on a 100x100 grid
mpl.contourf(x, y, z, (100, 100) , 50, interp=True)
mpl.colorbar()
mpl.legend(loc='lower right', numpoints=1)
mpl.show()
```

6.49 I/O: Fetch the CRUST2.0 model, convert it to tesseroids and calculate its gravity signal in parallel

```python
"""
I/O: Fetch the CRUST2.0 model, convert it to tesseroids and calculate its
gravity signal in parallel
"""
import time
from multiprocessing import Pool
from fatiando import gravmag, gridder, logger, utils, io
from fatiando.mesher import Tesseroid
from fatiando.vis import mpl, myv

log = logger.get()
log.info(logger.header())

# Get the data from their website and convert it to tesseroids
# Will download the archive and save it with the default name
log.info("Fetching CRUST2.0 model")
archive = io.fetch_crust2()
log.info("Converting to tesseroids")
model = io.crust2_to_tesseroids(archive)
log.info('  model size: %d' % (len(model)))

# Plot the tesseroid model
myv.figure(zdown=False)
myv.tesseroids(model, 'density')
myv.continents(linewidth=3)
myv.show()

# Make the computation grid
area = (-180, 180, -80, 80)
shape = (100, 100)
lons, lats, heights = gridder.regular(area, shape, z=250000)

# Divide the model into nproc slices and calculate them in parallel
log.info('Calculating...')
def calculate(chunk):
    return gravmag.tesseroid.gz(lons, lats, heights, chunk)
def split(model, nproc):
    chunksize = len(model)/nproc
    for i in xrange(nproc - 1):
        yield model[i*chunksize : (i + 1)*chunksize]
    yield model[(nproc - 1)*chunksize : ]
start = time.time()
nproc = 8
pool = Pool(processes=nproc)
gz = sum(pool.map(calculate, split(model, nproc)))
pool.close()
print "Time it took: %s" % (utils.sec2hms(time.time() - start))

log.info('Plotting...')
mpl.figure(figsize=(10, 4))
mpl.title('Crust gravity signal at 250km height')
bm = mpl.basemap(area, 'robin')
mpl.contourf(lons, lats, gz, shape, 35, basemap=bm)
cb = mpl.colorbar()
cb.set_label('mGal')
bm.drawcoastlines()
bm.drawmapboundary()
bm.drawparallels(range(-90, 90, 45), labels=[0, 1, 0, 0])
bm.drawmeridians(range(-180, 180, 60), labels=[0, 0, 0, 1])
mpl.show()
```

6.50 Meshing: Make and plot a 3D prism mesh

```
"""
Meshing: Make and plot a 3D prism mesh
"""
from fatiando import logger, mesher
from fatiando.vis import myv

log = logger.get()
log.info(logger.header())
log.info(__doc__)

mesh = mesher.PrismMesh(bounds=(-2, 2, -3, 3, 0, 1), shape=(4,4,4))
```

```python

myv.figure()
plot = myv.prisms(mesh)
axes = myv.axes(plot)
myv.show()
```

6.51 Meshing: Filter prisms from a 3D prism mesh based on their physical properties

```python
"""
Meshing: Filter prisms from a 3D prism mesh based on their physical properties
"""
from fatiando import logger, gridder, mesher
from fatiando.vis import myv

log = logger.get()
log.info(logger.header())
log.info(__doc__)

shape = (5, 20, 10)
bounds = (0, 100, 0, 200, 0, 50)
mesh = mesher.PrismMesh(bounds, shape)
# Fill the even prisms with 1 and odd with -1
def fill(i):
    if i%2 == 0:
        return 1
    return -1
mesh.addprop('density', [fill(i) for i in xrange(mesh.size)])

# Separate even and odd prisms
odd = mesher.vfilter(-1, 0, 'density', mesh)
even = mesher.vfilter(0, 1, 'density', mesh)

log.info("Showing solid ODD prisms and wireframe EVEN")
myv.figure()
myv.prisms(odd, prop='density', vmin=-1, vmax=1)
myv.prisms(even, prop='density', style='wireframe', vmin=-1, vmax=1)
myv.axes(myv.outline(bounds))
myv.show()
```

6.52 Meshing: Make and plot a 3D prism mesh with topography

```python
"""
Meshing: Make and plot a 3D prism mesh with topography
"""

from fatiando import logger, gridder, utils, mesher
from fatiando.vis import myv

log = logger.get()
log.info(logger.header())
log.info(__doc__)

x1, x2 = -100, 100
y1, y2 = -200, 200
bounds = (x1, x2, y1, y2, -200, 0)

```

```python
log.info("Generating synthetic topography")
x, y = gridder.regular((x1, x2, y1, y2), (50,50))
height = (100 +
          -50*utils.gaussian2d(x, y, 100, 200, x0=-50, y0=-100, angle=-60) +
          100*utils.gaussian2d(x, y, 50, 100, x0=80, y0=170))

log.info("Generating the prism mesh")
mesh = mesher.PrismMesh(bounds, (20,40,20))
mesh.carvetopo(x, y, height)

log.info("Plotting")
myv.figure()
myv.prisms(mesh)
myv.axes(myv.outline(bounds), fmt='%.0f')
myv.wall_north(bounds)
myv.show()
```

6.53 Meshing: Make a 3D prism mesh with depth-varying density

```python
"""
Meshing: Make a 3D prism mesh with depth-varying density
"""
from fatiando import logger, gridder, mesher
from fatiando.vis import myv

log = logger.get()
log.info(logger.header())
log.info(__doc__)

shape = (10, 20, 10)
nz, ny, nx = shape
mesh = mesher.PrismMesh((0, 100, 0, 200, 0, 50), shape)
def fill(i):
    k = i/(nx*ny)
    return k
mesh.addprop('density', [fill(i) for i in xrange(mesh.size)])

myv.figure()
myv.prisms(mesh, prop='density')
myv.axes(myv.outline(), fmt='%.0f')
myv.show()
```

6.54 Meshing: Generate a topographic model using prisms and calculate its gravity anomaly

```python
"""
Meshing: Generate a topographic model using prisms and calculate its gravity
anomaly
"""
from fatiando import logger, gridder, utils, mesher, gravmag
from fatiando.vis import myv, mpl

log = logger.get()
log.info(logger.header())
log.info(__doc__)

log.info("Generating synthetic topography")
```

```python
area = (-150, 150, -300, 300)
shape = (30, 15)
x, y = gridder.regular(area, shape)
height = (-80*utils.gaussian2d(x, y, 100, 200, x0=-50, y0=-100, angle=-60) +
          200*utils.gaussian2d(x, y, 50, 100, x0=80, y0=170))

log.info("Generating the 3D relief")
nodes = (x, y, -1*height)
relief = mesher.PrismRelief(0, gridder.spacing(area,shape), nodes)
relief.addprop('density', (2670 for i in xrange(relief.size)))

log.info("Calculating gz effect")
gridarea = (-80, 80, -220, 220)
gridshape = (100, 100)
xp, yp, zp = gridder.regular(gridarea, gridshape, z=-200)
gz = gravmag.prism.gz(xp, yp, zp, relief)

log.info("Plotting")
mpl.figure(figsize=(10,7))
mpl.subplot(1, 2, 1)
mpl.title("Synthetic topography")
mpl.axis('scaled')
mpl.pcolor(x, y, height, shape)
cb = mpl.colorbar()
cb.set_label("meters")
mpl.square(gridarea, label='Computation grid')
mpl.legend()
mpl.subplot(1, 2, 2)
mpl.title("Topographic effect")
mpl.axis('scaled')
mpl.pcolor(xp, yp, gz, gridshape)
cb = mpl.colorbar()
cb.set_label("mGal")
mpl.show()

myv.figure()
myv.prisms(relief, prop='density')
axes = myv.axes(myv.outline())
myv.wall_bottom(axes.axes.bounds, opacity=0.2)
myv.wall_north(axes.axes.bounds)
myv.show()
```

6.55 Meshing: Generate a 3D prism model of the topography

```python
"""
Meshing: Generate a 3D prism model of the topography
"""
from fatiando import logger, gridder, utils, mesher
from fatiando.vis import myv

log = logger.get()
log.info(logger.header())
log.info(__doc__)

log.info("Generating synthetic topography")
area = (-150, 150, -300, 300)
shape = (100, 50)
x, y = gridder.regular(area, shape)
height = (-80*utils.gaussian2d(x, y, 100, 200, x0=-50, y0=-100, angle=-60) +
          100*utils.gaussian2d(x, y, 50, 100, x0=80, y0=170))
```

