pyGIMLi

Geophysical Inversion & Modelling Library

GIMLi Documentation

Release 1.3.1+2.g7599abf9

Carsten Rücker, Thomas Günther & Florian Wagner

Dec 09, 2022

CONTENTS

CHAPTER

ONE

ABOUT PYGIMLI

1.1 Introduction

pyGIMLi is an open-source library for *modelling* (page 2) and *inversion* (page 2) and in geophysics. The object-oriented library provides management for structured and unstructured meshes in 2D and 3D, finite-element and finite-volume solvers, various geophysical forward operators, as well as Gauss-Newton based frameworks for constrained, joint and fully-coupled inversions with flexible regularization.

What is pyGIMLi suited for?

- analyze, visualize and invert geophysical data in a reproducible manner
- forward modelling of (geo)physical problems on complex 2D and 3D geometries
- inversion with flexible controls on a-priori information and regularization
- combination of different methods in constrained, joint and fully-coupled inversions
- teaching applied geophysics (e.g. in combination with Jupyter notebooks)

What is pyGIMLi **NOT** suited for?

- for people that expect a ready-made GUI for interpreting their data

1.2 Authors

We gratefully acknowledge all contributors to the pyGIMLi open-source project and look forward to your contribution!

- Carsten Rücker

  Berlin University of Technology, Department of Applied Geophysics, Berlin, Germany

  carsten@pygimli.org

- Thomas Günther

  Leibniz Institute for Applied Geophysics, Hannover, Germany

  thomas@pygimli.org

- Florian Wagner

  RWTH Aachen University, Institute for Applied Geophysics and Geothermal Energy, Aachen, Germany

florian@pygimli.org

- Friedrich Dinsel

  Berlin University of Technology, Department of Applied Geophysics, Berlin, Germany

  friedrich@pygimli.org

- Maximilian Weigand

  University of Bonn, Department of Geophysics, Bonn, Germany

- Andrea Balza

  RWTH Aachen University, Institute for Applied Geophysics and Geothermal Energy, Aachen, Germany

1.3 Inversion

One main task of pyGIMli is to carry out inversion, i.e. error-weighted minimization, for given forward routines and data. Various types of regularization on meshes (1D, 2D, 3D) with regular or irregular arrangement are available. There is flexible control of all inversion parameters. The default inversion framework is based on the generalized Gauss-Newton method.

Please see *Inversion* (page 225) for examples and more details.

1.4 Modelling

pyGIMLi comes with various geophysical forward operators, which can directly be used for a given problem. In addition, abstract finite-element and finite-volume interfaces are available to solve custom PDEs on a given mesh. See `pygimli.physics` (page 361) for a collection of forward operators and `pygimli.solver` (page 430) for the solver interface.

The modelling capabilities of pyGIMLi include:

- 1D, 2D, 3D discretizations
- linear and quadratic shape functions (automatic shape function generator for possible higher order)
- Triangle, Quads, Tetrahedron, Prism and Hexahedron, mixed meshes
- solver for elliptic problems (Helmholtz-type PDE)

Please see *Modelling* (page 214) for examples and more details.

1.5 License

1.6 Credits

We use or link some third-party software (beside the usual tool stack: cmake, gcc, boost, python, numpy, scipy, matplotlib) and are grateful for all the work made by the authors of these awesome open-source tools:

- libkdtree++: Maybe abandoned, mirror: https://github.com/nvmd/libkdtree
- meshio: https://github.com/nschloe/meshio
- pyplusplus: https://pypi.org/project/pyplusplus/
- pyvista: https://docs.pyvista.org/
- suitesparse, umfpack: https://people.engr.tamu.edu/davis/suitesparse.html
- Tetgen: http://wias-berlin.de/software/index.jsp?id=TetGen&lang=1
- Triangle: https://www.cs.cmu.edu/~quake/triangle.html

CHAPTER

TWO

CITING PYGIMLI

If you use pyGIMLi for your work, please cite this paper as:

> Rücker, C., Günther, T., Wagner, F.M., 2017. pyGIMLi: An open-source library for modelling and inversion in geophysics, Computers and Geosciences, 109, 106-123, doi: 10.1016/j.cageo.2017.07.011.

Scripts to reproduce the figures in the paper can be found at https://cg17.pygimli.org.

BibTeX code:

```
@article{Ruecker2017,
  title = {{pyGIMLi}: An open-source library for modelling and inversion in␣
↪geophysics},
  journal = {Computers and Geosciences},
  volume = {109},
  pages = {106--123},
  year = {2017},
  doi = {10.1016/j.cageo.2017.07.011},
  url = {https://www.sciencedirect.com/science/article/pii/S0098300417300584},
  author = {R\"ucker, C. and G\"unther, T. and Wagner, F. M.}
}
```

Other studies and extended abstracts about pyGIMLi

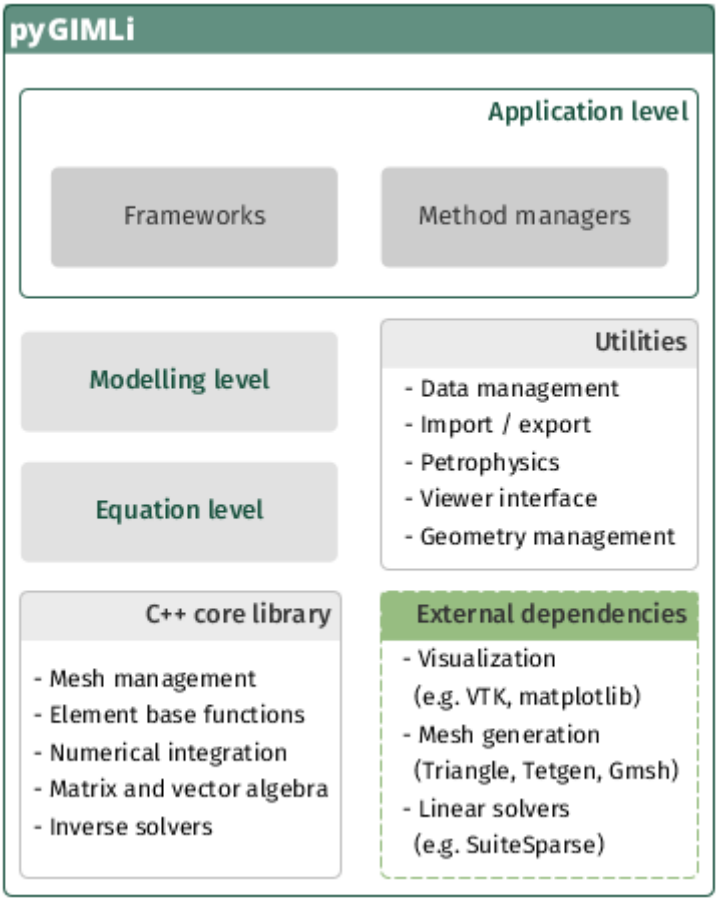
CHAPTER

THREE

SOFTWARE DESIGN

In applied geophysics, a lot of research efforts are directed towards the integration of different physical models and combined inversion approaches to estimate multi-physical subsurface parameters. Such approaches often require coupling of different software packages and file-based data exchange. The idea of pyGIMLi is to present a very flexible framework for geophysical modelling and inversion, which allows standard and customized modelling and inversion workflows to be realized in a reproducible manner. levels.

The software is written in Python on top of a C++ core library, which allows a combination of flexible scripting and numerical efficiency. pyGIMLi uses selected external dependencies for quality constrained mesh generation in 2D (*Triangle*) and 3D (*Tetgen*) and visualization in 2D (*Matplotlib*) and 3D (*Paraview*) for example. For solving linear systems we use the open-source collection *SuiteSparse* [**?**], which contains multi-frontal direct and iterative solvers as well as reordering algorithms.



pyGIMLi is organized in three different abstraction levels:

Application level

In the application level, ready-to-use method managers and frameworks are provided. Method managers (`pygimli.manager`) hold all relevant functionality related to a geophysical method. A method manager can be initialized with a data set and used to analyze and visualize this data set, create a corresponding mesh, and carry out an inversion. Various method managers are available in *pygimli.physics* (page 361). Frameworks (*pygimli.frameworks* (page 279)) are generalized abstractions of standard and advanced inversions tasks such as time-lapse or joint inversion for example. Since frameworks communicate through a unified interface, they are method independent.

Modelling level

In the modelling level, users can set up customized forward operators that map discretized parameter distributions to a data vector. Once defined, it is straightforward to set up a corresponding inversion workflow or combine the forward operator with existing ones.

Equation level

The underlying equation level allows to directly access the finite element (*pygimli.solver.solveFiniteElements()* (page 450)) and finite volume (*pygimli.solver.solveFiniteVolume()* (page 452)) solvers to solve various partial differential equations on unstructured meshes, i.e. to approach various physical problems with possibly complex 2D and 3D geometries.

CHAPTER

FOUR

INSTALLATION

On all platforms, we recommend to install pyGIMLi via the conda package manager contained in the Anaconda distribution. For details on how to install Anaconda, we refer to: https://docs.anaconda.com/anaconda/install/

To avoid conflicts with other packages, we recommend to install pygimli in a separate environment. Here we call this environment *pg*, but you can give it any name. Note that this environment has to be created only once.

Open a terminal (Linux & Mac) or the Anaconda Prompt (Windows) and type:

```
conda create -n pg -c gimli -c conda-forge pygimli=1.3.1
```

If you are using Windows or Mac, a new environment named "pg" should be visible in the Anaconda Navigator. If you want to use pyGIMLi from the command line, you have to activate the environment. You can put this line in your *~/.bashrc* file so that it is activated automatically if you open a terminal.

```
conda activate pg
```

After that you can use pyGIMLi with your text editor of choice and a terminal.

4.1 Usage with Spyder or JupyterLab

Depending on your preferences, you can also install third-party software such as the MATLAB-like integrated development environment (https://www.spyder-ide.org):

```
conda install -c conda-forge spyder
```

Or alternatively, the web-based IDE JupyterLab (https://jupyterlab.readthedocs.io):

```
conda install -c conda-forge jupyterlab
```

4.2 pyGIMLi on Google Colab

Even though still experimental, pyGIMLi can be run on Google Colab without any installation on the own computer. Just create a new Notebook and install the condacolab package by

```
!pip install -q condacolab
import condacolab
condacolab.install()
```

After doing so, the kernel is automatically restarted, so import condacolab again and install pygimli using mamba just as mentioned above.

```
import condacolab
condacolab.check()
!mamba install -c gimli pygimli=1.3.1
```

4.3 Testing the installation

To test if everything works correctly you can do the following:

```
python -c "import pygimli; pygimli.test(show=False, onlydoctests=True)"
```

4.4 Staying up-to-date

Update your pyGIMLi installation from time to time, if want to have the newest functionality.

```
conda update -c gimli -c conda-forge pygimli
```

If there something went wrong and you are running an old, not further supported python version, consider a fresh install in a new clean environment. The only drawback of using conda is that you are bound to the rhythm in which we update the conda packages. In order to work with the latest Python codes you should create an environment with the latest pyGIMLi C++ core only,

```
conda create -n pgcore -c gimli -c conda-forge pgcore
```

retrieve the source code by git

```
git clone https://github.com/gimli-org/gimli
cd gimli
```

and install pygimli as a development package

```
conda develop .
```

Alternatively you could set the PYTHONPATH variable but you would have to care for dependencies by yourself.

Later you can just update the pygimli code by

```
git pull
```

Only if you need recent changes to the C++ core, you have to compile pyGIMLi using your systems toolchain as described in https://www.pygimli.org/compilation.html#sec-build

If you want to compile pyGIMLi from source, check out sec:build.

CHAPTER

FIVE

FREQUENTLY ASKED QUESTIONS

Note: This section is still under construction. If you have a have a question (about GIMLi) that is not covered here, please open a GitHub issue.

5.1 General

5.1.1 What is the difference between BERT and GIMLi?

GIMLi is a general library for modelling and inversion in geophysics. BERT builds upon this framework and provides numerous high-level and user-friendly functions and applications specifically tailored for DC resistivity studies.

5.2 Installation

5.2.1 Python 2 or Python 3?

Short answer: Python 3. Long answer: Currently, pygimli is functionable with all major Python versions from 3.7 on. When compiling from source, it is important that *boost_python* is build against the same Python version you want to use. However, many (scientific) Python projects recommend to use Python 3 for various reasons. We droped support for Python 2. In most cases you can translate your existing Python 2 scripts to Python 3 by running 2to3.

5.2.2 What do I have to do to use pygimli in Spyder?

For Linux users it should be sufficient to have proper environment settings:

```
export PYTHONPATH=$PYTHONPATH:$HOME/src/gimli/gimli/python
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/src/gimli/build/lib
export PATH=$PATH:$HOME/src/gimli/build/bin
```

Note, these setting will ony affect applications that have been started from the current console. To make these settings permanent, make sure to add them to your *.bashrc*.

Window users should use the conda/anaconda install which sets all appropriate paths to run pygimli in spyder.

If this does not work, Windows users probably need to tell spyder where your manual pygimli installation can be found by an appropriate setting of the *PYTHONPATH* in the spyder Menubar:/tools/PYTHONPATH manager.

If you start spyder from Windows startmenu you probably need to set the correct *PATH* somewhere in your windows personal settings.

You can test you installation within the console via:

```
python -c 'import pygimli as pg; print(pg.version())'
```

If there is something wrong with the environment settings you get an error like this:

```
python: can't open file 'import pygimli as pg; print(pg.version())':
[Errno 2] No such file or directory
```

5.3 Weird findings

5.3.1 My script called sip.py and nothing works

Rename your file to something different. One of the prerequisite library (pyQT) import the file sip.py as module for they own and just stuck.

5.3.2 Segfault on import

Try *python -s -c "import pygimli"*. The *-s* option ensures that only system packages are used. This avoids conflicts with local (pip) packages.

5.3.3 CXXABI_1.3.9 not included in libstdc++.so.6

When installing conda packages on older machines, the above error may occur. If so, check out the suggestion made here.

CHAPTER

SIX

EXAMPLES

Here you find examples for modelling and inversion of various geophysical methods as well as interesting usage examples of pyGIMLi. If you have used pyGIMLi for an interesting application yourself, please send us your example.

6.1 Mesh generation

6.2 Seismic refraction and traveltime tomography

6.3 Electrical resistivity tomography

6.4 Induced polarization

6.5 Gravimetry and magnetics

6.6 Miscellaneous

Interdisciplinary and non-geophysical (e.g. fluid flow) examples.

6.7 Inversion

6.7.1 Mesh generation

6.7.1.1 Meshing the Omega aka. BERT logo

This is a fun example creating a logo for the BERT software. It illustrates the possibility to hand over matplotlib path objects to the TriangleWrapper.

```
import matplotlib as mpl
import matplotlib.pyplot as plt

import pygimli as pg
```

We start by generating a matplotlib path respresenting the Ω character.

```
logo_path = mpl.textpath.TextPath((0, 0), r'$\Omega$', size=5)
patch = mpl.patches.PathPatch(logo_path)
```

The vertices of the path are defined as mesh nodes and connected with edges.

```
poly = pg.Mesh(2)

for n in patch.get_verts() * 10:
    poly.createNodeWithCheck(n)

for i in range(poly.nodeCount() - 1):
    poly.createEdge(poly.node(i), poly.node(i + 1))

poly.createEdge(poly.node(poly.nodeCount() - 1), poly.node(0))
```

```
<pygimli.core._pygimli_.Edge object at 0x7fe8eaa7ea60>
```

We create mesh from the polygone and set the x values as the data for a color transition.

```
mesh = pg.meshtools.createMesh(poly, area=5)
```

Last, we create a BERT caption, visualize the mesh and fine-tune the figure.

```
fig, ax = plt.subplots(figsize=(4, 3))
ax.axis('off')
offset = -10
t = ax.text(mesh.xmin() + (mesh.xmax()-mesh.xmin())/2, offset, 'BERT',
            horizontalalignment='center', size=40, fontweight='bold')
pg.show(mesh, pg.x(mesh.cellCenters()), ax=ax, cMap='Spectral_r',
        logScale=False, showLater=True, showMesh=True, colorBar=False)
ax.set_ylim(offset, mesh.ymax())
```


```
(-10, 37.0)
```

6.7.1.2 CAD to mesh tutorial

In this example you will learn how to create a geometry in FreeCAD and then export and mesh it using Gmsh.

Gmsh comes with a build-in CAD engine for defining a geometry, as shown in the flexible mesh generation example, but using a parametric CAD program such as FreeCAD is much more intuitive and flexible.

For this tutorial you will need Gmsh and its Python API (application programming interface). These can be installed by the command below inside your (new) conda environment.

```
conda install -c conda-forge gmsh python-gmsh
```

In case you also want to try out FreeCAD, installing it from their website will give you the most up to date version.

This example is based on an ERT modeling and inversion experiment on a small dike. However, this FreeCAD → Gmsh workflow can easily be translated to other geophysical methods. The geometry and acquisition design come from the IDEA League master thesis of Joost Gevaert. The target in this example is to find the geometry of a sand channel underneath the dike.

FreeCAD: create the geometry

Two geometries have to be created. One for modeling and one for inversion. When the same meshes are used for modeling and inversion, the geometry of the sand channel is alreadyincluded in the structure of the mesh. Therefore, the mesh itself would act as prior information to the inversion. The modeling geometry consists of three regions: the outer region; the inner region (same as inversion region in this example) and the sand channel. The inversion geometry consists of two regions: the outer region and the inversion region.

The geometries are defined in three steps:

1. Each region of the geometry designed separately in the Part workbench, or in the Part Design workbench for more complicated geometries. To get familiar with the part design workbench, this FreeCAD-tutorial with some videos is great.
2. Merge all regions into one single ”compsolid”, i.e.composite solid. Meaning one object that consists of multiple solids that share the interfaces between the solids.
3. Export the geometry in `.brep` *

(1) The outer and inversion regions of this dike example were created in the Part Design workbench, by making a sketch and then extruding it with the Pad option. See the Inversion-Region in the object tree in the figure below. You can also have a look at how these geometries were created by downloading the .FCStd FreeCAD files and playing around with them. The sand channel is a simple cube, created in the Part workbench. Dimensions: L = 8.0 m ; W = 15.0 m ; H = 2.0 m. Position: x = 7.5 m ; y = -1.5 m ; z = -2.3 m.

(2) The trick then lies in merging these shapes into a single compsolid. This is done in the following steps:

1. Open a new project and merge all objects, i.e. regions (File → Merge project. . .) into this project

2. In the Part workbench, select all objects and create Boolean Fragments (Part → Split → Boolean Fragments)
3. Select the newly created BooleanFragments in the object tree and change its Mode property to CompSolid, see the figure below.
4. Keep BooleanFragments selected and then apply a Compound Filter to it (Part → Compound → Compound Filter)
5. Quality check the obtained geometry. Select the newly created CompoundFilter from the object tree and click Check Geometry (Part → Check Geometry). SOLID: in the Shape Content, should match the number of objects merged when creating the Boolean Fragments, 3 in this example. COMPSOLID: should be 1. Always, also for other geometries. COMPOUND: should be 0. Always. COMPSOLID: 1 and COMPOUND: 0 indicates that the objects were indeed merged correctly to one single compsolid, see the figure below.

(3) Select the CompounSolid from the object tree and export (File → Export. . .) as .brep.

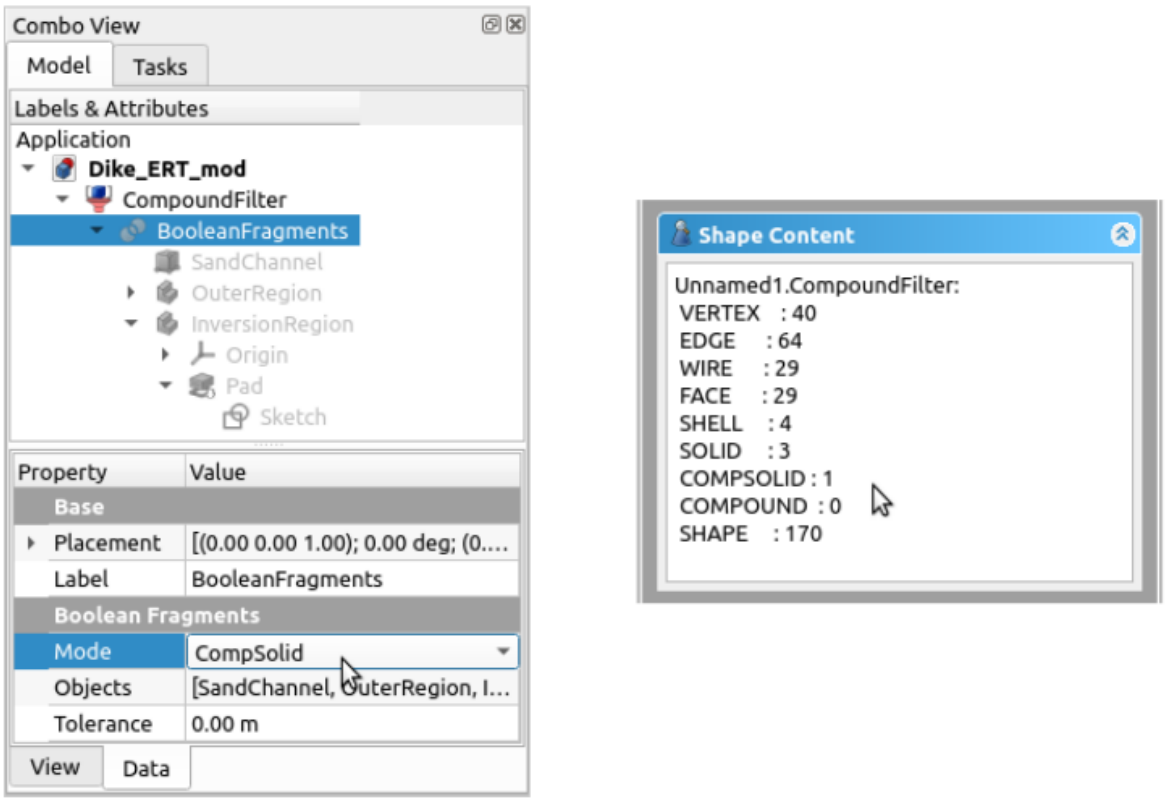


Fig. 1: FreeCAD important dialogs for making a correct compsolid.

- It must be `.brep`. This is the native format of the OpenCascade

CAD engine on which both FreeCAD and Gmsh run. `.step` (also `.stp`) is the standardized CAD exchange format, for some reason this format does not export the shape as a compound solid. Gmsh can also read `.stl` and `.iges` files. `.stl` files only contain surface information and cannot easily be reedited. `.iges` is an old format for which development stopped after 1996 and geometries are not always imported correctly.

Gmsh: mesh the geometry

Meshing with Gmsh is incredibly versatile, but has a very steep learning curve. Here we use the Python Application Programming Interface (API). To get familiar with the Python API, the Gmsh tutorials (overview) were converted to Python scripts and additional demos are also provided. I will mention or provide links to relevant tutorials and demos, have a look at these for extra context.

Let's start by importing our geometry into Gmsh:

```
import numpy as np
import pygimli as pg
gmsh = pg.optImport("gmsh", "do this tutorial. Install by running: pip install␣
↪gmsh")

# Download all nessesary files
geom_filename = pg.getExampleFile("cad/dike_mod.brep")
elec_pos_filename = pg.getExampleFile("cad/elec_pos.csv")

if gmsh:
    # Starting it up (tutorial t1.py)
    gmsh.initialize()
    gmsh.option.setNumber("General.Terminal", 1)
    gmsh.model.add("dike_mod")
    # Load a BREP file (t20.py & demo step_assembly.py)
    # .brep files don't contain info about units, so scaling has to␣
↪be applied
    gmsh.option.setNumber("Geometry.OCCScaling", 0.001)
    volumes = gmsh.model.occ.importShapes(geom_filename)
```

Before diving into local mesh refinement, putting the electrodes in the mesh and assigning region, boundary and electrode markers, the .brep geometry file should be checked. Especially check whether the meshes of two adjacent regions share nodes on their interfaces. The mesh can be viewed by running the following lines of code:

```
# Run this code after every change in the mesh to see what changed.
gmsh.model.occ.synchronize()
gmsh.model.mesh.generate(3)
gmsh.fltk.run()
```

Tips for viewing the mesh:

1. Double left clicking opens a menu in where you can set geometry and mesh visibility.
2. Tools → Visibility opens a window in which you can select parts of the mesh and geometry. Here you can find the tags of the elementary entities of the geometry. It is also handy later to QC whether physical groups were set correctly.
3. Clip the mesh and geometry with Tools → Clipping.
4. The number of elements ect. can be found in the Tools → Statistics window.

Make sure to quickly write down the Gmsh volume tags of the outer region, dike and channel and the surface tags of the free surface and the underground boundary of the box. You will need this in the next step.

If importing and meshing the .brep geometry went correctly, great! Next we include the electrodes (from Excel file) into the geometry and define the Characteristic Length (CL) for each region and the electrodes. The CL is defined at each Point and dictates the mesh size at that point. The mesh size between points is interpolated linearly, by default.

```
cl_elec = 0.1
cl_dike = 0.6
cl_outer = 30
# Gmsh geometry tags of relevant parts. Find the tags in the Gmsh␣
↪interface.
tags = {"outer region": 2,
    "dike": 3,
    "channel": 1,
    "surface": [7, 11, 12, 13, 21, 23, 24,
                25, 27, 29, 30, 31],
    "boundary": [8, 14, 15, 16, 20],      # "Underground Box Boundary"
    "electrodes": []}

if gmsh:
    # Syncronize CAD representation with the Gmsh model (t1.py)
    # Otherwise gmsh.model.get* methods don't work.
    gmsh.model.occ.synchronize()
    # Set mesh sizes for the dike and outer region.
    # The order, in which mesh sizes are set, matters. Big -> Small
    gmsh.model.mesh.setSize(            # Especially t16.py, also t2; 15;␣
↪18; 21
        gmsh.model.getBoundary(         # get dimTags of boundary␣
↪elements of
            (3, tags["outer region"]),  # dimTag: (dim, tag)
            recursive=True),            # recursive -> dimTags of points
        cl_outer)
    gmsh.model.mesh.setSize(
        gmsh.model.getBoundary((3, tags["dike"]),recursive=True),
        cl_dike)
```

Now reload the script, mesh the geometry again and have a look how the mesh changed. The next step is adding the electrodes to the mesh. The grid on the dike has 152 electrodes. These points are added in Gmsh as points 201-352, to prevent clashing with points already defined in the geometry.

```
if gmsh:
    # positions: np.array([elec#, x, y, z, y "over ground"])
    pos = np.genfromtxt(elec_pos_filename, delimiter=",", skip_header=1)
    # Electrodes are put at 2 cm depth, such that they can be␣
↪embeded in the volume of the dike.
    # Embeding the electrodes into the surface elements complicates␣
↪meshing.
    elec_depth = 0.02               # elec depth [m]
    pos[:, 3] = pos[:, 3] - elec_depth
    # Add the electrodes to the Gmsh model and put the tags into the␣
↪Dict
    for xyz in pos:
```

(continues on next page)

(continued from previous page)

```
        tag = int(200 + xyz[0])
        gmsh.model.occ.addPoint(xyz[1], xyz[2], xyz[3], cl_elec, tag)
        tags["electrodes"].append(tag)
    # Embed electrodes in dike volume. (t15.py)
    gmsh.model.occ.synchronize()
    gmsh.model.mesh.embed(0, tags["electrodes"], 3, tags["dike"])
```

Reload the Gmsh script and mesh it again to see the result. Further mesh refinement is then possible with so-called background fields. Taking a quick look at Gmsh tutorial t10.geo is highly recommended. It shows a wide range of possible background fields. In this example a Distance field is defined from the electrodes and then a Threshold field is applied as the background field:

```
# LcMax -                         /------------------
#                               /
#                             /
#                           /
# LcMin -o----------------/
#        |                |       |
#      Point           DistMin DistMax
# Field 1: Distance to electrodes

if gmsh:
    gmsh.model.mesh.field.add("Distance", 1)
    gmsh.model.mesh.field.setNumbers(1, "NodesList", tags["electrodes"])
    # Field 2: Threshold that dictates the mesh size of the␣
↪background field
    gmsh.model.mesh.field.add("Threshold", 2)
    gmsh.model.mesh.field.setNumber(2, "IField", 1)
    gmsh.model.mesh.field.setNumber(2, "LcMin", cl_elec)
    gmsh.model.mesh.field.setNumber(2, "LcMax", cl_dike)
    gmsh.model.mesh.field.setNumber(2, "DistMin", 0.2)
    gmsh.model.mesh.field.setNumber(2, "DistMax", 1.5)
    gmsh.model.mesh.field.setNumber(2, "StopAtDistMax", 1)
    gmsh.model.mesh.field.setAsBackgroundMesh(2)
```

Again reload the Gmsh script and mesh it, to see the result. As the last step in creating the mesh, the physical groups have to be defined, such PyGIMLi recognize regions, boundaries and the electrodes, see `help(pg.meshtools.readGmsh)`. Make sure to follow the same Physical Group tag number conventions for marking the regions, surfaces and points as used in PyGIMLi.

```
if gmsh:
    # Physical Volumes, "Regions" in pyGIMLi
    pgrp = gmsh.model.addPhysicalGroup(3, [tags["outer region"]], 1)  #(dim,␣
↪tag, pgrp tag)
    gmsh.model.setPhysicalName(3, pgrp, "Outer Region")     # Physical group␣
↪name in Gmsh
    pgrp = gmsh.model.addPhysicalGroup(3, [tags["dike"]], 2)
    gmsh.model.setPhysicalName(3, pgrp, "Dike")
    pgrp = gmsh.model.addPhysicalGroup(3, [tags["channel"]], 3)
    gmsh.model.setPhysicalName(3, pgrp, "Channel")
```

(continues on next page)

(continued from previous page)

```
    # Physical Surfaces, "Boundaries" in pyGIMLi,
    # pgrp tag = 1 --> Free Surface | pgrp tag > 1 --> Mixed BC
    pgrp = gmsh.model.addPhysicalGroup(2, tags["surface"], 1)
    gmsh.model.setPhysicalName(2, pgrp, "Surface")
    pgrp = gmsh.model.addPhysicalGroup(2, tags["boundary"], 2)
    gmsh.model.setPhysicalName(2, pgrp, "Underground Boundary")
    # Physical Points, "Electrodes / Sensors" in pyGIMLi, pgrp tag 99
    pgrp = gmsh.model.addPhysicalGroup(0, tags["electrodes"], 99)
    gmsh.model.setPhysicalName(0, pgrp, "Electrodes")

    # Generate the mesh and write the mesh file
    gmsh.model.occ.synchronize()
    gmsh.model.mesh.generate(3)
    gmsh.write("dike_mod.msh")
    gmsh.finalize()
```

The final mesh should look something like the figure below. Check whether the Physical Groups are defined correctly using the Visibility window as shown in the figure. Finally make the inversion mesh in the same way as the modeling mesh. The differences being that (a) there should be no sand channel in the geometry of the inversion mesh. Meaning that there are also only 2 volumes: the outer region and the dike, i.e. inversion region. And (b) that the mesh does not have to be as fine. cl elec = 0.25 and cl dike = 1.2 were used for the inversion mesh in the attached .msh file. Besides changing the mesh size by playing around with the CL, the general mesh size can also be adapted in Gmsh by changing the Global mesh size factor (double left click).

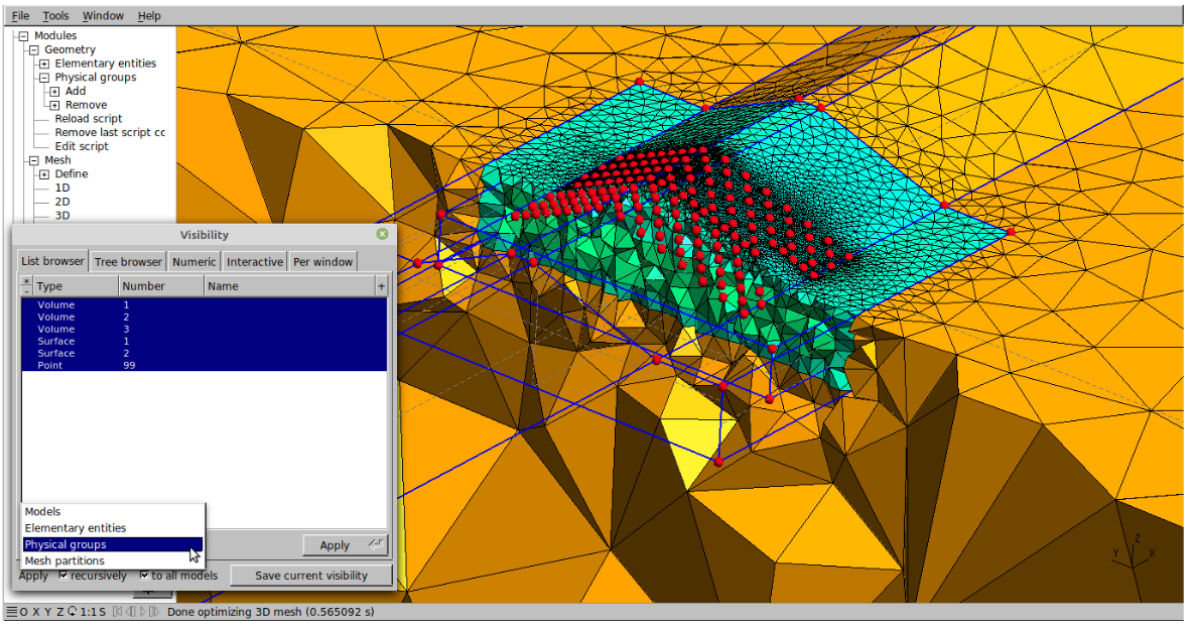


Fig. 2: Gmsh visualizatin of the modeling mesh of the dike, with visibility dialog.

Additional very useful material

- Meshing terrain from a .STL file with Gmsh
- Meshing with Gmsh from QGIS
- FreeCAD GeoMatics workbench (replaces GeoData workbench) allows for GPS, LiDAR and GIS data to be imported to FreeCAD

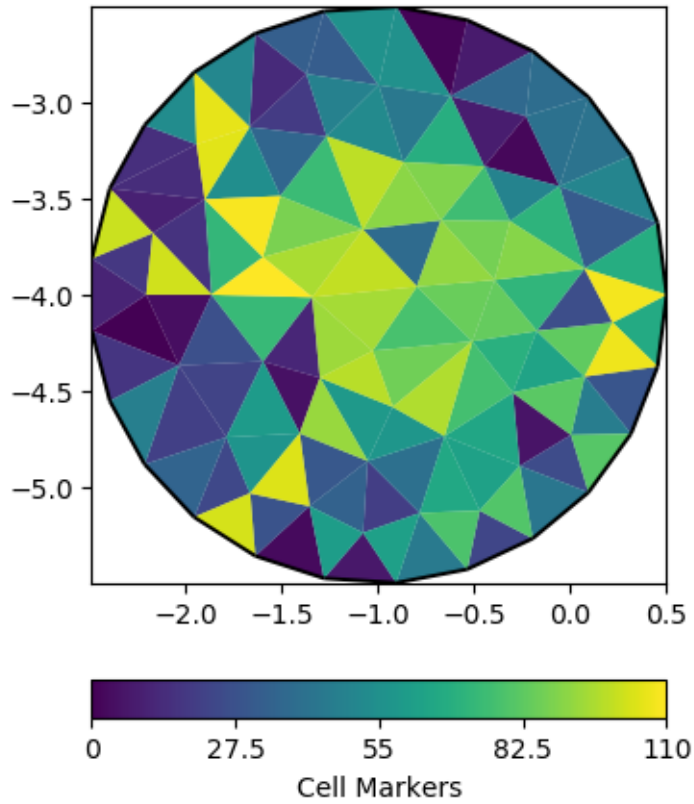
6.7.1.3 Extrude a 2D mesh to 3D

This example shows how to extrude a 2D mesh to 3D. This can be helpful for closed laboratory geometries for example. If you are looking for more flexible ways to create 3D meshes, have a look at TetGen and Gmsh.

```
import numpy as np

import pygimli as pg
import pygimli.meshtools as mt
```

We start by generating a 2D mesh.

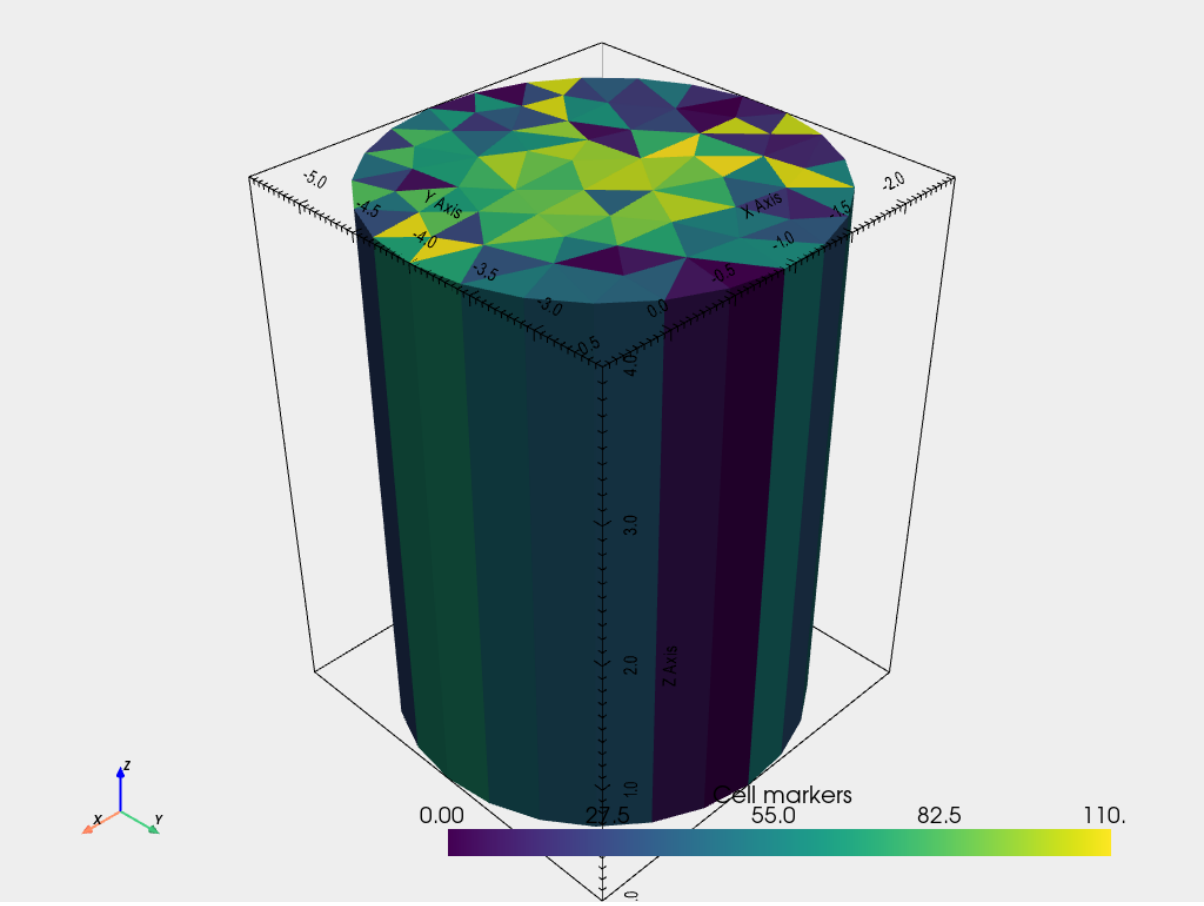
```
plc = mt.createCircle([-1, -4], radius=1.5, area=0.1, nSegments=25)
circle = mt.createMesh(plc)
for cell in circle.cells():
    cell.setMarker(cell.id())
pg.show(circle, circle.cellMarkers(), label="Cell Markers")
```



```
(<matplotlib.axes._subplots.AxesSubplot object at 0x7fe8ea901760>, <matplotlib.
↪colorbar.Colorbar object at 0x7fe8aabb69d0>)
```

We now extrude this mesh to 3D given a z vector.

```
z = np.geomspace(1, 5, 5)-1
cylinder = pg.meshtools.extrudeMesh(circle, a=z)
pg.show(cylinder, cylinder.cellMarkers(), label="Cell markers")
```




```
(<pyvista.plotting.plotting.Plotter object at 0x7fe8ea8dc7f0>, None)
```

6.7.1.4 Flexible mesh generation using Gmsh

In this example, we learn how to define arbitrary geometries, boundaries, and regions using an external mesh generator (Gmsh).

Task:

Construct a mesh with arbitrary geometry, boundaries and regions for computations in GIMLi.

Problem:

For complex geometries, mesh construction using the poly tools can be cumbersome and lacks of straightforward visual inspection.

Solution:

Create the mesh in Gmsh, a 3D finite element mesh generator with parametric input and advanced visualization capabilities, and convert it to GIMLi for subsequent modelling and inversion.

When the scientific task requires a complex finite-element discretization (i.e. incorporation of structural information, usage of a complete electrode model (CEM), etc.), external meshing tools with visualization capabilities may be the option of choice for some users. In general, the bindings provided by pygimli allow to interface any external mesh generation software.

This HowTo presents an example using Gmsh [**?**]. Gmsh allows for parametric input, i.e. physical boundaries and regions (and any other input) can be specified interactively using the graphical user interface or Gmsh's own scripting language. A lot of profound tutorials can be found on the Gmsh website (http://www.gmsh.info) or elsewhere. Here, a crosshole ERT example with geological a priori information is presented with a focus on the usage in GIMLi.

Geometry

We start with the definition of several points to layout the main geometry. A point is created via the graphical user interface as illustrated in the following figure.

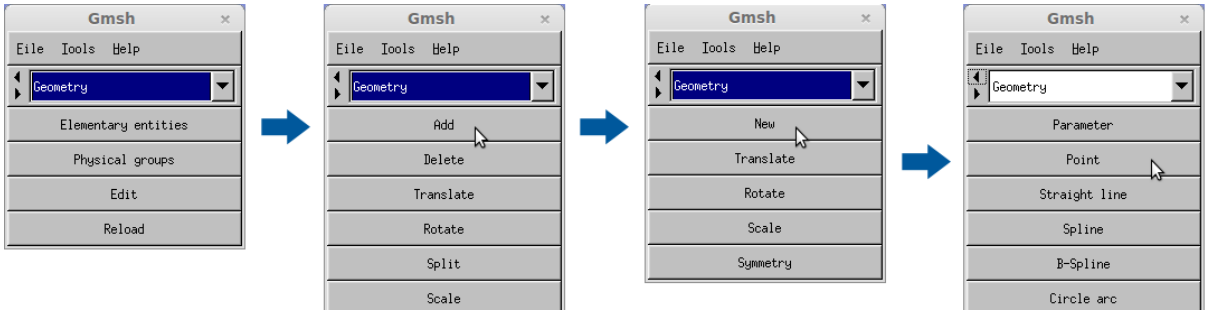


Fig. 3: Steps to create a point via the graphical user interface.

We create a large domain to solve the forward problem and specify the coordinates as well as a characteristic length to constrain the relative size of the mesh elements at that point (this is also useful for near-electrode refinement for example).

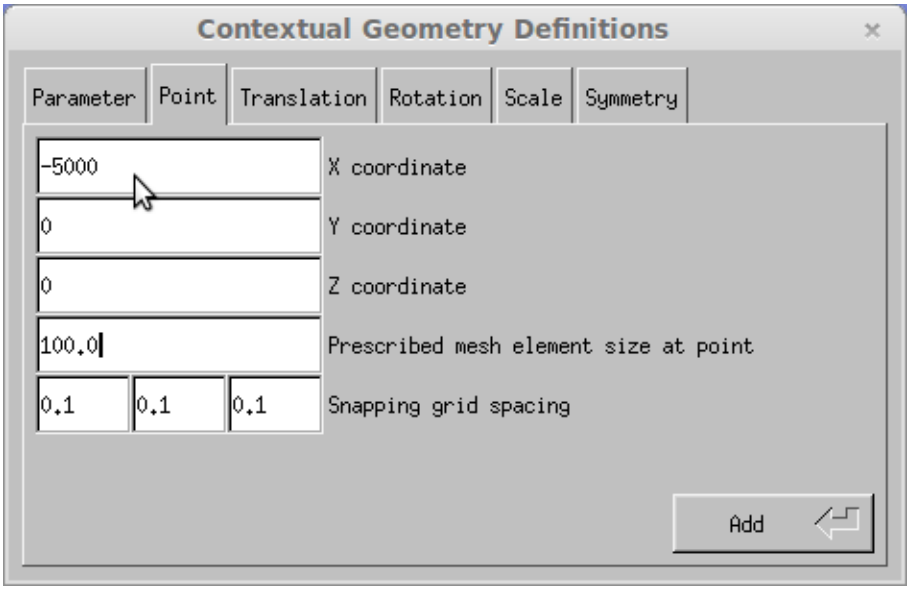


Fig. 4: Setting the parameters of a point.

In the geometry file being created, this step would correspond the following command.

```
Point(1) = {-5000, 0, 0, cl1};
```

During this HowTo, we will switch between the GUI input and the scripting language. Gmsh's reload button allows for quick and straightforward interaction between both modes.

Note that it is convenient to replace the characteristic length by a variable During this HowTo, we will switch between the GUI input and the scripting language. Subsequent to the definition of the corner points, we can set up the boundaries by connecting the points created, as shown below.

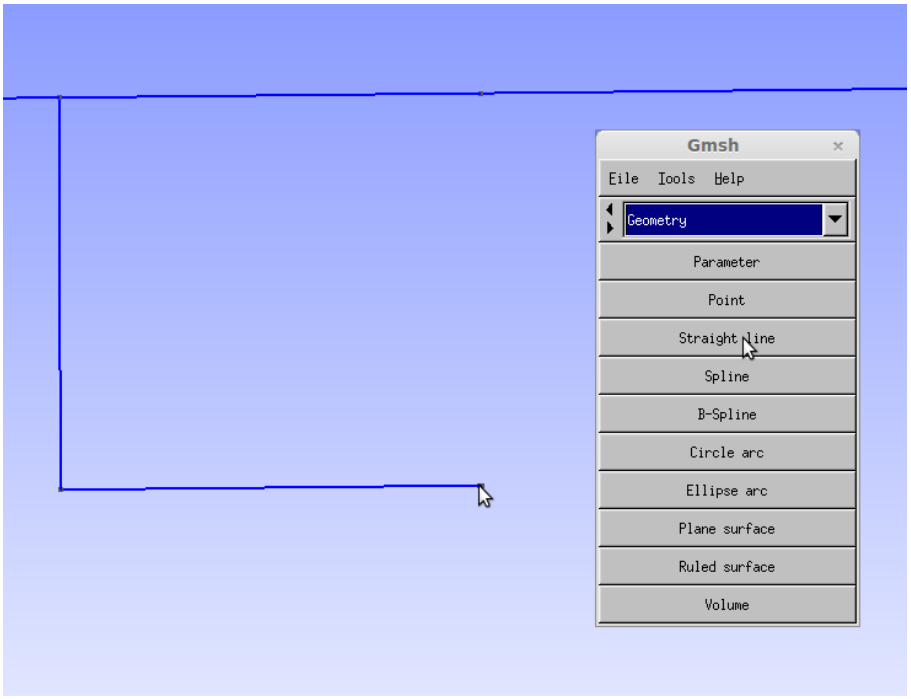


Fig. 5: Connecting geometric points using straight lines.

The corresponding command in the script to connect points 1 and 5 would look like this:

```
Line(1) = {1, 5};
```

Obviously, we also have to define the electrode positions:

```
For i In {0:9}
Point(newp) = {15,0,-4*(i+2),cl2}; // Borehole 1
Point(newp) = {35,0,-4*(i+2),cl2}; // Borehole 2
EndFor
```

Similarly, we define a set of points describing a geological body and connect them with a spline curve:

```
Spline(100) = {31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
               41, 42, 43, 44, 45, 29, 30, 31};
```

After the definition of all points and lines, we can define the three surfaces. A surface is created by selecting *Plane Surface* from the menu and clicking on the bounding lines and the holes (if present). The following figure illustrates the definition of the outer surface.

When the surfaces (or volumes in 3D) have been defined, the mesh can be generated by simply clicking the 2D button in the mesh menu (*Mesh - 2D*). As you will notice, the electrodes are not located on node

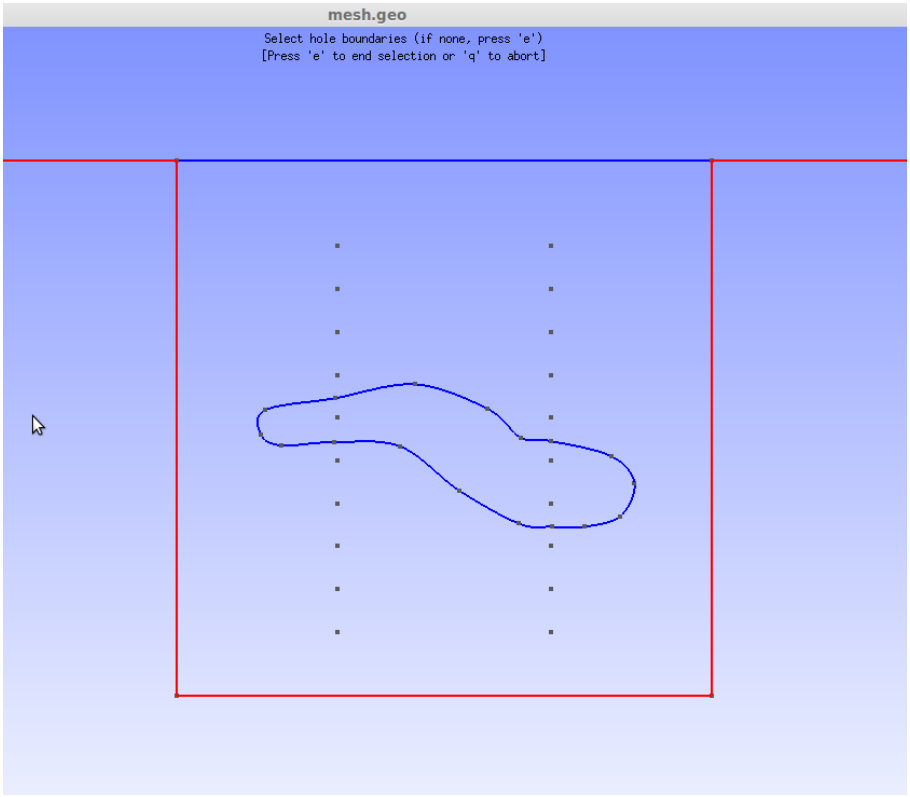


Fig. 6: Creating a surface by clicking on the boundaries.

points, as they # do not layout any geometric feature. To change this, we can embed them in the surfaces (*Mesh - Define - Embedded Points*) or directly in the script via:

```
Point{10, 12, 14, ..., 17, 23, 25, 27} In Surface{106};
Point{18, 19, 21} In Surface{104}; // electrodes within the target
```

Boundaries and regions

Since Gmsh allows for parametric input, we can finally specify the boundary conditions and region marker. This is done in the *Physical Groups* section under Geometry. The group numbers can be changed within the script. Number 1 is assigned to a Neumann-type boundary condition and number 2 to a mixed one.

```
Physical Line(1) = {3, 2, 1}; // Free surface
Physical Line(2) = {4, 5, 6}; // Mixed boundary conditions
```

The indices of the regions will directly map to the region marker in BERT.

```
Physical Surface(1) = {102}; // Outer region
Physical Surface(2) = {106}; // Inversion region
Physical Surface(3) = {104}; // Geological body
```

Finally, we assign all electrodes to a Physical Group with the marker 99.

```
Physical Point(99) = {9, 11, ..., 24, 26, 28}; // electrode marker (99)
```

That's it! Now, you can re-run the meshing algorithm and save the result.

Note that in addition to the characteristic length at each point, there are many different ways to constrain the element size (in general or locally) and the resulting mesh quality, which will not be discussed here.

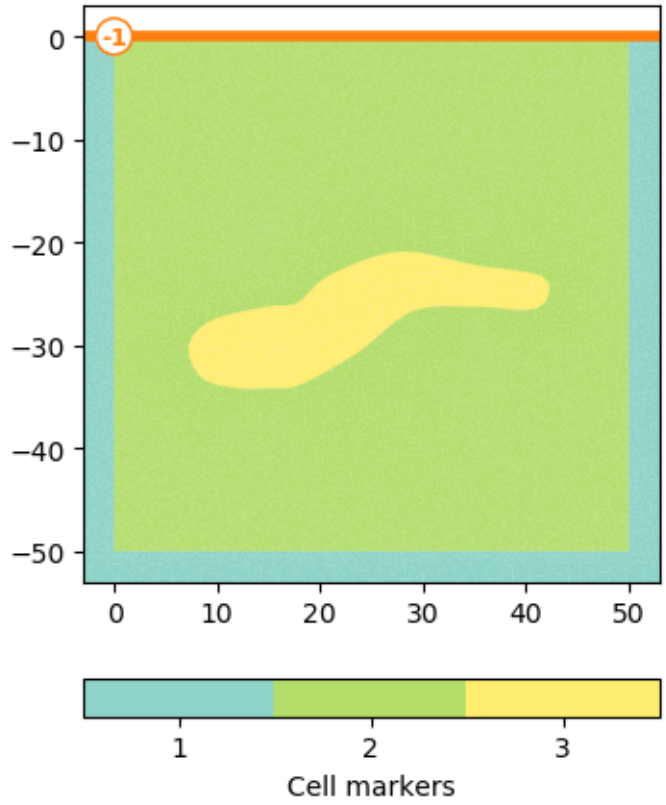
The final geometry can be downloaded here and meshed in the GUI or via the command:

```
gmsh -2 -o mesh.msh mesh.geo
```

Import to GIMLi

Any Gmsh output (2D and 3D) can be imported using pygimli and subsequently saved to the binary format.

```
import subprocess

import matplotlib.pyplot as plt

import pygimli as pg
from pygimli.meshtools import readGmsh


filename = pg.getExampleFile("gmsh/2d_tutorial.geo")

try:
    subprocess.call(
        ["gmsh", "-format", "msh2", "-2", "-o", "mesh.msh", filename])
    gmsh = True
except OSError:
    print("Gmsh needs to be installed for this example.")
    gmsh = False

fig, ax = plt.subplots()
if gmsh:
    mesh = readGmsh("mesh.msh", verbose=True)
    pg.show(mesh, ax=ax, markers=True, clipBoundaryMarkers=True)
    ax.set_xlim(-3, 53)
    ax.set_ylim(-53, 3)
else:
    ax.set_title("Gmsh needs to be installed for this example")
pg.wait()
```




```
Reading mesh.msh...

  Nodes: 12415
  Entries: 24848
    Points: 20
    Lines: 220
    Triangles: 24608
    Quads: 0
    Tetrahedra: 0

Creating mesh object...

  Dimension: 2-D
  Boundary types: 2 (-2, -1)
  Regions: 3 (1, 2, 3)
  Marked nodes: 20 (99,)

Done.

  Mesh: Nodes: 12415 Cells: 24608 Boundaries: 37022
```

For the sake of illustration, the example presented was chosen to be simple and two-dimensional, although Gmsh and the import function provided allow for much more...

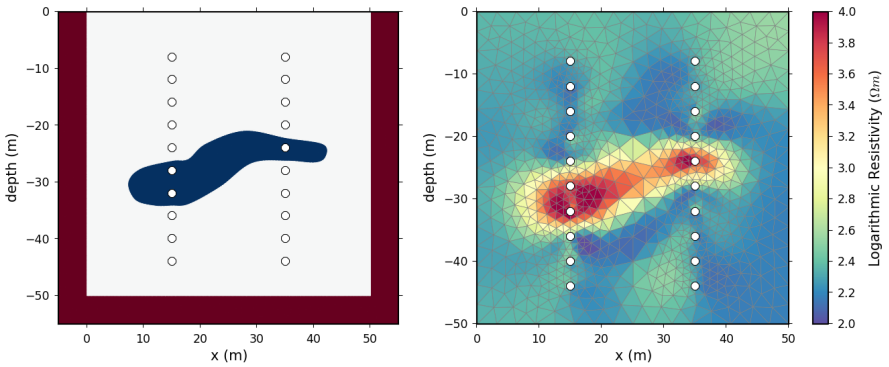


Fig. 7: Synthetic example.

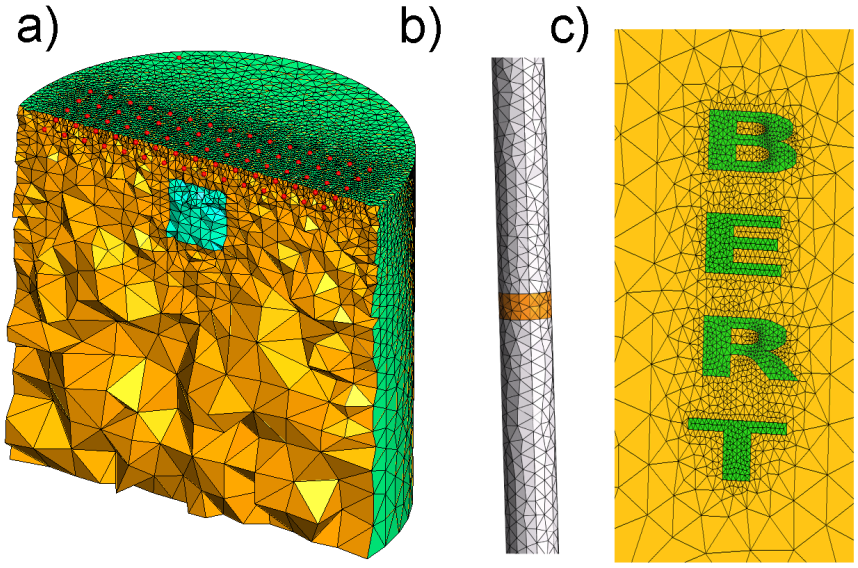


Fig. 8: Additional Gmsh examples: a) Laboratory sandbox model. b) Finite discretization of a ring-shaped electrode. c) And more!

6.7.1.5 Building a hybrid mesh in 2D

In some cases, the modelling domain may require flexibility in one region and equidistant structure in another. In this short example, we demonstrate how to accomplish this for a two-dimensional mesh consisting of a region with regularly spaced quadrilaterals and a region with unstructured triangles.

We start by importing numpy, matplotlib and pygimli with its required components.

```
import numpy as np

import pygimli as pg
import pygimli.meshtools as mt
```

We continue by building a regular grid and assign the marker 2 to all cells.

```
xmin, xmax = 0., 50.
zmin, zmax = -50., -25.

xreg = np.linspace(xmin, xmax, 13)
zreg = np.linspace(zmin, zmax, 13)

mesh1 = mt.createGrid(xreg, zreg, marker=2)
mesh1.setCellMarkers([2]*mesh1.cellCount())
print(mesh1)
```

```
Mesh: Nodes: 169 Cells: 144 Boundaries: 312
```

Next, we build an unstructured region on top by creating the polygon and calling triangle via pygimli's TriangleWrapper.

```
poly = pg.Mesh(2)  # empty 2d mesh
nStart = poly.createNode(xmin, zmax, 0.0)

nA = nStart
for x in xreg[1:]:
    nB = poly.createNode(x, zmax, 0.0)
    poly.createEdge(nA, nB)
    nA = nB

z2 = 0.
nA = poly.createNode(xmax, z2, 0.0)
poly.createEdge(nB, nA)
nB = poly.createNode(xmin, z2, 0.0)
poly.createEdge(nA, nB)
poly.createEdge(nB, nStart)

mesh2 = mt.createMesh(poly, quality=31)
mesh2.setCellMarkers([1]*mesh2.cellCount())
print(mesh2)
```

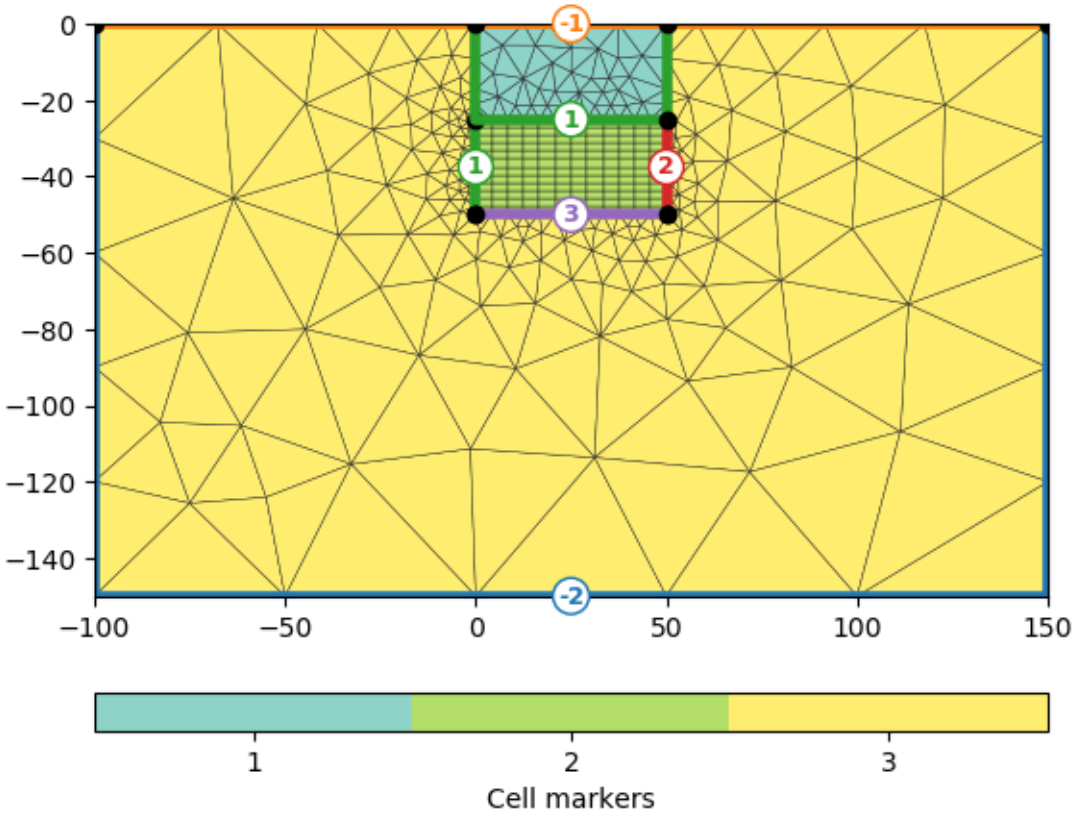
```
Mesh: Nodes: 52 Cells: 77 Boundaries: 128
```

Finally, the grid and the unstructured mesh can be merged to single mesh for further modelling.

```
mesh3 = mt.mergeMeshes([mesh1, mesh2])
```

Of course, you can treat the hybrid mesh like any other mesh and append a triangle boundary for example with the function `pygimli.meshtools.grid.appendTriangleBoundary()`.

```
mesh = mt.appendTriangleBoundary(mesh3, xbound=100., ybound=100., quality=31,␣
↪smooth=True,
                                 marker=3, isSubSurface=True, addNodes=5)

pg.show(mesh, markers=True, showMesh=True)
pg.wait()
```



6.7.2 Seismic refraction and traveltime tomography

6.7.2.1 2D Refraction modeling and inversion

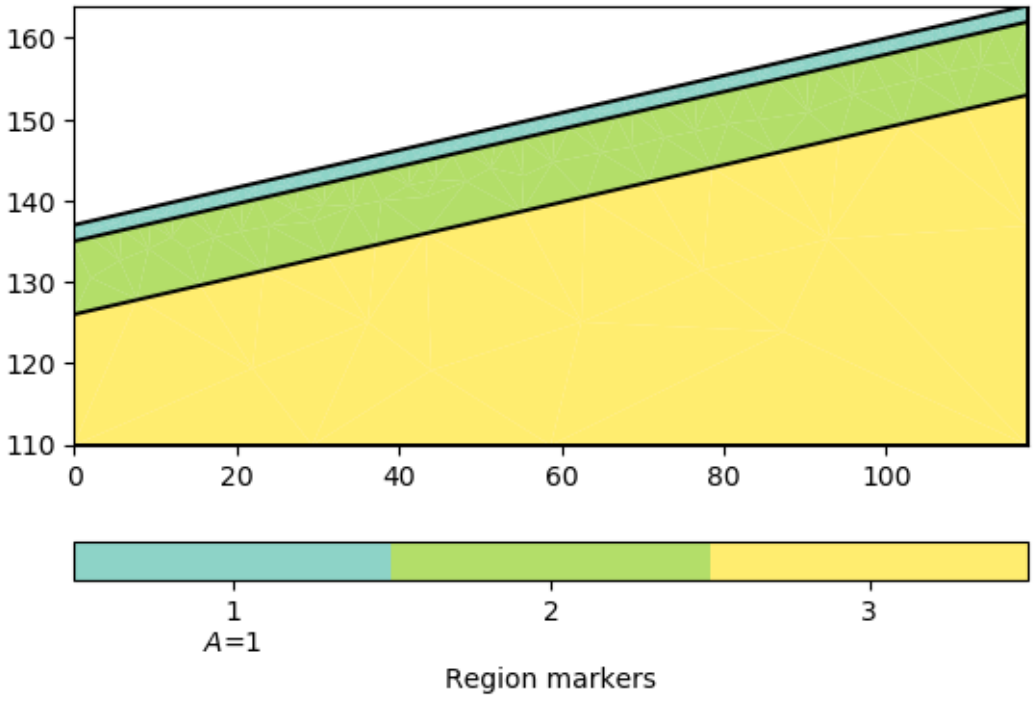
This example shows how to use the TravelTime manager to generate the response of a three-layered sloping model and to invert the synthetic noisified data.

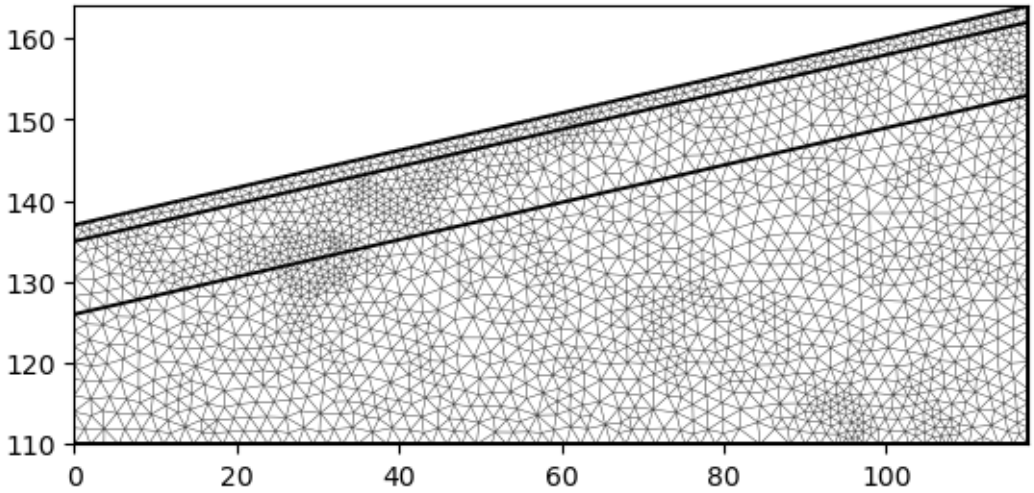
```
import numpy as np

import pygimli as pg
import pygimli.meshtools as mt
from pygimli.physics import TravelTimeManager
```

Model setup

We start by creating a three-layered slope (The model is taken from the BSc thesis of Constanze Reinken conducted at the University of Bonn).

```
layer1 = mt.createPolygon([[0.0, 137], [117.5, 164], [117.5, 162], [0.0, 135]],
                          isClosed=True, marker=1, area=1)
layer2 = mt.createPolygon([[0.0, 126], [0.0, 135], [117.5, 162], [117.5, 153]],
                          isClosed=True, marker=2)
layer3 = mt.createPolygon([[0.0, 110], [0.0, 126], [117.5, 153], [117.5, 110]],
                          isClosed=True, marker=3)

slope = (164 - 137) / 117.5

geom = layer1 + layer2 + layer3

# If you want no sloped flat earth geometry .. comment out the next␣
↪2 lines
# geom = mt.createWorld(start=[0.0, 110], end=[117.5, 137],␣
↪layers=[137-2, 137-11])
# slope = 0.0

pg.show(geom)

mesh = mt.createMesh(geom, quality=34.3, area=3, smooth=[1, 10])
pg.show(mesh)
```






```
(<matplotlib.axes._subplots.AxesSubplot object at 0x7fe8ea88f5b0>, None)
```

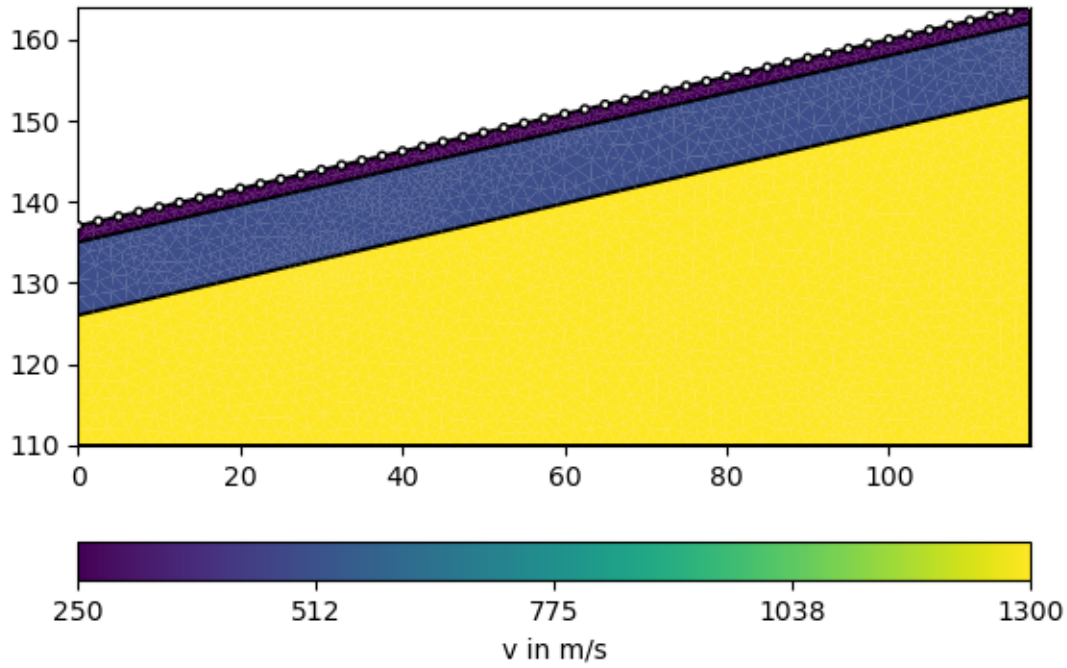
Next we define geophone positions and a measurement scheme, which consists of shot and receiver indices.

```
numberGeophones = 48
sensors = np.linspace(0., 117.5, numberGeophones)
scheme = pg.physics.traveltime.createRAData(sensors)

# Adapt sensor positions to slope
pos = np.array(scheme.sensors())
for x in pos[:, 0]:
    i = np.where(pos[:, 0] == x)
    new_y = x * slope + 137
    pos[i, 1] = new_y

scheme.setSensors(pos)
```

Synthetic data generation

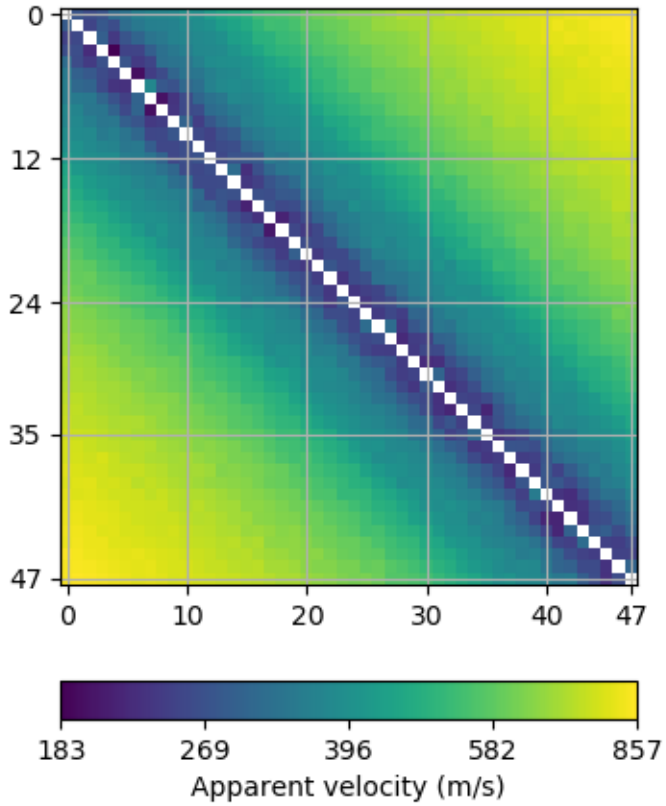
Now we initialize the TravelTime manager and asssign P-wave velocities to the layers. To this end, we create a map from cell markers 0 through 3 to velocities (in m/s) and generate a velocity vector. To check whether the model looks correct, we plot it along with the sensor positions.

```
mgr = TravelTimeManager()
vp = np.array(mesh.cellMarkers())
vp[vp == 1] = 250
vp[vp == 2] = 500
vp[vp == 3] = 1300

ax, _ = pg.show(mesh, vp, colorBar=True, logScale=False, label='v in m/s')
pg.viewer.mpl.drawSensors(ax, scheme.sensors(), diam=1.0,
                          facecolor='white', edgecolor='black')
```



We use this model to create noisified synthetic data and look at the traveltime data matrix. Note, we force a specific noise seed as we want reproducable results for testing purposes. TODO: show first arrival traveltime curves.

```
data = mgr.simulate(slowness=1.0 / vp, scheme=scheme, mesh=mesh,
                    noiseLevel=0.001, noiseAbs=0.001, seed=1337,
                    verbose=True)
mgr.showData(data)
```




```
((<matplotlib.image.AxesImage object at 0x7fe8aadf7c40>, <matplotlib.colorbar.
↪Colorbar object at 0x7fe8f54fc4f0>), None)
```

Inversion

Now we invert the synthetic data. We need a new independent mesh without information about the layered structure. This mesh can be created manual or guessd automatic from the data sensor positions (in this example). We tune the maximum cell size in the parametric domain to 15m^2

```
vest = mgr.invert(data, secNodes=2, paraMaxCellSize=15.0,
                  maxIter=10, verbose=True)
np.testing.assert_array_less(mgr.inv.inv.chi2(), 1.1)
```

```
fop: <pygimli.physics.traveltime.modelling.TravelTimeDijkstraModelling object␣
↪at 0x7fe8fa4f0720>
Data transformation: <pygimli.core._pygimli_.RTrans object at 0x7fe8fa4f0400>
Model transformation: <pygimli.core._pygimli_.RTransLog object at 0x7fe8fa4f04f0>
min/max (data): 0.0069/0.14
min/max (error): 0.8%/14.67%
min/max (start model): 2.0e-04/0.002
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
inv.iter 2 ... chi² = 2.11 (dPhi = 39.55%) lam: 20
--------------------------------------------------------------------------------
inv.iter 3 ... chi² = 1.57 (dPhi = 23.92%) lam: 20.0
```

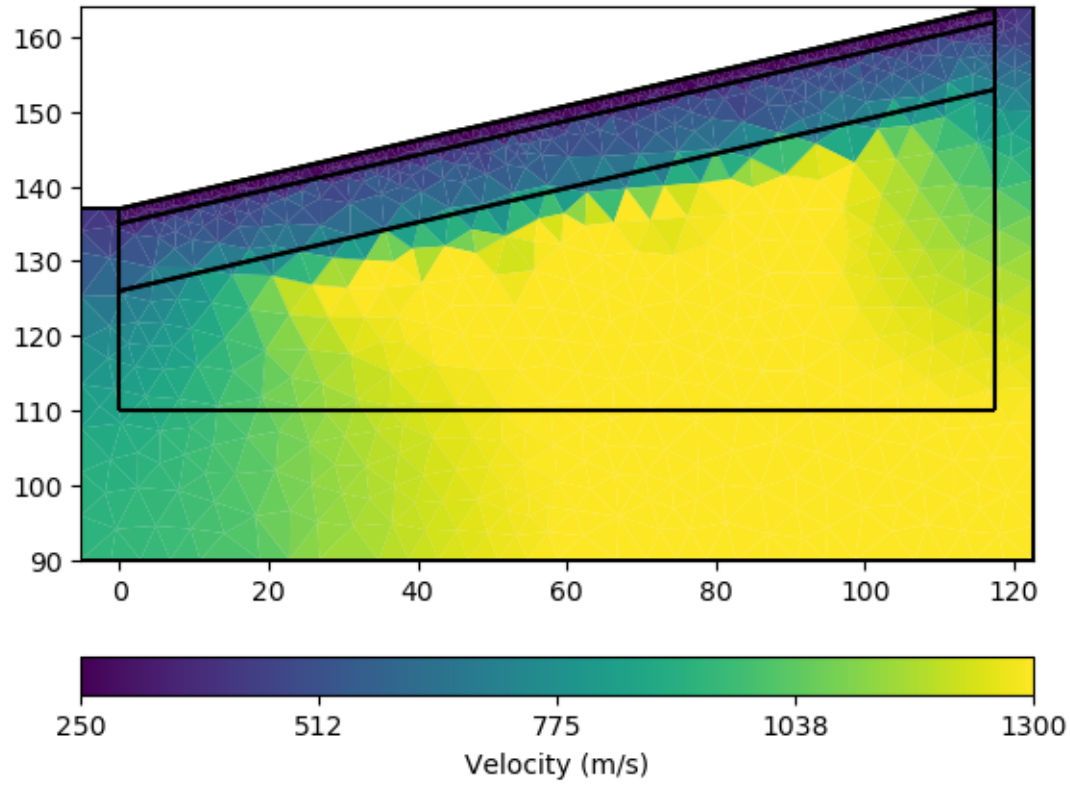
(continues on next page)

(continued from previous page)

```
--------------------------------------------------------------------------------
inv.iter 4 ... chi² = 1.22 (dPhi = 19.64%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 5 ... chi² = 1.07 (dPhi = 11.46%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 6 ... chi² = 0.97 (dPhi = 7.53%) lam: 20.0


################################################################################
#                     Abort criterion reached: chi² <= 1 (0.97)                #
################################################################################
```

The manager also holds the method showResult that is used to plot the result. Note that only covered cells are shown by default. For comparison we plot the geometry on top.

```
ax, _ = mgr.showResult(cMin=min(vp), cMax=max(vp), logScale=False)
pg.show(geom, ax=ax, fillRegion=False, regionMarker=False)
```



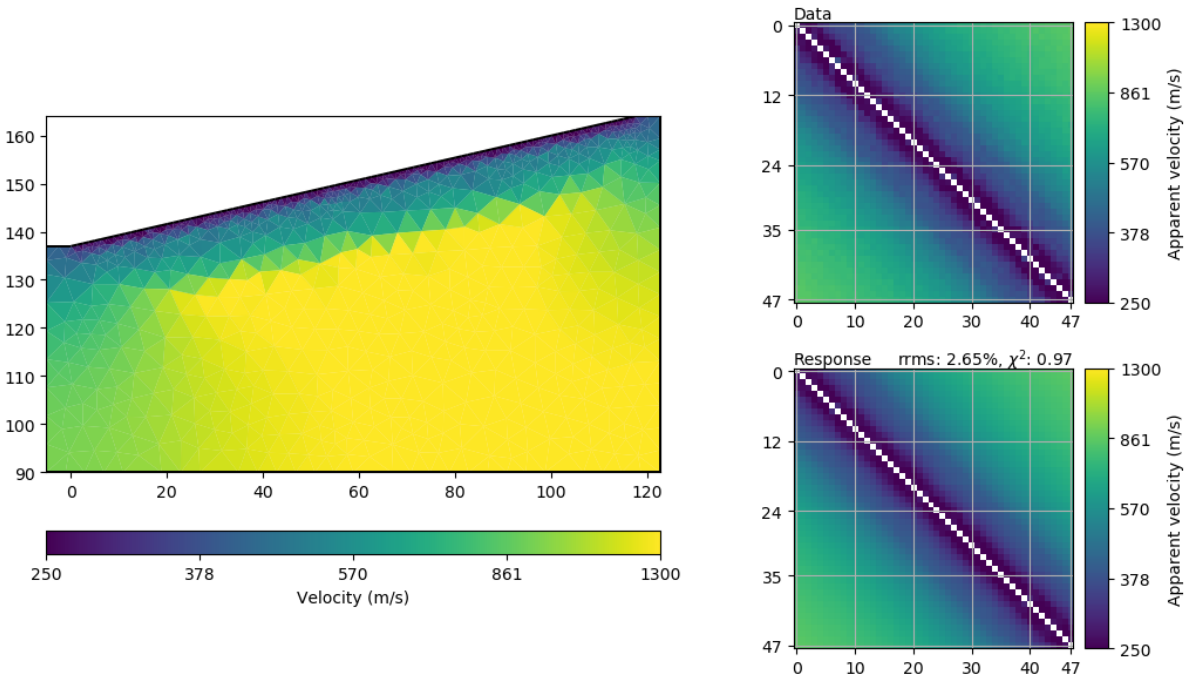

```
(<matplotlib.axes._subplots.AxesSubplot object at 0x7fe8f54fc9a0>, None)
```

Note that internally the following is called

```
ax, _ = pg.show(ra.mesh, vest, label="Velocity [m/s]", **kwargs)
```

Another useful method is to show the model along with its response on the data.

```
mgr.showResultAndFit(cMin=min(vp), cMax=max(vp))
```




```
<Figure size 1100x600 with 6 Axes>
```

Note: Takeaway message

A default data inversion with checking of the data consists of only few lines. Check out *Field data inversion ("Koenigsee")* (page 53).

Total running time of the script: (0 minutes 15.947 seconds)

6.7.2.2 Crosshole traveltime tomography

Seismic and ground penetrating radar (GPR) methods are frequently applied to image the shallow subsurface. While novel developments focus on inverting the full waveform, ray-based approximations are still widely used in practice and offer a computationally efficient alternative. Here, we demonstrate the modelling of traveltimes and their inversion for the underlying slowness distribution for a crosshole scenario.

We start by importing the necessary packages.

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

import pygimli as pg
import pygimli.meshtools as mt
import pygimli.physics.traveltime as tt
# from pygimli.physics.traveltime import TravelTimeManager,␣
↪createCrossholeData
```

(continues on next page)

(continued from previous page)

```
pg.utils.units.quants['vel']['cMap'] = 'inferno_r'
```

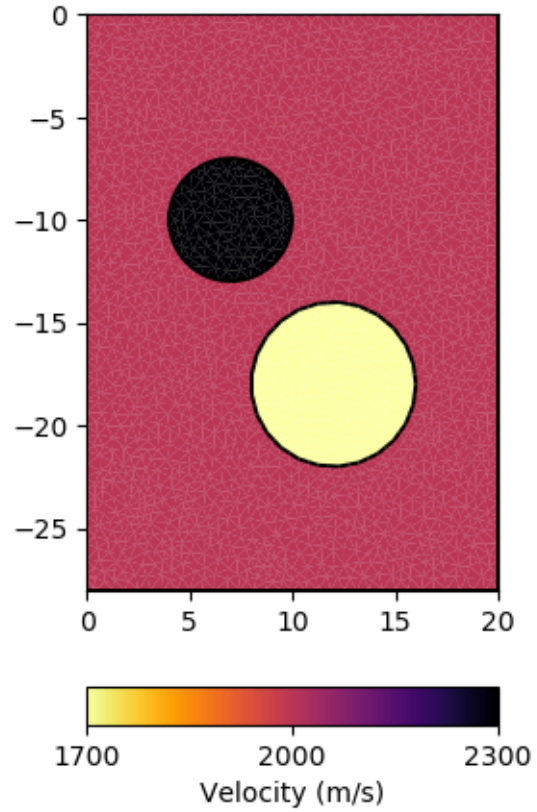
Geometry setup

Next, we build the crosshole acquisition geometry with two shallow boreholes.

```
# Acquisition parameters
bh_spacing = 20.0
bh_length = 25.0
sensor_spacing = 2.5

world = mt.createRectangle(start=[0, -(bh_length + 3)], end=[bh_spacing, 0.0],
                           marker=0)

depth = -np.arange(sensor_spacing, bh_length + sensor_spacing, sensor_spacing)

sensors = np.zeros((len(depth) * 2, 2))  # two boreholes
sensors[len(depth):, 0] = bh_spacing  # x
sensors[:, 1] = np.hstack([depth] * 2)  # y
```

Traveltime calculations work on unstructured meshes and structured grids. We demonstrate this here by simulating the synthetic data on an unstructured mesh and inverting it on a simple structured grid.

```
# Create forward model and mesh
c0 = mt.createCircle(pos=(7.0, -10.0), radius=3, nSegments=25, marker=1)
c1 = mt.createCircle(pos=(12.0, -18.0), radius=4, nSegments=25, marker=2)
geom = world + c0 + c1
for sen in sensors:
    geom.createNode(sen)

mesh_fwd = mt.createMesh(geom, quality=34, area=0.25)
model = np.array([2000., 2300, 1700])[mesh_fwd.cellMarkers()]
pg.show(mesh_fwd, model,
        label=pg.unit('vel'), cMap=pg.cmap('vel'), nLevs=3, logScale=False)
```




```
(<matplotlib.axes._subplots.AxesSubplot object at 0x7fe8f56a0520>, <matplotlib.
↪colorbar.Colorbar object at 0x7fe8fa4b0550>)
```

Synthetic data generation

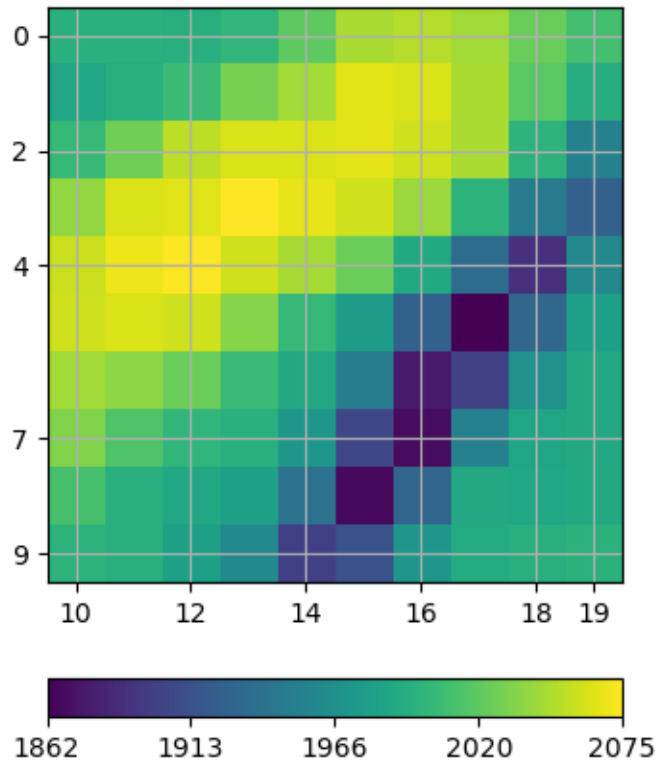
Next, we create an empty DataContainer and fill it with sensor positions and all possible shot-receiver pairs for the two-borehole scenario.

```
scheme = tt.createCrossholeData(sensors)
```

The forward simulation is performed with a few lines of code. We initialize an instance of the Refraction manager and call its *simulate* function with the mesh, the scheme and the slowness model (1 / velocity). We also add 0.1% relative and 10 microseconds of absolute noise.

Secondary nodes allow for more accurate forward simulations. Check out the paper by Giroux & Larouche (2013) to learn more about it.

```
mgr = tt.TravelTimeManager()
data = tt.simulate(mesh=mesh_fwd, scheme=scheme, slowness=1./model,
                   secNodes=4, noiseLevel=0.001, noiseAbs=1e-5, seed=1337)

tt.showVA(data, usePos=False)
```

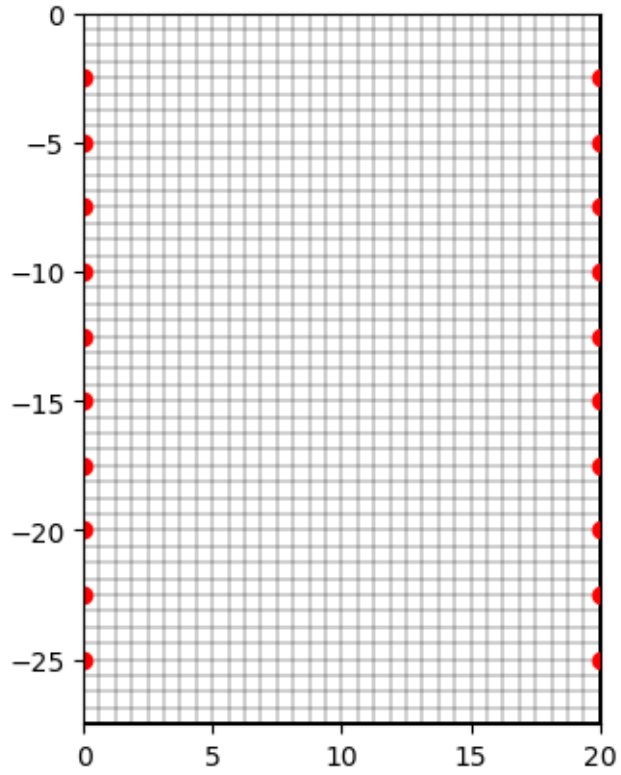



```
(<matplotlib.image.AxesImage object at 0x7fe8fa1c8790>, <matplotlib.colorbar.
↪Colorbar object at 0x7fe8ea8fa7c0>)
```

Inversion

Now we create a structured grid as inversion mesh

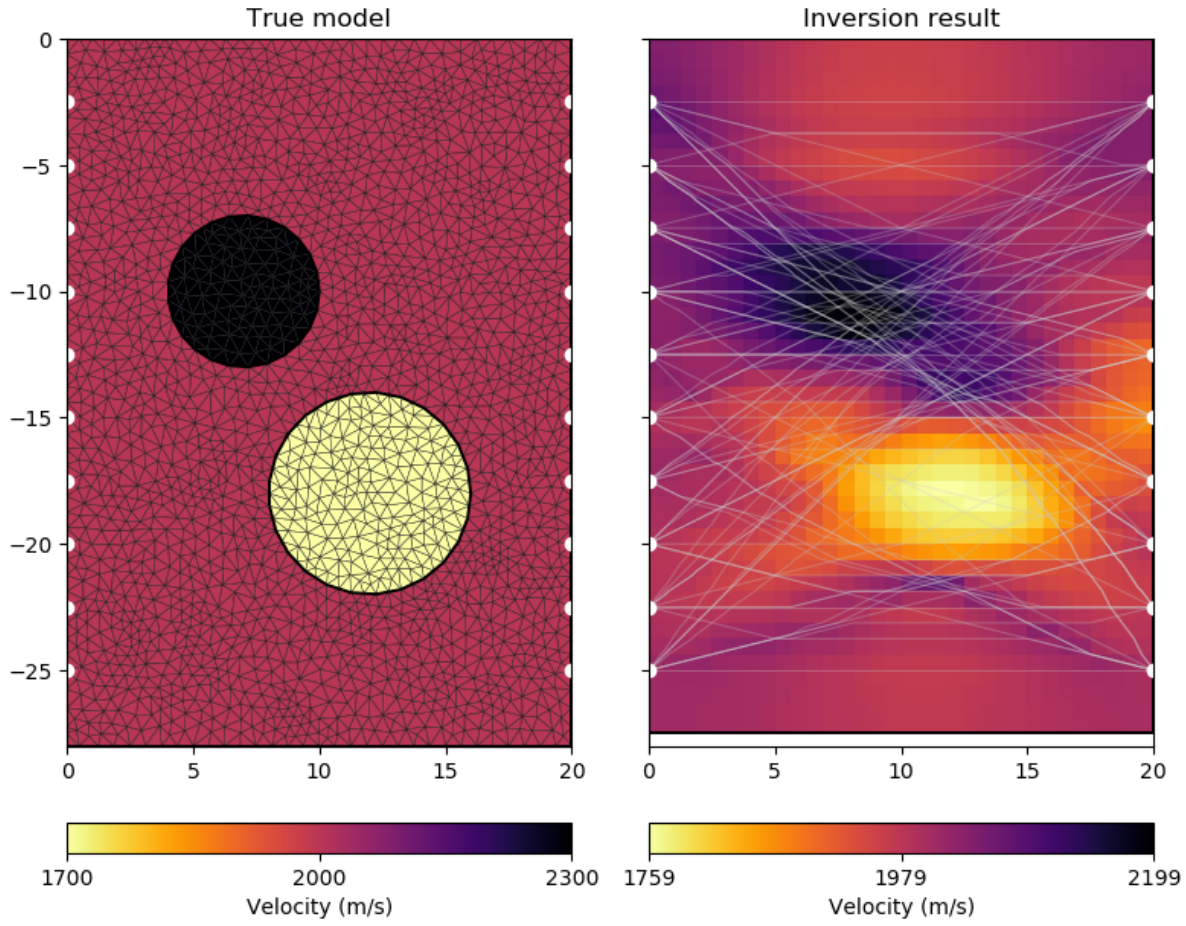
```
refinement = 0.25
x = np.arange(0, bh_spacing + refinement, sensor_spacing * refinement)
y = -np.arange(0.0, bh_length + 3, sensor_spacing * refinement)
mesh = pg.meshtools.createGrid(x, y)

ax, _ = pg.show(mesh, hold=True)
ax.plot(sensors[:, 0], sensors[:, 1], "ro")

invmodel = mgr.invert(data, mesh=mesh, secNodes=3, lam=1000, zWeight=1.0,
                      useGradient=False, verbose=True)
print("chi^2 = {:.2f}".format(mgr.inv.chi2()))  # Look at the data fit
```




```
fop: <pygimli.physics.traveltime.modelling.TravelTimeDijkstraModelling object␣
↪at 0x7fe8f561e590>
Data transformation: <pygimli.core._pygimli_.RTrans object at 0x7fe8f561e540>
Model transformation: <pygimli.core._pygimli_.RTransLog object at 0x7fe8f561e270>
min/max (data): 0.0096/0.02
min/max (error): 0.17%/0.2%
min/max (start model): 5.0e-04/5.0e-04
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
inv.iter 2 ... chi² = 3.33 (dPhi = 79.88%) lam: 1000
--------------------------------------------------------------------------------
inv.iter 3 ... chi² = 2.39 (dPhi = 25.49%) lam: 1000.0
--------------------------------------------------------------------------------
inv.iter 4 ... chi² = 1.48 (dPhi = 17.02%) lam: 1000.0
--------------------------------------------------------------------------------
inv.iter 5 ... chi² = 1.4 (dPhi = 3.05%) lam: 1000.0
--------------------------------------------------------------------------------
inv.iter 6 ... chi² = 1.27 (dPhi = 6.7%) lam: 1000.0
--------------------------------------------------------------------------------
inv.iter 7 ... chi² = 1.22 (dPhi = 2.41%) lam: 1000.0
--------------------------------------------------------------------------------
inv.iter 8 ... chi² = 1.13 (dPhi = 1.84%) lam: 1000.0
################################################################################
#                 Abort criteria reached: dPhi = 1.84 (< 2.0%)                 #
################################################################################
chi^2 = 1.13
```

Finally, we visualize the true model and the inversion result next to each other.

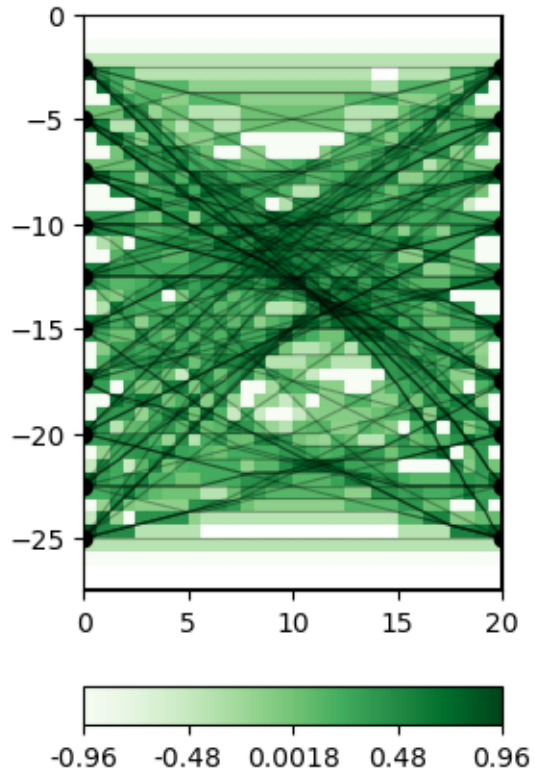
```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 7), sharex=True, sharey=True)
ax1.set_title("True model")
ax2.set_title("Inversion result")

pg.show(mesh_fwd, model, ax=ax1, showMesh=True,
        label=pg.unit('vel'), cMap=pg.cmap('vel'), nLevs=3)

for ax in (ax1, ax2):
    ax.plot(sensors[:, 0], sensors[:, 1], "wo")

mgr.showResult(ax=ax2, logScale=False, nLevs=3)
mgr.drawRayPaths(ax=ax2, color="0.8", alpha=0.3)
fig.tight_layout()
```



Coverage and ray paths

Note how the rays are attracted by the high velocity anomaly while circumventing the low-velocity region. This is also reflected in the coverage, which can be visualized as follows:

```
fig, ax = plt.subplots()
mgr.showCoverage(ax=ax, cMap="Greens")
mgr.drawRayPaths(ax=ax, color="k", alpha=0.3)
ax.plot(sensors[:, 0], sensors[:, 1], "ko")
```




```
[<matplotlib.lines.Line2D object at 0x7fe8fa245220>]
```

White regions indicate the model null space, i.e. cells that are not traversed by any ray.

Total running time of the script: (0 minutes 36.746 seconds)

6.7.2.3 Raypaths in layered and gradient models

This example performs raytracing for a two-layer and a vertical gradient model and compares the resulting traveltimes to existing analytical solutions. An approximation of the raypath is found by finding the shortest-path through a grid of nodes. The possible angular coverage is small when only corner points of a cell (primary nodes) are used for this purpose. The angular coverage, and hence the numerical accuracy of traveltime calculations, can be significantly improved by a few secondary nodes along the cell edges. Details can be found in Giroux & Larouche (2013).