```

log.info("Generating the 3D relief")
nodes = (x, y, -1*height) # -1 is to convert height to z coordinate
reference = 0 # z coordinate of the reference surface
relief = mesher.PrismRelief(reference, gridder.spacing(area, shape), nodes)
relief.addprop('density', (2670 for i in xrange(relief.size)))

log.info("Plotting")
myv.figure()
myv.prisms(relief, prop='density', edges=False)
axes = myv.axes(myv.outline())
myv.wall_bottom(axes.axes.bounds, opacity=0.2)
myv.wall_north(axes.axes.bounds)
myv.show()
```

6.56 Meshing: Generate a SquareMesh and get the physical properties from an image

```
"""
Meshing: Generate a SquareMesh and get the physical properties from an image
"""
import urllib
from fatiando import logger, mesher
from fatiando.vis import mpl

log = logger.get()
log.info(logger.header())
log.info(__doc__)

# Make a square mesh
mesh = mesher.SquareMesh((0, 5000, 0, 5000), (150, 150))
# Fetch the image from the online docs
urllib.urlretrieve(
    'http://fatiando.readthedocs.org/en/latest/_static/logo.png', 'logo.png')
# Load the image as the P wave velocity (vp) property of the mesh
mesh.img2prop('logo.png', 5000, 10000, 'vp')

mpl.figure()
mpl.title('P wave velocity model of the Earth')
mpl.squaremesh(mesh, prop='vp')
cb = mpl.colorbar()
cb.set_label("Vp (km/s)")
mpl.show()
```

6.57 Meshing: Make and plot a tesseroid mesh

```
"""
Meshing: Make and plot a tesseroid mesh
"""
from fatiando import mesher
from fatiando.vis import myv

mesh = mesher.TesseroidMesh((-60, 60, -30, 30, 100000, -500000), (10, 10, 10))

myv.figure(zdown=False)
myv.tesseroids(mesh)
myv.earth(opacity=0.3)
myv.continents()
myv.meridians(range(-180, 180, 30))
myv.parallels(range(-90, 90, 30))
myv.show()
```

6.58 Meshing: Make and plot a tesseroid mesh with topography

```
"""
Meshing: Make and plot a tesseroid mesh with topography
"""

from fatiando import logger, gridder, utils, mesher
from fatiando.vis import myv

log = logger.get()
log.info(logger.header())
log.info(__doc__)

w, e = -2, 2
s, n = -2, 2
bounds = (w, e, s, n, 500000, 0)

log.info("Generating synthetic topography")
x, y = gridder.regular((w, e, s, n), (50, 50))
height = (250000 +
          -100000*utils.gaussian2d(x, y, 1, 5, x0=-1, y0=-1, angle=-60) +
          250000*utils.gaussian2d(x, y, 1, 1, x0=0.8, y0=1.7))

mesh = mesher.TesseroidMesh(bounds, (20, 50, 50))
mesh.carvetopo(x, y, height)

scene = myv.figure(zdown=False)
myv.tesseroids(mesh)
myv.earth(opacity=0.3)
myv.continents()
scene.scene.camera.position = [21592740.078245595, 22628783.944262519, -28903782.916664094]
scene.scene.camera.focal_point = [5405474.2152075395, -1711034.715136874, 2155879.3486608281]
scene.scene.camera.view_angle = 1.6492674416639987
scene.scene.camera.view_up = [0.91713422625547714, -0.1284658947859818, 0.37730799740742887]
scene.scene.camera.clipping_range = [20169510.286021926, 69721043.718536735]
scene.scene.camera.compute_view_plane_normal()
scene.scene.render()
myv.show()
```

6.59 Seismic: 2D epicenter estimation assuming a homogeneous and flat Earth

```
"""
Seismic: 2D epicenter estimation assuming a homogeneous and flat Earth
"""
import sys
import numpy
from fatiando import logger, mesher, seismic, utils, gridder, vis, inversion

log = logger.get()
log.info(logger.header())
log.info(__doc__)
```

```python

area = (0, 10, 0, 10)
vp, vs = 2, 1
model = [mesher.Square(area, props={'vp':vp, 'vs':vs})]

log.info("Choose the location of the receivers")
vis.mpl.figure()
ax = vis.mpl.subplot(1, 1, 1)
vis.mpl.axis('scaled')
vis.mpl.suptitle("Choose the location of the receivers")
rec_points = vis.mpl.pick_points(area, ax, marker='^', color='r')

log.info("Choose the location of the receivers")
vis.mpl.figure()
ax = vis.mpl.subplot(1, 1, 1)
vis.mpl.axis('scaled')
vis.mpl.suptitle("Choose the location of the source")
vis.mpl.points(rec_points, '^r')
src = vis.mpl.pick_points(area, ax, marker='*', color='y')
if len(src) > 1:
    log.error("Don't be greedy! Pick only one point as the source")
    sys.exit()

log.info("Generating synthetic travel-time data")
srcs, recs = utils.connect_points(src, rec_points)
ptime = seismic.ttime2d.straight(model, 'vp', srcs, recs)
stime = seismic.ttime2d.straight(model, 'vs', srcs, recs)
ttresiduals, error = utils.contaminate(stime - ptime, 0.10, percent=True,
                                        return_stddev=True)

log.info("Will solve the inverse problem using the Levenberg-Marquardt method")
solver = inversion.gradient.levmarq(initial=(0, 0), maxit=1000, tol=10**(-3))
result = seismic.epic2d.homogeneous(ttresiduals, recs, vp, vs, solver)
estimate, residuals = result
predicted = ttresiduals - residuals

log.info("Build a map of the goal function")
shape = (100, 100)
xs, ys = gridder.regular(area, shape)
goals = seismic.epic2d.mapgoal(xs, ys, ttresiduals, recs, vp, vs)

log.info("Plotting")
vis.mpl.figure(figsize=(10,4))
vis.mpl.subplot(1, 2, 1)
vis.mpl.title('Epicenter + %d recording stations' % (len(recs)))
vis.mpl.axis('scaled')
vis.mpl.contourf(xs, ys, goals, shape, 50)
vis.mpl.points(src, '*y', label="True")
vis.mpl.points(recs, '^r', label="Stations")
vis.mpl.points([estimate], '*g', label="Estimate")
vis.mpl.set_area(area)
vis.mpl.legend(loc='lower right', shadow=True, numpoints=1, prop={'size':12})
vis.mpl.xlabel("X")
vis.mpl.ylabel("Y")
ax = vis.mpl.subplot(1, 2, 2)
vis.mpl.title('Travel-time residuals + 10% error')
s = numpy.arange(len(ttresiduals)) + 1
width = 0.3
vis.mpl.bar(s - width, ttresiduals, width, color='g', label="Observed",
            yerr=error)
vis.mpl.bar(s, predicted, width, color='r', label="Predicted")
ax.set_xticks(s)
vis.mpl.legend(loc='upper right', shadow=True, prop={'size':12})
vis.mpl.xlabel("Station number")
vis.mpl.ylabel("Travel-time residual")
vis.mpl.show()
```

6.60 Seismic: 2D epicenter estimation on a flat Earth using equality constraints

```
"""
Seismic: 2D epicenter estimation on a flat Earth using equality constraints
"""
import sys
import numpy
from fatiando import logger, mesher, seismic, utils, gridder, vis, inversion

log = logger.get()
log.info(logger.header())
log.info(__doc__)

log.info("The data are noisy and receiver locations are bad.")
log.info("So use a bit of regularization.")

area = (0, 10, 0, 10)
vp, vs = 2, 1
model = [mesher.Square(area, props={'vp':vp, 'vs':vs})]

log.info("Generating synthetic travel-time data")
src = (8, 7)
stations = 10
srcs, recs = utils.connect_points([src], [(4, 6), (5, 5.9), (6, 6)])
ptime = seismic.ttime2d.straight(model, 'vp', srcs, recs)
stime = seismic.ttime2d.straight(model, 'vs', srcs, recs)
error_level = 0.05
ttr_true = stime - ptime
ttr, error = utils.contaminate(ttr_true, error_level, percent=True,
                               return_stddev=True)

log.info("Choose the initial estimate for the gradient solvers")
vis.mpl.figure()
ax = vis.mpl.subplot(1, 1, 1)
vis.mpl.axis('scaled')
vis.mpl.suptitle("Choose the initial estimate for the gradient solvers")
vis.mpl.points(recs, '^r')
vis.mpl.points(srcs, '*y')
initial = vis.mpl.pick_points(area, ax, marker='*', color='k')
if len(initial) > 1:
    log.error("Don't be greedy! Pick only one initial estimate")
    sys.exit()
initial = initial[0]

ref = {'y':7}
equality = 0.1
log.info("Will solve the inverse problem using Newton's method")
log.info("and with equality constaints for stability")
nsolver = inversion.gradient.newton(initial)
newton = [initial]
iterator = seismic.epic2d.homogeneous(ttr, recs, vp, vs, nsolver,
    equality=equality, ref=ref, iterate=True)
for e, r in iterator:
    newton.append(e)
newton_predicted = ttr - r
```

```python

log.info("and the Steepest Descent method")
sdsolver = inversion.gradient.steepest(initial, step=0.1)
steepest = [initial]
iterator = seismic.epic2d.homogeneous(ttr, recs, vp, vs, sdsolver,
    equality=equality, ref=ref, iterate=True)
for e, r in iterator:
    steepest.append(e)
steepest_predicted = ttr - r

log.info("... and also the Levenberg-Marquardt algorithm for comparison")
lmsolver = inversion.gradient.levmarq(initial)
levmarq = [initial]
iterator = seismic.epic2d.homogeneous(ttr, recs, vp, vs, lmsolver,
    equality=equality, ref=ref, iterate=True)
for e, r in iterator:
    levmarq.append(e)
levmarq_predicted = ttr - r

log.info("Build a map of the goal function")
shape = (100, 100)
xs, ys = gridder.regular(area, shape)
goals = seismic.epic2d.mapgoal(xs, ys, ttr, recs, vp, vs, equality=equality,
    ref=ref)

log.info("Plotting")
vis.mpl.figure(figsize=(14, 4))
vis.mpl.subplot(1, 3, 1)
vis.mpl.title('Epicenter + recording stations')
vis.mpl.axis('scaled')
vis.mpl.contourf(xs, ys, goals, shape, 50)
vis.mpl.points(recs, '^r', label="Stations")
vis.mpl.points(newton, '.-r', size=5, label="Newton")
vis.mpl.points([newton[-1]], '*r')
vis.mpl.points(levmarq, '.-k', size=5, label="Lev-Marq")
vis.mpl.points([levmarq[-1]], '*k')
vis.mpl.points(steepest, '.-m', size=5, label="Steepest")
vis.mpl.points([steepest[-1]], '*m')
vis.mpl.points([src], '*y', label="True")
vis.mpl.set_area(area)
vis.mpl.legend(loc='lower left', shadow=True, numpoints=1, prop={'size':10})
vis.mpl.xlabel("X")
vis.mpl.ylabel("Y")
ax = vis.mpl.subplot(1, 3, 2)
vis.mpl.title('Travel-time residuals + %g%s error' % (100.*error_level, '%'))
s = numpy.arange(len(ttr)) + 1
width = 0.2
vis.mpl.bar(s - 2*width, ttr, width, color='y', label="Observed", yerr=error)
vis.mpl.bar(s - width, newton_predicted, width, color='r', label="Newton")
vis.mpl.bar(s, levmarq_predicted, width, color='k', label="Lev-Marq")
vis.mpl.bar(s + 1*width, steepest_predicted, width, color='m', label="Steepest")
vis.mpl.plot(s - 1.5*width, ttr_true, '^-y', linewidth=2, label="Noise-free")
ax.set_xticks(s)
vis.mpl.legend(loc='lower right', shadow=True, prop={'size':10})
vis.mpl.xlabel("Station number")
vis.mpl.ylabel("Travel-time residual")
ax = vis.mpl.subplot(1, 3, 3)
vis.mpl.title('Number of iterations')
width = 0.5
vis.mpl.bar(1, len(newton), width, color='r', label="Newton")
vis.mpl.bar(2, len(levmarq), width, color='k', label="Lev-Marq")
vis.mpl.bar(3, len(steepest), width, color='m', label="Steepest")
ax.set_xticks([])
```

```
vis.mpl.grid()
vis.mpl.legend(loc='upper left', shadow=True, prop={'size':10})
vis.mpl.show()
```

6.61 Seismic: Show steps taken by different algorithms for 2D epicenter estimation on a flat Earth

```
"""
Seismic: Show steps taken by different algorithms for 2D epicenter estimation
on a flat Earth
"""
import sys
import numpy
from fatiando import logger, mesher, seismic, utils, gridder, vis, inversion

log = logger.get()
log.info(logger.header())
log.info(__doc__)

area = (0, 10, 0, 10)
vp, vs = 2, 1
model = [mesher.Square(area, props={'vp':vp, 'vs':vs})]

log.info("Choose the location of the receivers")
vis.mpl.figure()
ax = vis.mpl.subplot(1, 1, 1)
vis.mpl.axis('scaled')
vis.mpl.suptitle("Choose the location of the receivers")
rec_points = vis.mpl.pick_points(area, ax, marker='^', color='r')

log.info("Choose the location of the receivers")
vis.mpl.figure()
ax = vis.mpl.subplot(1, 1, 1)
vis.mpl.axis('scaled')
vis.mpl.suptitle("Choose the location of the source")
vis.mpl.points(rec_points, '^r')
src = vis.mpl.pick_points(area, ax, marker='*', color='y')
if len(src) > 1:
    log.error("Don't be greedy! Pick only one point as the source")
    sys.exit()

log.info("Generating synthetic travel-time data")
srcs, recs = utils.connect_points(src, rec_points)
ptime = seismic.ttime2d.straight(model, 'vp', srcs, recs)
stime = seismic.ttime2d.straight(model, 'vs', srcs, recs)
error_level = 0.1
ttr_true = stime - ptime
ttr, error = utils.contaminate(ttr_true, error_level, percent=True,
                                  return_stddev=True)

log.info("Choose the initial estimate for the gradient solvers")
vis.mpl.figure()
ax = vis.mpl.subplot(1, 1, 1)
vis.mpl.axis('scaled')
vis.mpl.suptitle("Choose the initial estimate for the gradient solvers")
vis.mpl.points(rec_points, '^r')
vis.mpl.points(src, '*y')
initial = vis.mpl.pick_points(area, ax, marker='*', color='k')
if len(initial) > 1:
    log.error("Don't be greedy! Pick only one initial estimate")
    sys.exit()
initial = initial[0]

log.info("Will solve the inverse problem using Newton's method")
nsolver = inversion.gradient.newton(initial)
newton = [initial]
iterator = seismic.epic2d.homogeneous(ttr, recs, vp, vs, nsolver, iterate=True)
for e, r in iterator:
    newton.append(e)
newton_predicted = ttr - r

log.info("and the Steepest Descent method")
sdsolver = inversion.gradient.steepest(initial)
steepest = [initial]
iterator = seismic.epic2d.homogeneous(ttr, recs, vp, vs, sdsolver, iterate=True)
for e, r in iterator:
    steepest.append(e)
steepest_predicted = ttr - r

log.info("... and also the Levenberg-Marquardt algorithm for comparison")
lmsolver = inversion.gradient.levmarq(initial)
levmarq = [initial]
iterator = seismic.epic2d.homogeneous(ttr, recs, vp, vs, lmsolver, iterate=True)
for e, r in iterator:
    levmarq.append(e)
levmarq_predicted = ttr - r

log.info("Build a map of the goal function")
shape = (100, 100)
xs, ys = gridder.regular(area, shape)
goals = seismic.epic2d.mapgoal(xs, ys, ttr, recs, vp, vs)

log.info("Plotting")
vis.mpl.figure(figsize=(14,4))
vis.mpl.subplot(1, 3, 1)
vis.mpl.title('Epicenter + %d recording stations' % (len(rec_points)))
vis.mpl.axis('scaled')
vis.mpl.contourf(xs, ys, goals, shape, 50)
vis.mpl.points(recs, '^r', label="Stations")
vis.mpl.points(newton, '.-r', size=5, label="Newton")
vis.mpl.points([newton[-1]], '*r')
vis.mpl.points(levmarq, '.-k', size=5, label="Lev-Marq")
vis.mpl.points([levmarq[-1]], '*k')
vis.mpl.points(steepest, '.-m', size=5, label="Steepest")
vis.mpl.points([steepest[-1]], '*m')
vis.mpl.points(src, '*y', label="True")
vis.mpl.set_area(area)
vis.mpl.legend(loc='upper left', shadow=True, numpoints=1, prop={'size':10})
vis.mpl.xlabel("X")
vis.mpl.ylabel("Y")
ax = vis.mpl.subplot(1, 3, 2)
vis.mpl.title('Travel-time residuals + %g%s error' % (100.*error_level, '%'))
s = numpy.arange(len(ttr)) + 1
width = 0.2
vis.mpl.bar(s - 2*width, ttr, width, color='y', label="Observed", yerr=error)
vis.mpl.bar(s - width, newton_predicted, width, color='r', label="Newton")
vis.mpl.bar(s, levmarq_predicted, width, color='k', label="Lev-Marq")
vis.mpl.bar(s + 1*width, steepest_predicted, width, color='m', label="Steepest")
vis.mpl.plot(s - 1.5*width, ttr_true, '^-y', linewidth=2, label="Noise-free")
ax.set_xticks(s)
vis.mpl.legend(loc='lower right', shadow=True, prop={'size':10})
vis.mpl.ylim(0, 4.5)
vis.mpl.xlabel("Station number")
vis.mpl.ylabel("Travel-time residual")
ax = vis.mpl.subplot(1, 3, 3)
vis.mpl.title('Number of iterations')
width = 0.5
vis.mpl.bar(1, len(newton), width, color='r', label="Newton")
vis.mpl.bar(2, len(levmarq), width, color='k', label="Lev-Marq")
vis.mpl.bar(3, len(steepest), width, color='m', label="Steepest")
ax.set_xticks([])
vis.mpl.grid()
vis.mpl.legend(loc='lower right', shadow=True, prop={'size':10})
vis.mpl.show()
```

6.62 Seismic: Invert vertical seismic profile (VSP) traveltimes for the velocity of the layers