```
from math import asin, tan
```

(continues on next page)

(continued from previous page)

```
import matplotlib.pyplot as plt
import numpy as np
import pygimli as pg
import pygimli.meshtools as mt
from pygimli.viewer.mpl import drawMesh

from pygimli.physics import TravelTimeManager
```

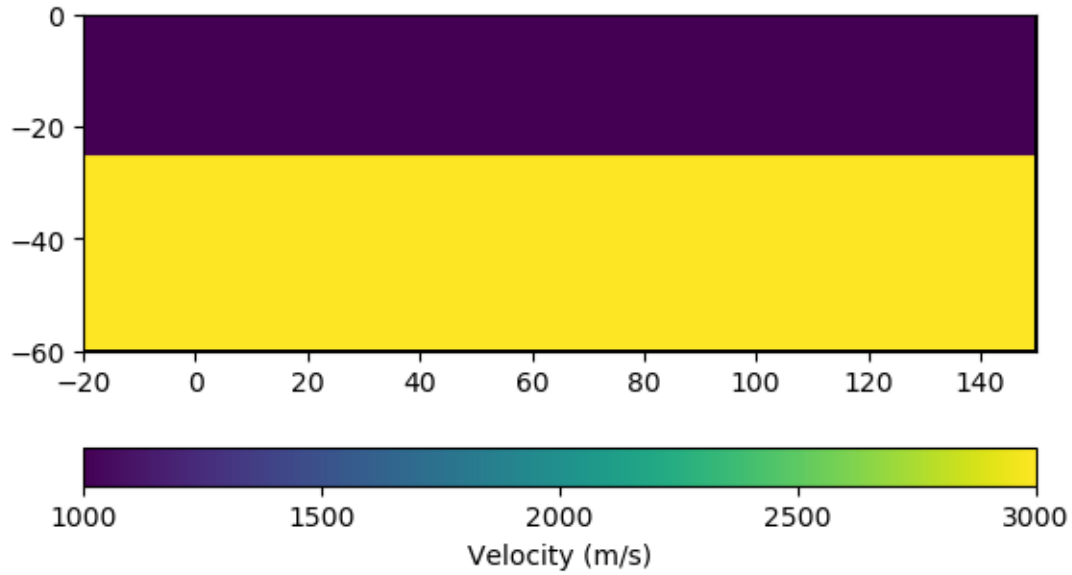
Two-layer model

We start by building a regular grid.

```
mesh_layered = mt.createGrid(
    np.arange(-20, 155, step=5, dtype=float), np.linspace(-60, 0, 13))
```

We now construct the velocity vector for the two-layer case by iterating over the cells. Cells above 25 m depth are assigned $v = 1000$ m/s and cells below are assigned $v = 3000$ m/s.

```
vel_layered = np.zeros(mesh_layered.cellCount())
for cell in mesh_layered.cells():
    if cell.center().y() < -25:
        vel = 3000.0
    else:
        vel = 1000.0
    vel_layered[cell.id()] = vel

pg.show(mesh_layered, vel_layered, label="Velocity (m/s)")
```




```
(<matplotlib.axes._subplots.AxesSubplot object at 0x7fe8fa7d7670>, <matplotlib.
↪colorbar.Colorbar object at 0x7fe8fa513eb0>)
```

We now define the analytical solution. The traveltime at a given offset x is the minimum of the direct and critically refracted wave, where the latter is governed by Snell's law.

```
def analyticalSolution2Layer(x, zlay=25, v1=1000, v2=3000):
    """Analytical solution for 2 layer case."""
    tdirect = np.abs(x) / v1  # direct wave
    alfa = asin(v1 / v2)  # critically refracted wave angle
    xreflec = tan(alfa) * zlay * 2.  # first critically refracted
    trefrac = (x - xreflec) / v2 + xreflec * v2 / v1**2
    return np.minimum(tdirect, trefrac)
```

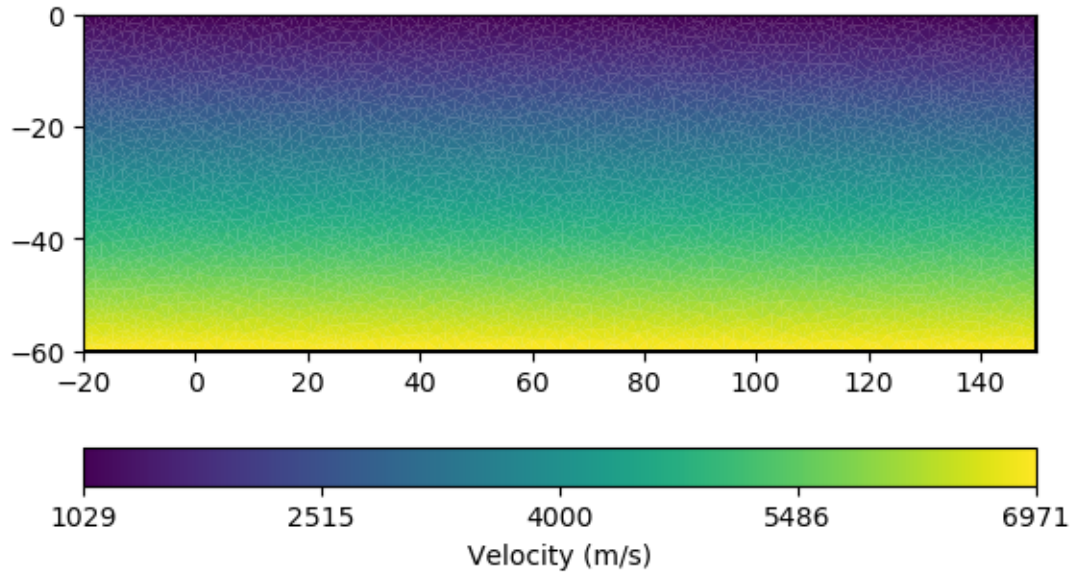
Vertical gradient model

We first create an unstructured mesh:

```
sensors = np.arange(131, step=10.0)
plc = mt.createWorld([-20, -60], [150, 0], worldMarker=False)
for pos in sensors:
    plc.createNode([pos, 0.0])
mesh_gradient = mt.createMesh(plc, quality=33, area=3)
```

A vertical gradient model, i.e. $v(z) = a + bz$, is defined per cell.

```
a = 1000
b = 100

vel_gradient = []
for node in mesh_gradient.nodes():
    vel_gradient.append(a + b * abs(node.y()))
vel_gradient = pg.meshtools.nodeDataToCellData(mesh_gradient,
                                               np.array(vel_gradient))
pg.show(mesh_gradient, vel_gradient, label="Velocity (m/s)")
```



```
(<matplotlib.axes._subplots.AxesSubplot object at 0x7fe8fa4ef310>, <matplotlib.
↪colorbar.Colorbar object at 0x7fe8f55456d0>)
```

The traveltime for a gradient velocity model is given by:

$$v = \left| b^{-1} cosh^{-1}\left(1 + \frac{b^2 x^2}{2a^2}\right)\right|$$

```
def analyticalSolutionGradient(x, a=1000, b=100):
    """Analytical solution for gradient model."""
    tdirect = np.abs(x) / a  # direct wave
    tmp = 1 + ((b**2 * np.abs(x)**2) / (2 * a**2))
    trefrac = np.abs(b**-1 * np.arccosh(tmp))
    return np.minimum(tdirect, trefrac)
```

The loop below calculates the travel times and makes the comparison plot.

```
fig, ax = plt.subplots(3, 2, figsize=(10, 10), sharex=True)

for j, (case, mesh, vel) in enumerate(zip(["layered", "gradient"],
                                          [mesh_layered, mesh_gradient],
                                          [vel_layered, vel_gradient])):
    pg.boxprint(case)
    if case == "gradient":
        ana = analyticalSolutionGradient
    elif case == "layered":
        ana = analyticalSolution2Layer
    for boundary in mesh.boundaries():
        boundary.setMarker(0)

    xmin, xmax = mesh.xmin(), mesh.xmax()
    mesh.createNeighborInfos()

    # In order to use the Dijkstra, we extract the surface positions␣
↪>0
    mx = pg.x(mesh)
    my = pg.y(mesh)
    px = np.sort(mx[my == 0.0])

    # A data container with index arrays named s (shot) and g␣
↪(geophones) is
    # created and filled with the positions and shot/geophone␣
↪indices.
    data = pg.DataContainer()
    data.registerSensorIndex('s')
    data.registerSensorIndex('g')

    for i, pxi in enumerate(px):
        data.createSensor([pxi, 0.0])
        if pxi == 0.0:
```

(continues on next page)

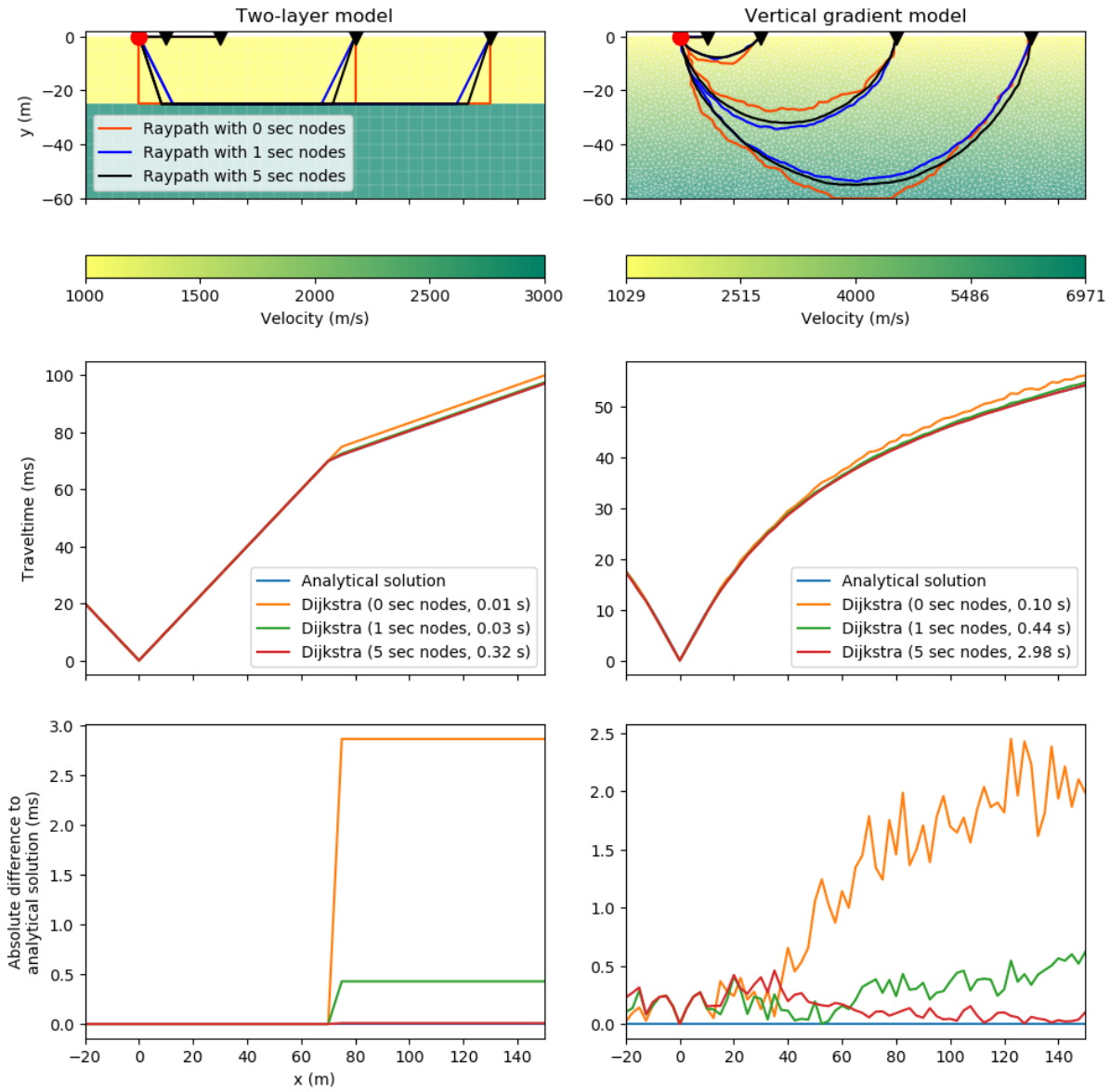
(continued from previous page)

```
            source = i

    nData = len(px)
    data.resize(nData)
    data['s'] = [source] * nData # only one shot at first sensor
    data['g'] = range(nData)  # and all sensors are receiver geophones

    # Draw initial mesh with velocity distribution
    pg.show(mesh, vel, ax=ax[0, j], label="Velocity (m/s)", hold=True,
            logScale=False, cMap="summer_r", coverage=0.7)
    drawMesh(ax[0, j], mesh, color="white", lw=0.21)

    # We compare the accuracy for 0-5 secondary nodes
    sec_nodes = [0, 1, 5]
    t_all = []
    durations = []
    paths = []

    mgr = TravelTimeManager()

    cols = ["orangered", "blue", "black"]
    recs = [1, 3, 8, 13]

    for i, n in enumerate(sec_nodes):

        # Perform traveltime calculations and log time with pg.tic() &
↪ pg.toc()
        pg.tic()
        res = mgr.simulate(vel=vel, scheme=data, mesh=mesh, secNodes=n)
        # We need to copy res['t'] here because res['t'] is a␣
↪reference to
        # an array in res, and res will be removed in the next␣
↪iteration.
        # Unfortunately, we don't have any reverence counting for␣
↪core objects yet.
        t_all.append(res['t'].array())
        durations.append(pg.dur())
        pg.toc("Raytracing with %d secondary nodes:" % n)

        for r, p in enumerate(recs):
            if r == 0:
                lab = "Raypath with %d sec nodes" % n
            else:
                lab = None

            recNode = mgr.fop.mesh().findNearestNode([sensors[p], 0.0])
            sourceNode = mgr.fop.mesh().findNearestNode([0.0, 0.0])

            path = mgr.fop.dijkstra.shortestPath(sourceNode, recNode)
```

(continues on next page)

(continued from previous page)

```
            points = mgr.fop.mesh().positions(withSecNodes=True)[path].array()
            ax[0, j].plot(points[:,0], points[:,1], cols[i], label=lab)

    t_ana = ana(px)

    # Upper subplot
    ax[1, j].plot(px, t_ana * 1000, label="Analytical solution")

    for i, n in enumerate(sec_nodes):
        ax[1, j].plot(px, t_all[i] * 1000,
                      label="Dijkstra (%d sec nodes, %.2f s)" % (n,␣
↪durations[i]))

    ax[2, j].plot(px, np.zeros_like(px), label="Zero line")  # to keep color␣
↪cycle

    for i, n in enumerate(sec_nodes):
        ax[2, j].plot(px, np.abs(t_all[i] - t_ana) * 1000)

    ax[1, j].legend()

    # Draw sensor positions for the selected receivers
    for p in recs:
        ax[0, j].plot(sensors[p], 0.0, "kv", ms=10)
    ax[0, j].plot(0.0, 0.0, "ro", ms=10)
    ax[0, j].set_ylim(mesh.ymin(), 2)

ax[0, 0].set_title("Two-layer model")
ax[0, 1].set_title("Vertical gradient model")
ax[0, 0].legend()
ax[0, 0].set_ylabel("y (m)")
ax[1, 0].set_ylabel("Traveltime (ms)")
ax[2, 0].set_ylabel("Absolute difference to\nanalytical solution (ms)")
ax[2, 0].set_xlabel("x (m)")
fig.tight_layout()
```




```
################################################################################
#                                    layered                                   #
################################################################################
Raytracing with 0 secondary nodes: Elapsed time is 0.01 seconds.
Raytracing with 1 secondary nodes: Elapsed time is 0.03 seconds.
Raytracing with 5 secondary nodes: Elapsed time is 0.32 seconds.
################################################################################
#                                    gradient                                  #
################################################################################
Raytracing with 0 secondary nodes: Elapsed time is 0.10 seconds.
Raytracing with 1 secondary nodes: Elapsed time is 0.44 seconds.
Raytracing with 5 secondary nodes: Elapsed time is 2.98 seconds.
```

Total running time of the script: (0 minutes 12.270 seconds)

6.7.2.4 Field data inversion ("Koenigsee")

This minimalistic example shows how to use the Refraction Manager to invert a field data set. Here, we consider the Koenigsee data set, which represents classical refraction seismics data set with slightly heterogeneous overburden and some high-velocity bedrock. The data file can be found in the pyGIMLi example data repository.

```
# We import pyGIMLi and the traveltime module.

import pygimli as pg
import pygimli.physics.traveltime as tt
```

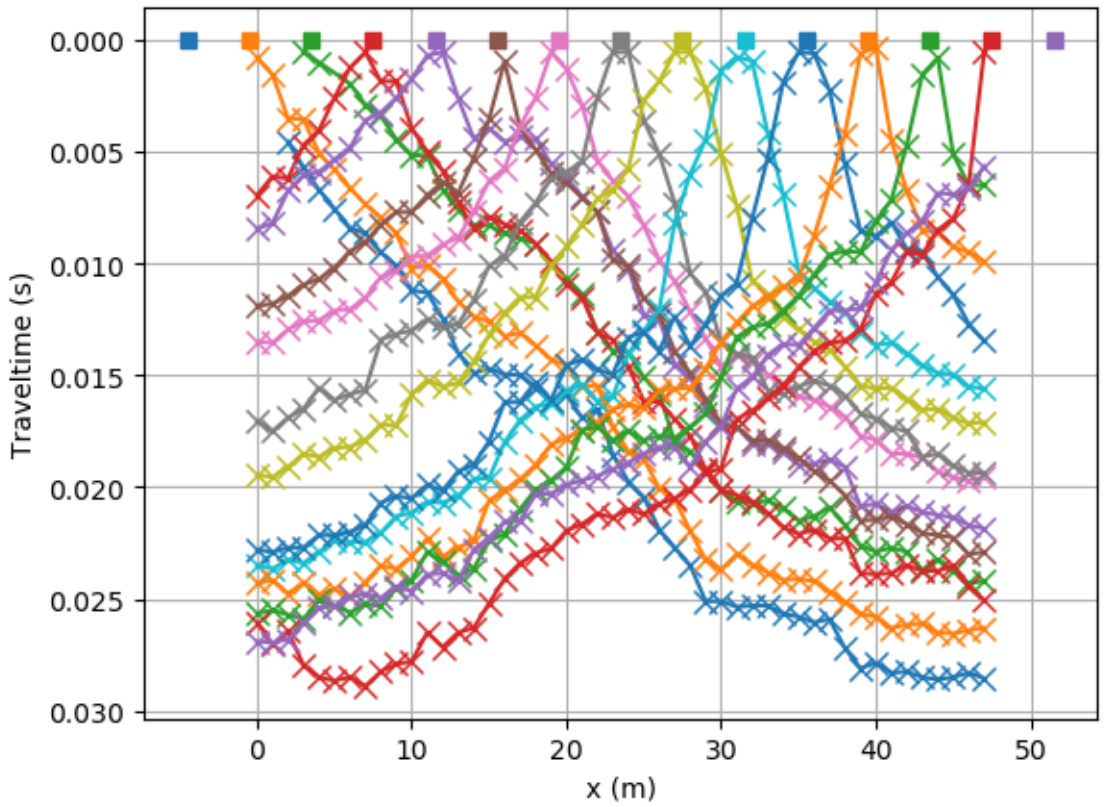
The helper function *pg.getExampleData* downloads the data set to a temporary location and loads it. Printing the data reveals that there are 714 data points using 63 sensors (shots and geophones) with the data columns s (shot), g (geophone), and t (traveltime). By default, there is also a validity flag.

```
data = pg.getExampleData("traveltime/koenigsee.sgt", verbose=True)
print(data)
```

```
[::::::::::::::::::::::::::::::::::::: 83% ::::::::::::::::::::::
↪] 8193 of 9844 complete
[::::::::::::::::::::::::::::::::::::: 100%␣
↪::::::::::::::::::::::::::::::::::::] 9844 of 9844 complete
md5: 641890bb17cb2bdf052cbc348669dfd0
Data: Sensors: 63 data: 714, nonzero entries: ['g', 's', 't', 'valid']
```

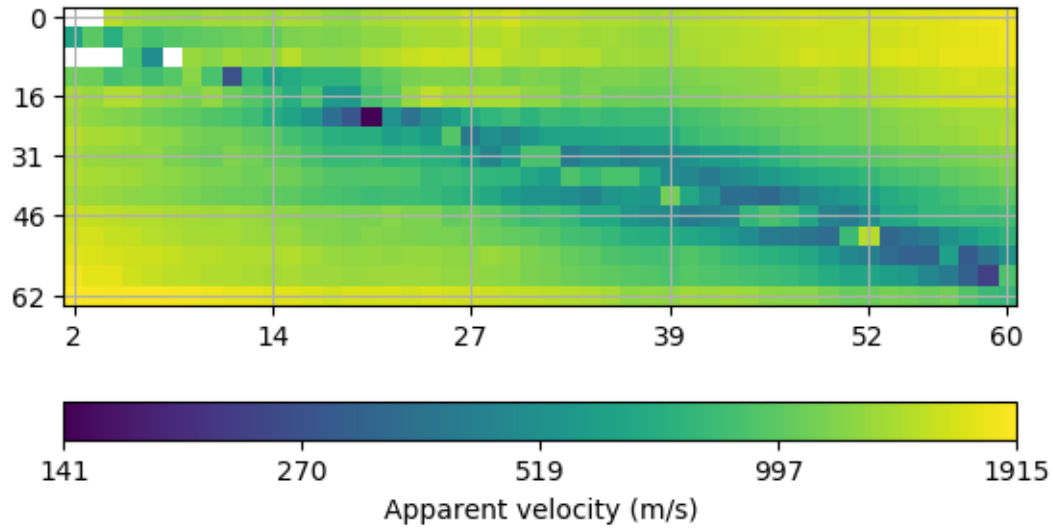
Let's have a look at the data in the form of traveltime curves.

```
fig, ax = pg.plt.subplots()
tt.drawFirstPicks(ax, data)
```



We initialize the refraction manager.

```
mgr = tt.TravelTimeManager(data)

# Alternatively, one can plot a matrix plot of apparent velocities␣
↪which is the
# more general function also making sense for crosshole data.

ax, cbar = mgr.showData()
```



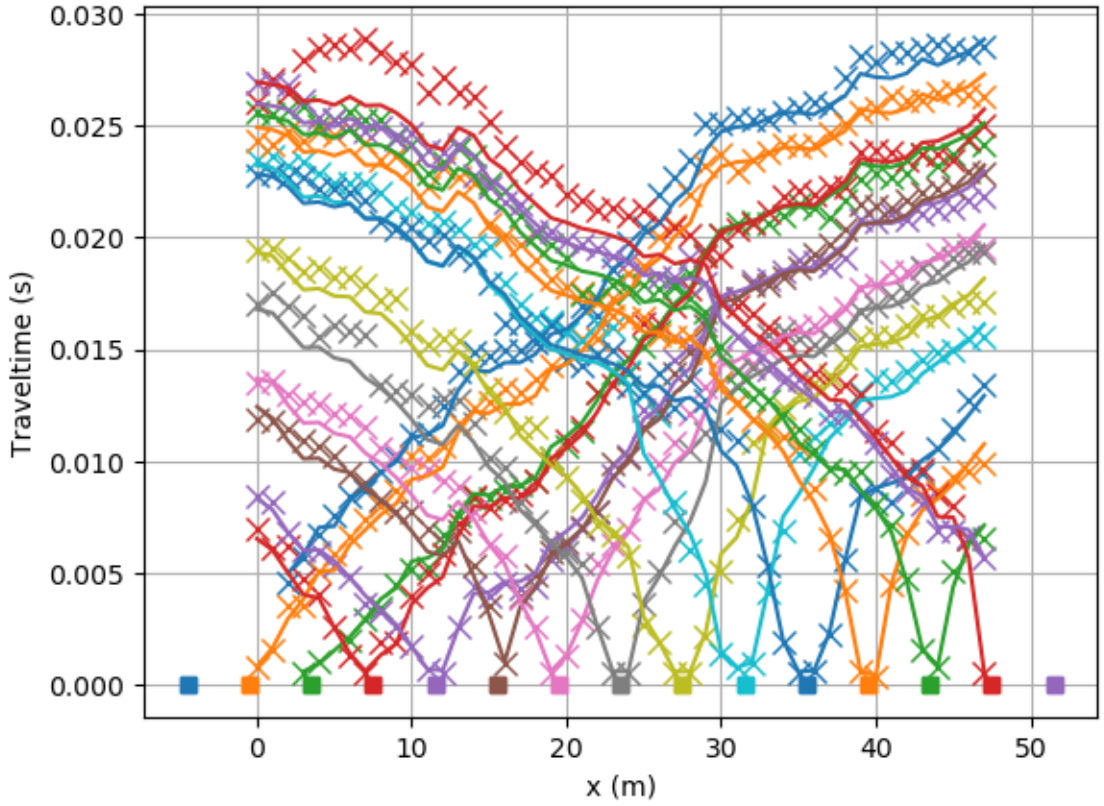
Finally, we call the *invert* method and plot the result.The mesh is created based on the sensor positions on-the-fly.

```
mgr.invert(secNodes=3, paraMaxCellSize=5.0,
           zWeight=0.2, vTop=500, vBottom=5000, verbose=1)
```

```
fop: <pygimli.physics.traveltime.modelling.TravelTimeDijkstraModelling object␣
↪at 0x7fe8fa708d10>
Data transformation: <pygimli.core._pygimli_.RTrans object at 0x7fe8fa71a360>
Model transformation (cumulative):
         0 <pygimli.core._pygimli_.RTransLogLU object at 0x7fe8fa0f96a0>
min/max (data): 3.5e-04/0.03
min/max (error): 3%/3%
min/max (start model): 2.0e-04/0.002
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
inv.iter 2 ... chi² = 9.5 (dPhi = 29.3%) lam: 20
--------------------------------------------------------------------------------
inv.iter 3 ... chi² = 8.28 (dPhi = 12.52%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 4 ... chi² = 7.15 (dPhi = 13.17%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 5 ... chi² = 6.08 (dPhi = 14.57%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 6 ... chi² = 5.23 (dPhi = 13.51%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 7 ... chi² = 4.75 (dPhi = 8.61%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 8 ... chi² = 4.35 (dPhi = 7.55%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 9 ... chi² = 4.2 (dPhi = 3.06%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 10 ... chi² = 4.01 (dPhi = 3.99%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 11 ... chi² = 3.9 (dPhi = 2.53%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 12 ... chi² = 3.78 (dPhi = 2.65%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 13 ... chi² = 3.71 (dPhi = 1.65%) lam: 20.0
################################################################################
#                 Abort criteria reached: dPhi = 1.65 (< 2.0%)                 #
################################################################################

1204 [1160.9584092975203,...,2558.8712173776776]
```

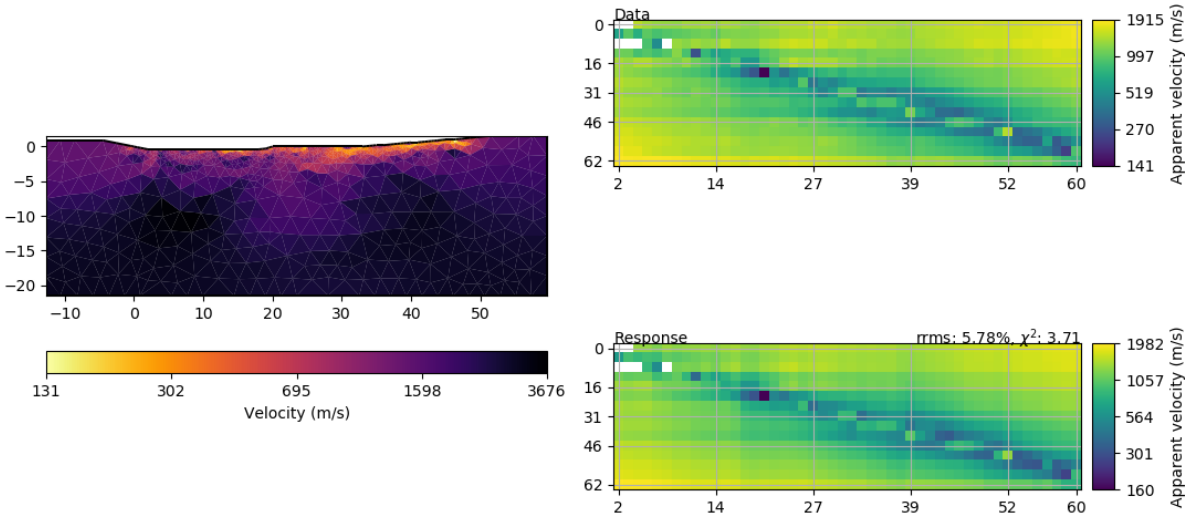
First have a look at the data fit. Plot the measured (crosses) and modelled (lines) traveltimes.

```
ax, cbar = mgr.showData(firstPicks=True, linewidth=0)
tt.drawFirstPicks(ax, data, mgr.inv.response, marker=None)
```



Show resulting tomogram along with fit. You may want to save your results.

```
mgr.showResultAndFit()
mgr.saveResult()  # saves the results (mesh, velocity, vtk) in a folder
```




```
'./20221209-00.33/TravelTimeManager'
```

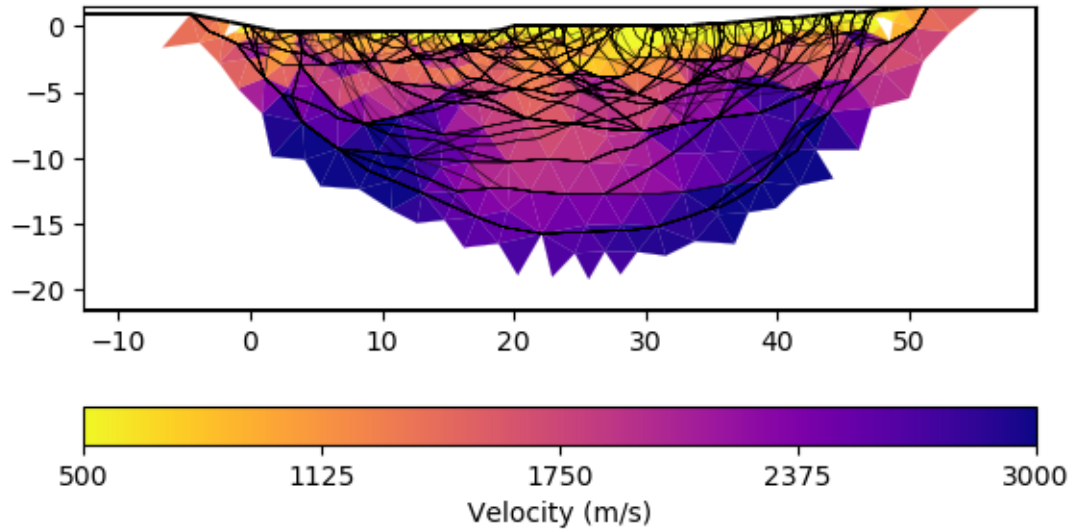
You can plot only the model and customize with a bunch of keywords

```
ax, cbar = mgr.showResult(logScale=False, cMin=500, cMax=3000, cMap="plasma_r",
                          coverage=mgr.standardizedCoverage())
mgr.drawRayPaths(ax=ax, color="k", lw=0.3, alpha=0.5)
```

(continues on next page)

(continued from previous page)

```
# mgr.coverage() yields the ray coverage in m and␣
↪standardizedCoverage as 0/1
```




```
<matplotlib.collections.LineCollection object at 0x7fe8ab0fb3d0>
```

You can play around with the gradient starting model (*vTop* and *vBottom* arguments) and the regularization strength *lam* and customize the mesh.

Total running time of the script: (0 minutes 34.286 seconds)

6.7.2.5 Refraction in 3D

This example shows refracted ray paths in a three-dimensional vertical gradient medium.

Note: This is a placeholder/proof-of-concept. The code should be refactored partly to *tt.showRayPaths()*

```
import numpy as np
import pygimli as pg
import pygimli.meshtools as mt
from pygimli.physics import traveltime
from pygimli.viewer.pv import drawSensors

pyvista = pg.optImport("pyvista")
```

Build mesh.

```
depth = 15
width = 30
plc = mt.createCube(size=[width, width, depth], pos=[0, 0, -depth/2], area=5)
```

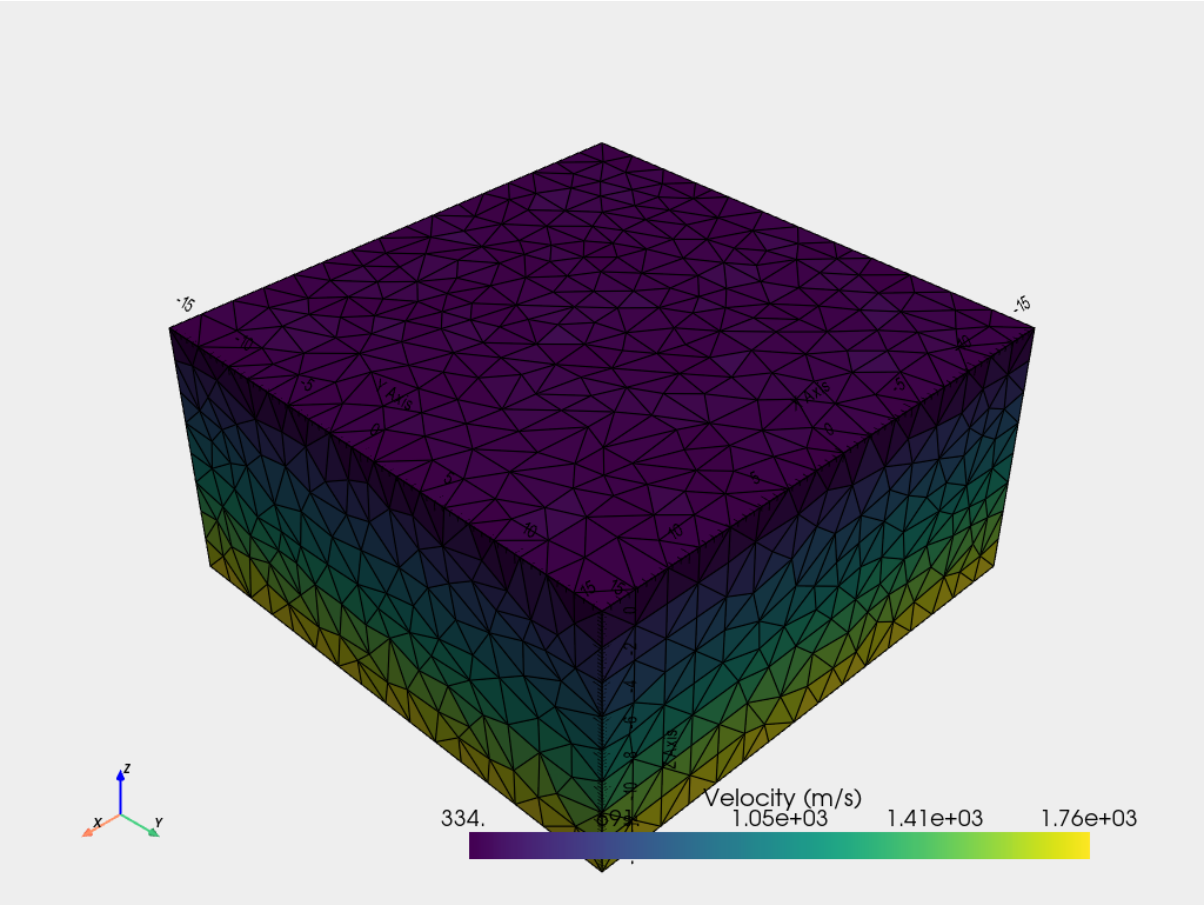
(continues on next page)

(continued from previous page)

```
n_sensors = 8
sensors = np.zeros((n_sensors, 3))
sensors[0, 0] = 15
sensors[0, 1] = -10
sensors[1:, 0] = -15
sensors[1:, 1] = np.linspace(-15, 15, n_sensors - 1)

for pos in sensors:
    plc.createNode(pos)
mesh = mt.createMesh(plc)
mesh.createSecondaryNodes(1)
```

Create vertical gradient model.

```
vel = 300 + -pg.z(mesh.cellCenters()) * 100

if pyvista:
    label = pg.utils.unit("vel")
    pg.show(mesh, vel, label=label, showMesh=True)
```



Set-up data container.

```
data = traveltime.createRAData(sensors)
data.markInvalid(data("s") > 1)
```

(continues on next page)

(continued from previous page)

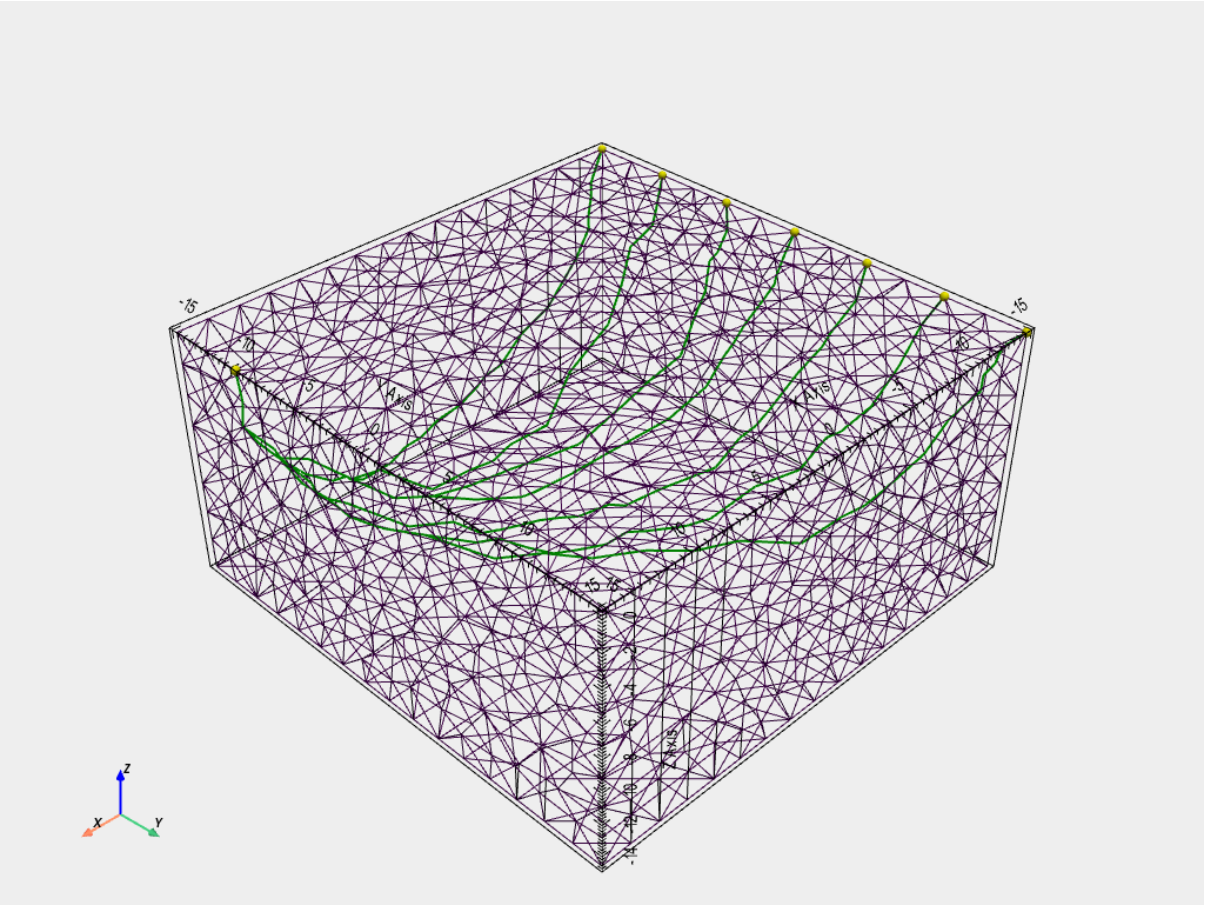
```
data.set("t", np.zeros(data.size()))
data.removeInvalid()
```

Do raytracing.

```
fop = pg.core.TravelTimeDijkstraModelling(mesh, data)

# This is to show single raypaths.
dij = pg.core.Dijkstra(fop.createGraph(1 / vel))
dij.setStartNode(mesh.findNearestNode([15, -10, 0]))

rays = []
for receiver in sensors[1:]:
    ni = dij.shortestPathTo(mesh.findNearestNode(receiver))
    pos = mesh.positions(withSecNodes=True)[ni]
    segs = np.zeros((len(pos), 3))
    segs[:, 0] = pg.x(pos)
    segs[:, 1] = pg.y(pos)
    segs[:, 2] = pg.z(pos)
    rays.append(segs)
```

Plot final ray paths.

```
if pyvista:
    plotter, _ = pg.show(mesh, style='wireframe', line_width=0.1,
                         hold=True)
    drawSensors(plotter, sensors, diam=0.5, color='yellow')

    for ray in rays:
        for i in range(len(ray) - 1):
            start = tuple(ray[i])
            stop = tuple(ray[i + 1])
            line = pyvista.Line(start, stop)
            plotter.add_mesh(line, color='green', line_width=2)

    plotter.show()
```



6.7.3 Electrical resistivity tomography

6.7.3.1 2D ERT modeling and inversion

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

import pygimli as pg
import pygimli.meshtools as mt
from pygimli.physics import ert
```

Geometry definition

Create geometry definition for the modelling domain. `worldMarker=True` indicates the default boundary conditions for the ERT

```
world = mt.createWorld(start=[-50, 0], end=[50, -50], layers=[-1, -5],
                       worldMarker=True)
```

Create some heterogeneous circular anomaly

```
block = mt.createCircle(pos=[-5, -3.], radius=[4, 1], marker=4,
                        boundaryMarker=10, area=0.1)
```

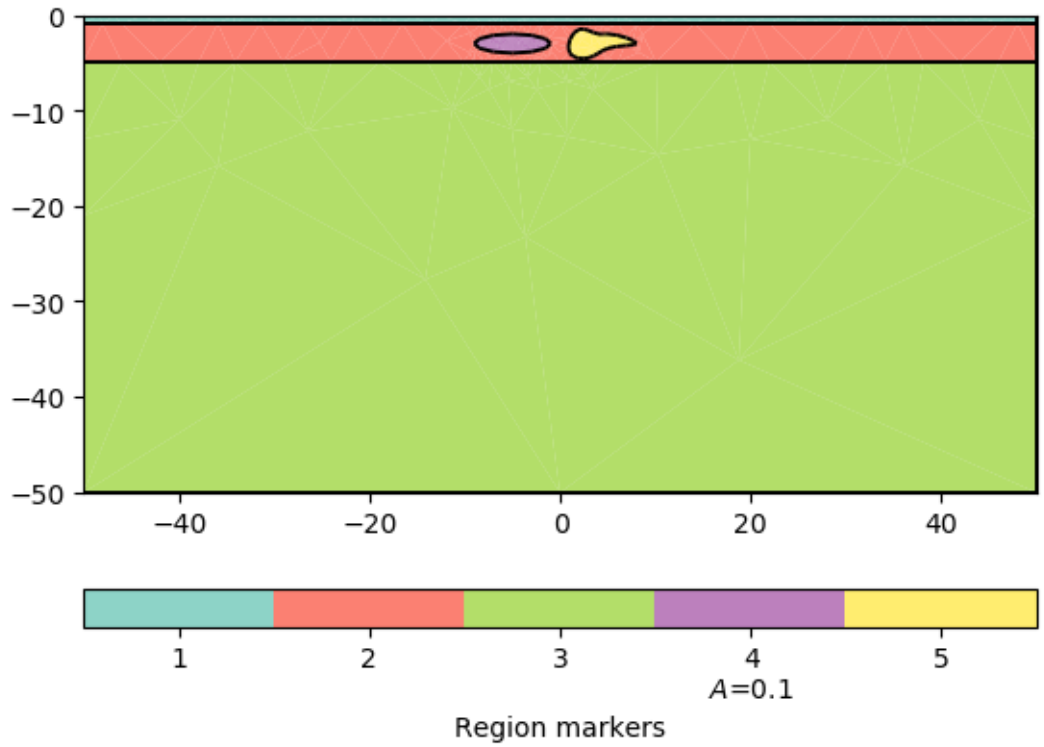
```
poly = mt.createPolygon([(1,-4), (2,-1.5), (4,-2), (5,-2),
                         (8,-3), (5,-3.5), (3,-4.5)], isClosed=True,
                         addNodes=3, interpolate='spline', marker=5)
```

Merge geometry definition into a Piecewise Linear Complex (PLC)

```
geom = world + block + poly
```

Optional: show the geometry

```
pg.show(geom)
```




```
(<matplotlib.axes._subplots.AxesSubplot object at 0x7fe8ab1aa9a0>, None)
```

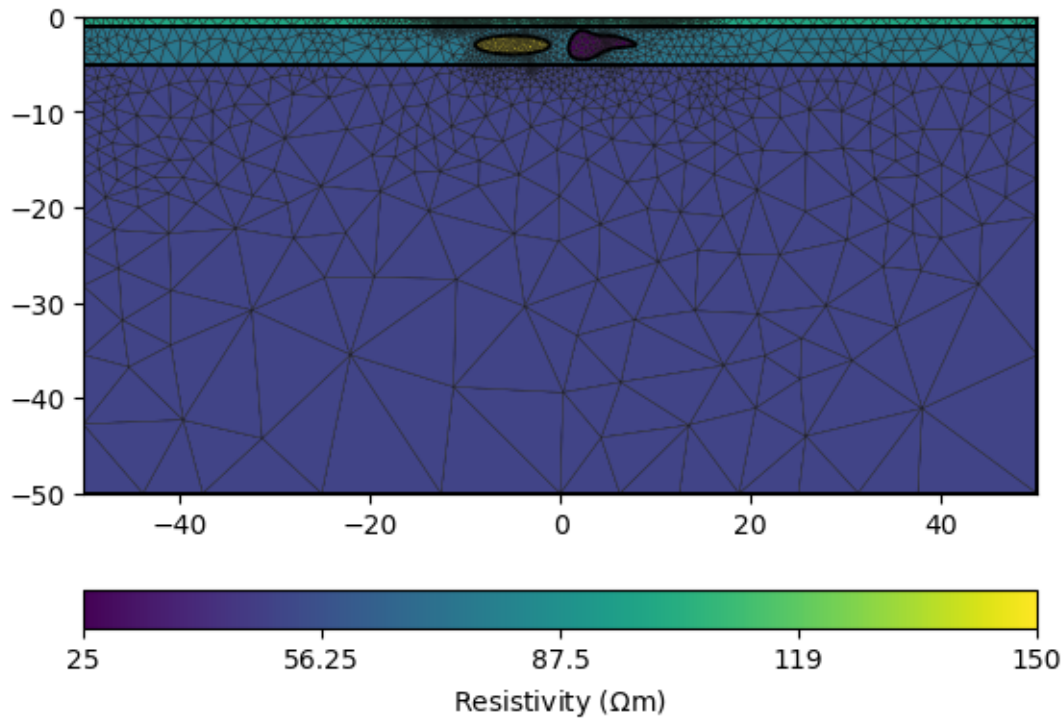
Synthetic data generation

Create a Dipole Dipole ('dd') measuring scheme with 21 electrodes.

```
scheme = ert.createData(elecs=np.linspace(start=-15, stop=15, num=21),
                        schemeName='dd')
```

Put all electrode (aka sensors) positions into the PLC to enforce mesh refinement. Due to experience, its convenient to add further refinement nodes in a distance of 10% of electrode spacing to achieve sufficient numerical accuracy.

```
for p in scheme.sensors():
    geom.createNode(p)
    geom.createNode(p - [0, 0.1])

# Create a mesh for the finite element modelling with appropriate␣
↪mesh quality.
mesh = mt.createMesh(geom, quality=34)

# Create a map to set resistivity values in the appropriate regions
# [[regionNumber, resistivity], [regionNumber, resistivity], [...]
rhomap = [[1, 100.],
          [2, 75.],
          [3, 50.],
          [4, 150.],
          [5, 25]]

# Take a look at the mesh and the resistivity distribution
pg.show(mesh, data=rhomap, label=pg.unit('res'), showMesh=True)
```




```
(<matplotlib.axes._subplots.AxesSubplot object at 0x7fe8f562d2b0>, <matplotlib.
↪colorbar.Colorbar object at 0x7fe8f54fcca0>)
```

Perform the modeling with the mesh and the measuring scheme itself and return a data container with apparent resistivity values, geometric factors and estimated data errors specified by the noise setting. The noise is also added to the data. Here 1% plus 1µV. Note, we force a specific noise seed as we want reproducable results for testing purposes.

```
data = ert.simulate(mesh, scheme=scheme, res=rhomap, noiseLevel=1,
                    noiseAbs=1e-6, seed=1337)
```

(continues on next page)

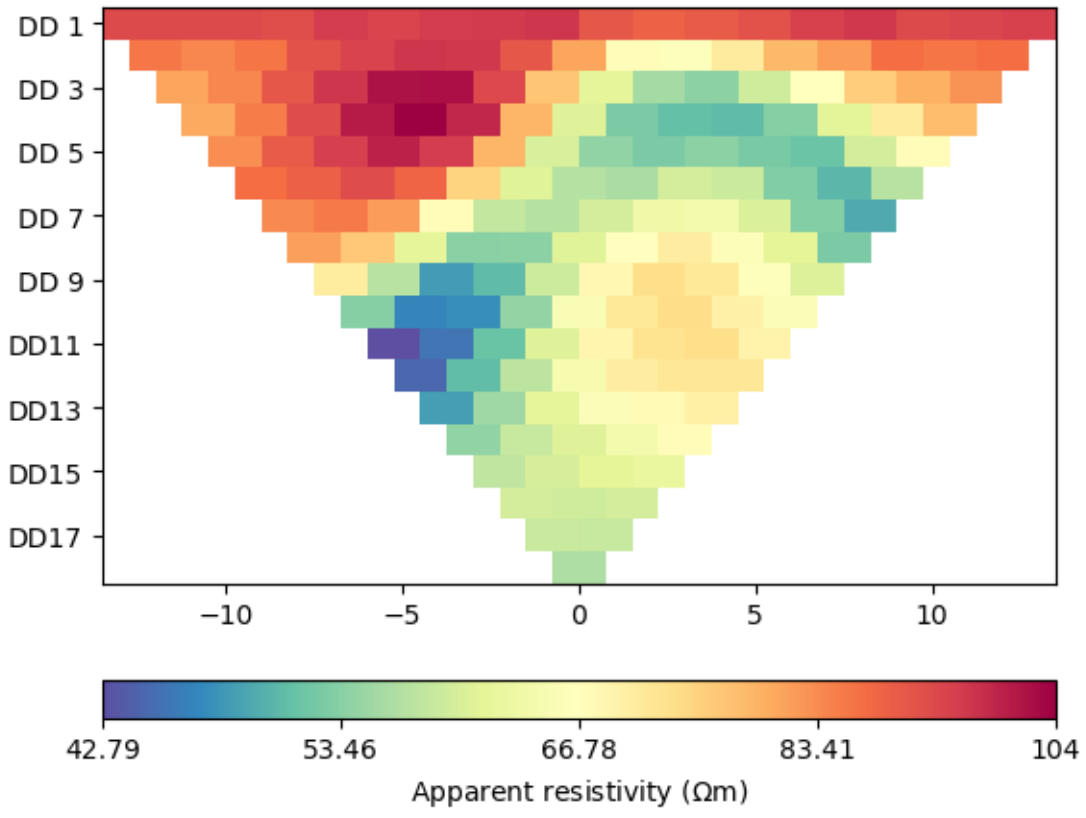
(continued from previous page)

```
pg.info(np.linalg.norm(data['err']), np.linalg.norm(data['rhoa']))
pg.info('Simulated data', data)
pg.info('The data contains:', data.dataMap().keys())

pg.info('Simulated rhoa (min/max)', min(data['rhoa']), max(data['rhoa']))
pg.info('Selected data noise %(min/max)', min(data['err'])*100, max(data['err
↪'])*100)
```

```
relativeError set to a value > 0.5 .. assuming this is a percentage Error level␣
↪dividing them by 100
Data error estimate (min:max)  0.010000294838286121 : 0.01056761917552525
```

Optional: you can filter all values and tokens in the data container. Its possible that there are some negative data values due to noise and huge geometric factors. So we need to remove them.

```
data.remove(data['rhoa'] < 0)
pg.info('Filtered rhoa (min/max)', min(data['rhoa']), max(data['rhoa']))

# You can save the data for further use
data.save('simple.dat')

# You can take a look at the data
ert.show(data)
```



```
(<matplotlib.axes._subplots.AxesSubplot object at 0x7fe8f56b1f10>, <matplotlib.
↪colorbar.Colorbar object at 0x7fe8f555b6d0>)
```

Inversion with the ERTManager

Initialize the ERTManager, e.g. with a data container or a filename.

```
mgr = ert.ERTManager('simple.dat')
```

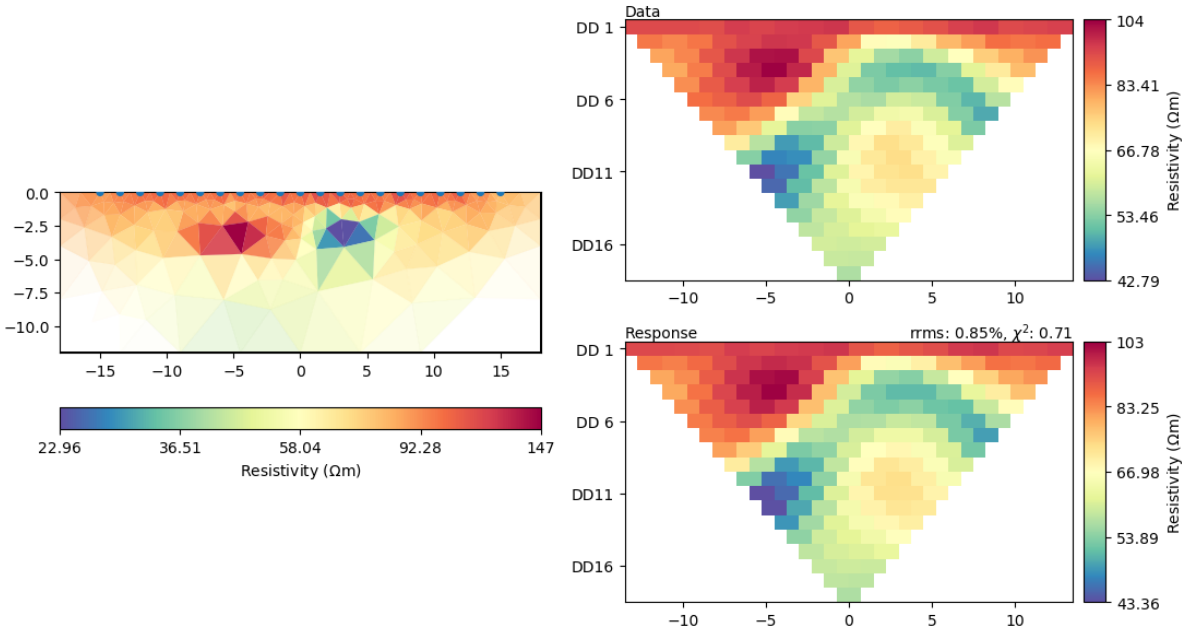
Run the inversion with the preset data. The Inversion mesh will be created with default settings.

```
inv = mgr.invert(lam=20, verbose=True)
np.testing.assert_approx_equal(mgr.inv.chi2(), 0.7, significant=1)
```

```
fop: <pygimli.physics.ert.ertModelling.ERTModelling object at 0x7fe8fa462770>
Data transformation: <pygimli.core._pygimli_.RTransLogLU object at␣
↪0x7fe8fa462950>
Model transformation: <pygimli.core._pygimli_.RTransLog object at 0x7fe8fa4621d0>
min/max (data): 42.79/104
min/max (error): 1%/1.06%
min/max (start model): 67.94/67.94
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
inv.iter 2 ... chi² = 2.99 (dPhi = 74.55%) lam: 20
--------------------------------------------------------------------------------
inv.iter 3 ... chi² = 0.71 (dPhi = 59.71%) lam: 20.0


################################################################################
#                    Abort criterion reached: chi² <= 1 (0.71)                 #
################################################################################
```

Let the ERTManger show you the model of the last successful run and how it fits the data. Shows data, model response, and model.

```
mgr.showResultAndFit()
meshPD = pg.Mesh(mgr.paraDomain) # Save copy of para mesh for plotting␣
↪later
```



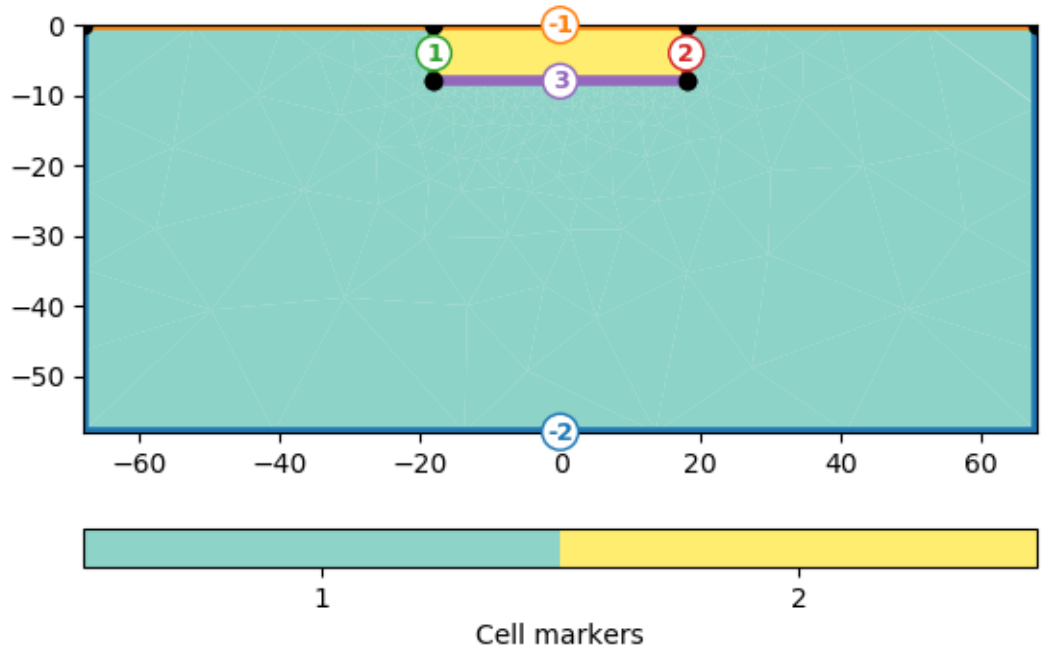
You can also provide your own mesh (e.g., a structured grid if you like them) Note, that x and y coordinates needs to be in ascending order to ensure that all the cells in the grid have the correct orientation, i.e., all cells need to be numbered counter-clockwise and the boundary normal directions need to point outside.

```
inversionDomain = pg.createGrid(x=np.linspace(start=-18, stop=18, num=33),
                                y=-pg.cat([0], pg.utils.grange(0.5, 8, n=5))[::-
↪1],
                                marker=2)
```

Inversion with custom mesh

The inversion domain for ERT problems needs a boundary that represents the far regions in the subsurface of the halfspace. Give a cell marker lower than the marker for the inversion region, the lowest cell marker in the mesh will be the inversion boundary region by default.

```
grid = pg.meshtools.appendTriangleBoundary(inversionDomain, marker=1,
                                           xbound=50, ybound=50)
pg.show(grid, markers=True)
```




```
(<matplotlib.axes._subplots.AxesSubplot object at 0x7fe8fa7fde20>, <matplotlib.
↪colorbar.Colorbar object at 0x7fe8f539aac0>)
```

The Inversion can be called with data and mesh as argument as well

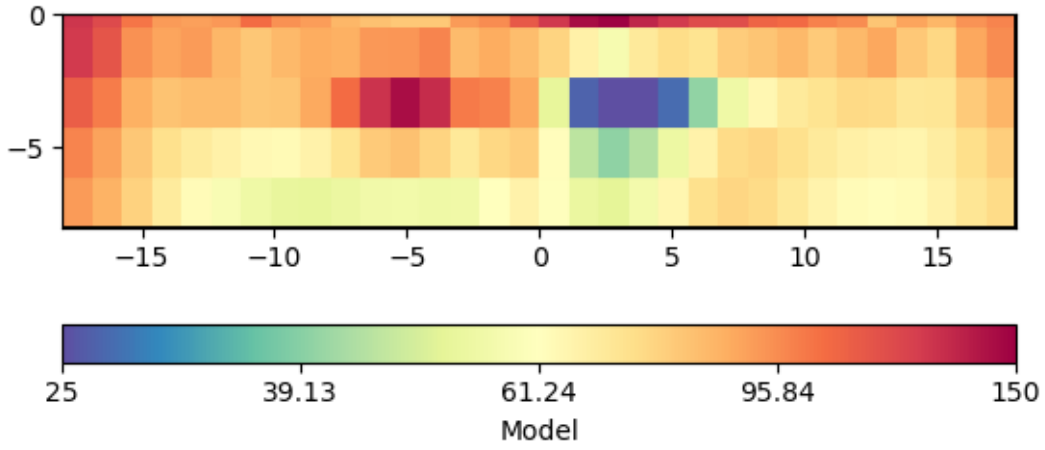
```
model = mgr.invert(data, mesh=grid, lam=20, verbose=True)
np.testing.assert_approx_equal(mgr.inv.chi2(), 1.4, significant=2)
```

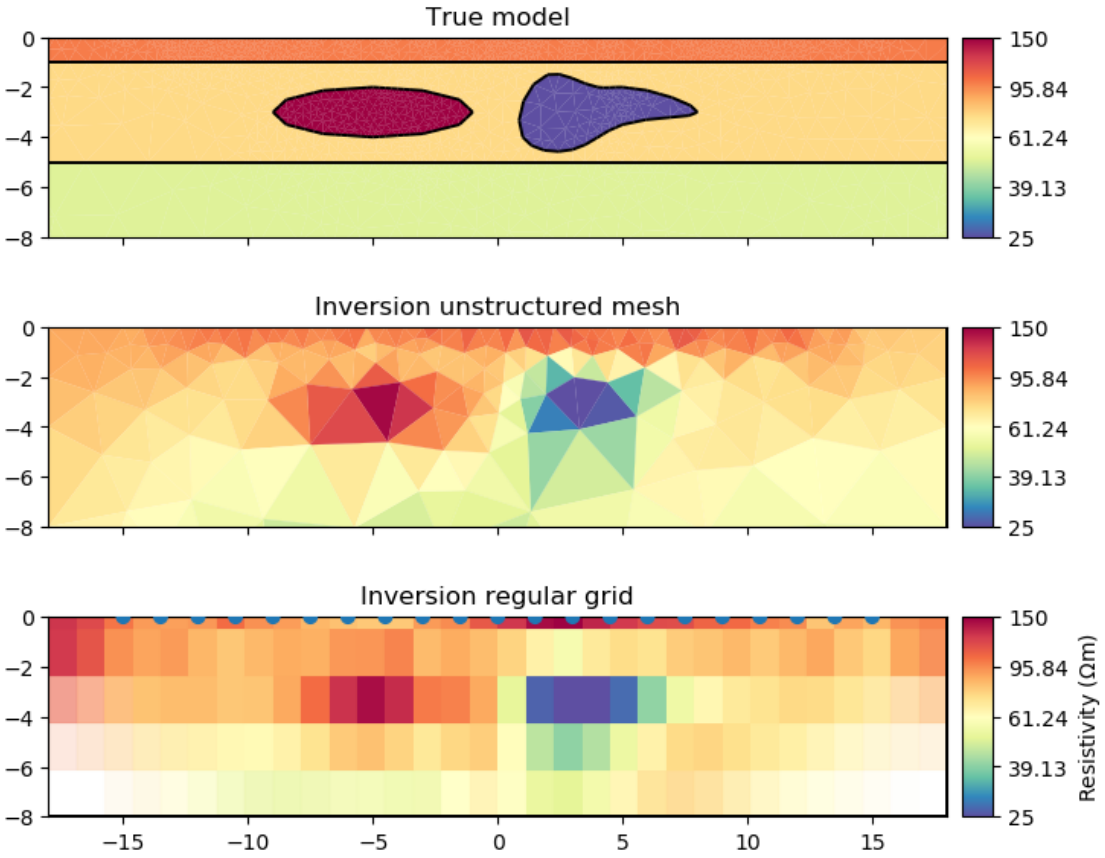
```
fop: <pygimli.physics.ert.ertModelling.ERTModelling object at 0x7fe8fa462770>
Data transformation: <pygimli.core._pygimli_.RTransLogLU object at␣
↪0x7fe8fa462950>
Model transformation: <pygimli.core._pygimli_.RTransLog object at 0x7fe8fa4621d0>
min/max (data): 42.79/104
min/max (error): 1%/1.06%
min/max (start model): 67.94/67.94
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
inv.iter 2 ... chi² = 5.37 (dPhi = 91.06%) lam: 20
--------------------------------------------------------------------------------
inv.iter 3 ... chi² = 2.02 (dPhi = 44.6%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 4 ... chi² = 1.51 (dPhi = 11.03%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 5 ... chi² = 1.43 (dPhi = 2.04%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 6 ... chi² = 1.42 (dPhi = 0.09%) lam: 20.0
################################################################################
#                  Abort criteria reached: dPhi = 0.09 (< 2.0%)                #
################################################################################
```

Visualization

You can of course get access to mesh and model and plot them for your own. Note that the cells of the parametric domain of your mesh might be in a different order than the values in the model array if regions are used. The manager can help to permutate them into the right order.

```
modelPD = mgr.paraModel(model)  # do the mapping
pg.show(mgr.paraDomain, modelPD, label='Model', cMap='Spectral_r',
        logScale=True, cMin=25, cMax=150)

pg.info('Inversion stopped with chi^2 = {0:.3}'.format(mgr.fw.chi2()))

fig, (ax1, ax2, ax3) = plt.subplots(3,1, sharex=True, sharey=True, figsize=(8,
↪7))

pg.show(mesh, rhomap, ax=ax1, hold=True, cMap="Spectral_r", logScale=True,
        orientation="vertical", cMin=25, cMax=150)
pg.show(meshPD, inv, ax=ax2, hold=True, cMap="Spectral_r", logScale=True,
        orientation="vertical", cMin=25, cMax=150)
mgr.showResult(ax=ax3, cMin=25, cMax=150, orientation="vertical")

labels = ["True model", "Inversion unstructured mesh", "Inversion regular grid"]
for ax, label in zip([ax1, ax2, ax3], labels):
    ax.set_xlim(mgr.paraDomain.xmin(), mgr.paraDomain.xmax())
    ax.set_ylim(mgr.paraDomain.ymin(), mgr.paraDomain.ymax())
    ax.set_title(label)
```





6.7.3.2 ERT field data with topography

Simple example of data measured over a slagdump demonstrating:

- 2D inversion with topography
- geometric factor generation
- topography effect

```
import pygimli as pg
from pygimli.physics import ert
```

Get some example data with topography, typically by a call like data = ert.load("filename.dat") that supports various file formats

```
data = pg.getExampleFile('ert/slagdump.ohm', load=True, verbose=True)
print(data)
```

```
[::::::::::::::::::::::::::::::::::: 100%␣
↪::::::::::::::::::::::::::::::::::::] 5435 of 5435 complete
md5: f27293809709e8fdc89bfb0a55e6e1ec
Data: Sensors: 38 data: 222, nonzero entries: ['a', 'b', 'm', 'n', 'r', 'valid
↪']
```

The data file does not contain geometric factors (token field 'k'), so we create them based on the given topography.

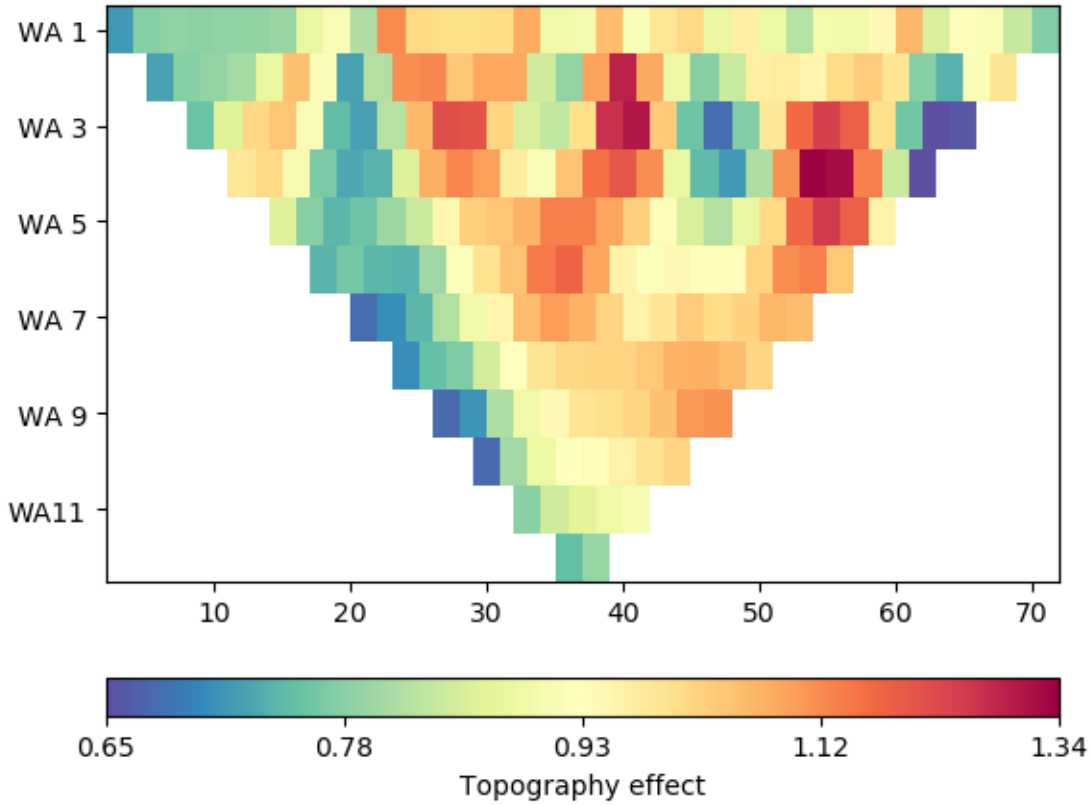
```
data['k'] = ert.createGeometricFactors(data, numerical=True)
```

We initialize the ERTManager for further steps and eventually inversion.

```
mgr = ert.ERTManager(sr=False)
```

It might be interesting to see the topography effect, i.e the ratio between the numerically computed geometry factor and the analytical formula

```
k0 = ert.createGeometricFactors(data)
ert.showData(data, vals=k0/data['k'], label='Topography effect')
```




```
(<matplotlib.axes._subplots.AxesSubplot object at 0x7fe8f582ab50>, <matplotlib.
↪colorbar.Colorbar object at 0x7fe8da959160>)
```

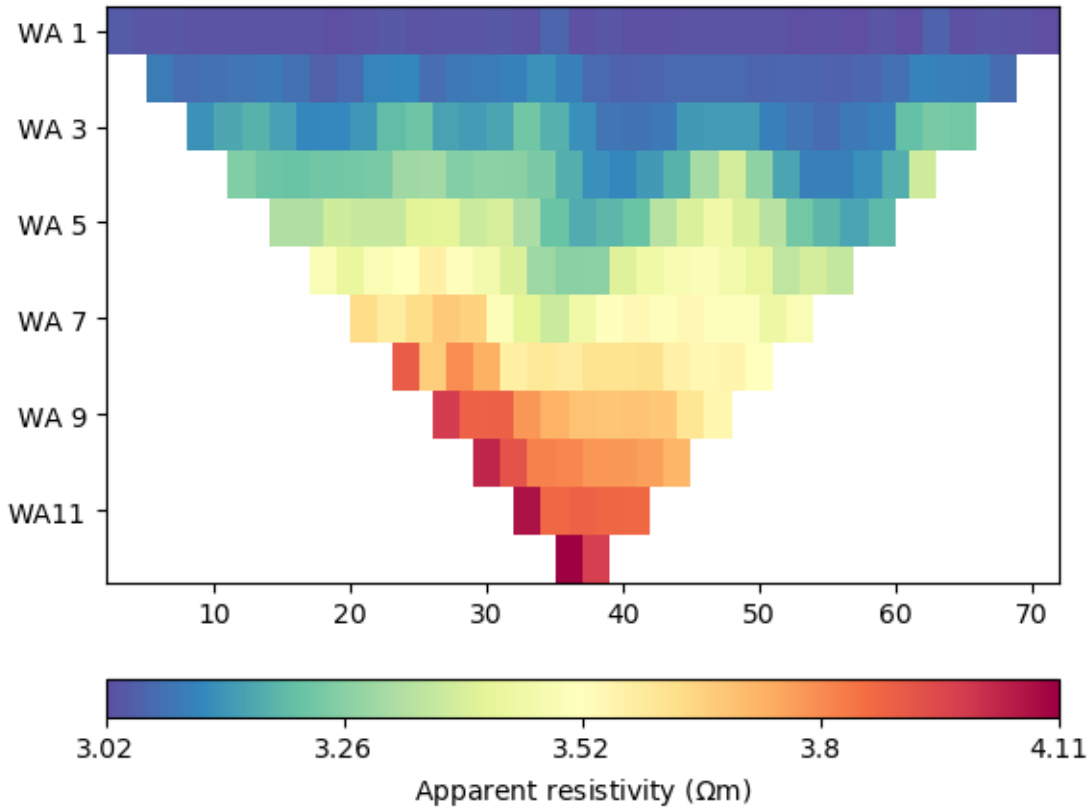
The data container has no apparent resistivities (token field 'rhoa') yet. We can let the Manager fix this later for us (as we now have the 'k' field), or we do it manually.

```
mgr.checkData(data)
print(data)
```

```
Data: Sensors: 38 data: 222, nonzero entries: ['a', 'b', 'k', 'm', 'n', 'r',
↪'rhoa', 'valid']
```

The data container does not necessarily contain data errors data errors (token field 'err'), requiring us to enter data errors. We can let the manager guess some defaults for us automaticly or set them manually

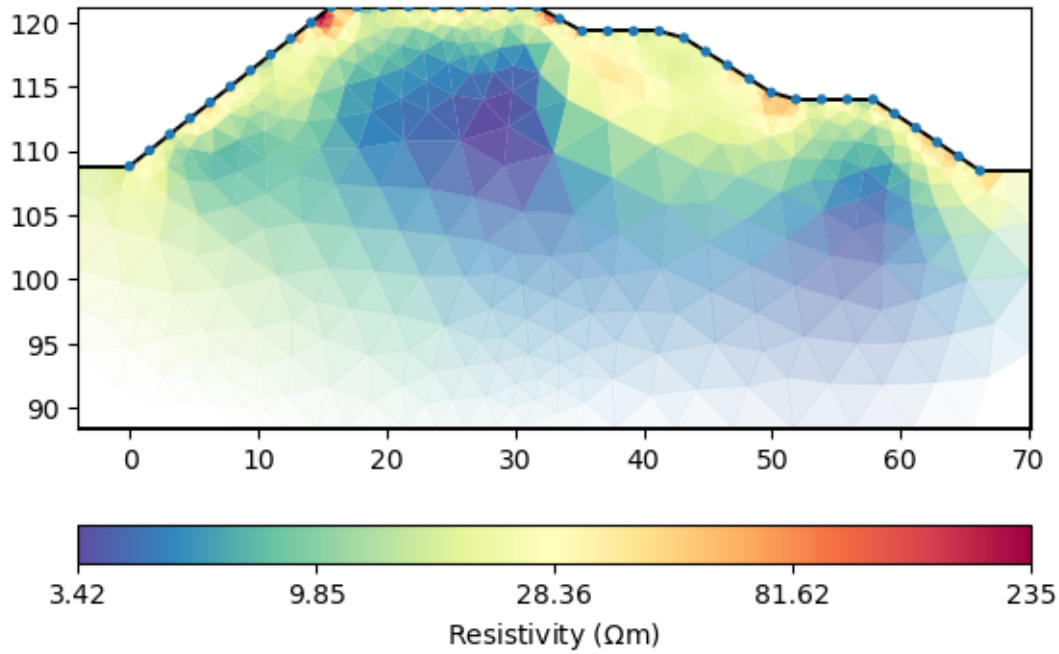
```
data['err'] = ert.estimateError(data, absoluteUError=5e-5, relativeError=0.03)
# or manually:
# data['err'] = data_errors  # somehow
ert.show(data, data['err']*100)
```




```
(<matplotlib.axes._subplots.AxesSubplot object at 0x7fe8f57cf400>, <matplotlib.
↪colorbar.Colorbar object at 0x7fe8fa4a7520>)
```

Now the data have all necessary fields (‘rhoa’, ‘err’ and ‘k’) so we can run the inversion. The inversion mesh will be created with some optional values for the parametric mesh generation.

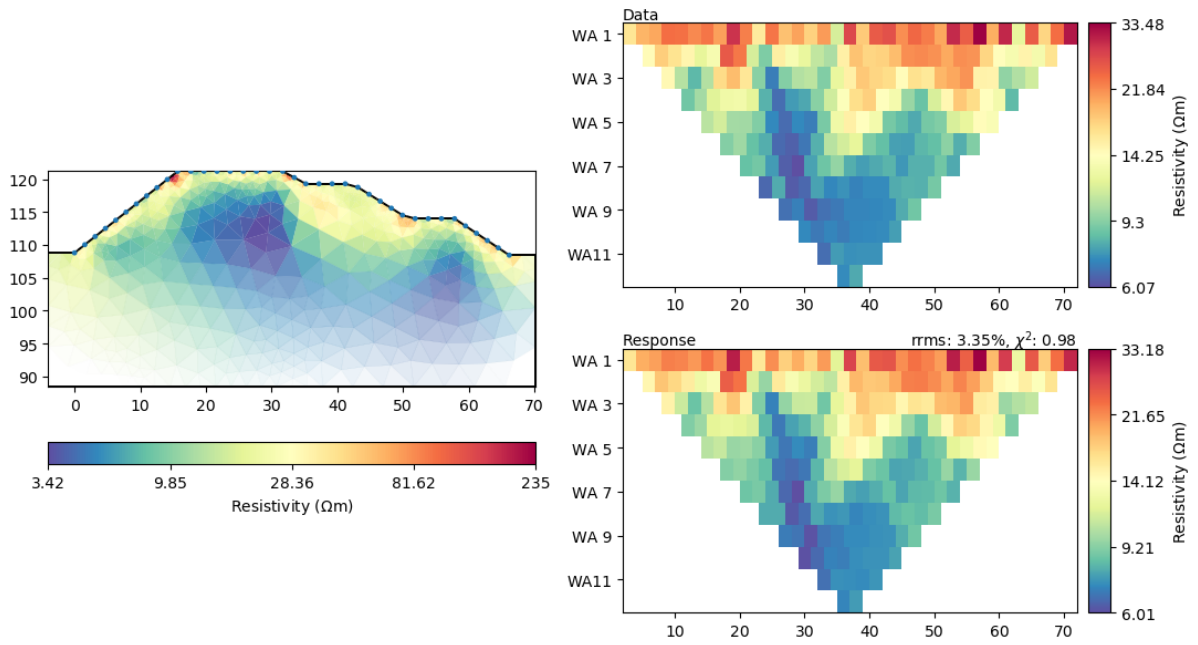
```
mod = mgr.invert(data, lam=10, verbose=True,
                 paraDX=0.3, paraMaxCellSize=10, paraDepth=20, quality=33.6)

mgr.showResult()
```




```
fop: <pygimli.physics.ert.ertModelling.ERTModelling object at 0x7fe8ab084680>
Data transformation: <pygimli.core._pygimli_.RTransLogLU object at␣
↪0x7fe8ab084950>
Model transformation: <pygimli.core._pygimli_.RTransLog object at 0x7fe8ab084630>
min/max (data): 6.07/33.48
min/max (error): 3.02%/4.11%
min/max (start model): 10.65/10.65
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
inv.iter 2 ... chi² = 1.58 (dPhi = 80.44%) lam: 10
-------------------------------------------------------------------------------
inv.iter 3 ... chi² = 1.01 (dPhi = 11.83%) lam: 10.0
-------------------------------------------------------------------------------
inv.iter 4 ... chi² = 0.98 (dPhi = 0.53%) lam: 10.0


#########################################################################
#                   Abort criterion reached: chi² <= 1 (0.98)           #
#########################################################################

(<matplotlib.axes._subplots.AxesSubplot object at 0x7fe8fa7d0af0>, <matplotlib.
↪colorbar.Colorbar object at 0x7fe8aafebe50>)
```

We can view the resulting model in the usual way.

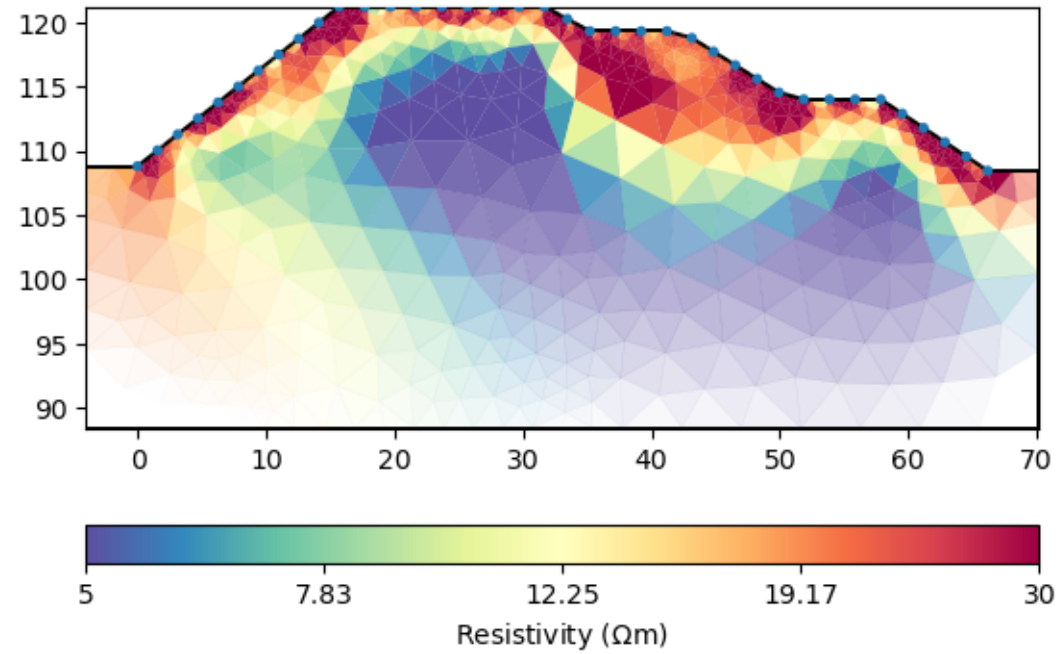
```
mgr.showResultAndFit()
# np.testing.assert_approx_equal(ert.inv.chi2(), 1.10883,␣
↪significant=3)
```




```
<Figure size 1100x600 with 6 Axes>
```

Or just plot the model only using your own options.

```
mgr.showResult(mod, cMin=5, cMax=30, cMap="Spectral_r", logScale=True)
```




```
(<matplotlib.axes._subplots.AxesSubplot object at 0x7fe8aabeed90>, <matplotlib.
↪colorbar.Colorbar object at 0x7fe8aaca9610>)
```

6.7.3.3 2D FEM modelling on two-layer example

Compare 2D FEM modelling with 1D VES sounding with and without complex resistivity values.

```
import numpy as np

import pygimli as pg
import pygimli.meshtools as mt
from pygimli.physics import ert
# from pygimli.physics.ert import simulate as simulateERT
from pygimli.physics.ert import VESModelling, VESCModelling
# from pygimli.physics.ert import createERTData
```

First we create a data configuration of a 1D Schlumberger sounding with 20 electrodes and and increasing MN/2 electrode spacing from 1m to 24m.

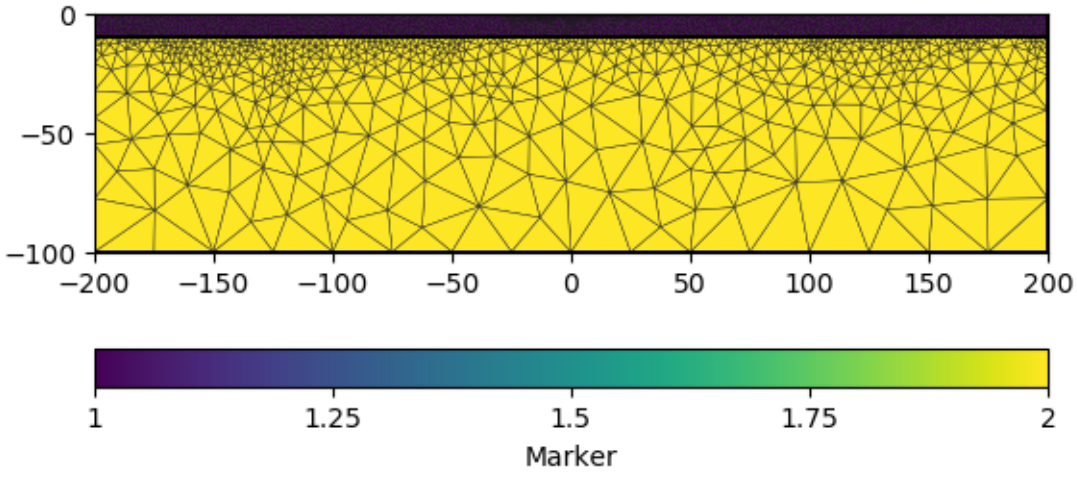
```
scheme = ert.createData(pg.utils.grange(start=1, end=24, dx=1, n=10, log=True),
                        sounding=True)
```

First we create a geometry that covers the sought geometry. We start with a 2 dimensional simulation world of a bounding box [-200, -100] [200, 0], the layer at -5m and some suitable requested cell sizes.

```
plc = mt.createWorld(start=[-200, -100], end=[200, 0],
                     layers=[-10], area=[5.0, 500])
```

To achieve a necessary numerical accuracy, we need some local mesh refinement in the vicinity of the electrodes. However, since we don't need the electrode (aka sensor) positions to be present as nodes in the geometry, we only add forced mesh nodes near the electrode positions, right below the earths surface.

```
for s in scheme.sensors():
    plc.createNode(s + [0.0, -0.2])

# Now we can create our forward modeling mesh.
mesh = mt.createMesh(plc, quality=33)

pg.show(mesh, data=mesh.cellMarkers(), label='Marker', showMesh=True)
```



```
(<matplotlib.axes._subplots.AxesSubplot object at 0x7fe8a930a310>, <matplotlib.
↪colorbar.Colorbar object at 0x7fe8aae97070>)
```

It is usually a good idea to calculate with a p2-refined mesh. However, you should be careful for larger meshes since the numerical efford will be highly increased.

```
mesh = mesh.createP2()
```

Perform the modeling using the static convenience call for ERT. Res is the resistivity mapping regarding the regions of the given geometry. Region with marker 1 is the upper layer, maker 2 is the background

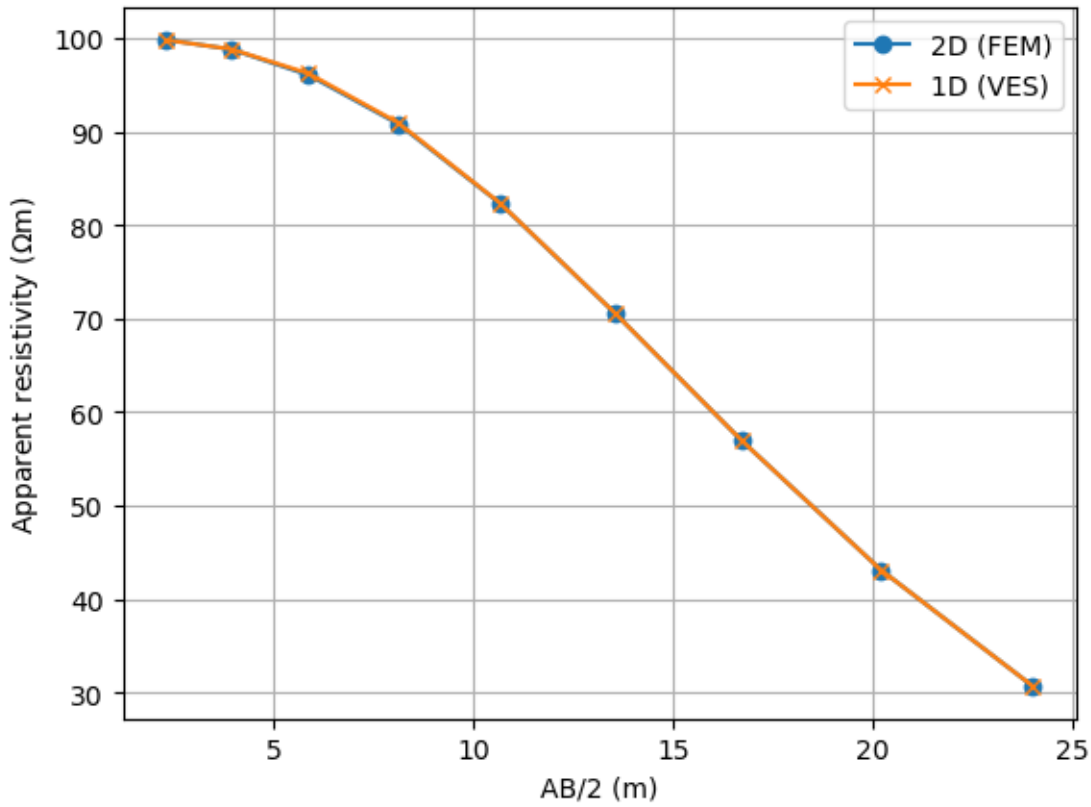
```
data = ert.simulate(mesh, res=[[1, 100.0], [2, 1.0]],
                    scheme=scheme, verbose=False)
```

1D VES

```
x = pg.x(scheme)
ab2 = (x[scheme('b')] - x[scheme('a')])/2
mn2 = (x[scheme('n')] - x[scheme('m')])/2
ves = VESModelling(ab2=ab2, mn2=mn2)
```

Plot results

```
fig, ax = pg.plt.subplots(1, 1)
ax.plot(ab2, data('rhoa'), '-o', label='2D (FEM)')
ax.plot(ab2, ves.response([10.0, 100.0, 1.0]), '-x', label='1D (VES)')
ax.set_xlabel('AB/2 (m)')
ax.set_ylabel('Apparent resistivity ($\Omega$m)')
ax.grid(1)
ax.legend()
```



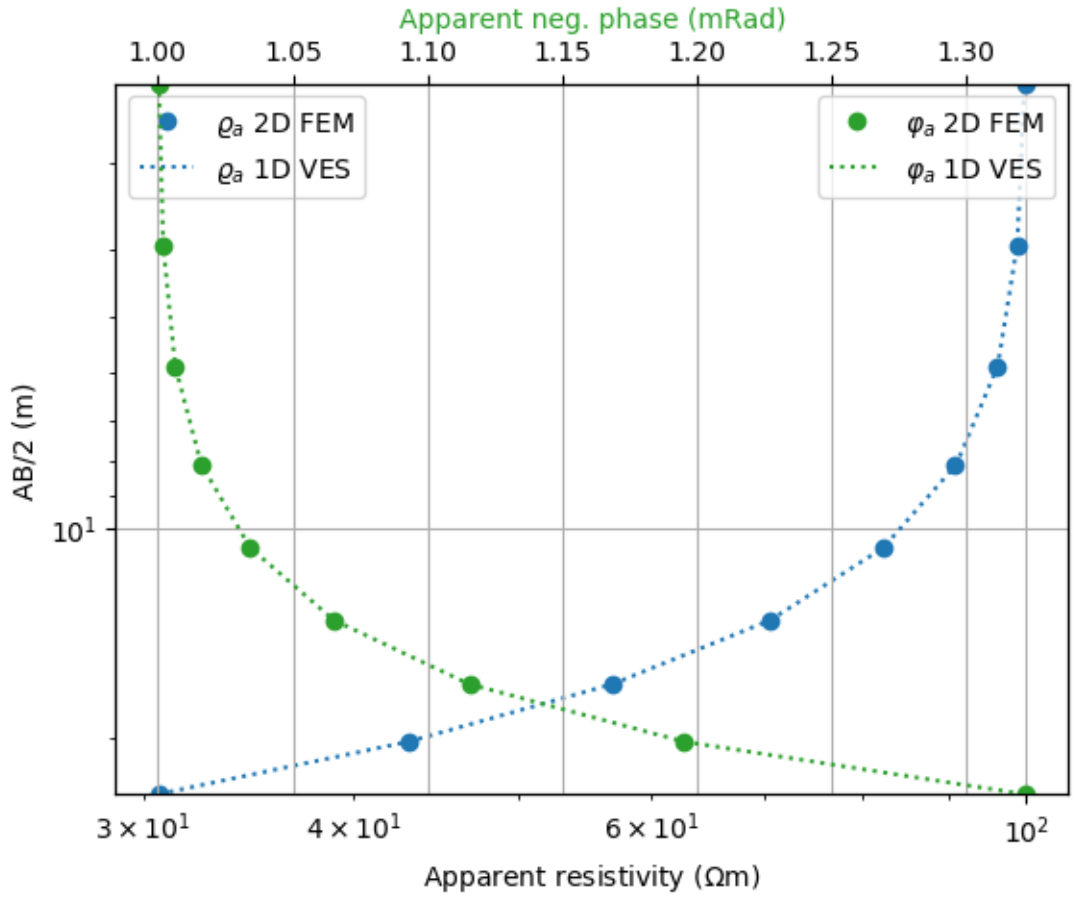

```
<matplotlib.legend.Legend object at 0x7fe8da9161f0>
```

We can easily repeat the above example using a complex resistivity model. defining amplitude and phase in negative mrad.

```
amps = np.array([100.0, 1.0])
phases = np.array([1.0, 10.0])
res = amps - 1j * amps * np.sin(phases/1000.)
data = ert.simulate(mesh, res=[[1, res[0]], [2, res[1]]],
                    scheme=scheme, verbose=False)

ves = VESCModelling(ab2=ab2, mn2=mn2)
rc = ves.response([10.0, 100.0, 1.0, phases[0]/1000, phases[1]/1000])
```

We can apply the default drawing routines for 1D VES data as well.

```
fig, ax = pg.plt.subplots(1, 1)
ves.drawData(ax, pg.cat(data('rhoa'), -data('phia')),
             labels=[r'$\varrho_a$ 2D FEM', r'$\varphi_a$ 2D FEM'],
             marker='o', linestyle='none')
ves.drawData(ax, rc,
              labels=[r'$\varrho_a$ 1D VES', r'$\varphi_a$ 1D VES'],
              marker=None)

np.testing.assert_approx_equal(data('rhoa')[0], 30.66351249, significant=5)
np.testing.assert_approx_equal(-data('phia')[0], 0.00132173865, significant=5)
```



6.7.3.4 Geoelectrics in 2.5D

This example shows geoelectrical (DC resistivity) forward modelling in 2.5 D, i.e. a 2D conductivity distribution and 3D point sources, to illustrate the modeling level. For ready-made ERT forward simulations in practice, please refer to the example on ERT modeling and inversion using the ERTManager.

Let us start with the gouverning partial differential equation of Poisson type

$$\nabla \cdot (\sigma \nabla u) = -I\delta(\vec{r} - \vec{r}_s) \in R^3$$

The source term (point electrode) is three-dimensional, but the distribution of the electrical conductivity $\sigma(x,y)$ should be 2D so we apply a Fourier cosine transform from $u(x,y,z) \mapsto u(x,k,z)$ with the wave number k. ($D^{(a)}(u(x,y,z)) \mapsto i^{|a|}k^a u(x,z)$)

$$\nabla \cdot (\sigma \nabla u) - \sigma k^2 u = -I\delta(\vec{r} - \vec{r}_s) \in R^2$$

$$\frac{\partial}{\partial x}\left(\sigma \frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial z}\left(\sigma \frac{\partial u}{\partial z}\right) - \sigma k^2 u = -I\delta(x - x_s)\delta(z - z_s) \in R^2$$

$$\frac{\partial u}{\partial \vec{n}} = 0 \quad \text{at the Surface} \quad (z = 0)$$

$$\frac{\partial u}{\partial \vec{n}} = au \quad \text{in the Subsurface} \quad (z < 0)$$

```
import matplotlib
import numpy as np
```

(continues on next page)

(continued from previous page)

```
import pygimli as pg

from pygimli.viewer.mpl import drawStreams
```

We know the exact solution by analytical formulas:

$$u = \frac{1}{2\pi\sigma} \cdot (K_0(\|r - r_s^+\|k) + K_0(\|r - r_s^-\|k))$$

with K0 being the Bessel function of first kind, and the normal and mirror sources r+ and r-. We define a function for it

```
def uAnalytical(p, sourcePos, k, sigma=1):
    """Calculates the analytical solution for the 2.5D geoelectrical␣
↪problem.

    Solves the 2.5D geoelectrical problem for one wave number k.
    It calculates the normalized (for injection current 1 A and␣
↪sigma=1 S/m)
    potential at position p for a current injection at position␣
↪sourcePos.
    Injection at the subsurface is recognized via mirror sources␣
↪along the
    surface at depth=0.

    Parameters
    ----------
    p : pg.Pos
        Position for the sought potential
    sourcePos : pg.Pos
        Current injection position.
    k : float
        Wave number

    Returns
    -------
    u : float
        Solution u(p)
    """
    r1A = (p - sourcePos).abs()
    # Mirror on surface at depth=0
    r2A = (p - pg.Pos([1.0, -1.0])*sourcePos).abs()

    if r1A > 1e-12 and r2A > 1e-12:
        return  1 / (2.0 * np.pi) * 1/sigma * \
            (pg.math.besselK0(r1A * k) + pg.math.besselK0(r2A * k))
    else:
        return 0.
```

We assume the so-called mixed boundary conditions (Dey & Morrison, 1979).

$$\sigma k \frac{\mathbf{r} \cdot \mathbf{n}}{|r|} \frac{K_1(|r - r_s|k)}{K_0(|r - r_s|k)}$$

```
def mixedBC(boundary, userData):
    """Mixed boundary conditions.

    Define the derivative of the analytical solution regarding the␣
↪outer normal
    direction :math:`\vec{n}`. So we can define the values for mixed␣
↪boundary
    condition :math:`\frac{\partial u}{\partial \vec{n}} = -au`
    for the boundaries on the subsurface.

    """
    ### ignore surface boundaries for wildcard boundary condition
    if boundary.norm()[1] == 1.0:
        return 0

    sourcePos = userData['sourcePos']
    k = userData['k']
    sigma = userData['s']
    r1 = boundary.center() - sourcePos

    # Mirror on surface at depth=0
    r2 = boundary.center() - pg.Pos(1.0, -1.0) * sourcePos
    r1A = r1.abs()
    r2A = r2.abs()

    n = boundary.norm()
    if r1A > 1e-12 and r2A > 1e-12:
        alpha = sigma * k * ((r1.dot(n)) / r1A * pg.math.besselK1(r1A * k) +
                             (r2.dot(n)) / r2A * pg.math.besselK1(r2A * k)) / \
            (pg.math.besselK0(r1A * k) + pg.math.besselK0(r2A * k))

        return alpha

        # Note, the above is the same like:
        beta = 1.0
        return [alpha, beta, 0.0]

    else:
        return 0.0
```

We assemble the right-hand side (rhs) for the singular current term by hand since this cannot be done efficiently by *pg.solve* yet. We basically search for the cell containing the source and project the point using its shape functions *N*.

```
def rhsPointSource(mesh, source):
    """Define function for the current source term.

    :math:`\delta(x-pos), \int f(x) \delta(x-pos)=f(pos)=N(pos)`
    Right hand side entries will be shape functions(pos)
    """
```

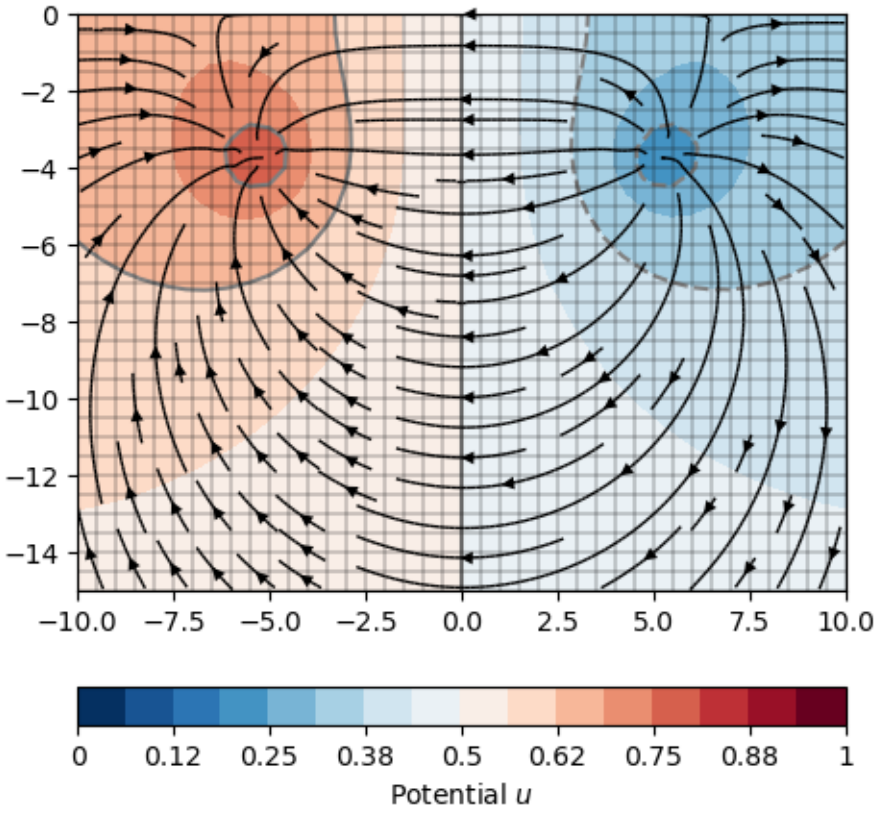
(continues on next page)

(continued from previous page)

```
    rhs = pg.Vector(mesh.nodeCount())

    cell = mesh.findCell(source)
    rhs.setVal(cell.N(cell.shape().rst(source)), cell.ids())
    return rhs
```

Now we create a suitable mesh and solve the equation with *pg.solve*. Note that we use a mesh with quadratic shape functions by calling *createP2*.

```
mesh = pg.createGrid(x=np.linspace(-10.0, 10.0, 41),
                     y=np.linspace(-15.0,  0.0, 31))
mesh = mesh.createP2()

sourcePosA = [-5.25, -3.75]
sourcePosB = [+5.25, -3.75]

k = 1e-2
sigma = 1.0
bc={'Robin': {'1,2,3': mixedBC}}
u = pg.solve(mesh, a=sigma, b=-sigma * k*k,
             rhs=rhsPointSource(mesh, sourcePosA),
             bc=bc, userData={'sourcePos': sourcePosA, 'k': k, 's':sigma},
             verbose=True)

u -= pg.solve(mesh, a=sigma, b=-sigma * k*k,
              rhs=rhsPointSource(mesh, sourcePosB),
              bc=bc, userData={'sourcePos': sourcePosB, 'k': k, 's':sigma},
              verbose=True)

# The solution is shown by calling

ax = pg.show(mesh, data=u, cMap="RdBu_r", cMin=-1, cMax=1,
             orientation='horizontal', label='Potential $u$',
             nCols=16, nLevs=9, showMesh=True)[0]

# Additionally to the image of the potential we want to see the␣
↪current flow.
# The current flows along the gradient of our solution and can be␣
↪plotted as
# stream lines. By default, the drawStreams method draws one␣
↪segment of a
# stream line per cell of the mesh. This can be a little confusing␣
↪for dense
# meshes so we can give a second (coarse) mesh as a new cell base␣
↪to draw the
# streams. If `drawStreams` gets scalar data, the gradients will be␣
↪calculated.
gridCoarse = pg.createGrid(x=np.linspace(-10.0, 10.0, 20),
                           y=np.linspace(-15.0,   .0, 20))
drawStreams(ax, mesh, u, coarseMesh=gridCoarse, color='Black')
```




```
Mesh:  Mesh: Nodes: 3741 Cells: 1200 Boundaries: 2470
Assembling time:  0.022
Solving time:  0.025
Mesh:  Mesh: Nodes: 3741 Cells: 1200 Boundaries: 2470
Assembling time:  0.02
Solving time:  0.039
```

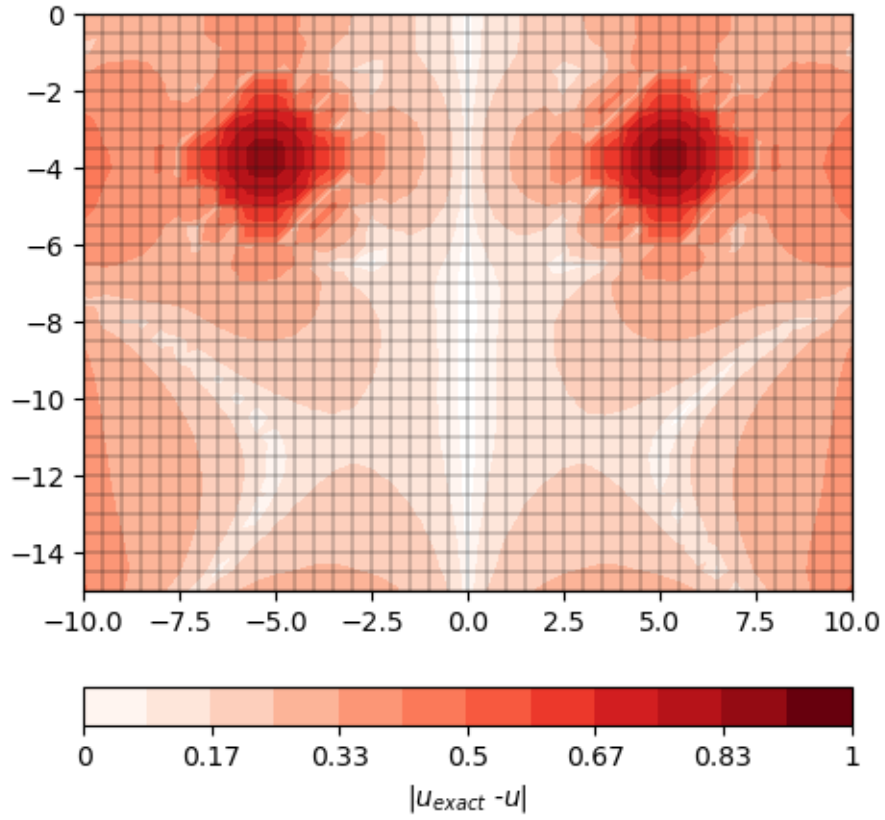
We know the exact solution so we can compare it to the numerical results. Unfortunately, the point source singularity does not allow a good integration measure for the accuracy of the resulting field so we just look for the differences.

```
uAna = pg.Vector(list(map(lambda p__: uAnalytical(p__, sourcePosA, k, sigma),
                          mesh.positions())))
uAna -= pg.Vector(list(map(lambda p__: uAnalytical(p__, sourcePosB, k, sigma),
                           mesh.positions())))

ax = pg.show(mesh, data=pg.abs(uAna-u), cMap="Reds",
             orientation='horizontal', label='|$u_{exact}$ -$u$|',
             logScale=True, cMin=1e-7, cMax=1e-1,
             contourLines=False,
             nCols=12, nLevs=7,
             showMesh=True)[0]

#print('l2:', pg.pf(pg.solver.normL2(uAna-u)))
print('L2:', pg.pf(pg.solver.normL2(uAna-u, mesh)))
print('H1:', pg.pf(pg.solver.normH1(uAna-u, mesh)))
```

(continues on next page)

(continued from previous page)

```
np.testing.assert_approx_equal(pg.solver.normL2(uAna-u, mesh),
                               0.02415, significant=3)
```




```
L2: 0.02
H1: 0.19
```

6.7.3.5 Four-point sensitivities

In this example, we illustrate how to visualize the sensitivities of four-point arrays. You can easily loop over the plotting command to create something like: https://www.youtube.com/watch?v=lt1qV-2d5Ps

```
import numpy as np
import matplotlib.pyplot as plt
import pygimli as pg
import pygimli.meshtools as mt
from pygimli.physics import ert
```

We start by creating a ERT data container with three four-point arrays.

```
scheme = pg.DataContainerERT()

nelecs = 10
pos = np.zeros((nelecs, 2))
pos[:, 0] = np.linspace(5, 25, nelecs)
```

(continues on next page)

(continued from previous page)

```
scheme.setSensorPositions(pos)

measurements = np.array((
    [0, 3, 6, 9],  # Dipole-Dipole
    [0, 9, 3, 6],  # Wenner
    [0, 9, 4, 5]   # Schlumberger
))

for i, elec in enumerate("abmn"):
    scheme[elec] = measurements[:,i]

scheme["k"] = ert.createGeometricFactors(scheme)
```

Now we set up a 2D mesh.

```
world = mt.createWorld(start=[0, 0], end=[30, -10], worldMarker=True)
for pos in scheme.sensorPositions():
    world.createNode(pos)

mesh = mt.createMesh(world, area=.05, quality=33, marker=1)
```

As a last step we invoke the ERT manager and calculate the Jacobian for a homogeneous half-space.

```
fop = ert.ERTModelling()
fop.setData(scheme)
fop.setMesh(mesh)

model = np.ones(mesh.cellCount())
fop.createJacobian(model)
```

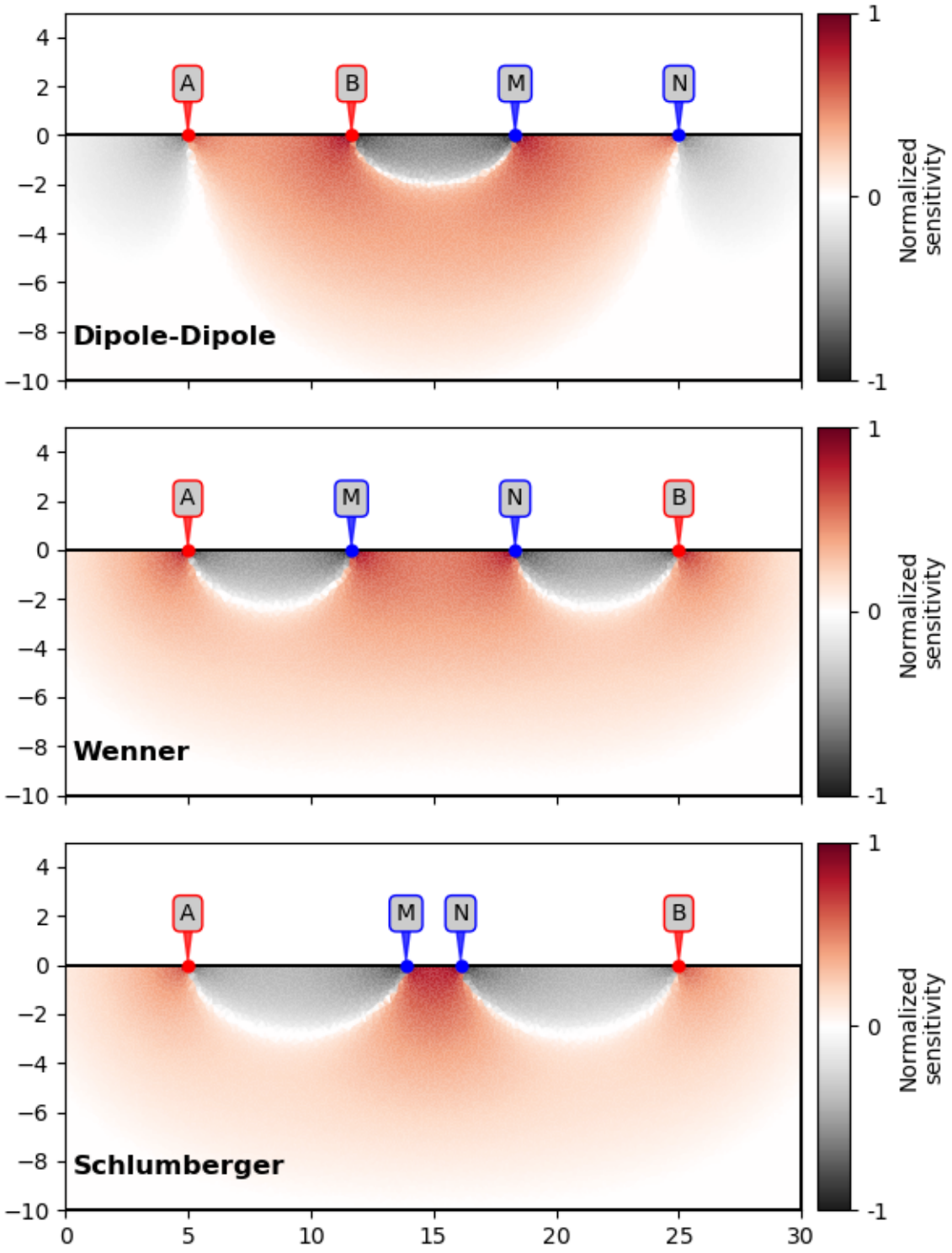
Final visualization

```
def getABMN(scheme, idx):
    """ Get coordinates of four-point cfg with id `idx` from␣
↪DataContainerERT
    `scheme`."""
    coords = {}
    for elec in "abmn":
        elec_id = int(scheme(elec)[idx])
        elec_pos = scheme.sensorPosition(elec_id)
        coords[elec] = elec_pos.x(), elec_pos.y()
    return coords


def plotABMN(ax, scheme, idx):
    """ Visualize four-point configuration on given axes. """
    coords = getABMN(scheme, idx)
    for elec in coords:
        x, y = coords[elec]
        if elec in "ab":
```

(continues on next page)

(continued from previous page)

```
            color = "red"
        else:
            color = "blue"
        ax.plot(x, y, marker=".", color=color, ms=10)
        ax.annotate(elec.upper(), xy=(x, y), ha="center", fontsize=10, bbox=dict(
            boxstyle="round", fc=(0.8, 0.8, 0.8), ec=color), xytext=(0, 20),
                    textcoords='offset points', arrowprops=dict(
                        arrowstyle="wedge, tail_width=.5", fc=color, ec=color,
                        patchA=None, alpha=0.75))
        ax.plot(coords["a"][0],)

labels = ["Dipole-Dipole", "Wenner", "Schlumberger"]
fig, ax = plt.subplots(scheme.size(), 1, sharex=True, figsize=(6,8))

for i, sens in enumerate(fop.jacobian()):
    # Label in lower-left corner
    ax[i].text(.01, .15, labels[i], horizontalalignment='left',
               verticalalignment='top', transform=ax[i].transAxes, fontsize=12,
               fontweight="bold")

    # Electrode annotations
    plotABMN(ax[i], scheme, i)

    # Log-scaled and normalized sensitivity
    normsens = pg.utils.logDropTol(sens/mesh.cellSizes(), 8e-4)
    normsens /= np.max(normsens)
    pg.show(mesh, normsens, cMap="RdGy_r", ax=ax[i], orientation="vertical",
            label="Normalized\nsensitivity", nLevs=3, cMin=-1, cMax=1)

fig.tight_layout()
```


A
B
M
N
Dipole-Dipole
Wenner
Schlumberger
Normalized sensitivity
1
0
-1
4
2
0
−2
−4
−6
−8
−10
0
5
10
15
20
25
30

6.7.3.6 3D modeling in a closed geometry

This is a synthetic model of an experimental tank with a highly heterogeneous resistivity, motivated by the BAM Berlin.

Geometry: 0.99m x 0.5m x 1.0m Data: 48 Electrodes and 588 Measurements defined in modeltank.shm Each 24 electrodes are located at two opposite sides of the tank.

We use the pygimli meshtools to create a PLC of the tank and an inhomogeneity. The needed mesh is created by calling tetgen.

```
import numpy as np

import pygimli as pg
import pygimli.meshtools as mt
from pygimli.physics import ert
```

In contrast to field measurements, experimental tanks have well-defined spatial dimensions and need different boundary conditions (BC).

As there is no current flow through the tanks boundary at all, homogeneous (Neumann) BC are defined for the whole boundary. Neumann BC are natural (intrinsic) for the finite element simulations. link{tutorial:fem:bc}, so we just need to define a cube geometry including region markers.

```
plc = mt.createCube(size=[0.99, 0.5, 1.0], pos=[0.495, 0.25], boundaryMarker=1)
```

We first read the measuring scheme file and add the electrodes as nodes with the marker -99 to the geometry.

```
filename = pg.getExampleFile("ert/modeltank.shm")
shm = pg.DataContainerERT(filename)

for s in shm.sensors():
    plc.createNode(s, marker=-99)
```

There are two small problems to overcome for simulating Neumann bodies.

First, we always need dipole current injection since there can be no current flow out of the closed boundaries of our experimental tank. (Note that by default single poles are simulated and superpositioned.) Therefore we define a reference electrode position inside the PLC, with a marker -999, somewhere away from the electrodes.

```
plc.createNode([0.5, 0.5, -0.5], marker=-999)
```

```
ID: 56, Marker: -999    RVector3: (0.5, 0.5, -0.5)
```

The second problem for pure Neumann domains is the non-uniqueness of the partial differential equation (there are only partial derivatives of the electric potential so an arbitrary value might be added, i.e. calibrated).

Therefore we add calibration node with marker -1000 where the potential is fixed , somewhere on the boundary and far from the electrodes.

```
plc.createNode([0.75, 0.25, 0.5], marker=-1000)
```

```
ID: 57, Marker: -1000   RVector3: (0.75, 0.25, 0.5)
```

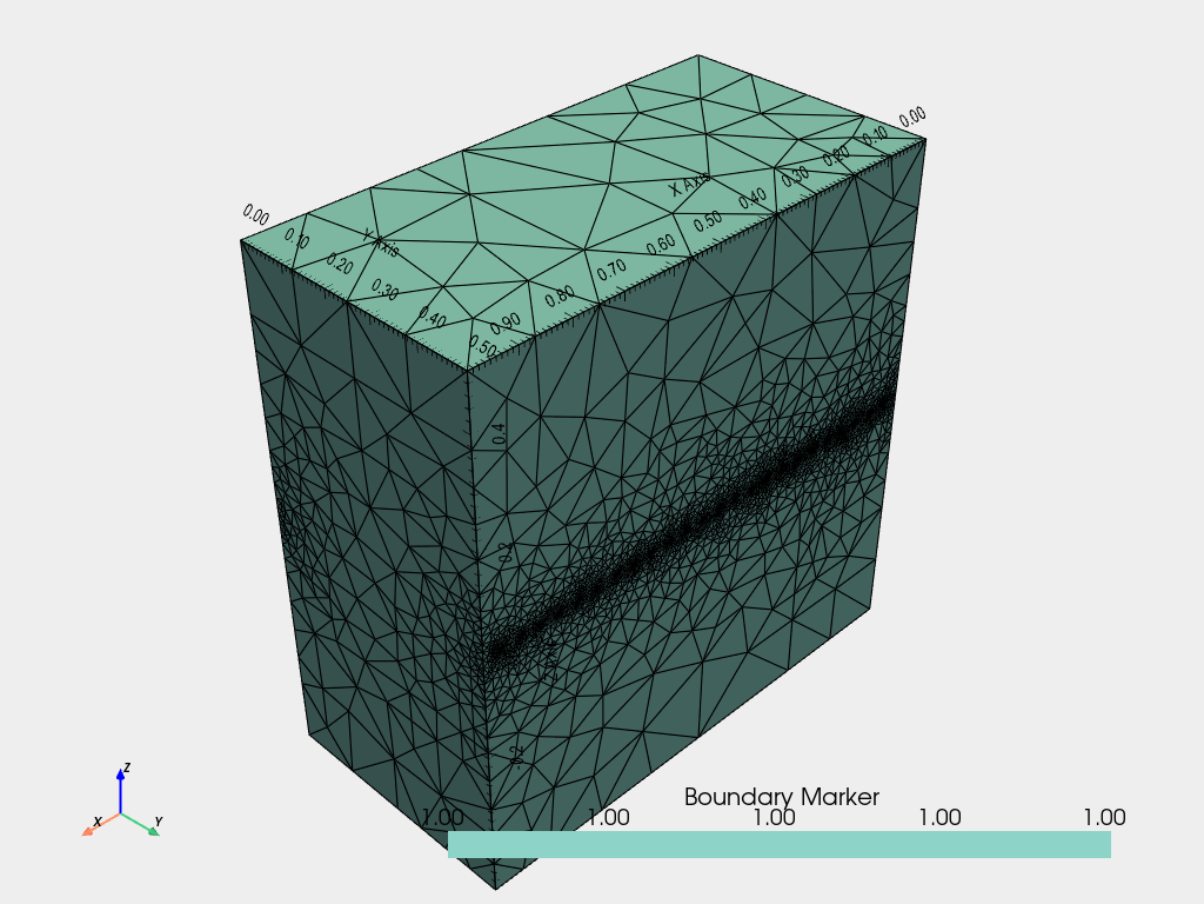
For sufficient numerical accuracy it is generally a good idea to refine the mesh in the vicinity of the electrodes positions. We force the local mesh refinement by an additional node at 1 mm distance in -z-direction.

```
for s in plc.positions(pg.find(plc.nodeMarkers() == -99)):
    plc.createNode(s - [0.0, 0.0, 1e-3])

# Also refine the reference node
plc.createNode([0.5, 0.5, -0.5 - 1e-3])
#pg.show(plc, markers=True, showMesh=True)
```

```
ID: 106, Marker: 0      RVector3: (0.5, 0.5, -0.501)
```

Create the tetrahedron mesh (calling the tetgen mesh generator)

```
mesh = mt.createMesh(plc)
pg.show(mesh, markers=True, showMesh=True)
```




```
(<pyvista.plotting.plotting.Plotter object at 0x7fe8f562d640>, None)
```

First we want to simulate our ERT response for a homogeneous resistivity of 1 :math:`Omega`m. Usually, simulate will calculate apparent resistivities (rhoa) and put them into the returned DataContainerERT. However, for the calculation of rhoa, geometric factors (k) are expected in the data container. If

simulate does not find any k in the scheme file, it tries to determine them itself, either analytically (halfspace) or numerically (topography). Note that the automatic numerical calculating can only be a fallback mode. You usually want to keep full control about this calculation because you want the accuracy as high as possible by providing a special mesh with higher quality, lower max cell area, finer local mesh refinement, or quadratic base functions (p2).

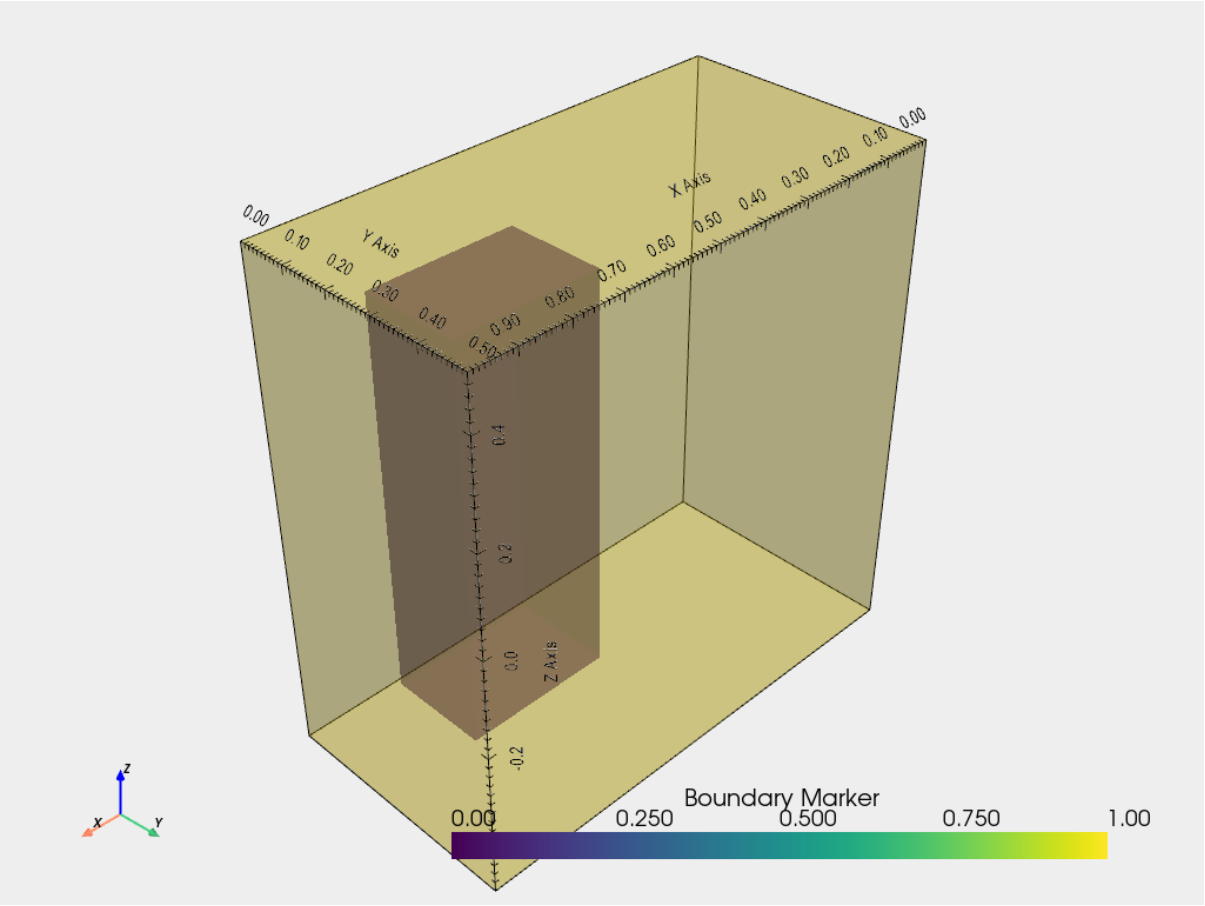
We don't want the automatic k generation here because we want also to demonstrate how you can solve this task yourself. The argument 'calcOnly=True' omits the check for valid k factors and will add the simulated voltages (u) in the returned DataContainerERT. The simulation current (i) is 1 A by default. In addition, the flag 'sr=False' omits the singularity removal technique (default) which not applicable in absence of (analytic) primary potential.

```
hom = ert.simulate(mesh, res=1.0, scheme=shm, sr=False,
                   calcOnly=True, verbose=True)

hom.save('homogeneous.ohm', 'a b m n u')
```

```
1
```

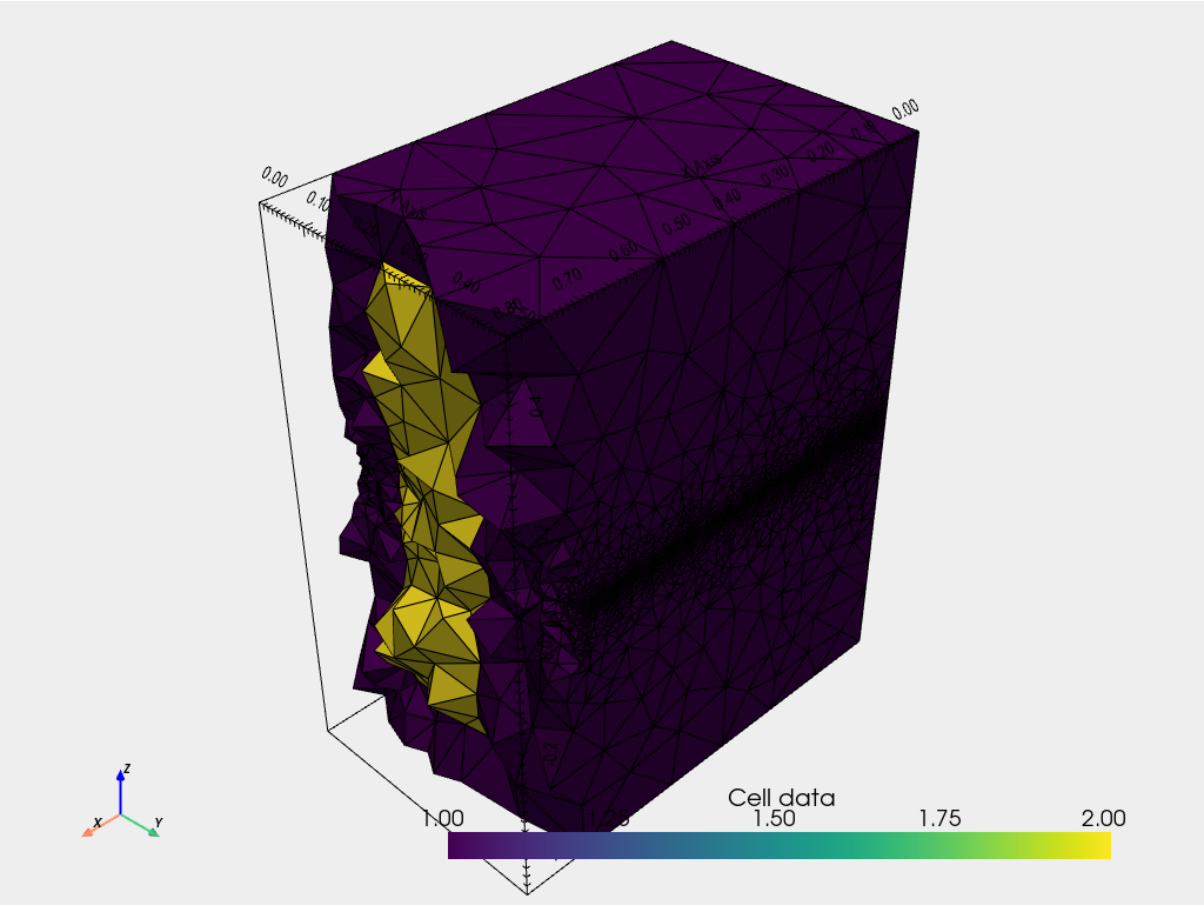
We now create an inhomogeneity (cube) and merge it with the above PLC. marker=2 ensures all cells of the cube being associated with a cell marker 2.

```
cube = mt.createCube(size=[0.3, 0.2, 0.8], pos=[0.7, 0.2], marker=2)
plc += cube

pg.show(plc, alpha=0.3)
mesh = mt.createMesh(plc)
```



Also its advisable to control the geometry in a 3D viewer. We recommend Paraview https://www.paraview.org/ and will export the geometry and in the vtk file format.

```
# plc.exportVTK('plc')
# mesh.exportVTK('mesh')
pg.show(mesh, mesh.cellMarkers(), showMesh=True,
        filter={'clip':{'origin':(0.7, 0, 0.0)},})
```




```
(<pyvista.plotting.plotting.Plotter object at 0x7fe8da9afee0>, None)
```

Now that we have a mesh with different regions and cell markers, we can define a relationship between region marker and resistivity by defining a appropriate list. We also set sr=False here because singularity removal would need highly accurate primary potentials which can also only be calculated numerically.

```
res = [[1, 10.0], [2, 100.0]]  # map markers 1 and 2 to 10 and 100 Ohmm,␣
↪resp.
het = ert.simulate(mesh, res=res, scheme=shm, sr=False,
                   calcOnly=True, verbose=True)
```

The apparent resistivity for a homogeneous model of 1 Ohmm should be 1 Ohmm. Therefore we can take the inverse of the modeled resistances for the homogeneous model and use it as geometric factors to find the apparent resistivities for the inhomogeneous model.

```
het.set('k', 1.0/ (hom('u') / hom('i')))
het.set('rhoa', het('k') * het('u') / het('i'))

het.save('simulated.dat', 'a b m n rhoa k u i')

np.testing.assert_approx_equal(het('rhoa')[0], 9.5, 1)

# np.testing.assert_approx_equal(het('k')[0], 0.820615269548)
```

For such kind of simulations, the homogeneous part should be high accurate because it is usually needed once after storing the geometric factors. Note, do not forget to add some noise if you plan to invert such

simulated data.

see also:

TODO * inversion example

checks: TODO: * any idea for a Figure here? * maybe show data and effect of topography * Alternatively, we can create a real cavity by changing the marker in isHole flag for createCube (check)

Total running time of the script: (0 minutes 35.114 seconds)

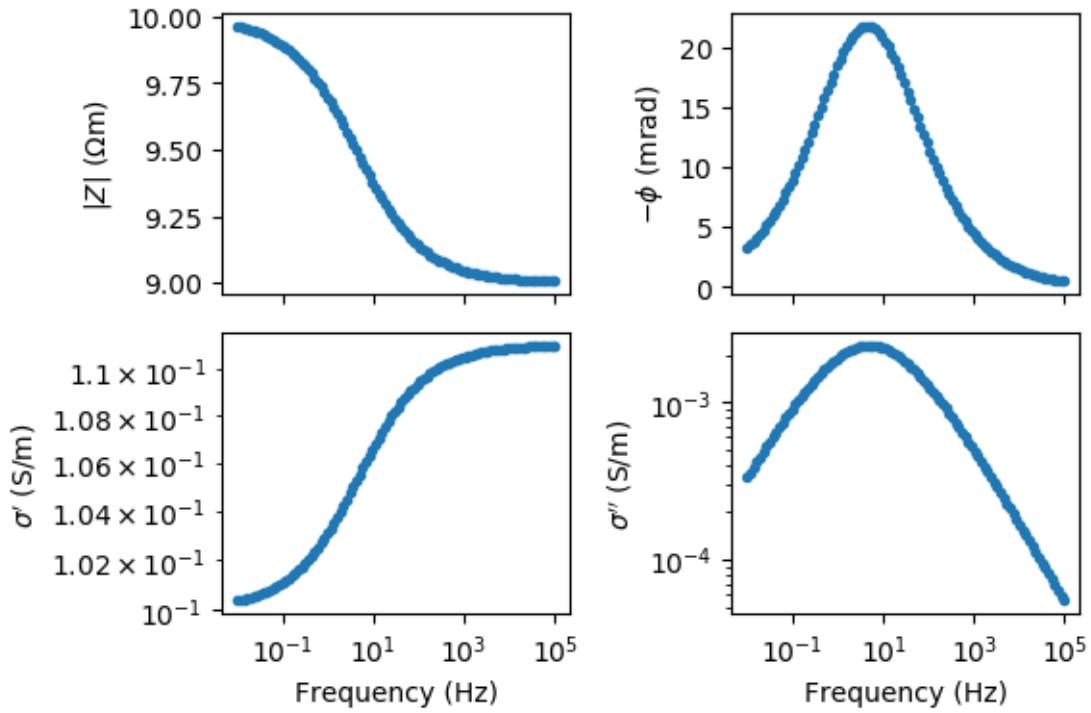
6.7.4 Induced polarization

6.7.4.1 Generating SIP signatures

This example highlights some of the capabilities of pyGimli to generate spectral induced polarization (SIP) signatures.

Generate a Cole-Cole signature

```
from pygimli.physics.SIP import modelColeColeRho
import numpy as np
import pygimli as pg
import matplotlib as mpl

f = np.logspace(-2, 5, 100)

Z = modelColeColeRho(f, rho=10, m=0.1, tau=0.04, c=0.5)

fig, axes = pg.plt.subplots(2, 2, figsize=(15 / 2.54, 10 / 2.54), sharex=True)

ax = axes[0, 0]
ax.semilogx(f, np.abs(Z), '.-')
ax.set_ylabel(r'$|Z|$ ($\Omega$m)')
ax = axes[0, 1]
ax.semilogx(f, -np.angle(Z) * 1e3, '.-')
ax.set_ylabel(r'$-\phi$ (mrad)')
ax = axes[1, 0]
Y = 1 / Z
ax.loglog(f, np.real(Y), '.-')
ax.set_xlabel('Frequency (Hz)')
ax.set_ylabel(r"$\sigma'$ (S$/$m)")
ax = axes[1, 1]
ax.loglog(f, np.imag(Y), '.-')
ax.set_xlabel('Frequency (Hz)')
ax.set_ylabel(r"$\sigma''$ (S$/$m)")

for ax in axes.flat:
    ax.xaxis.set_major_locator(mpl.ticker.LogLocator(numticks=5))
fig.tight_layout()
```



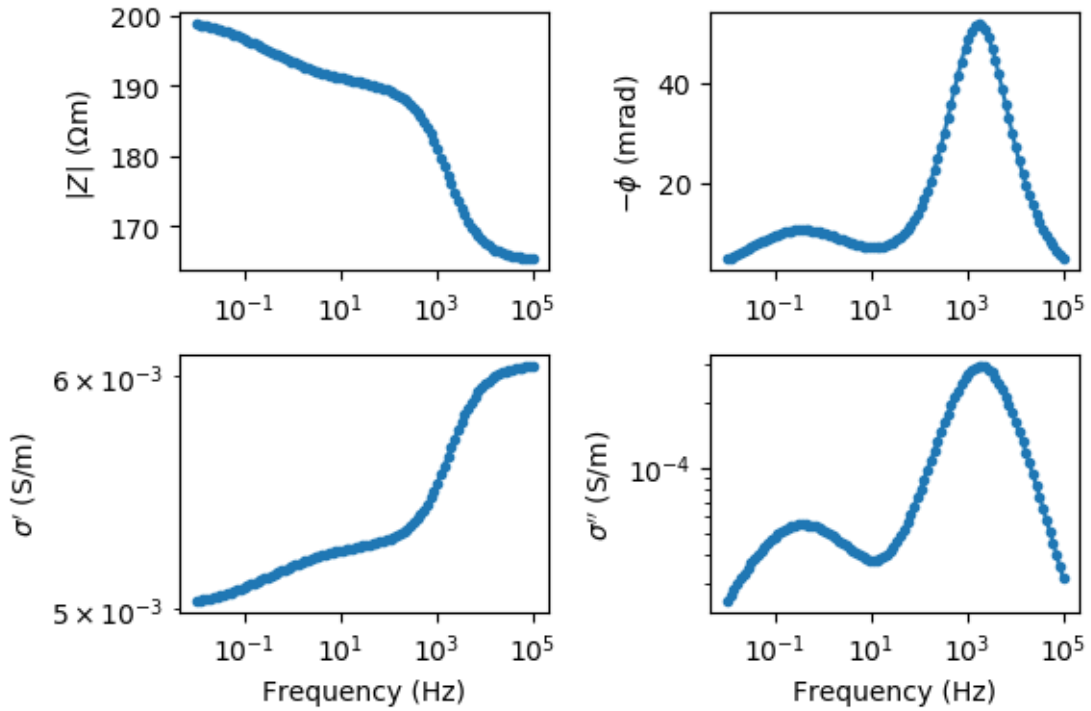
Generate a double Cole-Cole signature

```
from pygimli.physics.SIP import modelColeColeRho
import numpy as np
import pygimli as pg

f = np.logspace(-2, 5, 100)

# term1
Z1 = modelColeColeRho(f, rho=1, m=0.1, tau=0.5, c=0.5)
# term2
Z2 = modelColeColeRho(f, rho=1, m=0.25, tau=0.0001, c=0.8)
# create sum
rho0 = 100
Z = rho0 * (Z1 + Z2)

fig, axes = pg.plt.subplots(2, 2, figsize=(15 / 2.54, 10 / 2.54))
ax = axes[0, 0]
ax.semilogx(f, np.abs(Z), '.-')
ax.set_ylabel(r'$|Z|$ ($\Omega$m)')
ax = axes[0, 1]
ax.semilogx(f, -np.angle(Z) * 1e3, '.-')
ax.set_ylabel(r'$-\phi$ (mrad)')
ax = axes[1, 0]
Y = 1 / Z
ax.loglog(f, np.real(Y), '.-')
ax.set_xlabel('Frequency (Hz)')
ax.set_ylabel(r"$\sigma'$ (S$/$m)")
ax = axes[1, 1]
ax.loglog(f, np.imag(Y), '.-')
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel('Frequency (Hz)')
ax.set_ylabel(r"$\sigma''$ (S$/$m)")

for ax in axes.flat:
    ax.xaxis.set_major_locator(
        mpl.ticker.LogLocator(numticks=5)
    )
fig.tight_layout()
```



6.7.4.2 Fitting SIP signatures

This example highlights some of the capabilities of pyGimli to analyze spectral induced polarization (SIP) signatures.

Author: *Maximilian Weigand, University of Bonn*

Import pyGIMLi and related stuff for SIP Spectra

```
from pygimli.physics.SIP import SIPSpectrum, modelColeColeRho
import numpy as np
import pygimli as pg
```

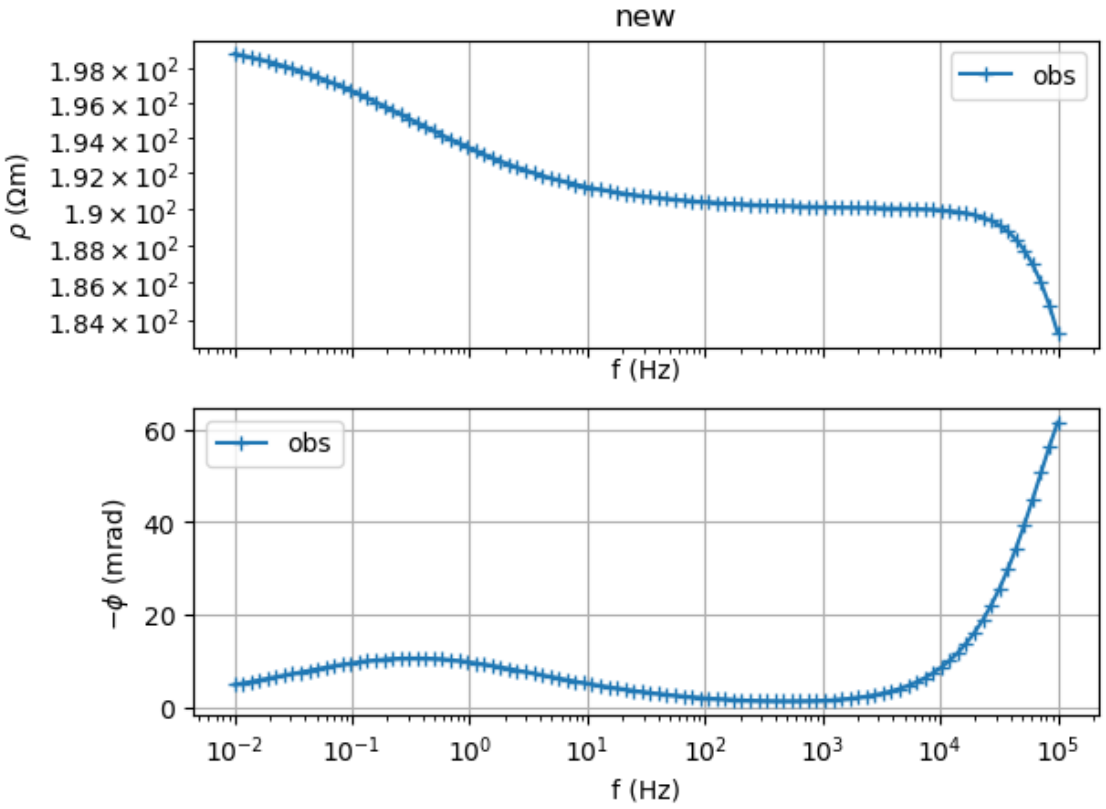
1. Generate synthetic data with a Double-Cole-Cole Model and initialize a SIPSpectrum object

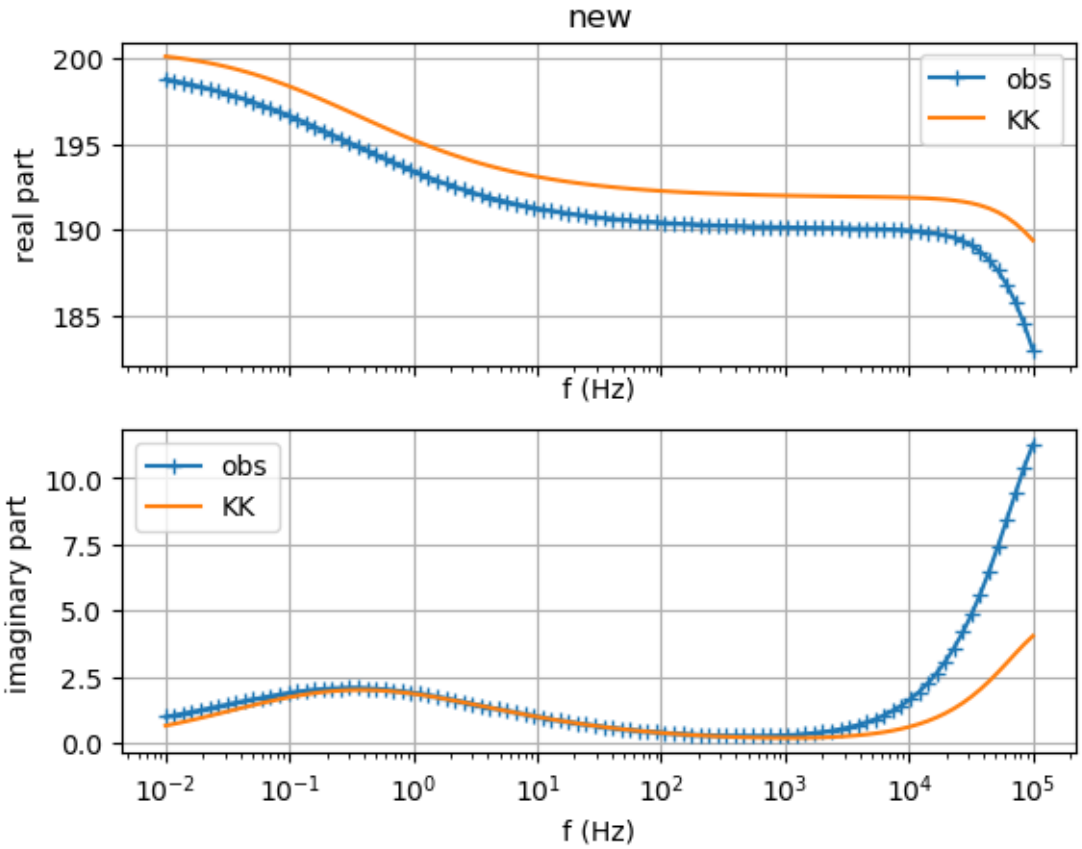
```
f = np.logspace(-2, 5, 100)
Z1 = modelColeColeRho(f, rho=1, m=0.1, tau=0.5, c=0.5)
Z2 = modelColeColeRho(f, rho=1, m=0.25, tau=1e-6, c=1.0)

rho0 = 100 # (Ohm m)
```

(continues on next page)

(continued from previous page)

```
Z = rho0 * (Z1 + Z2)

sip = SIPSpectrum(f=f, amp=np.abs(Z), phi=-np.angle(Z))
# Note the minus sign for the phases: we need to provide -phase[rad]

sip.showData()
sip.showDataKK()  # check Kramers-Kronig relations
```

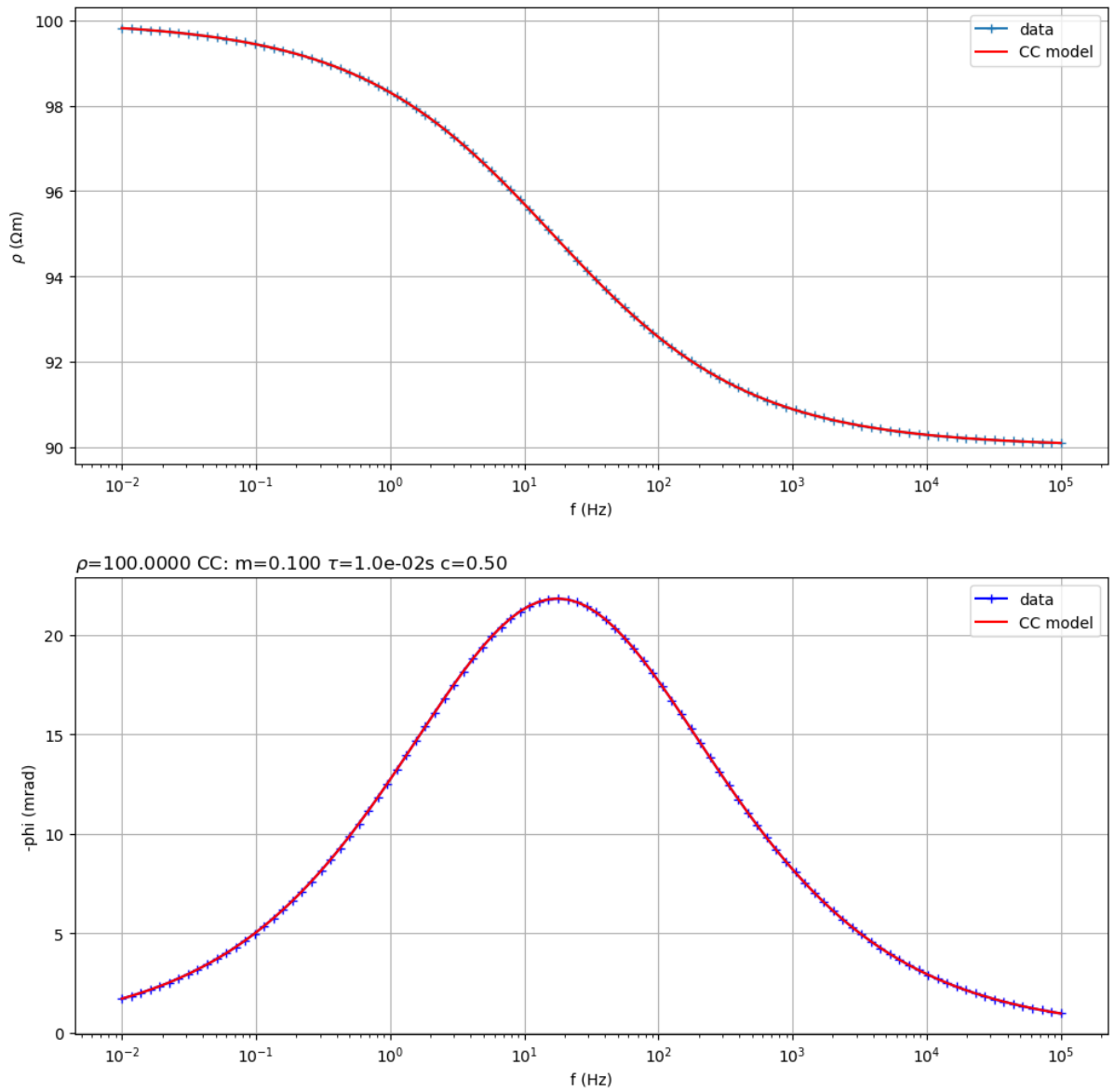





```
(<Figure size 640x480 with 2 Axes>, array([<matplotlib.axes._subplots.
↪AxesSubplot object at 0x7fe8a930e400>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x7fe8f56cfa90>],
      dtype=object))
```

2. Fit a Cole-Cole model from synthetic data

```
Z = modelColeColeRho(f, rho=100, m=0.1, tau=0.01, c=0.5)
# TODO data need some noise

sip = SIPSpectrum(f=f, amp=np.abs(Z), phi=-np.angle(Z))
sip.fitColeCole(useCond=False, verbose=False)  # works for both rho and␣
↪sigma models
sip.showAll()
```




```
(<Figure size 1200x1200 with 2 Axes>, array([<matplotlib.axes._subplots.
↪AxesSubplot object at 0x7fe8ab0067f0>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x7fe8aaced220>],
      dtype=object))
```

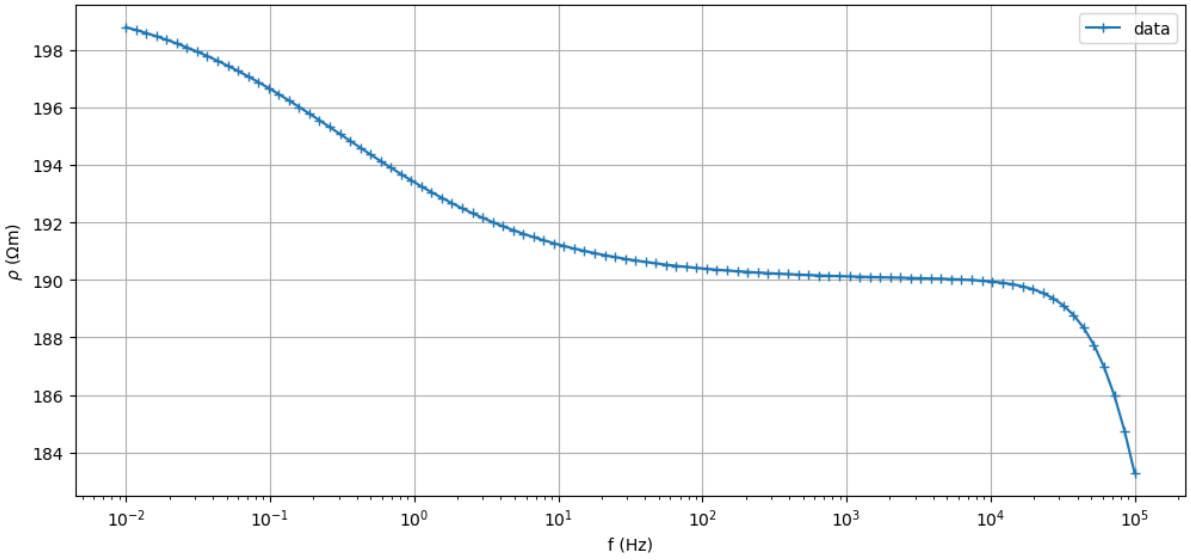
3. Fit a double Cole-Cole model

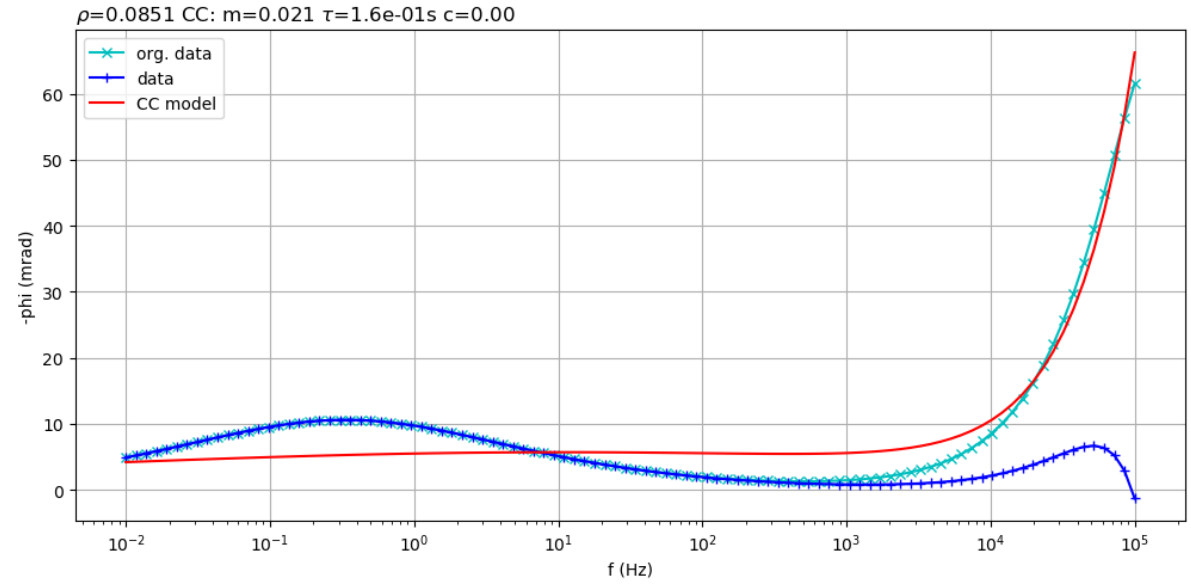
```
f = np.logspace(-2, 5, 100)
Z1 = modelColeColeRho(f, rho=1, m=0.1, tau=0.5, c=0.5)
Z2 = modelColeColeRho(f, rho=1, m=0.25, tau=1e-6, c=1.0)

rho0 = 100 #(Ohm m)
Z = rho0 * (Z1 + Z2)

# TODO data need some noise
sip = SIPSpectrum(f=f, amp=np.abs(Z), phi=-np.angle(Z))
sip.fitCCEM(verbose=False) # fit an SIP Cole-Cole term and an EM term␣
```

(continues on next page)

(continued from previous page)

```
↪(also Cole-Cole)
sip.showAll()
```






```
(<Figure size 1200x1200 with 2 Axes>, array([<matplotlib.axes._subplots.
↪AxesSubplot object at 0x7fe8fa6e2790>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x7fe8fa6dfbb0>],
      dtype=object))
```

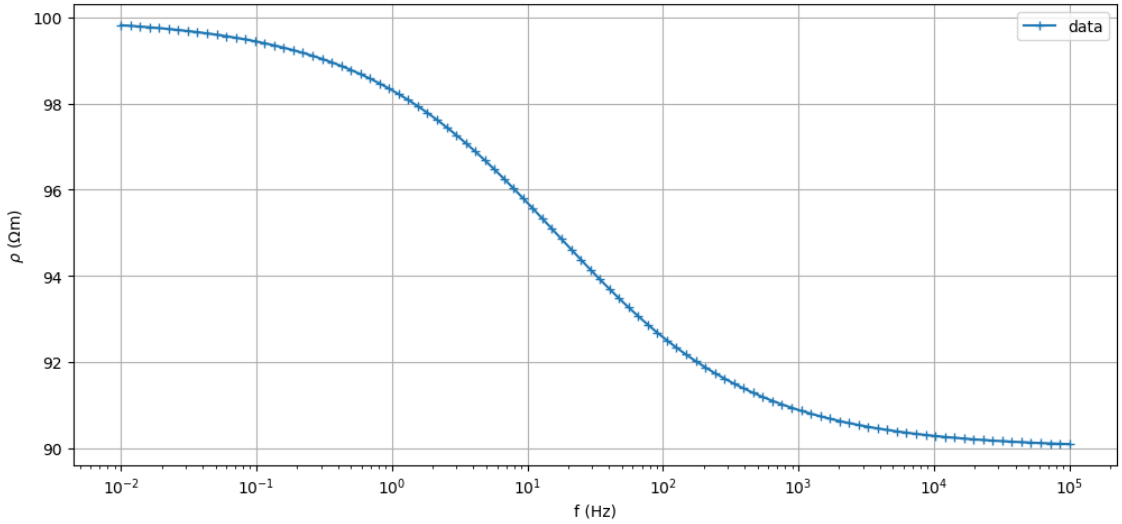
3. Fit a Cole-Cole model to

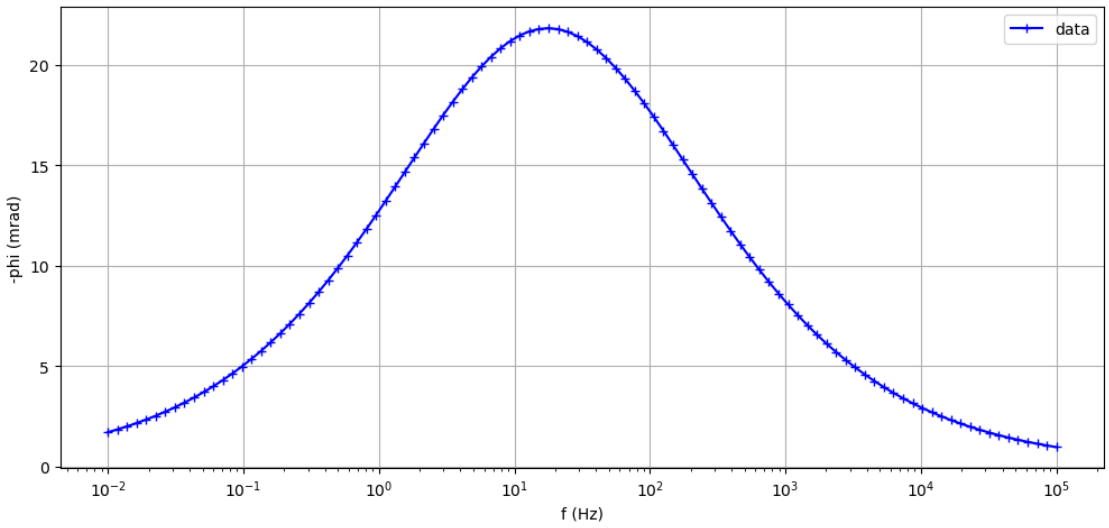
```
f = np.logspace(-2, 5, 100)
Z = modelColeColeRho(f, rho=100, m=0.1, tau=0.01, c=0.5)
sip = SIPSpectrum(f=f, amp=np.abs(Z), phi=-np.angle(Z))

sip.showAll()
sip.fitDebyeModel(new=True, showFit=True)
```

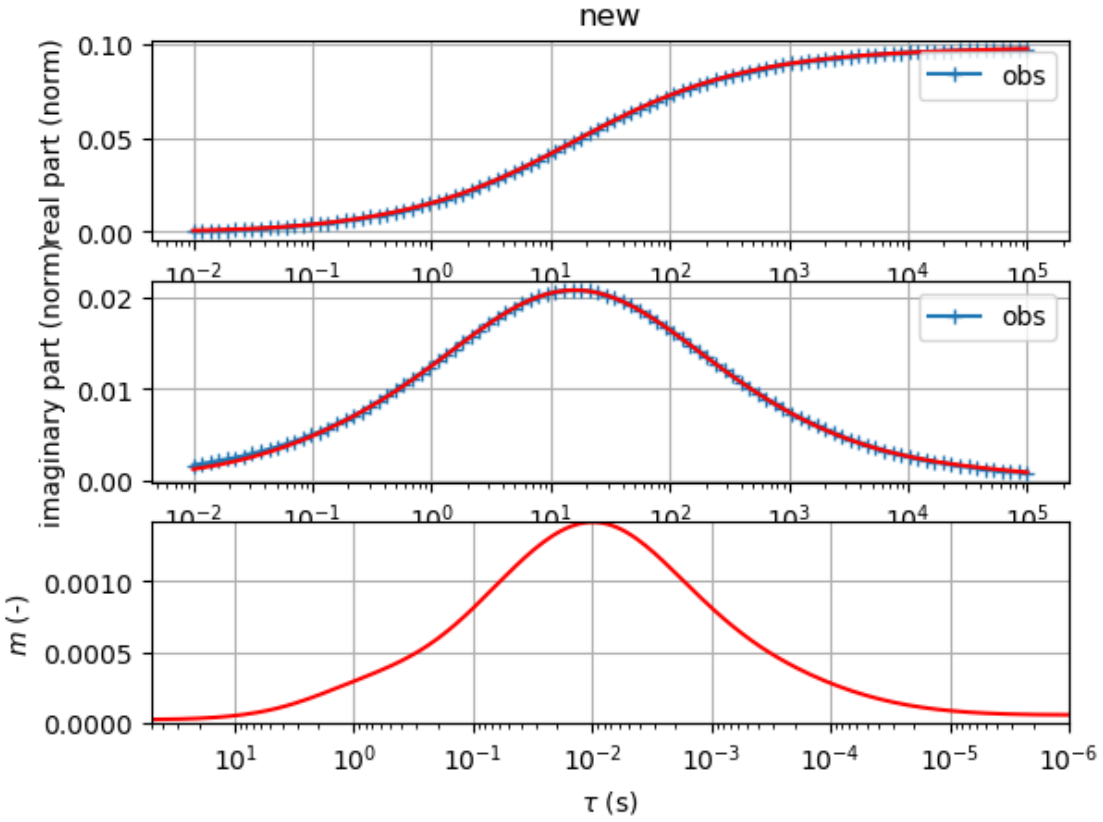
(continues on next page)

(continued from previous page)

```
pg.wait()
```








```
ARMS= 0.0001261341197615591 RRMS= 0.12935406598390997
```

Note: This tutorial was kindly contributed by Maximilian Weigand (University of Bonn). If you also want to contribute an interesting example, check out our contribution guidelines https://www.pygimli.org/contrib.html.

6.7.4.3 Complex-valued electrical modeling

In this example, an electrical complex-valued forward modeling is conducted. The use of complex resistivities implies an out-of-phase polarization response of the subsurface, commonly being measured in the frequency domain as complex resistivity (CR), or, if multiple frequencies are measured, also referred to as spectral induced polarization (SIP). Please note that the time-domain induced polarization (TDIP) is a compound signature over a wide range of frequencies.

It is common to parameterize the complex resistivities using magnitude (in Ωphi` (in mrad), although the pyGIMLi forward operator takes real and imaginary parts.

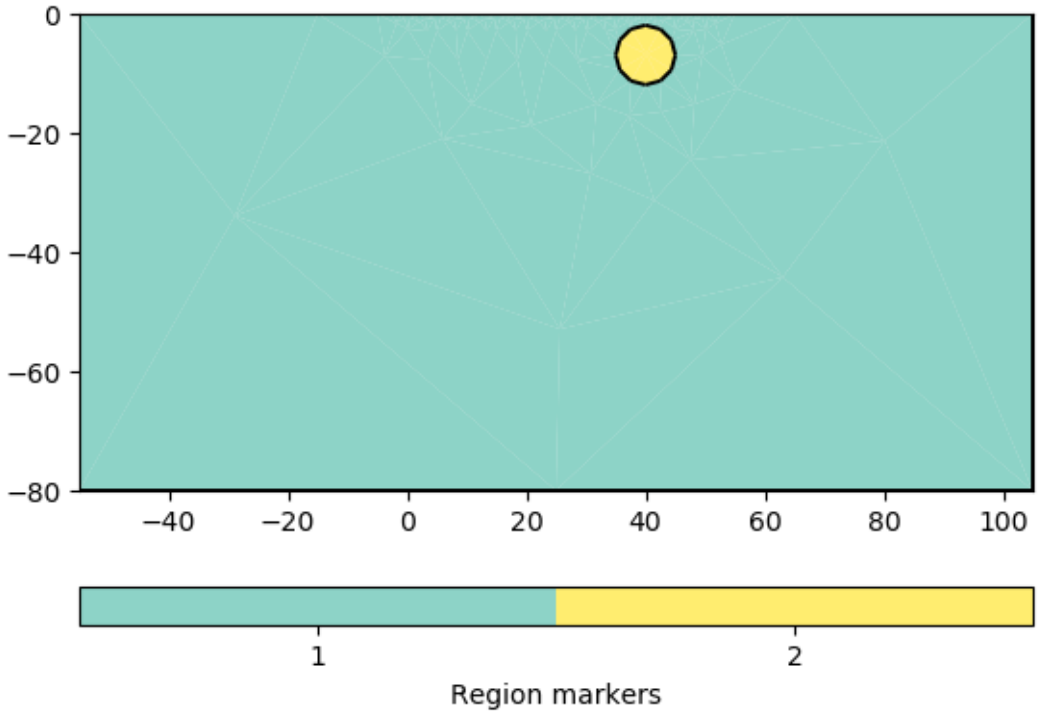
```
import numpy as np
import matplotlib.pylab as plt

import pygimli as pg
import pygimli.meshtools as mt
from pygimli.physics import ert
```

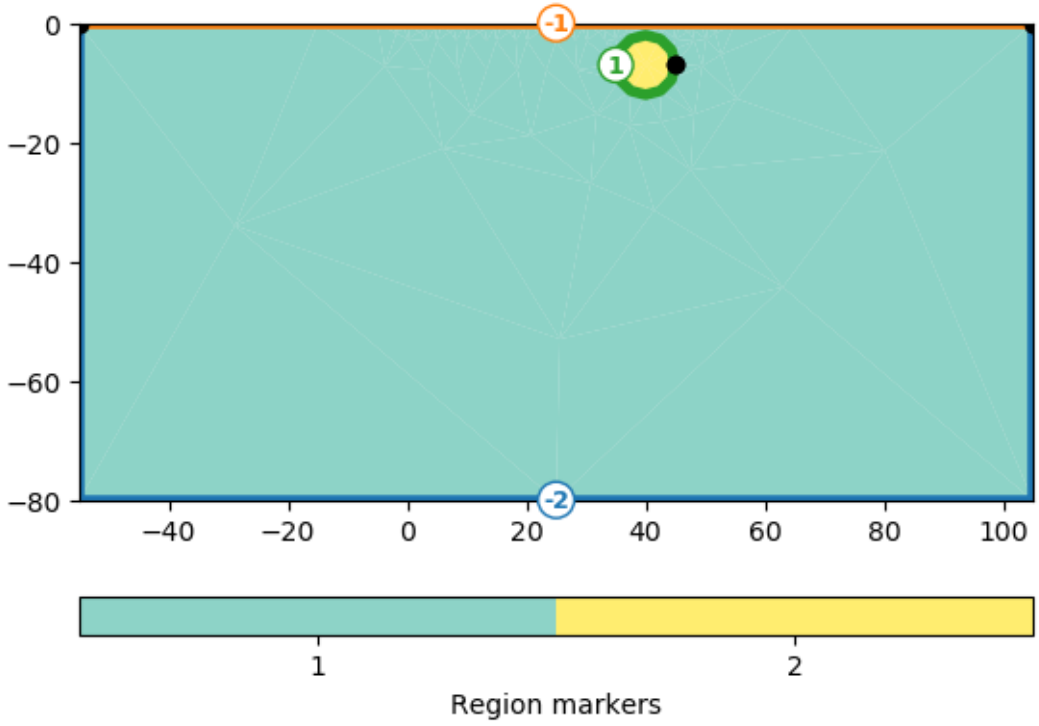
Create a measurement scheme for 51 electrodes with a spacing of 1m

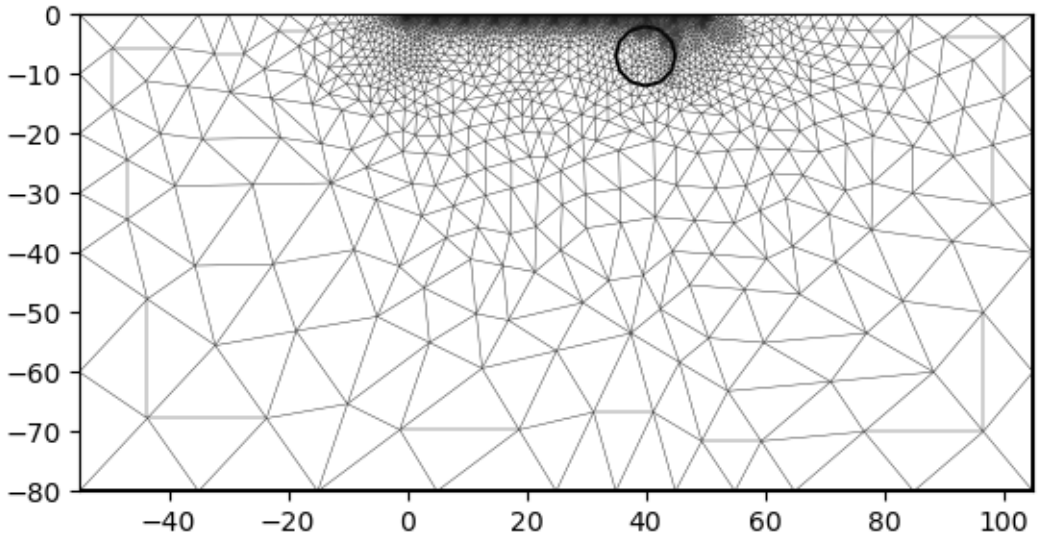
```
scheme = ert.createData(
    elecs=np.linspace(start=0, stop=50, num=51),
    schemeName='dd'
)
```

Mesh generation

```
world = mt.createWorld(
    start=[-55, 0], end=[105, -80], worldMarker=True)

polarizable_anomaly = mt.createCircle(
    pos=[40, -7], radius=5, marker=2
)

plc = world + polarizable_anomaly

# local refinement of mesh near electrodes
for s in scheme.sensors():
    plc.createNode(s + [0.0, -0.2])

mesh_coarse = mt.createMesh(plc, quality=33)
# additional refinements
mesh = mesh_coarse.createH2()

# Create a P2-optimized mesh (quadratic shape functions)
mesh = mesh.createP2()

pg.show(plc, marker=True)
pg.show(plc, markers=True)
pg.show(mesh)
```

- 
0
−20
−40
−60
−80
−40
−20
0
20
40
60
80
100
1
2
Region markers

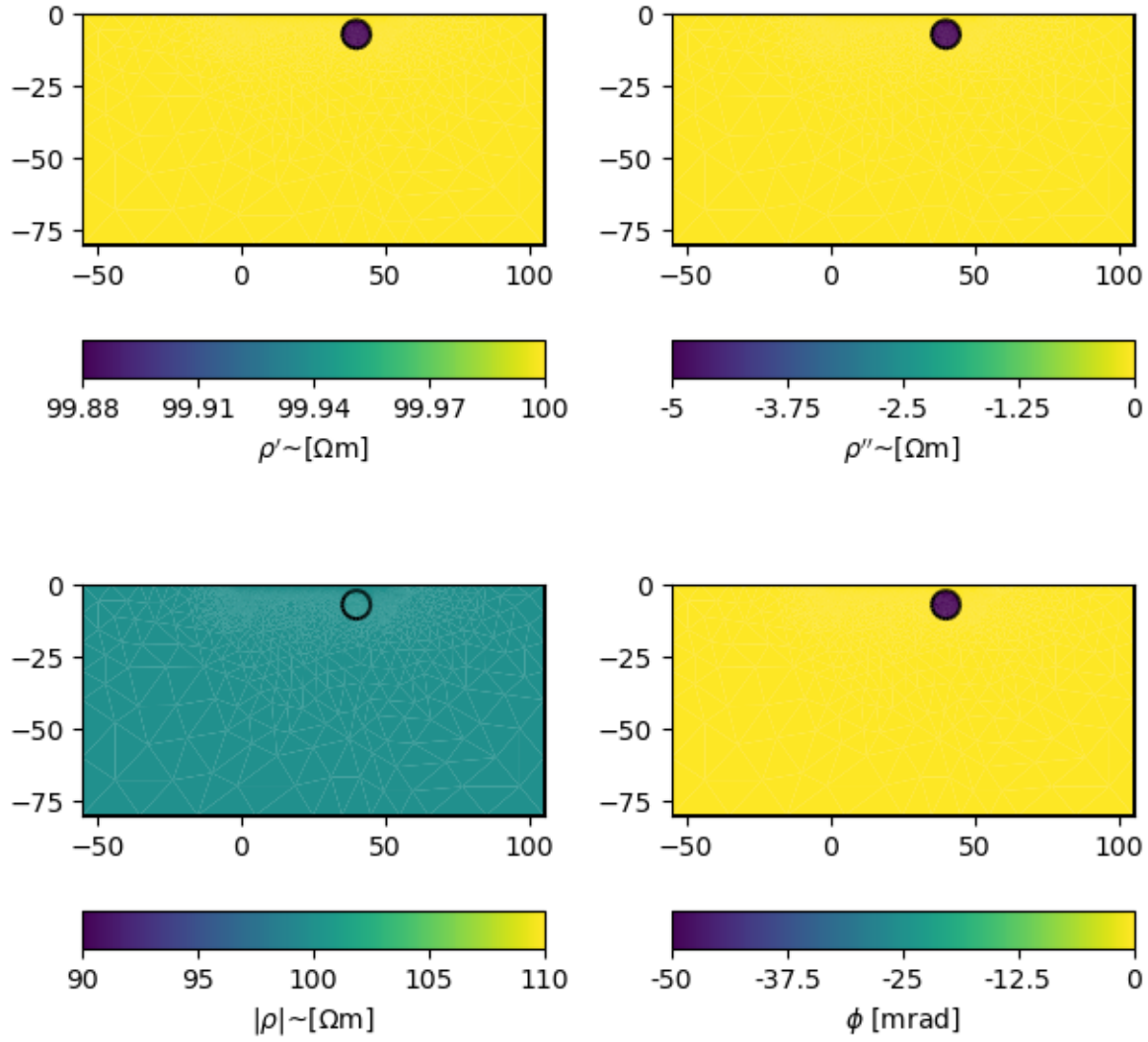
- 
-1
1
-2
0
−20
−40
−60
−80
−40
−20
0
20
40
60
80
100
1
2
Region markers




```
(<matplotlib.axes._subplots.AxesSubplot object at 0x7fe8fa434520>, None)
```

Prepare the model parameterization We have two markers here: 1: background 2: circle anomaly Parameters must be specified as a complex number, here converted by the utility function `pygimli.utils.complex.toComplex()`.

```
rhomap = [
    [1, pg.utils.complex.toComplex(100, 0 / 1000)],
    # Magnitude: 100 ohm m, Phase: -50 mrad
    [2, pg.utils.complex.toComplex(100, -50 / 1000)],
]

# For visualization, map the rhomap into the actual mesh, resulting␣
↪in a rho
# vector with a complex resistivity associated with each mesh cell.
rho = pg.solver.parseArgToArray(rhomap, mesh.cellCount(), mesh)
fig, axes = plt.subplots(2, 2, figsize=(16 / 2.54, 16 / 2.54))
pg.show(mesh, data=np.real(rho), ax=axes[0, 0], label=r"$\rho'$~[$\Omega$m]")
pg.show(mesh, data=np.imag(rho), ax=axes[0, 1], label=r"$\rho''$~[$\Omega$m]")
pg.show(mesh, data=np.abs(rho), ax=axes[1, 0], label=r"$|\rho$|~[$\Omega$m]")
pg.show(
    mesh, data=np.arctan2(np.imag(rho), np.real(rho))*1000,
    ax=axes[1, 1], label=r"$\phi$ [mrad]"
)
fig.tight_layout()
fig.show()
```



Do the actual forward modeling

```
data = ert.simulate(
    mesh,
    res=rhomap,
    scheme=scheme,
    # noiseAbs=0.0,
    # noiseLevel=0.0,
)
```

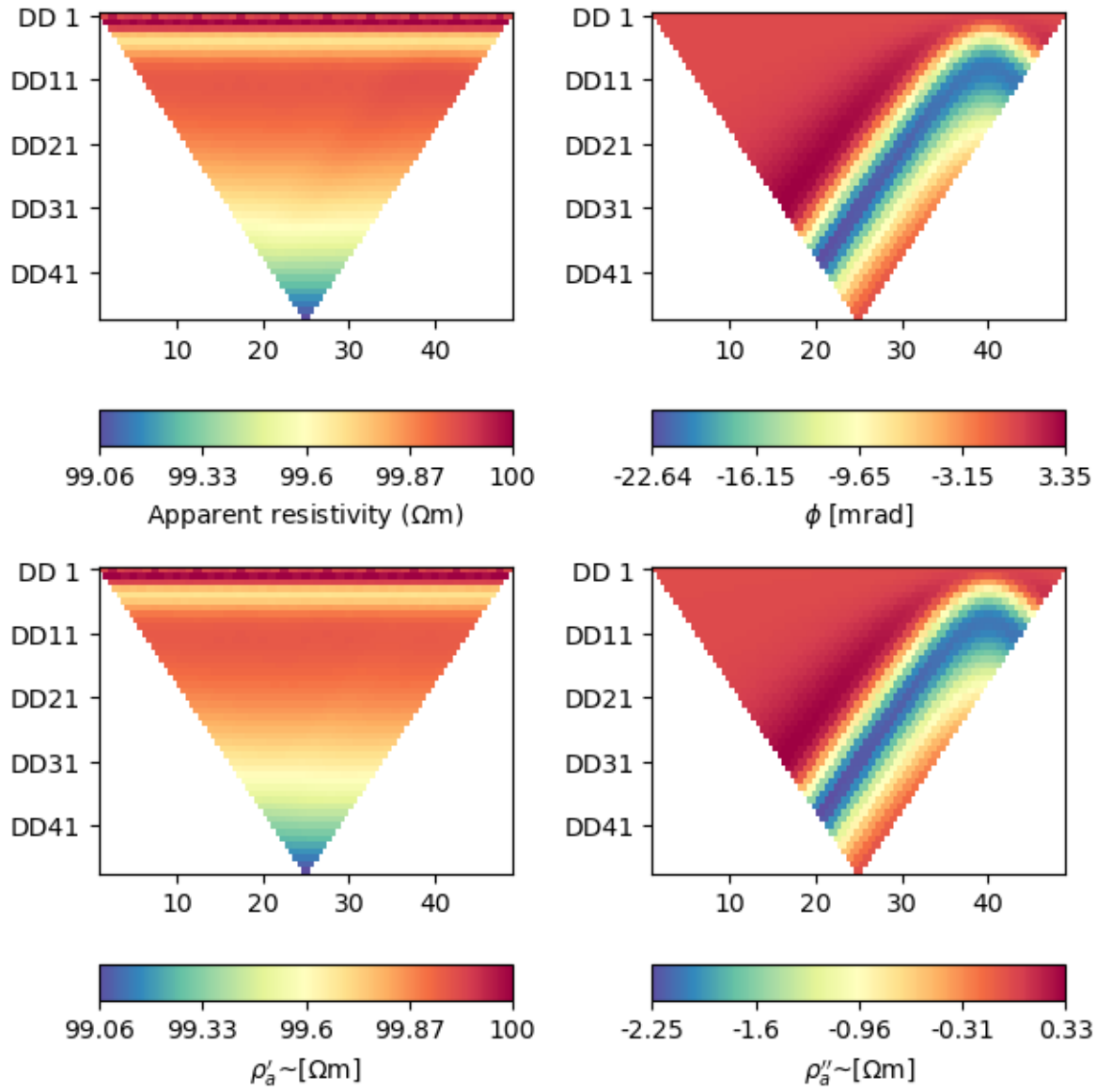
Visualize the modeled data Convert magnitude and phase into a complex apparent resistivity

```
rho_a_complex = data['rhoa'].array() * np.exp(1j * data['phia'].array())

# Please note the apparent negative (resistivity) phases!
fig, axes = plt.subplots(2, 2, figsize=(16 / 2.54, 16 / 2.54))
ert.showERTData(data, vals=data['rhoa'], ax=axes[0, 0])
# phia is stored in radians, but usually plotted in milliradians
ert.showERTData(
    data, vals=data['phia'] * 1000, label=r'$\phi$ [mrad]', ax=axes[0, 1])
```

(continues on next page)

(continued from previous page)

```
ert.showERTData(
    data, vals=np.real(rho_a_complex), ax=axes[1, 0],
    label=r"$\rho_a'$~[$\Omega$m]"
)
ert.showERTData(
    data, vals=np.imag(rho_a_complex), ax=axes[1, 1],
    label=r"$\rho_a''$~[$\Omega$m]"
)
fig.tight_layout()
fig.show()
```



Total running time of the script: (0 minutes 19.692 seconds)

6.7.4.4 Naive complex-valued electrical inversion

This example presents a quick and dirty proof-of-concept for a complex-valued inversion, similar to Kemna, 2000. The normal equations are solved using numpy, and no optimization with respect to running time and memory consumptions are applied. As such this example is only a technology demonstration and should **not** be used for real-world inversion of complex resistivity data!

Kemna, A.: Tomographic inversion of complex resistivity – theory and application, Ph.D. thesis, Ruhr-Universität Bochum, doi:10.1111/1365-2478.12013, 2000.