```
"""
Seismic: Invert vertical seismic profile (VSP) traveltimes for the velocity of the
layers
"""
import numpy
from fatiando import logger, utils, seismic, vis

log = logger.get()
log.info(logger.header())
log.info(__doc__)

log.info("Draw the layered model")
# The limits in velocity and depths
area = (0, 10000, 0, 100)
vmin, vmax, zmin, zmax = area
figure = vis.mpl.figure()
vis.mpl.xlabel("Velocity (m/s)")
vis.mpl.ylabel("Depth (m)")
thickness, velocity = vis.mpl.draw_layers(area, figure.gca())

log.info("Generating synthetic data")
zp = numpy.arange(zmin + 0.5, zmax, 0.5)
tts, error = utils.contaminate(
    seismic.profile.vertical(thickness, velocity, zp),
    0.02, percent=True, return_stddev=True)

log.info("Preparing for the inversion")
damping = 100.
estimates = []
for i in xrange(30):
    p, r = seismic.profile.ivertical(utils.contaminate(tts, error), zp,
        thickness, damping=damping)
    estimates.append(1./p)
estimate = utils.vecmean(estimates)
predicted = seismic.profile.vertical(thickness, estimate, zp)

log.info("Plotting results...")
vis.mpl.figure(figsize=(12,5))
vis.mpl.subplot(1, 2, 1)
vis.mpl.grid()
vis.mpl.title("Vertical seismic profile")
vis.mpl.plot(tts, zp, 'ok', label='Observed')
vis.mpl.plot(predicted, zp, '-r', linewidth=3, label='Predicted')
vis.mpl.legend(loc='upper right', numpoints=1)
vis.mpl.xlabel("Travel-time (s)")
vis.mpl.ylabel("Z (m)")
vis.mpl.ylim(sum(thickness), 0)
vis.mpl.subplot(1, 2, 2)
vis.mpl.grid()
vis.mpl.title("Velocity profile")
for p in estimates:
    vis.mpl.layers(thickness, p, '-r', linewidth=2, alpha=0.2)
vis.mpl.layers(thickness, estimate, 'o-k', linewidth=2, label='Mean estimate')
vis.mpl.layers(thickness, velocity, '--b', linewidth=2, label='True')
vis.mpl.ylim(zmax, zmin)
vis.mpl.xlim(vmin, vmax)
leg = vis.mpl.legend(loc='upper right', numpoints=1)
leg.get_frame().set_alpha(0.5)
vis.mpl.xlabel("Velocity (m/s)")
vis.mpl.ylabel("Z (m)")
vis.mpl.show()
```

6.63 Seismic: Interactive forward modeling of 1D vertical seismic profile (VSP) data in layered media

```
"""
Seismic: Interactive forward modeling of 1D vertical seismic profile (VSP) data in
layered media
"""
import numpy
from fatiando.gui.simple import Lasagne

thickness = [10, 20, 5, 10, 45, 80]
zp = numpy.arange(0.5, sum(thickness), 0.5)
vmin, vmax = 500, 10000
app = Lasagne(thickness, zp, vmin, vmax)
app.run()
```

6.64 Seismic: Invert vertical seismic profile (VSP) traveltimes using sharpness regularization

```
"""
Seismic: Invert vertical seismic profile (VSP) traveltimes using sharpness
regularization
"""
import numpy
from fatiando import logger, utils, seismic, vis

log = logger.get()
log.info(logger.header())
log.info(__doc__)

log.info("Generating synthetic data")
thickness = [10, 20, 10, 30, 40, 60]
velocity = [2000, 1000, 5000, 1000, 3000, 6000]
zp = numpy.arange(1, sum(thickness), 1, dtype='f')
tts, error = utils.contaminate(
    seismic.profile.vertical(thickness, velocity, zp),
    0.02, percent=True, return_stddev=True)

log.info("Preparing for the inversion using 5 m thick layers")
thick = 10.
```

```
mesh = [thick]*int(sum(thickness)/thick)
sharp = 0.00015
beta = 10.**(-12)
estimates = []
for i in xrange(30):
    p, r = seismic.profile.ivertical(utils.contaminate(tts, error),
        zp, mesh, sharp=sharp, beta=beta)
    estimates.append(1./p)
estimate = utils.vecmean(estimates)
predicted = seismic.profile.vertical(mesh, estimate, zp)

log.info("Plotting results...")
vis.mpl.figure(figsize=(12,5))
vis.mpl.subplot(1, 2, 1)
vis.mpl.grid()
vis.mpl.title("Vertical seismic profile")
vis.mpl.plot(tts, zp, 'ok', label='Observed')
vis.mpl.plot(predicted, zp, '-r', linewidth=3, label='Predicted')
vis.mpl.legend(loc='upper right', numpoints=1)
vis.mpl.xlabel("Travel-time (s)")
vis.mpl.ylabel("Z (m)")
vis.mpl.ylim(sum(thickness), 0)
vis.mpl.subplot(1, 2, 2)
vis.mpl.grid()
vis.mpl.title("True velocity + sharp estimate")
for p in estimates:
    vis.mpl.layers(mesh, p, '-r', linewidth=2, alpha=0.2)
vis.mpl.layers(mesh, estimate, '.-k', linewidth=2, label='Mean estimate')
vis.mpl.layers(thickness, velocity, '--b', linewidth=2, label='True')
vis.mpl.ylim(sum(thickness), 0)
vis.mpl.xlim(0, 10000)
vis.mpl.legend(loc='upper right', numpoints=1)
vis.mpl.xlabel("Velocity (m/s)")
vis.mpl.ylabel("Z (m)")
vis.mpl.show()
```

6.65 Seismic: Invert vertical seismic profile (VSP) traveltimes using smoothness regularization

```
"""
Seismic: Invert vertical seismic profile (VSP) traveltimes using smoothness
regularization
"""
import numpy
from fatiando import logger, utils, seismic, vis

log = logger.get()
log.info(logger.header())
log.info(__doc__)

log.info("Generating synthetic data")
thickness = [10, 20, 10, 30, 40, 60]
velocity = [2000, 1000, 5000, 1000, 2500, 6000]
zp = numpy.arange(1., sum(thickness), 1., dtype='f')
tts, error = utils.contaminate(
    seismic.profile.vertical(thickness, velocity, zp),
    0.02, percent=True, return_stddev=True)

log.info("Preparing for the inversion using 5 m thick layers")
thick = 10.
```

```
mesh = [thick]*int(sum(thickness)/thick)
smooth = 50.
estimates = []
for i in xrange(30):
    p, r = seismic.profile.ivertical(utils.contaminate(tts, error), zp, mesh,
        smooth=smooth)
    estimates.append(1./p)
estimate = utils.vecmean(estimates)
predicted = seismic.profile.vertical(mesh, estimate, zp)

log.info("Plotting results...")
vis.mpl.figure(figsize=(12,5))
vis.mpl.subplot(1, 2, 1)
vis.mpl.grid()
vis.mpl.title("Vertical seismic profile")
vis.mpl.plot(tts, zp, 'ok', label='Observed')
vis.mpl.plot(predicted, zp, '-r', linewidth=3, label='Predicted')
vis.mpl.legend(loc='upper right', numpoints=1)
vis.mpl.xlabel("Travel-time (s)")
vis.mpl.ylabel("Z (m)")
vis.mpl.ylim(sum(thickness), 0)
vis.mpl.subplot(1, 2, 2)
vis.mpl.grid()
vis.mpl.title("True velocity + smooth estimate")
for p in estimates:
    vis.mpl.layers(mesh, p, '-r', linewidth=2, alpha=0.2)
vis.mpl.layers(mesh, estimate, '.-k', linewidth=2, label='Mean estimate')
vis.mpl.layers(thickness, velocity, '--b', linewidth=2, label='True')
vis.mpl.ylim(sum(thickness), 0)
vis.mpl.xlim(0, 10000)
vis.mpl.legend(loc='upper right', numpoints=1)
vis.mpl.xlabel("Velocity (m/s)")
vis.mpl.ylabel("Z (m)")
vis.mpl.show()
```

6.66 Seismic: 2D straight-ray tomography using damping regularization