Note: This is a technology demonstration. Don't use this code for research. If you require a complex-valued inversion, please contact us at info@pygimli.org

---

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

import pygimli as pg
import pygimli.meshtools as mt
from pygimli.physics import ert
```

For reference we later plot the true complex resistivity model as reference

```
def get_scheme():
    scheme = ert.createData(elecs=np.linspace(start=0, stop=50, num=51),
                            schemeName='dd')
    # Not strictly required, but we switch potential electrodes to␣
↪yield
    # positive geometric factors. Note that this was also done for␣
↪the
    # synthetic data inverted later.
    m = scheme['m']
    n = scheme['n']
    scheme['m'] = n
    scheme['n'] = m
    scheme.set('k', [1 for x in range(scheme.size())])
    return scheme


def get_fwd_mesh():
    """Generate the forward mesh (with embedded anomalies)"""
    scheme = get_scheme()

    # Mesh generation
    world = mt.createWorld(
        start=[-55, 0], end=[105, -80], worldMarker=True)

    conductive_anomaly = mt.createCircle(
        pos=[10, -7], radius=5, marker=2
    )
```

(continues on next page)

(continued from previous page)

```
    polarizable_anomaly = mt.createCircle(
        pos=[40, -7], radius=5, marker=3
    )

    plc = mt.mergePLC((world, conductive_anomaly, polarizable_anomaly))

    # local refinement of mesh near electrodes
    for s in scheme.sensors():
        plc.createNode(s + [0.0, -0.2])

    mesh_coarse = mt.createMesh(plc, quality=33)
    mesh = mesh_coarse.createH2()
    return mesh


def generate_forward_data():
    """Generate synthetic forward data that we then invert"""
    scheme = get_scheme()

    mesh = get_fwd_mesh()

    rhomap = [
        [1, pg.utils.complex.toComplex(100, 0 / 1000)],
        # Magnitude: 50 ohm m, Phase: -50 mrad
        [2, pg.utils.complex.toComplex(50, 0 / 1000)],
        [3, pg.utils.complex.toComplex(100, -50 / 1000)],
    ]

    rho = pg.solver.parseArgToArray(rhomap, mesh.cellCount(), mesh)
    fig, axes = plt.subplots(2, 1, figsize=(16 / 2.54, 16 / 2.54))
    pg.show(
        mesh,
        data=np.log(np.abs(rho)),
        ax=axes[0],
        label=r"$log_{10}(|\rho|~[\Omega m])$"
    )
    pg.show(mesh, data=np.abs(rho), ax=axes[1], label=r"$|\rho|~[\Omega m]$")
    pg.show(
        mesh, data=np.arctan2(np.imag(rho), np.real(rho)) * 1000,
        ax=axes[1],
        label=r"$\phi$ [mrad]",
        cMap='jet_r'
    )
    data = ert.simulate(
        mesh,
        res=rhomap,
        scheme=scheme,
        # noiseAbs=0.0,
        # noiseLevel=0.0,
```

(continues on next page)

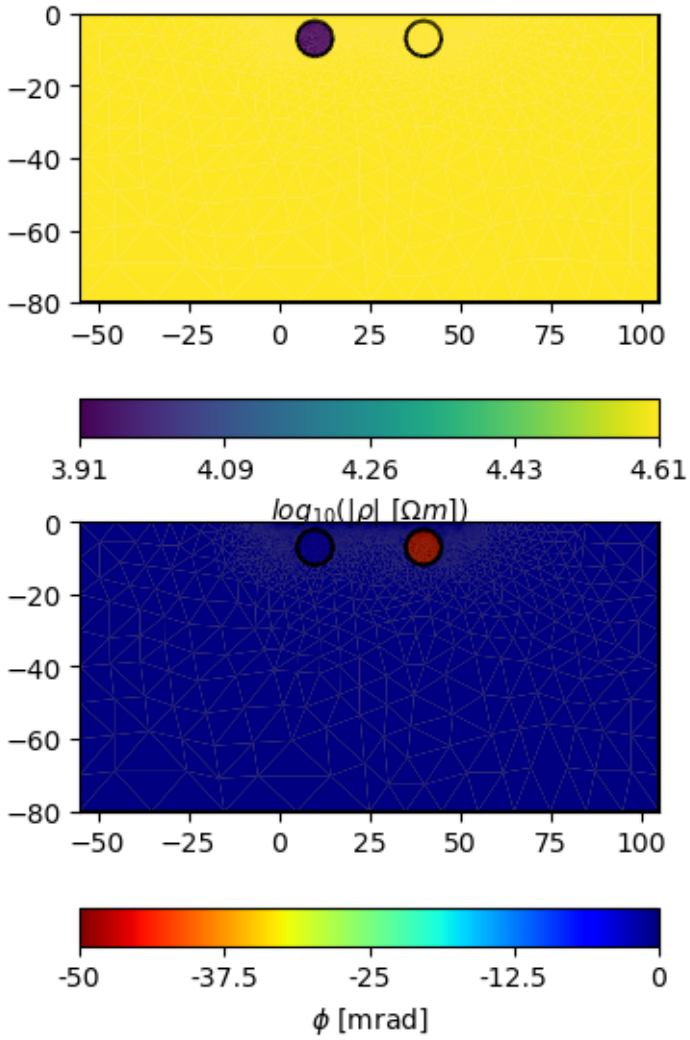
(continued from previous page)

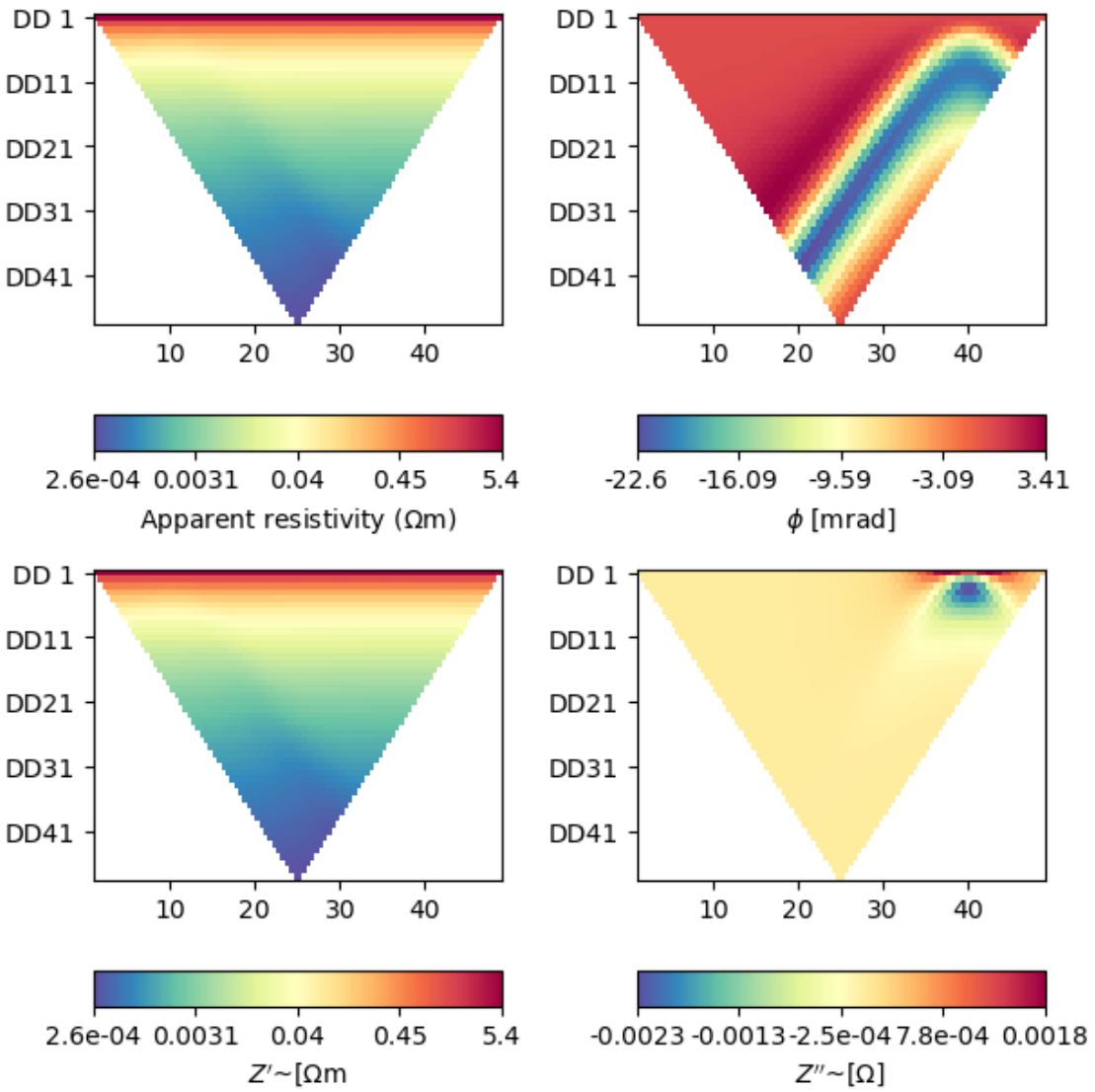
```
    )

    r_complex = data['rhoa'].array() * np.exp(1j * data['phia'].array())

    # Please note the apparent negative (resistivity) phases!
    fig, axes = plt.subplots(2, 2, figsize=(16 / 2.54, 16 / 2.54))
    ert.showERTData(data, vals=data['rhoa'], ax=axes[0, 0])

    # phia is stored in radians, but usually plotted in milliradians
    ert.showERTData(
        data, vals=data['phia'] * 1000, label=r'$\phi$ [mrad]', ax=axes[0, 1])

    ert.showERTData(
        data, vals=np.real(r_complex), ax=axes[1, 0],
        label=r"$Z'$~[$\Omega$m"
    )
    ert.showERTData(
        data, vals=np.imag(r_complex), ax=axes[1, 1],
        label=r"$Z''$~[$\Omega$]"
    )
    fig.tight_layout()
    fig.show()
    return r_complex


data_rcomplex = generate_forward_data()


def plot_fwd_model(axes):
    """This function plots the forward model used to generate the␣
↪data

     """
    # Mesh generation
    world = mt.createWorld(
        start=[-55, 0], end=[105, -80], worldMarker=True)

    conductive_anomaly = mt.createCircle(
        pos=[10, -7], radius=5, marker=2
    )

    polarizable_anomaly = mt.createCircle(
        pos=[40, -7], radius=5, marker=3
    )

    plc = mt.mergePLC((world, conductive_anomaly, polarizable_anomaly))

    # local refinement of mesh near electrodes
    for s in scheme.sensors():
```

(continues on next page)

(continued from previous page)

```
        plc.createNode(s + [0.0, -0.2])

    mesh_coarse = mt.createMesh(plc, quality=33)
    mesh = mesh_coarse.createH2()

    rhomap = [
        [1, pg.utils.complex.toComplex(100, 0 / 1000)],
        # Magnitude: 50 ohm m, Phase: -50 mrad
        [2, pg.utils.complex.toComplex(50, 0 / 1000)],
        [3, pg.utils.complex.toComplex(100, -50 / 1000)],
    ]

    rho = pg.solver.parseArgToArray(rhomap, mesh.cellCount(), mesh)
    pg.show(
        mesh,
        data=np.log(np.abs(rho)),
        ax=axes[0],
        label=r"$log_{10}(|\rho|~[\Omega m])$"
    )
    pg.show(mesh, data=np.abs(rho), ax=axes[1], label=r"$|\rho|~[\Omega m]$")
    pg.show(
        mesh, data=np.arctan2(np.imag(rho), np.real(rho)) * 1000,
        ax=axes[2],
        label=r"$\phi$ [mrad]",
        cMap='jet_r'
    )
    fig.tight_layout()
    fig.show()
```


0
−20
−40
−60
−80
−50
−25
0
25
50
75
100
3.91
4.09
4.26
4.43
4.61
$log_{10}(|\rho|\ [\Omega m])$
0
−20
−40
−60
−80
−50
−25
0
25
50
75
100
-50
-37.5
-25
-12.5
0
ϕ [mrad]



Create a measurement scheme for 51 electrodes, spacing 1

```
scheme = ert.createData(elecs=np.linspace(start=0, stop=50, num=51),
                        schemeName='dd')
# Not strictly required, but we switch potential electrodes to␣
↪yield positive
# geometric factors. Note that this was also done for the synthetic␣
↪data
# inverted later.
m = scheme['m']
n = scheme['n']
scheme['m'] = n
scheme['n'] = m
scheme.set('k', [1 for x in range(scheme.size())])
```

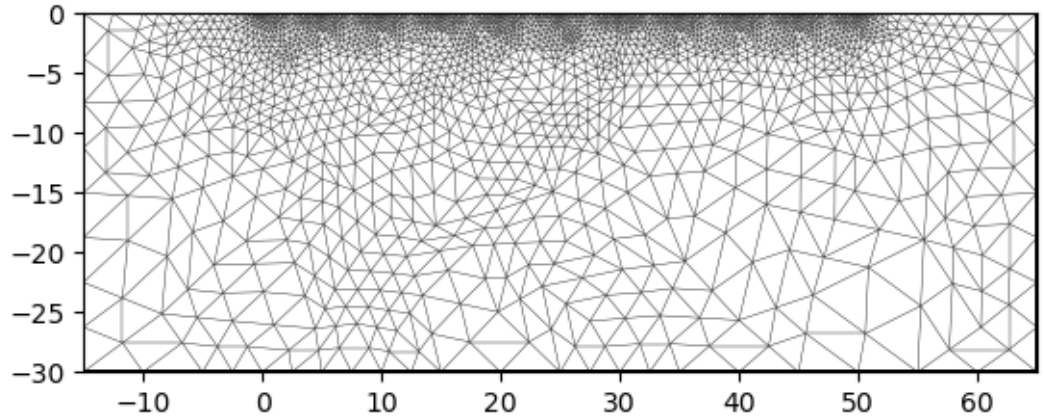
Mesh generation for the inversion

```
world = mt.createWorld(
    start=[-15, 0], end=[65, -30], worldMarker=False, marker=2)
```

(continues on next page)

(continued from previous page)

```
# local refinement of mesh near electrodes
for s in scheme.sensors():
    world.createNode(s + [0.0, -0.4])

mesh_coarse = mt.createMesh(world, quality=33)
mesh = mesh_coarse.createH2()
for nr, c in enumerate(mesh.cells()):
    c.setMarker(nr)
pg.show(mesh)
```




```
(<matplotlib.axes._subplots.AxesSubplot object at 0x7fe89af73a60>, None)
```

Define start model of the inversion [magnitude, phase]

```
start_model = np.ones(mesh.cellCount()) * pg.utils.complex.toComplex(
    80, -0.01 / 1000)
```

Initialize the complex forward operator

```
fop = ert.ERTModelling(
    sr=False,
    verbose=True,
)
fop.setComplex(True)
fop.setData(scheme)
fop.setMesh(mesh, ignoreRegionManager=True)
fop.mesh()
```

```
Mesh: Nodes: 2373 Cells: 4472 Boundaries: 6844
```

Compute response for the starting model

```
start_re_im = pg.utils.squeezeComplex(start_model)
f_0 = np.array(fop.response(start_re_im))
```

(continues on next page)

(continued from previous page)

```
# Compute the Jacobian for the starting model
J_block = fop.createJacobian(start_re_im)
J_re = np.array(J_block.mat(0))
J_im = np.array(J_block.mat(1))
J0 = J_re + 1j * J_im
```

Regularization matrix

```
rm = fop.regionManager()
rm.setMesh(mesh) # need to set here manually because of␣
↪ignoreRegionManager=True
rm.setVerbose(True)
rm.setConstraintType(2)

Wm = pg.matrix.SparseMapMatrix()
rm.fillConstraints(Wm)
#print(Wm)
#print(Wm.values())
Wm = pg.utils.sparseMatrix2coo(Wm)
#print(Wm)
```

read-in data and determine error parameters filename = pg.getExampleFile(

'CR/synthetic_modeling/data_rre_rim.dat', load=False, verbose=True)

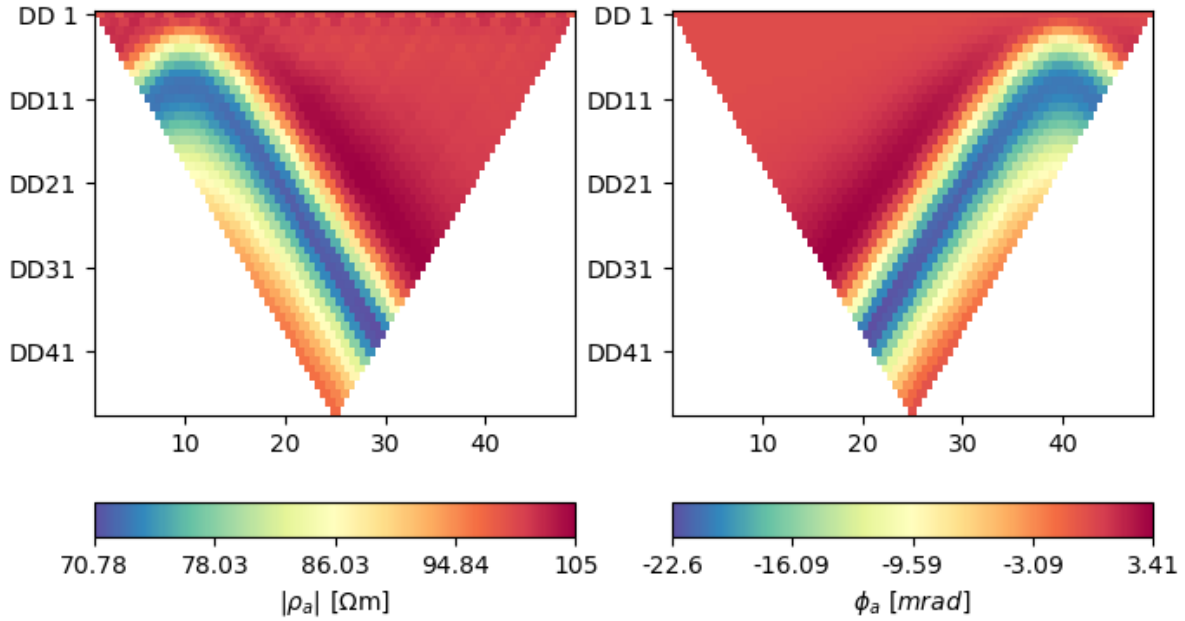
data_rre_rim = np.loadtxt(filename) N = int(data_rre_rim.size / 2) d_rcomplex = data_rre_rim[:N] + 1j * data_rre_rim[N:]

```
N = data_rcomplex.shape[0]
dmag = np.abs(data_rcomplex)
dpha = np.arctan2(data_rcomplex.imag, data_rcomplex.real) * 1000

fig, axes = plt.subplots(1, 2, figsize=(20 / 2.54, 10 / 2.54))
k = np.array(ert.createGeometricFactors(scheme))
ert.showERTData(
    scheme, vals=dmag * k, ax=axes[0], label=r'$|\rho_a|~[\Omega$m]')
ert.showERTData(scheme, vals=dpha, ax=axes[1], label=r'$\phi_a~[mrad]$')

# real part: log-magnitude
# imaginary part: phase [rad]
d_rlog = np.log(data_rcomplex)

# add some noise
np.random.seed(42)

noise_magnitude = np.random.normal(
    loc=0,
    scale=np.exp(d_rlog.real) * 0.04
)

# absolute phase error
```

(continues on next page)

(continued from previous page)

```
noise_phase = np.random.normal(
    loc=0,
    scale=np.ones(N) * (0.5 / 1000)
)

d_rlog = np.log(np.exp(d_rlog.real) + noise_magnitude) + 1j * (
    d_rlog.imag + noise_phase)

# crude error estimations
rmag_linear = np.exp(d_rlog.real)
err_mag_linear = rmag_linear * 0.04 + np.min(rmag_linear)
err_mag_log = np.abs(1 / rmag_linear * err_mag_linear)

Wd = np.diag(1.0 / err_mag_log)
WdTwd = Wd.conj().dot(Wd)
```



Put together one iteration of a naive inversion in log-log transformation d = log(V) m = log(sigma)

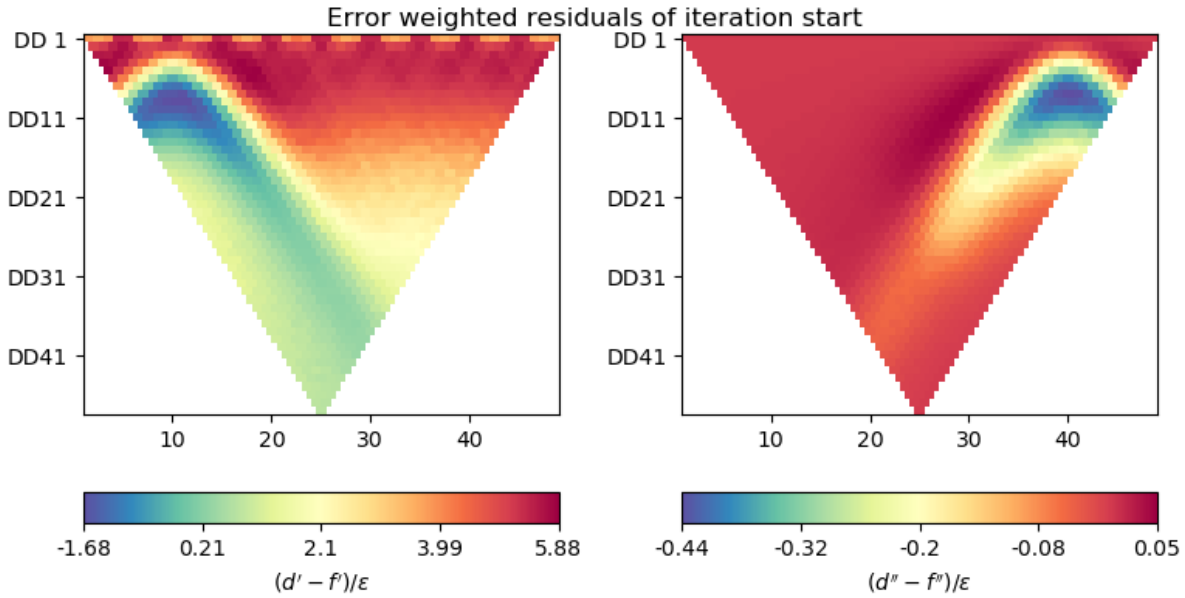
```
def plot_inv_pars(filename, d, response, Wd, iteration='start'):
    """Plot error-weighted residuals"""
    if 0:
        fig, axes = plt.subplots(1, 1, figsize=(20 / 2.54, 10 / 2.54))

        psi = np.abs(Wd.dot(d - response))

        ert.showERTData(
            scheme, vals=psi, ax=axes,
            label=r"$(d' - f') / \epsilon$"
        )
    else:
        fig, axes = plt.subplots(1, 2, figsize=(20 / 2.54, 10 / 2.54))
```

(continues on next page)

(continued from previous page)

```
        psi = Wd.dot(d - response)
        ert.showERTData(
            scheme, vals=psi.real, ax=axes[0],
            label=r"$(d' - f') / \epsilon$"
        )
        ert.showERTData(
            scheme, vals=psi.imag, ax=axes[1],
            label=r"$(d'' - f'') / \epsilon$"
        )

        fig.suptitle(
            'Error weighted residuals of iteration {}'.format(iteration), y=1.
↪00)

    fig.tight_layout()
```

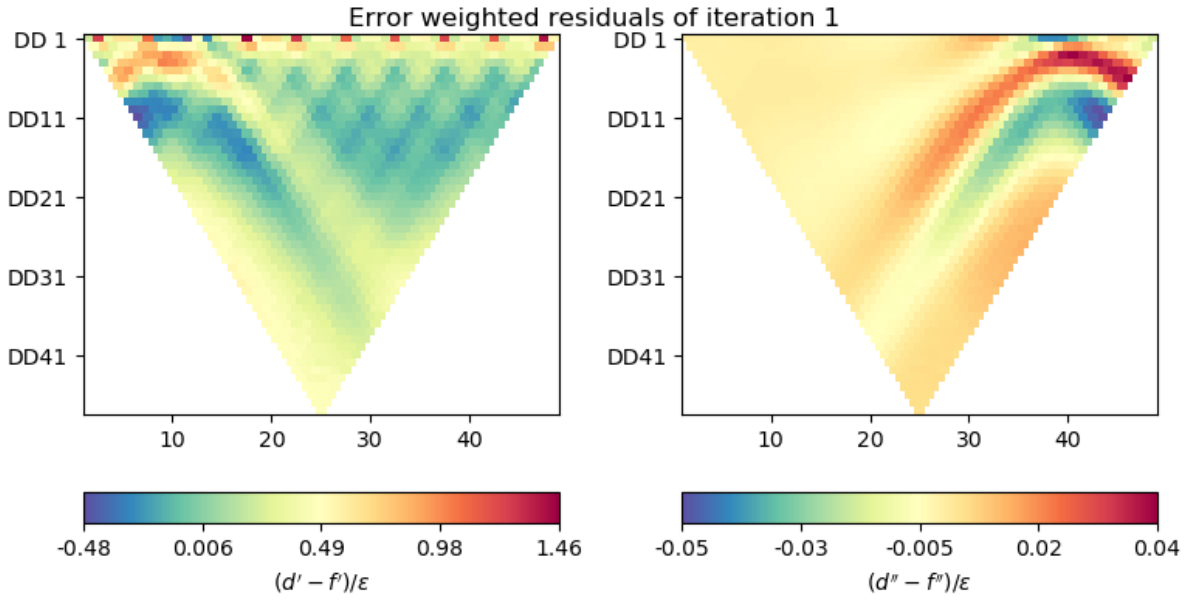
```
m_old = np.log(start_model)
# d = np.log(pg.utils.toComplex(data_rre_rim))
d = np.log(data_rcomplex)
response = np.log(pg.utils.toComplex(f_0))
# tranform to log-log sensitivities
J = J0 / np.exp(response[:, np.newaxis]) * np.exp(m_old)[np.newaxis, :]
lam = 100

plot_inv_pars('stats_it0.jpg', d, response, Wd)

# only one iteration is implemented here!
for i in range(1):
    print('-' * 80)
    print('Iteration {}'.format(i + 1))

    term1 = J.conj().T.dot(WdTwd).dot(J) + lam * Wm.T.dot(Wm)
    # term1_inverse = np.linalg.inv(term1)
    term2 = J.conj().T.dot(WdTwd).dot(d - response) - lam * Wm.T.dot(Wm).dot(
        m_old)
    # model_update = term1_inverse.dot(term2)
    model_update = np.linalg.solve(
        term1,
        term2
    )

    print('Model Update')
    print(model_update)

    m1 = np.array(m_old + 1.0 * model_update).squeeze()
```




```
--------------------------------------------------------------------------------
Iteration 1
Model Update
[0.19006749-7.60207166e-05j 0.18687016-5.68031681e-05j
 0.19731258-1.17414927e-04j ... 0.11804832-1.64495115e-03j
 0.11011347-1.67094449e-03j 0.11465995-1.66143964e-03j]
```

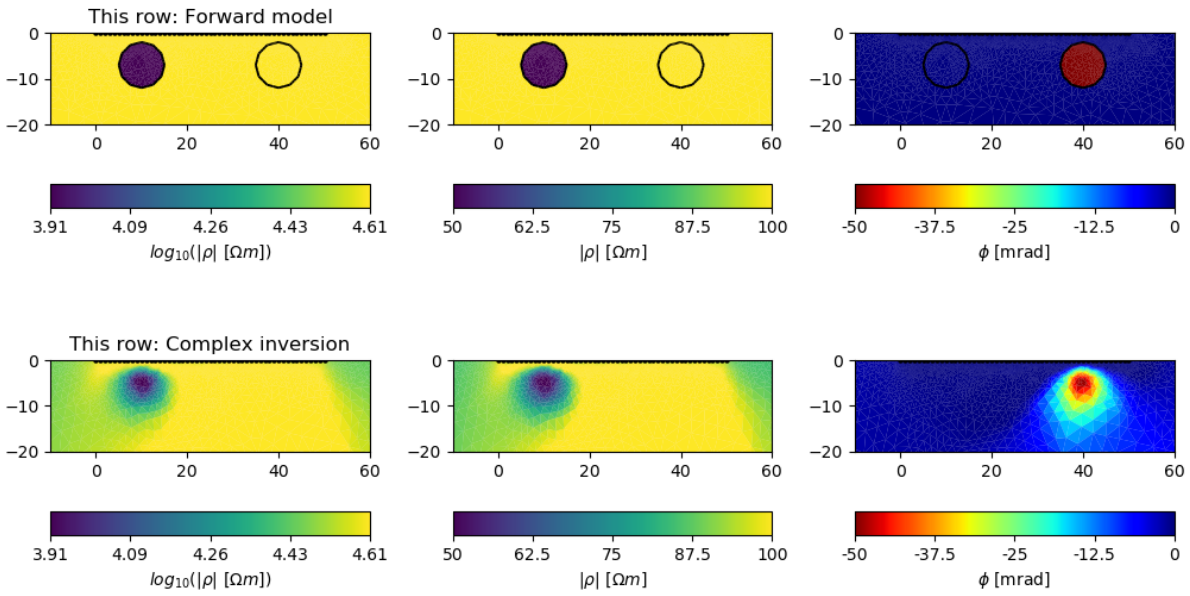
Now plot the residuals for the first iteration

```
m_old = m1

# Response for Starting model
m_re_im = pg.utils.squeezeComplex(np.exp(m_old))
response_re_im = np.array(fop.response(m_re_im))
response = np.log(pg.utils.toComplex(response_re_im))

plot_inv_pars('stats_it{}.jpg'.format(i + 1), d, response, Wd, iteration=1)
```



And finally, plot the inversion results

```
fig, axes = plt.subplots(2, 3, figsize=(26 / 2.54, 15 / 2.54))
plot_fwd_model(axes[0, :])
axes[0, 0].set_title('This row: Forward model')

pg.show(
    mesh, data=m1.real, ax=axes[1, 0],
    cMin=np.log(50),
    cMax=np.log(100),
    label=r"$log_{10}(|\rho|~[\Omega m])$"
)
pg.show(
    mesh, data=np.exp(m1.real), ax=axes[1, 1],
    cMin=50, cMax=100,
    label=r"$|\rho|~[\Omega m]$"
)
pg.show(
    mesh, data=m1.imag * 1000, ax=axes[1, 2], cMap='jet_r',
    label=r"$\phi$ [mrad]",
    cMin=-50, cMax=0,
)

axes[1, 0].set_title('This row: Complex inversion')

for ax in axes.flat:
    ax.set_xlim(-10, 60)
    ax.set_ylim(-20, 0)
    for s in scheme.sensors():
        ax.scatter(s[0], s[1], color='k', s=5)

fig.tight_layout()
```



Total running time of the script: (0 minutes 45.108 seconds)

6.7.5 Gravimetry and magnetics

6.7.5.1 Gravimetry in 2D - Part I

Simple gravimetric field solution.

Calculate for the gravimetric potential u

$$\frac{\partial u}{\partial z}$$

along a profile for a cylindrical heterogeneity with different approaches.

```
import numpy as np
import pygimli as pg
from pygimli.meshtools import createCircle, createWorld, createMesh

from pygimli.physics.gravimetry import gradUCylinderHoriz, solveGravimetry

radius = 2.  # [m]
depth = 5.  # [m]
pos = [0., -depth]
dRho = 100

x = np.arange(-20, 20.1, .5)
pnts = np.array([x, np.zeros(len(x))]).T
```

Analytical solution first

```
gz_a = gradUCylinderHoriz(pnts, radius, dRho, pos)[:, 1]
```

Integration for a 2D polygon after [**?**]

```
circ = createCircle([0, -depth], radius=radius, marker=2, area=0.1,
                    nSegments=16)
gz_p = solveGravimetry(circ, dRho, pnts, complete=False)
```

Integration for complete 2D mesh after [?]

```
world = createWorld(start=[-20, 0], end=[20, -10], marker=1)
mesh = createMesh([world, circ])
dRhoC = pg.solver.parseMapToCellArray([[1, 0.0], [2, dRho]], mesh)
gc_m = solveGravimetry(mesh, dRhoC, pnts)
```

Finite Element solution for u

```
world = createWorld(start=[-200, 200], end=[200, -200], marker=1)

# Add some nodes to the measurement points to increase the accuracy␣
↪a bit
[world.createNode(x_, 0.0,  1) for x_ in x]
plc = world + circ
mesh = createMesh(plc, quality=34)
mesh = mesh.createP2()

density = pg.solver.parseMapToCellArray([[1, 0.0], [2, dRho]], mesh)
u = pg.solver.solve(mesh, a=1, f=density, bc={'Dirichlet': {-2:0, -1:0}})
```

Calculate gradient of gravimetric potential $\frac{\partial u}{\partial(x,z)}$

```
dudz = np.zeros(len(pnts))

for i, p in enumerate(pnts):
    c = mesh.findCell(p)
    g = c.grad(p, u)
    dudz[i] = -g[1] * 4. * np.pi * pg.physics.constants.GmGal  # why 4 pi␣
↪here?
```

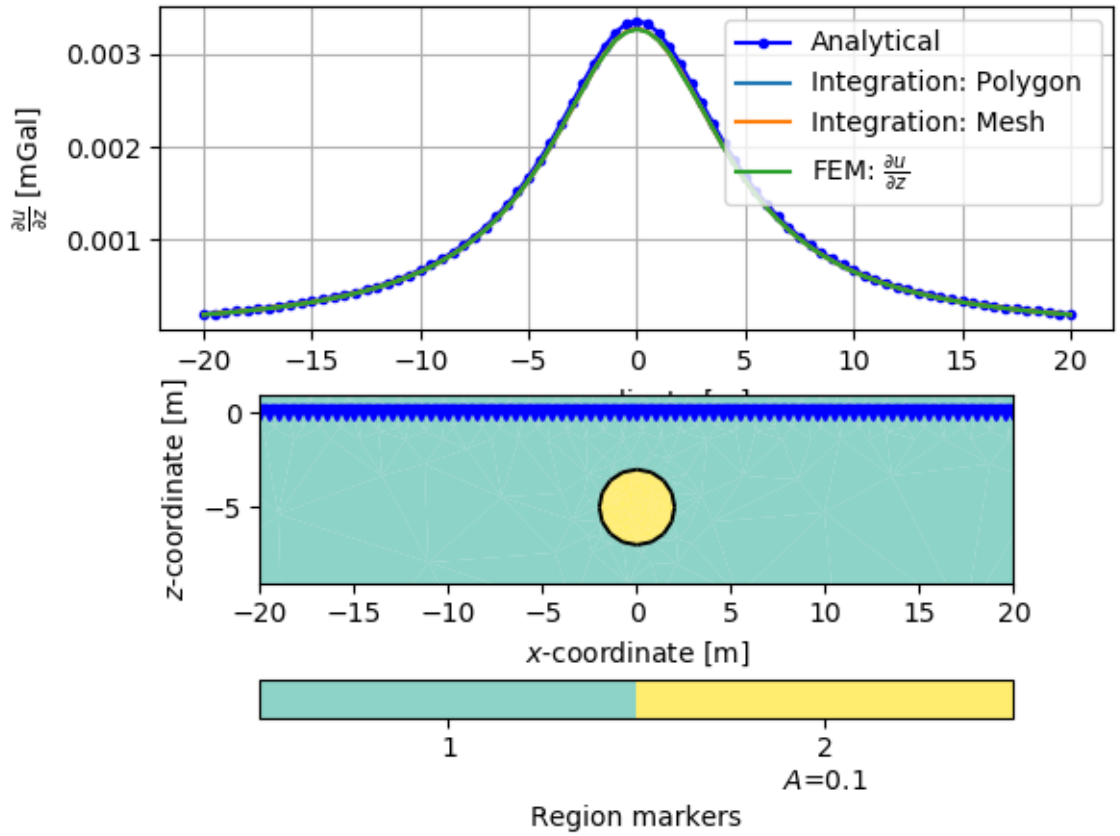
Finishing the plots

```
ax1 = pg.plt.subplot(2, 1, 1)
ax1.plot(x, gz_a, '-b', marker='.', label='Analytical')
ax1.plot(x, gz_p, label='Integration: Polygon ')
ax1.plot(x, gc_m, label='Integration: Mesh')
ax1.plot(x, dudz, label=r'FEM: $\frac{\partial u}{\partial z}$')

ax2 = pg.plt.subplot(2, 1, 2)
pg.show(plc, ax=ax2)
ax2.plot(x, x*0,  'bv')

ax1.set_ylabel(r'$\frac{\partial u}{\partial z}$ [mGal]')
ax1.set_xlabel('$x$-coordinate [m]')
ax1.grid()
ax1.legend()
```

(continues on next page)

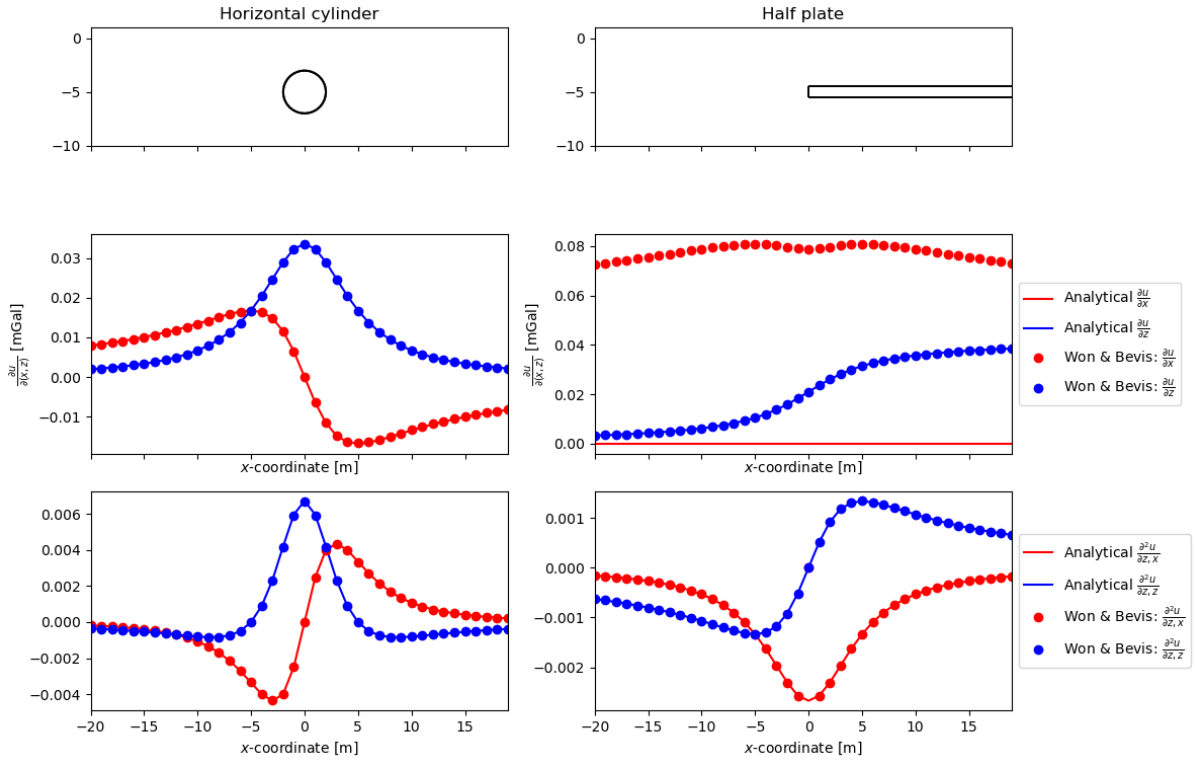
(continued from previous page)

```
ax2.set_aspect(1)
ax2.set_xlabel('$x$-coordinate [m]')
ax2.set_ylabel('$z$-coordinate [m]')
ax2.set_ylim((-9, 1))
ax2.set_xlim((-20, 20))

pg.wait()
```



6.7.5.2 Semianalytical Gravimetry and Geomagnetics in 2D

Simple gravimetric and magnetostatic field caluculation using integration approach after [**?**].




```
import numpy as np
import pygimli as pg
from pygimli.meshtools import createCircle
from pygimli.physics.gravimetry import (gradGZCylinderHoriz,
                                        gradGZHalfPlateHoriz,
                                        gradUCylinderHoriz,
                                        gradUHalfPlateHoriz, solveGravimetry)

radius = 2.
depth = 5.
rho = 1000.0

x = np.arange(-20, 20, 1)
pnts = np.zeros((len(x), 2))
pnts[:, 0] = x
pos = [0, -depth]


def plot(x, a1, ga, gza, a2, g, gz, legend=True):
    a1.plot(x, ga[:, 0],  label=r'Analytical $\frac{\partial u}{\partial x}$',␣
↪c="red")
    a1.plot(x, ga[:, 1],  label=r'Analytical $\frac{\partial u}{\partial z}$',␣
↪c="blue")

    a1.plot(x, g[:, 0], label=r'Won & Bevis: $\frac{\partial u}{\partial x}$',
            marker='o', linewidth=0, c="red")
    a1.plot(x, g[:, 2], label=r'Won & Bevis: $\frac{\partial u}{\partial z}$',
            marker='o', linewidth=0, c="blue")
```

(continues on next page)

(continued from previous page)

```
    a2.plot(x, gza[:, 0],
            label=r'Analytical $\frac{\partial^2 u}{\partial z,x}$', c="red")
    a2.plot(x, gza[:, 1],
            label=r'Analytical $\frac{\partial^2 u}{\partial z,z}$', c="blue")

    a2.plot(x, gz[:, 0], marker='o', linestyle='',
            label=r'Won & Bevis: $\frac{\partial^2 u}{\partial z,x}$', c="red")
    a2.plot(x, gz[:, 2], marker='o', linestyle='',
            label=r'Won & Bevis: $\frac{\partial^2 u}{\partial z,z}$', c="blue")
    a1.set_xlabel('$x$-coordinate [m]')
    a1.set_ylabel(r'$\frac{\partial u}{\partial (x,z)}$ [mGal]')

    a2.set_xlabel('$x$-coordinate [m]')

    if legend:
        a1.legend(loc='center left', bbox_to_anchor=(1, 0.5))
        a2.legend(loc='center left', bbox_to_anchor=(1, 0.5))

fig, ax = pg.plt.subplots(nrows=3, ncols=2, figsize=(12,8), sharex=True)

# Horizontal cylinder
circ = createCircle([0, -depth], radius=radius, marker=2, area=0.1,
                    nSegments=32)

pg.show(circ, ax=ax[0,0], fillRegion=False)

ga = gradUCylinderHoriz(pnts, radius, rho, pos=pos)
gza = gradGZCylinderHoriz(pnts, radius, rho, pos=pos)
g, gz = solveGravimetry(circ, rho, pnts, complete=True)

plot(x, ax[1,0], ga, gza, ax[2,0], g, gz, legend=False)

# Half plate
thickness = 1
mesh = pg.createGrid(x=np.linspace(0,5000),
                     y=[-depth-thickness/2., -depth+thickness/2.0])
pg.show(mesh, ax=ax[0,1])

ga = gradUHalfPlateHoriz(pnts, thickness, rho, pos=[0, -depth])
gza = gradGZHalfPlateHoriz(pnts, thickness, rho, pos=[0, -depth])
g, gz = solveGravimetry(mesh, rho, pnts, complete=True)

plot(x, ax[1,1], ga, gza, ax[2,1], g, gz)

labels = ["Horizontal cylinder", "Half plate"]
for ax, label in zip(ax[0], labels):
    ax.set_title(label)
    ax.set_aspect("equal")
```

(continues on next page)

(continued from previous page)

```
    ax.set_xlim(left=x[0], right=x[-1])
    ax.set_ylim(bottom=-depth*2, top=1)

fig.tight_layout()
pg.wait()
```

6.7.5.3 3D magnetics modelling and inversion

Based on the synthetic model of Li & Oldenburg (1996), we demonstrate 3D inversion of magnetic data. The forward operator bases on the formula given by Holstein et al. (2007).

In the following, we will build the model, create synthetic data, and do inversion using a depth-weighting function as outlined in the paper.

```
import numpy as np
import pygimli as pg
from pygimli.viewer import pv
from pygimli.physics.gravimetry import MagneticsModelling
```

Synthetic model and data generation

The grid is 1x1x0.5 km with a spacing of 50 m.

```
dx = 50
x = np.arange(0., 1001, dx)
y = np.arange(0., 1001, dx)
z = np.arange(0., 501, dx)
grid = pg.createGrid(x, y, z)
print(grid)
```

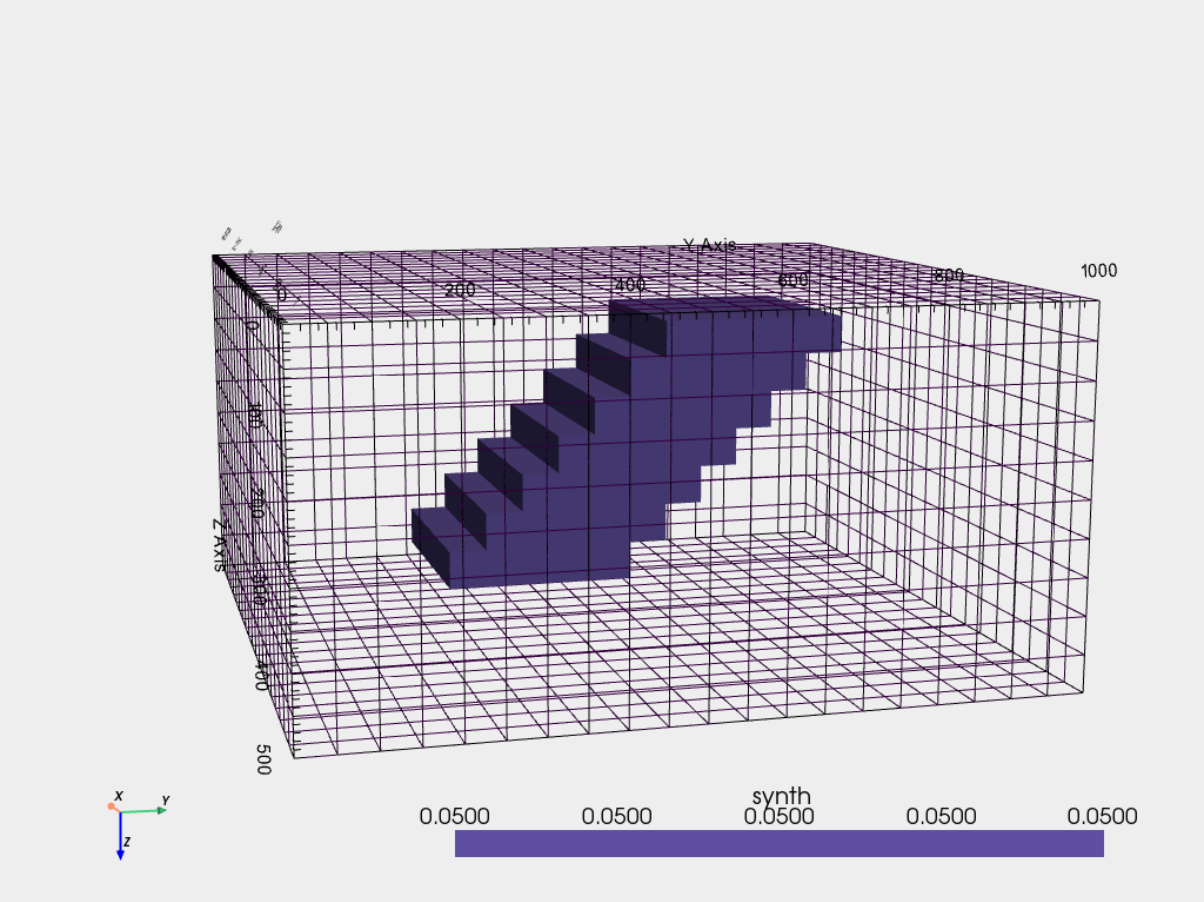
```
Mesh: Nodes: 4851 Cells: 4000 Boundaries: 12800
```

We create a 3D matrix that is later filled as vector into the grid. The model consists of zeros and patches of 5x6 cells per depth slice that are shifted by one cell for subsequent cells.

```
v = np.zeros((len(z)-1, len(y)-1, len(x)-1))
for i in range(7):
    v[1+i, 11-i:16-i, 7:13] = 0.05

grid["synth"] = v.ravel()
```

We show the model making use of the pyvista package that can be called by `pygimli.show()`. The mesh itself is shown as a wireframe, the anomaly is plotted as a surface plot using a threshold filter. After the first call with `hold=True`, the plotter is used to draw any subsequent plots that can also be slices or clips. Moreover, the camera position is set so that the vertical axis is going downwards (x is Northing and y is Easting as common in magnetics).

```
pl, _ = pg.show(grid, style="wireframe", hold=True)
pv.drawMesh(pl, grid, label="synth", style="surface", cMap="Spectral_r",
            filter={"threshold": dict(value=0.05, scalars="synth")})
pl.camera_position = "yz"
pl.camera.roll = 90
pl.camera.azimuth = 180 - 15
pl.camera.elevation = 10
pl.camera.zoom(1.2)
pl.show()
```




```
False
```

For the computation of the total field, we define the global magnetic field using the IGRF (total field, inclination and declination) settings given in the paper. Any global field can also be retrieved by the `pyIGRF` module.

```
F, I, D = 50000e-9, 75, 25
H = F * np.cos(np.deg2rad(I))
X = H * np.cos(np.deg2rad(D))
Y = H * np.sin(np.deg2rad(D))
Z = F * np.sin(np.deg2rad(I))
igrf = [D, I, H, X, Y, Z, F]

py, px = np.meshgrid(x, y)
```

(continues on next page)

(continued from previous page)

```
px = px.ravel()
py = py.ravel()
points = np.column_stack((px, py, -np.ones_like(px)*20))

# The forward operator
cmp = ["TFA"]  # ["Bx", "By", "Bz"]
fop = MagneticsModelling(grid, points, cmp, igrf)
model = pg.Vector(grid.cellCount(), 1.0)
data = fop.response(grid["synth"])
```

```
[                    0%                    ] 1 of 441 complete
[                    0%                    ] 2 of 441 complete
[                    1%                    ] 3 of 441 complete
[                    1%                    ] 4 of 441 complete
[                    1%                    ] 5 of 441 complete
[                    1%                    ] 6 of 441 complete
[+                   2%                    ] 7 of 441 complete
[+                   2%                    ] 8 of 441 complete
[+                   2%                    ] 9 of 441 complete
[+                   2%                    ] 10 of 441 complete
[+                   2%                    ] 11 of 441 complete
[+                   3%                    ] 12 of 441 complete
[+                   3%                    ] 13 of 441 complete
[+                   3%                    ] 14 of 441 complete
[+                   3%                    ] 15 of 441 complete
[++                  4%                    ] 16 of 441 complete
[++                  4%                    ] 17 of 441 complete
[++                  4%                    ] 18 of 441 complete
[++                  4%                    ] 19 of 441 complete
[++                  5%                    ] 20 of 441 complete
[++                  5%                    ] 21 of 441 complete
[++                  5%                    ] 22 of 441 complete
[++                  5%                    ] 23 of 441 complete
[++                  5%                    ] 24 of 441 complete
[++                  6%                    ] 25 of 441 complete
[++                  6%                    ] 26 of 441 complete
[++                  6%                    ] 27 of 441 complete
[++                  6%                    ] 28 of 441 complete
[+++                 7%                    ] 29 of 441 complete
[+++                 7%                    ] 30 of 441 complete
[+++                 7%                    ] 31 of 441 complete
[+++                 7%                    ] 32 of 441 complete
[+++                 7%                    ] 33 of 441 complete
[+++                 8%                    ] 34 of 441 complete
[+++                 8%                    ] 35 of 441 complete
[+++                 8%                    ] 36 of 441 complete
[+++                 8%                    ] 37 of 441 complete
[+++                 9%                    ] 38 of 441 complete
[+++                 9%                    ] 39 of 441 complete
```

(continues on next page)

(continued from previous page)

```
[+++                 9%                  ] 40 of 441 complete
[+++                 9%                  ] 41 of 441 complete
[++++               10%                  ] 42 of 441 complete
[++++               10%                  ] 43 of 441 complete
[++++               10%                  ] 44 of 441 complete
[++++               10%                  ] 45 of 441 complete
[++++               10%                  ] 46 of 441 complete
[++++               11%                  ] 47 of 441 complete
[++++               11%                  ] 48 of 441 complete
[++++               11%                  ] 49 of 441 complete
[++++               11%                  ] 50 of 441 complete
[+++++              12%                  ] 51 of 441 complete
[+++++              12%                  ] 52 of 441 complete
[+++++              12%                  ] 53 of 441 complete
[+++++              12%                  ] 54 of 441 complete
[+++++              12%                  ] 55 of 441 complete
[+++++              13%                  ] 56 of 441 complete
[+++++              13%                  ] 57 of 441 complete
[+++++              13%                  ] 58 of 441 complete
[+++++              13%                  ] 59 of 441 complete
[+++++              14%                  ] 60 of 441 complete
[+++++              14%                  ] 61 of 441 complete
[+++++              14%                  ] 62 of 441 complete
[+++++              14%                  ] 63 of 441 complete
[++++++             15%                  ] 64 of 441 complete
[++++++             15%                  ] 65 of 441 complete
[++++++             15%                  ] 66 of 441 complete
[++++++             15%                  ] 67 of 441 complete
[++++++             15%                  ] 68 of 441 complete
[++++++             16%                  ] 69 of 441 complete
[++++++             16%                  ] 70 of 441 complete
[++++++             16%                  ] 71 of 441 complete
[++++++             16%                  ] 72 of 441 complete
[++++++             17%                  ] 73 of 441 complete
[++++++             17%                  ] 74 of 441 complete
[++++++             17%                  ] 75 of 441 complete
[++++++             17%                  ] 76 of 441 complete
[++++++             17%                  ] 77 of 441 complete
[+++++++            18%                  ] 78 of 441 complete
[+++++++            18%                  ] 79 of 441 complete
[+++++++            18%                  ] 80 of 441 complete
[+++++++            18%                  ] 81 of 441 complete
[+++++++            19%                  ] 82 of 441 complete
[+++++++            19%                  ] 83 of 441 complete
[+++++++            19%                  ] 84 of 441 complete
[+++++++            19%                  ] 85 of 441 complete
[++++++++           20%                  ] 86 of 441 complete
[++++++++           20%                  ] 87 of 441 complete
[++++++++           20%                  ] 88 of 441 complete
```

(continues on next page)

(continued from previous page)

```
[++++++++          20%                ] 89 of 441 complete
[++++++++          20%                ] 90 of 441 complete
[++++++++          21%                ] 91 of 441 complete
[++++++++          21%                ] 92 of 441 complete
[++++++++          21%                ] 93 of 441 complete
[++++++++          21%                ] 94 of 441 complete
[++++++++          22%                ] 95 of 441 complete
[++++++++          22%                ] 96 of 441 complete
[++++++++          22%                ] 97 of 441 complete
[++++++++          22%                ] 98 of 441 complete
[++++++++          22%                ] 99 of 441 complete
[+++++++++         23%                ] 100 of 441 complete
[+++++++++         23%                ] 101 of 441 complete
[+++++++++         23%                ] 102 of 441 complete
[+++++++++         23%                ] 103 of 441 complete
[+++++++++         24%                ] 104 of 441 complete
[+++++++++         24%                ] 105 of 441 complete
[+++++++++         24%                ] 106 of 441 complete
[+++++++++         24%                ] 107 of 441 complete
[+++++++++         24%                ] 108 of 441 complete
[++++++++++        25%                ] 109 of 441 complete
[++++++++++        25%                ] 110 of 441 complete
[++++++++++        25%                ] 111 of 441 complete
[++++++++++        25%                ] 112 of 441 complete
[++++++++++        26%                ] 113 of 441 complete
[++++++++++        26%                ] 114 of 441 complete
[++++++++++        26%                ] 115 of 441 complete
[++++++++++        26%                ] 116 of 441 complete
[++++++++++        27%                ] 117 of 441 complete
[++++++++++        27%                ] 118 of 441 complete
[++++++++++        27%                ] 119 of 441 complete
[++++++++++        27%                ] 120 of 441 complete
[++++++++++        27%                ] 121 of 441 complete
[+++++++++++       28%                ] 122 of 441 complete
[+++++++++++       28%                ] 123 of 441 complete
[+++++++++++       28%                ] 124 of 441 complete
[+++++++++++       28%                ] 125 of 441 complete
[+++++++++++       29%                ] 126 of 441 complete
[+++++++++++       29%                ] 127 of 441 complete
[+++++++++++       29%                ] 128 of 441 complete
[+++++++++++       29%                ] 129 of 441 complete
[+++++++++++       29%                ] 130 of 441 complete
[+++++++++++       30%                ] 131 of 441 complete
[+++++++++++       30%                ] 132 of 441 complete
[+++++++++++       30%                ] 133 of 441 complete
[+++++++++++       30%                ] 134 of 441 complete
[++++++++++++      31%                ] 135 of 441 complete
[++++++++++++      31%                ] 136 of 441 complete
[++++++++++++      31%                ] 137 of 441 complete
```

(continues on next page)

(continued from previous page)

```
[++++++++++++      31%                  ] 138 of 441 complete
[++++++++++++      32%                  ] 139 of 441 complete
[++++++++++++      32%                  ] 140 of 441 complete
[++++++++++++      32%                  ] 141 of 441 complete
[++++++++++++      32%                  ] 142 of 441 complete
[++++++++++++      32%                  ] 143 of 441 complete
[+++++++++++++     33%                  ] 144 of 441 complete
[+++++++++++++     33%                  ] 145 of 441 complete
[+++++++++++++     33%                  ] 146 of 441 complete
[+++++++++++++     33%                  ] 147 of 441 complete
[+++++++++++++     34%                  ] 148 of 441 complete
[+++++++++++++     34%                  ] 149 of 441 complete
[+++++++++++++     34%                  ] 150 of 441 complete
[+++++++++++++     34%                  ] 151 of 441 complete
[+++++++++++++     34%                  ] 152 of 441 complete
[+++++++++++++     35%                  ] 153 of 441 complete
[+++++++++++++     35%                  ] 154 of 441 complete
[+++++++++++++     35%                  ] 155 of 441 complete
[+++++++++++++     35%                  ] 156 of 441 complete
[++++++++++++++    36%                  ] 157 of 441 complete
[++++++++++++++    36%                  ] 158 of 441 complete
[++++++++++++++    36%                  ] 159 of 441 complete
[++++++++++++++    36%                  ] 160 of 441 complete
[++++++++++++++    37%                  ] 161 of 441 complete
[++++++++++++++    37%                  ] 162 of 441 complete
[++++++++++++++    37%                  ] 163 of 441 complete
[++++++++++++++    37%                  ] 164 of 441 complete
[++++++++++++++    37%                  ] 165 of 441 complete
[++++++++++++++    38%                  ] 166 of 441 complete
[++++++++++++++    38%                  ] 167 of 441 complete
[++++++++++++++    38%                  ] 168 of 441 complete
[++++++++++++++    38%                  ] 169 of 441 complete
[+++++++++++++++   39%                  ] 170 of 441 complete
[+++++++++++++++   39%                  ] 171 of 441 complete
[+++++++++++++++   39%                  ] 172 of 441 complete
[+++++++++++++++   39%                  ] 173 of 441 complete
[+++++++++++++++   39%                  ] 174 of 441 complete
[+++++++++++++++   40%                  ] 175 of 441 complete
[+++++++++++++++   40%                  ] 176 of 441 complete
[+++++++++++++++   40%                  ] 177 of 441 complete
[+++++++++++++++   40%                  ] 178 of 441 complete
[++++++++++++++++  41%                  ] 179 of 441 complete
[++++++++++++++++  41%                  ] 180 of 441 complete
[++++++++++++++++  41%                  ] 181 of 441 complete
[++++++++++++++++  41%                  ] 182 of 441 complete
[++++++++++++++++  41%                  ] 183 of 441 complete
[++++++++++++++++  42%                  ] 184 of 441 complete
[++++++++++++++++  42%                  ] 185 of 441 complete
[++++++++++++++++  42%                  ] 186 of 441 complete
```

(continues on next page)

(continued from previous page)

```
[++++++++++++++++  42%                ] 187 of 441 complete
[++++++++++++++++  43%                ] 188 of 441 complete
[++++++++++++++++  43%                ] 189 of 441 complete
[++++++++++++++++  43%                ] 190 of 441 complete
[++++++++++++++++  43%                ] 191 of 441 complete
[++++++++++++++++++ 44%               ] 192 of 441 complete
[++++++++++++++++++ 44%               ] 193 of 441 complete
[++++++++++++++++++ 44%               ] 194 of 441 complete
[++++++++++++++++++ 44%               ] 195 of 441 complete
[++++++++++++++++++ 44%               ] 196 of 441 complete
[++++++++++++++++++ 45%               ] 197 of 441 complete
[++++++++++++++++++ 45%               ] 198 of 441 complete
[++++++++++++++++++ 45%               ] 199 of 441 complete
[++++++++++++++++++ 45%               ] 200 of 441 complete
[++++++++++++++++++ 46%               ] 201 of 441 complete
[++++++++++++++++++ 46%               ] 202 of 441 complete
[++++++++++++++++++ 46%               ] 203 of 441 complete
[++++++++++++++++++ 46%               ] 204 of 441 complete
[++++++++++++++++++ 46%               ] 205 of 441 complete
[++++++++++++++++++ 47%               ] 206 of 441 complete
[++++++++++++++++++ 47%               ] 207 of 441 complete
[++++++++++++++++++ 47%               ] 208 of 441 complete
[++++++++++++++++++ 47%               ] 209 of 441 complete
[++++++++++++++++++ 48%               ] 210 of 441 complete
[++++++++++++++++++ 48%               ] 211 of 441 complete
[++++++++++++++++++ 48%               ] 212 of 441 complete
[++++++++++++++++++ 48%               ] 213 of 441 complete
[++++++++++++++++++ 49%               ] 214 of 441 complete
[++++++++++++++++++ 49%               ] 215 of 441 complete
[++++++++++++++++++ 49%               ] 216 of 441 complete
[++++++++++++++++++ 49%               ] 217 of 441 complete
[++++++++++++++++++ 49%               ] 218 of 441 complete
[++++++++++++++++++ 50%               ] 219 of 441 complete
[++++++++++++++++++ 50%               ] 220 of 441 complete
[++++++++++++++++++ 50%               ] 221 of 441 complete
[++++++++++++++++++ 50%               ] 222 of 441 complete
[++++++++++++++++++ 51%               ] 223 of 441 complete
[++++++++++++++++++ 51%               ] 224 of 441 complete
[++++++++++++++++++ 51%               ] 225 of 441 complete
[++++++++++++++++++ 51%               ] 226 of 441 complete
[++++++++++++++++++ 51%               ] 227 of 441 complete
[++++++++++++++++++ 52%               ] 228 of 441 complete
[++++++++++++++++++ 52%               ] 229 of 441 complete
[++++++++++++++++++ 52%               ] 230 of 441 complete
[++++++++++++++++++ 52%               ] 231 of 441 complete
[++++++++++++++++++ 53%               ] 232 of 441 complete
[++++++++++++++++++ 53%               ] 233 of 441 complete
[++++++++++++++++++ 53%               ] 234 of 441 complete
[++++++++++++++++++ 53%               ] 235 of 441 complete
```

(continues on next page)

(continued from previous page)

```
[++++++++++++++++++ 54%                ] 236 of 441 complete
[++++++++++++++++++ 54%                ] 237 of 441 complete
[++++++++++++++++++ 54%                ] 238 of 441 complete
[++++++++++++++++++ 54%                ] 239 of 441 complete
[++++++++++++++++++ 54%                ] 240 of 441 complete
[++++++++++++++++++ 55%                ] 241 of 441 complete
[++++++++++++++++++ 55%                ] 242 of 441 complete
[++++++++++++++++++ 55%                ] 243 of 441 complete
[++++++++++++++++++ 55%                ] 244 of 441 complete
[++++++++++++++++++ 56%                ] 245 of 441 complete
[++++++++++++++++++ 56%                ] 246 of 441 complete
[++++++++++++++++++ 56%                ] 247 of 441 complete
[++++++++++++++++++ 56%                ] 248 of 441 complete
[++++++++++++++++++ 56%                ] 249 of 441 complete
[++++++++++++++++++ 57%                ] 250 of 441 complete
[++++++++++++++++++ 57%                ] 251 of 441 complete
[++++++++++++++++++ 57%                ] 252 of 441 complete
[++++++++++++++++++ 57%                ] 253 of 441 complete
[++++++++++++++++++ 58%                ] 254 of 441 complete
[++++++++++++++++++ 58%                ] 255 of 441 complete
[++++++++++++++++++ 58%                ] 256 of 441 complete
[++++++++++++++++++ 58%                ] 257 of 441 complete
[++++++++++++++++++ 59%                ] 258 of 441 complete
[++++++++++++++++++ 59%                ] 259 of 441 complete
[++++++++++++++++++ 59%                ] 260 of 441 complete
[++++++++++++++++++ 59%                ] 261 of 441 complete
[++++++++++++++++++ 59%                ] 262 of 441 complete
[++++++++++++++++++ 60% +              ] 263 of 441 complete
[++++++++++++++++++ 60% +              ] 264 of 441 complete
[++++++++++++++++++ 60% +              ] 265 of 441 complete
[++++++++++++++++++ 60% +              ] 266 of 441 complete
[++++++++++++++++++ 61% +              ] 267 of 441 complete
[++++++++++++++++++ 61% +              ] 268 of 441 complete
[++++++++++++++++++ 61% +              ] 269 of 441 complete
[++++++++++++++++++ 61% +              ] 270 of 441 complete
[++++++++++++++++++ 61% +              ] 271 of 441 complete
[++++++++++++++++++ 62% ++             ] 272 of 441 complete
[++++++++++++++++++ 62% ++             ] 273 of 441 complete
[++++++++++++++++++ 62% ++             ] 274 of 441 complete
[++++++++++++++++++ 62% ++             ] 275 of 441 complete
[++++++++++++++++++ 63% ++             ] 276 of 441 complete
[++++++++++++++++++ 63% ++             ] 277 of 441 complete
[++++++++++++++++++ 63% ++             ] 278 of 441 complete
[++++++++++++++++++ 63% ++             ] 279 of 441 complete
[++++++++++++++++++ 63% ++             ] 280 of 441 complete
[++++++++++++++++++ 64% ++             ] 281 of 441 complete
[++++++++++++++++++ 64% ++             ] 282 of 441 complete
[++++++++++++++++++ 64% ++             ] 283 of 441 complete
[++++++++++++++++++ 64% ++             ] 284 of 441 complete
```

(continues on next page)

(continued from previous page)

```
[++++++++++++++++++ 65% +++             ] 285 of 441 complete
[++++++++++++++++++ 65% +++             ] 286 of 441 complete
[++++++++++++++++++ 65% +++             ] 287 of 441 complete
[++++++++++++++++++ 65% +++             ] 288 of 441 complete
[++++++++++++++++++ 66% +++             ] 289 of 441 complete
[++++++++++++++++++ 66% +++             ] 290 of 441 complete
[++++++++++++++++++ 66% +++             ] 291 of 441 complete
[++++++++++++++++++ 66% +++             ] 292 of 441 complete
[++++++++++++++++++ 66% +++             ] 293 of 441 complete
[++++++++++++++++++ 67% +++             ] 294 of 441 complete
[++++++++++++++++++ 67% +++             ] 295 of 441 complete
[++++++++++++++++++ 67% +++             ] 296 of 441 complete
[++++++++++++++++++ 67% +++             ] 297 of 441 complete
[++++++++++++++++++ 68% ++++            ] 298 of 441 complete
[++++++++++++++++++ 68% ++++            ] 299 of 441 complete
[++++++++++++++++++ 68% ++++            ] 300 of 441 complete
[++++++++++++++++++ 68% ++++            ] 301 of 441 complete
[++++++++++++++++++ 68% ++++            ] 302 of 441 complete
[++++++++++++++++++ 69% ++++            ] 303 of 441 complete
[++++++++++++++++++ 69% ++++            ] 304 of 441 complete
[++++++++++++++++++ 69% ++++            ] 305 of 441 complete
[++++++++++++++++++ 69% ++++            ] 306 of 441 complete
[++++++++++++++++++ 70% +++++           ] 307 of 441 complete
[++++++++++++++++++ 70% +++++           ] 308 of 441 complete
[++++++++++++++++++ 70% +++++           ] 309 of 441 complete
[++++++++++++++++++ 70% +++++           ] 310 of 441 complete
[++++++++++++++++++ 71% +++++           ] 311 of 441 complete
[++++++++++++++++++ 71% +++++           ] 312 of 441 complete
[++++++++++++++++++ 71% +++++           ] 313 of 441 complete
[++++++++++++++++++ 71% +++++           ] 314 of 441 complete
[++++++++++++++++++ 71% +++++           ] 315 of 441 complete
[++++++++++++++++++ 72% +++++           ] 316 of 441 complete
[++++++++++++++++++ 72% +++++           ] 317 of 441 complete
[++++++++++++++++++ 72% +++++           ] 318 of 441 complete
[++++++++++++++++++ 72% +++++           ] 319 of 441 complete
[++++++++++++++++++ 73% ++++++          ] 320 of 441 complete
[++++++++++++++++++ 73% ++++++          ] 321 of 441 complete
[++++++++++++++++++ 73% ++++++          ] 322 of 441 complete
[++++++++++++++++++ 73% ++++++          ] 323 of 441 complete
[++++++++++++++++++ 73% ++++++          ] 324 of 441 complete
[++++++++++++++++++ 74% ++++++          ] 325 of 441 complete
[++++++++++++++++++ 74% ++++++          ] 326 of 441 complete
[++++++++++++++++++ 74% ++++++          ] 327 of 441 complete
[++++++++++++++++++ 74% ++++++          ] 328 of 441 complete
[++++++++++++++++++ 75% ++++++          ] 329 of 441 complete
[++++++++++++++++++ 75% ++++++          ] 330 of 441 complete
[++++++++++++++++++ 75% ++++++          ] 331 of 441 complete
[++++++++++++++++++ 75% ++++++          ] 332 of 441 complete
[++++++++++++++++++ 76% +++++++         ] 333 of 441 complete
```

(continues on next page)

(continued from previous page)

```
[++++++++++++++++++ 76% +++++++          ] 334 of 441 complete
[++++++++++++++++++ 76% +++++++          ] 335 of 441 complete
[++++++++++++++++++ 76% +++++++          ] 336 of 441 complete
[++++++++++++++++++ 76% +++++++          ] 337 of 441 complete
[++++++++++++++++++ 77% +++++++          ] 338 of 441 complete
[++++++++++++++++++ 77% +++++++          ] 339 of 441 complete
[++++++++++++++++++ 77% +++++++          ] 340 of 441 complete
[++++++++++++++++++ 77% +++++++          ] 341 of 441 complete
[++++++++++++++++++ 78% ++++++++         ] 342 of 441 complete
[++++++++++++++++++ 78% ++++++++         ] 343 of 441 complete
[++++++++++++++++++ 78% ++++++++         ] 344 of 441 complete
[++++++++++++++++++ 78% ++++++++         ] 345 of 441 complete
[++++++++++++++++++ 78% ++++++++         ] 346 of 441 complete
[++++++++++++++++++ 79% ++++++++         ] 347 of 441 complete
[++++++++++++++++++ 79% ++++++++         ] 348 of 441 complete
[++++++++++++++++++ 79% ++++++++         ] 349 of 441 complete
[++++++++++++++++++ 79% ++++++++         ] 350 of 441 complete
[++++++++++++++++++ 80% ++++++++         ] 351 of 441 complete
[++++++++++++++++++ 80% ++++++++         ] 352 of 441 complete
[++++++++++++++++++ 80% ++++++++         ] 353 of 441 complete
[++++++++++++++++++ 80% ++++++++         ] 354 of 441 complete
[++++++++++++++++++ 80% ++++++++         ] 355 of 441 complete
[++++++++++++++++++ 81% +++++++++        ] 356 of 441 complete
[++++++++++++++++++ 81% +++++++++        ] 357 of 441 complete
[++++++++++++++++++ 81% +++++++++        ] 358 of 441 complete
[++++++++++++++++++ 81% +++++++++        ] 359 of 441 complete
[++++++++++++++++++ 82% +++++++++        ] 360 of 441 complete
[++++++++++++++++++ 82% +++++++++        ] 361 of 441 complete
[++++++++++++++++++ 82% +++++++++        ] 362 of 441 complete
[++++++++++++++++++ 82% +++++++++        ] 363 of 441 complete
[++++++++++++++++++ 83% ++++++++++       ] 364 of 441 complete
[++++++++++++++++++ 83% ++++++++++       ] 365 of 441 complete
[++++++++++++++++++ 83% ++++++++++       ] 366 of 441 complete
[++++++++++++++++++ 83% ++++++++++       ] 367 of 441 complete
[++++++++++++++++++ 83% ++++++++++       ] 368 of 441 complete
[++++++++++++++++++ 84% ++++++++++       ] 369 of 441 complete
[++++++++++++++++++ 84% ++++++++++       ] 370 of 441 complete
[++++++++++++++++++ 84% ++++++++++       ] 371 of 441 complete
[++++++++++++++++++ 84% ++++++++++       ] 372 of 441 complete
[++++++++++++++++++ 85% ++++++++++       ] 373 of 441 complete
[++++++++++++++++++ 85% ++++++++++       ] 374 of 441 complete
[++++++++++++++++++ 85% ++++++++++       ] 375 of 441 complete
[++++++++++++++++++ 85% ++++++++++       ] 376 of 441 complete
[++++++++++++++++++ 85% ++++++++++       ] 377 of 441 complete
[++++++++++++++++++ 86% +++++++++++      ] 378 of 441 complete
[++++++++++++++++++ 86% +++++++++++      ] 379 of 441 complete
[++++++++++++++++++ 86% +++++++++++      ] 380 of 441 complete
[++++++++++++++++++ 86% +++++++++++      ] 381 of 441 complete
[++++++++++++++++++ 87% +++++++++++      ] 382 of 441 complete
```

(continues on next page)

(continued from previous page)

```
[++++++++++++++++++ 87% ++++++++++     ] 383 of 441 complete
[++++++++++++++++++ 87% ++++++++++     ] 384 of 441 complete
[++++++++++++++++++ 87% ++++++++++     ] 385 of 441 complete
[++++++++++++++++++ 88% ++++++++++     ] 386 of 441 complete
[++++++++++++++++++ 88% ++++++++++     ] 387 of 441 complete
[++++++++++++++++++ 88% ++++++++++     ] 388 of 441 complete
[++++++++++++++++++ 88% ++++++++++     ] 389 of 441 complete
[++++++++++++++++++ 88% ++++++++++     ] 390 of 441 complete
[++++++++++++++++++ 89% ++++++++++    ] 391 of 441 complete
[++++++++++++++++++ 89% ++++++++++    ] 392 of 441 complete
[++++++++++++++++++ 89% ++++++++++    ] 393 of 441 complete
[++++++++++++++++++ 89% ++++++++++    ] 394 of 441 complete
[++++++++++++++++++ 90% ++++++++++    ] 395 of 441 complete
[++++++++++++++++++ 90% ++++++++++    ] 396 of 441 complete
[++++++++++++++++++ 90% ++++++++++    ] 397 of 441 complete
[++++++++++++++++++ 90% ++++++++++    ] 398 of 441 complete
[++++++++++++++++++ 90% ++++++++++    ] 399 of 441 complete
[++++++++++++++++++ 91% ++++++++++++   ] 400 of 441 complete
[++++++++++++++++++ 91% ++++++++++++   ] 401 of 441 complete
[++++++++++++++++++ 91% ++++++++++++   ] 402 of 441 complete
[++++++++++++++++++ 91% ++++++++++++   ] 403 of 441 complete
[++++++++++++++++++ 92% ++++++++++++   ] 404 of 441 complete
[++++++++++++++++++ 92% ++++++++++++   ] 405 of 441 complete
[++++++++++++++++++ 92% ++++++++++++   ] 406 of 441 complete
[++++++++++++++++++ 92% ++++++++++++   ] 407 of 441 complete
[++++++++++++++++++ 93% ++++++++++++   ] 408 of 441 complete
[++++++++++++++++++ 93% ++++++++++++   ] 409 of 441 complete
[++++++++++++++++++ 93% ++++++++++++   ] 410 of 441 complete
[++++++++++++++++++ 93% ++++++++++++   ] 411 of 441 complete
[++++++++++++++++++ 93% ++++++++++++   ] 412 of 441 complete
[++++++++++++++++++ 94% ++++++++++++++  ] 413 of 441 complete
[++++++++++++++++++ 94% ++++++++++++++  ] 414 of 441 complete
[++++++++++++++++++ 94% ++++++++++++++  ] 415 of 441 complete
[++++++++++++++++++ 94% ++++++++++++++  ] 416 of 441 complete
[++++++++++++++++++ 95% ++++++++++++++  ] 417 of 441 complete
[++++++++++++++++++ 95% ++++++++++++++  ] 418 of 441 complete
[++++++++++++++++++ 95% ++++++++++++++  ] 419 of 441 complete
[++++++++++++++++++ 95% ++++++++++++++  ] 420 of 441 complete
[++++++++++++++++++ 95% ++++++++++++++  ] 421 of 441 complete
[++++++++++++++++++ 96% ++++++++++++++  ] 422 of 441 complete
[++++++++++++++++++ 96% ++++++++++++++  ] 423 of 441 complete
[++++++++++++++++++ 96% ++++++++++++++  ] 424 of 441 complete
[++++++++++++++++++ 96% ++++++++++++++  ] 425 of 441 complete
[++++++++++++++++++ 97% ++++++++++++++++ ] 426 of 441 complete
[++++++++++++++++++ 97% ++++++++++++++++ ] 427 of 441 complete
[++++++++++++++++++ 97% ++++++++++++++++ ] 428 of 441 complete
[++++++++++++++++++ 97% ++++++++++++++++ ] 429 of 441 complete
[++++++++++++++++++ 98% ++++++++++++++++ ] 430 of 441 complete
[++++++++++++++++++ 98% ++++++++++++++++ ] 431 of 441 complete
```

(continues on next page)

(continued from previous page)

```
[+++++++++++++++++ 98% +++++++++++++++ ] 432 of 441 complete
[+++++++++++++++++ 98% +++++++++++++++ ] 433 of 441 complete
[+++++++++++++++++ 98% +++++++++++++++ ] 434 of 441 complete
[+++++++++++++++++ 99% ++++++++++++++++] 435 of 441 complete
[+++++++++++++++++ 99% ++++++++++++++++] 436 of 441 complete
[+++++++++++++++++ 99% ++++++++++++++++] 437 of 441 complete
[+++++++++++++++++ 99% ++++++++++++++++] 438 of 441 complete
[++++++++++++++++ 100% ++++++++++++++++] 439 of 441 complete
[++++++++++++++++ 100% ++++++++++++++++] 440 of 441 complete
[++++++++++++++++ 100% ++++++++++++++++] 441 of 441 complete
```

Just like in the paper, the data are contaminated with an error model consisting of relative and absolute noise of 2% and 1 nT, respectively.

```
err = 0.01
noise_level = 1e-9
relError = noise_level / np.abs(data) + err
data *= np.random.randn(*data.shape)*relError + 1.0
```

Depth weighting

In the paper of Li & Oldenburg (1996), they propose a depth weighting of the constraints with the formula

$$w_z = \frac{1}{(z+z_0)^{3/2}}$$

```
# depth weighting
bz = np.array([b.center().z() for b in grid.boundaries() if not b.outside()])
z0 = 25
wz = 10 / (bz+z0)**1.5
fop.region(0).setConstraintWeights(wz)
```

Inversion

The inversion is rather straightforward using the standard inversion framework `pygimli.Inversion`.

```
inv = pg.Inversion(fop=fop, verbose=True)  # , debug=True)
inv.setRegularization(limits=[0, 0.1])  # to limit values
startModel = pg.Vector(grid.cellCount(), 0.001)
invmodel = inv.run(data, relError, lam=10., startModel=1e-3, verbose=True)
grid["inv"] = invmodel
```

```
fop: <pygimli.physics.gravimetry.MagneticsModelling.MagneticsModelling object␣
↪at 0x7fe8fa2e4180>
Data transformation: <pygimli.core._pygimli_.RTrans object at 0x7fe8f5837be0>
Model transformation (cumulative):
```

(continues on next page)

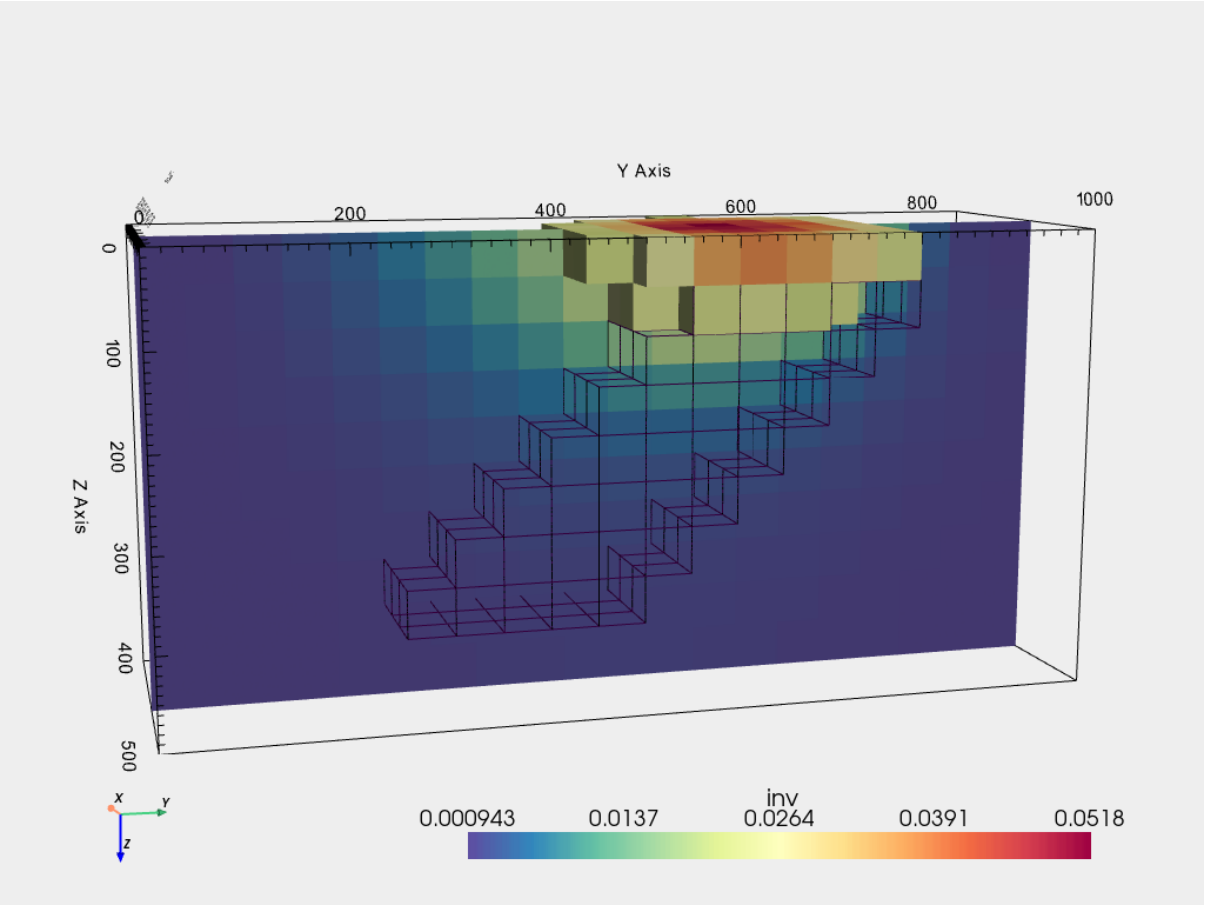
(continued from previous page)

```
         0 <pygimli.core._pygimli_.RTransLogLU object at 0x7fe8f5837be0>
min/max (data): -7.8e-08/5.6e-07
min/max (error): 1.18%/1527%
min/max (start model): 1.0e-03/1.0e-03
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
inv.iter 2 ... chi² = 293.0 (dPhi = 58.54%) lam: 10.0
--------------------------------------------------------------------------------
inv.iter 3 ... chi² = 8.54 (dPhi = 93.72%) lam: 10.0
--------------------------------------------------------------------------------
inv.iter 4 ... chi² = 1.3 (dPhi = 35.12%) lam: 10.0
--------------------------------------------------------------------------------
inv.iter 5 ... chi² = 1.13 (dPhi = 0.84%) lam: 10.0
################################################################################
#                 Abort criteria reached: dPhi = 0.84 (< 2.0%)                 #
################################################################################
```

Visualization

For showing the model, we again use the threshold filter with a value of 0.02. For comparison with the synthetic model, we plot the latter as a wireframe.

```
pl, _ = pg.show(grid, label="synth", style="wireframe", hold=True,
                filter={"threshold": dict(value=0.025, scalars="synth")})
pv.drawMesh(pl, grid, label="inv", style="surface", cMap="Spectral_r",
            filter={"threshold": dict(value=0.02, scalars="inv")})
pv.drawMesh(pl, grid, label="inv", style="surface", cMap="Spectral_r",
            filter={"slice": dict(normal=[-1, 0, 0], origin=[500, 600, 250])})
pl.camera_position = "yz"
pl.camera.roll = 90
pl.camera.azimuth = 180 - 15
pl.camera.elevation = 10
pl.camera.zoom(1.2)
pl.show()
```

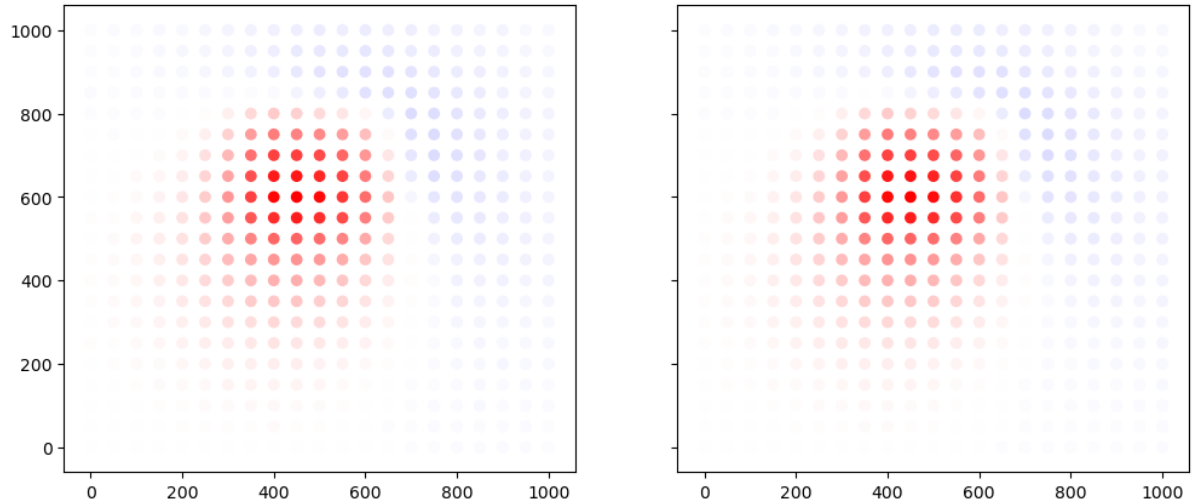



```
False
```

The model can nicely outline the top part of the anomalous body, but not its depth extent.

We compare the data and model response by means of scatter plots:

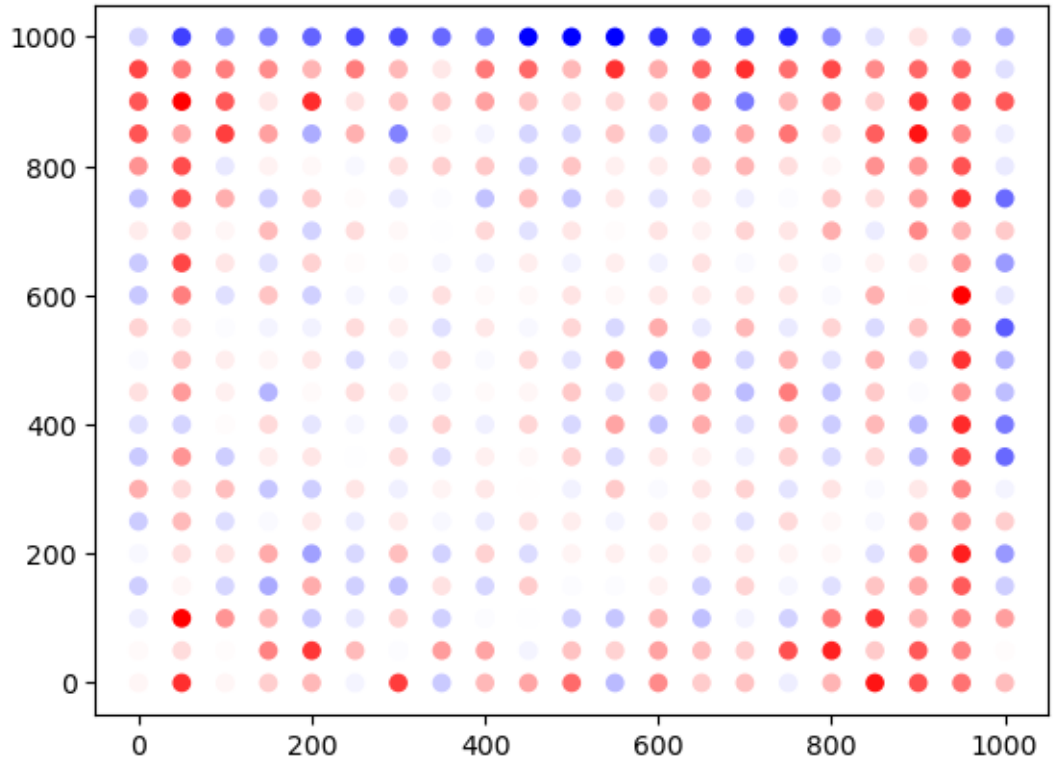
```
fig, ax = pg.plt.subplots(ncols=2, figsize=(12, 5), sharex=True, sharey=True)
vals = data * 1e9
mm = np.max(np.abs(vals))
ax[0].scatter(px, py, c=vals, cmap="bwr", vmin=-mm, vmax=mm);
ax[1].scatter(px, py, c=inv.response*1e9, cmap="bwr", vmin=-mm, vmax=mm);
```



```
<matplotlib.collections.PathCollection object at 0x7fe8f546c460>
```

Alternatively, we can also plot the error-weighted misfit.

```
misfit = (inv.response*1e9-vals) / (relError * np.abs(data) * 1e9)
pg.plt.scatter(py, px, c=misfit, cmap="bwr", vmin=-3, vmax=3);
```




```
<matplotlib.collections.PathCollection object at 0x7fe89afe3910>
```

References

- Li, Y. & Oldenburg, D. (1996): 3-D inversion of magnetic data. Geophysics 61(2), 394-408.
- Holstein, H., Sherratt, E.M., Reid, A.B. (2007): Gravimagnetic field tensor gradiometry formulas for uniform polyhedra, SEG Ext. Abstr.

Total running time of the script: (0 minutes 15.978 seconds)

6.7.6 Miscellaneous

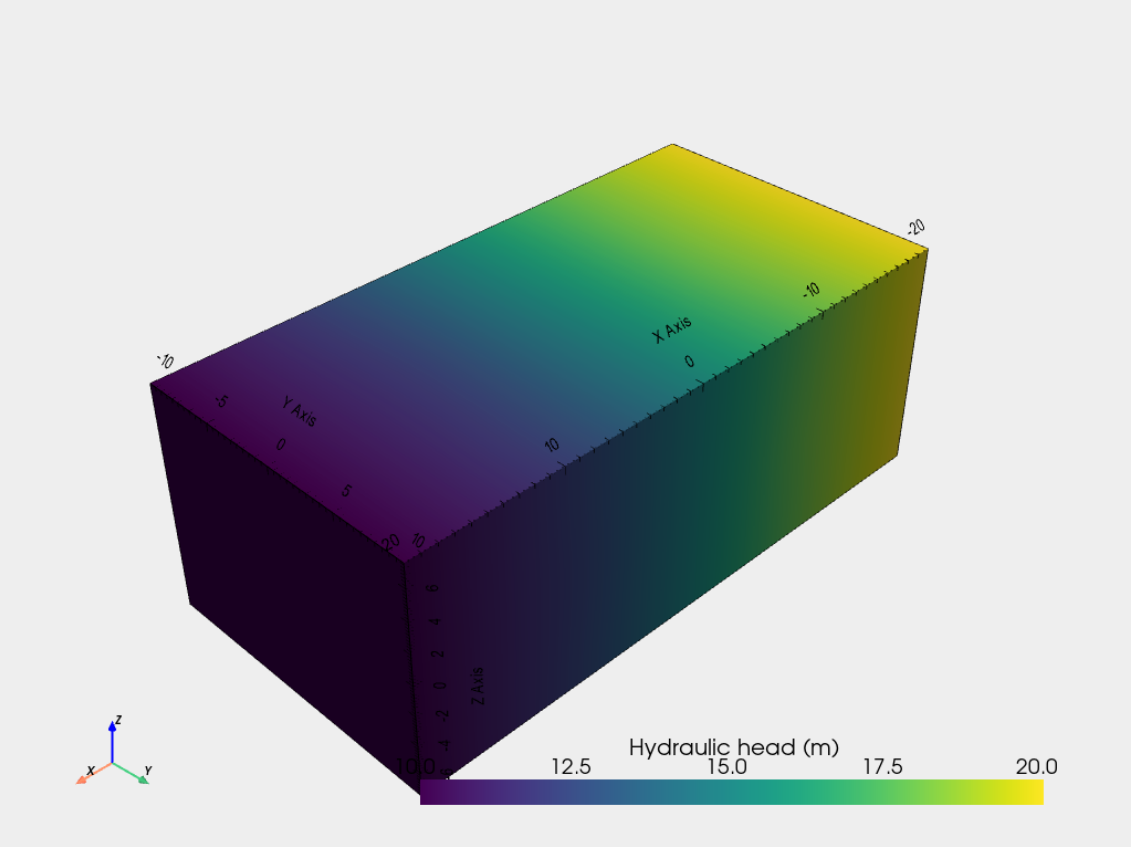
Interdisciplinary and non-geophysical (e.g. fluid flow) examples.

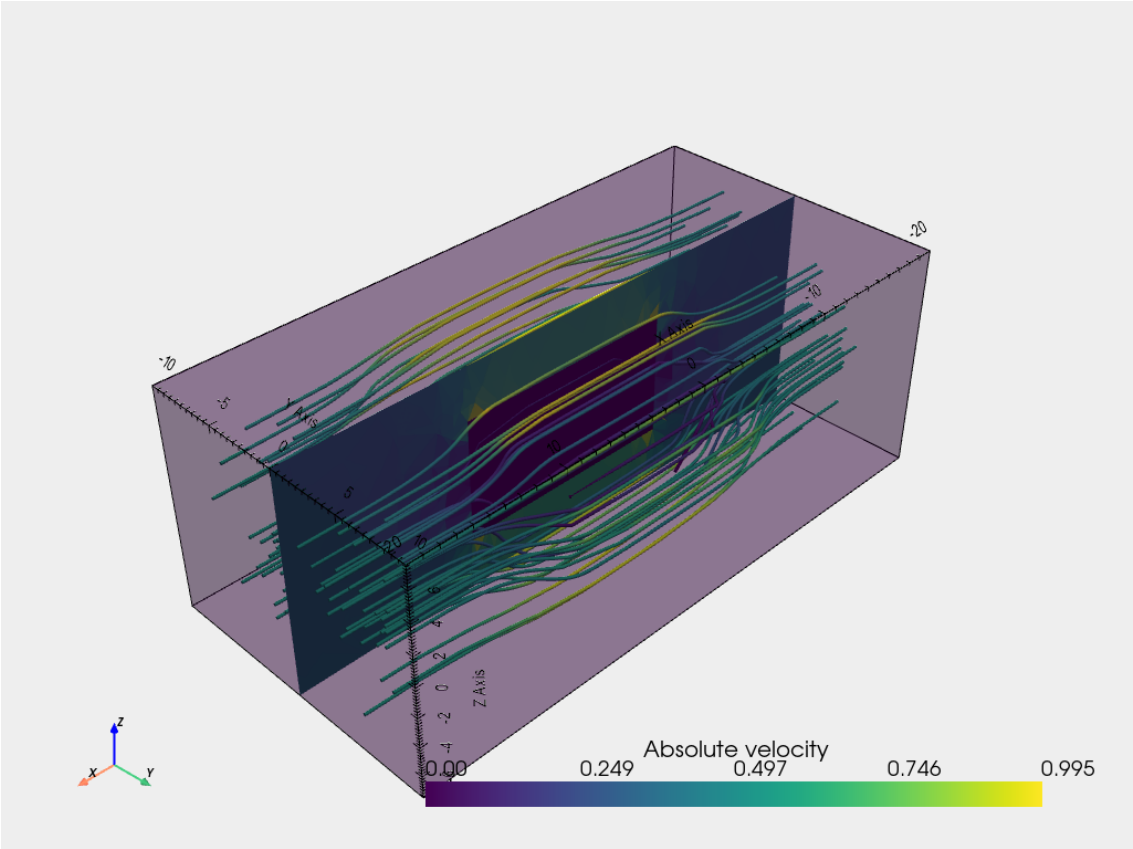
6.7.6.1 3D Darcy flow

Here we illustrate Darcy flow in a heterogeneous 3D body. We use the general *pygimli.solver.solveFiniteElements()* (page 450) to solve Darcy's law:

$$\nabla \cdot (K \nabla p) = 0$$

The sought hydraulic velocity distribution can then be calculated as the gradient field of $\mathbf{v} = -\nabla p$.

- 




```
<pyvista.plotting.plotting.Plotter object at 0x7fe8aae7e970>
```

```
import numpy as np

import pygimli as pg
import pygimli.meshtools as mt
from pygimli.viewer.pv import drawStreamLines, drawSlice

plc = mt.createCube(size=[40, 20, 15], marker=1, boundaryMarker=0)
cube = mt.createCube(size=[15, 15, 8], marker=2, boundaryMarker=0)
geom = plc + cube

mesh = mt.createMesh(geom, area=4)

for bound in mesh.boundaries():
    x = bound.center().x()
    if x == mesh.xmin():
        bound.setMarker(1)
    elif x == mesh.xmax():
        bound.setMarker(2)
```

(continues on next page)

(continued from previous page)

```
kMap = {1: 1e-4, 2: 1e-6}
kArray = pg.solver.parseMapToCellArray(list(kMap), mesh) # dict does not work
kArray = np.column_stack([kArray] * 3)

bc = {"Dirichlet": {1: 20.0, 2: 10.0}}


h = pg.solver.solveFiniteElements(mesh, kMap, bc=bc)
vel = -pg.solver.grad(mesh, h) * kArray

pg.show(mesh, h, label="Hydraulic head (m)")

ax, _ = pg.show(mesh, alpha=0.3, hold=True, colorBar=False)
drawStreamLines(ax, mesh, vel, radius=.1, source_radius=10)
drawSlice(ax, mesh, normal=[0,1,0], data=pg.abs(vel), label="Absolute velocity")
# ax.show()
```

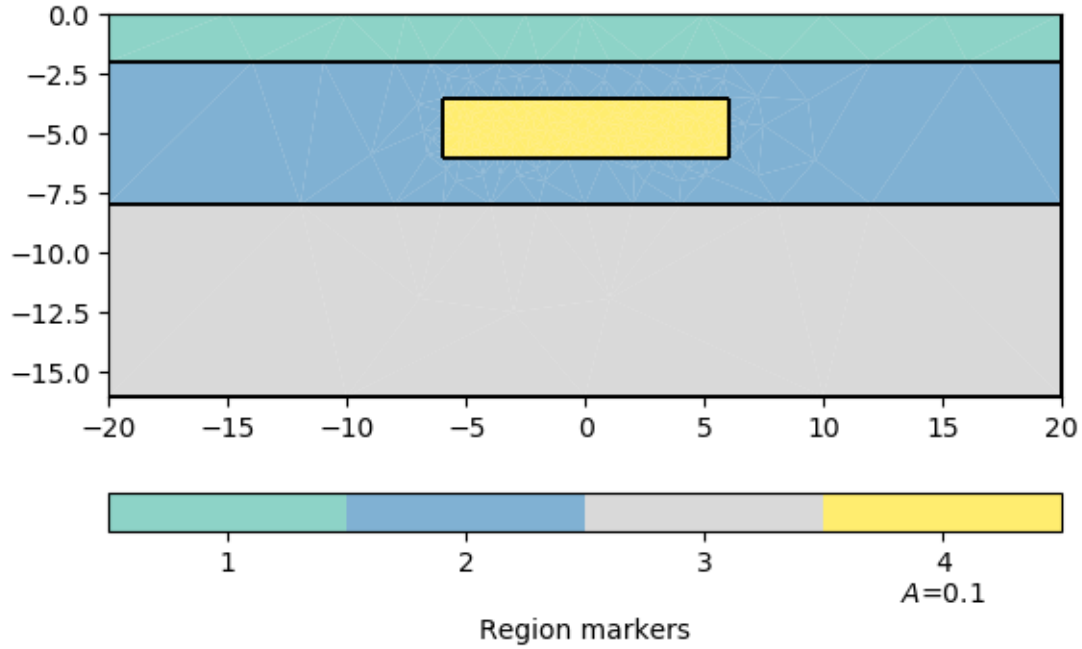
6.7.6.2 Hydrogeophysical modeling

Coupled hydrogeophysical modeling example. This essentially represents the forward modeling step of the example presented in section 3.2 of the pyGIMLi paper.

```
import numpy as np

import pygimli as pg
import pygimli.meshtools as mt
import pygimli.physics.ert as ert
import pygimli.physics.petro as petro
from pygimli.physics import ERTManager
```

Create geometry definition for the modeling domain

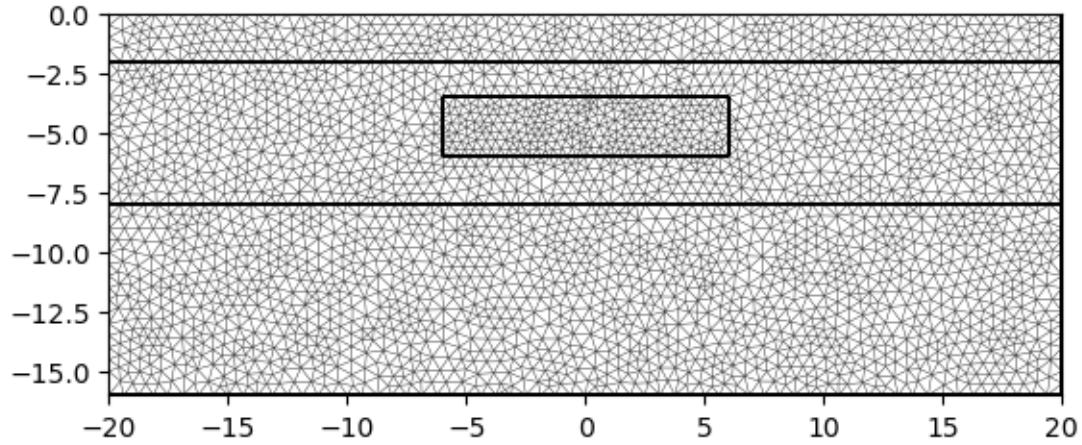
```
world = mt.createWorld(start=[-20, 0], end=[20, -16], layers=[-2, -8],
                       worldMarker=False)

# Create a heterogeneous block
block = mt.createRectangle(start=[-6, -3.5], end=[6, -6.0], marker=4,
                           boundaryMarker=10, area=0.1)

# Merge geometrical entities
geom = world + block
pg.show(geom, boundaryMarker=True)
```




```
(<matplotlib.axes._subplots.AxesSubplot object at 0x7fe8f546ca90>, None)
```

Create a mesh from the geometry definition

```
mesh = mt.createMesh(geom, quality=32, area=0.2, smooth=[1, 10])
pg.show(mesh)
```



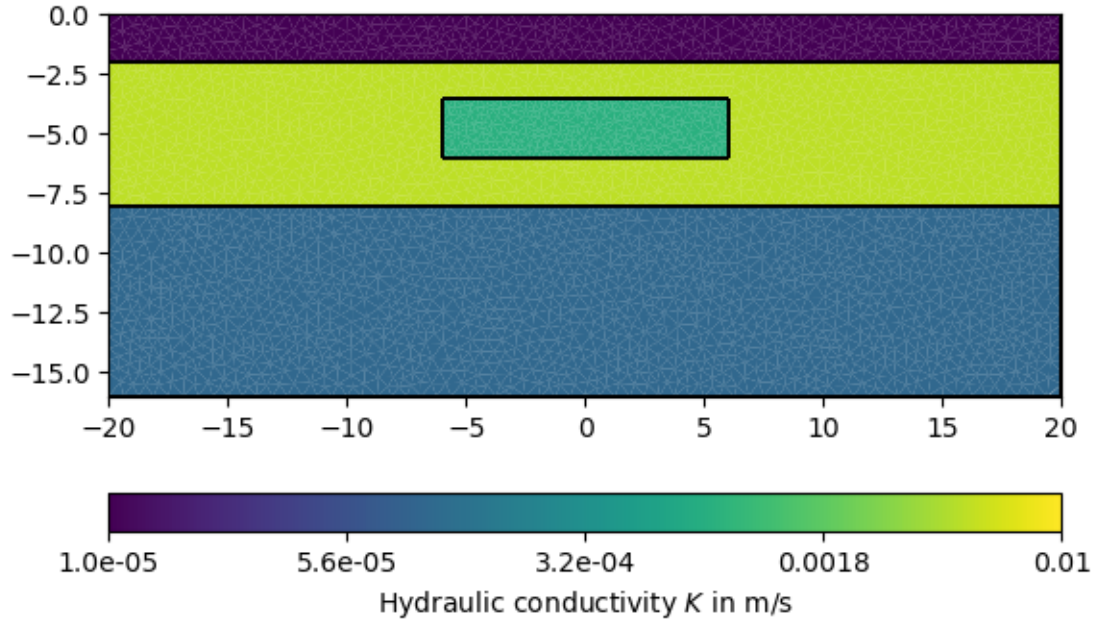

```
(<matplotlib.axes._subplots.AxesSubplot object at 0x7fe8f582a250>, None)
```

Fluid flow in a porous medium of slow non-viscous and non-frictional hydraulic movement is governed by Darcy's Law according to:

$$K^{-1}\mathbf{v} + \nabla p = 0$$
$$\nabla \cdot \mathbf{v} = 0$$
$$\text{leading to } \nabla \cdot (K \nabla p) = 0 \quad \text{on} \quad \Omega$$

We begin by defining isotropic values of hydraulic conductivity K and mapping these to each mesh cell:

```
# Map regions to hydraulic conductivity in $m/s$
kMap = [[1, 1e-8], [2, 5e-3], [3, 1e-4], [4, 8e-4]]

# Map conductivity value per region to each cell in the given mesh
K = pg.solver.parseMapToCellArray(kMap, mesh)

pg.show(mesh, data=K, label='Hydraulic conductivity $K$ in m$/$s', cMin=1e-5,
        cMax=1e-2, logScale=True, grid=True)
```




```
(<matplotlib.axes._subplots.AxesSubplot object at 0x7fe89adaf9a0>, <matplotlib.
↪colorbar.Colorbar object at 0x7fe8fa420c70>)
```

The problem further boundary conditions of the hydraulic potential. We use $p = p_0 = 0.75$ m on the left and $p = 0$ on the right boundary of the modelling domain, equaling a hydraulic gradient of 1.75%.

```
# Dirichlet conditions for hydraulic potential
left = 0.75
right = 0.0
pBound = {"Dirichlet": {1: left, 2: left, 3: left, 5: right, 6: right, 7: right}}
```

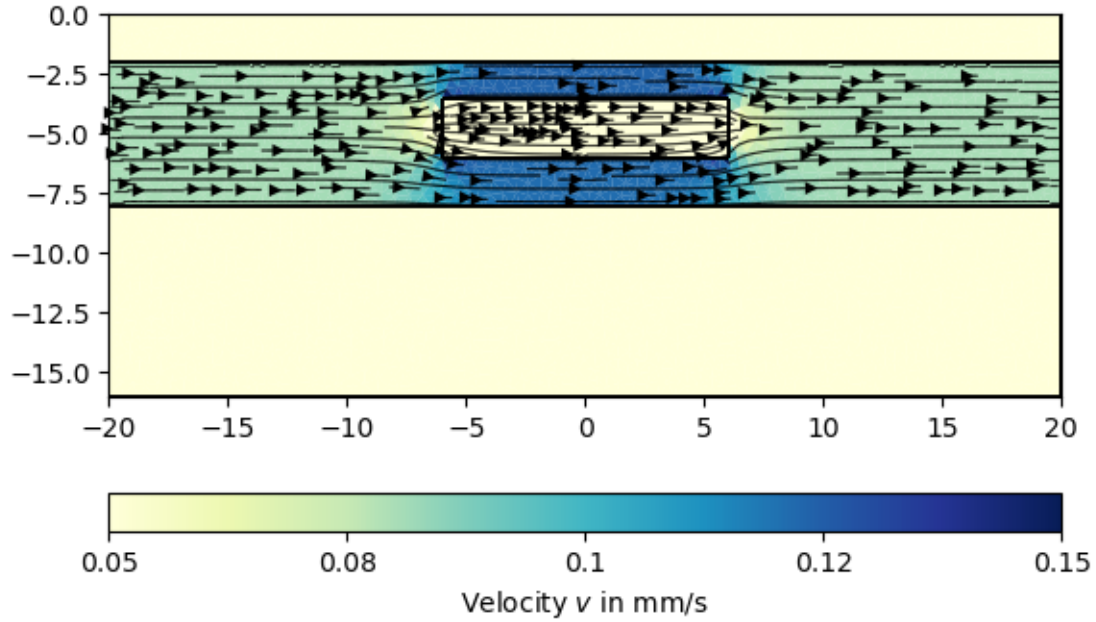
We can now call the finite element solver with the generated mesh, hydraulic conductivity and the boundary condition. The sought hydraulic velocity distribution can then be calculated as the gradient field of $\mathbf{v} = -\nabla p$ and visualized using the generic *pg.show()* function.

```
# Solve for hydraulic potential
p = pg.solver.solveFiniteElements(mesh, a=K, bc=pBound)

# Solve velocity as gradient of hydraulic potential
vel = -pg.solver.grad(mesh, p) * np.asarray([K, K, K]).T

ax, _ = pg.show(mesh, data=pg.abs(vel) * 1000, cMin=0.05, cMax=0.15,
```

(continues on next page)

(continued from previous page)

```
                label='Velocity $v$ in mm$/$s', cMap='YlGnBu', hold=True)
ax, _ = pg.show(mesh, data=vel, ax=ax, color='k', linewidth=0.8, dropTol=1e-5,
                hold=True)
```



In the next step, we use this velocity field to simulate the dynamic movement of a particle (e.g., salt) concentration $c(\mathbf{r},t)$ in the aquifer by using the advection-diffusion equation:

$$\frac{\partial c}{\partial t} = \underbrace{\nabla \cdot (D \nabla c)}_{\text{Diffusion / Dispersion}} \underbrace{- \nabla \cdot (\mathbf{v} \nabla c)}_{\text{Advection}} + S$$

```
S = pg.Vector(mesh.cellCount(), 0.0)

# Fill injection source vector for a fixed injection position
sourceCell = mesh.findCell([-19.1, -4.6])
S[sourceCell.id()] = 1.0 / sourceCell.size()  # g/(l s)
```

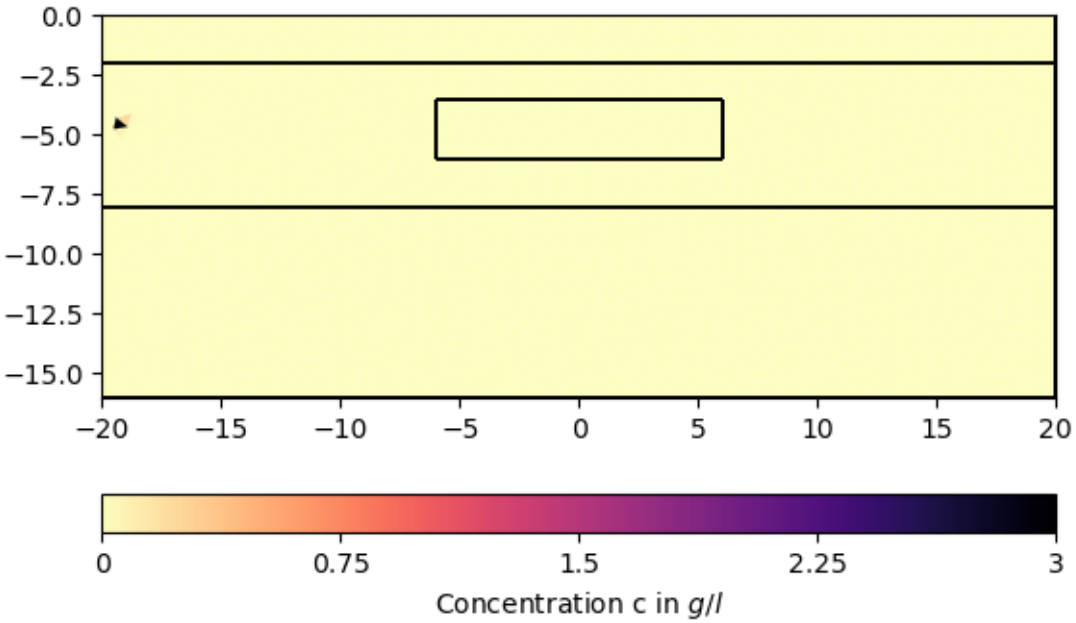
We define a time vector and common velocity-depending dispersion coefficient $D = \alpha|\mathbf{v}|$ with a dispersivity $\alpha = 1 \cdot 10^{-2}$ m. We solve the advection-diffsuion equation on the equation level with the finite volume solver, which results in a particle concentration :math:`c(mathbf{r},t)$ (in g/l) for each cell center and time step.

```
# Choose 800 time steps for 6 days in seconds
t = pg.utils.grange(0, 6 * 24 * 3600, n=800)

# Create dispersitivity, depending on the absolute velocity
dispersion = pg.abs(vel) * 1e-2

# Solve for injection time, but we need velocities on cell nodes
veln = mt.cellDataToNodeData(mesh, vel)
c1 = pg.solver.solveFiniteVolume(mesh, a=dispersion, f=S, vel=veln, times=t,
```

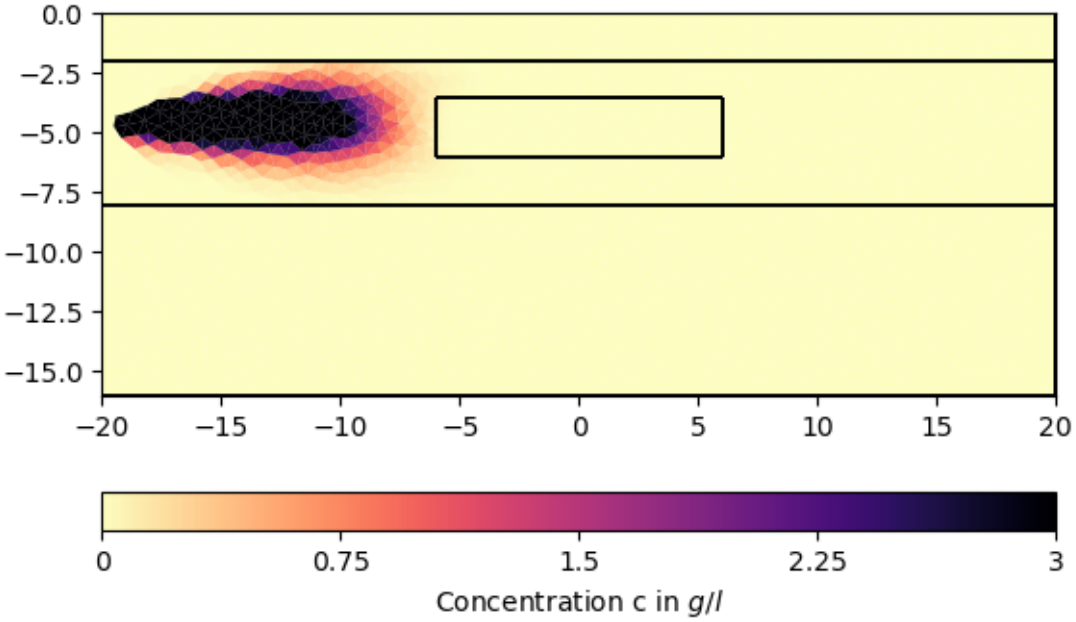
(continues on next page)

(continued from previous page)

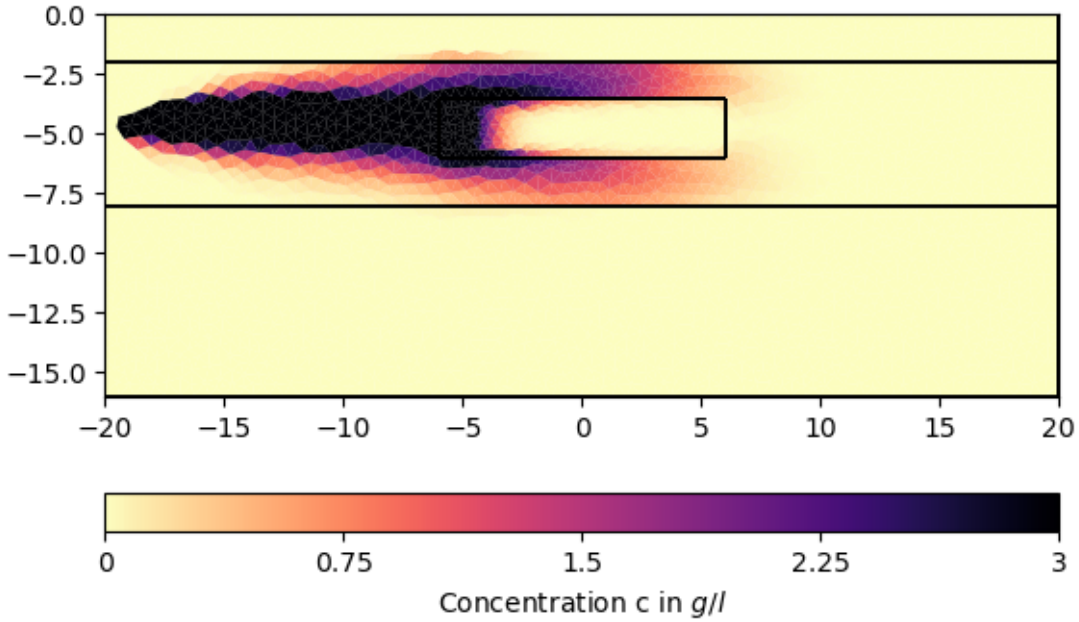
```
                                scheme='PS', verbose=0)

# Solve without injection starting with last result
c2 = pg.solver.solveFiniteVolume(mesh, a=dispersion, f=0, vel=veln, u0=c1[-1],
                                 times=t, scheme='PS', verbose=0)
# Stack results together
c = np.vstack((c1, c2))

# We can now visualize the result:
for ci in c[1:][::200]:
    pg.show(mesh, data=ci * 0.001, cMin=0, cMax=3, cMap="magma_r",
            label="Concentration c in $g/l$")
```

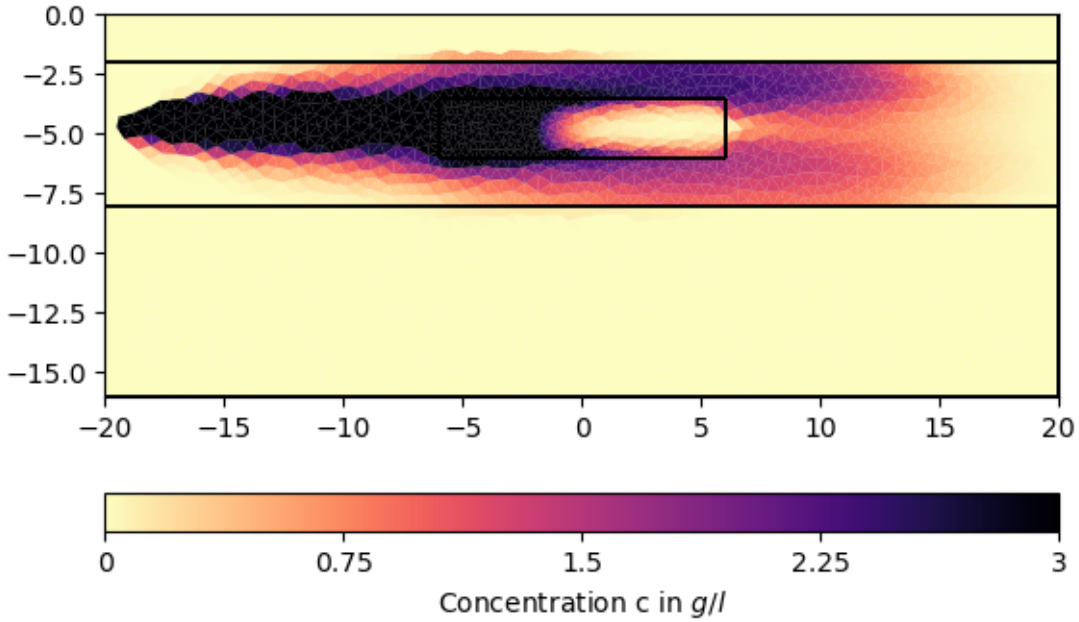


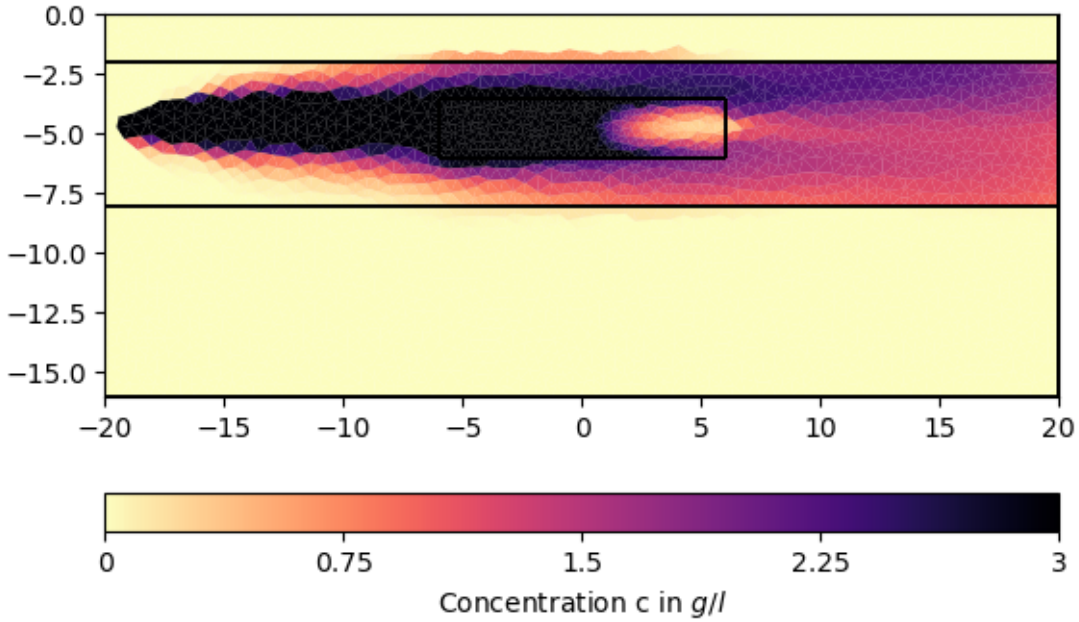
-

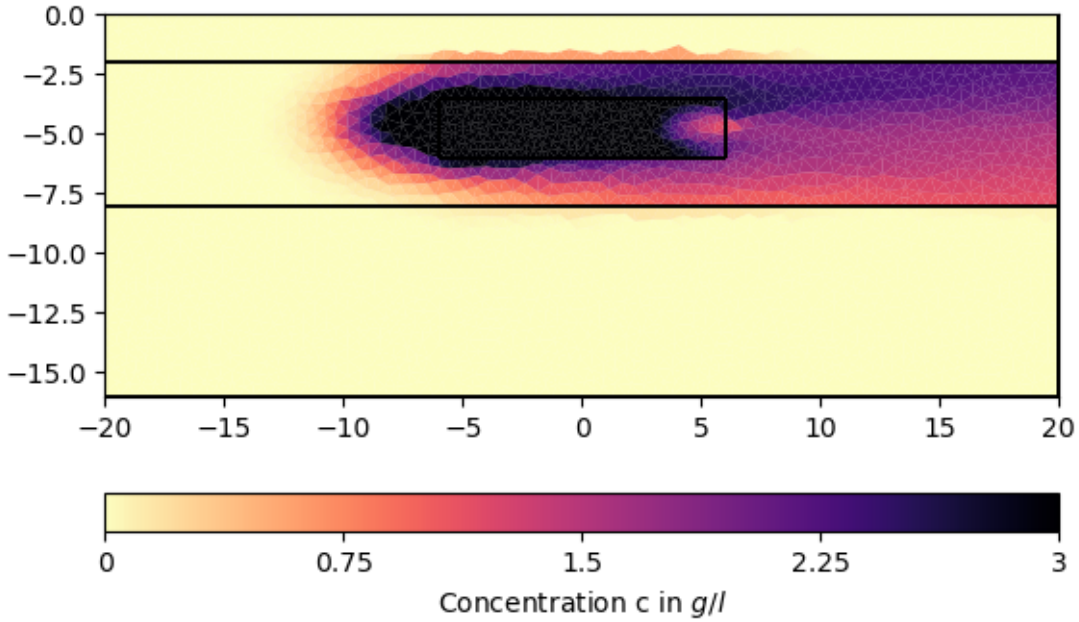


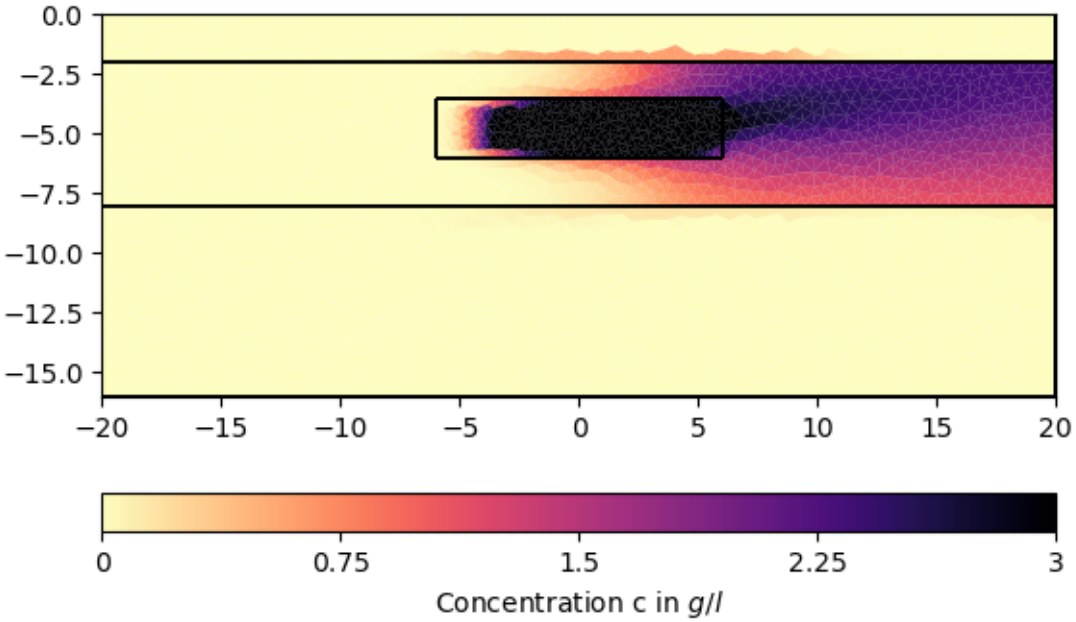
-

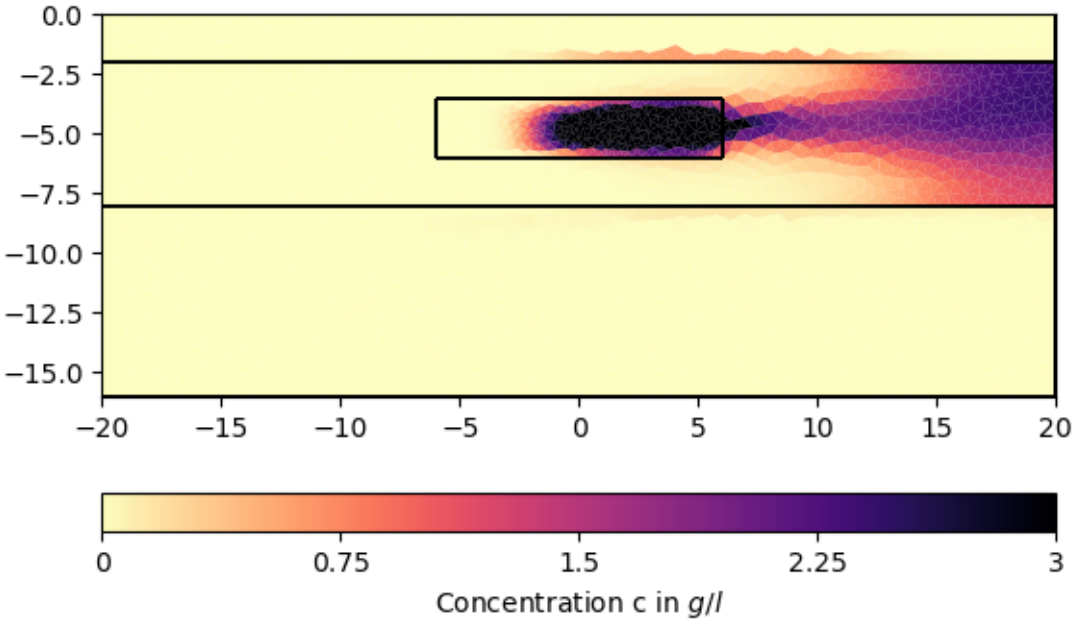
- 
0.0
−2.5
−5.0
−7.5
−10.0
−12.5
−15.0
−20
−15
−10
−5
0
5
10
15
20
0
0.75
1.5
2.25
3
Concentration c in g/l

- 
0.0
−2.5
−5.0
−7.5
−10.0
−12.5
−15.0
−20
−15
−10
−5
0
5
10
15
20
0
0.75
1.5
2.25
3
Concentration c in g/l

- 
0.0
−2.5
−5.0
−7.5
−10.0
−12.5
−15.0
−20
−15
−10
−5
0
5
10
15
20
0
0.75
1.5
2.25
3
Concentration c in g/l

- 
0.0
−2.5
−5.0
−7.5
−10.0
−12.5
−15.0
−20
−15
−10
−5
0
5
10
15
20
0
0.75
1.5
2.25
3
Concentration c in g/l



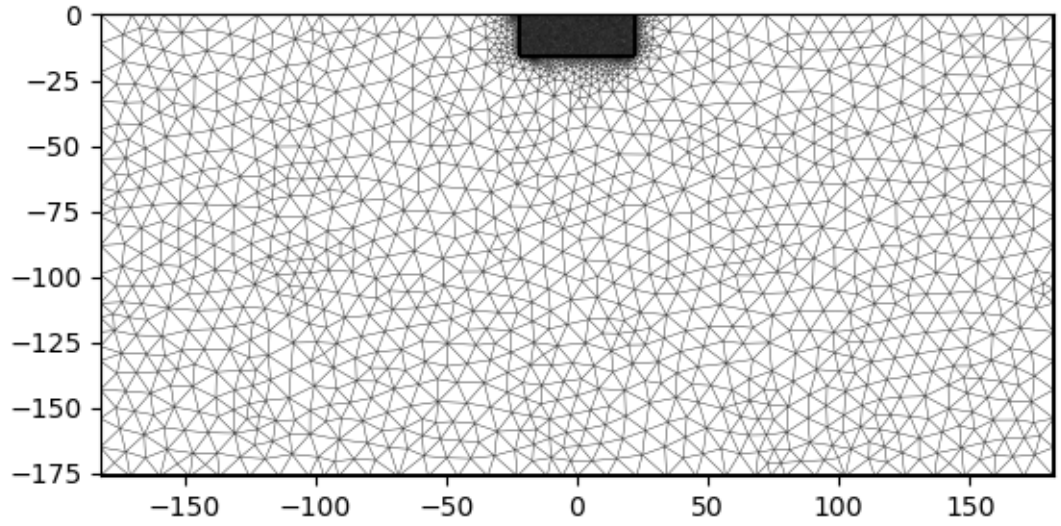
- 


```
/usr/lib/python3/dist-packages/scipy/sparse/linalg/dsolve/linsolve.py:296:␣
↪SparseEfficiencyWarning: splu requires CSC matrix format
  warn('splu requires CSC matrix format', SparseEfficiencyWarning)
```

Simulate time-lapse electrical resistivity measurements.

Create a dipole-dipole measurement scheme and a suitable mesh for ERT forward simulations.

```
ertScheme = ert.createERTData(pg.utils.grange(-20, 20, dx=1.0), schemeName='dd')

meshERT = mt.createParaMesh(ertScheme, quality=33, paraMaxCellSize=0.2,
                            boundaryMaxCellSize=50, smooth=[1, 2])
pg.show(meshERT)
```



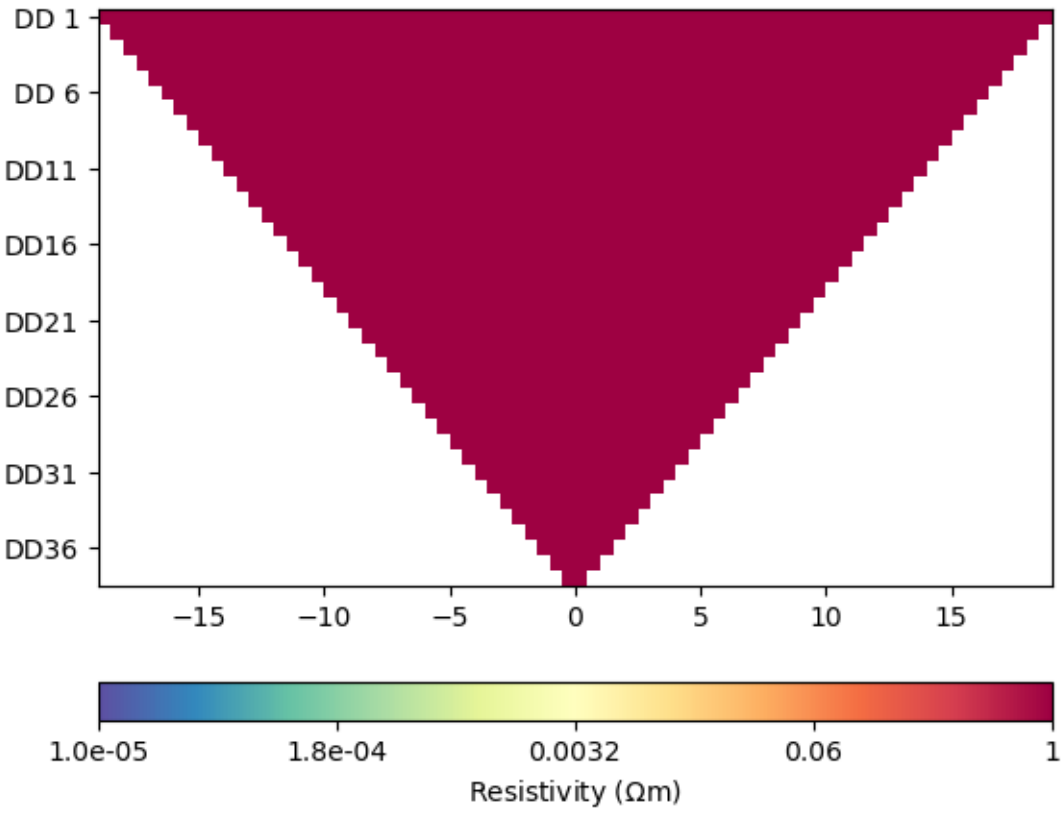

```
(<matplotlib.axes._subplots.AxesSubplot object at 0x7fe8994b2d90>, None)
```

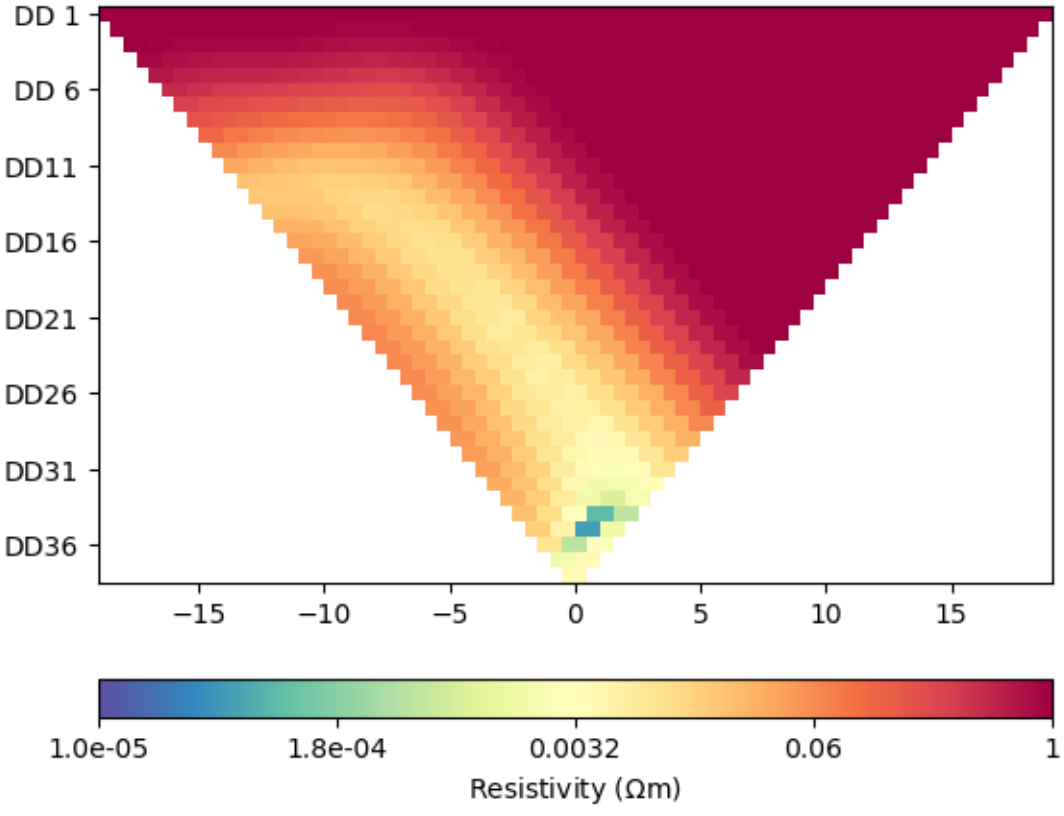
Use simulated concentrations to calculate bulk resistivity using Archie's Law and fill matrix with apparent resistivity ratios with respect to a background model:

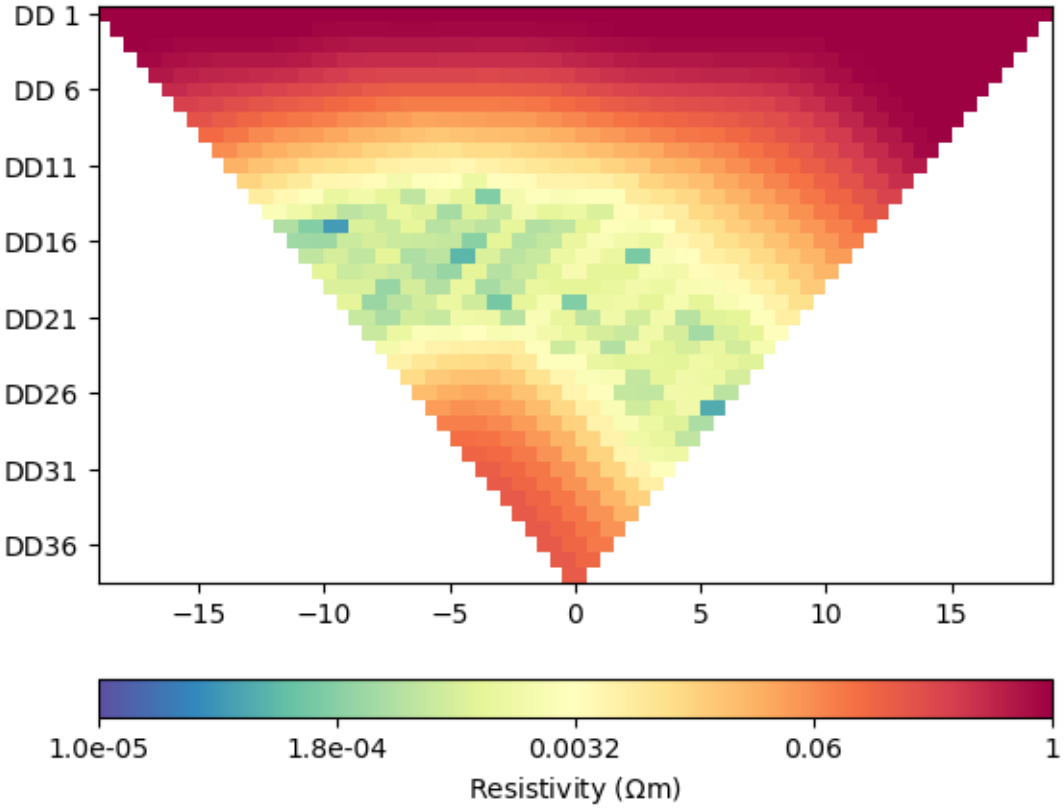
```
# Select 10 time frame to simulate ERT data
timesERT = pg.IVector(np.floor(np.linspace(0, len(c) - 1, 10)))

# Create conductivity of fluid for salt concentration $c$
sigmaFluid = c[timesERT] * 0.1 + 0.01

# Calculate bulk resistivity based on Archie's Law
resBulk = petro.resistivityArchie(rFluid=1. / sigmaFluid, porosity=0.3, m=1.3,
                                  mesh=mesh, meshI=meshERT, fill=1)

# apply background resistivity model
rho0 = np.zeros(meshERT.cellCount()) + 1000.
for c in meshERT.cells():
    if c.center()[1] < -8:
        rho0[c.id()] = 150.
    elif c.center()[1] < -2:
        rho0[c.id()] = 500.
resis = pg.Matrix(resBulk)
for i, rbI in enumerate(resBulk):
    resis[i] = 1. / ((1. / rbI) + 1. / rho0)
```

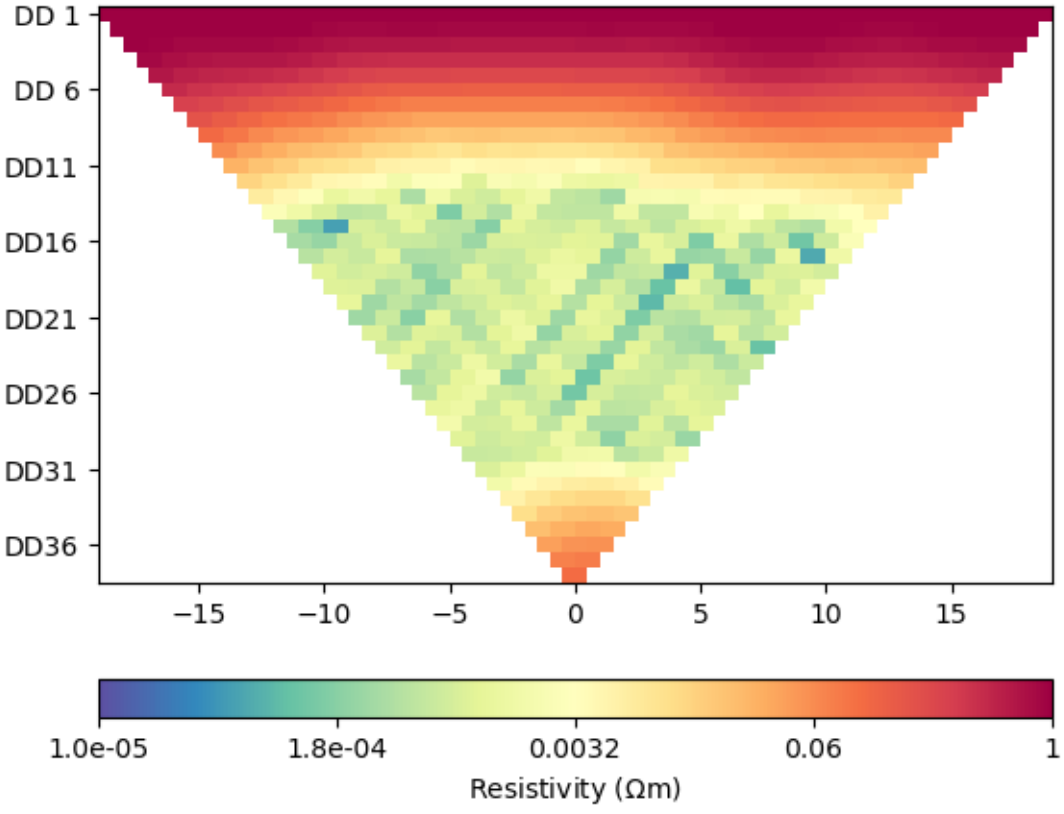
Initialize and call the ERT manager for electrical simulation:

```
ERT = ERTManager(verbose=False)
# Run  simulation for  the apparent resistivities
rhoa = ERT.simulate(meshERT, res=resis, scheme=ertScheme,
                    returnArray=True, verbose=False)

for i in range(4):
    ERT.showData(ertScheme, vals=rhoa[i]/rhoa[0], cMin=1e-5, cMax=1)
```

- 
DD 1
DD 6
DD11
DD16
DD21
DD26
DD31
DD36
−15
−10
−5
0
5
10
15
1.0e-05
1.8e-04
0.0032
0.06
1
Resistivity (Ωm)

- 
DD 1
DD 6
DD11
DD16
DD21
DD26
DD31
DD36
−15
−10
−5
0
5
10
15
1.0e-05
1.8e-04
0.0032
0.06
1
Resistivity (Ωm)



- 

Total running time of the script: (0 minutes 28.077 seconds)

6.7.7 Inversion

6.7.7.1 DC-EM Joint inversion

This is an old script from early pyGIMLi jointly inverting direct current (DC) and electromagnetic (EM) soundings on the modelling abstraction level. Note that this is not recommended as a basis for programming, because there is a dedicated framework for classical joint inversion. However, it explains what happens under the hood in the much simpler script that follows.

The case has been documented by [**?**].

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

import pygimli as pg
from pygimli.viewer.mpl import drawModel1D
```

First, we define a modelling class that calls two other classes and pastes their results to one vector.

```
class DCEM1dModelling(pg.core.ModellingBase):
    """Modelling jointing DC and EM 1Dforward operators."""

    def __init__(self, nlay, ab2, mn2, freq, coilspacing, verbose=False):
        """Init number of layers, AB/2, MN/2, frequencies & coil␣
↪spacing."""
        pg.core.ModellingBase.__init__(self, verbose)
        self.nlay_ = nlay
        self.fDC_ = pg.core.DC1dModelling(nlay, ab2, mn2, verbose)
        self.fEM_ = pg.core.FDEM1dModelling(nlay, freq, coilspacing, verbose)
        self.mesh_ = pg.meshtools.createMesh1DBlock(nlay)
        self.setMesh(self.mesh_)

    def response(self, model):
        """Return concatenated response of DC and EM FOPs."""
        return pg.cat(self.fDC_(model), self.fEM_(model))
```

The actual script starts here. There are some options to play with

```
noiseEM = 1.  # absolute (per cent of primary signal)
noiseDC = 3.  # in per cent
lamEM, lamDC, lamDCEM = 300., 500., 500.  # regularization strength
verbose = False
```

First we create a synthetic model.

```
nlay = 3  # number of layers
thk = pg.Vector(nlay - 1, 15.0)  # 15m thickness each
res = pg.Vector(nlay, 200.0)  # 200 Ohmm
res[1] = 10.
res[2] = 50.
model = pg.cat(thk, res)  # paste together to one model
```

We first set up EM forward operator and generate synthetic data with noise

```
coilspacing = 50.
nf = 10
freq = pg.Vector(nf, 110.)
for i in range(nf-1):
    freq[i+1] = freq[i] * 2.

fEM = pg.core.FDEM1dModelling(nlay, freq, coilspacing)
dataEM = fEM(model)
dataEM += pg.randn(len(dataEM), seed=1234) * noiseEM
```

We define model transformations: logarithms and log with upper+lower bounds

```
transRhoa = pg.trans.TransLog()
transThk = pg.trans.TransLog()
transRes = pg.trans.TransLogLU(1., 1000.)
transEM = pg.trans.Trans()
fEM.region(0).setTransModel(transThk)
fEM.region(1).setTransModel(transRes)
```

We set up the independent EM inversion and run the model.

```
invEM = pg.core.Inversion(dataEM, fEM, transEM, True, True)
modelEM = pg.Vector(nlay * 2 - 1, 50.)
invEM.setModel(modelEM)
invEM.setAbsoluteError(noiseEM)
invEM.setLambda(lamEM)
invEM.setMarquardtScheme(0.9)
modelEM = invEM.run()
respEM = invEM.response()
```

Next we set up the DC forward operator and generate synthetic data with noise

```
ab2 = pg.Vector(20, 3.)
na = len(ab2)
mn2 = pg.Vector(na, 1.0)
for i in range(na-1):
    ab2[i+1] = ab2[i] * 1.3
fDC = pg.core.DC1dModelling(nlay, ab2, mn2)
dataDC = fDC(model)
dataDC *= 1. + pg.randn(len(dataDC), seed=1234) * noiseDC / 100.

fDC.region(0).setTransModel(transThk)
fDC.region(1).setTransModel(transRes)

# We set up the independent DC inversion and let it run.
invDC = pg.core.Inversion(dataDC, fDC, transRhoa, verbose)
modelDC = pg.Vector(nlay*2-1, 20.)
invDC.setModel(modelDC)
invDC.setRelativeError(noiseDC/100.)
invDC.setLambda(lamDC)
invDC.setMarquardtScheme(0.9)
```

(continues on next page)

(continued from previous page)

```
modelDC = invDC.run()
respDC = invDC.response()
```

Next we create a the joint forward operator (see class above).

```
fDCEM = DCEM1dModelling(nlay, ab2, mn2, freq, coilspacing)
fDCEM.region(0).setTransModel(transThk)
fDCEM.region(1).setTransModel(transRes)
```

We setup the joint inversion combining, transformations, data and errors.

```
transData = pg.trans.TransCumulative()
transData.add(transRhoa, na)
transData.add(transEM, nf*2)
invDCEM = pg.core.Inversion(pg.cat(dataDC, dataEM), fDCEM, transData, verbose)
modelDCEM = pg.Vector(nlay * 2 - 1, 20.)
invDCEM.setModel(modelDCEM)
err = pg.cat(dataDC * noiseDC / 100., pg.Vector(len(dataEM), noiseEM))
invDCEM.setAbsoluteError(err)
invDCEM.setLambda(lamDCEM)
invDCEM.setMarquardtScheme(0.9)
modelDCEM = invDCEM.run()
respDCEM = invDCEM.response()
```

The results of the inversion are plotted for comparison.

```
for inv in [invEM, invDC, invDCEM]:
    inv.echoStatus()
print([invEM.chi2(), invDC.chi2(), invDCEM.chi2()])  # chi-square values
```

```
[1.0574072316294534, 1.1414069871330406, 1.1706466870746692]
```

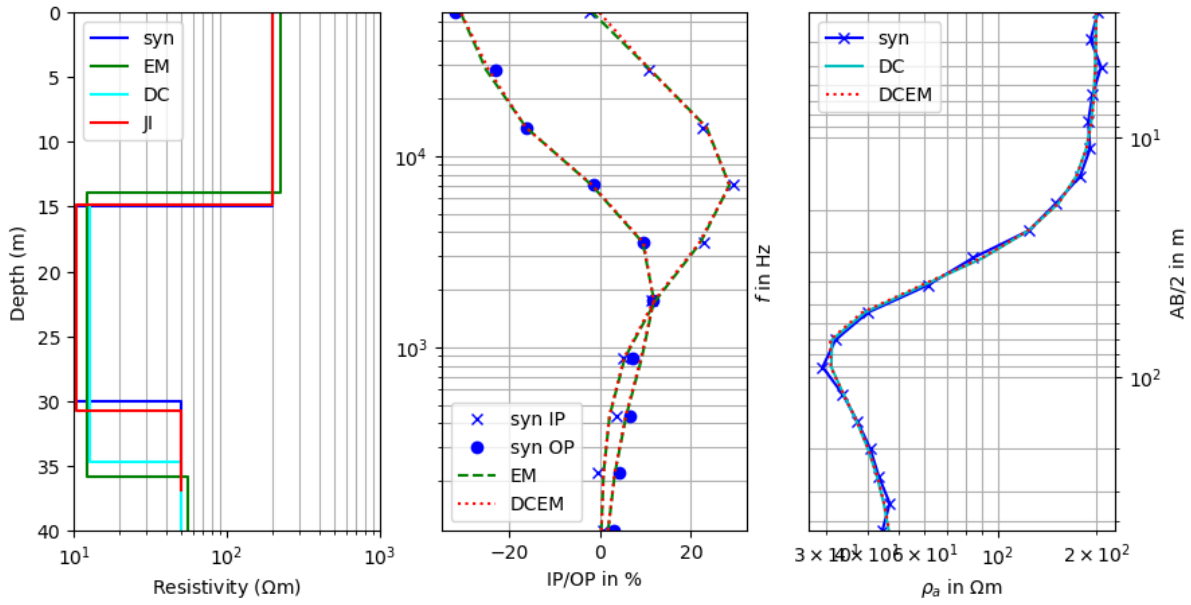
%% We finally plot the results

```
fig = plt.figure(1, figsize=(10, 5))
ax1 = fig.add_subplot(131)
drawModel1D(ax1, thk, res, plot='semilogx', color='blue')
drawModel1D(ax1, modelEM[0:nlay-1], modelEM[nlay-1:nlay*2-1], color='green')
drawModel1D(ax1, modelDC[0:nlay-1], modelDC[nlay-1:nlay*2-1], color='cyan')
drawModel1D(ax1, modelDCEM[0:nlay-1], modelDCEM[nlay-1:nlay*2-1],
            color='red')
ax1.legend(('syn', 'EM', 'DC', 'JI'))
ax1.set_xlim((10., 1000.))
ax1.set_ylim((40., 0.))
ax1.grid(which='both')
ax2 = fig.add_subplot(132)
ax2.semilogy(dataEM[0:nf], freq, 'bx', label='syn IP')
ax2.semilogy(dataEM[nf:nf*2], freq, 'bo', label='syn OP')
ax2.semilogy(respEM[0:nf], freq, 'g--', label='EM')
ax2.semilogy(respEM[nf:nf*2], freq, 'g--')
```

(continues on next page)

(continued from previous page)

```
ax2.semilogy(respDCEM[na:na+nf], freq, 'r:', label='DCEM')
ax2.semilogy(respDCEM[na+nf:na+nf*2], freq, 'r:')
ax2.set_ylim((min(freq), max(freq)))
ax2.set_xlabel("IP/OP in %")
ax2.set_ylabel("$f$ in Hz")
ax2.yaxis.set_label_position("right")
ax2.grid(which='both')
ax2.legend(loc="best")
ax3 = fig.add_subplot(133)
ax3.loglog(dataDC, ab2, 'bx-', label='syn')
ax3.loglog(respDC, ab2, 'c-', label='DC')
ax3.loglog(respDCEM[0:na], ab2, 'r:', label='DCEM')
# ax3.axis('tight')
ax3.set_ylim((max(ab2), min(ab2)))
ax3.grid(which='both')
ax3.set_xlabel(r"$\rho_a$ in $\Omega$m")
ax3.set_ylabel("AB/2 in m")
ax3.yaxis.set_ticks_position("right")
ax3.yaxis.set_label_position("right")
ax3.legend(loc="best")
pg.wait()
```




```
# Günther, T. (2013): On Inversion of Frequency Domain␣
↪Electromagnetic Data in
# Salt Water Problems - Sensitivity and Resolution. Ext. Abstr.,␣
↪19th European
# Meeting of Environmental and Engineering Geophysics, Bochum,␣
↪Germany.
```

6.7.7.2 Petrophysical joint inversion

Joint inversion of different geophysical techniques helps to improve both resolution and interpretability of the resulting images. Different data sets can be directly coupled, if there is a link to an underlying target parameter. In this example, ERT and traveltime data are inverted for water saturation. For details see section 3.3 of the pyGIMLi paper (https://cg17.pygimli.org).

```
import numpy as np
import pygimli as pg

from pygimli import meshtools as mt
from pygimli.physics import ert
from pygimli.physics import traveltime as tt
from pygimli.physics.petro import transFwdArchieS as ArchieTrans
from pygimli.physics.petro import transFwdWyllieS as WyllieTrans
from pygimli.frameworks import (PetroInversionManager,
                                JointPetroInversionManager)
```

We start with defining two helper functions.

```
def createSynthModel():
    """Return the modelling mesh, the porosity distribution and the
       parametric mesh for inversion.
    """
    # Create the synthetic model
    world = mt.createCircle(boundaryMarker=-1, nSegments=64)
    tri = mt.createPolygon([[-0.8, -0], [-0.5, -0.7], [0.7, 0.5]],
                           isClosed=True, area=0.0015)
    c1 = mt.createCircle(radius=0.2, pos=[-0.2, 0.5], nSegments=32,
                         area=0.0025, marker=3)
    c2 = mt.createCircle(radius=0.2, pos=[0.32, -0.3], nSegments=32,
                         area=0.0025, marker=3)

    poly = mt.mergePLC([world, tri, c1, c2])

    poly.addRegionMarker([0.0, 0, 0], 1, area=0.0015)
    poly.addRegionMarker([-0.9, 0, 0], 2, area=0.0015)

    c = mt.createCircle(radius=0.99, nSegments=16, start=np.pi, end=np.pi*3)
    [poly.createNode(p.pos(), -99) for p in c.nodes()]
    mesh = pg.meshtools.createMesh(poly, q=34.4, smooth=[1, 10])
    mesh.scale(1.0/5.0)
    mesh.rotate([0., 0., 3.1415/3])
    mesh.rotate([0., 0., 3.1415])

    petro = pg.solver.parseArgToArray([[1, 0.9], [2, 0.6], [3, 0.3]],
                                      mesh.cellCount(), mesh)

    # Create the parametric mesh that only reflect the domain␣
↪geometry
    world = mt.createCircle(boundaryMarker=-1, nSegments=32, area=0.0051)
```

(continues on next page)

(continued from previous page)

```
    paraMesh = pg.meshtools.createMesh(world, q=34.0, smooth=[1, 10])
    paraMesh.scale(1.0/5.0)

    return mesh, paraMesh, petro


def showModel(ax, model, mesh, petro=1, cMin=None, cMax=None, label=None,
              cMap=None, showMesh=False):
    """Utility function to show and save models for the CG paper."""
    if cMin is None:
        cMin = 0.3
    if cMax is None:
        cMax = 0.9

    if cMap is None:
        cMap = 'viridis'
    if petro:
        ax, _ = pg.show(mesh, model, label=label,
                        logScale=False, cMin=cMin, cMax=cMax, cMap=cMap, ax=ax)
    else:
        ax, _ = pg.show(mesh, model, label=label,
                        logScale=True, cMin=cMin, cMax=cMax, cMap=cMap, ax=ax)

    ticks = [-.2, -.1, 0, .1, .2]
    ax.xaxis.set_ticks(ticks)
    ax.yaxis.set_ticks(ticks)

    pg.viewer.mpl.drawSensors(ax, ertData.sensorPositions(), diam=0.005)

    # despine(ax=ax, offset=5, trim=True)
    if showMesh:
        pg.viewer.mpl.drawSelectedMeshBoundaries(ax, mesh.boundaries(),
                                                 linewidth=0.3, color="0.2")
    return ax
```

Create synthetic model

```
mMesh, pMesh, saturation = createSynthModel()
```

Create Petrophysical models

```
ertTrans = ArchieTrans(rFluid=20, phi=0.3)
res = ertTrans(saturation)

ttTrans = WyllieTrans(vm=4000, phi=0.3)
vel = 1./ttTrans(saturation)

sensors = mMesh.positions()[mMesh.findNodesIdxByMarker(-99)]
```

Forward simulation

To create synthetic data sets, we assume 16 equally-spaced sensors on the circumferential boundary of the mesh. For the ERT modelling we build a complete dipole-dipole array. For the ultrasonic tomography we simulate the travel time for every possible sensor pair.

```
pg.info("Simulate ERT")
ertScheme = ert.createERTData(sensors, schemeName='dd', closed=1)
ertData = ert.simulate(mMesh, scheme=ertScheme, res=res, noiseLevel=0.01)

pg.info("Simulate Traveltime")
ttScheme = tt.createRAData(sensors)
ttData = tt.simulate(mMesh, scheme=ttScheme, vel=vel,
                     noiseLevel=0.01, noiseAbs=4e-6)
```

```
Data error estimate (min:max)  0.010000190585099936 : 0.01001232215607078
```

Conventional inversion

```
pg.info("ERT Inversion")
ERT = ert.ERTManager(verbose=False, sr=False)
resInv = ERT.invert(ertData, mesh=pMesh, zWeight=1, lam=20, verbose=False)
ERT.inv.echoStatus()

pg.info("Traveltime Inversion")
TT = tt.TravelTimeManager(verbose=False)
velInv = TT.invert(ttData, mesh=pMesh, lam=100, useGradient=0, zWeight=1.0)
TT.inv.echoStatus()
```

Petrophysical inversion (individually)

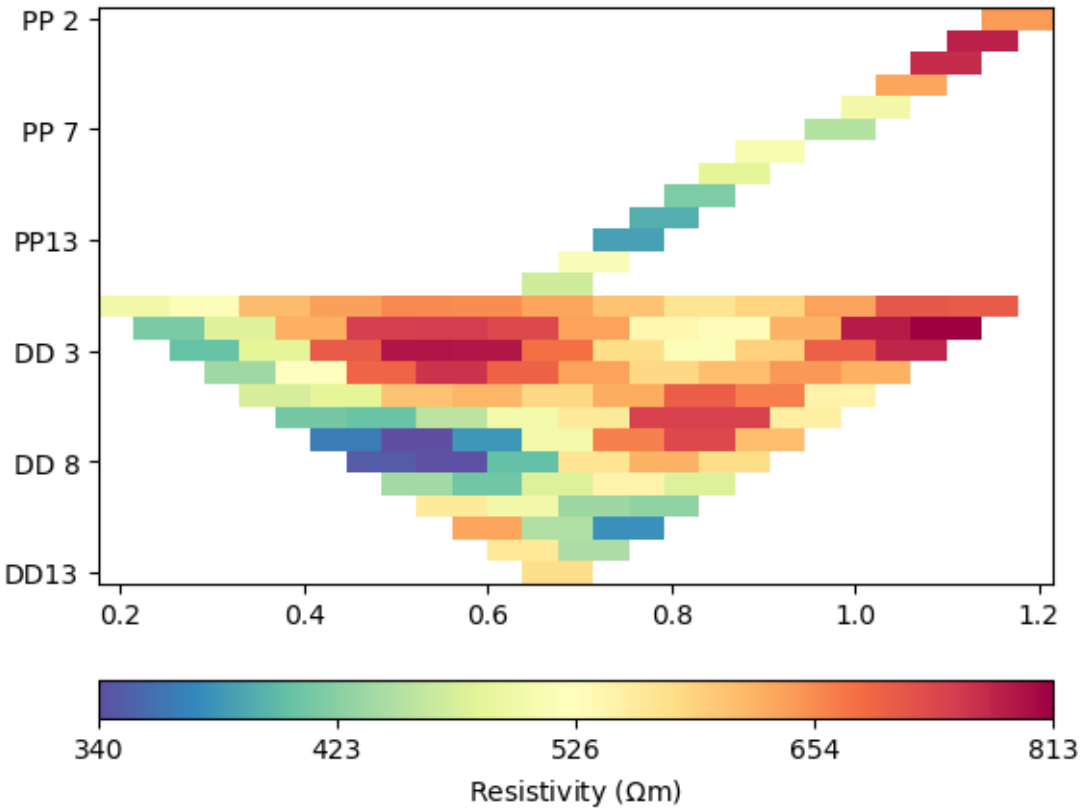
```
pg.info("ERT Petrogeophysical Inversion")
ERTPetro = PetroInversionManager(petro=ertTrans, mgr=ERT)
satERT = ERTPetro.invert(ertData, mesh=pMesh, limits=[0., 1.], lam=10,
                         verbose=False)
ERTPetro.inv.echoStatus()

pg.info("TT Petrogeophysical Inversion")
TTPetro = PetroInversionManager(petro=ttTrans, mgr=TT)
satTT = TTPetro.invert(ttData, mesh=pMesh, limits=[0., 1.], lam=5)
TTPetro.inv.echoStatus()
```

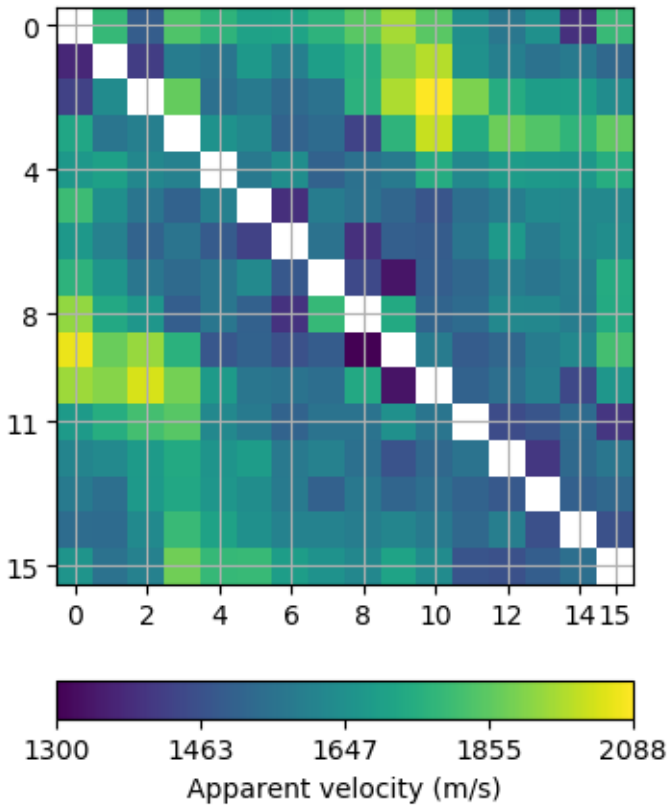
Petrophysical joint inversion

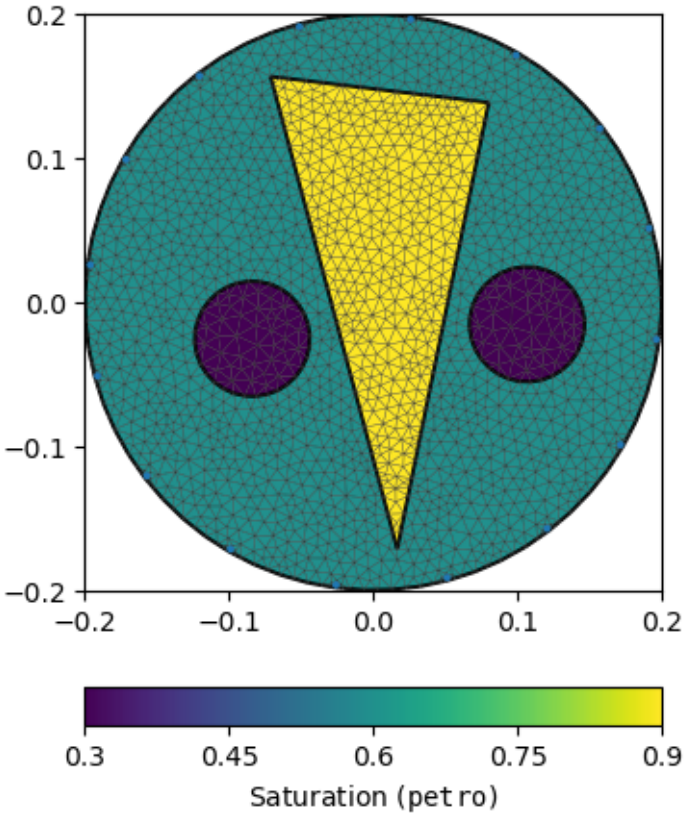
```
pg.info("Petrophysical Joint-Inversion TT-ERT")
JointPetro = JointPetroInversionManager(petros=[ertTrans, ttTrans],
                                        mgrs=[ERT, TT])
satJoint = JointPetro.invert([ertData, ttData], mesh=pMesh,
                             limits=[0., 1.], lam=5, verbose=False)
JointPetro.inv.echoStatus()
```

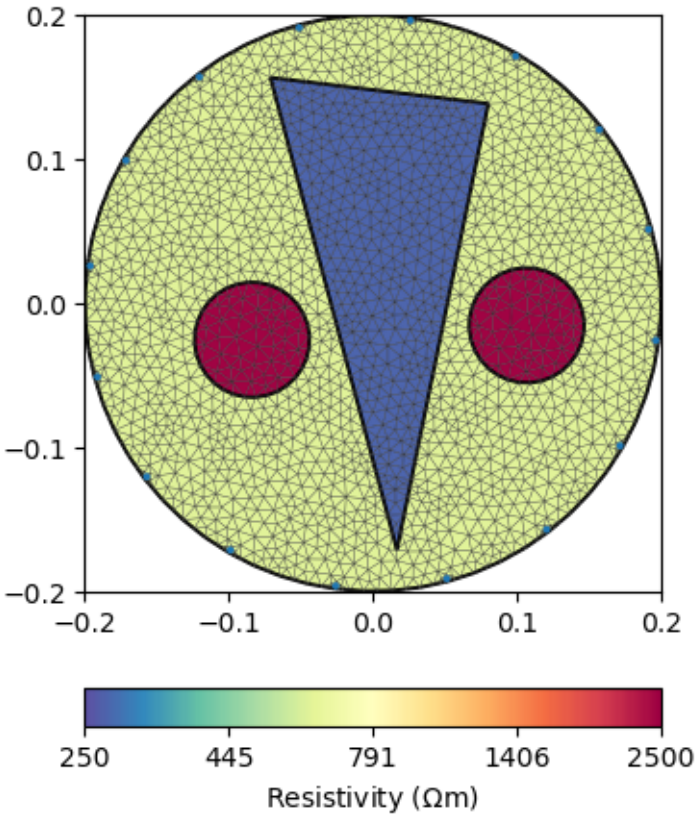
Visualization

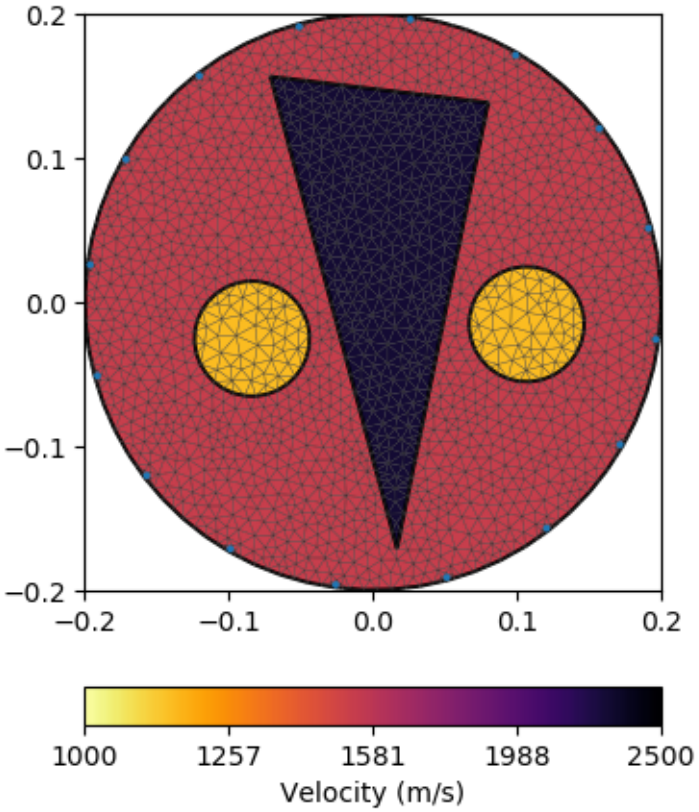
```
ERT.showData(ertData)
TT.showData(ttData)

axs = [None]*8

showModel(axs[0], saturation, mMesh, showMesh=True,
          label=r'Saturation (${\tt petro}$)')
showModel(axs[1], res, mMesh, petro=0, cMin=250, cMax=2500, showMesh=1,
          label=pg.unit('res'), cMap=pg.cmap('res'))
showModel(axs[5], vel, mMesh, petro=0, cMin=1000, cMax=2500, showMesh=1,
          label=pg.unit('vel'), cMap=pg.cmap('vel'))
showModel(axs[2], resInv, pMesh, 0, cMin=250, cMax=2500,
          label=pg.unit('res'), cMap=pg.cmap('res'))
showModel(axs[6], velInv, pMesh, 0, cMin=1000, cMax=2500,
          label=pg.unit('vel'), cMap=pg.cmap('vel'))
showModel(axs[3], satERT, pMesh,
          label=r'Saturation (${\tt satERT}$)')
showModel(axs[7], satTT, pMesh,
          label=r'Saturation (${\tt satTT}$)')
showModel(axs[4], satJoint, pMesh,
          label=r'Saturation (${\tt satJoint}$)')
```

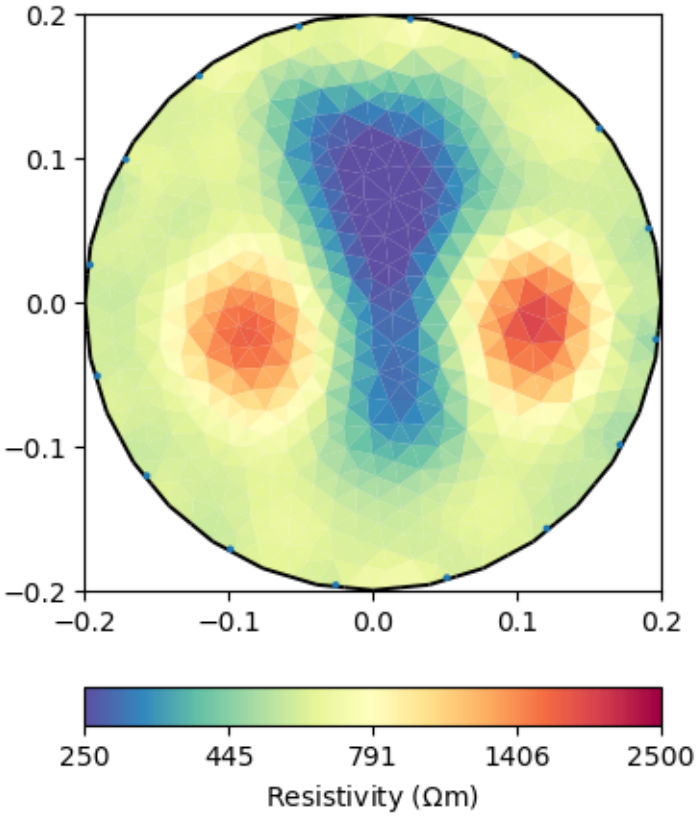
- 
PP 2
PP 7
PP13
DD 3
DD 8
DD13
0.2
0.4
0.6
0.8
1.0
1.2
340
423
526
654
813
Resistivity (Ωm)

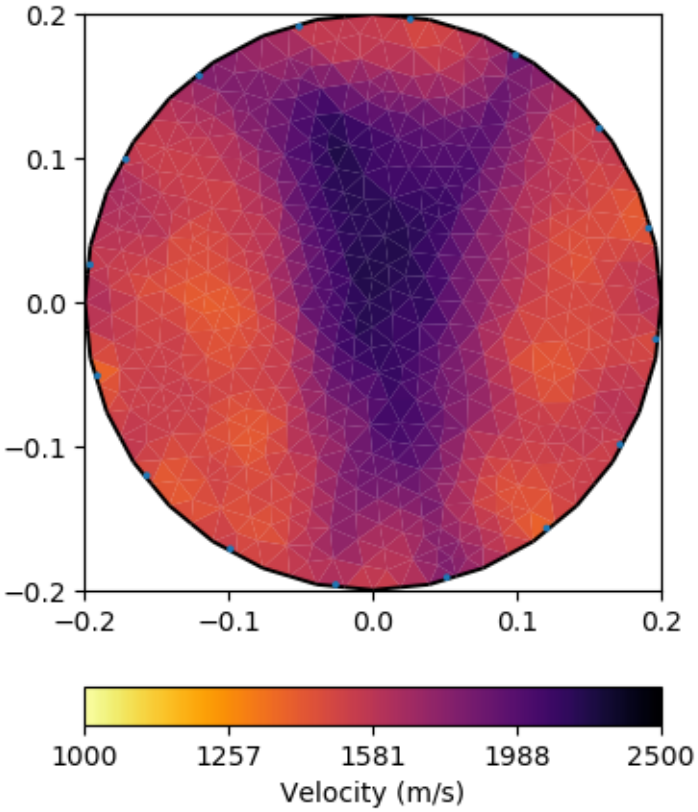
- 
0
4
8
11
15
0
2
4
6
8
10
12
14
15
1300
1463
1647
1855
2088
Apparent velocity (m/s)

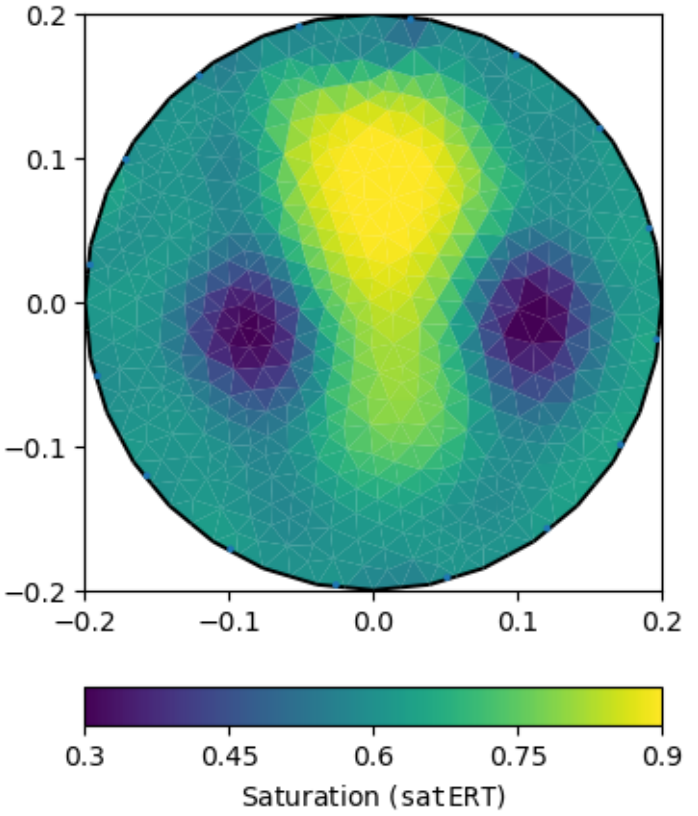

0.2
0.1
0.0
−0.1
−0.2
−0.2
−0.1
0.0
0.1
0.2
0.3
0.45
0.6
0.75
0.9
Saturation (petro)

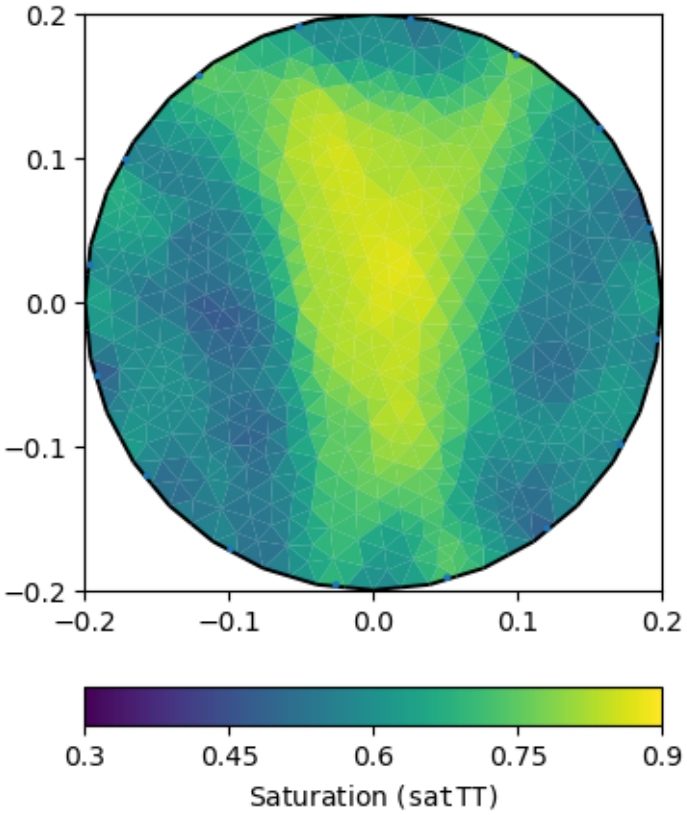
- 
0.2
0.1
0.0
−0.1
−0.2
−0.2
−0.1
0.0
0.1
0.2
250
445
791
1406
2500
Resistivity (Ωm)

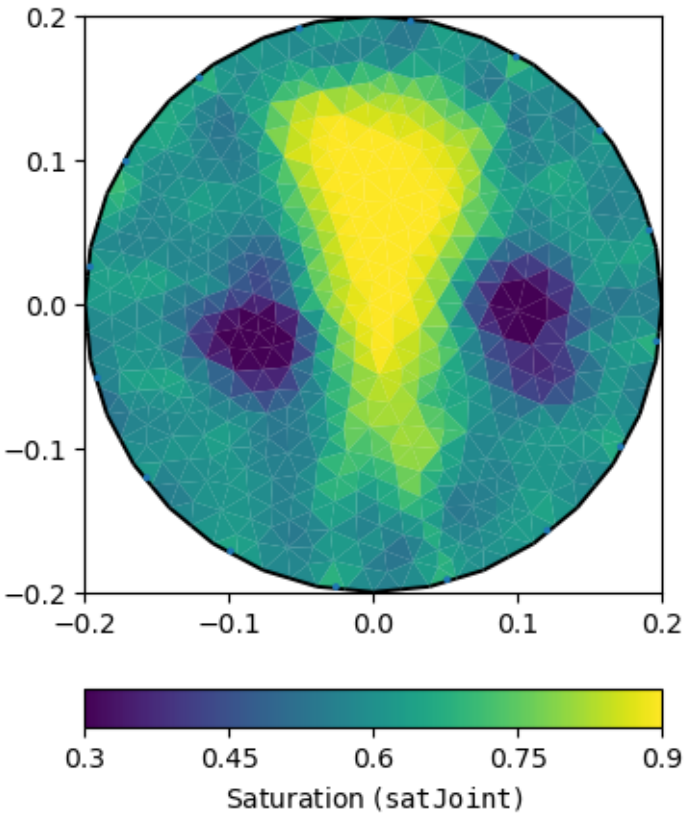
- 
0.2
0.1
0.0
−0.1
−0.2
−0.2
−0.1
0.0
0.1
0.2
1000
1257
1581
1988
2500
Velocity (m/s)

- 
0.2
0.1
0.0
−0.1
−0.2
−0.2
−0.1
0.0
0.1
0.2
250
445
791
1406
2500
Resistivity (Ωm)

- 
0.2
0.1
0.0
−0.1
−0.2
−0.2
−0.1
0.0
0.1
0.2
1000
1257
1581
1988
2500
Velocity (m/s)

- 
0.2
0.1
0.0
−0.1
−0.2
−0.2
−0.1
0.0
0.1
0.2
0.3
0.45
0.6
0.75
0.9
Saturation (satERT)



- 


```
Detecting small distances, using mm accuracy
```

(continues on next page)

(continued from previous page)

```
<matplotlib.axes._subplots.AxesSubplot object at 0x7fe8ab1cb3a0>
```

Total running time of the script: (0 minutes 15.023 seconds)

6.7.7.3 Incorporating prior data into ERT inversion

Prior data can often help to overcome ambiguity in the inversion process. Here we demonstrate the use of direct push (DP) data in an ERT inversion of data collected at the surface.

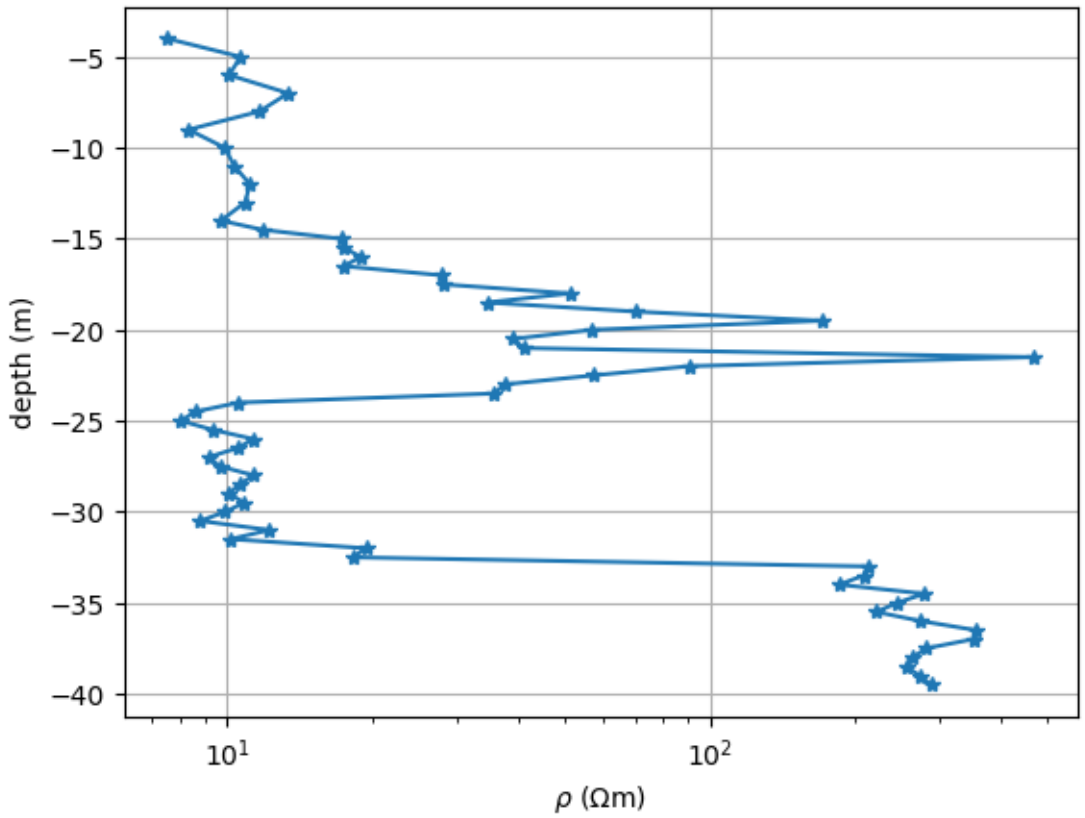
```
import matplotlib.pyplot as plt
import numpy as np
import pygimli as pg
from pygimli.physics import ert
from pygimli.frameworks import PriorModelling, JointModelling
```

The prior data

This field data is from a site with layered sands and clays over a resistive bedrock. We load it from the example repository.

As a position of x=155m (center of the profile) we have a borehole/direct push with known in-situ-data. We load the three-column file using numpy.

```
x, z, r = pg.getExampleData("ert/bedrock.txt").T
fig, ax = plt.subplots()
ax.semilogx(r, z, "*-")
ax.set_xlabel(r"$\rho$ ($\Omega$m)")
ax.set_ylabel("depth (m)")
ax.grid(True)
```

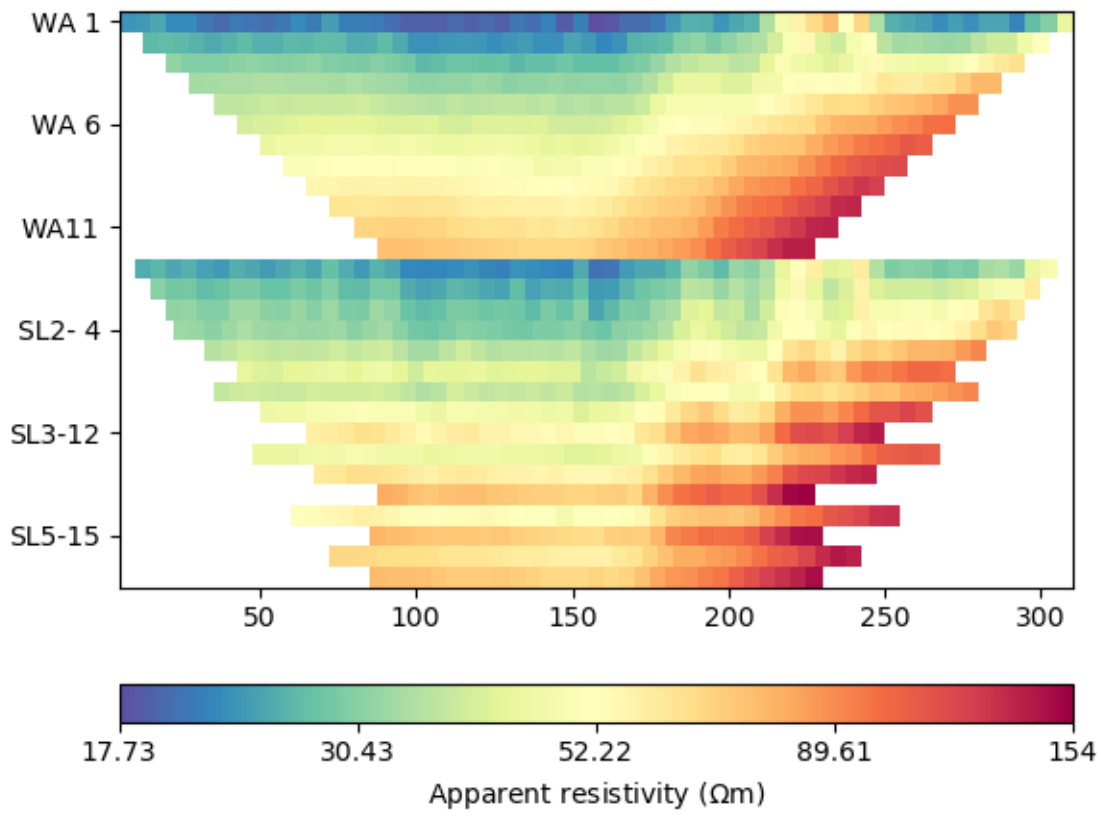


We mainly see four layers: 1. a conductive (clayey) overburden of about 17m thickness, 2. a medium resistivity interbedding of silt and sand, about 7m thick 3. again clay with 8m thickness 4. the resistive bedrock with a few hundred :math:`Omega` m

The ERT data

We load the ERT data from the example repository and plot the pseudosection.

```
data = pg.getExampleData("ert/bedrock.dat")
print(data)
ax, cb = ert.show(data)
```




```
Data: Sensors: 64 data: 1223, nonzero entries: ['a', 'b', 'err', 'm', 'n',
↪'rhoa', 'valid']
```

The apparent resistivities show increasing values with larger spacings with no observable noise. We first compute geometric factors and estimate an error model using rather low values for both error parts.

```
data["k"] = ert.geometricFactors(data)
data["err"] = ert.estimateError(data, relativeError=0.02, absoluteUError=50e-6)
# ax, cb = ert.show(data, data["err"]*100, label="error (%)")
mgr = ert.ERTManager(data, verbose=True)
mgr.invert(paraDepth=80, quality=34.6, paraMaxCellSize=100)
# mgr.showFit()
```

```
fop: <pygimli.physics.ert.ertModelling.ERTModelling object at 0x7fe8717659a0>
Data transformation: <pygimli.core._pygimli_.RTransLogLU object at␣
↪0x7fe871765770>
Model transformation: <pygimli.core._pygimli_.RTransLog object at 0x7fe8717657c0>
min/max (data): 17.73/154
min/max (error): 2.02%/2.94%
min/max (start model): 48.34/48.34
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
inv.iter 2 ... chi² = 4.12 (dPhi = 17.77%) lam: 20
--------------------------------------------------------------------------------
inv.iter 3 ... chi² = 3.33 (dPhi = 17.18%) lam: 20.0
--------------------------------------------------------------------------------
```

(continues on next page)

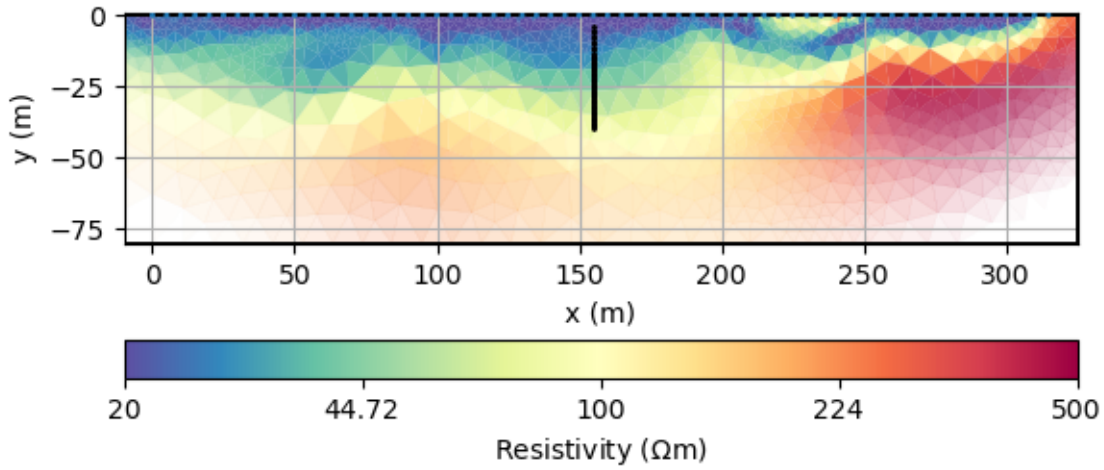
(continued from previous page)

```
inv.iter 4 ... chi² = 2.39 (dPhi = 23.22%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 5 ... chi² = 0.52 (dPhi = 54.25%) lam: 20.0


################################################################################
#                    Abort criterion reached: chi² <= 1 (0.52)                 #
################################################################################

1674 [17.34290938386768,...,173.3671584065751]
```

For reasons of comparability, we define a unique colormap and store all options in a dictionary to be used in subsequent show commands.

We plot the result with these and plot the DP points onto the mesh.

```
kw = dict(cMin=20, cMax=500, logScale=True, cMap="Spectral_r",
          xlabel="x (m)", ylabel="y (m)")
ax, cb = mgr.showResult(**kw)
ax.plot(x, z, ".", color="black", linestyle="None", markersize=2)
ax.grid(True)
```



We want to extract the resistivity from the mesh at the positions where the prior data are available. To this end, we create a list of positions (`pg.Pos` class) and use a forward operator that picks the values from a model vector according to the cell where the position is in. See the regularization tutorial for details about that.

```
posVec = [pg.Pos(pos) for pos in zip(x, z)]
para = pg.Mesh(mgr.paraDomain)  # make a copy
para.setCellMarkers(pg.IVector(para.cellCount()))
fopDP = PriorModelling(para, posVec)
```

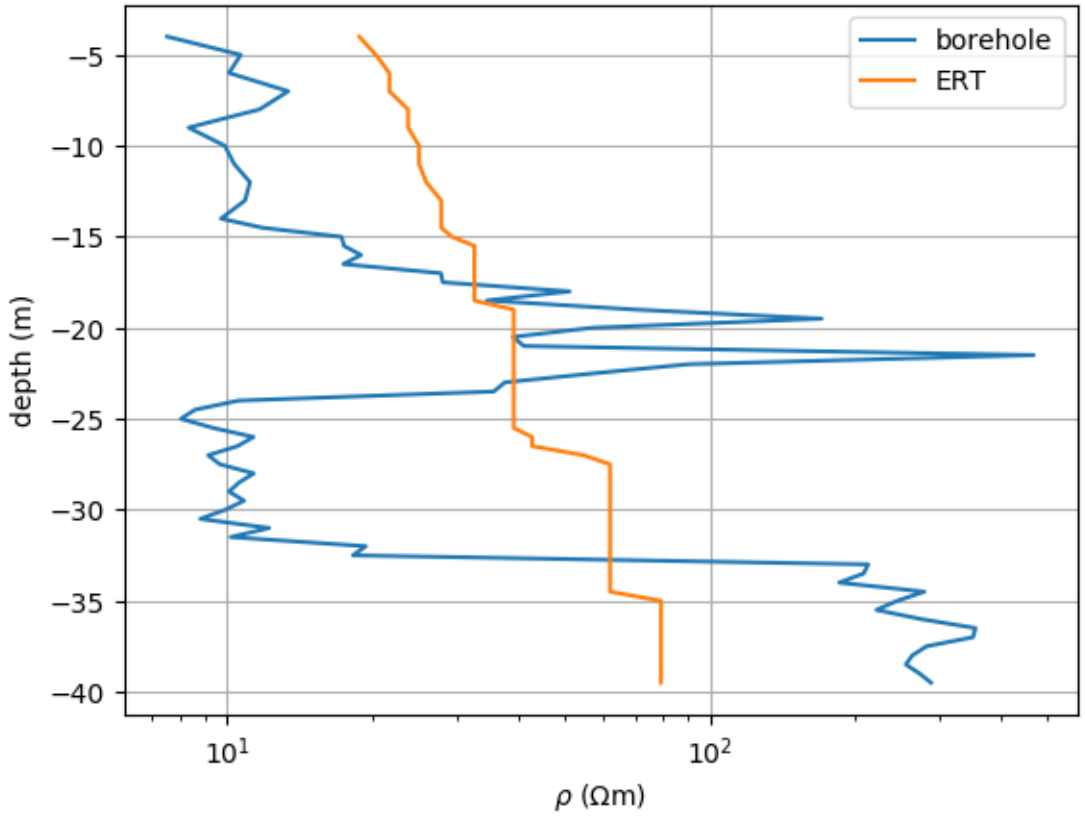
We can now use it to retrieve the model values, store it and plot it along with the measured values.

```
fig, ax = plt.subplots()
ax.semilogx(r, z, label="borehole")
```

(continues on next page)

(continued from previous page)

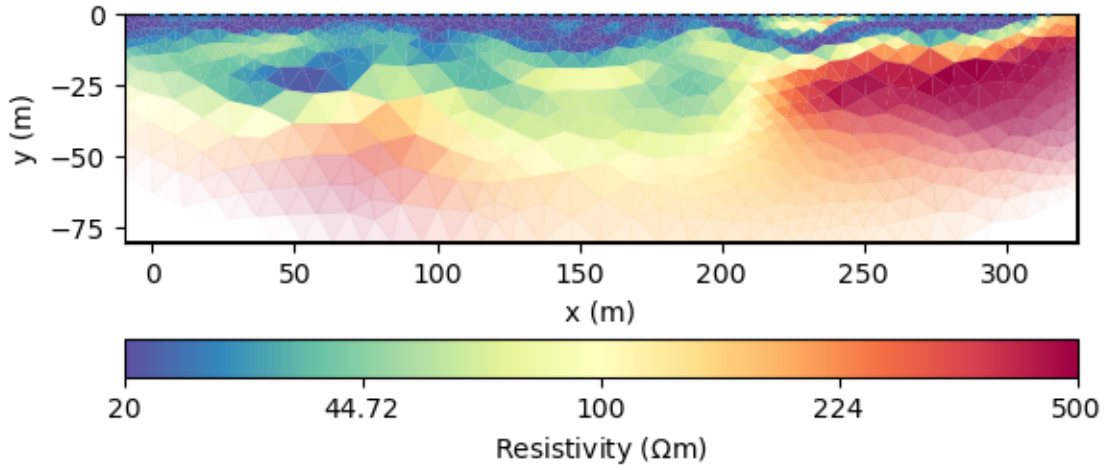
```
res1 = fopDP(mgr.model)
ax.semilogx(res1, z, label="ERT")
ax.set_xlabel(r"$\rho$ ($\Omega$m)")
ax.set_ylabel("depth (m)")
ax.grid(True)
ax.legend()
```




```
<matplotlib.legend.Legend object at 0x7fe87a415cd0>
```

Even though the tomogram looks interesting, the resistivity seems to follow a simple gradient. This is apparently a lack of resolution. Our assumption of an overall smooth image is wrong. Therefore we use an anisotropic smoothness using the vertical weighting factor `zWeight`.

```
mgr.inv.setRegularization(zWeight=0.2)
mgr.invert()
ax, cb = mgr.showResult(**kw)
res2 = fopDP(mgr.model)
```



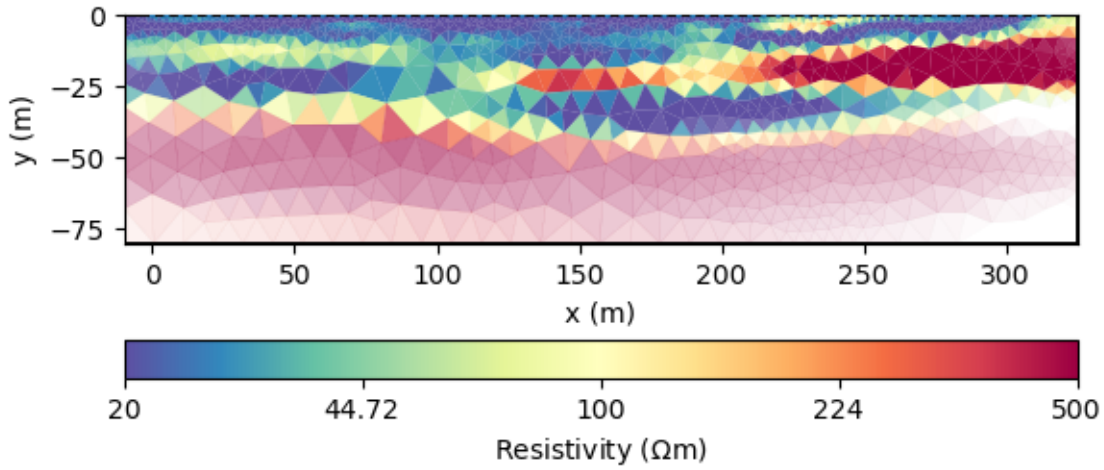

```
fop: <pygimli.physics.ert.ertModelling.ERTModelling object at 0x7fe8717659a0>
Data transformation: <pygimli.core._pygimli_.RTransLogLU object at␣
↪0x7fe871765770>
Model transformation (cumulative):
         0 <pygimli.core._pygimli_.RTransLogLU object at 0x7fe89987eb20>
min/max (data): 17.73/154
min/max (error): 2.02%/2.94%
min/max (start model): 48.34/48.34
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
inv.iter 2 ... chi² = 12.26 (dPhi = 6.22%) lam: 20
--------------------------------------------------------------------------------
inv.iter 3 ... chi² = 11.52 (dPhi = 6.02%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 4 ... chi² = 10.84 (dPhi = 5.83%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 5 ... chi² = 10.08 (dPhi = 6.95%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 6 ... chi² = 9.16 (dPhi = 9.05%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 7 ... chi² = 7.99 (dPhi = 12.73%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 8 ... chi² = 6.05 (dPhi = 24.05%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 9 ... chi² = 1.03 (dPhi = 80.29%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 10 ... chi² = 0.26 (dPhi = 62.21%) lam: 20.0


################################################################################
#                    Abort criterion reached: chi² <= 1 (0.26)                 #
################################################################################
```

We observe a much more structured result with stronger vertical gradients that are, however, not continuous. In the middle of the profile we can see a short layer of increased resistivity.

As alternative, we can use a geostatistic model. The vertical range can be well estimated from the DP

data using a variogram analysis, we guess 5m. For the horizontal one, we can only guess a 10m higher value.

```
mgr.inv.setRegularization(2, correlationLengths=[50, 5])
mgr.invert()
ax, cb = mgr.showResult(**kw)
res3 = fopDP(mgr.model)
```




```
fop: <pygimli.physics.ert.ertModelling.ERTModelling object at 0x7fe8717659a0>
Data transformation: <pygimli.core._pygimli_.RTransLogLU object at␣
↪0x7fe871765770>
Model transformation (cumulative):
         0 <pygimli.core._pygimli_.RTransLogLU object at 0x7fe89987eac0>
min/max (data): 17.73/154
min/max (error): 2.02%/2.94%
min/max (start model): 48.34/48.34
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
inv.iter 2 ... chi² = 12.94 (dPhi = 83.49%) lam: 20
--------------------------------------------------------------------------------
inv.iter 3 ... chi² = 1.6 (dPhi = 85.92%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 4 ... chi² = 1.07 (dPhi = 29.43%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 5 ... chi² = 0.31 (dPhi = 58.29%) lam: 20.0


#########################################################################
#                   Abort criterion reached: chi² <= 1 (0.31)           #
#########################################################################
```

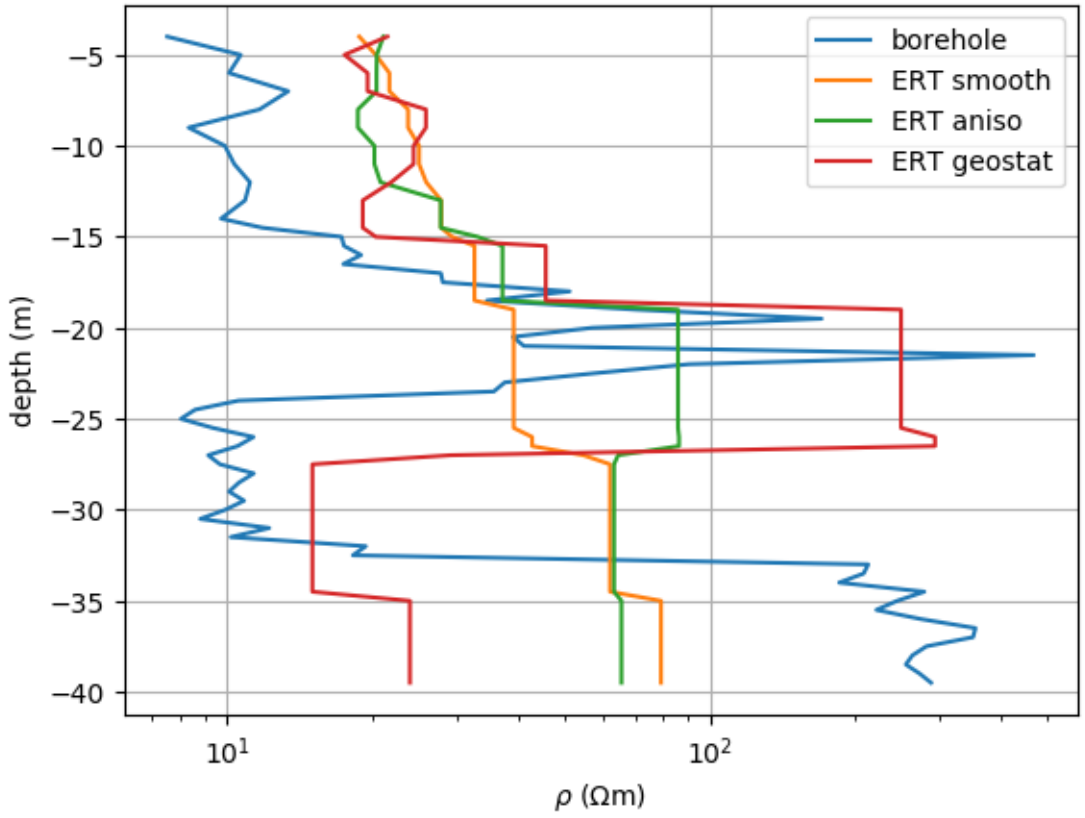
Let's compare the three resistivity soundings with the ground truth.

```
fig, ax = plt.subplots()
ax.semilogx(r, z, label="borehole")
ax.semilogx(res1, z, label="ERT smooth")
```

(continues on next page)

(continued from previous page)

```
ax.semilogx(res2, z, label="ERT aniso")
ax.semilogx(res3, z, label="ERT geostat")
ax.set_xlabel(r"$\rho$ ($\Omega$m)")
ax.set_ylabel("depth (m)")
ax.grid()
ax.legend()
```




```
<matplotlib.legend.Legend object at 0x7fe89994ee50>
```

The anisotropic regularization starts to see the good conductor, but only the geostatistical regularization operator is able to retrieve values that are close to the direct push. Both show the conductor too deep.