```
"""
Seismic: 2D straight-ray tomography using damping regularization

Uses synthetic data and a model generated from an image file.
"""
import urllib
import time
import numpy
from fatiando import logger, mesher, utils, seismic, vis

log = logger.get()
log.info(logger.header())
log.info(__doc__)

area = (0, 500000, 0, 500000)
shape = (30, 30)
model = mesher.SquareMesh(area, shape)
# Fetch the image from the online docs
urllib.urlretrieve(
    'http://fatiando.readthedocs.org/en/latest/_static/logo.png', 'logo.png')
model.img2prop('logo.png', 4000, 10000, 'vp')

```

```
# Make some travel time data and add noise
log.info("Generating synthetic travel-time data")
src_loc = utils.random_points(area, 80)
rec_loc = utils.circular_points(area, 30, random=True)
srcs, recs = utils.connect_points(src_loc, rec_loc)
start = time.time()
tts = seismic.ttime2d.straight(model, 'vp', srcs, recs, par=True)
log.info("  time: %s" % (utils.sec2hms(time.time() - start)))
tts, error = utils.contaminate(tts, 0.01, percent=True, return_stddev=True)
# Make the mesh
mesh = mesher.SquareMesh(area, shape)
# and run the inversion
estimate, residuals = seismic.srtomo.run(tts, srcs, recs, mesh, damping=10**9)
# Convert the slowness estimate to velocities and add it the mesh
mesh.addprop('vp', seismic.srtomo.slowness2vel(estimate))

# Calculate and print the standard deviation of the residuals
# it should be close to the data error if the inversion was able to fit the data
log.info("Assumed error: %g" % (error))
log.info("Standard deviation of residuals: %g" % (numpy.std(residuals)))

vis.mpl.figure(figsize=(14, 5))
vis.mpl.subplot(1, 2, 1)
vis.mpl.axis('scaled')
vis.mpl.title('Vp synthetic model of the Earth')
vis.mpl.squaremesh(model, prop='vp', cmap=vis.mpl.cm.seismic)
cb = vis.mpl.colorbar()
cb.set_label('Velocity')
vis.mpl.points(src_loc, '*y', label="Sources")
vis.mpl.points(rec_loc, '^r', label="Receivers")
vis.mpl.legend(loc='lower left', shadow=True, numpoints=1, prop={'size':10})
vis.mpl.m2km()
vis.mpl.subplot(1, 2, 2)
vis.mpl.axis('scaled')
vis.mpl.title('Tomography result (damped)')
vis.mpl.squaremesh(mesh, prop='vp', vmin=4000, vmax=10000,
    cmap=vis.mpl.cm.seismic)
cb = vis.mpl.colorbar()
cb.set_label('Velocity')
vis.mpl.m2km()
vis.mpl.figure()
vis.mpl.grid()
vis.mpl.title('Residuals (data with %.4f s error)' % (error))
vis.mpl.hist(residuals, color='gray', bins=10)
vis.mpl.xlabel("seconds")
vis.mpl.show()
```

6.67 Seismic: 2D straight-ray tomography using sharpness (total variation) regularization

```
"""
Seismic: 2D straight-ray tomography using sharpness (total variation)
regularization

Uses synthetic data and a model generated from an image file.
"""
import urllib
import time
from os import path
import numpy
```

```
from fatiando import logger, mesher, utils, seismic, vis

log = logger.get()
log.info(logger.header())
log.info(__doc__)

area = (0, 100000, 0, 100000)
shape = (30, 30)
model = mesher.SquareMesh(area, shape)
# Fetch the image from the online docs
urllib.urlretrieve(
    'http://fatiando.readthedocs.org/en/latest/_static/logo.png', 'logo.png')
vmin, vmax = 4000, 10000
model.img2prop('logo.png', vmin, vmax, 'vp')

# Make some travel time data and add noise
log.info("Generating synthetic travel-time data")
src_loc = utils.random_points(area, 80)
rec_loc = utils.circular_points(area, 30, random=True)
srcs, recs = utils.connect_points(src_loc, rec_loc)
start = time.time()
tts = seismic.ttime2d.straight(model, 'vp', srcs, recs, par=True)
log.info("  time: %s" % (utils.sec2hms(time.time() - start)))
tts, error = utils.contaminate(tts, 0.01, percent=True, return_stddev=True)
# Make the mesh
mesh = mesher.SquareMesh(area, shape)
# and run the inversion
estimate, residuals = seismic.srtomo.run(tts, srcs, recs, mesh, sharp=5*10**5)
# Convert the slowness estimate to velocities and add it the mesh
mesh.addprop('vp', seismic.srtomo.slowness2vel(estimate))

# Calculate and print the standard deviation of the residuals
# it should be close to the data error if the inversion was able to fit the data
log.info("Assumed error: %f" % (error))
log.info("Standard deviation of residuals: %f" % (numpy.std(residuals)))

vis.mpl.figure(figsize=(14, 5))
vis.mpl.subplot(1, 2, 1)
vis.mpl.axis('scaled')
vis.mpl.title('Vp synthetic model of the Earth')
vis.mpl.squaremesh(model, prop='vp', vmin=vmin, vmax=vmax,
    cmap=vis.mpl.cm.seismic)
cb = vis.mpl.colorbar()
cb.set_label('Velocity')
vis.mpl.points(src_loc, '*y', label="Sources")
vis.mpl.points(rec_loc, '^r', label="Receivers")
vis.mpl.legend(loc='lower left', shadow=True, numpoints=1, prop={'size':10})
vis.mpl.m2km()
vis.mpl.subplot(1, 2, 2)
vis.mpl.axis('scaled')
vis.mpl.title('Tomography result (sharp)')
vis.mpl.squaremesh(mesh, prop='vp', vmin=vmin, vmax=vmax,
    cmap=vis.mpl.cm.seismic)
cb = vis.mpl.colorbar()
cb.set_label('Velocity')
vis.mpl.m2km()
vis.mpl.figure()
vis.mpl.grid()
vis.mpl.title('Residuals (data with %.4f s error)' % (error))
vis.mpl.hist(residuals, color='gray', bins=10)
vis.mpl.xlabel("seconds")
vis.mpl.show()
```

6.68 Seismic: 2D straight-ray tomography using smoothness regularization

```python
"""
Seismic: 2D straight-ray tomography using smoothness regularization

Uses synthetic data and a model generated from an image file.
"""
import urllib
import time
from os import path
import numpy
from fatiando import logger, mesher, utils, seismic, vis

log = logger.get()
log.info(logger.header())
log.info(__doc__)

area = (0, 500000, 0, 500000)
shape = (30, 30)
model = mesher.SquareMesh(area, shape)
# Fetch the image from the online docs
urllib.urlretrieve(
    'http://fatiando.readthedocs.org/en/latest/_static/logo.png', 'logo.png')
model.img2prop('logo.png', 4000, 10000, 'vp')

# Make some travel time data and add noise
log.info("Generating synthetic travel-time data")
src_loc = utils.random_points(area, 80)
rec_loc = utils.circular_points(area, 30, random=True)
srcs, recs = utils.connect_points(src_loc, rec_loc)
start = time.time()
tts = seismic.ttime2d.straight(model, 'vp', srcs, recs, par=True)
log.info("  time: %s" % (utils.sec2hms(time.time() - start)))
tts, error = utils.contaminate(tts, 0.01, percent=True, return_stddev=True)
# Make the mesh
mesh = mesher.SquareMesh(area, shape)
# and run the inversion
estimate, residuals = seismic.srtomo.run(tts, srcs, recs, mesh, smooth=2*10**9)
# Convert the slowness estimate to velocities and add it the mesh
mesh.addprop('vp', seismic.srtomo.slowness2vel(estimate))

# Calculate and print the standard deviation of the residuals
# it should be close to the data error if the inversion was able to fit the data
log.info("Assumed error: %g" % (error))
log.info("Standard deviation of residuals: %g" % (numpy.std(residuals)))

vis.mpl.figure(figsize=(14, 5))
vis.mpl.subplot(1, 2, 1)
vis.mpl.axis('scaled')
vis.mpl.title('Vp synthetic model of the Earth')
vis.mpl.squaremesh(model, prop='vp', cmap=vis.mpl.cm.seismic)
cb = vis.mpl.colorbar()
cb.set_label('Velocity')
vis.mpl.points(src_loc, '*y', label="Sources")
vis.mpl.points(rec_loc, '^r', label="Receivers")
vis.mpl.legend(loc='lower left', shadow=True, numpoints=1, prop={'size':10})
vis.mpl.m2km()
vis.mpl.subplot(1, 2, 2)
vis.mpl.axis('scaled')
vis.mpl.title('Tomography result (smoothed)')
vis.mpl.squaremesh(mesh, prop='vp', vmin=4000, vmax=10000,
    cmap=vis.mpl.cm.seismic)
cb = vis.mpl.colorbar()
cb.set_label('Velocity')
vis.mpl.m2km()
vis.mpl.figure()
vis.mpl.grid()
vis.mpl.title('Residuals (data with %.4f s error)' % (error))
vis.mpl.hist(residuals, color='gray', bins=10)
vis.mpl.xlabel("seconds")
vis.mpl.show()
```

6.69 Seismic: 2D straight-ray tomography of large data sets and models using sparse matrices

```python
"""
Seismic: 2D straight-ray tomography of large data sets and models using
sparse matrices

Uses synthetic data and a model generated from an image file.