One alternative could be to use the interfaces as structural constraints in the neighborhood of the borehole. See ERT with structural constraints example

As the DP data is not only good for comparison, we want to use its values as data in inversion. This is easily accomplished by taking the mapping operator that we already use for interpolation as a forward operator.

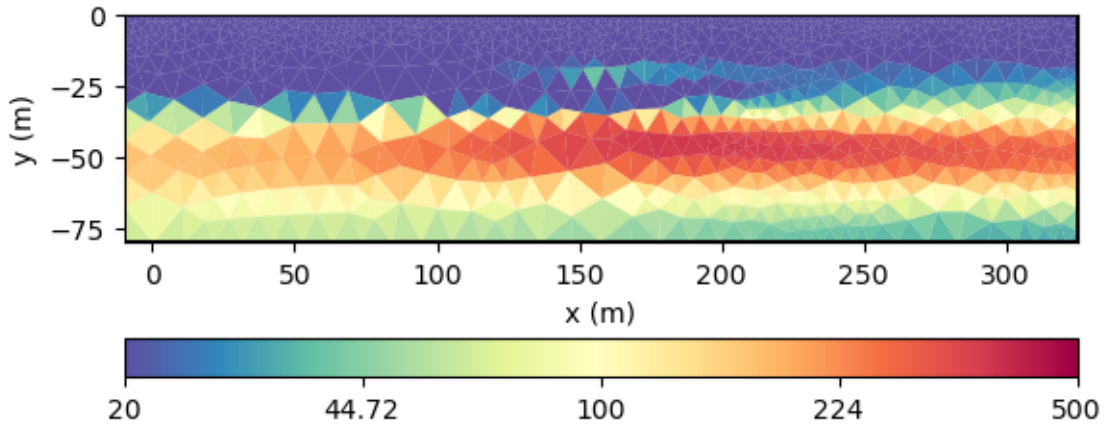
We set up an inversion with this mesh, logarithmic transformations and invert the model.

```
inv = pg.Inversion(fop=fopDP, verbose=True)
inv.mesh = para
tLog = pg.trans.TransLog()
inv.modelTrans = tLog
inv.dataTrans = tLog
# inv.setRegularization(zWeight=0.2)
```

(continues on next page)

(continued from previous page)

```
inv.setRegularization(correlationLengths=[50, 5])
rError = np.ones_like(r)*0.1
model = inv.run(r, rError)
ax, cb = pg.show(para, model, **kw)
```




```
fop: <pygimli.frameworks.modelling.PriorModelling object at 0x7fe8ab0e5950>
Data transformation: <pygimli.core._pygimli_.RTransLog object at 0x7fe89a08f680>
Model transformation (cumulative):
         0 <pygimli.core._pygimli_.RTransLogLU object at 0x7fe89a0bd460>
min/max (data): 7.52/469
min/max (error): 10%/10%
min/max (start model): 17.84/17.84
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
inv.iter 2 ... chi² = 73.71 (dPhi = 0.19%) lam: 20
--------------------------------------------------------------------------------
inv.iter 3 ... chi² = 73.71 (dPhi = 0.05%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 4 ... chi² = 73.71 (dPhi = 0.04%) lam: 20.0
################################################################################
#                  Abort criteria reached: dPhi = 0.04 (< 2.0%)                #
################################################################################
```

Apparently, the geostatistical operator can be used to extrapolate values with given assumptions.

Joint inversion of ERT and DP data

We now use the framework `JointModelling` to combine the ERT and the DP forward operators. So we set up a new ERT modelling operator and join it with `fopDP`.

```
# fopERT = ert.ERTModelling()
# fopERT.setMesh(mesh)
# fopERT.setData(data) # not necessary as done by JointModelling
# fopJoint = JointModelling([fopERT, fopDP])
fopJoint = JointModelling([mgr.fop, fopDP])
```

(continues on next page)

(continued from previous page)

```
# fopJoint.setMesh(para)
fopJoint.setData([data, pg.Vector(r)])  # needs to have .size() attribute!
```

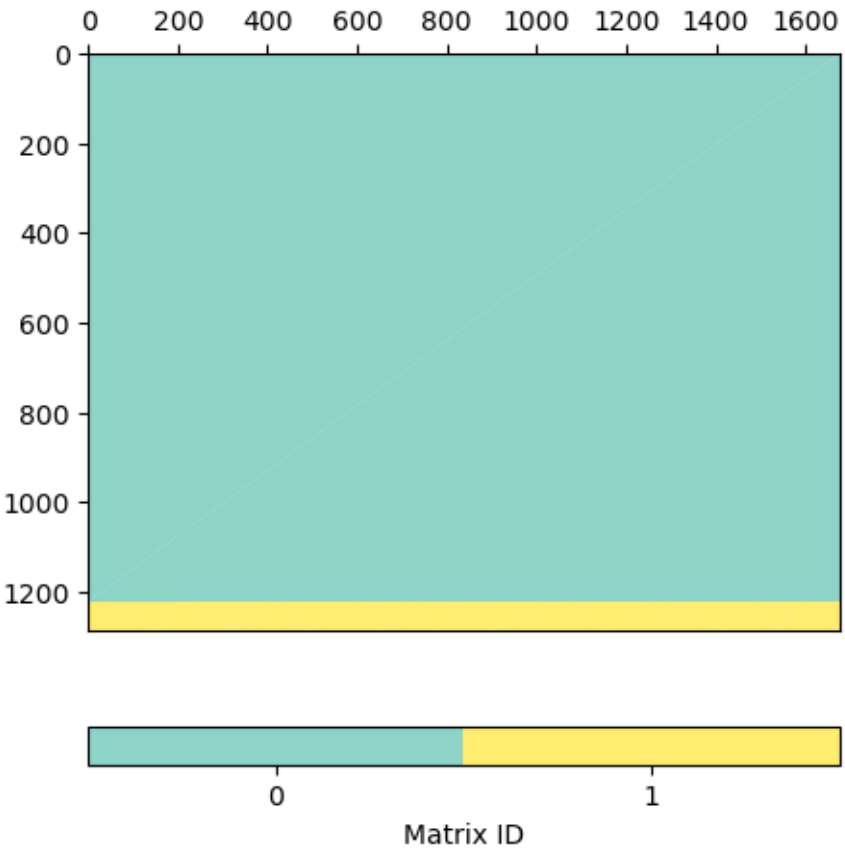
We first test the joint forward operator. We create a modelling vector of constant resistivity and distribute the model response into the two parts that can be looked at individually.

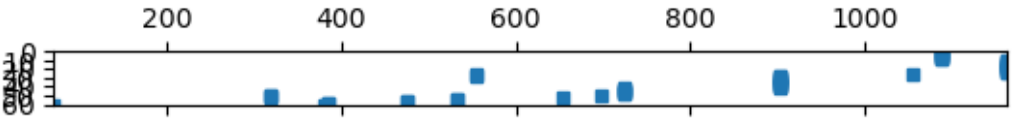
```
model = pg.Vector(para.cellCount(), 100)
response = fopJoint(model)
respERT = response[:data.size()]
respDP = response[data.size():]
print(respDP)
# ax, cb = ert.show(data, respERT)
```

```
[100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0,␣
↪100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0,␣
↪100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0,␣
↪100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0,␣
↪100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0,␣
↪100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0]
```

The jacobian can be looked up

```
# test Jacobian
fopJoint.createJacobian(model)  # works
J = fopJoint.jacobian()
print(type(J))  # wrong type

ax, cb = pg.show(J)
print(J.mat(0))
ax, cb = pg.show(J.mat(1), markersize=4)
```

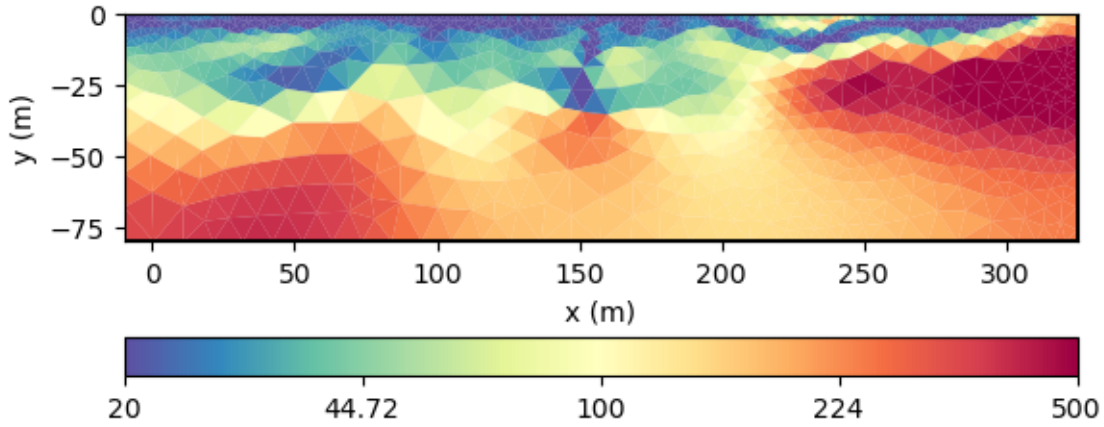





```
<class 'pygimli.core._pygimli_.RBlockMatrix'>
RMatrix: 1223 x 1674
```

For the joint inversion, concatenate the data and error vectors, create a new inversion instance, set logarithmic transformations and run the inversion.

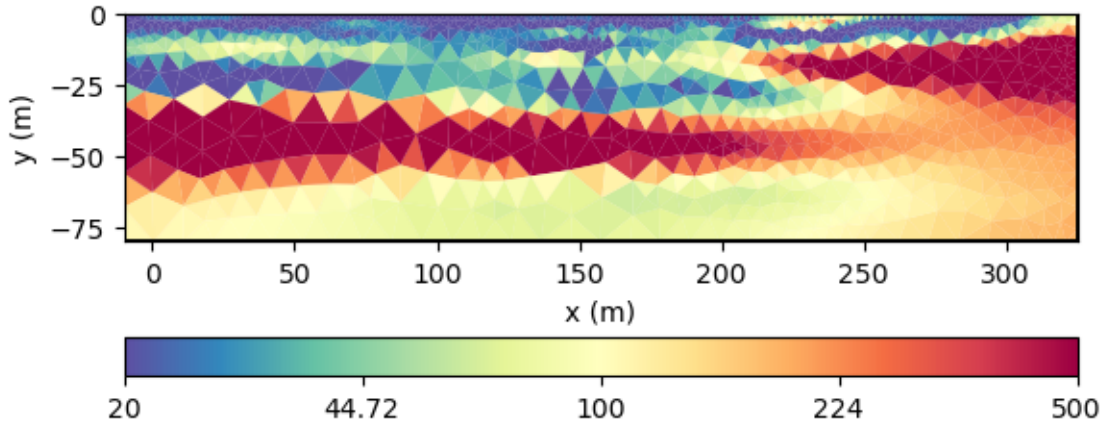
```
dataVec = np.concatenate((data["rhoa"], r))
errorVec = np.concatenate((data["err"], rError))
inv = pg.Inversion(fop=fopJoint, verbose=True)
transLog = pg.trans.TransLog()
inv.modelTrans = transLog
inv.dataTrans = transLog
inv.run(dataVec, errorVec, startModel=model)

ax, cb = pg.show(para, inv.model, **kw)
```




```
fop: <pygimli.frameworks.modelling.JointModelling object at 0x7fe8ab0e5130>
Data transformation: <pygimli.core._pygimli_.RTransLog object at 0x7fe899fe6a90>
Model transformation (cumulative):
         0 <pygimli.core._pygimli_.RTransLogLU object at 0x7fe8716ccf40>
min/max (data): 7.52/469
min/max (error): 2.02%/10%
min/max (start model): 100/100
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
inv.iter 2 ... chi² = 8.5 (dPhi = 41.97%) lam: 20
--------------------------------------------------------------------------------
inv.iter 3 ... chi² = 4.38 (dPhi = 46.99%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 4 ... chi² = 3.87 (dPhi = 10.91%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 5 ... chi² = 3.85 (dPhi = 0.46%) lam: 20.0
################################################################################
#                 Abort criteria reached: dPhi = 0.46 (< 2.0%)                 #
################################################################################
```

We have a local improvement of the model in the neighborhood of the borehole. Now we want to use geostatistics to get them further into the model.

```
inv.setRegularization(2, correlationLengths=[50, 5])
model = inv.run(dataVec, errorVec, startModel=model)
ax, cb = pg.show(para, model, **kw)
```




```
fop: <pygimli.frameworks.modelling.JointModelling object at 0x7fe8ab0e5130>
Data transformation: <pygimli.core._pygimli_.RTransLog object at 0x7fe899fe6a90>
Model transformation (cumulative):
         0 <pygimli.core._pygimli_.RTransLogLU object at 0x7fe8aaf89580>
min/max (data): 7.52/469
min/max (error): 2.02%/10%
min/max (start model): 100/100
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
inv.iter 2 ... chi² = 13.9 (dPhi = 58.75%) lam: 20
--------------------------------------------------------------------------------
inv.iter 3 ... chi² = 7.33 (dPhi = 46.08%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 4 ... chi² = 5.59 (dPhi = 22.78%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 5 ... chi² = 3.91 (dPhi = 28.27%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 6 ... chi² = 3.9 (dPhi = 0.27%) lam: 20.0
################################################################################
#                  Abort criteria reached: dPhi = 0.27 (< 2.0%)                  #
################################################################################
```

This model much better resembles the subsurface from all data and our expectations to it.

We split the model response in the ERT part and the DP part. The first is shown as misfit.

```
respERT = inv.response[:data.size()]
misfit = - respERT / data["rhoa"] * 100 + 100
# ax, cb = ert.show(data, misfit, cMap="bwr", cMin=-5, cMax=5)
```

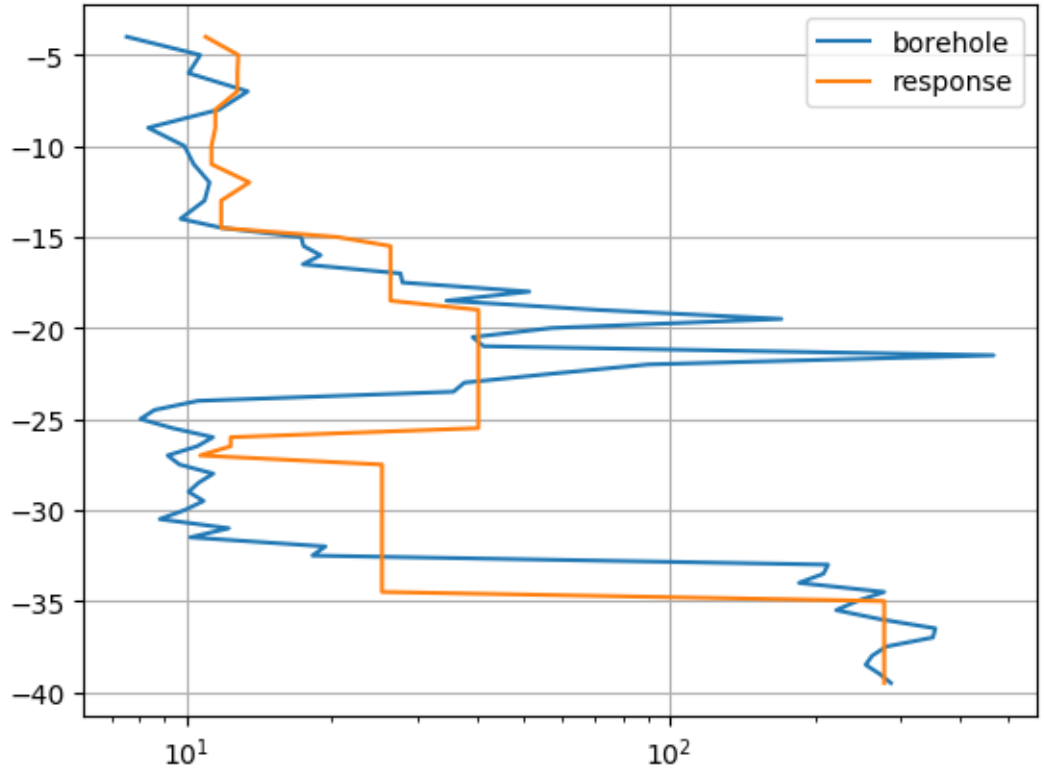
The second is shown as depth profile.

```
respDP = inv.response[data.size():]
fig, ax = plt.subplots()
ax.semilogx(r, z, label="borehole")
# resMesh = pg.interpolate(srcMesh=para, inVec=inv.model,␣
↪destPos=posVec)
# ax.semilogx(resMesh, z, label="ERT+DP")
```

(continues on next page)

(continued from previous page)

```
ax.semilogx(respDP, z, label="response")
ax.grid(True)
ax.legend()
```




```
<matplotlib.legend.Legend object at 0x7fe87164bd30>
```

The model response can much better resemble the given data compared to pure interpolation.

Note: Take-away messages

- (ERT) data inversion is highly ambiguous, particularly for hidden layers
- prior data can help to improve regularization
- structural data can be of great help, but only if extended
- point data improve images, but only locally with smoothness constraints
- geostatistical regularization can extrapolate point data
- incorporation of prior data with geostatistic regularization is best and simple

---

Total running time of the script: (3 minutes 26.761 seconds)

CHAPTER

SEVEN

TUTORIALS

Learn how pyGIMLi can be used for modelling and inversion. The listed tutorials with increasing complexity start with basic functionality such as mesh generation and visualization and dive into the generalized modelling and inversion concepts including managers and frameworks. Check out the *Examples* (page 17) for more specialized demonstrations and interesting case studies.

7.1 Basics

These introductory tutorials cover basic tasks such as meshes, data container, matrices, visualization, and more.

7.2 Meshes

These introductory tutorials cover meshes from simple generation, their elements, visualization, interpolation and more.

7.3 Modelling

Here we collect examples and user stories that illustrate the modelling part of pygimli.

7.4 Inversion

Simple inversions with increasing complexity.

7.4.1 Basics

These introductory tutorials cover basic tasks such as meshes, data container, matrices, visualization, and more.

7.4.1.1 GIMLi Basics

This is the first tutorial where we demonstrate the general use of *GIMLi* in Python, i.e., *pyGIMLi*.

The modelling as well as the inversion part of *GIMLi* often requires a spatial discretization for the domain of interest, the so called GIMLI::Mesh. This tutorial shows some basic aspects of handling a mesh.

First, the library needs to be imported. To avoid name clashes with other libraries we suggest to `import pygimli` and alias it to the simple abbreviation `pg`: CR

```
import pygimli as pg
```

Every part of the c++ namespace GIMLI is bound to python and can be used with the leading `pg`.

For instance get the current version for *GIMLi* with:

```
print(pg.__version__)
```

```
1.3.1+2.g7599abf9
```

Now that we know the name space GIMLI, we can create a first mesh. A mesh is represented by a collection of nodes, cells and boundaries, i.e., geometrical entities.

Note: A regularly spaced mesh consisting of rectangles or hexahedrons is usually called a grid. However, a grid is just a special variant of a mesh so GIMLi treats it the same. The only difference is how they are created.

GIMLi provides a collection of tools for mesh import, export and generation. A simple grid generation is built-in but we also provide wrappers for unstructured mesh generations, e.g., *Triangle*, *Tetgen* and *Gmsh*. To create a 2d grid you need to give two arrays/lists of sample points in x and y direction, in that order, or just numbers.

```
grid = pg.createGrid(x=[-1.0, 0.0, 1.0, 4.0], y=[-1.0, 0.0, 1.0, 4.0])
```

The returned object `grid` is an instance of GIMLI::Mesh and provides various methods for modification and io-operations. General information about the grid can be printed using the simple print() function.

```
print(grid)
```

```
Mesh: Nodes: 16 Cells: 9 Boundaries: 24
```

Or you can access them manually using different methods:

```
print('Mesh: Nodes:', grid.nodeCount(),
      'Cells:', grid.cellCount(),
      'Boundaries:', grid.boundaryCount())
```

```
Mesh: Nodes: 16 Cells: 9 Boundaries: 24
```

You can iterate through all cells of the general type GIMLI::Cell that also provides a lot of methods. Here we list the number of nodes and the node ids per cell:

```
for cell in grid.cells():
    print("Cell", cell.id(), "has", cell.nodeCount(),
          "nodes. Node IDs:", [n.id() for n in cell.nodes()])

print(type(grid.cell(0)))
```

```
Cell 0 has 4 nodes. Node IDs: [0, 1, 5, 4]
Cell 1 has 4 nodes. Node IDs: [1, 2, 6, 5]
Cell 2 has 4 nodes. Node IDs: [2, 3, 7, 6]
Cell 3 has 4 nodes. Node IDs: [4, 5, 9, 8]
Cell 4 has 4 nodes. Node IDs: [5, 6, 10, 9]
Cell 5 has 4 nodes. Node IDs: [6, 7, 11, 10]
Cell 6 has 4 nodes. Node IDs: [8, 9, 13, 12]
Cell 7 has 4 nodes. Node IDs: [9, 10, 14, 13]
Cell 8 has 4 nodes. Node IDs: [10, 11, 15, 14]
<class 'pygimli.core._pygimli_.Quadrangle'>
```

To generate the input arrays `x` and `y`, you can use the built-in GIMLI::Vector (pre-defined with values that are type double as `pg.Vector`), standard python lists or *numpy* arrays, which are widely compatible with *GIMLi* vectors.

```
import numpy as np

grid = pg.createGrid(x=np.linspace(-1.0, 1.0, 10),
                     y=1.0 - np.logspace(np.log10(1.0), np.log10(2.0), 10))
```

We can find that this new `grid` contains

```
print(grid.cellCount())
```

```
81
```

rectangles of type GIMLI::Quadrangle derived from the base type GIMLI::Cell, edges of type GIMLI::Edge, which are boundaries of the general type GIMLI::Boundary.

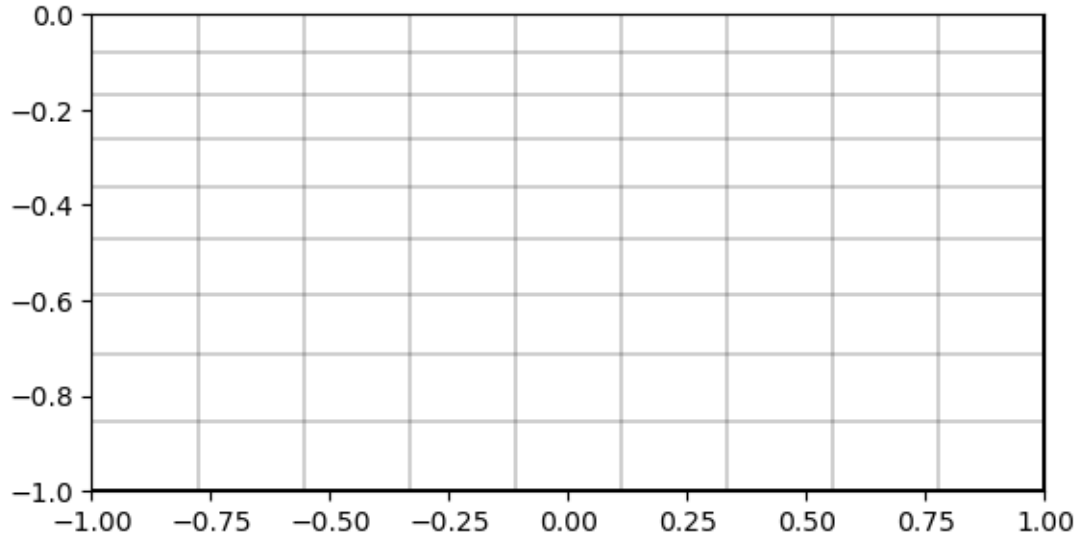
```
print(grid.boundaryCount())
```

```
180
```

The mesh can be saved and loaded in our binary mesh format `.bms`. Or exported into `.vtk` format for 2D or 3D visualization using *Paraview*.

However, we recommend visualizing 2-dimensional content using python scripts that provide better exports to graphics files (e.g., png, pdf, svg). In *pygimli* we provide some basic post-processing routines using the *matplotlib* visualization framework. The main visualization call is *pygimli.viewer.show* (page 508) which is sufficient for most meshes, fields, models and streamline views.

```
pg.viewer.show(grid)
pg.wait()
```



For more control you can also use the appropriate draw methods *pygimli.viewer.mpl.drawMesh* (page 485).

7.4.1.2 The DataContainer class

Data are often organized in a data container storing the individual data values as well as any description how they were obtained, e.g. the geometry of source and receivers.

So first a data container holds vectors like in a dictionary, however, all of them need to have the same length defined by the .size() method. Assume we want to store Vertical Electrical Sounding (VES) data.

```
# We start off with the typical imports
import numpy as np
import matplotlib.pyplot as plt
import pygimli as pg
```

We define logarithmically equidistant AB/2 spacings

```
ab2 = np.logspace(0, 3, 11)
print(ab2)
```

```
[   1.            1.99526231    3.98107171    7.94328235   15.84893192
   31.6227766    63.09573445  125.89254118  251.18864315  501.18723363
 1000.        ]
```

We create an empty data container

```
ves = pg.DataContainer()
print(ves)
```

```
Data: Sensors: 0 data: 0, nonzero entries: []
```

We feed it into the data container just like in a dictionary.

```
ves["ab2"] = ab2
ves["mn2"] = ab2 / 3
print(ves)
```

```
Data: Sensors: 0 data: 11, nonzero entries: ['ab2', 'mn2']
```

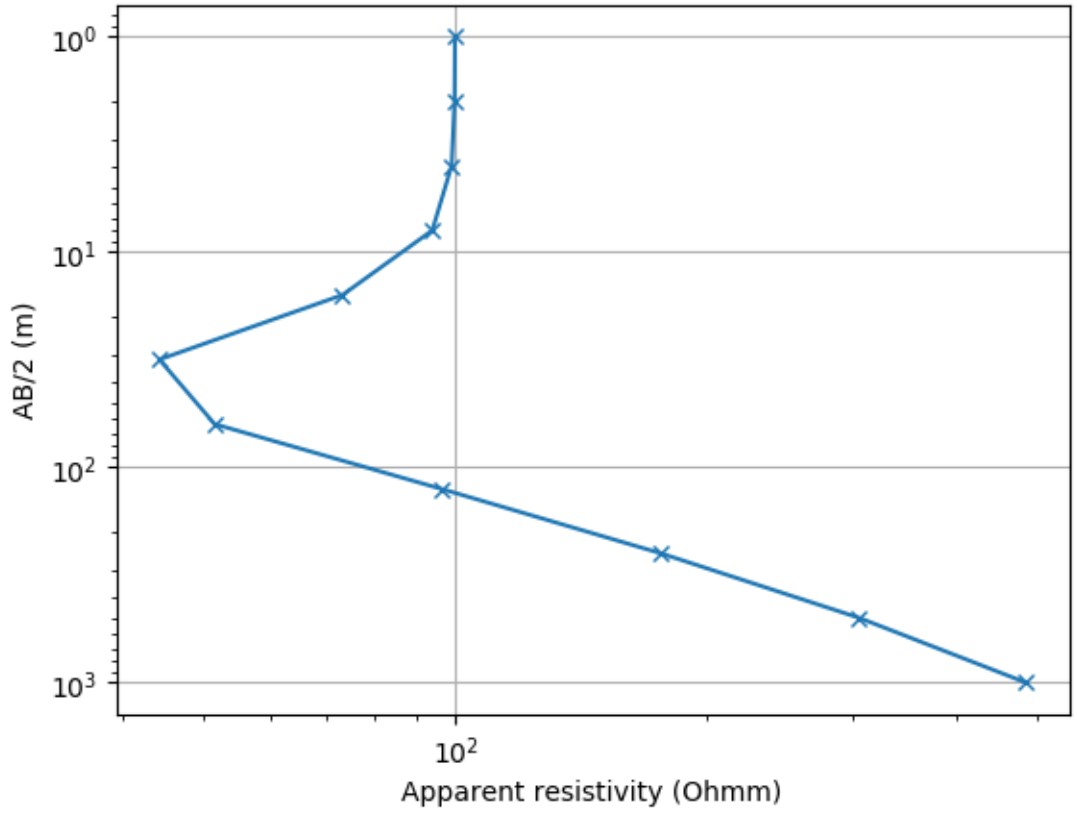
We now want to do a VES simulation and use the VES Manager for this task.

```
mgr = pg.physics.ert.VESManager()
model = [10, 10, 100, 10, 1000]
ves["rhoa"] = mgr.simulate(model, ab2=ves["ab2"], mn2=ves["mn2"])
print(ves)
```

```
Data: Sensors: 0 data: 11, nonzero entries: ['ab2', 'mn2', 'rhoa']
```

We can plot the sounding curve by assessing its fields

```
fig, ax = plt.subplots()
ax.loglog(ves["rhoa"], ves["ab2"], "x-");
ax.set_ylim(ax.get_ylim()[::-1])
ax.grid(True)
ax.set_xlabel("Apparent resistivity (Ohmm)")
ax.set_ylabel("AB/2 (m)");
```




```
Text(24.000000000000007, 0.5, 'AB/2 (m)')
```

A data container can be saved to disk

```
ves.save("ves.data")
print(open("ves.data").read())
```

```
0
# x y z
11
# ab2 mn2 rhoa valid
1.00000000000000e+00    3.33333333333333e-01    9.99842595368026e+01    0
1.99526231496888e+00    6.65087438322960e-01    9.98766666391267e+01    0
3.98107170553497e+00    1.32702390184499e+00    9.90710153790337e+01    0
7.94328234724281e+00    2.64776078241427e+00    9.39205974843448e+01    0
1.58489319246111e+01    5.28297730820371e+00    7.33018913900196e+01    0
3.16227766016838e+01    1.05409255338946e+01    4.41998154306538e+01    0
6.30957344480193e+01    2.10319114826731e+01    5.14873009414442e+01    0
1.25892541179417e+02    4.19641803931389e+01    9.61269991281245e+01    0
2.51188643150958e+02    8.37295477169860e+01    1.76491691336385e+02    0
5.01187233627272e+02    1.67062411209091e+02    3.05345771380821e+02    0
1.00000000000000e+03    3.33333333333333e+02    4.83494826609041e+02    0
0
```

The data are (along with a valid flat) in the second section. We can add arbitrary entries to the data container but define what to save.

```
ves["flag"] = pg.Vector(ves["rhoa"] > 100) + 1
print(ves)
ves.save("ves.data", "ab2 mn2 rhoa")
print(open("ves.data").read())
```

```
Data: Sensors: 0 data: 11, nonzero entries: ['ab2', 'flag', 'mn2', 'rhoa']
0
# x y z
0
# ab2 mn2 rhoa
0
```

We can mask or unmask the data with a boolean vector.

```
ves.markValid(ves["ab2"] > 2)
ves.save("ves.data", "ab2 rhoa")  # note that only valid data are saved!
print(ves)
```

```
Data: Sensors: 0 data: 11, nonzero entries: ['ab2', 'flag', 'mn2', 'rhoa',
↪'valid']
```

Data containers with indexed data

Assume we have data associate with a transmitter, receivers and a property U. The transmitter (Tx) and receiver (Rx) positions are stored separately and we refer them with an Index (integer). Therefore we define these fields index.

```
data = pg.DataContainer()
data.registerSensorIndex("Tx")
data.registerSensorIndex("Rx")
print(data)
```

```
Data: Sensors: 0 data: 0, nonzero entries: ['Rx', 'Tx']
```

Create a list of 10 sensors, 2m spacing

```
for x in np.arange(10):
    data.createSensor([x*2, 0])

print(data)
```

```
Data: Sensors: 10 data: 0, nonzero entries: ['Rx', 'Tx']
```

We want to use all of them (and two more!) as receivers and a constant transmitter of number 2.

```
data["Rx"] = np.arange(12)
# data["Tx"] = np.arange(9) # does not work as size matters!
data["Tx"] = pg.Vector(data.size(), 2)
print(data)
data.save("TxRx.data")
print(open("TxRx.data").read())
```

```
Data: Sensors: 10 data: 12, nonzero entries: ['Rx', 'Tx']
10
# x y z
0       0       0
2       0       0
4       0       0
6       0       0
8       0       0
10      0       0
12      0       0
14      0       0
16      0       0
18      0       0
12
# Rx Tx valid
1       3       0
2       3       0
3       3       0
4       3       0
```

(continues on next page)

(continued from previous page)

```
5       3       0
6       3       0
7       3       0
8       3       0
9       3       0
10      3       0
11      3       0
12      3       0
0
```

Again, we can mark the data validity.

```
data.markValid(data["Rx"] >= 0)
print(data["valid"])
print(data["Rx"])
```

```
12 [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
[ 0  1  2  3  4  5  6  7  8  9 10 11]
```

or check the data validity automatically.

```
data.checkDataValidity()
print(data["valid"])
data.removeInvalid()
print(data)
# data.save("TxRx.data");
```

```
10 [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
Data: Sensors: 10 data: 10, nonzero entries: ['Rx', 'Tx', 'valid']
```

Suppose we want to compute the horizontal offset between Tx and Rx. We first retrieve the x position and use Tx and Rx as indices.

```
sx = pg.x(data)
data["dist"] = np.abs(sx[data["Rx"]] - sx[data["Tx"]])
print(data["dist"])
```

```
10 [4.0, 2.0, 0.0, 2.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0]
```

It might be useful to only use data where transmitter is not receiver.

```
data.markInvalid(data["Rx"] == data["Tx"])
print(data)
# data.save("TxRx.data");
```

```
Data: Sensors: 10 data: 10, nonzero entries: ['Rx', 'Tx', 'dist', 'valid']
```

They are still there but can be removed.

```
data.removeInvalid()
print(data)
```

```
Data: Sensors: 10 data: 9, nonzero entries: ['Rx', 'Tx', 'dist', 'valid']
```

At any stage we can create a new sensor

```
data.createSensor(data.sensors()[-1])
print(data)  # no change
```

```
Data: Sensors: 10 data: 9, nonzero entries: ['Rx', 'Tx', 'dist', 'valid']
```

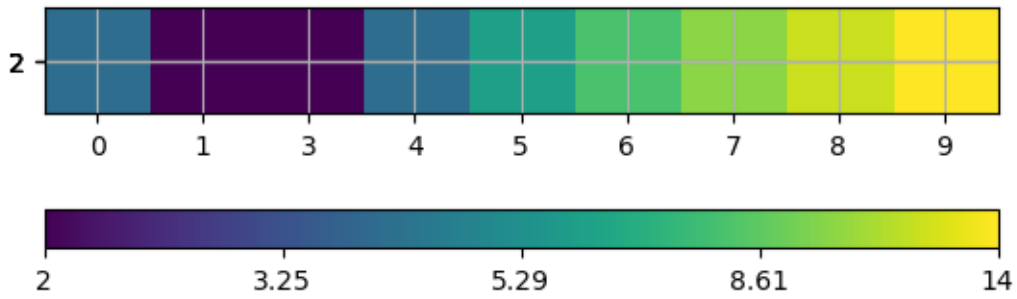
, however, not at a position where already a sensor is

```
data.createSensor(data.sensors()[-1]+0.1)
print(data)
# data.save("TxRx.data")
```

```
Data: Sensors: 11 data: 9, nonzero entries: ['Rx', 'Tx', 'dist', 'valid']
```

Any DataContainer (indexed or not) can be visualized as matrix plot

```
pg.viewer.mpl.showDataContainerAsMatrix(data, "Rx", "Tx", "dist");
```




```
found 9 x values
found 1 y values
Could not set x/y label:  9 [0.0, 1.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0] 9 [2.0,
↪ 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0]

(<matplotlib.axes._subplots.AxesSubplot object at 0x7fe8da91b250>, <matplotlib.
↪colorbar.Colorbar object at 0x7fe89995b580>)
```

Instead of marking and filtering one can remove directly

```
print(data["dist"])
data.remove(data["dist"] > 11)
print(data["dist"])
print(data)
```

```
9 [4.0, 2.0, 2.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0]
7 [4.0, 2.0, 2.0, 4.0, 6.0, 8.0, 10.0]
Data: Sensors: 11 data: 7, nonzero entries: ['Rx', 'Tx', 'dist', 'valid']
```

Similar to the nodes of a mesh, the sensor positions can be changed.

```
data.scale([2, 1])
data.translate([10, 0])
data.save("TxRx.data")
```

```
1
```

Suppose a receiver has not been used

```
data["Rx"][5] = data["Rx"][4]
data.removeUnusedSensors()
print(data)
```

```
Data: Sensors: 8 data: 7, nonzero entries: ['Rx', 'Tx', 'dist', 'valid']
```

or any measurement with it (as Rx or Tx) is corrupted

```
data.removeSensorIdx(2)
print(data)
```

```
Data: Sensors: 0 data: 0, nonzero entries: ['Rx', 'Tx']
```

There are specialized data containers with predefined indices like pg.DataContainerERT having indices for a, b, m and b electrodes. One can also add alias translators like C1, C2, P1, P2, so that dataERT[“P1”] will return dataERT[“m”]

7.4.1.3 Matrices

There is a large number of Matrix types that are all derived from the base class `MatrixBase`. They do not have to store elements but can be logical or wrappers. They just have to provide the functions `A.cols()`, `A.rows()` (column and row numbers), `A.mult(x)` $=Acdot x$ and `A.transMult(y)` $=A^Tcdot y$.

```
# We start off with the typical imports
import numpy as np
import pygimli as pg
```

Dense matrix `Matrix`

All elements are stored column-wise, i.e. all rows `A[i]` are of type `pg.Vector`. This matrix is used for storing dense data (like ERT Jacobians) and doing simple algebra.

```
A = pg.Matrix(3, 4)
A[0, 0] = 1
A[2, 2] = -1
A[1] = np.arange(4)
print(A)
x = np.arange(4) + 5
A*x
```

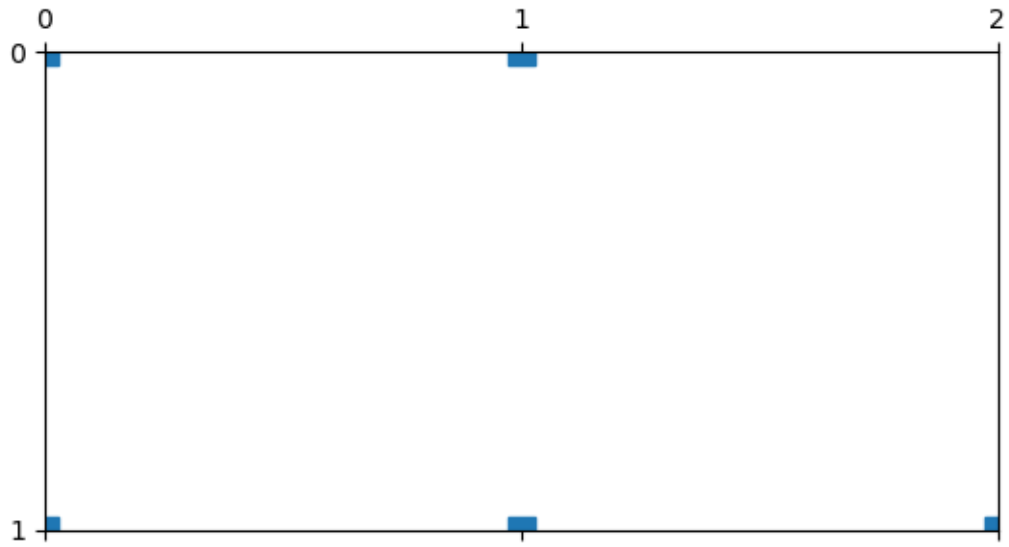
```
RMatrix: 3 x 4
4 [1.0, 0.0, 0.0, 0.0]
4 [0.0, 1.0, 2.0, 3.0]
4 [0.0, 0.0, -1.0, 0.0]


3 [5.0, 44.0, -7.0]
```

Exists also as complex matrix under `pg.matrix.CMatrix`.

Index-based sparse matrix `SparseMapMatrix`

Sparse matrix (most elements are zero) with single access and. Typical for traveltime Jacobian (only certain cells covered by individual rays) or constraint matrices.

```
A = pg.SparseMapMatrix(2, 3)
A.setVal(0, 0, -1)
A.setVal(0, 1, +1)  # first-order derivative
A.setVal(1, 0, -1)
A.setVal(1, 1, +2)
A.setVal(1, 2, -1)  # second-order derivative
x = [10, 12, 13]
print(A * x)
ax, _ = pg.show(A)
```




```
2 [2.0, 1.0]
```

Exists also as complex-valued variant `pg.matrix.CSparseMapMatrix`.

Column-compressed matrix `SparseMatrix`

Used for numerical approximation of partial differential equations like finite-element or finite volume. Not typically used unless efficiency is of importance. It exists also complex-valued as `pg.matrix.CSparseMatrix`.

Diagonal matrices

First, there is an identity matrix `IdentityMatrix`. No elements stored at all. Important for constraint matrices when combined into `BlockMatrix` (see below). More generally, there is a diagonal matrix `DiagonalMatrix` where only its diagonal is stored as vector.

```
A = pg.matrix.IdentityMatrix(3)  # , 2.0)
x = [10, 11, 12]
A*x
```

```
3 [10.0, 11.0, 12.0]
```

Weighted matrices

`MultLeftMatrix`/`MultRightMatrix`/`MultLeftRightMatrix`

Often, matrices are weighted from either side by a vector, e.g. data error weighting of the Jacobian matrix, data and model transformations, or weighting individual smoothness parts according to the roughness so that only the weighting is changed and not the matrix.

```
A = pg.Matrix(3, 4)
A += 1
w = [2, 3, -1]
B = pg.matrix.MultLeftMatrix(A, w)
x = np.arange(4)
print(A*x)
print(B*x)
```

```
3 [6.0, 6.0, 6.0]
3 [12.0, 18.0, -6.0]
```

Combinations of matrices

Logical matrices can combine different other matrices (of arbitrary type) avoiding double memory storage by multiplication (`Mult2Matrix`) or addition (`Add2Matrix`).

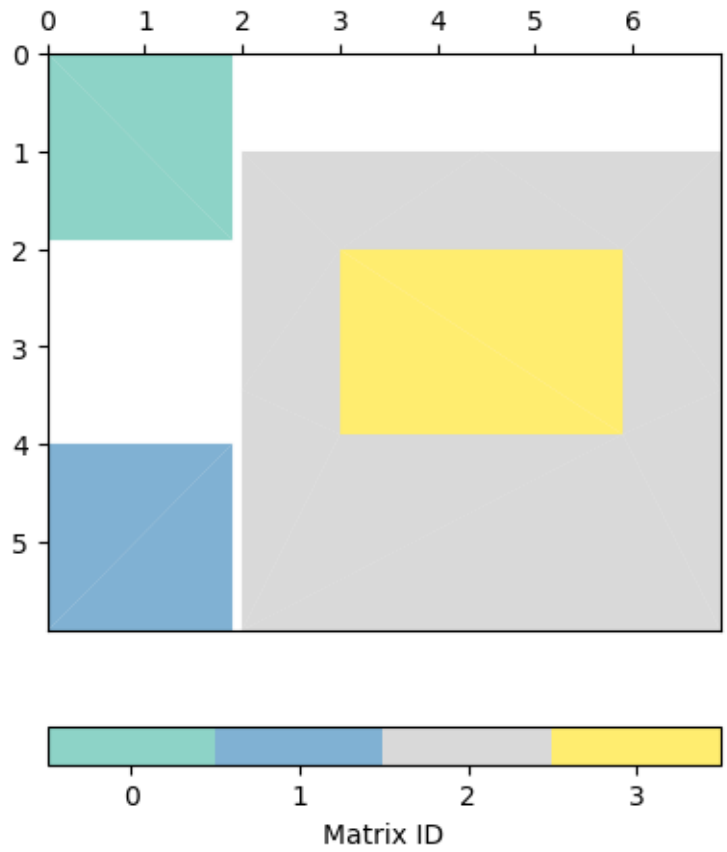
```
A = pg.Matrix(2, 3)
A += 1
B = pg.SparseMapMatrix(3, 4)
C = pg.matrix.Mult2Matrix(A, B)
x = np.arange(4)
C * x
```

```
2 [0.0, 0.0]
```

Block matrices

The most important type is the `BlockMatrix`, where arbitrary matrices are combined into a logical matrix. This is of importance for inversion frameworks: * joint inversion: Jacobian matrices are concatenated * combination of different constrains: combining different regularization * laterally, spatially or temporally constrained inversion: regularization between model cells of each frame but also between the frames Note that the matrices only have to be defined once and can appear multiply.

```
A = pg.BlockMatrix()
A1 = pg.Matrix(2, 2)
A.addMatrix(A1, 0, 0)
A.addMatrix(A1, 4, 0, scale=2.0)
A2 = pg.matrix.IdentityMatrix(5)
A.addMatrix(A2, 1, 2)
A3 = pg.SparseMapMatrix(2, 3)
A.addMatrix(A3, 2, 3)
print(A)
ax, _ = pg.show(A)
```




```
pg.matrix.BlockMatrix of size 6 x 7 consisting of 4 submatrices.
```

Matrix combinations

There are also simpler types of matrix combinations: * `H2Matrix`/`V2Matrix`: two matrices below/next to each other * `HNMatrix`/`VNMatrix`: one matrix repeated N times horizontally/vertically * `NDMatrix`: block diagonal matrix

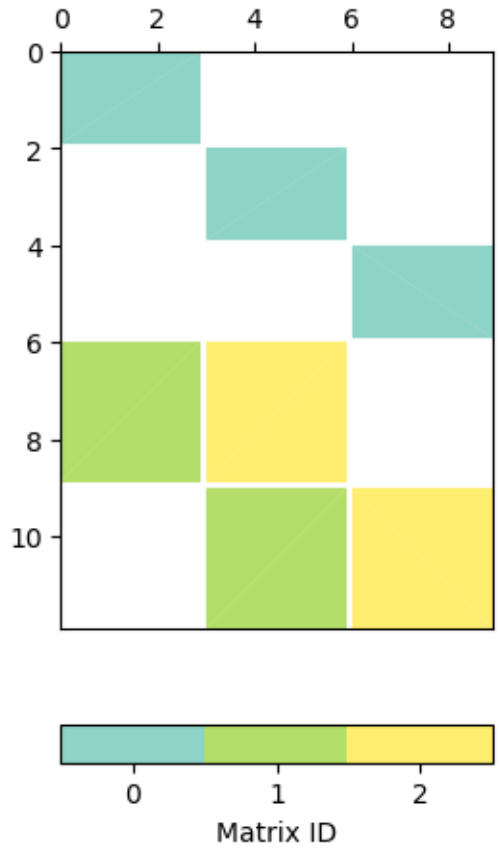
Matrix wrappers

TransposedMatrix avoids transposing any matrix by exchanging left/right mult. SquaredMatrix keeps only the matrix A but works as A^T @ A SquaredTransposeMatrix keeps only the matrix A but works as A @ A.T RealNumpyMatrix holds a real-valued numpy array ComplexNumpyMatrix holds a complex-valued numpy array in a real pg matrix NumpyMatrix

Matrix generators

Often, several matrices or even the same one have to be combined. RepeatHMatrix, RepeatVMatrix, RepeatDMatrix hold a single matrix that is repeated horizontally, vertically or diagonally. NDMatrix, FrameConstraintMatrix is a special generator for constraining cells of every (e.g. timelapse) frame and moreover the frames with each other.

```
F = pg.matrix.FrameConstraintMatrix(A3, 3)
ax, _ = pg.show(F)
print(F)
```




```
pg.matrix.BlockMatrix of size 12 x 9 consisting of 7 submatrices.
```

Geostatistical constraint matrix

For geostatistical constraints, a correlation matrix is computed using correlation lengths and angles to define their directions. To access its inverse root in a way that avoids matrix inversion, an eigenvalue decomposition is done and the eigenvalues D and -vectors Q are stored so that the operator

$$C^{-0.5} \cdot x = Q \cdot D^{-0.5} \cdot Q^T \cdot x$$

Jordi, C., Doetsch, J., Günther, T., Schmelzbach, C. & Robertsson, J.O.A. (2018): Geostatistical regularisation operators for geophysical inverse problems on irregular meshes. Geophysical Journal International 213, 1374- 1386, doi:10.1093/gji/ggy055.

7.4.2 Meshes

These introductory tutorials cover meshes from simple generation, their elements, visualization, interpolation and more.

7.4.2.1 The anatomy of a pyGIMLi mesh

In this tutorial we look at the anatomy of a GIMLI::Mesh. Although the example is simplisitic and two-dimensional, the individual members of the Mesh class and their methods can be inspected similarly for more complex meshes in 2D and 3D. This example is heavily inspired by the anatomy of a matplotlib plot, which we can also highly recommend for visualization with the *pygimli.viewer.mpl* (page 475).

We start by importing matplotlib and defining some helper functions for plotting.

```
import matplotlib.pyplot as plt

from matplotlib.patches import Circle
from matplotlib.patheffects import withStroke

def circle(x, y, text=None, radius=0.15, c="blue"):
    circle = Circle((x, y), radius, clip_on=False, zorder=10, linewidth=1,
                    edgecolor='black', facecolor=(0, 0, 0, .0125),
                    path_effects=[withStroke(linewidth=5, foreground='w')])
    ax.add_artist(circle)
    ax.plot(x, y, color=c, marker=".")
    ax.text(x, y - (radius + 0.05), text, backgroundcolor="white",
            ha='center', va='top', weight='bold', color=c)
```

We now import pygimli and create a simple grid/mesh with 3x3 cells.

```
import pygimli as pg
m = pg.createGrid(4,4)
```

The following code creates the main plot and shows how the different mesh entities can be called.

```
# Create matplotlib figure and set size
fig, ax = plt.subplots(figsize=(8,8), dpi=90)
ax.set_title("The anatomy of a pyGIMLi mesh", fontweight="bold")

# Visualize mesh with the generic pg.show command
pg.show(m, ax=ax)

# Number all cells
for cell in m.cells():
    node = cell.allNodes()[2]
    ax.text(node.x() - 0.5, node.y() - .05, "m.cell(%d)" % cell.id(),
            fontsize=11, ha="center", va="top", color="grey")

# Mark horizontal and vertical extent
circle(m.xmin(), m.xmax(), "m.xmin(),\nm.ymax() ", c="grey")
```

(continues on next page)

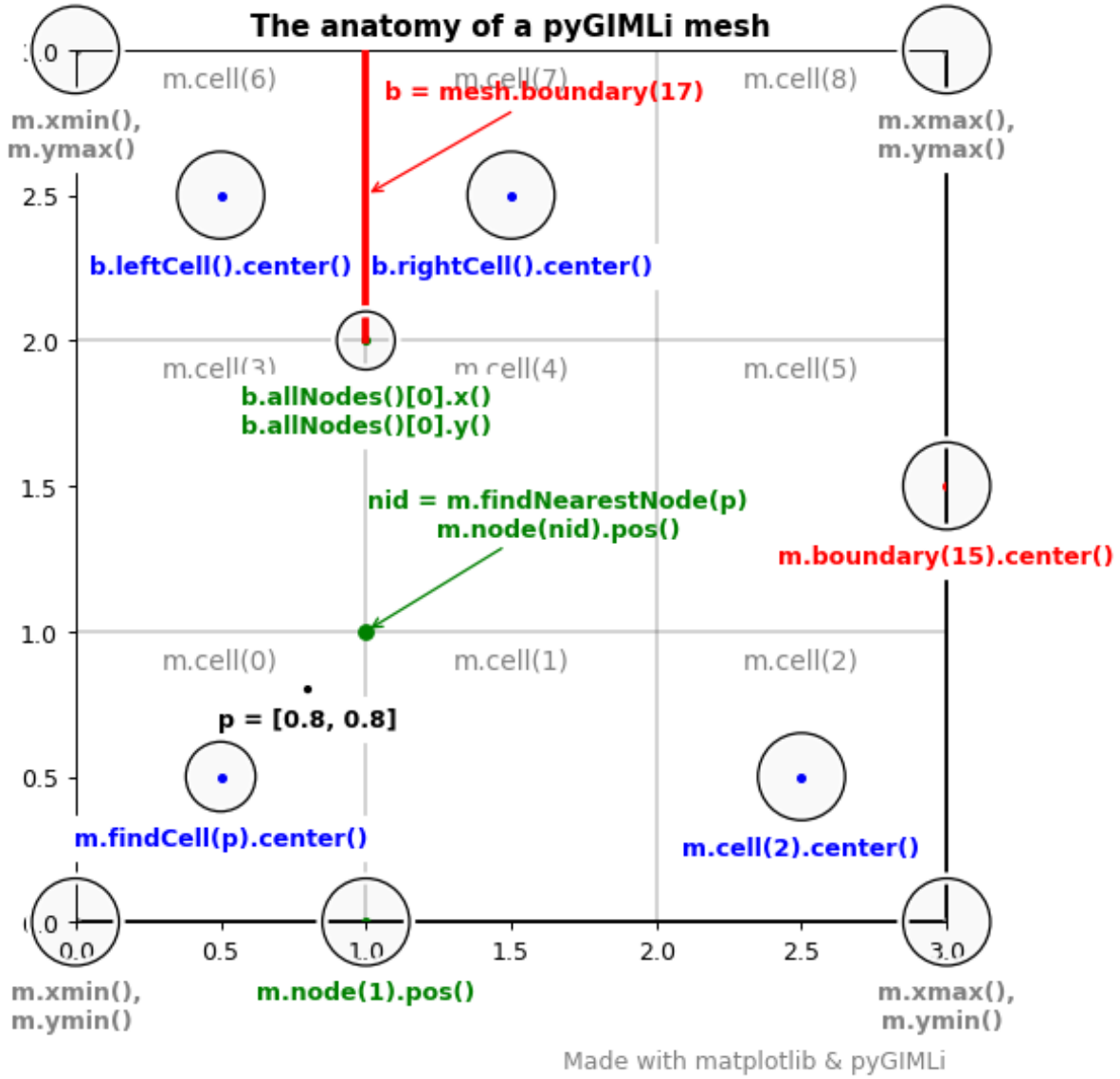
(continued from previous page)

```
circle(m.xmin(), m.ymin(), "m.xmin(),\nm.ymin() ", c="grey")
circle(m.xmax(), m.ymin(), "m.xmax(),\nm.ymin() ", c="grey")
circle(m.xmax(), m.ymax(), "m.xmax(),\nm.ymax() ", c="grey")

# Mark center of a cell
cid = 2
circle(m.cell(cid).center().x(), m.cell(cid).center().y(),
       "m.cell(%d).center()" % cid)

# Mark node
circle(m.node(1).x(), m.node(1).y(), "m.node(1).pos()", c="green")

# Find cell in which point p lies
p = [0.8, 0.8]
circle(p[0], p[1], "p = %s" % p, radius=0.01, c="black")

cell = m.findCell(p)
circle(cell.center().x(), cell.center().y(), "m.findCell(p).center()",
       radius=0.12)

# Find closest node to point p
nid = m.findNearestNode(p)
n = m.node(nid)
ax.plot(n.x(), n.y(), "go")
ax.annotate('nid = m.findNearestNode(p)\nm.node(nid).pos()', xy=(n.x(), n.y()),
            xycoords='data', xytext=(80, 40), textcoords='offset points',
            ha="center", weight='bold', color="green",
            arrowprops=dict(arrowstyle='->',
                            connectionstyle="arc",
                            color="green"))

# Mark boundary center
bid = 15
boundary_center = m.boundary(bid).center()
circle(boundary_center.x(), boundary_center.y(),
       "m.boundary(%d).center()" % bid, c="red")

# Mark boundary together with left and right cell
bid = 17
b = m.boundaries()[bid] # equivalent to mesh.boundary(17)
n1 = b.allNodes()[0]
n2 = b.allNodes()[1]
ax.plot([n1.x(), n2.x()], [n1.y(), n2.y()], "r-", lw=3, zorder=10)
ax.annotate('b = mesh.boundary(%d)' % bid, xy=(b.center().x(), b.center().y()),
            xycoords='data', xytext=(8, 40), textcoords='offset points',
            weight='bold', color="red",
            arrowprops=dict(arrowstyle='->',
                            connectionstyle="arc",
                            color="red"))
```

(continues on next page)

(continued from previous page)

```
circle(n1.x(), n1.y(), "b.allNodes()[0].x()\nb.allNodes()[0].y()",
       radius=0.1, c="green")

# Mark neighboring cells
left = b.leftCell()
right = b.rightCell()
circle(left.center().x(), left.center().y(),
       "b.leftCell().center()", c="blue")
circle(right.center().x(), right.center().y(),
       "b.rightCell().center()", c="blue")

ax.text(3.0, -0.5, "Made with matplotlib & pyGIMLi",
        fontsize=10, ha="right", color='.5')
fig.tight_layout(pad=5.7)
pg.wait()
```



7.4.2.2 The mesh class

The mesh class holds either geometric definitions (piece-wise linear complex - PLC) or discretizations of the subsurface. It contains of nodes, boundaries (edges in 2D, faces in 3D) and cells (triangles, quadrangles in 2D, hexahedra or tetrahedra in 3D) with associated markers and arbitrary data for nodes or cells.

https://www.pygimli.org/_tutorials_auto/1_basics/plot_2-anatomy_of_a_mesh.html gives a good overview on the properties of a mesh. Here we demonstrate how to create and manipulate meshes from the scratch.

We start off with the typical imports

```
import numpy as np
import pygimli as pg
```

We construct a mesh by creating nodes and cells.

```
mesh = pg.Mesh(2)  # 2D
n11 = mesh.createNode((0.0, 0.0))
n12 = mesh.createNode((1.0, 0.0))
n13 = mesh.createNode((2.0, 0.0))
n21 = mesh.createNode((0.0, 1.0))
n22 = mesh.createNode((1.0, 1.0), marker=1)
n23 = mesh.createNode((2.0, 1.0))
n31 = mesh.createNode((0.5, 1.7))
n32 = mesh.createNode((1.5, 1.7))
mesh.createQuadrangle(n11, n12, n22, n21, marker=4)
mesh.createQuadrangle(n12, n13, n23, n22, marker=4)
mesh.createTriangle(n21, n22, n31, marker=3)
mesh.createTriangle(n22, n23, n32, marker=3)
mesh.createTriangle(n31, n32, n22, marker=3)  # leave out
print(mesh)
```

```
Mesh: Nodes: 8 Cells: 5 Boundaries: 0
```

The function createNeighborInfos adds boundaries to nodes and cells.

```
mesh.createNeighborInfos()
print(mesh)
```

```
Mesh: Nodes: 8 Cells: 5 Boundaries: 12
```

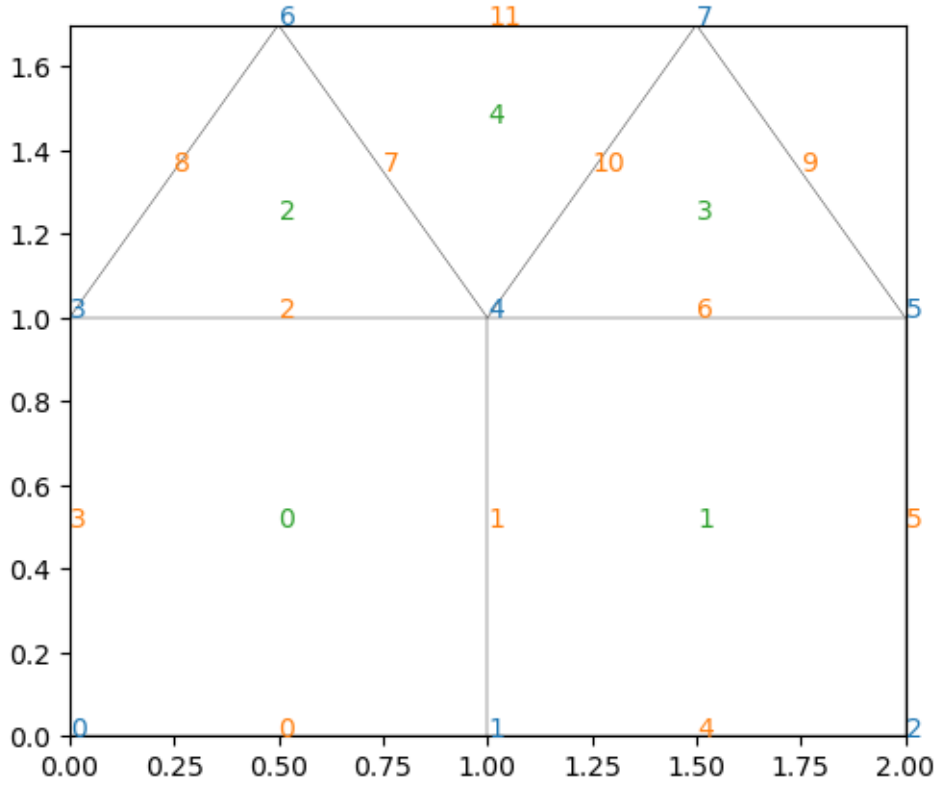
We can look only at a given mesh and add matplotlib objects to the plot

```
ax, _ = pg.show(mesh, showMesh=True)

for n in mesh.nodes():
    ax.text(n.x(), n.y(), str(n.id()), color="C0")
for b in mesh.boundaries():
    ax.text(b.center().x(), b.center().y(), str(b.id()), color="C1")
```

(continues on next page)

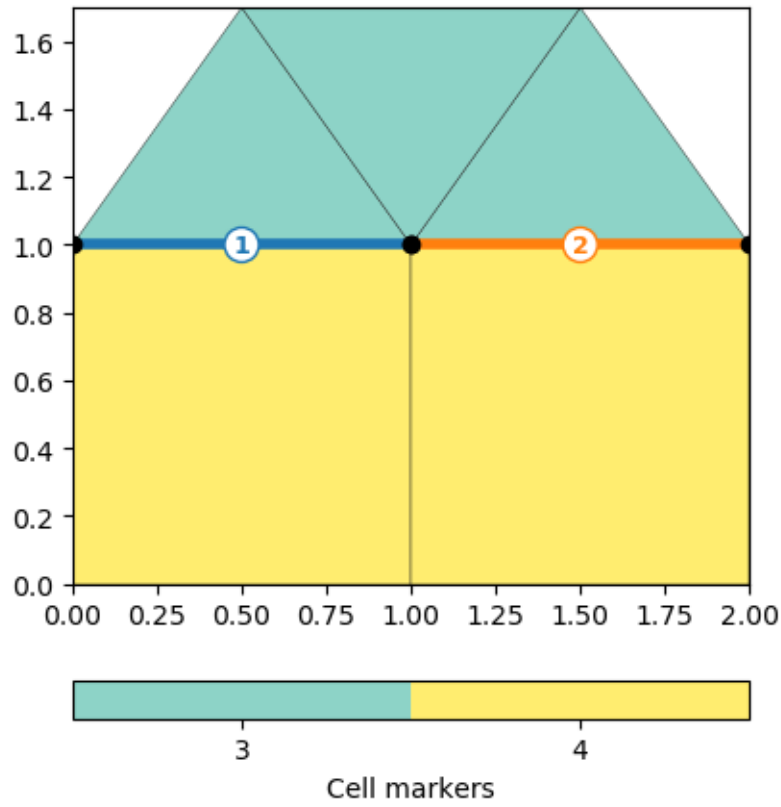
(continued from previous page)

```
for c in mesh.cells():
    ax.text(c.center().x(), c.center().y(), str(c.id()), color="C2")
```



Or we can change and show all (cell or boundary) markers of a mesh

```
mesh.boundary(2).setMarker(1)
mesh.boundary(6).setMarker(2)
ax, _ = pg.show(mesh, markers=True, showMesh=True)
```



We can iterate over all nodes, cells, or boundaries and retrieve or change their properties like node positions,

```
for n in mesh.nodes():
    print(n.id(), n.pos())
```

```
0 RVector3: (0.0, 0.0, 0.0)
1 RVector3: (1.0, 0.0, 0.0)
2 RVector3: (2.0, 0.0, 0.0)
3 RVector3: (0.0, 1.0, 0.0)
4 RVector3: (1.0, 1.0, 0.0)
5 RVector3: (2.0, 1.0, 0.0)
6 RVector3: (0.5, 1.7, 0.0)
7 RVector3: (1.5, 1.7, 0.0)
```

find all nodes of a given cell or find the neighbors,

```
for c in mesh.cells():
    print(c.id(), len(c.nodes()), c.center())
```

```
0 4 RVector3: (0.5, 0.5, 0.0)
1 4 RVector3: (1.5, 0.5, 0.0)
2 3 RVector3: (0.5, 1.2333333333333334, 0.0)
3 3 RVector3: (1.5, 1.2333333333333334, 0.0)
4 3 RVector3: (1.0, 1.4666666666666668, 0.0)
```

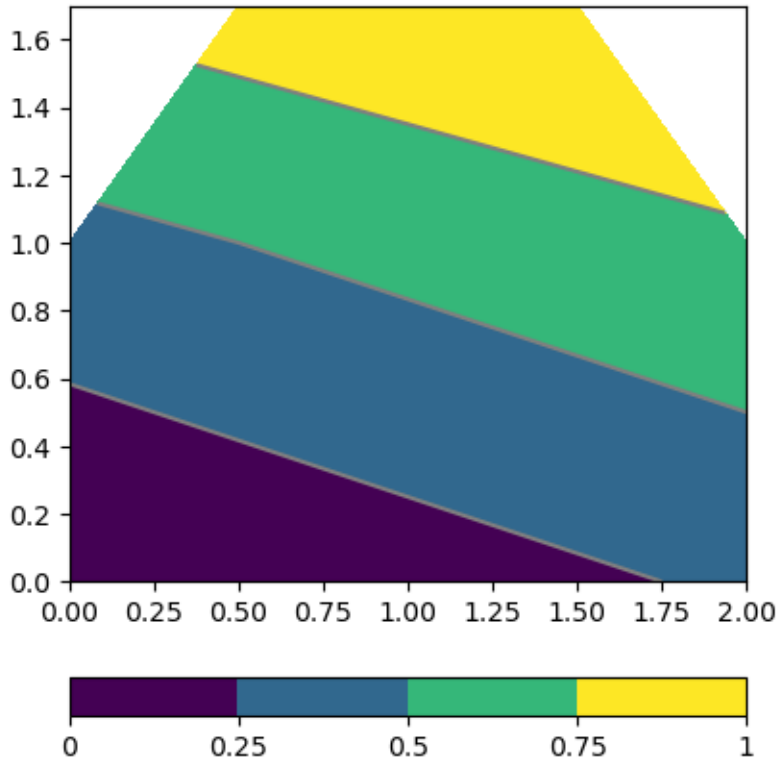
or find the neighboring cells for all inner boundaries.

```
for b in mesh.boundaries():
    print(b.id(), ": Nodes:", b.ids(), end=" ")
    if not b.outside():
        left = b.leftCell()
        right = b.rightCell()
        print(left.id(), right.id())
```

```
0 : Nodes: 2 [0, 1] 1 : Nodes: 2 [1, 4] 0 1
2 : Nodes: 2 [4, 3] 0 2
3 : Nodes: 2 [3, 0] 4 : Nodes: 2 [1, 2] 5 : Nodes: 2 [2, 5] 6 : Nodes: 2 [5, 4]␣
↪1 3
7 : Nodes: 2 [4, 6] 2 4
8 : Nodes: 2 [6, 3] 9 : Nodes: 2 [5, 7] 10 : Nodes: 2 [7, 4] 3 4
11 : Nodes: 2 [6, 7]
```

We visualize some related property and attribute it the mesh cells.

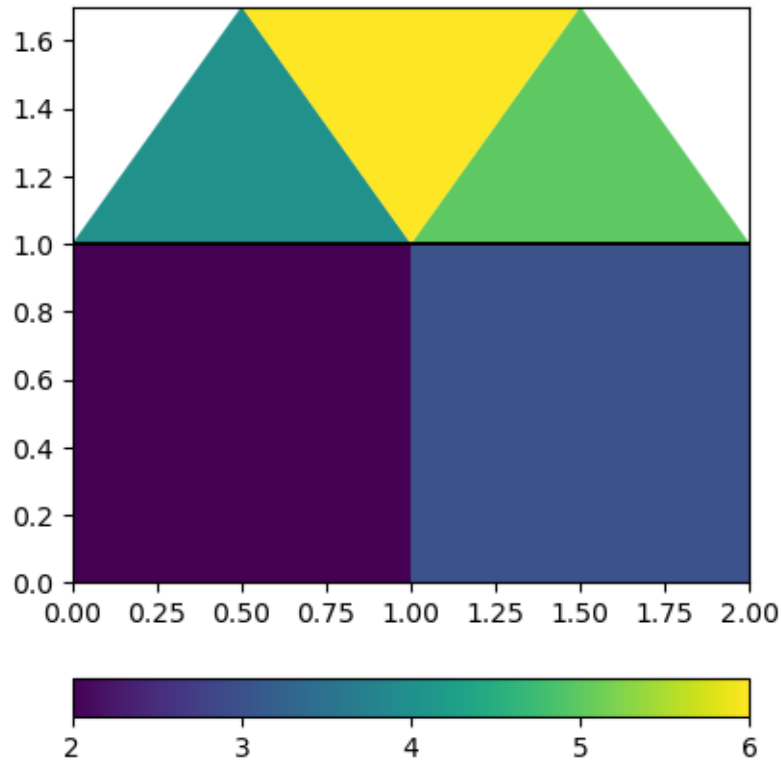
```
voltage = np.arange(mesh.nodeCount()) + 10
ax, cb = pg.show(mesh, voltage)
```



Node-based data are typically shown in form of contour lines.

We can add this (node-based) vector to the mesh as a property. The same can be done for cell-based properties.

```
mesh["voltage"] = voltage
mesh["velocity"] = np.arange(mesh.cellCount()) + 2
ax, cb = pg.show(mesh, "velocity")
```



Cell-based data are, on the other hand, valid for the whole cell, which is why they are typically shown by filled patches.

The VTK format can save these properties along with the mesh, point data like the voltage under POINT_DATA and cell data like velocity under CELL_DATA. It is particularly suited to save inversion results in one file. 3D vtk files can be nicely visualized by a number of programs, e.g. Paraview.

```
mesh.exportVTK("mesh.vtk")
with open("mesh.vtk") as f:
    print(f.read())
```

```
# vtk DataFile Version 3.0
d-2__ created by libgimli-v1.3.1-2-g7599abf9
ASCII
DATASET UNSTRUCTURED_GRID
POINTS 8 double
0       0       0
1       0       0
2       0       0
0       1       0
1       1       0
2       1       0
0.5     1.7     0
1.5     1.7     0
CELLS 5 22
4       0       1       4       3
4       1       2       5       4
3       3       4       6
```

(continues on next page)

(continued from previous page)

```
3       4       5       7
3       6       7       4
CELL_TYPES 5
9 9 5 5 5
CELL_DATA 5
SCALARS Marker double 1
LOOKUP_TABLE default
4 4 3 3 3
SCALARS velocity double 1
LOOKUP_TABLE default
2 3 4 5 6
POINT_DATA 8
SCALARS voltage double 1
LOOKUP_TABLE default
10 11 12 13 14 15 16 17
```

7.4.2.3 Mesh interpolation

In this tutorial, we look at the mesh interpolation options in GIMLi. Although the example shown here is in 2D, the same routines can be applied when converting 3D data to a 2D mesh for instance.

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

import pygimli as pg
from pygimli.viewer.mpl import drawMesh, drawModel
```

Create coarse and fine mesh with data

```
def create_mesh_and_data(n):
    nc = np.linspace(-2.0, 0.0, n)
    mesh = pg.meshtools.createMesh2D(nc, nc)
    mcx = pg.x(mesh.cellCenter())
    mcy = pg.y(mesh.cellCenter())
    data = np.cos(1.5 * mcx) * np.sin(1.5 * mcy)
    return mesh, data

coarse, coarse_data = create_mesh_and_data(5)
fine, fine_data = create_mesh_and_data(20)
```

Interpolate data to different meshes

We define two functions taking the input mesh, the input data and the output mesh as parameters and return the input data interpolated to the output mesh.

```
def nearest_neighbor_interpolation(inmesh, indata, outmesh, nan=99.9):
    """ Nearest neighbor interpolation. """
    outdata = []
    for pos in outmesh.cellCenters():
        cell = inmesh.findCell(pos)
        if cell:
            outdata.append(indata[cell.id()])
        else:
            outdata.append(nan)
    return outdata


def linear_interpolation(inmesh, indata, outmesh):
    """ Linear interpolation using `pg.interpolate()` """
    outdata = pg.Vector()  # empty
    pg.interpolate(srcMesh=inmesh, inVec=indata,
                   destPos=outmesh.cellCenters(), outVec=outdata)

    # alternatively you can use the interpolation matrix
    outdata = inmesh.interpolationMatrix(outmesh.cellCenters()) * \
              pg.core.cellDataToPointData(inmesh, indata)
    return outdata
```

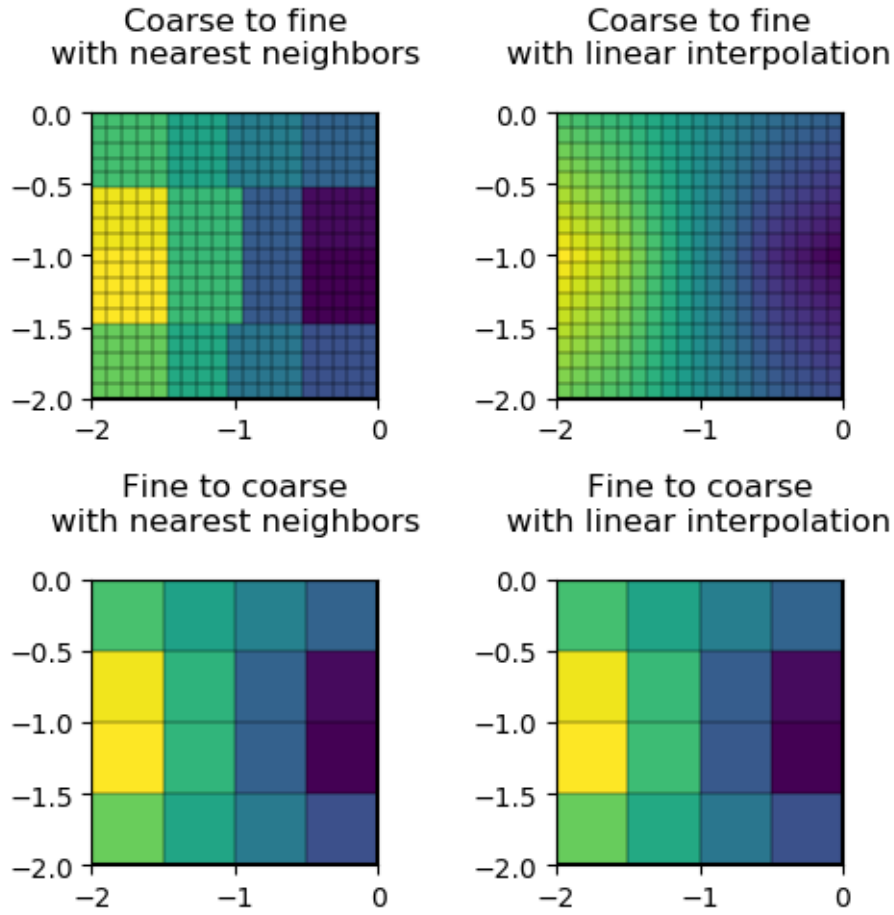
Visualization

```
meshes = [coarse, fine]
datasets = [coarse_data, fine_data]
ints = [nearest_neighbor_interpolation,
        linear_interpolation]

fig, ax = plt.subplots(2, 2, figsize=(5, 5))

# Coarse data to fine mesh
drawModel(ax[0, 0], fine, ints[0](coarse, coarse_data, fine), showCbar=False)
drawMesh(ax[0, 0], fine)
drawModel(ax[0, 1], fine, ints[1](coarse, coarse_data, fine), showCbar=False)
drawMesh(ax[0, 1], fine)

# Fine data to coarse mesh
drawModel(ax[1, 0], coarse, ints[0](fine, fine_data, coarse), showCbar=False)
drawMesh(ax[1, 0], coarse)
drawModel(ax[1, 1], coarse, ints[1](fine, fine_data, coarse), showCbar=False)
drawMesh(ax[1, 1], coarse)
```

(continues on next page)

(continued from previous page)

```
titles = ["Coarse to fine\nwith nearest neighbors",
          "Coarse to fine\nwith linear interpolation",
          "Fine to coarse\nwith nearest neighbors",
          "Fine to coarse\nwith linear interpolation"]

for a, title in zip(ax.flat, titles):
    a.set_title(title + "\n")

fig.tight_layout()
plt.show()
```



7.4.2.4 Quality of unstructured meshes

Problem:

Accurate numerical solutions require high quality meshes. In the case of unstructured triangular meshes (or tetrahedral meshes in 3D), relatively large and small angles can lead to discretization errors. Large angles can cause interpolation errors, while small angles can lead to ill-conditioned stiffness matrices.

Identification:

Some common 2D quality measures are implemented in *pygimli.meshtools* (page 309) and will be used in this tutorial. In 3D, we recommend to export the mesh

in VTK format and inspect mesh quality with ParaView (Filters -> Alphapetical -> Mesh quality).

Solution:

Meshes can be improved by adjusting cell sizes (*area* keyword) and the minimum allowed angle (*quality* keyword). *Gmsh* and other more advanced meshing tools also provide powerful mesh optimization algorithms. However, the numerical accurary may be improved at the expense of increased cell counts and thus longer computation times.

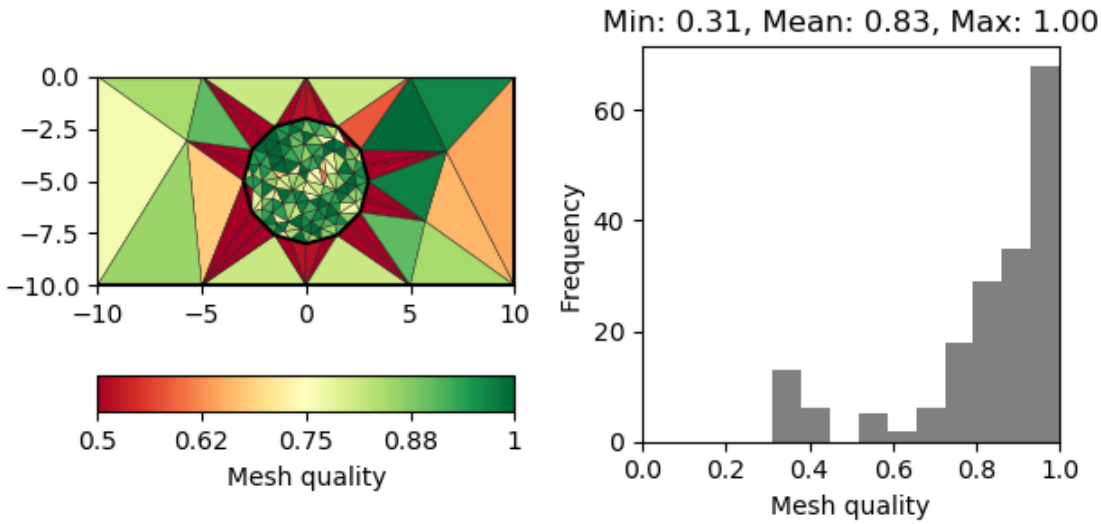
```
import matplotlib.pyplot as plt
import numpy as np

import pygimli as pg
from pygimli.meshtools import polytools as plc
from pygimli.meshtools import quality
```

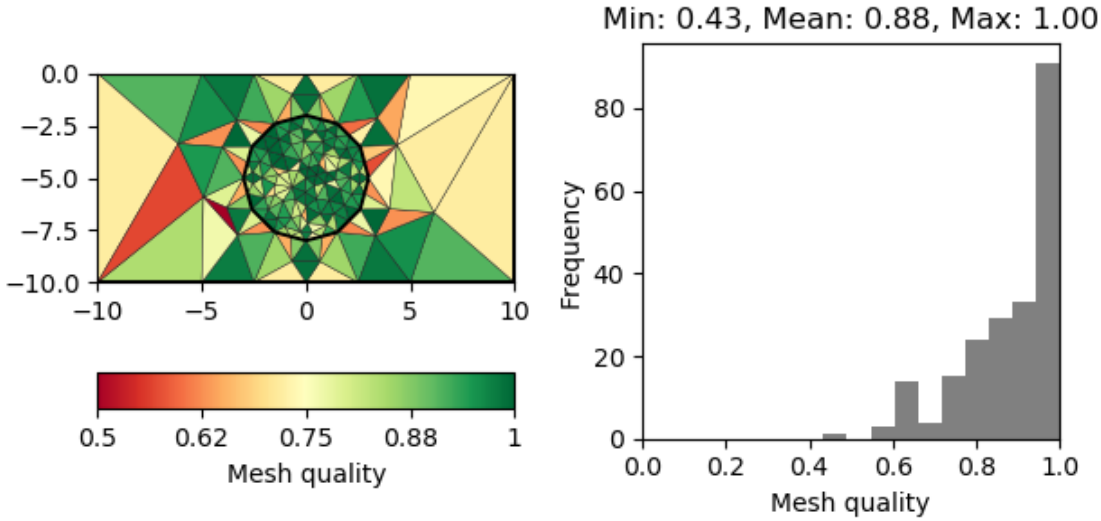
We start by creating a mesh with a refined object inside.

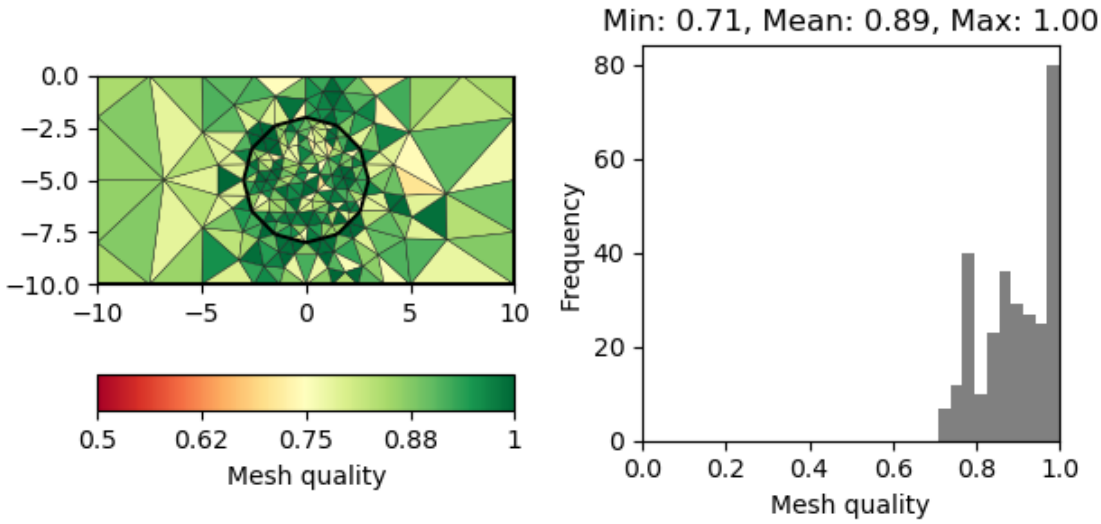
```
world = plc.createWorld(start=[-10, 0], end=[10, -10], marker=1,
                        worldMarker=False)
c1 = plc.createCircle(pos=[0.0, -5.0], radius=3.0, area=.3)
```

When calling the `pg.meshtools.createMesh()` function, a quality parameter can be forwarded to Triangle, which prescribes the minimum angle allowed in the final mesh. We can asses its effectiveness by creating different meshes and plotting the resulting quality. For this purpose, we define a showQuality function, which also plots a histogram of the mesh qualities.

```
def showQuality(mesh, qualities):
    fig, axes = plt.subplots(1, 2)
    axes[1].hist(qualities, color="grey")
    pg.show(mesh, qualities, ax=axes[0], cMin=0.5, cMax=1, hold=True,
            logScale=False, label="Mesh quality", cmap="RdYlGn", showMesh=True)
    s = "Min: %.2f, Mean: %.2f, Max: %.2f" % (
        np.min(qualities), np.mean(qualities), np.max(qualities))
    axes[1].set_title(s)
    axes[1].set_xlabel("Mesh quality")
    axes[1].set_ylabel("Frequency")
    axes[1].set_xlim(0, 1)

    # Figure resizing according to mesh dimesions
    x = mesh.xmax() - mesh.xmin()
    y = mesh.ymax() - mesh.ymin()
    width, height = fig.get_size_inches()
    fig.set_figheight(height * 1.3 * (y / x))
    fig.tight_layout()


for q in 10, 20, 30:
    m = pg.meshtools.createMesh([world, c1], quality=q)
    showQuality(m, quality(m))
```







Note that there is a decreasing number of problematic triangles (marked in red). However, the number of cells is increased siginficantly to achieve this.

Quality measures

There are numerous measures related to the area/volume, boundary lengths and angles of a cell (see[1] for a review). A straightforward measure considers the minimum angle in a triangle (normalized by 60 degrees). More sophisticated measures also take into account the cell size. A very common measure, often referred to as η, relates the area of a triangle a to its edge lengths l_1, l_2, l_3.

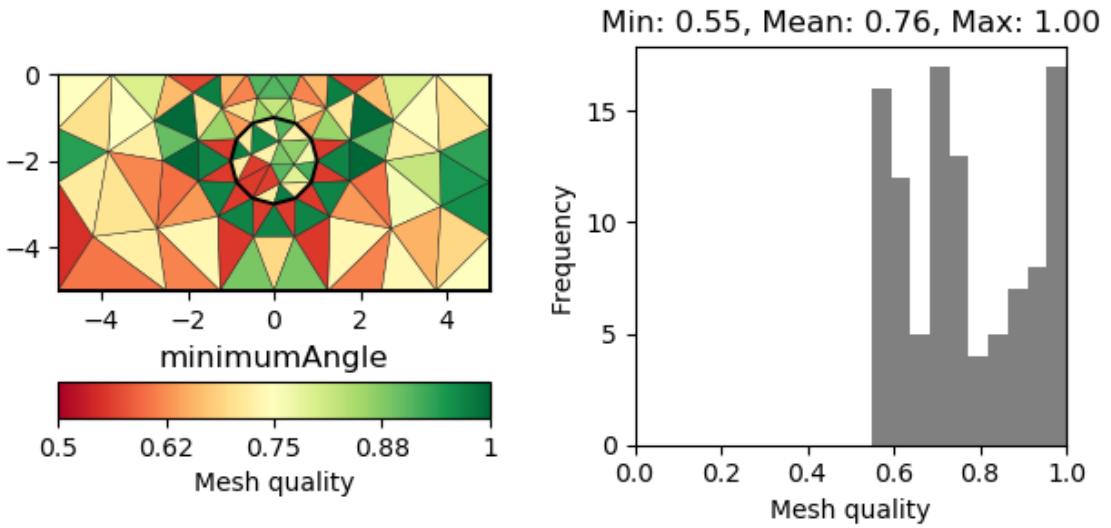
$$\eta = \frac{4\sqrt{3}a}{l_1^2 + l_2^2 + l_3^2}$$

The normalization factor $4\sqrt{3}$ ensures that a perfect triangle (equal edges) has a quality of 1. A popular measure also applicable for other cell types is the *Normalized shape ratio (NSR)*, which relates the circumradius R to the inradius of cell (r).

$$\rho = \frac{2r}{R}$$

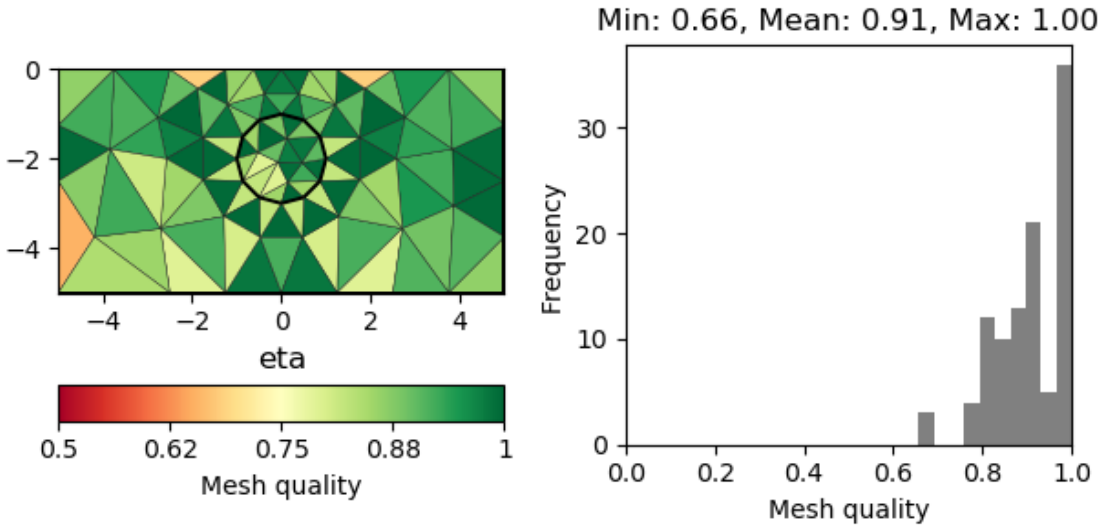
Again the factor 2 (3 in 3D) ensures that a perfect triangle has a quality of 1, whereas a flat triangle would have a quality of 0. The above mentioned quality measures are plotted below for the same mesh.

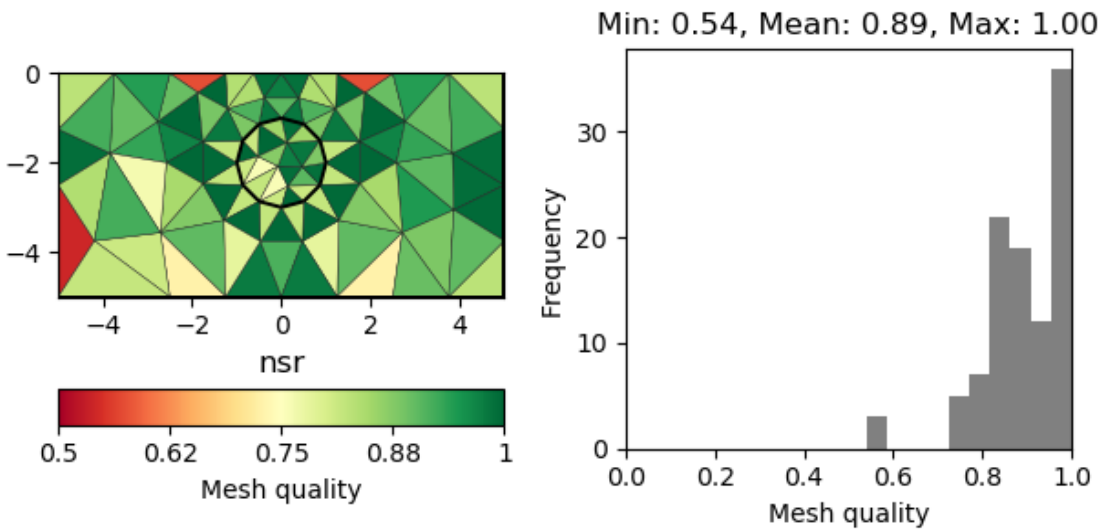
```
world = plc.createWorld(start=[-5, 0], end=[5, -5], marker=1,
                        worldMarker=False, area=2.)
c1 = plc.createCircle(pos=[0.0, -2.0], radius=1.0, area=.3)
mesh = pg.meshtools.createMesh([world, c1])

for measure in "minimumAngle", "eta", "nsr":
    showQuality(mesh, quality(mesh, measure))
    plt.title(measure)

plt.show()
```



-

[1] Field, D. A. (2000), Qualitative measures for initial meshes. Int. J. Numer. Meth. Engng., 47: 887–906.





References:

7.4.2.5 Region markers

Author: *Maximilian Weigand, University of Bonn*

A mesh can have different regions, which are defined by region markers for each cell. Region markers can be used to assign properties for forward modelling as well as to control the inversion behavior. This tutorial highlights the usage of regionMarkers, as well as some properties of regions if the **pygimli.meshtools** package is used to create complex compound meshes.

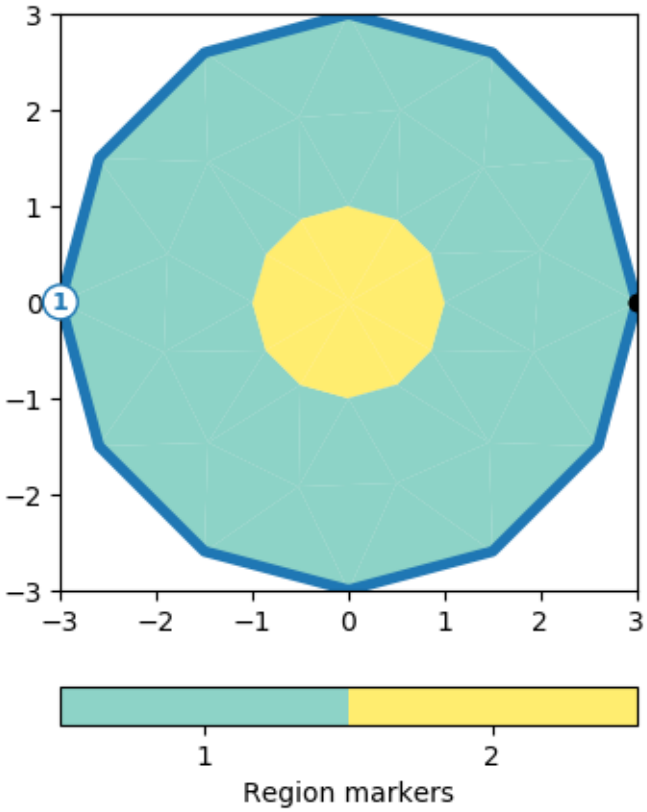
When constructing complex geometries out of basic geometric shapes (e.g., circle, rectangle, ...) we need to be careful with the region markers and their positions. This example shows how to use markerPositions to properly set region markers.

```
import pygimli as pg
import pygimli.meshtools as mt
```

In this first part we naively combine objects and assign markers to them, expecting two regions of concentric rings with markers 1 and 2. Note how the outer ring is assigned the marker 0 in the figure,

although we specified marker=1 for the larger circle? A marker value of 0 is assigned to a region if no region marker is found, indicating that the marker for the outer ring was overwritten/ignored by the inner circle, which was added later.

```
circle_outer = mt.createCircle(pos=[0.0, 0.0], radius=3.0, marker=1)

circle_inner = mt.createCircle(
    pos=[0.0, 0.0],
    radius=1.0,
    # area=.3,
    boundaryMarker=0,
    marker=2)

plc = circle_outer + circle_inner

ax, cb = pg.show(plc, markers=True)
```



The solution to this problem is the region marker, which defines the marker value of the region that it is placed in. By default all region markers are assigned the position (0,0,0), thereby overwriting each other (see black dots in figure below). If no region marker is present in a region, a marker value of 0 is assigned.