Since the image is big, use sparse matrices and a steepest descent solver
(it doesn't require Hessians).

WARNING: may take a long time to calculate.

"""
import urllib
import time
from os import path
import numpy
from fatiando import logger, mesher, utils, seismic, vis, inversion

log = logger.get()
log.info(logger.header())
log.info(__doc__)

area = (0, 100000, 0, 100000)
shape = (100, 100)
model = mesher.SquareMesh(area, shape)
# Fetch the image from the online docs
urllib.urlretrieve(
    'http://fatiando.readthedocs.org/en/latest/_static/logo.png', 'logo.png')
model.img2prop('logo.png', 4000, 10000, 'vp')

# Make some travel time data and add noise
log.info("Generating synthetic travel-time data")
src_loc = utils.random_points(area, 200)
rec_loc = utils.circular_points(area, 80, random=True)
srcs, recs = utils.connect_points(src_loc, rec_loc)
start = time.time()
ttimes = seismic.ttime2d.straight(model, 'vp', srcs, recs, par=True)
log.info("  time: %s" % (utils.sec2hms(time.time() - start)))
ttimes, error = utils.contaminate(ttimes, 0.01, percent=True,
    return_stddev=True)
# Make the mesh
mesh = mesher.SquareMesh(area, shape)
# Since the matrices are big, use the Steepest Descent solver to avoid dealing
# with Hessian matrices. It needs a starting guess, so start with 1000
inversion.gradient.use_sparse()
solver = inversion.gradient.steepest(1000*numpy.ones(mesh.size))
```

```
# and run the inversion
estimate, residuals = seismic.srtomo.run(ttimes, srcs, recs, mesh, sparse=True,
    solver=solver, smooth=0.01)
# Convert the slowness estimate to velocities and add it the mesh
mesh.addprop('vp', seismic.srtomo.slowness2vel(estimate))

# Calculate and print the standard deviation of the residuals
# it should be close to the data error if the inversion was able to fit the data
log.info("Assumed error: %f" % (error))
log.info("Standard deviation of residuals: %f" % (numpy.std(residuals)))

vis.mpl.figure(figsize=(14, 5))
vis.mpl.subplot(1, 2, 1)
vis.mpl.axis('scaled')
vis.mpl.title('Vp synthetic model of the Earth')
vis.mpl.squaremesh(model, prop='vp', vmin=4000, vmax=10000,
    cmap=vis.mpl.cm.seismic)
cb = vis.mpl.colorbar()
cb.set_label('Velocity')
vis.mpl.points(src_loc, '*y', label="Sources")
vis.mpl.points(rec_loc, '^r', label="Receivers")
vis.mpl.legend(loc='lower left', shadow=True, numpoints=1, prop={'size':10})
vis.mpl.subplot(1, 2, 2)
vis.mpl.axis('scaled')
vis.mpl.title('Tomography result')
vis.mpl.squaremesh(mesh, prop='vp', vmin=4000, vmax=10000,
    cmap=vis.mpl.cm.seismic)
cb = vis.mpl.colorbar()
cb.set_label('Velocity')
vis.mpl.figure()
vis.mpl.grid()
vis.mpl.title('Residuals (data with %.4f s error)' % (error))
vis.mpl.hist(residuals, color='gray', bins=15)
vis.mpl.xlabel("seconds")
vis.mpl.show()
vis.mpl.show()
```

6.70 Seismic: 2D finite difference simulation of elastic P and SV wave propagation

```
"""
Seismic: 2D finite difference simulation of elastic P and SV wave propagation
"""
from matplotlib import animation
import numpy as np
from fatiando import seismic, logger, gridder, vis

log = logger.get()

# Make a wave source from a mexican hat wavelet
sources = [seismic.wavefd.MexHatSource(25, 25, 100, 0.5, delay=1.5)]
# Set the parameters of the finite difference grid
shape = (100, 100)
spacing = (500, 500)
area = (0, spacing[1]*shape[1], 0, spacing[0]*shape[0])
# Make a density and S wave velocity model
dens = 2700*np.ones(shape)
svel = 3000*np.ones(shape)
pvel = 4000*np.ones(shape)

```

```
# Get the iterator. This part only generates an iterator object. The actual
# computations take place at each iteration in the for loop bellow
dt = 0.05
maxit = 300
timesteps = seismic.wavefd.elastic_psv(spacing, shape, pvel, svel, dens, dt,
    maxit, sources, sources, padding=0.5)

# This part makes an animation using matplotlibs animation API
vmin, vmax = -1*10**(-4), 1*10**(-4)
x, z = gridder.regular(area, shape)
fig = vis.mpl.figure()
ax_x = vis.mpl.subplot(1, 2, 1)
vis.mpl.axis('scaled')
# Start with everything zero and grab the plot so that it can be updated later
wavefieldx = vis.mpl.pcolor(x, z, np.zeros(shape).ravel(), shape, vmin=vmin,
    vmax=vmax)
# Make z positive down
vis.mpl.ylim(area[-1], area[-2])
vis.mpl.m2km()
vis.mpl.xlabel("x (km)")
vis.mpl.ylabel("z (km)")
# Do the same for z component of the displacement
ax_z = vis.mpl.subplot(1, 2, 2)
vis.mpl.axis('scaled')
wavefieldz = vis.mpl.pcolor(x, z, np.zeros(shape).ravel(), shape, vmin=vmin,
    vmax=vmax)
# Make z positive down
vis.mpl.ylim(area[-1], area[-2])
vis.mpl.m2km()
vis.mpl.xlabel("x (km)")
vis.mpl.ylabel("z (km)")
# This function updates the plot every few timesteps
steps_per_frame = 10
def animate(i):
    for t, update in enumerate(timesteps):
        if t == steps_per_frame - 1:
            ux, uz = update
            ax_x.set_title('ux time: %0.1f s' % (i*steps_per_frame*dt))
            wavefieldx.set_array(ux[0:-1,0:-1].ravel())
            ax_z.set_title('uz time: %0.1f s' % (i*steps_per_frame*dt))
            wavefieldz.set_array(ux[0:-1,0:-1].ravel())
            break
    return wavefieldx, wavefieldz
anim = animation.FuncAnimation(fig, animate,
    frames=maxit/steps_per_frame, interval=1, blit=False)
#anim.save('p_and_sv_waves.mp4', fps=10)
vis.mpl.show()
```

6.71 Seismic: 2D finite difference simulation of elastic SH wave propagation

```
"""
Seismic: 2D finite difference simulation of elastic SH wave propagation
"""
import numpy as np
from matplotlib import animation
from fatiando import seismic, logger, gridder, vis

log = logger.get()

```

```python
# Make a wave source from a mexican hat wavelet
sources = [seismic.wavefd.MexHatSource(25, 25, 100, 0.5, delay=1.5)]
# Set the parameters of the finite difference grid
shape = (100, 100)
spacing = (500, 500)
area = (0, spacing[1]*shape[1], 0, spacing[0]*shape[0])
# Make a density and S wave velocity model
dens = 2700*np.ones(shape)
svel = 3000*np.ones(shape)

# Get the iterator. This part only generates an iterator object. The actual
# computations take place at each iteration in the for loop bellow
dt = 0.05
maxit = 400
timesteps = seismic.wavefd.elastic_sh(spacing, shape, svel, dens, dt, maxit,
    sources, padding=0.5)

# This part makes an animation using matplotlibs animation API
vmin, vmax = -1*10**(-4), 1*10**(-4)
fig = vis.mpl.figure()
vis.mpl.axis('scaled')
x, z = gridder.regular(area, shape)
# Start with everything zero and grab the plot so that it can be updated later
wavefield = vis.mpl.pcolor(x, z, np.zeros(shape).ravel(), shape, vmin=vmin,
    vmax=vmax)
# Make z positive down
vis.mpl.ylim(area[-1], area[-2])
vis.mpl.m2km()
vis.mpl.xlabel("x (km)")
vis.mpl.ylabel("z (km)")
# This function updates the plot every few timesteps
steps_per_frame = 10
def animate(i):
    for t, u in enumerate(timesteps):
        if t == steps_per_frame - 1:
            vis.mpl.title('time: %0.1f s' % (i*steps_per_frame*dt))
            wavefield.set_array(u[0:-1,0:-1].ravel())
            break
    return wavefield,
anim = animation.FuncAnimation(fig, animate,
    frames=maxit/steps_per_frame, interval=1, blit=True)
#anim.save('sh_wave.mp4', fps=10)
vis.mpl.show()
```

6.72 Seismic: 2D finite difference simulation of elastic SH wave propagation in a homogeneous medium (no Love wave)

```python
"""
Seismic: 2D finite difference simulation of elastic SH wave propagation in a
homogeneous medium (no Love wave)
"""
import numpy as np
from matplotlib import animation
from fatiando import seismic, logger, gridder, vis

log = logger.get()

# Make a wave source from a mexican hat wavelet
sources = [seismic.wavefd.MexHatSource(4, 20, 100, 0.5, delay=1.5),
           seismic.wavefd.MexHatSource(6, 22, 100, 0.5, delay=1.75),
            seismic.wavefd.MexHatSource(8, 24, 100, 0.5, delay=2)]
# Set the parameters of the finite difference grid
shape = (80, 400)
spacing = (1000, 1000)
area = (0, spacing[1]*shape[1], 0, spacing[0]*shape[0])
# Make a density and S wave velocity homogeneous model
dens = 2700*np.ones(shape)
svel = 3000*np.ones(shape)

# Get the iterator. This part only generates an iterator object. The actual
# computations take place at each iteration in the for loop bellow
dt = 0.05
maxit = 2400
timesteps = seismic.wavefd.elastic_sh(spacing, shape, svel, dens, dt, maxit,
    sources, padding=0.8)

# This part makes an animation using matplotlibs animation API
rec = 300 # The grid node used to record the seismogram
vmin, vmax = -3*10**(-4), 3*10**(-4)
fig = vis.mpl.figure(figsize=(10,6))
vis.mpl.subplots_adjust(left=0.1, right=0.98)
vis.mpl.subplot(2, 1, 2)
vis.mpl.axis('scaled')
x, z = gridder.regular(area, shape)
wavefield = vis.mpl.pcolor(x, z, np.zeros(shape).ravel(), shape,
    vmin=vmin, vmax=vmax)
vis.mpl.plot([rec*spacing[1]], [2000], '^b')
vis.mpl.ylim(area[-1], area[-2])
vis.mpl.m2km()
vis.mpl.xlabel("x (km)")
vis.mpl.ylabel("z (km)")
vis.mpl.subplot(2, 1, 1)
seismogram, = vis.mpl.plot([], [], '-k')
vis.mpl.xlim(0, dt*maxit)
vis.mpl.ylim(vmin*10.**(6), vmax*10.**(6))
vis.mpl.xlabel("Time (s)")
vis.mpl.ylabel("Amplitude ($\mu$m)")
times = []
addtime = times.append
amps = []
addamp = amps.append
# This function updates the plot every few timesteps
steps_per_frame = 100
#steps_per_frame = 1
def animate(i):
    # i is the number of the animation frame
    for t, u in enumerate(timesteps):
        addamp(10.**(6)*u[0, rec])
        addtime(dt*(t + i*steps_per_frame))
        if t == steps_per_frame - 1:
            break
    vis.mpl.title('time: %0.1f s' % (i*steps_per_frame*dt))
    seismogram.set_data(times, amps)
    wavefield.set_array(u[0:-1,0:-1].ravel())
    return seismogram, wavefield
anim = animation.FuncAnimation(fig, animate, frames=maxit/steps_per_frame,
    interval=1, blit=False)
#anim.save('sh_wave.mp4', fps=100)
vis.mpl.show()
```

6.73 Seismic: 2D finite difference simulation of elastic SH wave propagation in a medium with a discontinuity (i.e., Moho), generating Love waves.

```python
"""
Seismic: 2D finite difference simulation of elastic SH wave propagation in a
medium with a discontinuity (i.e., Moho), generating Love waves.
"""
import numpy as np
from matplotlib import animation
from fatiando import seismic, logger, gridder, vis

log = logger.get()

# Make a wave source from a mexican hat wavelet
sources = [seismic.wavefd.MexHatSource(4, 20, 100, 0.5, delay=1.5),
           seismic.wavefd.MexHatSource(6, 22, 100, 0.5, delay=1.75),
           seismic.wavefd.MexHatSource(8, 24, 100, 0.5, delay=2)]
# Set the parameters of the finite difference grid
shape = (80, 400)
spacing = (1000, 1000)
area = (0, spacing[1]*shape[1], 0, spacing[0]*shape[0])
# Make a density and S wave velocity model
moho_index = 30
moho = moho_index*spacing[0]
dens = np.ones(shape)
dens[:moho_index,:] *= 2700
dens[moho_index:,:] *= 3100
svel = np.ones(shape)
svel[:moho_index,:] *= 3000
svel[moho_index:,:] *= 6000

# Get the iterator. This part only generates an iterator object. The actual
# computations take place at each iteration in the for loop bellow
dt = 0.05
maxit = 4200
timesteps = seismic.wavefd.elastic_sh(spacing, shape, svel, dens, dt, maxit,
    sources, padding=0.8)

# This part makes an animation using matplotlibs animation API
rec = 300 # The grid node used to record the seismogram
vmin, vmax = -3*10**(-4), 3*10**(-4)
fig = vis.mpl.figure(figsize=(10,6))
vis.mpl.subplots_adjust(left=0.1, right=0.98)
vis.mpl.subplot(2, 1, 2)
vis.mpl.axis('scaled')
x, z = gridder.regular(area, shape)
wavefield = vis.mpl.pcolor(x, z, np.zeros(shape).ravel(), shape,
    vmin=vmin, vmax=vmax)
vis.mpl.plot([rec*spacing[1]], [2000], '^b')
vis.mpl.hlines([moho], 0, area[1], 'k', '-')
vis.mpl.text(area[1] - 35000, moho + 10000, 'Moho')
vis.mpl.text(area[1] - 90000, 15000,
    r'$\rho = %g g/cm^3$ $\beta = %g km/s$' % (2.7, 3))
vis.mpl.text(area[1] - 90000, area[-1] - 10000,
    r'$\rho = %g g/cm^3$ $\beta = %g km/s$' % (3.1, 6))
vis.mpl.ylim(area[-1], area[-2])
vis.mpl.m2km()
vis.mpl.xlabel("x (km)")
vis.mpl.ylabel("z (km)")
vis.mpl.subplot(2, 1, 1)
```

```python
seismogram, = vis.mpl.plot([], [], '-k')
vis.mpl.xlim(0, dt*maxit)
vis.mpl.ylim(vmin*10.**(6), vmax*10.**(6))
vis.mpl.xlabel("Time (s)")
vis.mpl.ylabel("Amplitude ($\mu$m)")
times = []
addtime = times.append
amps = []
addamp = amps.append
# This function updates the plot every few timesteps
steps_per_frame = 100
#steps_per_frame = 1
def animate(i):
    # i is the number of the animation frame
    for t, u in enumerate(timesteps):
        addamp(10.**(6)*u[0, rec])
        addtime(dt*(t + i*steps_per_frame))
        if t == steps_per_frame - 1:
            vis.mpl.title('time: %0.1f s' % (i*steps_per_frame*dt))
            seismogram.set_data(times, amps)
            wavefield.set_array(u[0:-1,0:-1].ravel())
            break
    return seismogram, wavefield
anim = animation.FuncAnimation(fig, animate, frames=maxit/steps_per_frame,
    interval=1, blit=False)
#anim.save('love_wave.mp4', fps=100)
vis.mpl.show()
```

6.74 Seismic: 2D finite difference simulation of elastic P and SV wave propagation in a medium with a discontinuity (i.e., Moho), generating Rayleigh waves

```python
"""
Seismic: 2D finite difference simulation of elastic P and SV wave propagation
in a medium with a discontinuity (i.e., Moho), generating Rayleigh waves

WARNING: Can be very slow!
"""
from matplotlib import animation
import numpy as np
from fatiando import seismic, logger, gridder, vis

log = logger.get()

# Make some seismic sources using the mexican hat wavelet
sources = [seismic.wavefd.MexHatSource(4+i, 20+i, 50, 0.5, delay=1.5 + 0.25*i)
           for i in xrange(10)]
# Make the velocity and density models
shape = (80, 400)
spacing = (1000, 1000)
area = (0, spacing[1]*shape[1], 0, spacing[0]*shape[0])
moho_index = 30
moho = moho_index*spacing[0]
dens = np.ones(shape)
dens[:moho_index,:] *= 2700.
dens[moho_index:,:] *= 3100.
pvel = np.ones(shape)
pvel[:moho_index,:] *= 4000.
pvel[moho_index:,:] *= 8000.
```

```python
svel = np.ones(shape)
svel[:moho_index,:] *= 3000.
svel[moho_index:,:] *= 6000.