```
fig = ax.get_figure()
for nr, marker in enumerate(plc.regionMarkers()):
    print('Position marker number {}:'.format(nr + 1), marker.x(), marker.y(),
          marker.z())
    ax.scatter(marker.x(), marker.y(), s=(2 - nr) * 30, color='k')
```

(continues on next page)

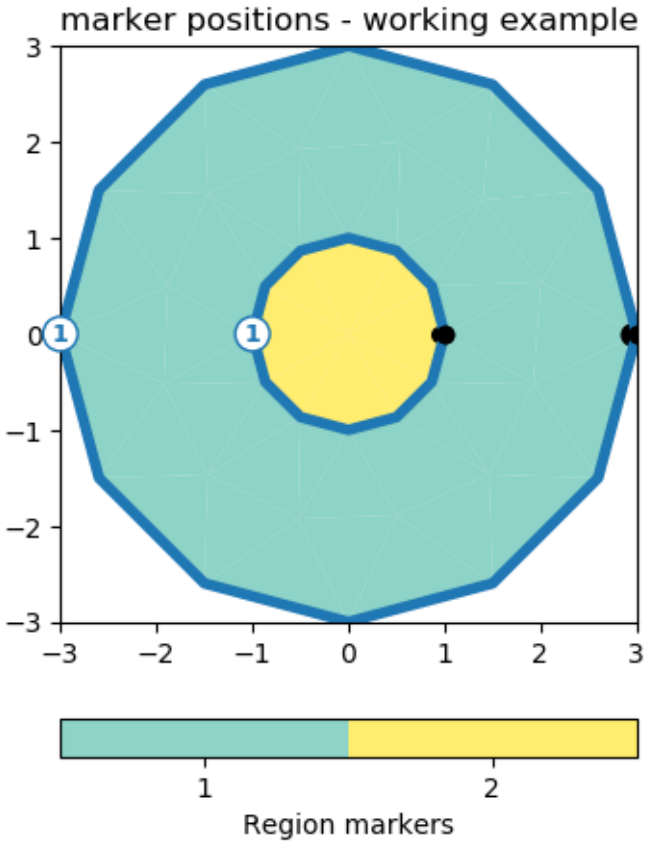
(continued from previous page)

```
ax.set_title('marker positions - non-working example')
fig.show()
```

```
Position marker number 1: 2.999 -3.700743415417188e-20 0.0
Position marker number 2: 0.999 1.8503717077085942e-20 0.0
```

Let us fix this issue by assigning region marker positions that are not overwritten by other objects when the geometries are merged (using the **markerPosition** parameter):

```
circle_outer = mt.createCircle(
    pos=[0.0, 0.0],
    radius=3.0,
    marker=1,
    markerPosition=[2.95, 0.0],
)

circle_inner = mt.createCircle(
    pos=[0.0, 0.0],
    radius=1.0,
    marker=2,
    markerPosition=[0.95, 0.0],
)

plc = circle_outer + circle_inner

ax, cb = pg.show(plc, markers=True)

fig = ax.get_figure()
for nr, marker in enumerate(plc.regionMarkers()):
    print('Position marker number {}:'.format(nr + 1), marker.x(), marker.y(),
          marker.z())
    ax.scatter(marker.x(), marker.y(), s=(2 - nr) * 30, color='k')
ax.set_title('marker positions - working example')
fig.show()
```




```
Position marker number 1: 2.95 0.0 0.0
Position marker number 2: 0.95 0.0 0.0
```

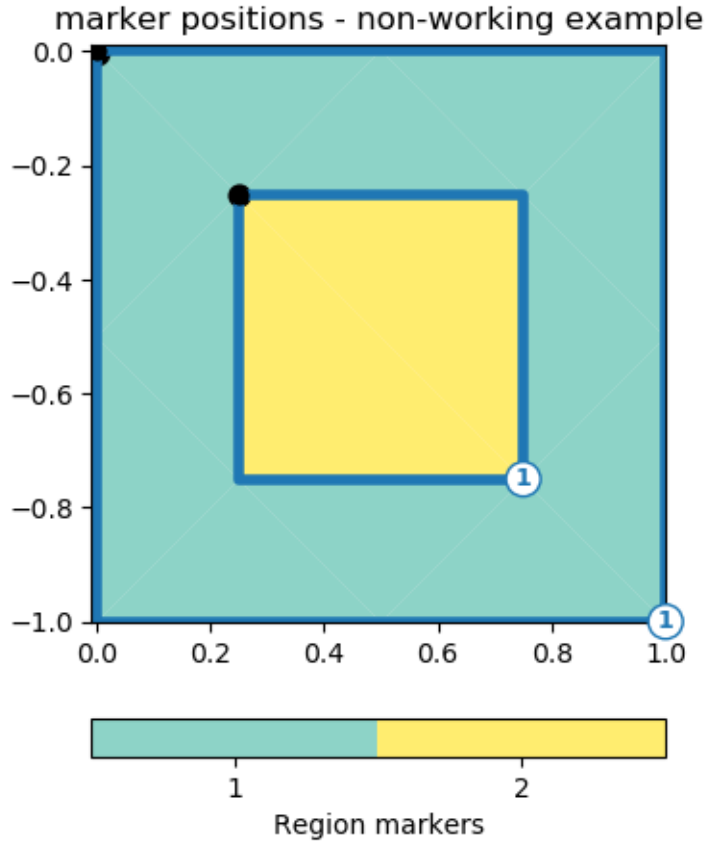
The same issue can occur for polygons. Polygons can assume complex forms, but for simplicity we create cubes here.

```
polygon1 = mt.createPolygon(
    [[0.0, 0.0], [1.0, 0.0], [1.0, -1.0], [0.0, -1]],
    isClosed=True,
    marker=1,
)

polygon2 = mt.createPolygon(
    [[0.25, -0.25], [0.75, -0.25], [0.75, -0.75], [0.25, -0.75]],
    isClosed=True,
    marker=2,
)

plc = polygon1 + polygon2

ax, cb = pg.show(plc, markers=True)

fig = ax.get_figure()
for nr, marker in enumerate(plc.regionMarkers()):
    print('Position marker number {}:'.format(nr + 1), marker.x(), marker.y(),
          marker.z())
```

(continues on next page)

(continued from previous page)

```
    ax.scatter(marker.x(), marker.y(), s=(4 - nr) * 20, color='k')
ax.set_title('marker positions - non-working example')
fig.show()
```




```
Position marker number 1: 0.0007071067811865475 -0.0007071067811865475 0.0
Position marker number 2: 0.2507071067811866 -0.2507071067811866 0.0
```

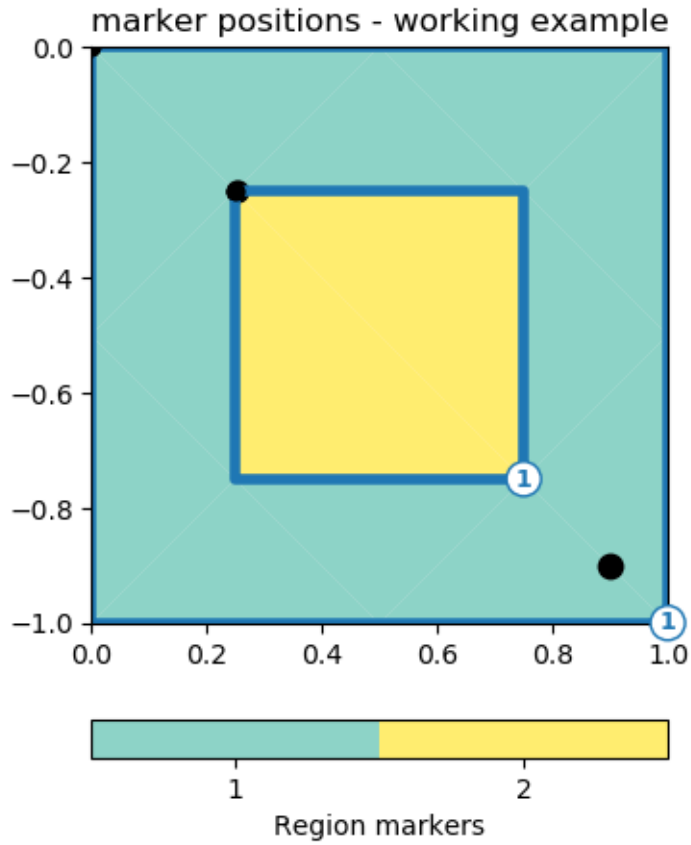
Again, we can simply fix with the **markerPosition** parameter

```
polygon1 = mt.createPolygon(
    [[0.0, 0.0], [1.0, 0.0], [1.0, -1.0], [0.0, -1]],
    isClosed=True,
    marker=1,
    markerPosition=[0.9, -0.9],
)

polygon2 = mt.createPolygon(
    [[0.25, -0.25], [0.75, -0.25], [0.75, -0.75], [0.25, -0.75]],
    isClosed=True,
    marker=2,
)

plc = polygon1 + polygon2

ax, cb = pg.show(plc, markers=True)
```

(continues on next page)

(continued from previous page)

```
fig = ax.get_figure()
for nr, marker in enumerate(plc.regionMarkers()):
    print('Position marker number {}:'.format(nr + 1), marker.x(), marker.y(),
          marker.z())
    ax.scatter(marker.x(), marker.y(), s=(4 - nr) * 20, color='k')
ax.set_title('marker positions - working example')
fig.show()
```




```
Position marker number 1: 0.9 -0.9 0.0
Position marker number 2: 0.2507071067811866 -0.2507071067811866 0.0
```

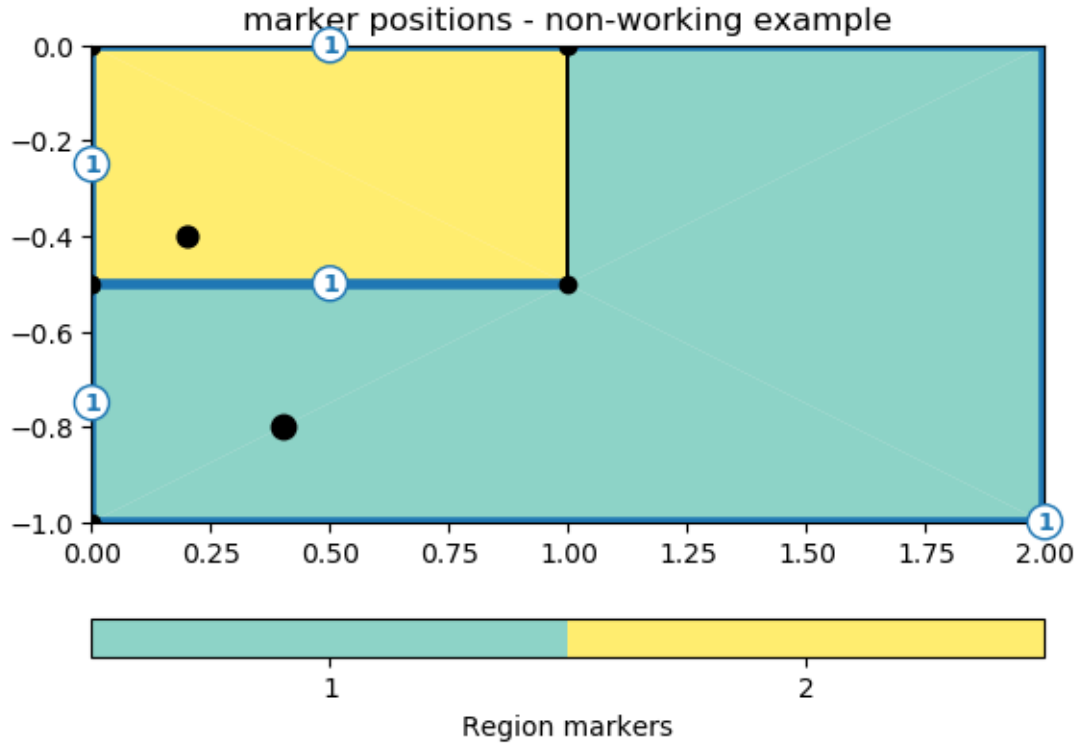
And finally, a similar example for rectangles…

```
rect1 = mt.createRectangle(
    start=[0.0, 0.0],
    end=[2.0, -1.0],
    isClosed=True,
    marker=1,
)

# move the rectangle by changing the center position
rect2 = mt.createRectangle(
    start=[0.0, 0.0],
    end=[1.0, -0.5],
```

(continues on next page)

(continued from previous page)

```
    isClosed=True,
    marker=2,
)

plc = rect1 + rect2

ax, cb = pg.show(plc, markers=True)

fig = ax.get_figure()
for nr, marker in enumerate(plc.regionMarkers()):
    print('Position marker number {}:'.format(nr + 1), marker.x(), marker.y(),
          marker.z())
    ax.scatter(marker.x(), marker.y(), s=(4 - nr) * 20, color='k')
ax.set_title('marker positions - non-working example')
fig.show()
```




```
Position marker number 1: 0.4 -0.8 0.0
Position marker number 2: 0.2 -0.4 0.0
```

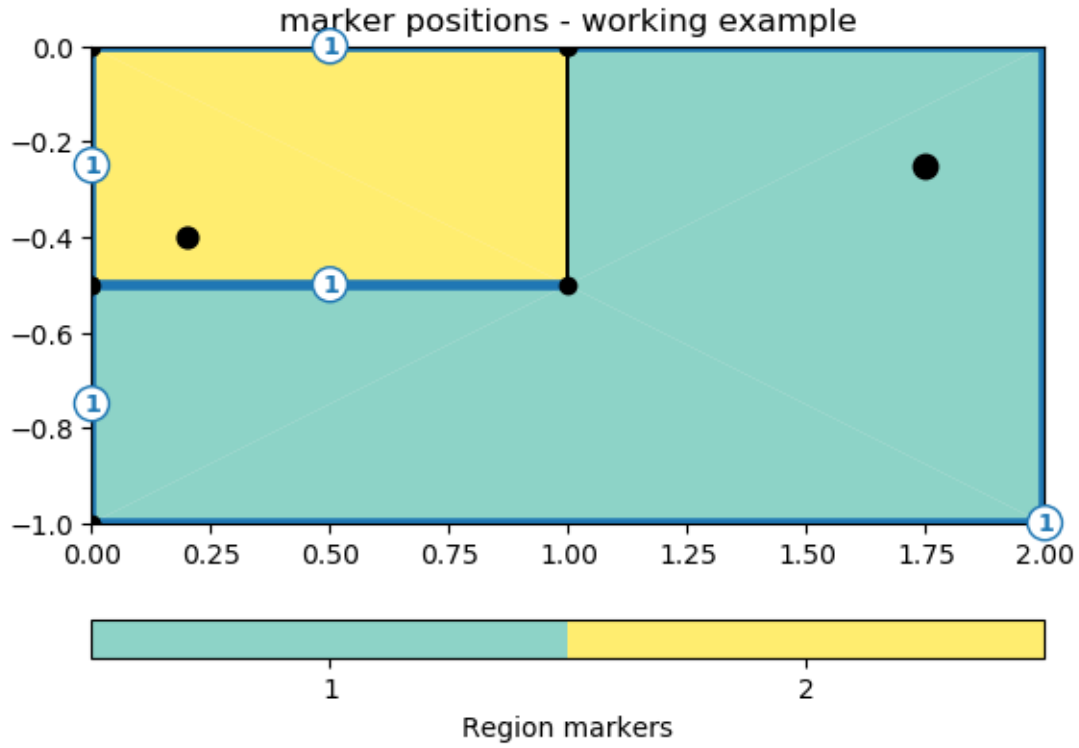
For the last time, fixing it. . .

```
rect1 = mt.createRectangle(
    start=[0.0, 0.0],
    end=[2.0, -1.0],
    isClosed=True,
    marker=1,
    markerPosition=[1.75, -0.25],
```

(continues on next page)

(continued from previous page)

```
)

# move the rectangle by changing the center position
rect2 = mt.createRectangle(
    start=[0.0, 0.0],
    end=[1.0, -0.5],
    isClosed=True,
    marker=2,
)

plc = rect1 + rect2

ax, cb = pg.show(plc, markers=True)

fig = ax.get_figure()
for nr, marker in enumerate(plc.regionMarkers()):
    print('Position marker number {}:'.format(nr + 1), marker.x(), marker.y(),
          marker.z())
    ax.scatter(marker.x(), marker.y(), s=(4 - nr) * 20, color='k')
ax.set_title('marker positions - working example')
fig.show()
```




```
Position marker number 1: 1.75 -0.25 0.0
Position marker number 2: 0.2 -0.4 0.0
```

Note: This tutorial was kindly contributed by Maximilian Weigand (University of Bonn). If you also

want to contribute an interesting example, check out our contribution guidelines.

7.4.3 Modelling

Here we collect examples and user stories that illustrate the modelling part of pygimli.

7.4.3.1 Basics of Finite Element Analysis

This tutorial covers the first steps into Finite Element computation referring the *M* (Modeling) in *pyGIMLi*.

We will not dig into deep details about the theory of the Finite Elements Analysis (FEA) here, as this can be found in several books, e.g., [**?**].

Anyhow, there is a little need for theory to understand what it means to use FEA for the solution of a boundary value problem. So we start with some basics.

Assuming Poisson's equation as a simple partial differential problem to be solved for the sought scalar field $u(\mathbf{r})$ within a modeling domain $\mathbf{r} \in \Omega$ with a non zero right hand side function f.

$$\begin{aligned} -\Delta u &= f \quad \text{in} \quad \Omega \\ u &= g \quad \text{on} \quad \partial\Omega \,. \end{aligned}$$

The Laplace operator $\Delta = \nabla \cdot \nabla$ given by the divergence of the gradient, is the sum of the second partial derivatives of the field $u(\mathbf{r})$ with respect to the Cartesian coordinates in 1D space $\mathbf{r} = (x)$, in 2D $\mathbf{r} = (x, y)$, or 3D space $\mathbf{r} = (x, y, z)$. On the boundary $\partial\Omega$ of the domain, we want known values of $u = g$ as so called Dirichlet boundary conditions.

A common approach to solve this problem is the method of weighted residuals. The base assumption states that an approximated solution $u_h \approx u$ will only satisfy the differential equation with a rest R: $\Delta u_h + f = R$. If we choose some weighting functions w, we can try to minimize the resulting residuum over our modeling domain as:

$$\int_\Omega Rw = 0 \,,$$

which leads to:

$$\int_\Omega -\Delta u_h w = \int_\Omega fw \,.$$

It is preferable to eliminate the second derivative in the Laplace operator, either due to integration by parts or by applying the product rule and Gauss's law. This leads to the so called weak formulation:

$$\begin{aligned} \int_\Omega \nabla u_h \nabla w - \int_{\partial\Omega} \mathbf{n} \nabla u_h w &= \int_\Omega fw \\ \int_\Omega \nabla u_h \nabla w &= \int_\Omega fw + \int_{\partial\Omega} \frac{\partial u_h}{\partial \mathbf{n}} w \,. \end{aligned}$$

We can solve these integrals after choosing an appropriate basis for an approximate solution u_h as:

$$u_h = \sum_i \mathrm{u}_i N_i \quad \text{with} \quad i = 0 \ldots \mathcal{N} \,.$$

The latter fundamental FEA relation discretizes the continuous solution u_h into a discrete values $\mathrm{u} = \{\mathrm{u}_i\}$ for a number of $i = 0 \ldots \mathcal{N}$ discrete points, usually called nodes.

The basis functions N_i can be understood as interpolation rules for the discrete solution between adjacent nodes and will be chosen later.

Now we can set the unknown weighting functions to be the same as the basis functions $w = N_j$ with $j = 0 \dots \mathcal{N}$ (Galerkin method)

$$\begin{aligned}\int_\Omega \sum_i \mathrm{u}_i \nabla N_i \nabla N_j & \\ &= \int_\Omega f_j N_j + \int_{\partial\Omega} h N_j \quad \text{with} \quad h \\ &= \frac{\partial u}{\partial \mathbf{n}}\end{aligned}$$

this can be rewritten with $h = 0$ as:

$$\begin{aligned}&\mathrm{Au} = \mathrm{b} \\ &\quad \text{with} \\ &\mathrm{A} = \{\mathrm{a}_{i,j}\} = \int_\Omega \nabla N_i \nabla N_j \quad \text{known as 'Stiffness matrix'} \\ &\mathrm{b} = \{\mathrm{b}_j\} = \int_\Omega f_j N_j \quad \text{known as 'Load vector'}\end{aligned}$$

The solution of this linear system of equations leads to the discrete solution $\mathrm{u} = \{\mathrm{u}_i\}$ for all $i = 1 \dots \mathcal{N}$ nodes inside the modeling domain.

For the practical part, the choice of the nodes is crucial. If we choose too little, the accuracy of the sought solution might be too small. If we choose too many, the dimension of the system matrix will be too large, which leads to higher memory consumption and calculation times.

To define the nodes, we discretize our modeling domain into cells, or the eponymous elements. Cells are basic geometric shapes like triangles or hexahedrons and are constructed from the nodes and collected in a mesh. See the tutorials about the mesh basics (*Basics* (page 177)). In summary, the discrete solutions of the differential equation using FEA on a specific mesh are defined on the node positions of the mesh.

The chosen mesh cells also define the base functions and the integration rules that are necessary to assemble the stiffness matrix and the load vector and will be discussed in a different tutorial (TOWRITE link here).

To finally solve our little example we still need to handle the application of the boundary condition $u = g$ which is called Dirichlet condition. Setting explicit values for our solution is not covered by the general Galerkin weighted residuum method but we can solve it algebraically. We reduce the linear system of equations by the known solutions $g = g_k$ for all k nodes on the affected boundary elements: (maybe move this to the BC tutorial)

$$\begin{aligned}&\mathrm{A_D} \cdot \mathrm{u} = \mathrm{b_D} \\ &\quad \text{with} \\ &\mathrm{A_D} = \{\mathrm{a}_{i,j}\} \quad \forall i, j \notin k \text{ and } 1 \text{ for } i, j \in k \\ &\mathrm{b_D} = \{\mathrm{b}_j\} - \mathrm{A} \cdot \mathrm{g} \quad \forall j \notin k \text{ and } g_k \text{ for } j \in k\end{aligned}$$

Now we have all parts together to assemble $\mathrm{A_D}$ and $\mathrm{b_D}$ and finally solve the given boundary value problem.

It is usually a good idea to test a numerical approach with known solutions. To keep things simple we create a modeling problem from the reverse direction. We choose a solution, calculate the right hand

side function and select the domain geometry suitable for nice Dirichlet values.

$$
\begin{aligned}
u(x,y) &= \sin(x)\sin(y) \\
-\Delta u &= f(x,y) = 2\sin(x)\sin(y) \\
\Omega \in I\!R^2 \quad &\text{on} \quad 0 \le x \le 2\pi,\ \ 0 \le y \le 2\pi \\
u = g &= 0 \quad \text{on} \quad \partial\Omega
\end{aligned}
$$

We now can solve the Poison equation applying the FEA capabilities of pygimli and compare the resulting approximate solution u with our known exact solution $u(x,y)$.

```
import numpy as np
import pygimli as pg
```

We start to define the modeling domain and functions for the exact solution and the values for the load vector. The desired mesh of our domain will be a grid with equidistant spacing in x and y directions.

```
domain = pg.createGrid(x=np.linspace(0.0, 2*np.pi, 25),
                       y=np.linspace(0.0, 2*np.pi, 25))

uExact = lambda pos: np.sin(pos[0]) * np.sin(pos[1])
f = lambda cell: 2.0 * np.sin(cell.center()[0]) * np.sin(cell.center()[1])
```

We use the existing shortcut functions for the assembling of the basic FEA system matrices and vectors. The implemented parts of the solving process are supposed to be dimension independent. You only need to find a valid mesh with the supported element types.

```
A = pg.solver.createStiffnessMatrix(domain)
b = pg.solver.createLoadVector(domain, f)
```

To apply the boundary condition we first need to identify all boundary elements. The default grid applies the following boundary marker on the outermost boundaries: 1 (left), 2(right), 3(top), and 4(bottom).

```
boundaries = pg.solver.parseArgToBoundaries({'1,2,3,4': 0.0}, domain)
```

parseArgToBoundaries is a helper function to collect a list of tupels (Boundary element, value), which can be used to apply the Dirichlet conditions.

```
pg.solver.assembleDirichletBC(A, boundaries, b)
```

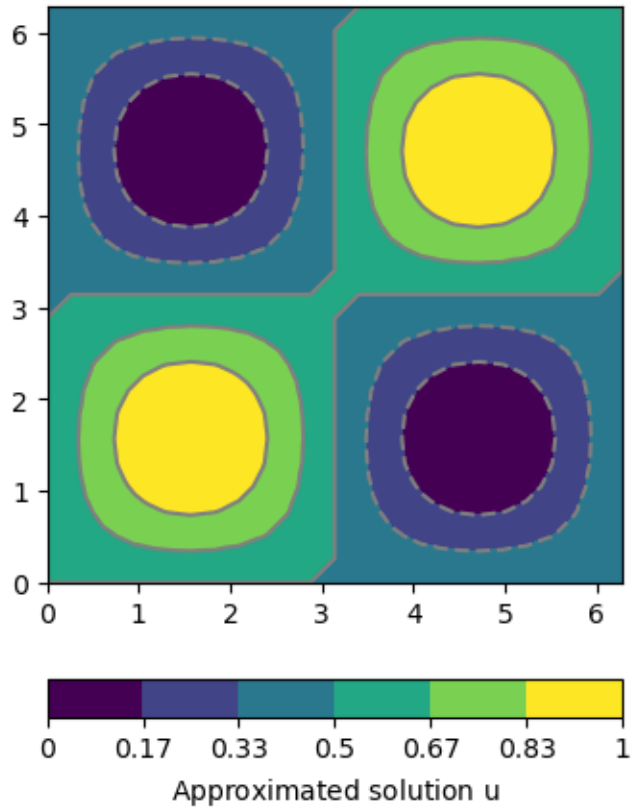
```
{25: 0.0, 0: 0.0, 50: 0.0, 75: 0.0, 100: 0.0, 125: 0.0, 150: 0.0, 175: 0.0,␣
↪200: 0.0, 225: 0.0, 250: 0.0, 275: 0.0, 300: 0.0, 325: 0.0, 350: 0.0, 375: 0.0,
↪ 400: 0.0, 425: 0.0, 450: 0.0, 475: 0.0, 500: 0.0, 525: 0.0, 550: 0.0, 575: 0.
↪0, 600: 0.0, 24: 0.0, 49: 0.0, 74: 0.0, 99: 0.0, 124: 0.0, 149: 0.0, 174: 0.0,
↪ 199: 0.0, 224: 0.0, 249: 0.0, 274: 0.0, 299: 0.0, 324: 0.0, 349: 0.0, 374: 0.
↪0, 399: 0.0, 424: 0.0, 449: 0.0, 474: 0.0, 499: 0.0, 524: 0.0, 549: 0.0, 574:␣
↪0.0, 599: 0.0, 624: 0.0, 1: 0.0, 2: 0.0, 3: 0.0, 4: 0.0, 5: 0.0, 6: 0.0, 7: 0.
↪0, 8: 0.0, 9: 0.0, 10: 0.0, 11: 0.0, 12: 0.0, 13: 0.0, 14: 0.0, 15: 0.0, 16: 0.
↪0, 17: 0.0, 18: 0.0, 19: 0.0, 20: 0.0, 21: 0.0, 22: 0.0, 23: 0.0, 601: 0.0,␣
↪602: 0.0, 603: 0.0, 604: 0.0, 605: 0.0, 606: 0.0, 607: 0.0, 608: 0.0, 609: 0.0,
↪ 610: 0.0, 611: 0.0, 612: 0.0, 613: 0.0, 614: 0.0, 615: 0.0, 616: 0.0, 617: 0.
↪0, 618: 0.0, 619: 0.0, 620: 0.0, 621: 0.0, 622: 0.0, 623: 0.0}
```

The approximate solution u can then be found as the solution of the linear system of equations.

```
u = pg.solver.linSolve(A, b)
```

The resulting scalar field can displayed with the *pg.show* shortcut.

```
pg.show(domain, u, label='Approximated solution $\mathrm{u}$', nLevs=7)
```




```
(<matplotlib.axes._subplots.AxesSubplot object at 0x7fe87abec850>, <matplotlib.
↪colorbar.Colorbar object at 0x7fe89a093730>)
```

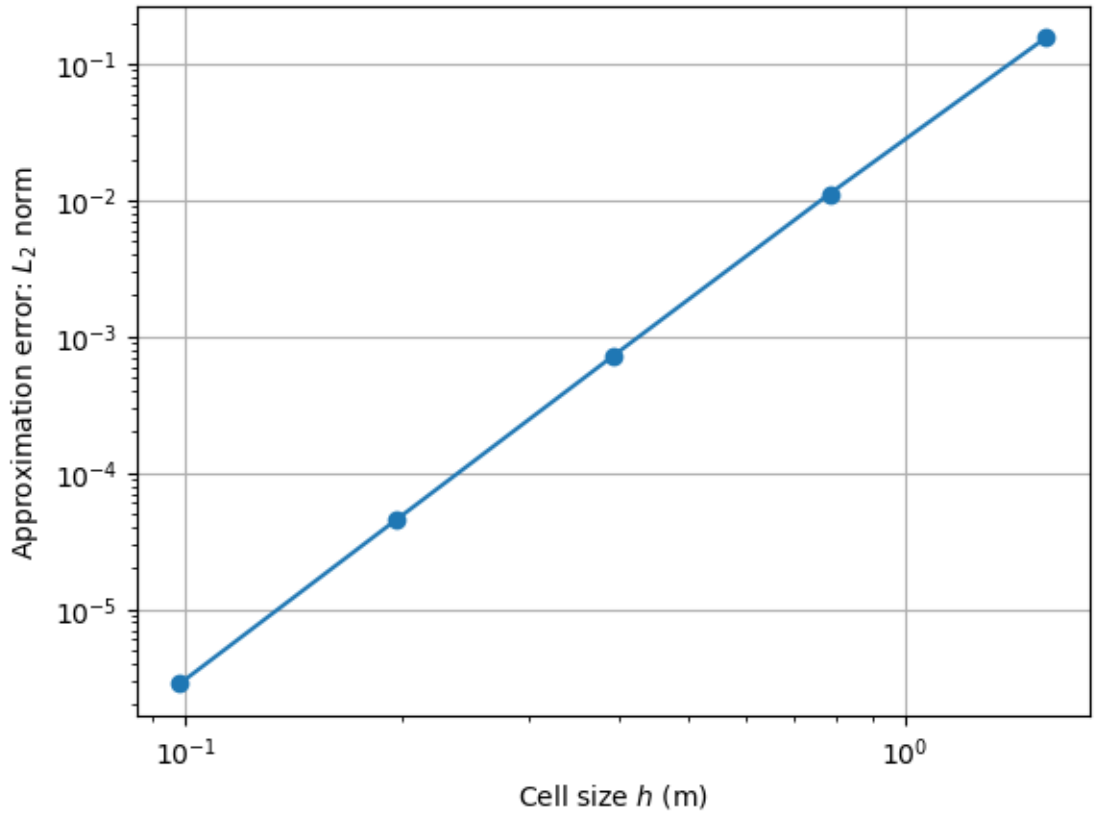
For analyzing the accuracy for the approximation we apply the L2 norm for the finite element space *pygimli.solver.normL2* (page 444) for a set of different solutions with decreasing cell size. Instead of using the the single assembling steps again, we apply our Finite Element shortcut function *pygimli.solver.solve* (page 449).

```
domain = pg.createGrid(x=np.linspace(0.0, 2*np.pi, 3),
                       y=np.linspace(0.0, 2*np.pi, 3))

h = []
l2 = []
for i in range(5):
    domain = domain.createH2()
    u_h = pg.solve(domain, f=f, bc={'Dirichlet':{'1:5': 0}})
    u = np.array([uExact(_) for _ in domain.positions()])
    l2.append(pg.solver.normL2(u - u_h, domain))
    h.append(min(domain.boundarySizes()))
```

(continues on next page)

(continued from previous page)

```
    print("NodeCount: {0}, h:{1}m, L2:{2}%".format(domain.nodeCount(),
                                                   h[-1], l2[-1]))

ax,_ = pg.show()
ax.loglog(h, l2, 'o-')
ax.set_ylabel('Approximation error: $L_2$ norm')
ax.set_xlabel('Cell size $h$ (m)')
ax.grid()
```




```
NodeCount: 25, h:1.5707963267948966m, L2:0.15650280987445644%
NodeCount: 81, h:0.785398163397447 4m, L2:0.011159591448055969%
NodeCount: 289, h:0.3926990816987228m, L2:0.0007191303723399217%
NodeCount: 1089, h:0.1963495408493614m, L2:4.528472476499632e-05%
NodeCount: 4225, h:0.09817477042467981m, L2:2.835595597971797e-06%
```

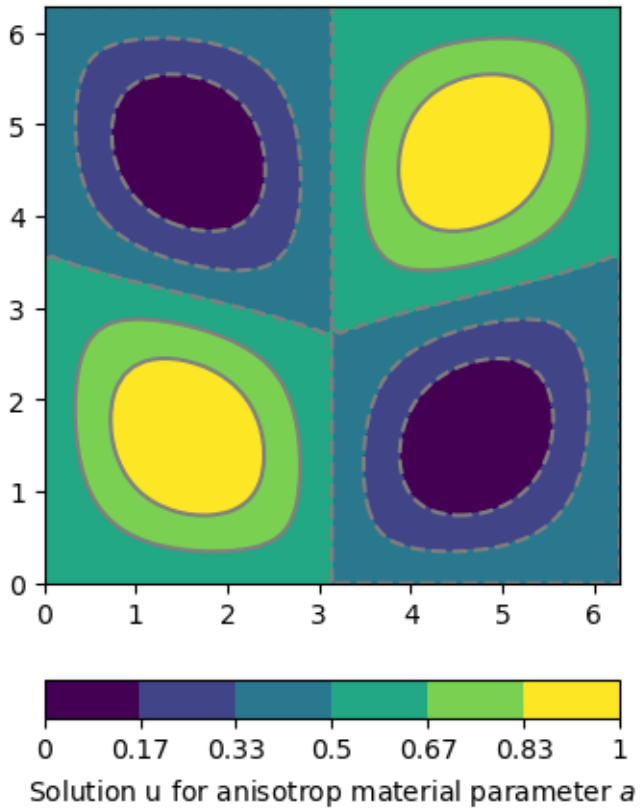
We calculated the examples before for a homogeneous material parameter a=1, but we can apply any heterogeneous values to0. One way is to create a list of parameter values, one for each cell of the domain. Currently the values for each cell can be of type float, complex, or real valued anisotropy or constitutive matrix. For illustration we show a calculation with an anisotropic material. We simply use the same setting as above and assume a -45 degree dipping angle in the left and 45 degree dipping in the right part of the domain. Maybe we will find someday a more meaningful example. If you have an idea please don't hesitate to share.

```
a = [None]*domain.cellCount()
for c in domain.cells():
```

(continues on next page)

(continued from previous page)

```
    if c.center()[0] < np.pi:
        a[c.id()] = pg.solver.createAnisotropyMatrix(lon=1.0, trans=10.0,
                                                     theta=-45/180 * np.pi)
    else:
        a[c.id()] = pg.solver.createAnisotropyMatrix(lon=1.0, trans=10.0,
                                                     theta=45/180 * np.pi)

u = pg.solve(domain, a=a, f=f, bc={'Dirichlet':{'*': 0}})
pg.show(domain, u, label='Solution $\mathrm{u}$ for anisotrop material␣
↪parameter $a$', nLevs=7)
```




```
(<matplotlib.axes._subplots.AxesSubplot object at 0x7fe89a1a3430>, <matplotlib.
↪colorbar.Colorbar object at 0x7fe8fa420760>)
```

7.4.3.2 Modelling with Boundary Conditions

We use the preceding example (Poisson equation on the unit square) but want to specify different boundary conditions on the four sides.

Again, we first import numpy and pygimli, the solver and post processing functionality.

```
import numpy as np
import pygimli as pg

from pygimli.solver import solve
```

(continues on next page)

(continued from previous page)

```
from pygimli.viewer import show
from pygimli.viewer.mpl import drawStreams
```

We create 21 x 21 node grid to solve on.

```
grid = pg.createGrid(x=np.linspace(-1.0, 1.0, 21),
                     y=np.linspace(-1.0, 1.0, 21))
```

We start considering inhomogeneous Dirichlet boundary conditions (BC).

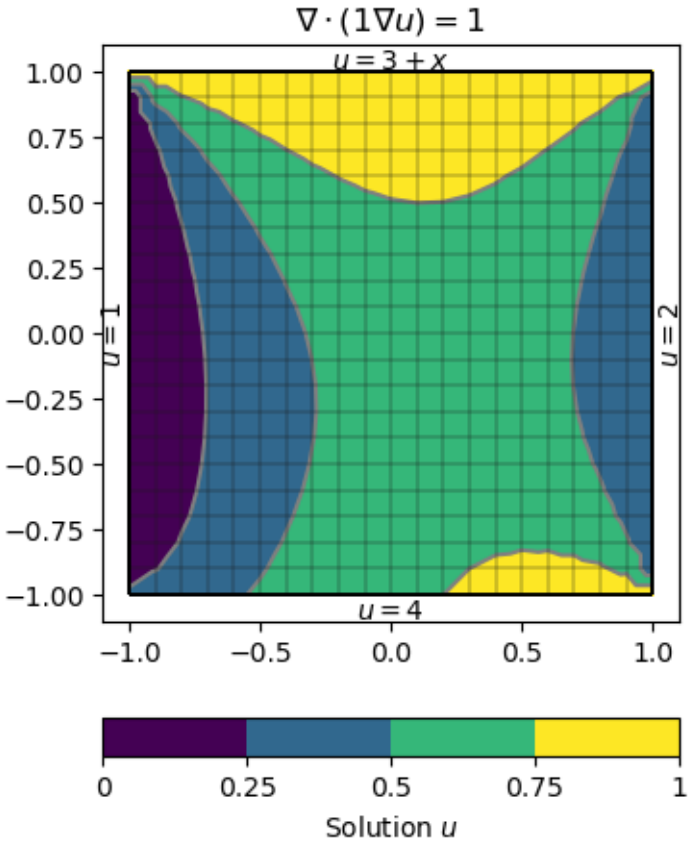
There are different ways of specifying BCs. They can be maps from markers to values, explicit functions or implicit (lambda) functions.

The boundary 1 (left) and 2 (right) are directly mapped to the values 1 and 2. On side 3 (top) a lambda function 3+x is used (p is the boundary position and p[0] its x coordinate. On side 4 (bottom) a function uDirichlet is used that simply returns 4 in this example but can compute anything as a function of the individual boundaries b.

```
def uDirichlet(boundary):
    """Return a solution value for a given boundary. Scalar values␣
↪are applied to all nodes of the boundary."""
    return 4.0

dirichletBC = {1: 1,                                          # left
               2: 2.0,                                        # right
               3: lambda boundary: 3.0 + boundary.center()[0], # top
               4: uDirichlet}                                 # bottom
```

The boundary conditions are passed using the bc keyword dictionary.

```
u = solve(grid, f=1., bc={'Dirichlet': dirichletBC})

# Note that showMesh returns the created figure ax and the created␣
↪colorBar.
ax, cbar = show(grid, data=u, label='Solution $u$')

show(grid, ax=ax)

ax.text(0, 1.01, '$u=3+x$', ha='center')
ax.text(-1.01, 0, '$u=1$', va='center', ha='right', rotation='vertical')
ax.text(0, -1.01, '$u=4$', ha='center', va='top')
ax.text(1.02, 0, '$u=2$', va='center', ha='left',  rotation='vertical')

ax.set_title('$\\nabla\cdot(1\\nabla u)=1$')

ax.set_xlim([-1.1, 1.1])  # some boundary for the text
ax.set_ylim([-1.1, 1.1])
```




```
(-1.1, 1.1)
```

Alternatively we can define the gradients of the solution on the boundary, i.e., Neumann type BC. This is done with another dictionary {marker: value} and passed by the bc dictionary.

```
neumannBC = {1: -0.5,  # left
             4: 2.5}  # bottom

dirichletBC = {3: 1.0}  # top

u = solve(grid, f=0., bc={'Dirichlet': dirichletBC, 'Neumann': neumannBC})
```

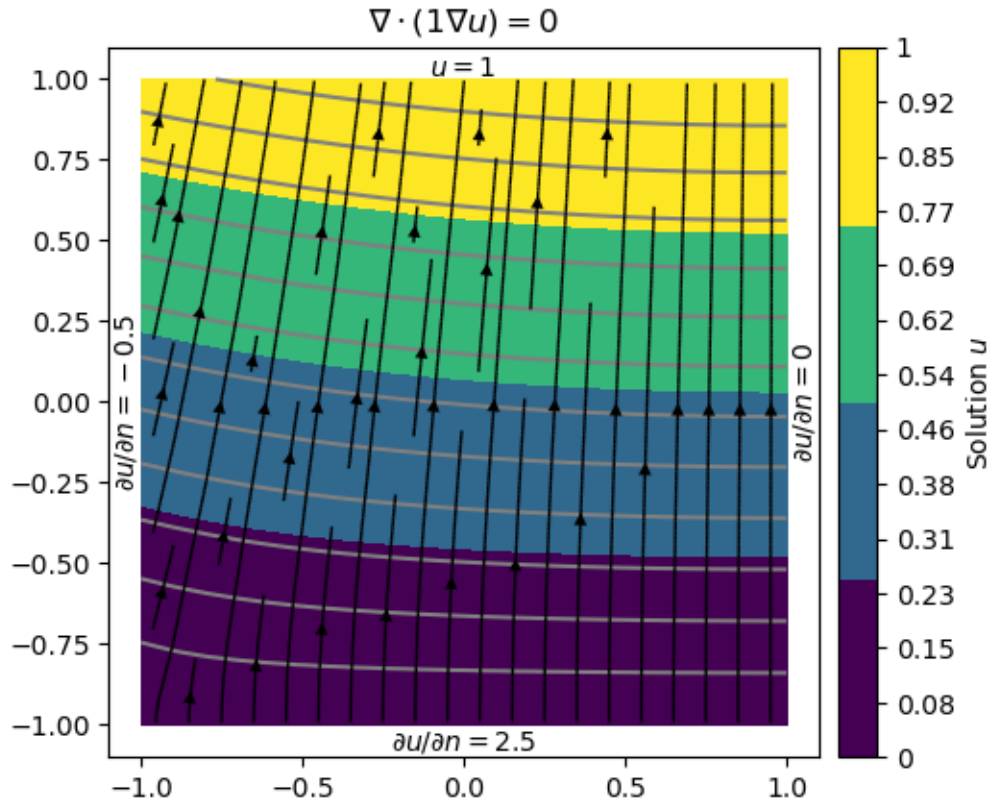
Note that on boundary 2 (right) has no BC explicitly applied leading to default (natural) BC that are of homogeneous Neumann type $\frac{\partial u}{\partial n} = 0$

```
ax = show(grid, data=u, filled=True, orientation='vertical',
          label='Solution $u$',
          levels=np.linspace(min(u), max(u), 14), hold=True)[0]

# Instead of the grid we now want to add streamlines to show the␣
↪gradients of
# the solution (i.e., the flow direction).

drawStreams(ax, grid, u)

ax.text(0.0, 1.01, '$u=1$',
```

(continues on next page)

(continued from previous page)

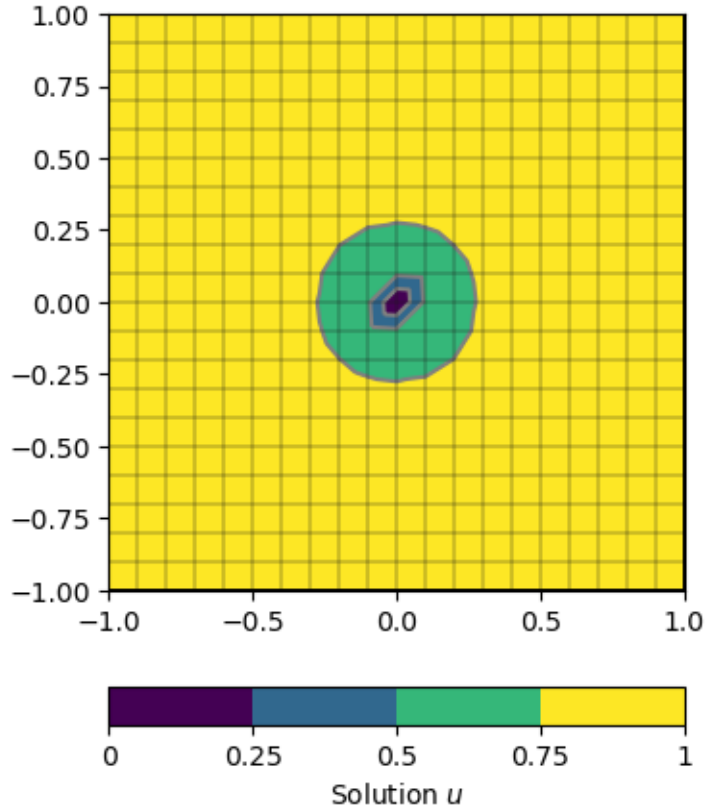
```
        horizontalalignment='center')  # top -- 3
ax.text(-1.0, 0.0, '$\partial u/\partial n=-0.5$',
        va='center', ha='right', rotation='vertical')  # left -- 1
ax.text(0.0, -1.01, '$\partial u/\partial n=2.5$',
        ha='center', va='top')  # bot -- 4
ax.text(1.01, 0.0, '$\partial u/\partial n=0$',
        va='center', ha='left', rotation='vertical')  # right -- 2

ax.set_title('$\\nabla\cdot(1\\nabla u)=0$')

ax.set_xlim([-1.1, 1.1])
ax.set_ylim([-1.1, 1.1])
```




```
(-1.1, 1.1)
```

Its also possible to force single nodes to fixed values too: Short test: setting the value for the center node to 1.0

```
u = solve(grid, f=1., bc={'Node': [grid.findNearestNode([0.0, 0.0]), 1.0]})

ax, _ = pg.show(grid, u, label='Solution $u$',)
show(grid, ax=ax)
```




```
(<matplotlib.axes._subplots.AxesSubplot object at 0x7fe89a249280>, None)
```

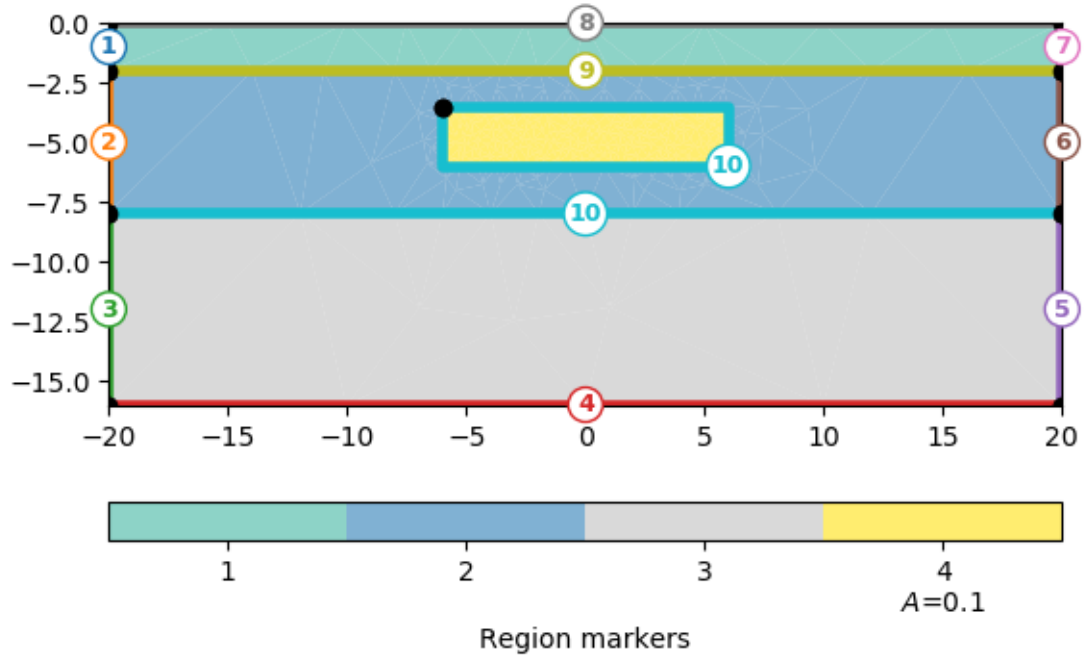
7.4.3.3 Heat equation in 2D

This tutorial solves the stationary heat equation in 2D. The example is taken from the pyGIMLi paper (https://cg17.pygimli.org).

```
import pygimli as pg
import pygimli.meshtools as mt
```

Create geometry definition for the modelling domain.

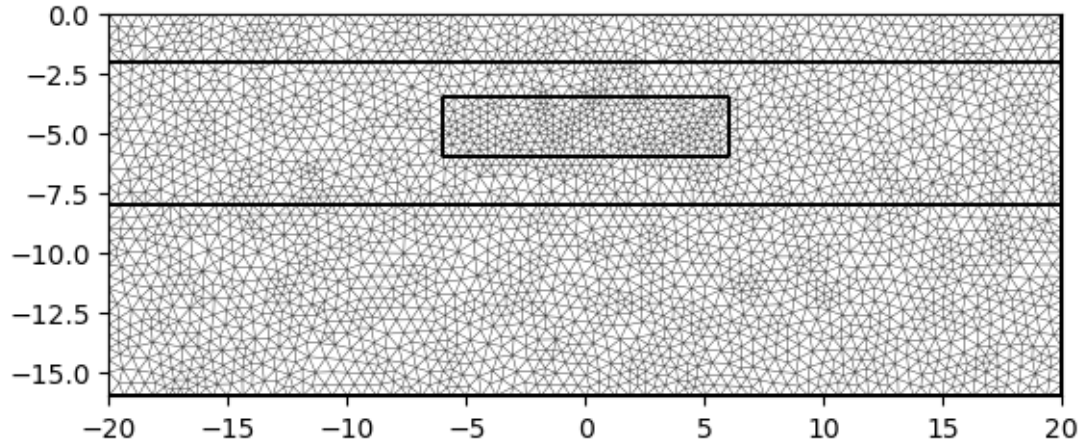
```
world = mt.createWorld(start=[-20, 0], end=[20, -16], layers=[-2, -8],
                       worldMarker=False)
# Create a heterogeneous block
block = mt.createRectangle(start=[-6, -3.5], end=[6, -6.0],
                           marker=4,  boundaryMarker=10, area=0.1)
# Merge geometrical entities
geom = world + block
pg.show(geom, markers=True)
```




```
(<matplotlib.axes._subplots.AxesSubplot object at 0x7fe8f56c6040>, None)
```

Create a mesh from based on the geometry definition. When calling the `pg.meshtools.createMesh()` function, a quality parameter can be forwarded to Triangle, which prescribes the minimum angle allowed in the final mesh. For a tutorial on the quality of the mesh please refer to : Mesh quality inspection [1] [1]: https://www.pygimli.org/_tutorials_auto/1_basics/plot_6-mesh-quality-inspection.html#sphx-glr-tutorials-auto-1-basics-plot-6-mesh-quality-inspection-py Note: Incrementing quality increases computer time, take precaution with quality values over 33.

```
mesh = mt.createMesh(geom, quality=33, area=0.2, smooth=[1, 10])
pg.show(mesh)
```

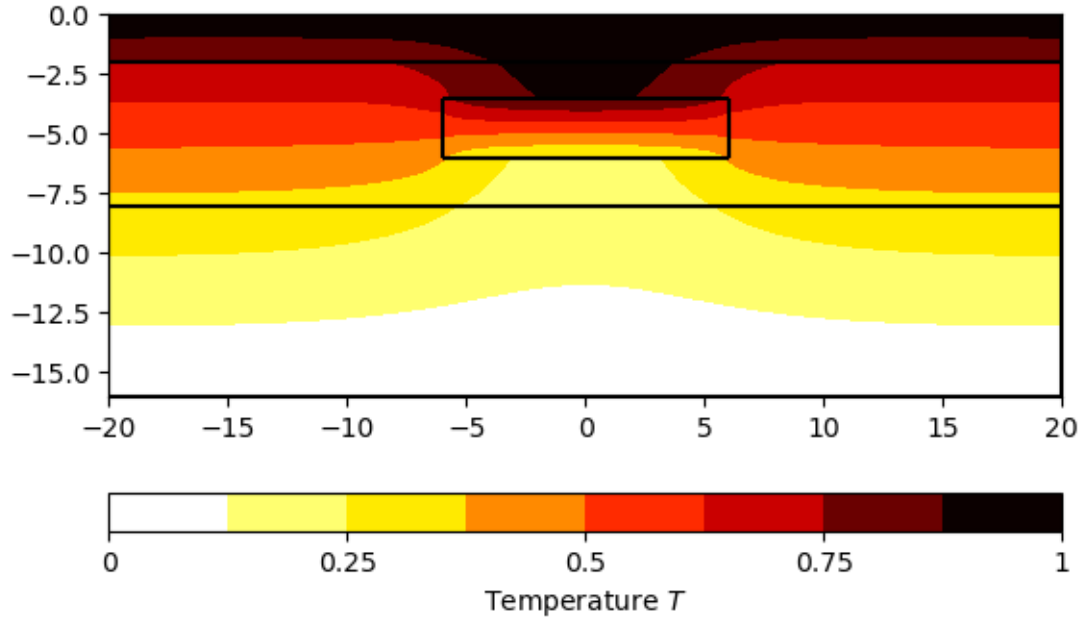



```
(<matplotlib.axes._subplots.AxesSubplot object at 0x7fe899fb9790>, None)
```

Call *pygimli.solver.solveFiniteElements()* (page 450) to solve the heat diffusion equation $\nabla \cdot (a\nabla T) = 0$ with $T(bottom) = 0$ (boundary marker 8) and $T(top) = 1$ (boundary marker 4), where a is the thermal diffusivity and T is the temperature distribution. We assign thermal diffusivities to the four # regions using their marker numbers in a dictionary (a) and the fixed temperatures at the

boundaries using Dirichlet boundary conditions with the respective markers in another dictionary (bc)

```
T = pg.solver.solveFiniteElements(mesh,
                                  a={1: 1.0, 2: 2.0, 3: 3.0, 4:0.1},
                                  bc={'Dirichlet': {8: 1.0, 4: 0.0}},␣
 ↪verbose=True)
ax, _ = pg.show(mesh, data=T, label='Temperature $T$',
                cMap="hot_r", nCols=8, contourLines=False)

pg.show(geom, ax=ax, fillRegion=False)
```




```
Mesh:  Mesh: Nodes: 3011 Cells: 5832 Boundaries: 8842
Assembling time:  0.04
Solving time:  0.019

(<matplotlib.axes._subplots.AxesSubplot object at 0x7fe899949f40>, None)
```

7.4.4 Inversion

Simple inversions with increasing complexity.

7.4.4.1 Simple fit

This tutorial shows how to do the simplest inversion case, a curve fit, by setting up a custom forward operator. The function to be fitted is`

$$f(x) = A * e^{-x/X}$$

with the two unknown coefficients A (a signal amplitude) and X (a decay rate). Both A and X are assumed to be positive which is often the case for physical properties. The easiest way to do this is via a logarithmic transformation of the model vector (containing A and X) which is very easily done in pyGIMLi.

First we import the pygimli library under a short name pg and the numerics library numpy. Additionally we load the python plotting module of the library matplotlib. Both are contained in most python distributions and systems.

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

We set up the modelling operator, i.e. to return $\mathbf{f}(\mathbf{x})$ for given model parameters A and X subsumed in a vector. In order to be able to use operator in inversion, we derive from the abstract modelling base class. The latter holds the main mimic of generating Jacobian and adminstrating the model, the regularization and so on. The only function to overwrite is **response()**. If no function **createJacobian** is provided, they are computed by brute force (forward calculations with altered parameters).

```
class ExpModelling(pg.Modelling):
    def __init__(self, xvec, verbose=False):
        super().__init__()
        self.x = xvec

    def response(self, model):
        return model[0] * pg.exp(-self.x / model[1])

    def createStartModel(self, dataVals):
        return pg.Vector([1.0, 3.0])
```

The init function saves the x vector and defines the parameterization, i.e. two independent parameters (a 1D mesh with 1 cell and 2 properties). The response function computes the function using A=model[0] and X=model[1] The function startModel defines a meaningful starting vector. There are other methods to set the starting model as inv.setModel() but this one is a default one for people who use the class and forget about a starting model.

```
# We first create an abscissa vector using numpy (note that pygimli␣
↪also
# provides an exp function and generate synthetic data with two␣
↪arbitrary A and
# X values.

x = np.arange(0, 1, 1e-2)
data = 10.5 * np.exp(- x / 550e-3)
```

We define an (absolute) error level and add Gaussian noise to the data.

```
error = 0.5
data += pg.randn(*data.shape)*error
relError = error / data
```

Next, an instance of the forward operator is created. We could use it for calculating the synthetic data using f.response([10.5, 0.55]) or just f([10.5, 0.55]). We create a real-valued (R) inversion passing the forward operator, the data. A verbose boolean flag could be added to provide some output the inversion, another one prints more and saves files for debugging.

```
f = ExpModelling(x)
inv = pg.Inversion(f)
```

We create a real-valued logarithmic transformation and apply it to the model. Similar could be done for the data which are by default treated linearly. We then set the error level that is used for data weighting. It can be a float number or a vector of data length. One can also set a relative error. Finally, we define the inversion style as Marquardt scheme (pure local damping with decreasing the regularization parameter subsequently) and start with a relatively large regularization strength to avoid overshoot. Finally run yields the coefficient vector and we plot some statistics.

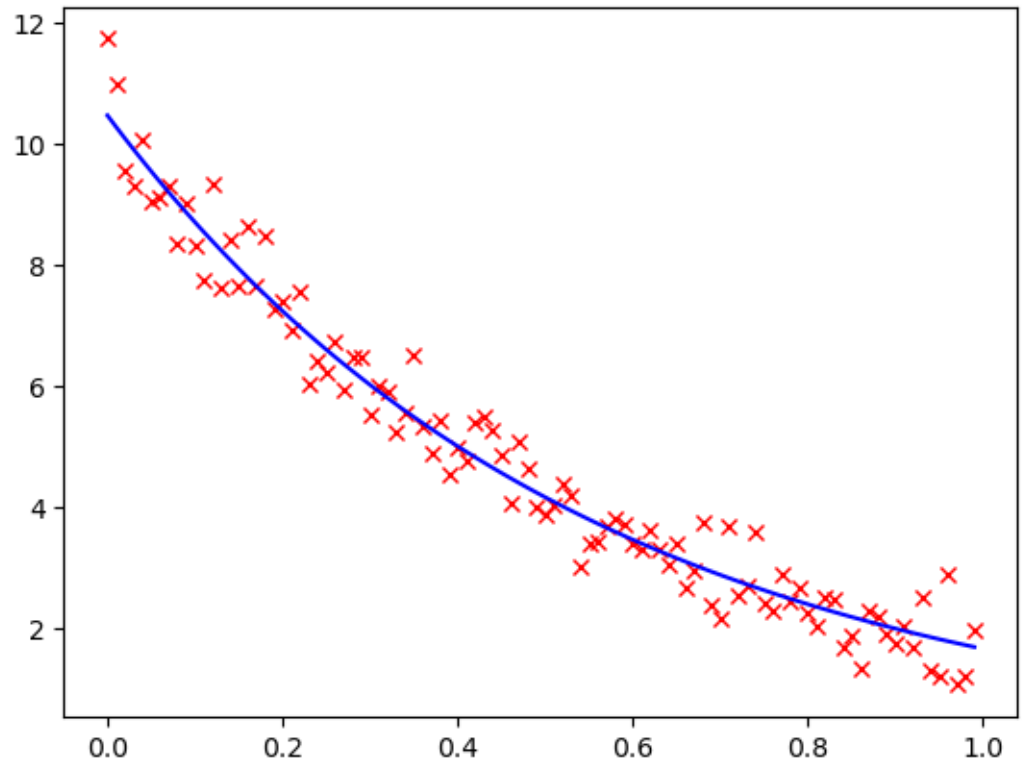
```
tLog = pg.trans.TransLog()
f.modelTrans = tLog
inv._inv.setMarquardtScheme()
inv._inv.setLambda(100)
coeff = inv.run(data, relError, verbose=True)
print(inv.relrms(), inv.chi2())
print(coeff)
```

```
fop: <__main__.ExpModelling object at 0x7fe878099ea0>
Data transformation: <pygimli.core._pygimli_.RTrans object at 0x7fe899886340>
Model transformation: <pygimli.core._pygimli_.RTransLog object at 0x7fe8780934a0>
min/max (data): 1.06/11.74
min/max (error): 4.26%/47.38%
min/max (start model): 1/3
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
inv.iter 2 ... chi^2 = 33.65 (dPhi = 54.42%) lam: 20
--------------------------------------------------------------------------------
inv.iter 3 ... chi^2 = 3.78 (dPhi = 88.77%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 4 ... chi^2 = 0.92 (dPhi = 75.53%) lam: 20.0


################################################################################
#                   Abort criterion reached: chi^2 <= 1 (0.92)                 #
################################################################################
16.442950902739785 0.9243295327672411
2 [10.475758144156835, 0.5412070551527863]
```

We see that after 5 iterations the absolute rms value equals the noise level corresponding to a chi-squared misfit value of 1 as it should be the case for synthetic data. The relative rms (in %) is less relevant here but can be for other applications. Additionally the ranges for model and model response are given and the objective function consisting of data misfit and model roughness times lambda. Note that due to the local regularization the second term does not contribute to Phi. Set verbose to True to see the whole course of inversion. The values of the coefficient vector (a GIMLi real vector) are as expected close (i.e. equivalent) to the synthetic model.

We finally create a plotting figure and plot both data and model response.

```
plt.figure()
plt.plot(x, data, 'rx', x, inv.response, 'b-')
```




```
[<matplotlib.lines.Line2D object at 0x7fe8aafbfdc0>, <matplotlib.lines.Line2D␣
↪object at 0x7fe8aafbfb50>]
```

The createMesh1D automatically attributed the markers 0 and 1 to the two model parameters A and X, respectively. Each marker leads to a region that can be individually treated, e.g. the starting value, lower or upper bounds, or all three at the same time (setParameters). This changes the model transformation which can of course be region-specific.

```
# f.region(0).setLowerBound(0.1)
# f.region(0).setStartModel(3)
# f.region(1).setParameters(0.3, 0.01, 1.0)
```

If these are set before the inversion is used, they are used automatically. We set the model by hand using the new starting model

```
# inv.setVerbose(True)
# inv.setModel(f.createStartModel())
# print(inv.run())
# inv.echoStatus()
```

The result is pretty much the same as before but for stronger equivalence or smoothness-constrained regularization prior information might help a lot.

7.4.4.2 Polyfit

This tutorial shows a flexible inversion with an own forward calculation that includes an own jacobian. We start with fitting a polynomial of degree P

$$f(x) = p_0 + p_1 x + \ldots + p_P x^P = \sum_{i=0}^{P} p_i x^i$$

to given data y. The unknown model is the coefficient vector $\mathbf{m} = [p_0, \ldots, p_P]$. The vectorized function for a vector $\mathbf{x} = [x_1, \ldots, x_N]^T$ can be written as matrix-vector product

$$\mathbf{f}(\mathbf{x}) = \mathbf{A}\mathbf{x} \quad \text{with} \quad \mathbf{A} = \begin{bmatrix} 1 & x_1 & \ldots & x_1^P \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_N & \ldots & x_N^P \end{bmatrix} = [\mathbf{1} \quad \mathbf{x} \quad \mathbf{x}^2 \ldots \mathbf{x}^P] \,.$$

We set up the modelling operator, i.e. to return $\mathbf{f}(\mathbf{x})$ for given p_i, as a class derived from the modelling base class. The latter holds the main mimic of generating Jacobian, gradients by brute force. The only function to overwrite is cw{response()}.

Python is a very flexible language for programming and scripting and has many packages for numerical computing and graphical visualization. For this reason, we built Python bindings and compiled the library pygimli. As a main advantage, all classes can be used and derived. This makes the use of GIMLi very easy for non-programmers. All existing modelling classes can be used, but it is also easy to create new modelling classes.

We exemplify this by the preceding example.

First, the library must be imported.

To avoid name clashes with other libraries we suggest to import *pygimli* and alias it to an easy name (as usually done for numpy or matplotlib), e.g. by

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

The modelling class is derived from ModellingBase, a constructor is defined and the response function is defined. Due to the linearity of the problem we store the matrix **A**, which is also the Jacobian matrix and use it for the forward calculation. A second function is just added as reference. We overwrite the method createJacobian as we know it but do nothing in the actual computation. If **J** depends on **m** this function must be filled.

```
class FunctionModelling(pg.core.ModellingBase):
    def __init__(self, nc, xvec, verbose=False):
        pg.core.ModellingBase.__init__(self, verbose)
        self.x_ = xvec
        self.nc_ = nc
        nx = len(xvec)
        self.regionManager().setParameterCount(nc)
        self.jacobian().resize(nx, nc)
        for i in range(self.nc_):
            self.jacobian().setCol(i, pg.math.pow(self.x_, i))
```

(continues on next page)

(continued from previous page)

```
    def response(self, model):
        return self.jacobian() * model

    def responseDirect(self, model):
        y = pg.Vector(len(self.x_), model[0])

        for i in range(1, self.nc_):
            y += pg.math.pow(self.x_, i) * model[i]

        return y

    def createJacobian(self, model):
        pass  # if J depends on the model you should work here

    def startModel(self):
        return pg.Vector(self.nc_, 0.5)
```

Let us create some synthetic data for some x values

```
x = np.arange(0., 10., 0.5)
y = 1.1 + 2.1 * x - 0.2 * x**2
noise = 0.5
y += np.random.randn(len(y)) * noise
```

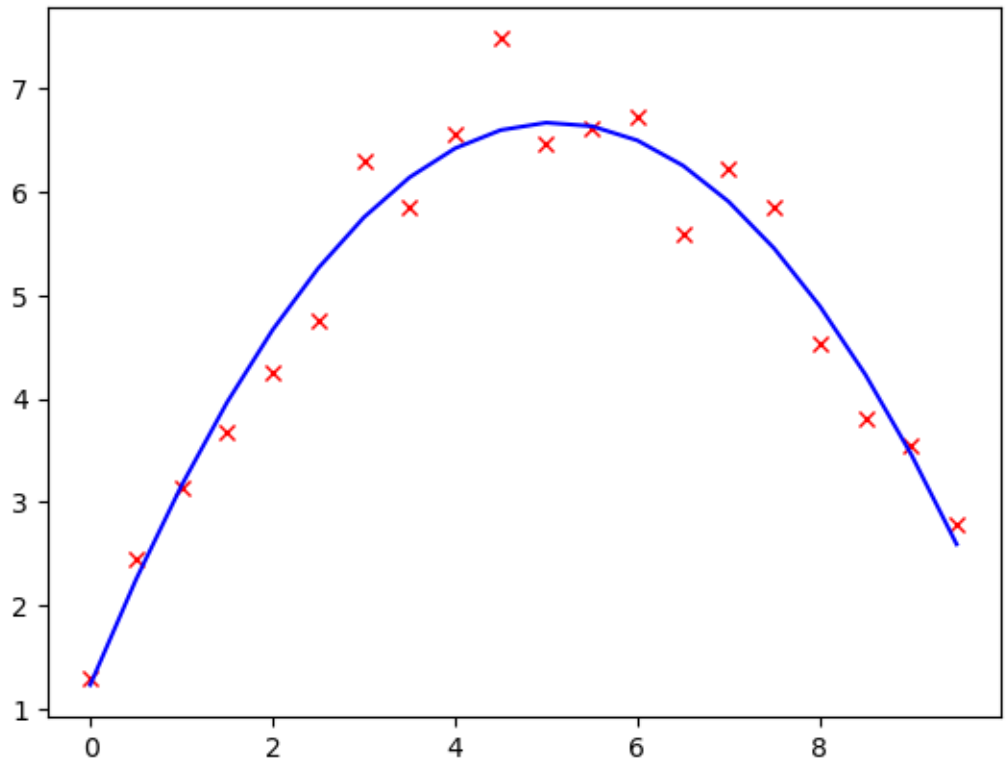
We now start by setting up the modelling operator, and inversion and run it.

```
fop = FunctionModelling(3, x)

# initialize inversion with data and forward operator and set␣
↪options
inv = pg.core.Inversion(y, fop)

# constant absolute error of 0.01 is 1% (not necessary, only for␣
↪chi^2)
inv.setAbsoluteError(noise)

# the problem is well-posed and does not need any regularization
inv.setLambda(0)

# actual inversion run yielding coefficient model
coeff = inv.run()
print(coeff)
```

```
3 [1.2361718033308833, 2.133871438905104, -0.20957213916471507]
```

The result is easily plotted by

```
plt.plot(x, y, 'rx', x, inv.response(), 'b-')
plt.show()
```



7.4.4.3 VES inversion for a blocky model

This tutorial shows how an built-in forward operator is used for inversion. A DC 1D (VES) modelling is used to generate data, noisify and invert them.

We import numpy, matplotlib and the 1D plotting function

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

import pygimli as pg
from pygimli.viewer.mpl import drawModel1D
```

some definitions before (model, data and error)

```
nlay = 4  # number of layers
lam = 200.  # (initial) regularization parameter
errPerc = 3.  # relative error of 3 percent
ab2 = np.logspace(-1, 2, 50)  # AB/2 distance (current electrodes)
mn2 = ab2 / 3.  # MN/2 distance (potential electrodes)
```

initialize the forward modelling operator

```
f = pg.core.DC1dModelling(nlay, ab2, mn2)
```

other ways are by specifying a Data Container or am/an/bm/bn distances

```
synres = [100., 500., 20., 800.]  # synthetic resistivity
synthk = [0.5, 3.5, 6.]  # synthetic thickness (nlay-th layer is␣
↪infinite)
```

the forward operator can be called by f.response(model) or simply f(model)

```
rhoa = f(synthk+synres)
rhoa = rhoa * (pg.randn(len(rhoa), seed=0) * errPerc / 100. + 1.)
```

create some transformations used for inversion

```
transThk = pg.trans.TransLog()  # log-transform ensures thk>0
transRho = pg.trans.TransLogLU(1, 1000)  # lower and upper bound
transRhoa = pg.trans.TransLog()  # log transformation for data
```

set model transformation for thickness and resistivity

```
f.region(0).setTransModel(transThk)  # 0=thickness
f.region(1).setTransModel(transRho)  # 1=resistivity
```

generate start model values from median app. resistivity & spread

```
paraDepth = max(ab2) / 3.  # rule-of-thumb for Wenner/Schlumberger
f.region(0).setStartValue(paraDepth / nlay / 2)
f.region(1).setStartValue(np.median(rhoa))
```

set up inversion

```
inv = pg.core.Inversion(rhoa, f, transRhoa, True)  # data vector, fop,␣
↪verbose
# could also be set by inv.setTransData(transRhoa)
```

set error model, regularization strength and Marquardt scheme

```
inv.setRelativeError(errPerc / 100.0)  # alternative: setAbsoluteError in␣
↪Ohmm
inv.setLambda(lam)  # (initial) regularization parameter
inv.setMarquardtScheme(0.9)  # decrease lambda by factor 0.9
model = f.createStartVector()  # creates from region start value
model[nlay] *= 1.5  # change default model by changing 2nd layer␣
↪resistivity
inv.setModel(model)  #
```

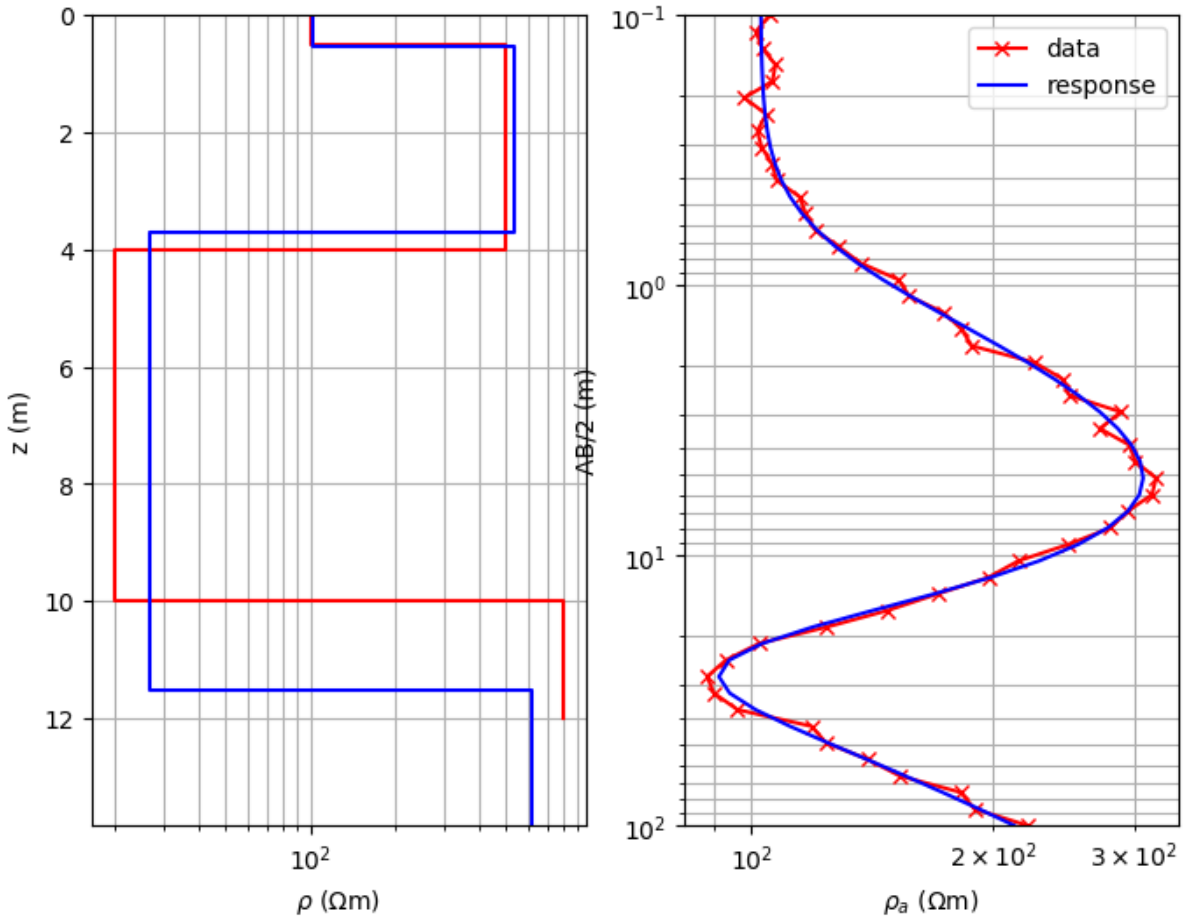
run actual inversion and extract resistivity and thickness

```
model = inv.run()  # result is a pg.Vector, but compatible to numpy␣
↪array
res, thk = model[nlay-1:nlay*2-1], model[0:nlay-1]
print('rrms={:.2f}%, chi^2={:.3f}'.format(inv.relrms(), inv.chi2()))
```

```
rrms=3.21%, chi^2=1.136
```

show estimated&synthetic models and data with model response in 2 subplots

```
fig, ax = plt.subplots(ncols=2, figsize=(8, 6))  # two-column figure
drawModel1D(ax[0], synthk, synres, plot='semilogx', color='r')
drawModel1D(ax[0], thk, res, color='b')
ax[0].grid(True, which='both')
ax[0].set_ylabel('z (m)')
ax[0].set_xlabel(r'$\rho$ ($\Omega$m)')
ax[1].loglog(rhoa, ab2, 'rx-', label='data')  # sounding curve
ax[1].loglog(inv.response(), ab2, 'b-', label='response')
ax[1].set_ylim((max(ab2), min(ab2)))  # downwards according to penetration
ax[1].grid(True, which='both')
ax[1].set_xlabel(r'$\rho_a$ ($\Omega$m)')
ax[1].set_ylabel('AB/2 (m)')
ax[1].legend(loc='best')
plt.show()
```



7.4.4.4 VES inversion for a smooth model

This tutorial shows how an built-in forward operator is used for an Occam type (smoothness-constrained) inversion with fixed regularization (most natural). A direct current (DC) one-dimensional (1D) VES (vertical electric sounding) modelling operator is used to generate data, add noise and inversion.

We import numpy numerics, mpl plotting, pygimli and the 1D plotting function

```
import numpy as np
np.random.seed(1337)

import matplotlib.pyplot as plt

import pygimli as pg
from pygimli.viewer.mpl import drawModel1D
```

```
synres = [100., 500., 20., 800.]  # synthetic resistivity
synthk = [4, 6, 10]  # synthetic thickness (lay layer is infinite)
ab2 = np.logspace(-1, 2, 25)  # 0.1 to 100 in 25 steps (8 points per␣
↪decade)
fBlock = pg.core.DC1dModelling(len(synres), ab2, ab2/3)
rhoa = fBlock(synthk+synres)
# The data are noisified using a
errPerc = 3.  # relative error of 3 percent
rhoa = rhoa * (np.random.randn(len(rhoa)) * errPerc / 100. + 1.)
```

The forward operator can be called by f.response(model) or simply f(model)

```
thk = np.logspace(-0.5, 0.5, 30)
f = pg.core.DC1dRhoModelling(thk, ab2, ab2/3)
```

Create some transformations used for inversion

```
transRho = pg.trans.TransLogLU(1, 1000)  # lower and upper bound
transRhoa = pg.trans.TransLog()  # log transformation also for data
```

Set up inversion

```
inv = pg.core.Inversion(rhoa, f, transRhoa, transRho, False)  # data vector,␣
↪f, ...
# The transformations can also be omitted and set individually by
# inv.setTransData(transRhoa)
# inv.setTransModel(transRho)
inv.setRelativeError(errPerc / 100.0)
```

Create a homogeneous starting model

```
model = pg.Vector(len(thk)+1, np.median(rhoa))  # uniform values
inv.setModel(model)  #
```

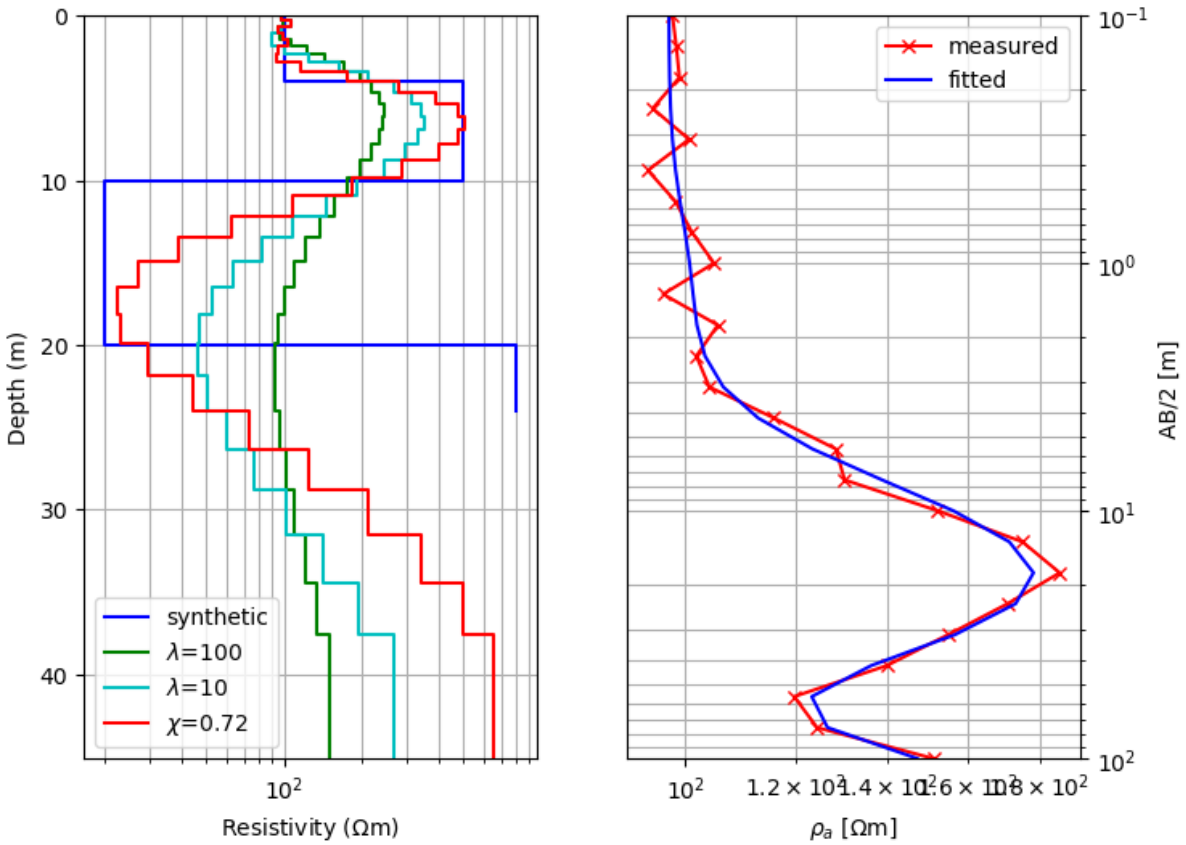
Set pretty large regularization strength and run inversion

```
print("inversion with lam=200")
inv.setLambda(100)
res100 = inv.run()  # result is a pg.Vector, but compatible to numpy␣
↪array
print('rrms={:.2f}%, chi^2={:.3f}'.format(inv.relrms(), inv.chi2()))
# Decrease the regularization (smoothness) and start (from old␣
↪result)
print("inversion with lam=20")
inv.setLambda(10)
res10 = inv.run()  # result is a pg.Vector, but compatible to numpy␣
↪array
print('rrms={:.2f}%, chi^2={:.3f}'.format(inv.relrms(), inv.chi2()))
# We now optimize lambda such that data are fitted within noise␣
↪(chi^2=1)
print("chi^2=1 optimized inversion")
resChi = inv.runChi1()  # ends up in a lambda of about 3
print("optimized lambda value:", inv.getLambda())
print('rrms={:.2f}%, chi^2={:.3f}'.format(inv.relrms(), inv.chi2()))
```

```
inversion with lam=200
rrms=5.49%, chi^2=3.373
inversion with lam=20
rrms=3.64%, chi^2=1.454
chi^2=1 optimized inversion
optimized lambda value: 0.7179364718731468
rrms=3.01%, chi^2=0.994
```

Show model (inverted and synthetic) as well as model response and data

```
fig, ax = plt.subplots(ncols=2, figsize=(8, 6))  # two-column figure
drawModel1D(ax[0], synthk, synres, color='b', label='synthetic',
            plot='semilogx')
drawModel1D(ax[0], thk, res100, color='g', label=r'$\lambda$=100')
drawModel1D(ax[0], thk, res10, color='c', label=r'$\lambda$=10')
drawModel1D(ax[0], thk, resChi, color='r',
            label=r'$\chi$={:.2f}'.format(inv.getLambda()))
ax[0].grid(True, which='both')
ax[0].legend(loc='best')
ax[1].loglog(rhoa, ab2, 'rx-', label='measured')
ax[1].loglog(inv.response(), ab2, 'b-', label='fitted')
ax[1].set_ylim((max(ab2), min(ab2)))
ax[1].grid(True, which='both')
ax[1].set_xlabel(r'$\rho_a$ [$\Omega$m]')
ax[1].set_ylabel('AB/2 [m]')
ax[1].yaxis.set_label_position('right')
ax[1].yaxis.set_ticks_position('right')
ax[1].legend(loc='best')
plt.show()
```



Total running time of the script: (0 minutes 21.878 seconds)

7.4.4.5 Regularization - concepts explained

In geophysical inversion, we minimize the data objective functional as the L2 norm of the misfit between data d and the forward response f of the model m, weighted by the data error $epsilon$:

$$\Phi_d = \sum_i^N \left(\frac{d_i - f_i(m)}{\varepsilon_i} \right)^2 = \|W_d(d - f(m))\|^2$$

As this minimization problem is non-unique and ill-posed, we introduce a regularization term Phi, weighted by a regularization parameter $lambda$:

$$\Phi = \Phi_d + \lambda\Phi_m$$

The regularization strength $lambda$ should be chosen so that the data are fitted within noise, i.e. $chi^2=Phi_d/N=1$.

In the term $Phi-m$ we put our expectations to the model, e.g. to be close to any prior model. In many cases we do not have much information and aim for the smoothest model that is able to fit our data. We decribe it by the operator W_m:

$$\Phi_m = \|W_m(m - m_{ref})\|^2$$

The regularization operator is defined by some constraint operator C weighted by some weighting function w so that $W_m=mbox{diag}(w) C$. The operator C can be a discrete smoothness operator, or the identity to keep the model close to the reference model m_{ref}.

We start with importing the numpy, matplotlib and pygimli libraries

```
import numpy as np
import matplotlib.pyplot as plt
import pygimli as pg
import pygimli.meshtools as mt
from pygimli.math.matrix import GeostatisticConstraintsMatrix
from pygimli.core.math import symlog
```

Regularization drives the model where the data are too weak to constrain the model. In order to explain different kinds of regularization (also called constraints), we use a very simple mapping forward operator: The values at certain positions are picked.

```
from pygimli.frameworks import PriorModelling
```

Implementation 1. determine the indices where the cells are

```
ind = [mesh.findCell(po).id() for po in pos]
```

2. forward response: take the model at indices

```
response = model[ind]
```

3. Jacobian matrix

```
J = pg.SparseMapMatrix()
J.resize(len(ind), mesh.cellCount())
for i, n in enumerate(self.ind):
    self.J.setVal(i, n, 1.0)
```

We exemplify this on behalf of a simple triangular mesh in a rectangular domain.

```
rect = mt.createRectangle(start=[0, -10], end=[10, 0])
mesh = mt.createMesh(rect, quality=34.5, area=0.3)
print(mesh)
```

```
Mesh: Nodes: 377 Cells: 684 Boundaries: 1060
```

We define two positions where we associate two arbitrary values.

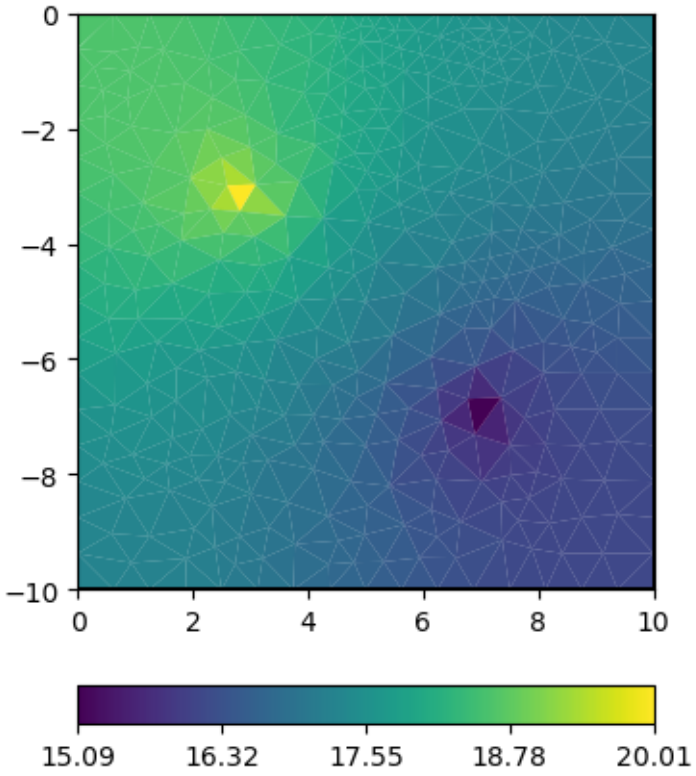
```
pos = [[3, -3], [7, -7]]
vals = np.array([20., 15.])
fop = PriorModelling(mesh, pos)
```

We set up an inversion instance with the forward operator and prepare the keywords for running the inversion always the same way: - the data vector - the error vector (as relative error) - a starting model value (could also be vector)

```
inv = pg.Inversion(fop=fop, verbose=False)
invkw = dict(dataVals=vals, errorVals=np.ones_like(vals)*0.03, startModel=10)
```

Classical smoothness constraints

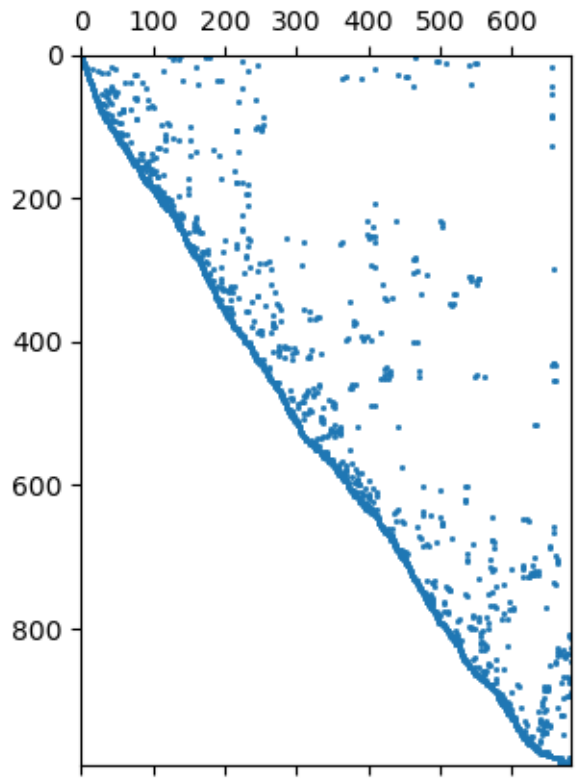
```
inv.setRegularization(cType=1)  # the default
result = inv.run(**invkw)
pg.show(mesh, result);
```




```
(<matplotlib.axes._subplots.AxesSubplot object at 0x7fe8f5515af0>, <matplotlib.
↪colorbar.Colorbar object at 0x7fe8997a9a90>)
```

We will have a closer look at the regularization matrix C.

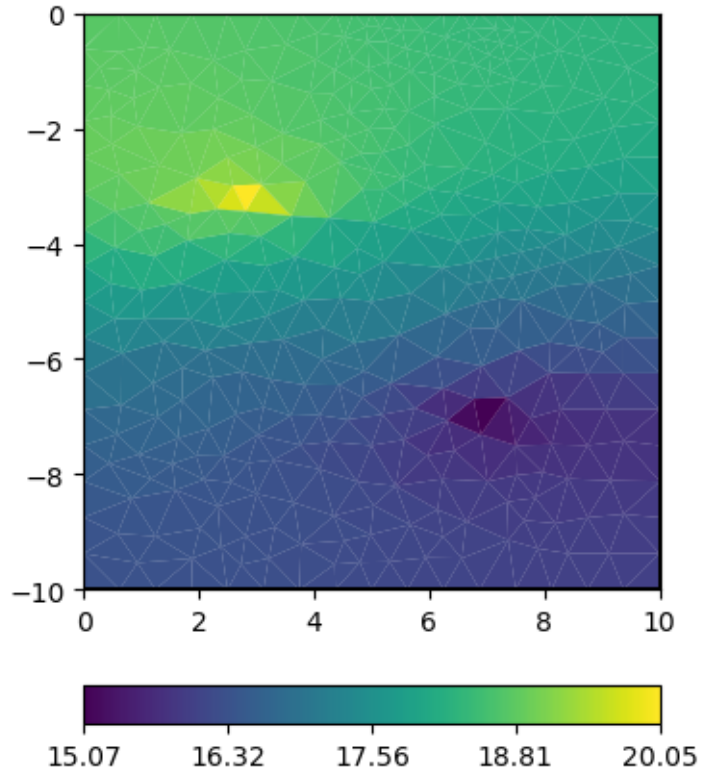
```
C = fop.constraints()
print(C.rows(), C.cols(), mesh)
ax, _ = pg.show(fop.constraints(), markersize=1)

row = C.row(111)
nz = np.nonzero(row)[0]
print(nz, row[nz])
```




```
992 684 Mesh: Nodes: 377 Cells: 684 Boundaries: 1060
[ 48 116] 2 [1.0, -1.0]
```

How does that change the regularization matrix C?

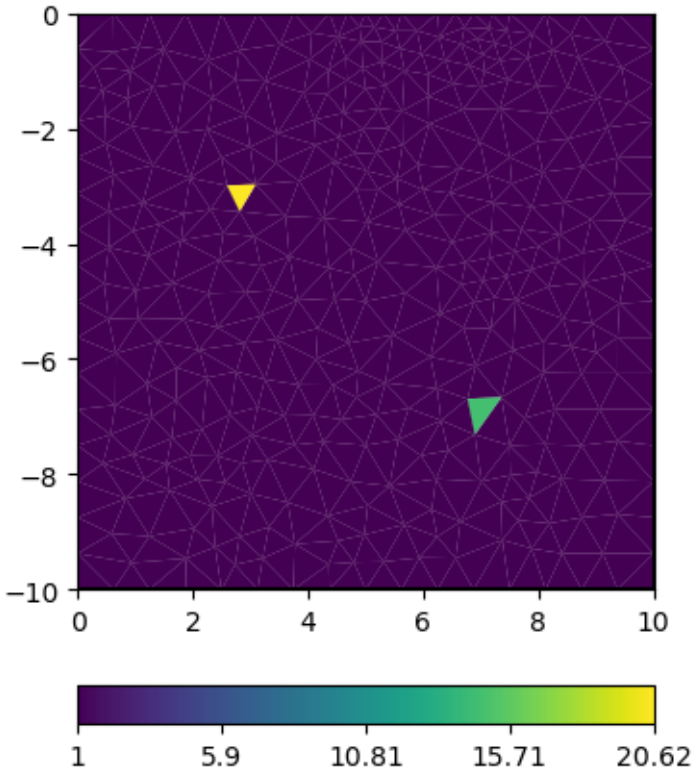
```
inv.setRegularization(cType=1, zWeight=0.2)  # the default
result = inv.run(**invkw)
pg.show(mesh, result)

RM = fop.regionManager()
cw = RM.constraintWeights()
print(min(cw), max(cw))
```




```
0.19999999999999996 1.0
```

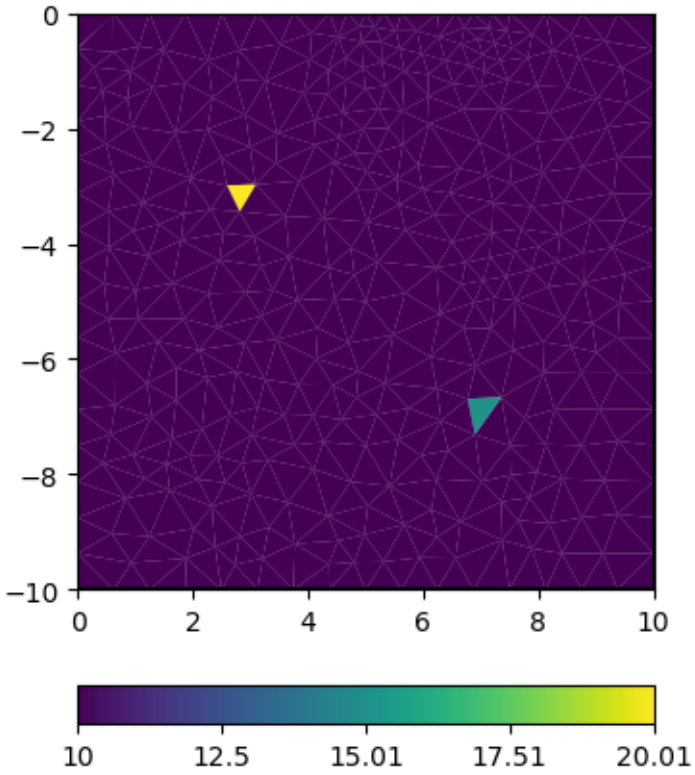
Now we try some other regularization options.

```
inv.setRegularization(cType=0)  # damping difference to starting model
result = inv.run(**invkw)
ax, _ = pg.show(mesh, result)
```



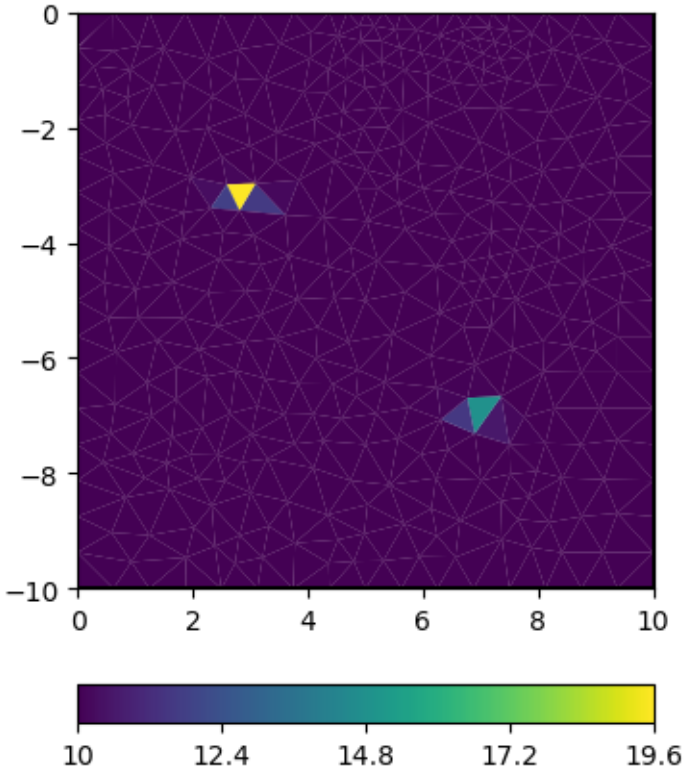
Obviously, the damping keeps the model small ($log 1=0$) as the starting model is NOT a reference model by default. We will enable this by specifying the `isReference` switch.

```
invkw["isReference"] = True
result = inv.run(**invkw)
ax, cb = pg.show(mesh, result)
```



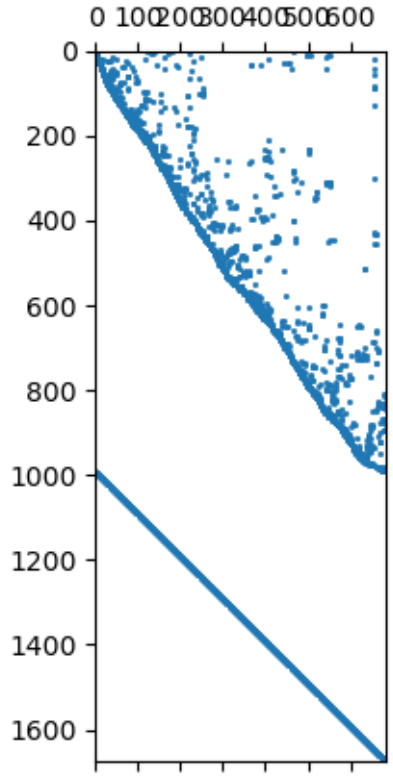
`cType=10` means a mix between 1st order smoothness (1) and damping (0)

```
inv.setRegularization(cType=10)  # mix of 1st order smoothing and damping
result = inv.run(**invkw)
ax, _ = pg.show(mesh, result)
```



In the matrix both contributions are under each other

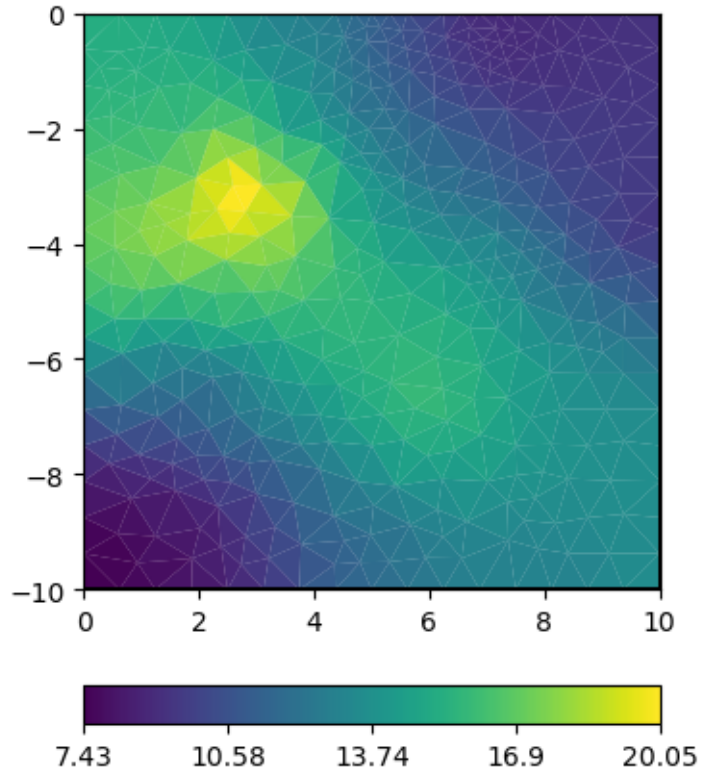
```
C = fop.constraints()
print(C.rows(), C.cols())
print(mesh)
ax, _ = pg.show(fop.constraints(), markersize=1)
```




```
1676 684
Mesh: Nodes: 377 Cells: 684 Boundaries: 1060
```

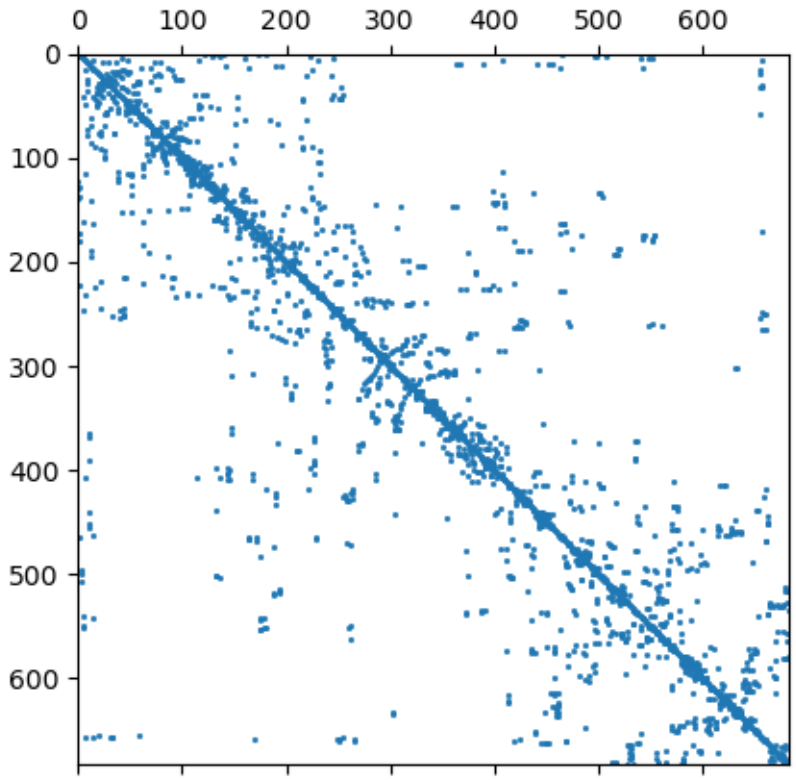
We see that we have the first order smoothness and the identity matrix below each other. We can also use a second-order (-1 2 -1) smoothness operator by `cType=2`.

```
inv.setRegularization(cType=2)  # 2nd order smoothing
result = inv.run(**invkw)
ax, _ = pg.show(mesh, result)
```



We have a closer look at the constraints matrix

```
C = fop.constraints()
print(C.rows(), C.cols(), mesh)
ax, _ = pg.show(C, markersize=1)
```



```
684 684 Mesh: Nodes: 377 Cells: 684 Boundaries: 1060
```

It looks like a Laplace operator and seems to have a wider range compared to first-order smoothness.