# Get the iterator. This part only generates an iterator object. The actual
# computations take place at each iteration in the for loop bellow
dt = 0.05
maxit = 4200
timesteps = seismic.wavefd.elastic_psv(spacing, shape, pvel, svel, dens, dt,
    maxit, sources, sources, padding=0.8)

# This part makes an animation using matplotlibs animation API
rec = 350 # The grid node used to record the seismogram
vmin, vmax = -10*10**(-4), 10*10**(-4)
fig = vis.mpl.figure(figsize=(16,7))
vis.mpl.subplots_adjust(left=0.08, right=0.98, top=0.95, bottom=0.08)
# A plot for the ux field
plotx = vis.mpl.subplot(3, 2, 1)
xseismogram, = vis.mpl.plot([0], [0], '-k')
vis.mpl.xlim(0, dt*maxit)
vis.mpl.ylim(vmin*10.**(6), vmax*10.**(6))
vis.mpl.xlabel("Time (s)")
vis.mpl.ylabel("Amplitude ($\mu$m)")
vis.mpl.subplot(3, 2, 3)
vis.mpl.axis('scaled')
x, z = gridder.regular(area, shape)
xwavefield = vis.mpl.pcolor(x, z, np.zeros(shape).ravel(), shape,
    vmin=vmin, vmax=vmax)
vis.mpl.plot([rec*spacing[1]], [2000], '^b')
vis.mpl.hlines([moho], 0, area[1], 'k', '-')
vis.mpl.ylim(area[-1], area[-2])
vis.mpl.m2km()
vis.mpl.xlabel("x (km)")
vis.mpl.ylabel("z (km)")
# A plot for the uz field
plotz = vis.mpl.subplot(3, 2, 2)
zseismogram, = vis.mpl.plot([0], [0], '-k')
vis.mpl.xlim(0, dt*maxit)
vis.mpl.ylim(vmin*10.**(6), vmax*10.**(6))
vis.mpl.xlabel("Time (s)")
vis.mpl.ylabel("Amplitude ($\mu$m)")
vis.mpl.subplot(3, 2, 4)
vis.mpl.axis('scaled')
x, z = gridder.regular(area, shape)
zwavefield = vis.mpl.pcolor(x, z, np.zeros(shape).ravel(), shape,
    vmin=vmin, vmax=vmax)
vis.mpl.plot([rec*spacing[1]], [2000], '^b')
vis.mpl.hlines([moho], 0, area[1], 'k', '-')
vis.mpl.ylim(area[-1], area[-2])
vis.mpl.m2km()
vis.mpl.xlabel("x (km)")
vis.mpl.ylabel("z (km)")
# And a plot for the particle movement in the seismic station
ax = vis.mpl.subplot(3, 1, 3)
vis.mpl.title("Particle movement")
vis.mpl.axis('scaled')
particle_movement, = vis.mpl.plot([0], [0], '-k')
vis.mpl.xlim(vmin*10.**(6), vmax*10.**(6))
vis.mpl.ylim(vmax*10.**(6), vmin*10.**(6))
vis.mpl.xlabel("ux ($\mu$m)")
vis.mpl.ylabel("uz ($\mu$m)")
ax.set_xticks(ax.get_xticks()[1:-1])
ax.set_yticks(ax.get_yticks()[1:-1])
```

```
# Record the amplitudes at the seismic station
times = []
addtime = times.append
xamps = []
addxamp = xamps.append
zamps = []
addzamp = zamps.append
# This function updates the plot every few timesteps
steps_per_frame = 100
#steps_per_frame = 1
def animate(i):
    for t, update in enumerate(timesteps):
        ux, uz = update
        addxamp(10.**(6)*ux[0, rec])
        addzamp(10.**(6)*uz[0, rec])
        addtime(dt*(t + i*steps_per_frame))
        if t == steps_per_frame - 1:
            break
    plotx.set_title('x component | time: %0.1f s' % (i*steps_per_frame*dt))
    xseismogram.set_data(times, xamps)
    xwavefield.set_array(ux[0:-1,0:-1].ravel())
    plotz.set_title('z component | time: %0.1f s' % (i*steps_per_frame*dt))
    zseismogram.set_data(times, zamps)
    zwavefield.set_array(uz[0:-1,0:-1].ravel())
    particle_movement.set_data(xamps, zamps)
    return xwavefield, xseismogram, zwavefield, zseismogram, particle_movement
anim = animation.FuncAnimation(fig, animate, interval=1, blit=False,
    frames=maxit/steps_per_frame)
#anim.save('rayleigh_wave.mp4', fps=100)
vis.mpl.show()
```

6.75 Vis: Plot a map using the Orthographic map projection and filled contours

```
"""
Vis: Plot a map using the Orthographic map projection and filled contours
"""
from fatiando import logger, gridder, utils, vis

log = logger.get()
log.info(logger.header())
log.info(__doc__)

# Generate some data to plot
area = (-40, 0, 10, -50)
shape = (100, 100)
lon, lat = gridder.regular(area, shape)
data = utils.gaussian2d(lon, lat, 10, 20, -20, -20, angle=-45)

# Now get a basemap to plot with some projection
bm = vis.mpl.basemap(area, 'ortho')

# And now plot everything passing the basemap to the plotting functions
vis.mpl.figure()
bm.bluemarble()
vis.mpl.contourf(lon, lat, data, shape, 12, basemap=bm)
vis.mpl.colorbar()
vis.mpl.show()
```

6.76 Vis: Plot a map using the Mercator map projection and pseudo-color

```python
"""
Vis: Plot a map using the Mercator map projection and pseudo-color
"""
from fatiando import logger, gridder, utils, vis

log = logger.get()
log.info(logger.header())
log.info(__doc__)

# Generate some data to plot
area = (-20, 40, 20, 80)
shape = (100, 100)
lon, lat = gridder.regular(area, shape)
data = utils.gaussian2d(lon, lat, 10, 20, 10, 60, angle=45)

# Now get a basemap to plot with some projection
bm = vis.mpl.basemap(area, 'merc')

# And now plot everything passing the basemap to the plotting functions
vis.mpl.figure(figsize=(5,8))
vis.mpl.pcolor(lon, lat, data, shape, basemap=bm)
vis.mpl.colorbar()
bm.drawcoastlines()
bm.drawmapboundary()
bm.drawcountries()
vis.mpl.draw_geolines(area, 10, 10, bm)
vis.mpl.show()
```

6.77 Vis: Plot a map using the Robinson map projection and contours

```python
"""
Vis: Plot a map using the Robinson map projection and contours
"""
from fatiando import logger, gridder, utils, vis

log = logger.get()
log.info(logger.header())
log.info(__doc__)

# Generate some data to plot
area = (-180, 180, -80, 80)
shape = (100, 100)
lon, lat = gridder.regular(area, shape)
data = utils.gaussian2d(lon, lat, 30, 60, 10, 30, angle=-60)

# Now get a basemap to plot with some projection
bm = vis.mpl.basemap(area, 'robin')

# And now plot everything passing the basemap to the plotting functions
vis.mpl.figure()
vis.mpl.contour(lon, lat, data, shape, 15, basemap=bm)
bm.drawcoastlines()
bm.drawmapboundary(fill_color='aqua')
bm.drawcountries()
bm.fillcontinents(color='coral')
```

```
vis.mpl.draw_geolines((-180, 180, -90, 90), 60, 30, bm)
vis.mpl.show()
```

6.78 Vis: Plot the Earth, continents, inner and outer core in 3D with Mayavi2

```
"""
Vis: Plot the Earth, continents, inner and outer core in 3D with Mayavi2
"""
from fatiando.vis import myv

myv.figure(zdown=False)
myv.continents(linewidth=2)
myv.earth(opacity=0.5)
myv.core(opacity=0.7)
myv.core(inner=True)
myv.show()
```

PYTHON MODULE INDEX

INDEX