Geostatistical regularization

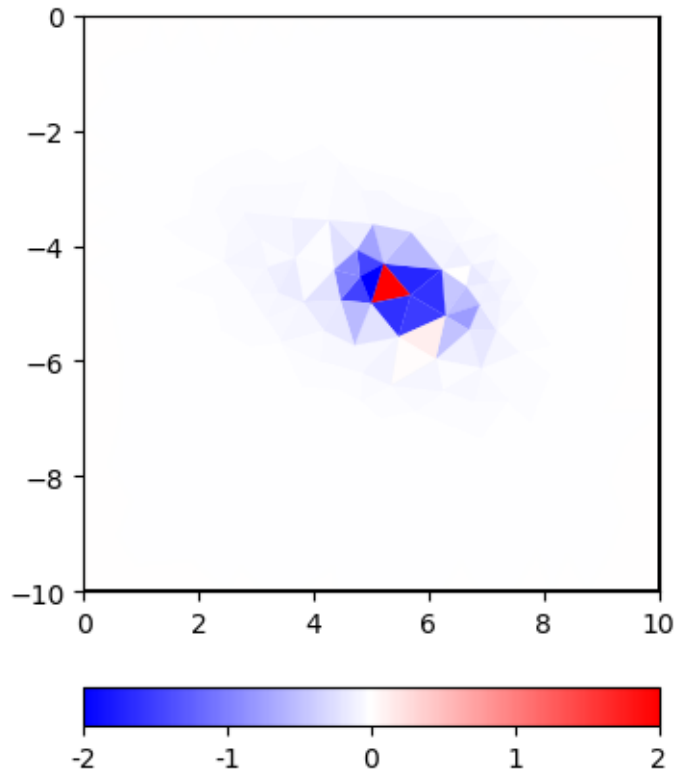
The idea is that not only neighbors are correlated to each other but to have a wider correlation by using an operator

$$\mathbf{C}_{\mathrm{M},ij} = \sigma^2 \exp\left(-\sqrt{\left(\frac{\mathbf{H}^x_{ij}}{I_x}\right)^2 + \left(\frac{\mathbf{H}^y_{ij}}{I_y}\right)^2}\right).$$

More details can be found in https://www.pygimli.org/_tutorials_auto/3_inversion/plot_6-geostatConstraints.html

We generate such a matrix and multiply it with a zero vector of just one 1. For displaying the wide range of magnitudes we use the symlog function

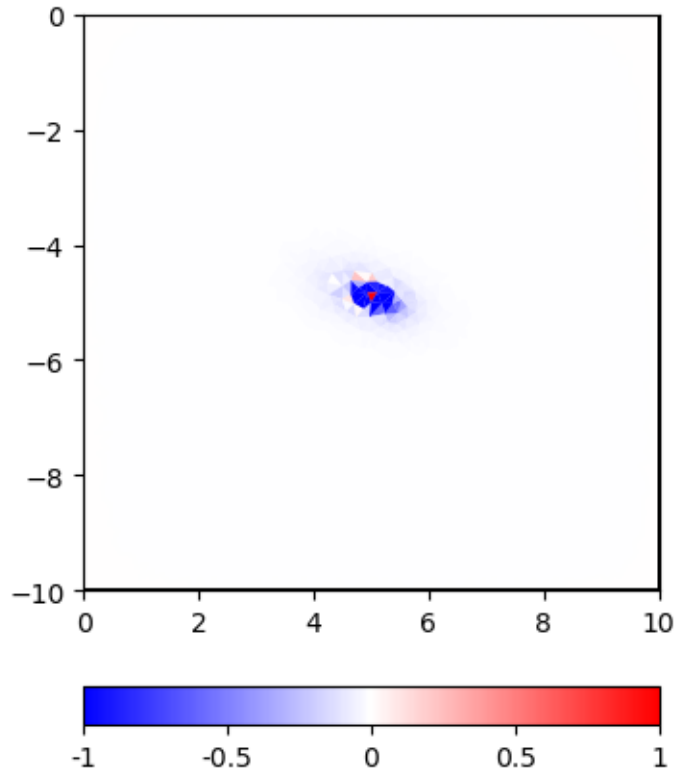
```
C = GeostatisticConstraintsMatrix(mesh=mesh, I=[8, 4], dip=-20)
print(C)

vec = pg.Vector(mesh.cellCount())
vec[mesh.findCell([5, -5]).id()] = 1.0
ax, _ = pg.show(mesh, symlog(C*vec, 1e-2), cMin=-2, cMax=2, cMap="bwr")
```




```
<pygimli.math.matrix.GeostatisticConstraintsMatrix object at 0x7fe8da959540>
```

For comparison, we use a much finer mesh and compute the same matrix

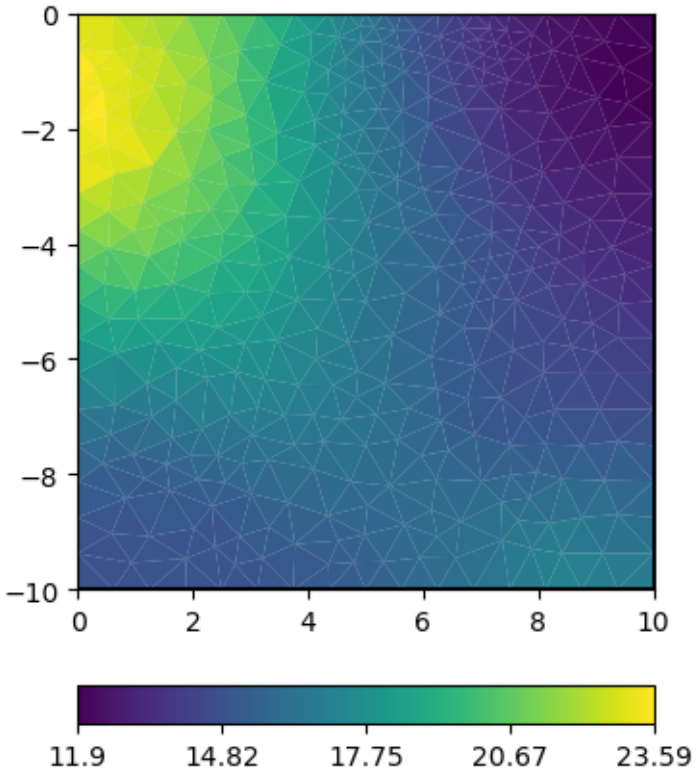
```
fineMesh = mt.createMesh(rect, area=0.03)
Cfine = GeostatisticConstraintsMatrix(mesh=fineMesh, I=[8, 4], dip=-20)
vec = pg.Vector(fineMesh.cellCount())
vec[fineMesh.findCell([5, -5]).id()] = 1.0
ax, _ = pg.show(fineMesh, symlog(Cfine*vec, 1e-2), cMin=-1, cMax=1, cMap="bwr")
```



Application

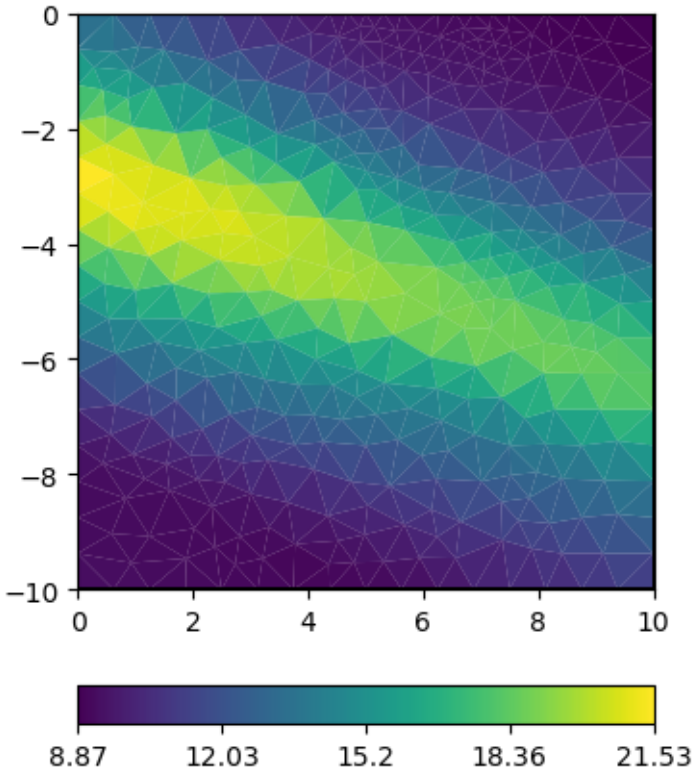
We can pass the correlation length directly to the inversion instance

```
inv.setRegularization(correlationLengths=[2, 2, 2])
result = inv.run(**invkw)
ax, cb = pg.show(mesh, result)
```



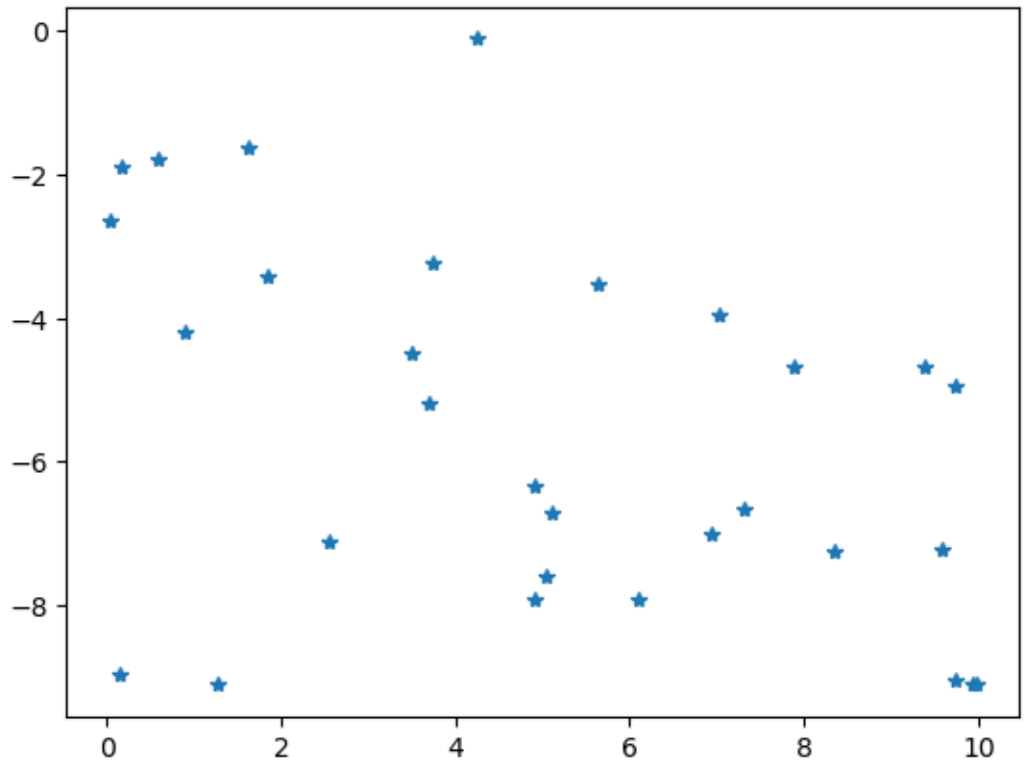
This look structurally similar to the second-order smoothness, but can drive values outside the expected range in regions of no data coverage. We change the correlation lengths and the dip to be inclining

```
inv.setRegularization(correlationLengths=[2, 0.5, 2], dip=-20)
result = inv.run(**invkw)
ax, cb = pg.show(mesh, result)
```



We now add many more points.

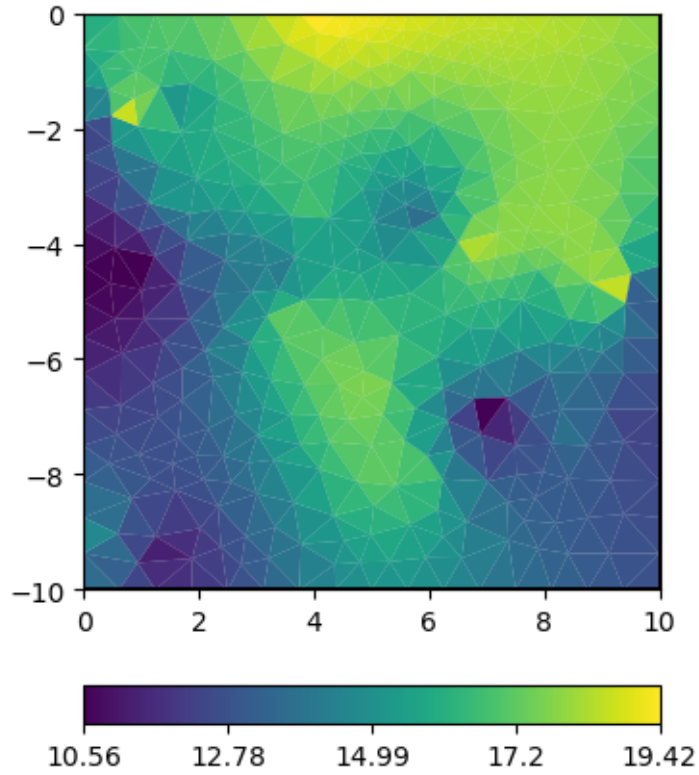
```
N = 30
x = np.random.rand(N) * 10
y = -np.random.rand(N) * 10
v = np.random.rand(N) * 10 + 10
plt.plot(x, y, "*")
```




```
[<matplotlib.lines.Line2D object at 0x7fe89a0de0a0>]
```

and repeat the above computations

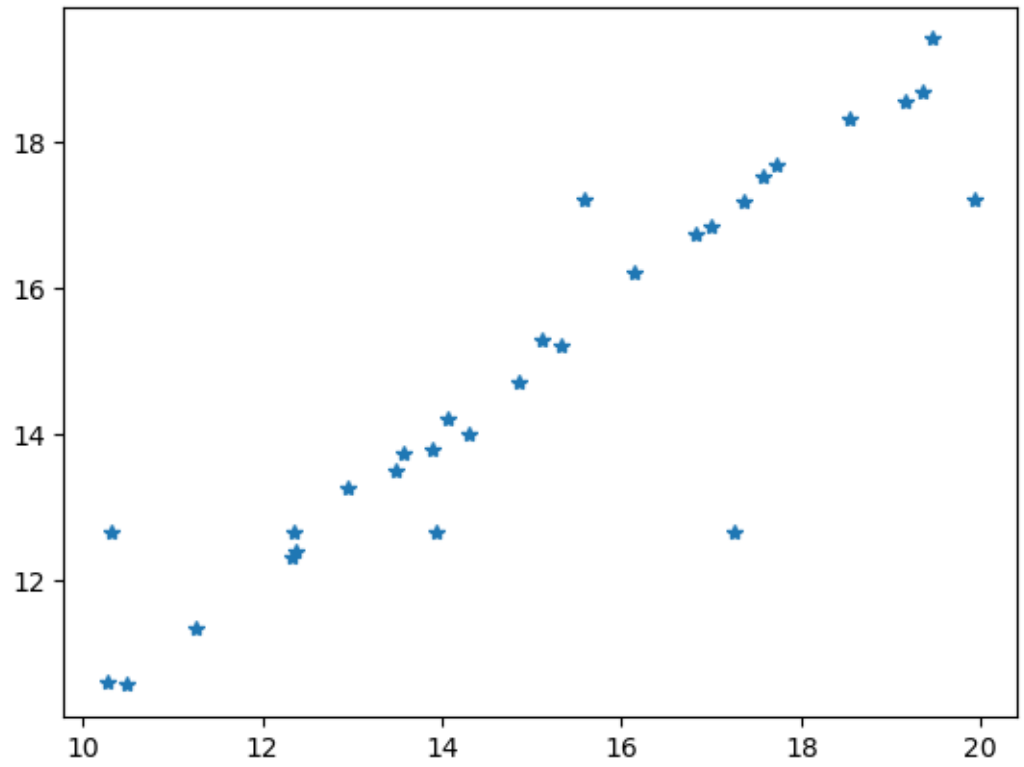
```
fop = PriorModelling(mesh, zip(x, y))
inv = pg.Inversion(fop=fop, verbose=True)
inv.setRegularization(correlationLengths=[4, 4])
result = inv.run(v, np.ones_like(v)*0.03, startModel=10)
ax, cb = pg.show(mesh, result)
```




```
fop: <pygimli.frameworks.modelling.PriorModelling object at 0x7fe8fa2bae50>
Data transformation: <pygimli.core._pygimli_.RTrans object at 0x7fe8da954520>
Model transformation (cumulative):
         0 <pygimli.core._pygimli_.RTransLogLU object at 0x7fe8da954520>
min/max (data): 10.28/19.93
min/max (error): 3%/3%
min/max (start model): 10/10
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
inv.iter 2 ... chi² = 6.16 (dPhi = 17.15%) lam: 20
--------------------------------------------------------------------------------
inv.iter 3 ... chi² = 6.15 (dPhi = 0.01%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 4 ... chi² = 6.15 (dPhi = 0.0%) lam: 20.0
################################################################################
#                  Abort criteria reached: dPhi = 0.0 (< 2.0%)                 #
################################################################################
```

Comparing the data with the model response is always a good idea.

```
plt.plot(v, inv.response, "*")
```

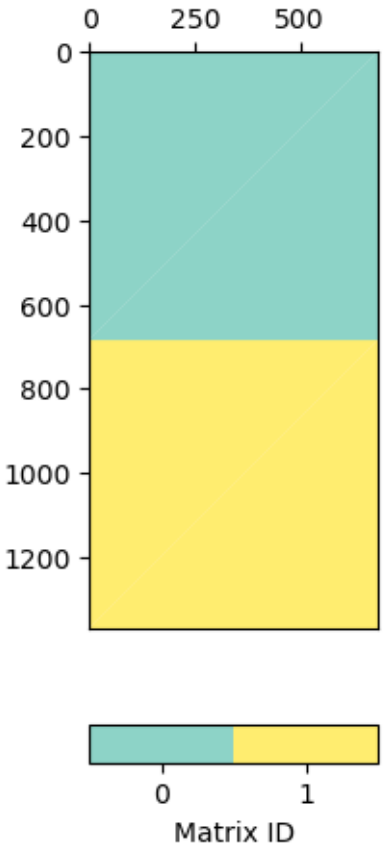



```
[<matplotlib.lines.Line2D object at 0x7fe899f1f9a0>]
```

Individual regularization operators

Say you want to combine geostatistic operators with a damping, you can create a block matrix pasting the matric vertically.

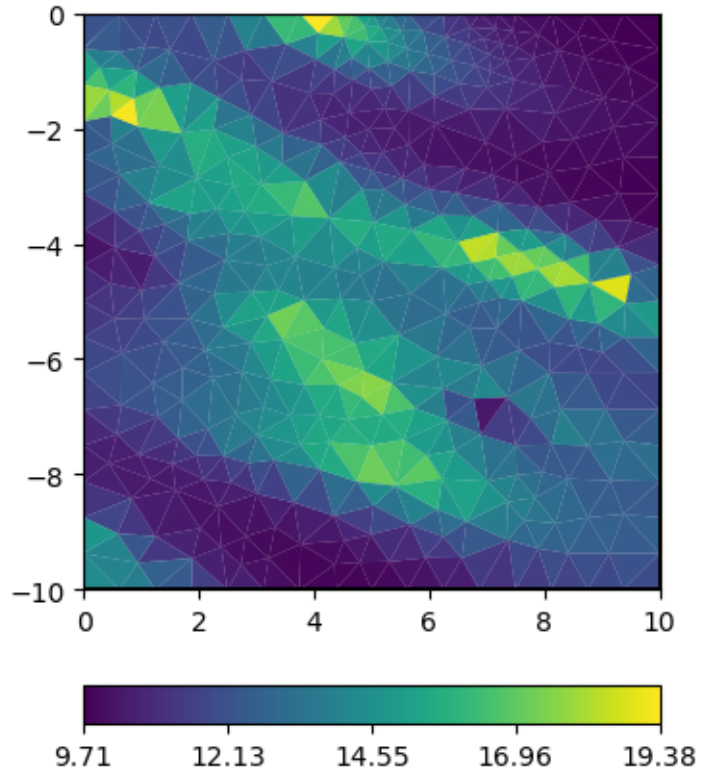
```
C = pg.matrix.BlockMatrix()
G = pg.matrix.GeostatisticConstraintsMatrix(mesh=mesh, I=[2, 0.5], dip=-20)
I = pg.matrix.IdentityMatrix(mesh.cellCount(), val=0.1)
C.addMatrix(G, 0, 0)
C.addMatrix(I, mesh.cellCount(), 0)
ax, _ = pg.show(C)
```



Note that in *pg.matrix* you find a lot of matrices and matrix generators.

We set this matrix directly and do the inversion.

```
fop.setConstraints(C)
result = inv.run(v, np.ones_like(v)*0.03, startModel=10, isReference=True)
ax, cb = pg.show(mesh, result)
```




```
fop: <pygimli.frameworks.modelling.PriorModelling object at 0x7fe8fa2bae50>
Data transformation: <pygimli.core._pygimli_.RTrans object at 0x7fe8ab142be0>
Model transformation (cumulative):
         0 <pygimli.core._pygimli_.RTransLogLU object at 0x7fe8997ae940>
min/max (data): 10.28/19.93
min/max (error): 3%/3%
min/max (start model): 10/10
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
inv.iter 2 ... chi² = 5.95 (dPhi = 23.43%) lam: 20
--------------------------------------------------------------------------------
inv.iter 3 ... chi² = 5.95 (dPhi = 0.02%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 4 ... chi² = 5.95 (dPhi = 0.0%) lam: 20.0
################################################################################
#                  Abort criteria reached: dPhi = 0.0 (< 2.0%)                 #
################################################################################
```

If you are using a method manager, you access the inversion instance by *mgr.inv* and the forward operator by *mgr.fop*.

Note: Take-away messages

- regularization drives the model where data are weak
- think and play with your assumptions to the model
- there are several predefined options
- geostatistical regularization can be superior, because: - it is mesh-independent - it better fills the

data gaps (e.g. 3D inversion of 2D profiles)

Total running time of the script: (0 minutes 31.201 seconds)

7.4.4.6 Geostatistical regularization

In this example we illustrate the use of geostatistical constraints on irregular meshes as presented by [**?**], compared to classical smoothness operators of first or second kind.

The elements of the covariance matrix $\mathbf{C}_\mathrm{M}$ are defined by the distances H between the model cells i and j into the three directions

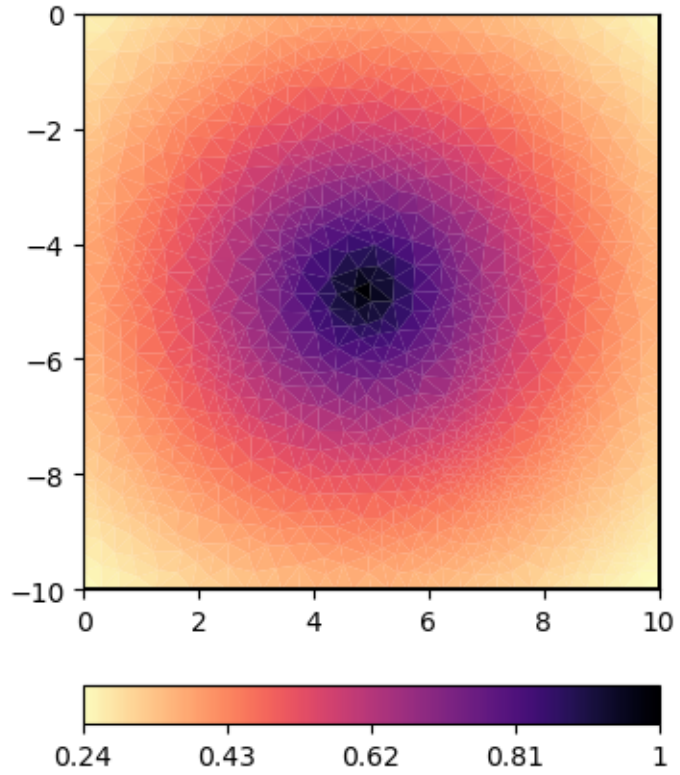
$$\mathbf{C}_{\mathrm{M},ij} = \sigma^2 \exp\left(-3\sqrt{\left(\frac{\mathbf{H}_{ij}^x}{I_x}\right)^2 + \left(\frac{\mathbf{H}_{ij}^y}{I_y}\right)^2 + \left(\frac{\mathbf{H}_{ij}^z}{I_z}\right)^2}\right).$$

It defines the correlation between model cells as a function of correlation lenghts (ranges) I_x, I_y, and I_z. Of course, the orientation of the coordinate axes is arbitrary and can be chosen by rotation. Let us illustrate this by a simple mesh:

Computing covariance and constraint matrices

We create a simple mesh using a box geometry

```
import matplotlib.pyplot as plt
from matplotlib.patches import CirclePolygon
from matplotlib.collections import PatchCollection
from matplotlib.colors import LogNorm
import numpy as np
import pygimli as pg
import pygimli.meshtools as mt

# We create a rectangular domain and mesh it with small triangles
rect = mt.createRectangle(start=[0, -10], end=[10, 0])
mesh = mt.createMesh(rect, quality=34.5, area=0.1)

# We compute such a covariance matrix by calling
CM = pg.utils.covarianceMatrix(mesh, I=5)  # I taken for both x and y
# We search for the cell where the midpoint (5, -5) is located in
ind = mesh.findCell([5, -5]).id()
# and plot the according column using index access (numpy)
ax, cb = pg.show(mesh, CM[:, ind], cMap="magma_r")
```



According to inverse theory, we use the square root of the covariance matrix as single-side regularization matrix C. It is computed by using an eigenvalue decomposition

$$\mathbf{C}_\mathrm{M} = \mathbf{QDQ}^T$$

based on LAPACK (numpy.linalg). The inverse square root is defined by

$$\mathbf{C}_\mathrm{M}^{-0.5} = \mathbf{QD}^{-0.5}\mathbf{Q}^T$$

In order to avoid a matrix inverse (square root), a special matrix is derived doing the decomposition and storing the eigenvectors and eigenvalues values. A multiplication is done by multiplying with Q and scaling with the diagonal. This matrix is implemented in the matrix module by the class `pg.matrix.Cm05Matrix`

```
Cm05 = pg.matrix.Cm05Matrix(CM)
```

However, this matrix does not return a zero vector for a constant vector

```
out = Cm05 * pg.Vector(mesh.cellCount(), 1.0)
print("min/max value ", min(out), max(out))
```

```
min/max value  0.02159243432153806 0.20503355104238655
```

as desired for a roughness operator. Therefore, an additional matrix called `pg.matrix.GeostatisticalConstraintsMatrix` was implemented where this spur is corrected for. It is, like the correlation matrix, created by a mesh, a list of correlation lengths I, a dip angle that distorts the x/y plane and a strike angle towards the third direction.

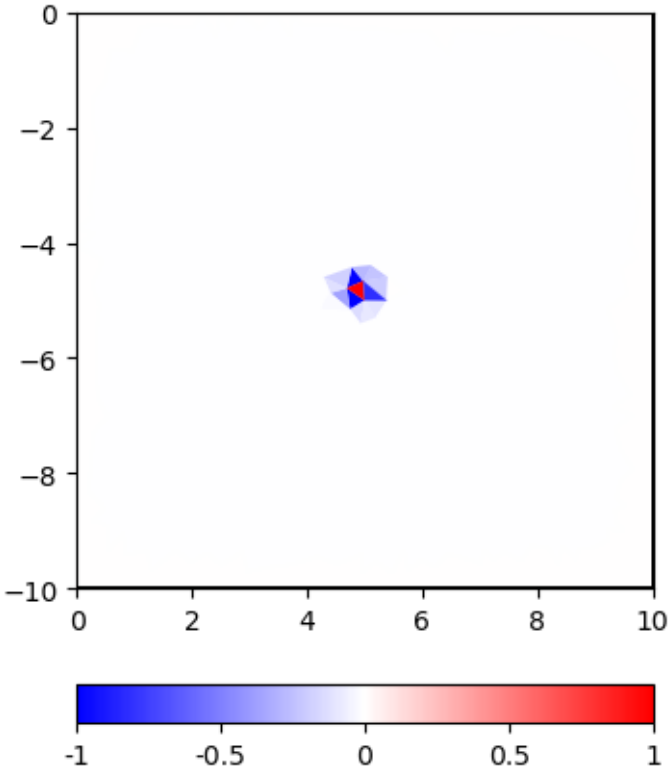
```
C = pg.matrix.GeostatisticConstraintsMatrix(mesh=mesh, I=5)
```

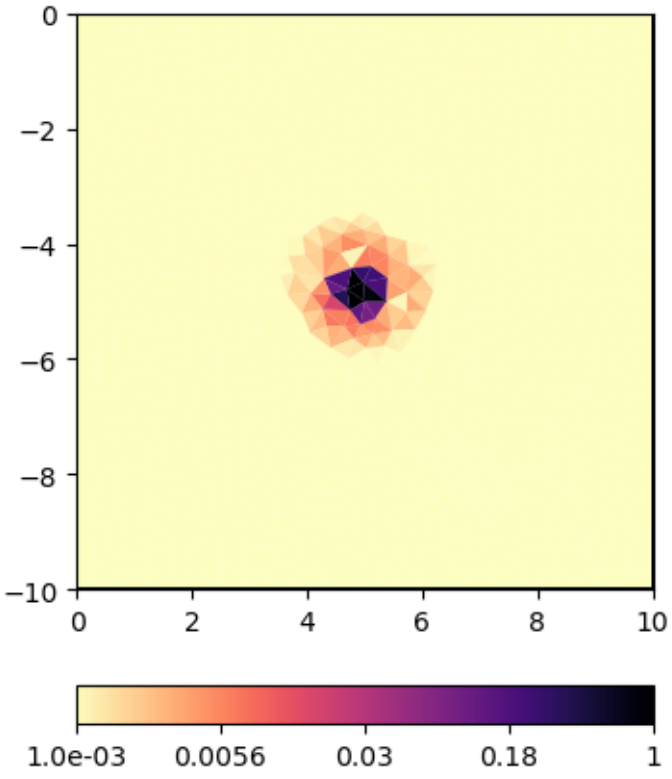
In order to extract a column, we generate a vector with a single 1, multiply

```
vec = pg.Vector(mesh.cellCount())
vec[ind] = 1.0
cor = C * vec
```

and plot it using a linear or logarithmic scale

```
kwLin = dict(cMin=-1, cMax=1, cMap="bwr")
ax, cb = pg.show(mesh, cor, **kwLin)
kwLog = dict(cMin=1e-3, cMax=1, cMap="magma_r", logScale=True)
ax, cb = pg.show(mesh, pg.abs(cor), **kwLog)
```

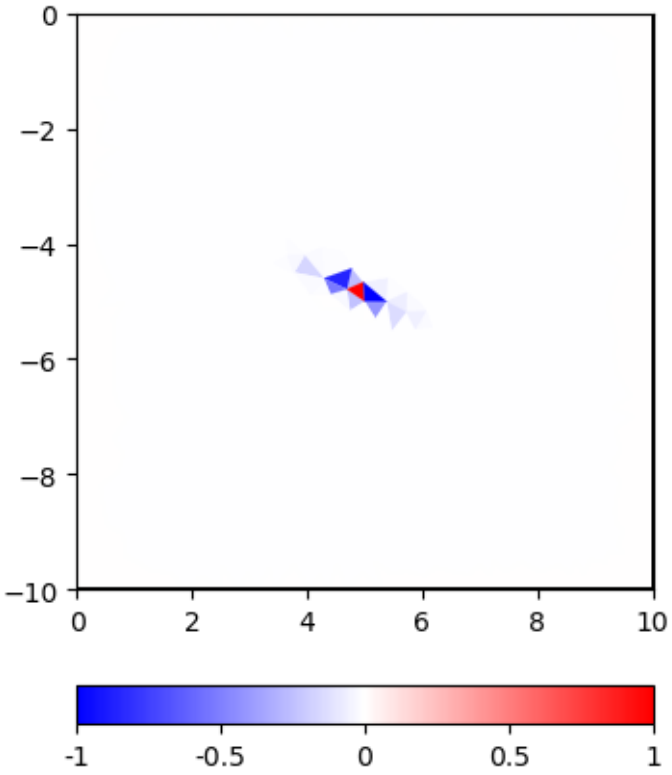
- 

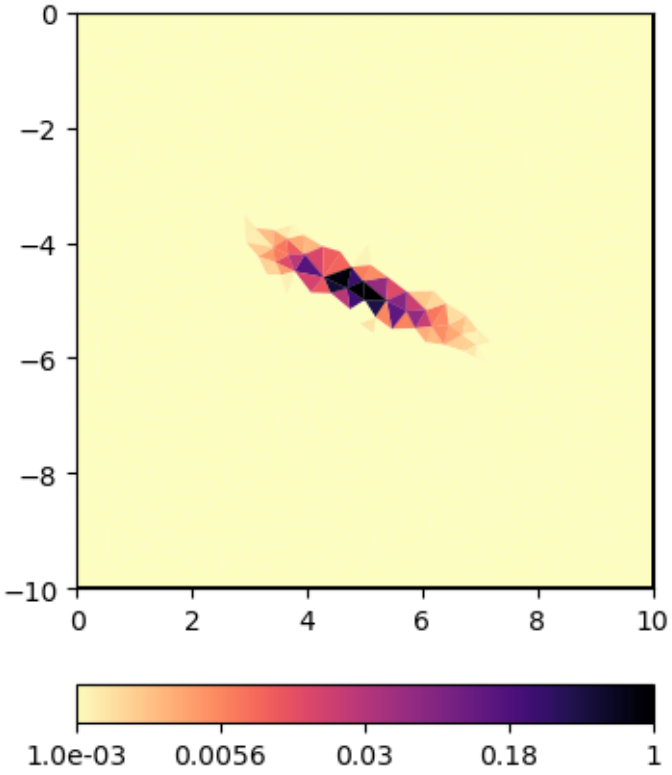


The constraints have a rather small footprint compared to the correlation if one considers values below a certain threshold as insignificant.

Such a matrix can also be defined for different ranges and dip angles, e.g.

```
Cdip = pg.matrix.GeostatisticConstraintsMatrix(mesh=mesh, I=[9, 2], dip=-25)
ax, cb = pg.show(mesh, Cdip * vec, **kwLin)
ax, cb = pg.show(mesh, pg.abs(Cdip * vec), **kwLog)
```





Even in the linear scale, but more in the log scale one can see the regularization footprint in the shape of an ellipsis.

In order to illustrate the role of the constraints, we use a very simple mapping forward operator that retrieves the values in the mesh at some given positions. The constraints are therefore used as interpolation operators. Note that the mapping forward operator can also be used for defining prior knowledge

if combined with another forward operator in a classical joint inversion framework. In the initialization, the indices are stored and a mapping matrix is created that projects the model vector to the forward response. This matrix is also the Jacobian matrix for the inversion.

```
class PriorFOP(pg.Modelling):
    """Forward operator for grabbing values."""

    def __init__(self, mesh, pos, **kwargs):
        """Init with mesh and some positions that are converted into␣
↪ids."""
        super().__init__(**kwargs)
        self.setMesh(mesh)
        self.ind = [mesh.findCell(po).id() for po in pos]
        self.J = pg.SparseMapMatrix()
        self.J.resize(len(self.ind), mesh.cellCount())
        for i, ii in enumerate(self.ind):
            self.J.setVal(i, ii, 1.0)

        self.setJacobian(self.J)

    def response(self, model):
        """Return values at the indexed cells."""
        return model[self.ind]

    def createJacobian(self, model):
        """Do nothing (linear)."""
        pass
```

Inversion with geostatistical constraints

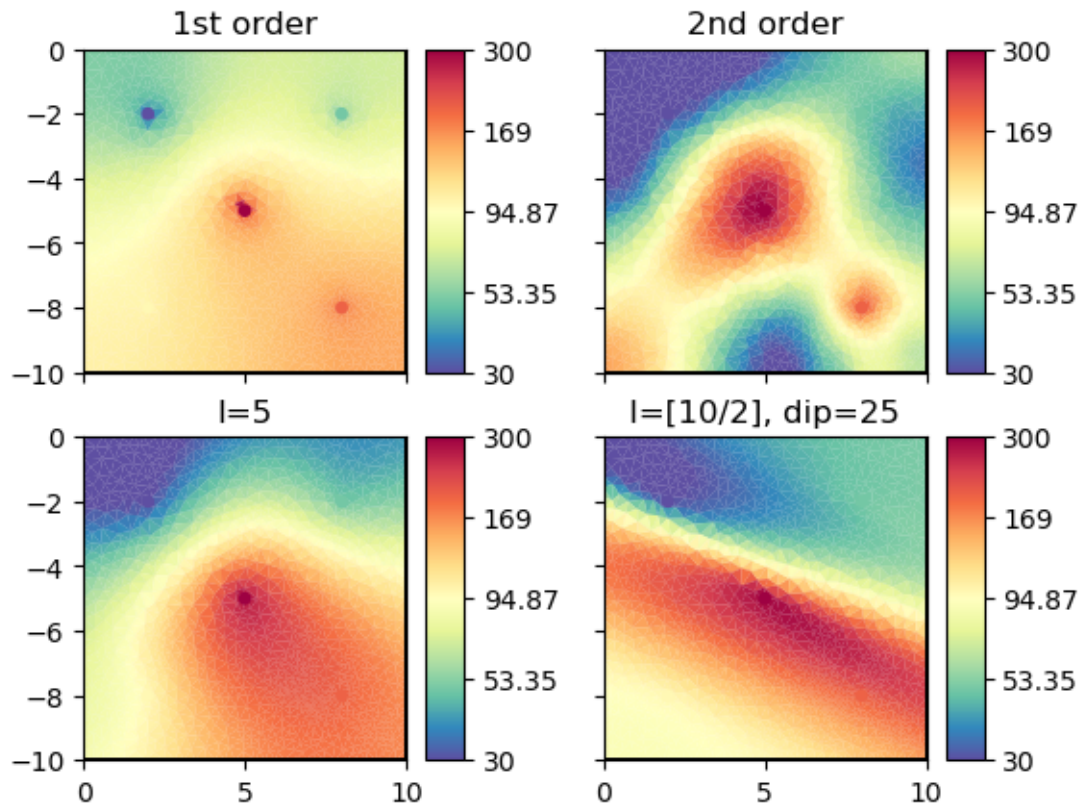
We choose some positions and initialize the forward operator

```
pos = [[2, -2], [8, -2], [5, -5], [2, -8], [8, -8]]
fop = PriorFOP(mesh, pos)
# For plotting the results, we create a figure and define some␣
↪plotting options
fig, ax = plt.subplots(nrows=2, ncols=2, sharex=True, sharey=True)
kw = dict(
    colorBar=True,
    cMin=30,
    cMax=300,
    orientation='vertical',
    cMap='Spectral_r',
    logScale=True)

# We want to use a homogenenous starting model
vals = [30, 50, 300, 100, 200]
# We assume a 5% relative accuracy of the values
error = pg.Vector(len(vals), 0.05)
# set up data and model transformation log-scaled
```

(continues on next page)

(continued from previous page)

```
tLog = pg.trans.TransLog()
inv = pg.Inversion(fop=fop)
inv.transData = tLog
inv.transModel = tLog
inv.lam = 40
inv.startModel = 30
# Initially, we use the first-order constraints (default)
res = inv.run(vals, error, cType=1, lam=30)
print(('Ctype=1: ' + '{:.1f} ' * 6).format(*fop(res), inv.chi2()))
pg.show(mesh, res, ax=ax[0, 0], **kw)
ax[0, 0].set_title("1st order")
np.testing.assert_array_less(inv.chi2(), 1.2)

# Next, we use the second order (curvature) constraint type
res = inv.run(vals, error, cType=2, lam=25)
print(('Ctype=2: ' + '{:.1f} ' * 6).format(*fop(res), inv.chi2()))
pg.show(mesh, res, ax=ax[0, 1], **kw)
ax[0, 1].set_title("2nd order")
np.testing.assert_array_less(inv.chi2(), 1.2)

# Now we set the geostatistic isotropic operator with 5m␣
↪correlation length
fop.setConstraints(C)
res = inv.run(vals, error, lam=15)
print(('Cg-5/5m: ' + '{:.1f} ' * 6).format(*fop(res), inv.chi2()))
pg.show(mesh, res, ax=ax[1, 0], **kw)
ax[1, 0].set_title("I=5")
np.testing.assert_array_less(inv.chi2(), 1.2)

# and finally we use the dipping constraint matrix
fop.setConstraints(Cdip)
res = inv.run(vals, error, lam=15)
print(('Cg-9/2m: ' + '{:.1f} ' * 6).format(*fop(res), inv.chi2()))
pg.show(mesh, res, ax=ax[1, 1], **kw)
ax[1, 1].set_title("I=[10/2], dip=25")
np.testing.assert_array_less(inv.chi2(), 1.2)

# plot the position of the priors
patches = [CirclePolygon(po, 0.2) for po in pos]
for ai in ax.flat:
    p = PatchCollection(patches, cmap=kw['cMap'])
    p.set_facecolor(None)
    p.set_array(np.array(vals))
    p.set_norm(LogNorm(kw['cMin'], kw['cMax']))
    ai.add_collection(p)
```



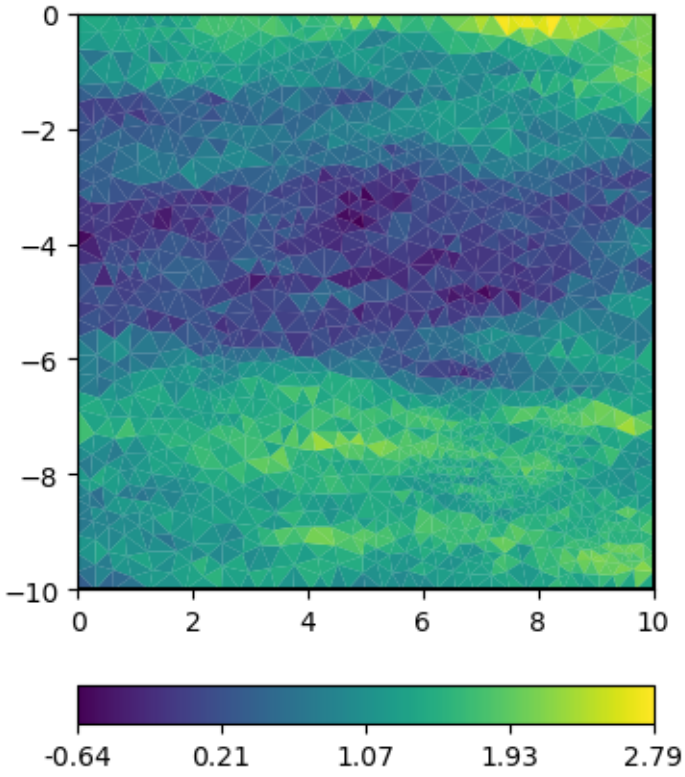

```
Ctype=1: 31.5 51.6 276.5 100.5 195.7 0.8

Ctype=2: 30.0 49.7 298.9 100.5 198.9 0.0

Cg-5/5m: 30.8 50.2 272.1 99.2 197.6 0.8

Cg-9/2m: 31.1 49.8 285.7 101.0 202.0 0.3
```

Note that all four regularization operators fit the data equivalently but the images (i.e. how the gaps between the data points are filled) are quite different. This is something we should have in mind using regularization.

Generating geostatistical media

For generating geostatistical media, one can use the function generateGeostatisticalModel. It computes a correlation matrix and multiplies it with a pseudo-random (randn) series. The arguments are the same as for the correlation or constraint matrices.

```
model = pg.utils.generateGeostatisticalModel(mesh, I=[20, 4])
ax, cb = pg.show(mesh, model)
```



Total running time of the script: (0 minutes 24.364 seconds)

7.4.4.7 Region-wise regularization

In this tutorial we like to demonstrate how to control the regularization of subsurface regions individually by using an ERT field case. The data is a 2d profile that was measured in 2005 on the bottom of a lake. The water body is of course influencing the fields and needs to be treated accordingly.

We first import pygimli and the modules for ERT and mesh building.

```
import pygimli as pg
from pygimli.physics import ert
import pygimli.meshtools as mt
```

Data and geometry

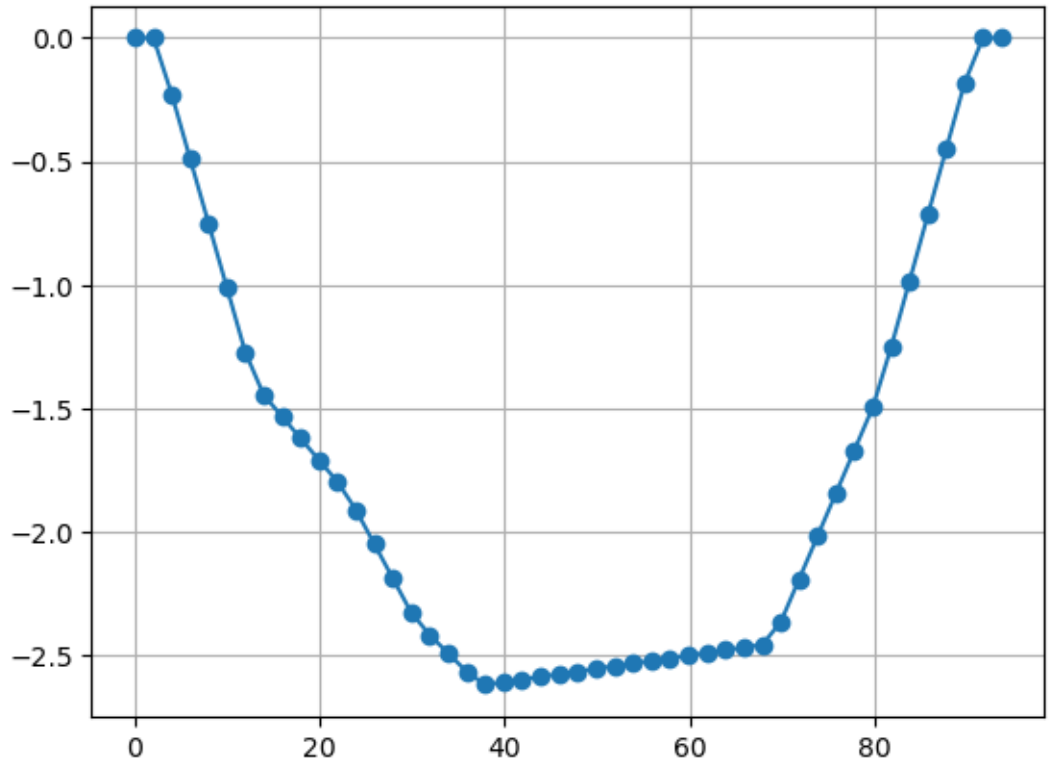
The data was measured across a shallow lake with the most electrodes being on the bottom of a lake. We used cables with 2m spaced takeouts.

```
data = pg.getExampleData("ert/lake.ohm")
print(data)
```

```
Data: Sensors: 48 data: 658, nonzero entries: ['a', 'b', 'err', 'i', 'm', 'n',
↪ 'r', 'u', 'valid']
```

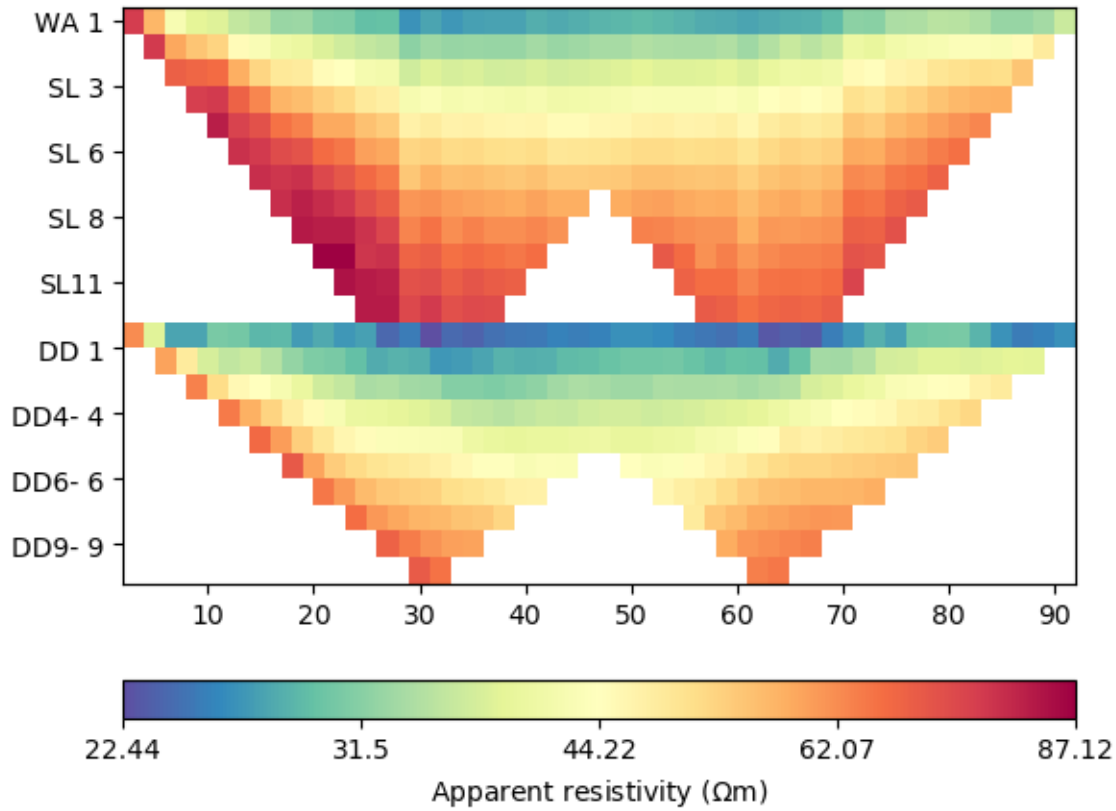
The data consists of 658 data with current and voltage using 48 electrodes. We first have a look at the electrode positions measured by a stick.

```
pg.plt.plot(pg.x(data), pg.z(data), "o-")
pg.plt.grid();
```



On both sides, two electrodes are on shore, but the others are on the bottom of a shallow lake with a maximum depth of 2.5m.

```
data["k"] = ert.geometricFactors(data)
data["rhoa"] = data["u"] / data["i"] * data["k"]
ert.show(data);
```




```
(<matplotlib.axes._subplots.AxesSubplot object at 0x7fe8f5866880>, <matplotlib.
↪colorbar.Colorbar object at 0x7fe8717809d0>)
```

We combined Wenner-Schlumberger (top) and Wenner-beta (bottom) data. The lowest resistivities correspond with the water resistivity of 22.5 $Omega$m.

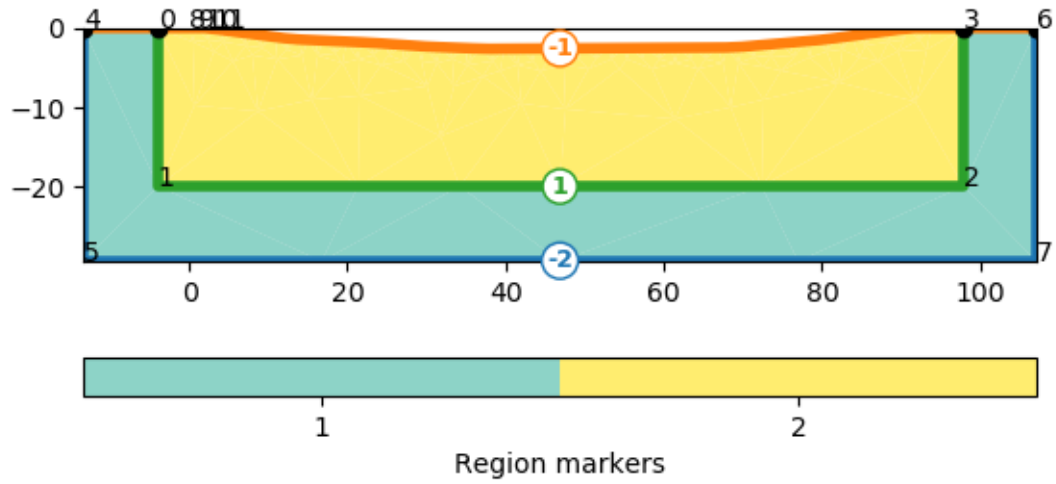
The contained errors are measured standard devitations and should not be used for inversion. Instead, we estimate new errors using 2% and 100microVolts.

```
data["err"] = ert.estimateError(data, relativeError=0.02, absoluteUError=1e-4)
print(max(data["err"]))
# pg.show(data, data["err"]*100, label="error (%)");
```

```
0.02584795321637427
```

Building a mesh with the water body

```
# We create a piece-wise linear complex (PLC) as for a case with␣
↪topography
plc = mt.createParaMeshPLC(data, paraDepth=20, boundary=0.1)
ax, _ = pg.show(plc, markers=True);
for i, n in enumerate(plc.nodes()[:12]):
    ax.text(n.x(), n.y(), str(i))
    print(i, n.x(), n.y())
```




```
0 -4.0 0.0
1 -4.0 -20.0
2 97.7452 -20.0
3 97.7452 0.0
4 -13.37452 0.0
5 -13.37452 -29.37452
6 107.11972 0.0
7 107.11972 -29.37452
8 0.0 0.0
9 1.0 0.0
10 2.0 0.0
11 2.993365 -0.11499999999999999
```

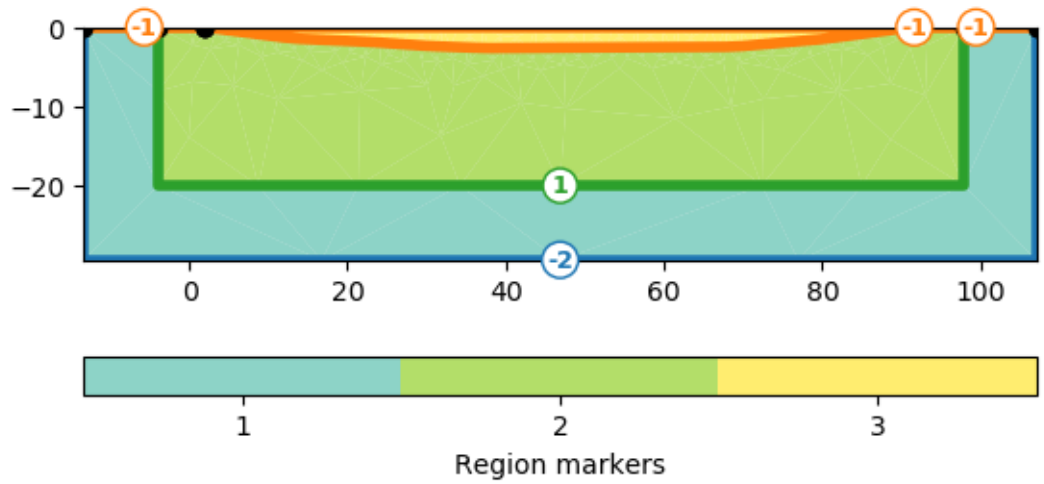
So node number 10 is the left one at the shore

```
for i in range(95, plc.nodeCount()):
    print(i, plc.node(i).x(), plc.node(i).y())
```

```
95 86.7804 -0.5833335
96 87.7715 -0.45
97 88.76255 -0.3166665
98 89.75359999999999 -0.183333
99 90.7494 -0.0916665
100 91.7452 0.0
101 92.7452 0.0
102 93.7452 0.0
```

and 100 the first on the other side. We connect nodes 10 and 100 by an edge

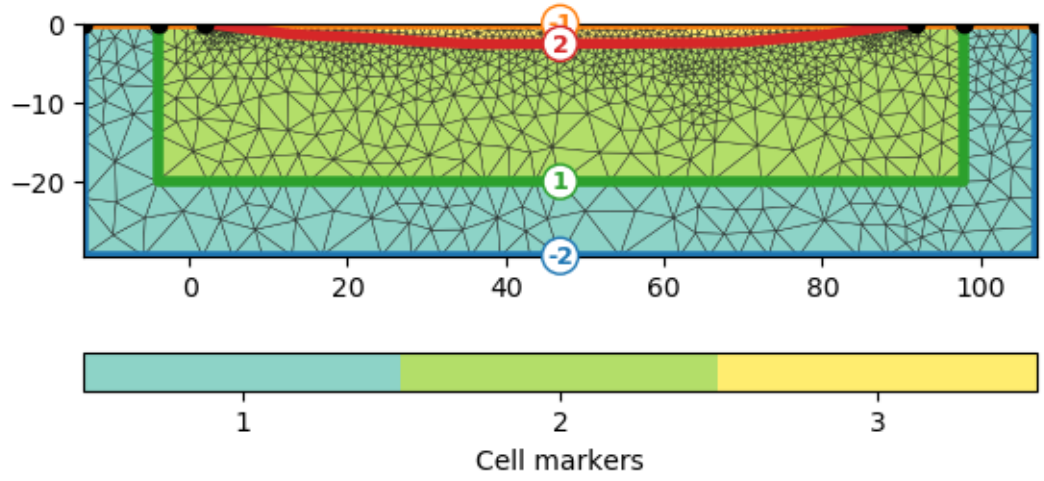
```
plc.createEdge(plc.node(10), plc.node(100), marker=-1)
plc.addRegionMarker([50, -0.1], marker=3)
pg.show(plc, markers=True);
```




```
(<matplotlib.axes._subplots.AxesSubplot object at 0x7fe87164b100>, None)
```

As the lake bottom is not a surface boundary (-1) anymore, but an inside boundary, we set its marker >0 by iterating through all boundaries.

```
mesh = mt.createMesh(plc, quality=34.4)
for b in mesh.boundaries():
    if b.marker() == -1 and not b.outside():
        b.setMarker(2)

print(mesh)
pg.show(mesh, markers=True, showMesh=True);
```




```
Mesh: Nodes: 1136 Cells: 2103 Boundaries: 3238

(<matplotlib.axes._subplots.AxesSubplot object at 0x7fe899f1fbe0>, <matplotlib.
↪colorbar.Colorbar object at 0x7fe8ea3460d0>)
```

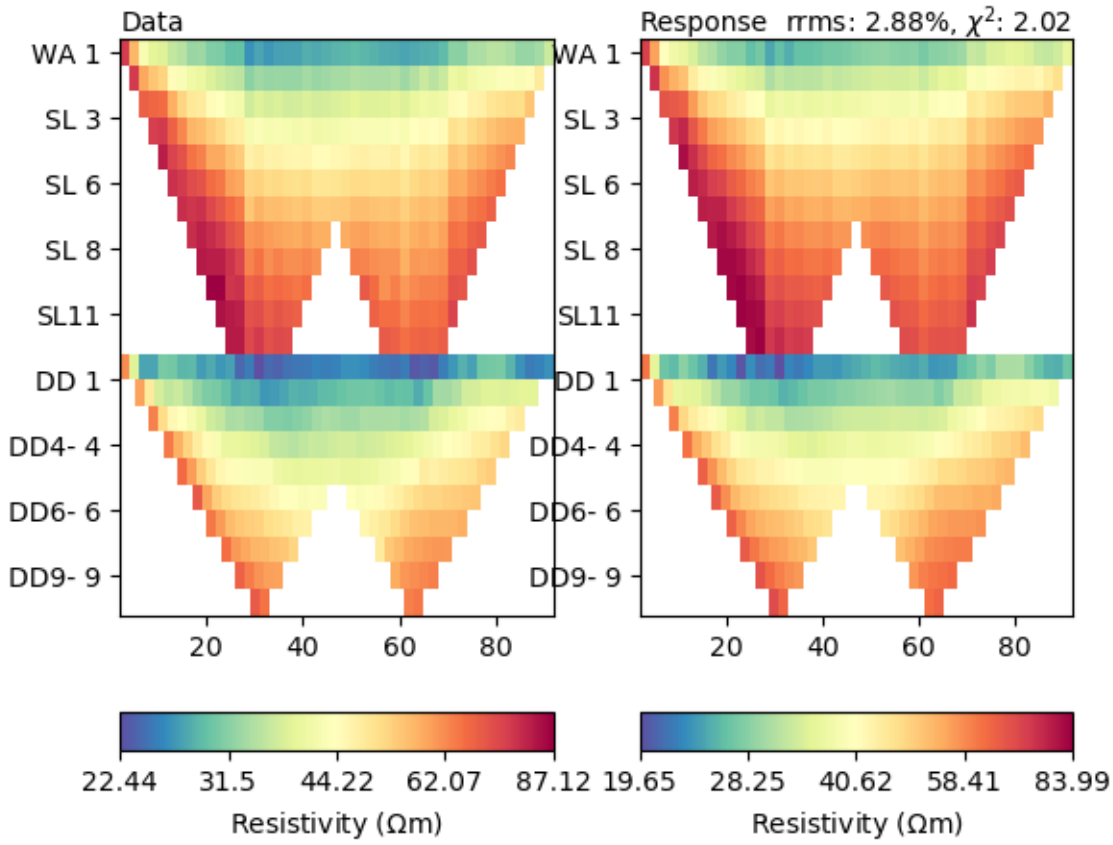
Inversion with the ERT manager

```
mgr = ert.ERTManager(data, verbose=True)
mgr.setMesh(mesh)  # use this mesh for all subsequent runs
mgr.invert()
# mgr.invert(mesh=mesh) would only temporally use the mesh
```

```
fop: <pygimli.physics.ert.ertModelling.ERTModelling object at 0x7fe8ab0057c0>
Data transformation: <pygimli.core._pygimli_.RTransLogLU object at␣
↪0x7fe899e53d60>
Model transformation: <pygimli.core._pygimli_.RTransLog object at 0x7fe8ab0058b0>
min/max (data): 22.44/87.12
min/max (error): 2%/2.58%
min/max (start model): 47.2/47.2
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
inv.iter 2 ... chi² = 19.5 (dPhi = 66.94%) lam: 20
--------------------------------------------------------------------------------
inv.iter 3 ... chi² = 7.89 (dPhi = 58.65%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 4 ... chi² = 5.38 (dPhi = 31.09%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 5 ... chi² = 3.63 (dPhi = 31.34%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 6 ... chi² = 2.04 (dPhi = 41.11%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 7 ... chi² = 2.02 (dPhi = 0.81%) lam: 20.0
################################################################################
#                  Abort criteria reached: dPhi = 0.81 (< 2.0%)                #
################################################################################

1838 [208.14321162887464,...,96.31630761372877]
```

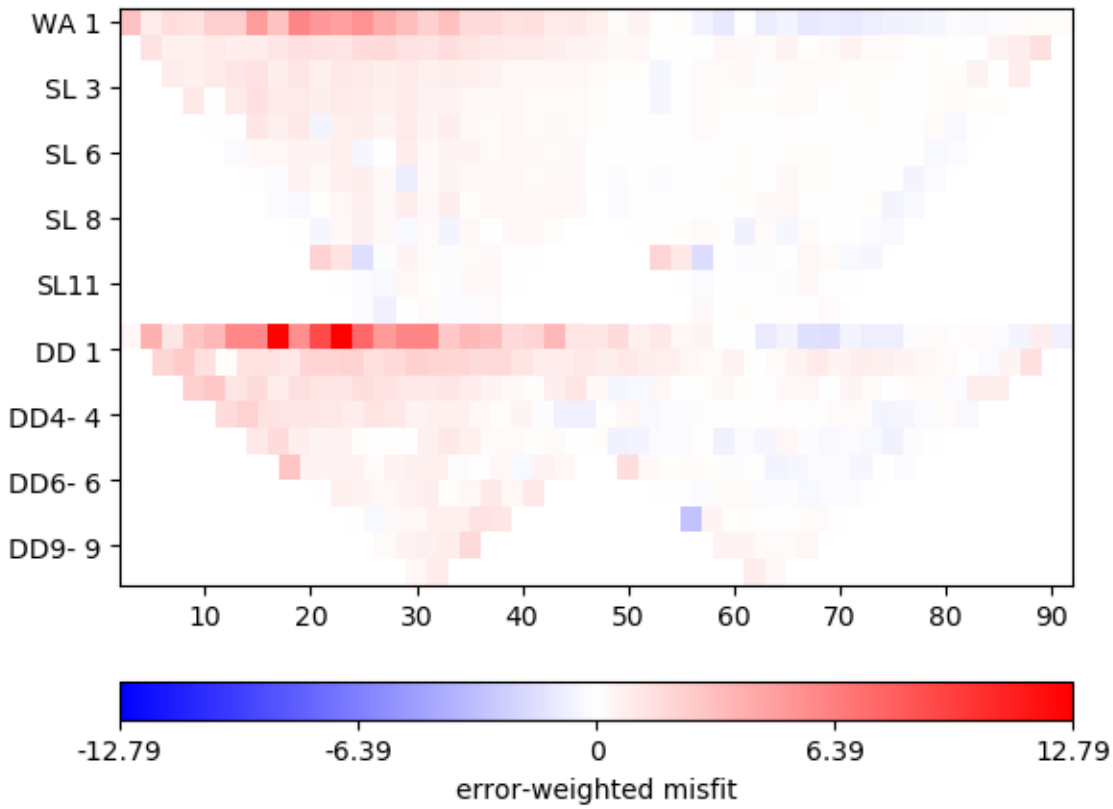
The fit is obviously not perfect. So we have a look at data and model response.

```
ax = mgr.showFit()
```



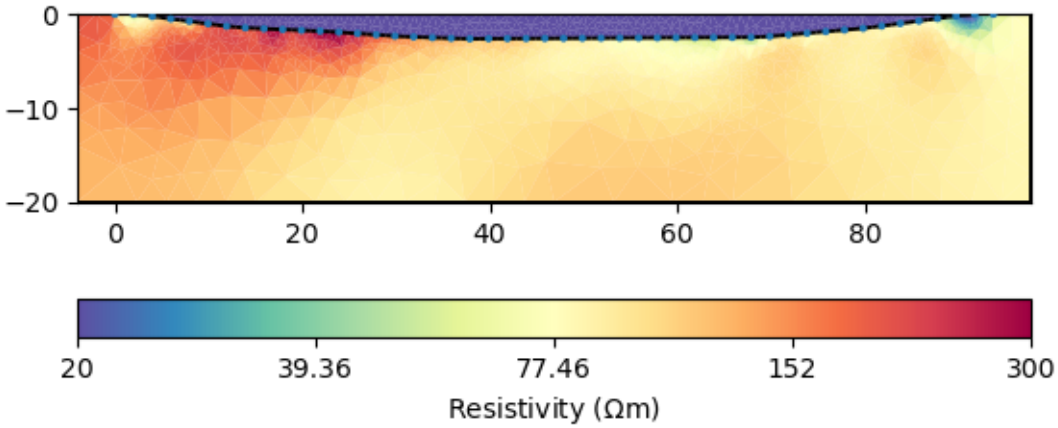
Both look very similar, but let us look at the misfit function in detail.

```
mgr.showMisfit(errorWeighted=True)
```



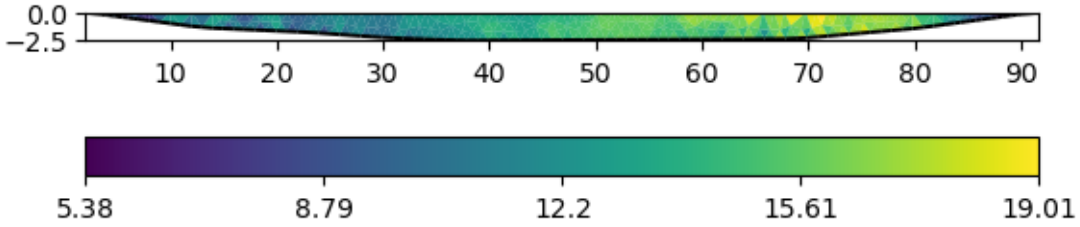
There is still systematics in the misfit. Ideally it should be a random distribution of Gaussian noise.

```
cov = pg.Vector(mgr.model.size(), 1.0)
kw = dict(cMin=20, cMax=300, logScale=True, cMap="Spectral_r", coverage=cov)
ax, cb = mgr.showResult(**kw)
```



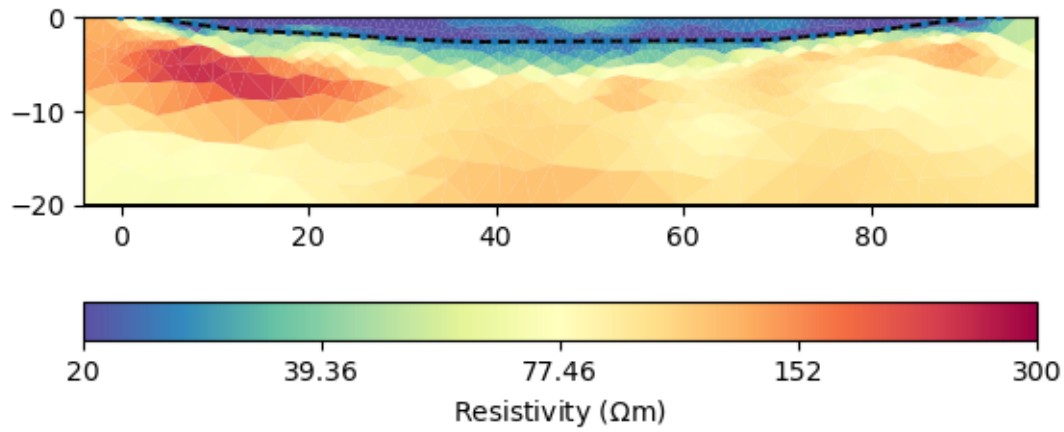
Apparently, the two regions are already decoupled from each other which makes sense. Let us look in detail at the water cells by extracting the water body.

```
water = mesh.createSubMesh(mesh.cells(mesh.cellMarkers() == 3))
resWater = mgr.model[len(mgr.model)-water.cellCount():]
ax, cb = pg.show(water, resWater)
```



Apparently, all values are below the expected 22.5$Omega$m and some are implausibly low. Therefore we should try to limit them. Moreover, the subsurface structures do not look very “layered”, which is why we make the smoothness anisotropic.

```
mgr.inv.setRegularization(zWeight=0.1)
mgr.invert()
# mgr.invert(zWeight=0.1)  # only temporarily
ax, cb = mgr.showResult(**kw)
```

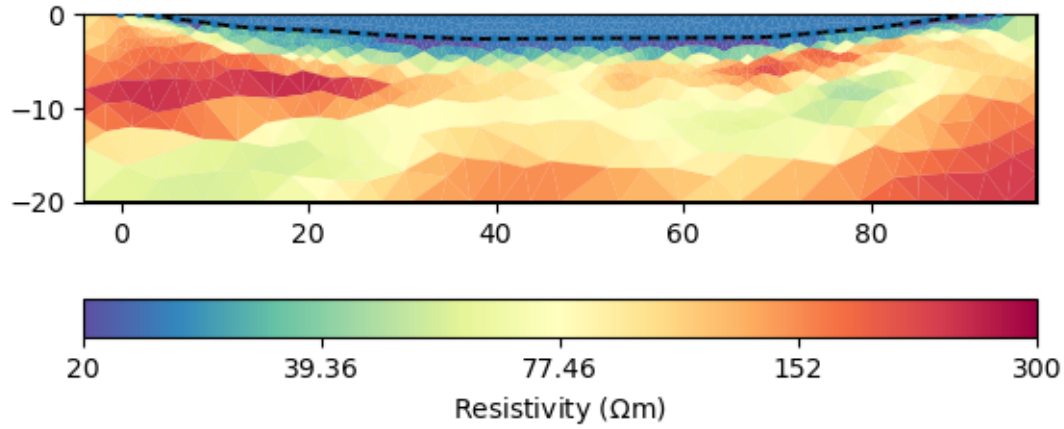



```
fop: <pygimli.physics.ert.ertModelling.ERTModelling object at 0x7fe8ab0057c0>
Data transformation: <pygimli.core._pygimli_.RTransLogLU object at␣
↪0x7fe899e53d60>
Model transformation (cumulative):
         0 <pygimli.core._pygimli_.RTransLogLU object at 0x7fe8a90c70a0>
         1 <pygimli.core._pygimli_.RTransLogLU object at 0x7fe8a90c70a0>
min/max (data): 22.44/87.12
min/max (error): 2%/2.58%
min/max (start model): 47.2/47.2
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
inv.iter 2 ... chi² = 12.57 (dPhi = 74.44%) lam: 20
-------------------------------------------------------------------------------
inv.iter 3 ... chi² = 3.19 (dPhi = 74.26%) lam: 20.0
-------------------------------------------------------------------------------
inv.iter 4 ... chi² = 0.97 (dPhi = 68.33%) lam: 20.0


#################################################################################
#                    Abort criterion reached: chi² <= 1 (0.97)                  #
#################################################################################
```

Region-specific regularization

We first want to limit the resistivity of the water between some plausible bounds around our measurements.

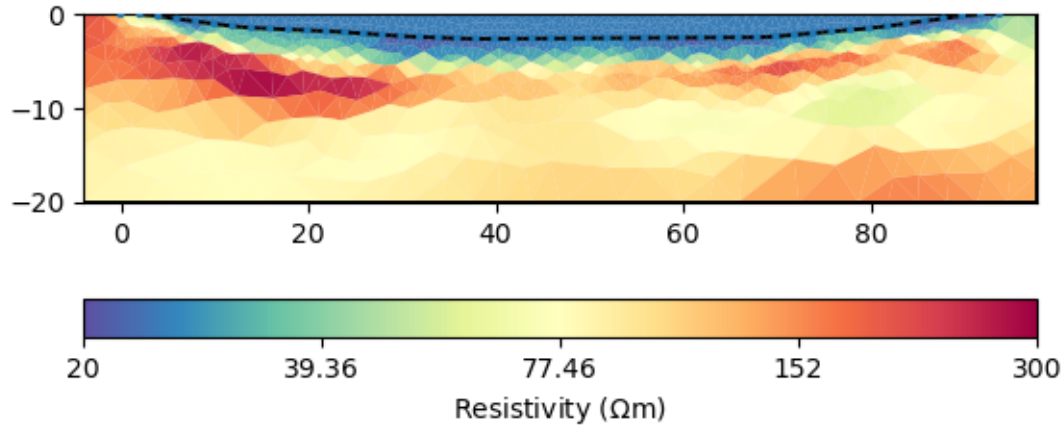
```
mgr.inv.setRegularization(3, limits=[20, 25], trans="log")
mgr.invert()
ax, cb = mgr.showResult(**kw)
```




```
fop: <pygimli.physics.ert.ertModelling.ERTModelling object at 0x7fe8ab0057c0>
Data transformation: <pygimli.core._pygimli_.RTransLogLU object at␣
↪0x7fe899e53d60>
Model transformation (cumulative):
         0 <pygimli.core._pygimli_.RTransLogLU object at 0x7fe8997aee80>
         1 <pygimli.core._pygimli_.RTransLogLU object at 0x7fe8997ae4c0>
min/max (data): 22.44/87.12
min/max (error): 2%/2.58%
min/max (start model): 47.2/47.2
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
inv.iter 2 ... chi² = 7.11 (dPhi = 88.78%) lam: 20
--------------------------------------------------------------------------------
inv.iter 3 ... chi² = 0.99 (dPhi = 84.77%) lam: 20.0


################################################################################
#                    Abort criterion reached: chi² <= 1 (0.99)                 #
################################################################################
```

As a result of the log-log transform, we have a homogeneous body but below the lake bottom values below 20, maybe due to clay content or maybe as compensation of limiting the water resistivity too strong. We could limit the subsurface, too.

```
mgr.inv.setRegularization(2, limits=[20, 2000], trans="log")
mgr.invert()
ax, cb = mgr.showResult(**kw)
```



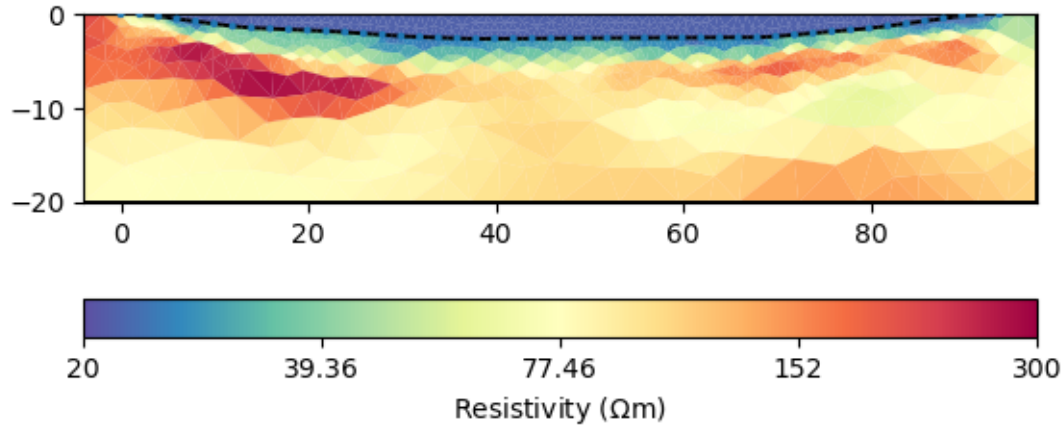

```
fop: <pygimli.physics.ert.ertModelling.ERTModelling object at 0x7fe8ab0057c0>
Data transformation: <pygimli.core._pygimli_.RTransLogLU object at␣
↪0x7fe899e53d60>
Model transformation (cumulative):
         0 <pygimli.core._pygimli_.RTransLogLU object at 0x7fe8ab327f40>
         1 <pygimli.core._pygimli_.RTransLogLU object at 0x7fe8fa7b0dc0>
min/max (data): 22.44/87.12
min/max (error): 2%/2.58%
min/max (start model): 47.2/47.2
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
inv.iter 2 ... chi² = 8.3 (dPhi = 67.62%) lam: 20
--------------------------------------------------------------------------------
inv.iter 3 ... chi² = 1.5 (dPhi = 78.96%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 4 ... chi² = 0.52 (dPhi = 54.1%) lam: 20.0


################################################################################
#                    Abort criterion reached: chi² <= 1 (0.52)                 #
################################################################################
```

Apparently, this makes it harder to fit the data accurately. So maybe an increased clay content can be responsible for resistivity below 20$Omega$m in the mud.

Model reduction

Another option is to treat the water body as a homogeneous body with only one unknown in the inversion.

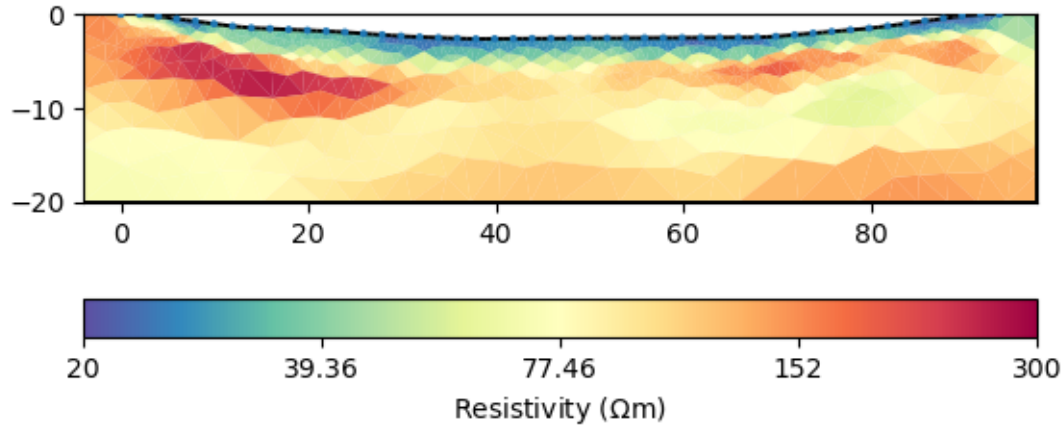
```
mgr.inv.setRegularization(limits=[0, 0], trans="log")
mgr.inv.setRegularization(3, single=True)
mgr.invert()
ax, cb = mgr.showResult(**kw)

print(mgr.inv.model)
print(min(mgr.model))
```




```
fop: <pygimli.physics.ert.ertModelling.ERTModelling object at 0x7fe8ab0057c0>
Data transformation: <pygimli.core._pygimli_.RTransLogLU object at␣
↪0x7fe899e53d60>
Model transformation (cumulative):
         0 <pygimli.core._pygimli_.RTransLogLU object at 0x7fe899a7cdc0>
         1 <pygimli.core._pygimli_.RTransLogLU object at 0x7fe8ea2f3e80>
min/max (data): 22.44/87.12
min/max (error): 2%/2.58%
min/max (start model): 47.2/47.2
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
inv.iter 2 ... chi^2 = 9.95 (dPhi = 62.96%) lam: 20
--------------------------------------------------------------------------------
inv.iter 3 ... chi^2 = 1.99 (dPhi = 77.96%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 4 ... chi^2 = 0.54 (dPhi = 65.24%) lam: 20.0


################################################################################
#                   Abort criterion reached: chi^2 <= 1 (0.54)                 #
################################################################################
1406 [32.320138266282186,...,21.80517248258542]
21.80517248258542
```

The last value represents the value for the lake, close to our measurement. This value can, however, also be set beforehand.

```
mgr.inv.setRegularization(3, fix=22.5)
mgr.invert()
ax, cb = mgr.showResult(**kw)
```

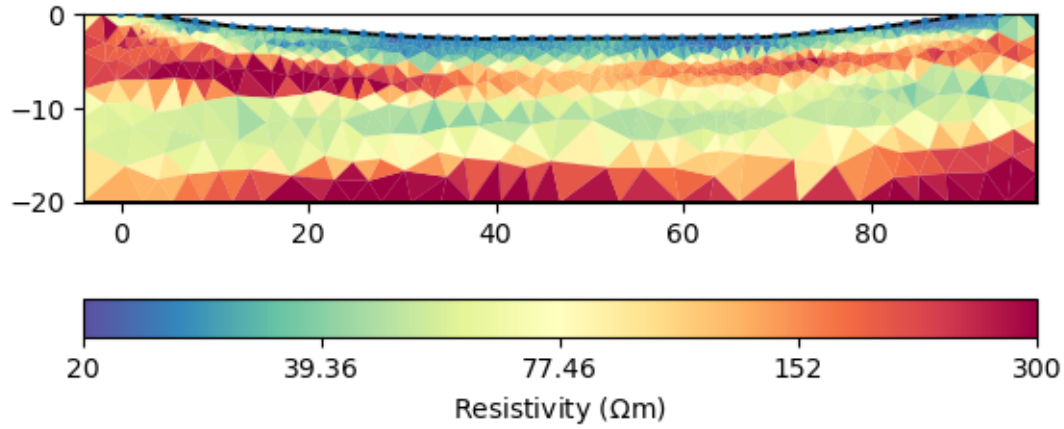



```
fop: <pygimli.physics.ert.ertModelling.ERTModelling object at 0x7fe8ab0057c0>
Data transformation: <pygimli.core._pygimli_.RTransLogLU object at␣
↪0x7fe899e53d60>
Model transformation (cumulative):
         0 <pygimli.core._pygimli_.RTransLogLU object at 0x7fe8fa7ef820>
min/max (data): 22.44/87.12
min/max (error): 2%/2.58%
min/max (start model): 47.2/47.2
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
inv.iter 2 ... chi^2 = 6.15 (dPhi = 67.51%) lam: 20
--------------------------------------------------------------------------------
inv.iter 3 ... chi^2 = 1.24 (dPhi = 77.13%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 4 ... chi^2 = 0.51 (dPhi = 47.97%) lam: 20.0


################################################################################
#                    Abort criterion reached: chi^2 <= 1 (0.51)                #
################################################################################
```

We see that the lake does not appear anymore as it is not a part of the inversion mesh `mgr.paraDomain` anymore.

Instead of the standard smoothness we use geostatistical regularization.

```
mgr.inv.setRegularization(2, correlationLengths=[30, 2])
mgr.invert()
ax, cb = mgr.showResult(**kw)
```

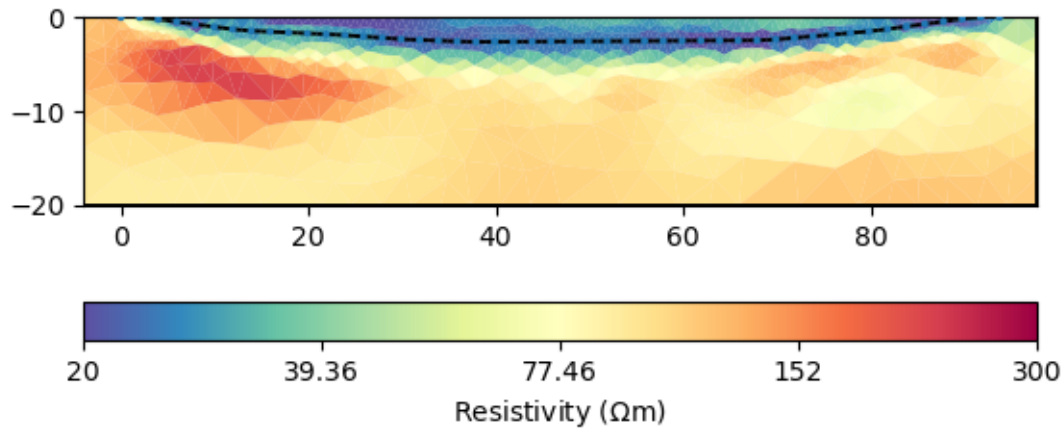



```
fop: <pygimli.physics.ert.ertModelling.ERTModelling object at 0x7fe8ab0057c0>
Data transformation: <pygimli.core._pygimli_.RTransLogLU object at␣
↪0x7fe899e53d60>
Model transformation (cumulative):
         0 <pygimli.core._pygimli_.RTransLogLU object at 0x7fe8fa7b0dc0>
min/max (data): 22.44/87.12
min/max (error): 2%/2.58%
min/max (start model): 47.2/47.2
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
inv.iter 2 ... chi² = 3.06 (dPhi = 76.35%) lam: 20
--------------------------------------------------------------------------------
inv.iter 3 ... chi² = 0.61 (dPhi = 76.06%) lam: 20.0


################################################################################
#                    Abort criterion reached: chi² <= 1 (0.61)                 #
################################################################################
```

Region coupling

In case (does not make sense here) the two regions should be coupled to each other, you can set so-called inter-region constraints.

```
mgr = ert.ERTManager(data, verbose=True)
mgr.setMesh(mesh)
print(mgr.fop.regionManager().regionCount())
mgr.inv.setRegularization(cType=1, zWeight=0.2)
mgr.fop.setInterRegionCoupling(2, 3, 1.0)  # normal coupling
mgr.invert()
ax, cb = mgr.showResult(**kw)
```




```
3
fop: <pygimli.physics.ert.ertModelling.ERTModelling object at 0x7fe8999df3b0>
Data transformation: <pygimli.core._pygimli_.RTransLogLU object at␣
↪0x7fe8999df630>
Model transformation (cumulative):
         0 <pygimli.core._pygimli_.RTransLogLU object at 0x7fe8f54d81c0>
         1 <pygimli.core._pygimli_.RTransLogLU object at 0x7fe8aae826a0>
min/max (data): 22.44/87.12
min/max (error): 2%/2.58%
min/max (start model): 47.2/47.2
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
inv.iter 2 ... chi² = 10.43 (dPhi = 67.94%) lam: 20
--------------------------------------------------------------------------------
inv.iter 3 ... chi² = 1.95 (dPhi = 80.02%) lam: 20.0
--------------------------------------------------------------------------------
inv.iter 4 ... chi² = 0.55 (dPhi = 65.97%) lam: 20.0


################################################################################
#                  Abort criterion reached: chi² <= 1 (0.55)                   #
################################################################################
```

The general image is of course similar, but the structures are mirrored around the lake bottom. Moreover the resistivity in the lake is far too high. Note that all of the obtained images are equivalent with respect to data and errors.

Take-away messages

- always have a look at the data fit and get hands on data errors
- a lot of different models are able to fit the data, particularly in the full space
- regions can be very specifically controlled
- constrain or fix whenever possibly (and reliable)
- sometimes geostatistic constraints outperform classical smoothness but sometimes not

- play with regularization and keep looking at data fit

Total running time of the script: (1 minutes 52.318 seconds)

CHAPTER

EIGHT

PYGIMLI API REFERENCE

Note: In the following, all Python modules, functions and classes are documented. For a reference of the C++ core library of the code, please visit: http://pygimli.org/gimliapi/.

Module overview

frameworks (page 279)	Unified and method independent inversion frameworks.
math (page 301)	Math functions and matrices.
meshtools (page 309)	Mesh generation and modification.
physics (page 361)	Module containing submodules for various geophysical methods.
solver (page 430)	General physics independent solver interface.
testing (page 455)	Testing utilities
utils (page 456)	Useful utility functions.
viewer (page 475)	Interface for 2D and 3D visualizations.

8.1 pygimli.frameworks

Unified and method independent inversion frameworks.

8.1.1 Overview

Functions

fit (page 281)(funct, data[, err])	Generic function fitter.
harmfit (page 281)(y[, x, error, nc, resample, lam, ...])	GIMLi-based curve-fit by harmonic functions.
harmfitNative (page 282)(y[, x, nc, xc, err])	Python-based curve-fit by harmonic functions.

Classes

Block1DInversion (page 282)([fop])	Inversion of layered models (including layer thickness)
Block1DModelling (page 283)([nPara, nLayers])	General forward operator for 1D layered models.
HarmFunctor (page 283)(A, coeff, xmin, xSpan)	Functor for harmonic functions plus offset and drift.
Inversion (page 283)([fop, inv])	Basic inversion framework.
JointModelling (page 287)(fopList)	Cumulative (joint) forward operator.
JointPetroInversionManager (page 287)(petros, mgrs)	Joint inversion targeting at the same parameter through petrophysics.
LCInversion (page 288)([fop])	Quasi-2D Laterally constrained inversion (LCI) framework.
LCModelling (page 289)(fop, **kwargs)	2D Laterally constrained (LC) modelling.
LinearModelling (page 289)(A)	Modelling class for linearized problems with a given matrix.
MarquardtInversion (page 290)([fop])	Marquardt scheme, i.e. local damping with decreasing strength.
MeshMethodManager (page 290)(**kwargs)	
MeshModelling (page 291)(**kwargs)	Modelling class with a mesh discretization.
MethodManager (page 292)([fop, fw, data])	General manager to maintenance a measurement method.
MethodManager1d (page 296)([fop])	Method Manager base class for managers on a 1d discretization.
Modelling (page 296)(**kwargs)	Abstract Forward Operator.
MultiFrameModelling (page 298)(modellingOperator[, scalef])	Full frame (multiple fop parallel) forward modelling.
ParameterInversionManager (page 299)([funct, fop])	Framework to invert unconstrained parameters.
ParameterModelling (page 299)([funct])	Model with symbolic parameter names instead of numbers
PetroInversionManager (page 300)(petro[, mgr])	Class for petrophysical inversion (s.
PetroModelling (page 300)(fop, petro, **kwargs)	Combine petrophysical relation with the modelling class f(p).
PriorModelling (page 301)([mesh, pos])	Forward operator for grabbing values out of a mesh (prior data).

8.1.2 Functions

pygimli.frameworks.**fit**(*funct*, *data*, *err=None*, ***kwargs*)

Generic function fitter.

Fit data to a given function.

Parameters

- **funct** (*callable*) – Function with the first argmument as data space, e.g., x, t, f, Nr. .. Any following arguments are the parameters to be fit. Except if a verbose flag if used.
- **data** (*iterable (float)*) – Data values
- **err** (*iterable (float) [None]*) – Data error values in %/100. Default is 1% if None are given.
- ***dataSpace*** (*iterable*) – Keyword argument of the data space of len(data). The name need to fit the first argument of funct.

Returns

- **model** (*array*) – Fitted model parameter.
- **response** (*array*) – Model response.

Example

```
>>> import pygimli as pg
>>>
>>> func = lambda t, a, b: a*np.exp(b*t)
>>> t = np.linspace(1, 2, 20)
>>> data = func(t, 1.1, 2.2)
>>> model, response = pg.frameworks.fit(func, data, t=t)
>>> print(pg.core.round(model, 1e-5))
2 [1.1, 2.2]
>>> _ = pg.plt.plot(t, data, 'o', label='data')
>>> _ = pg.plt.plot(t, response, label='response')
>>> _ = pg.plt.legend()
```

pygimli.frameworks.**harmfit**(*y*, *x=None*, *error=None*, *nc=42*, *resample=None*, *lam=0.1*, *window=None*, *verbose=False*, *dosave=False*, *lineSearch=True*, *robust=False*, *maxiter=20*)

GIMLi-based curve-fit by harmonic functions.

Parameters

- **y** (*1d-array - values to be fitted*) –
- **x** (*1d-array(len(y)) - data abscissa data. default: [0 .. len(y))*) –
- **error** (*1d-array(len(y)) error of y. default (absolute error = 0.01)*) –
- **nc** (*int - Number of harmonic coefficients*) –

- **resample** (*1d-array - resample y to x using fitting coeffients*) –
- **window** (*int - just fit data inside window bounds*) –

Returns

- **response** (*1d-array(len(resample) or len(x)) - smoothed values*)
- **coefficients** (*1d-array - fitting coefficients*)

pygimli.frameworks.**harmfitNative**(*y*, *x=None*, *nc=None*, *xc=None*, *err=None*)

Python-based curve-fit by harmonic functions.

Parameters

- **y** (*iterable*) – values of a curve to be fitted
- **x** (*iterable*) – abscissa, if none [0..len(y))
- **nc** (*int*) – number of coefficients
- **err** (*iterable*) – absolute data error
- **xc** (*iterable*) – abscissa to predict y on (otherwise equal to x)

8.1.3 Classes

class pygimli.frameworks.**Block1DInversion**(*fop=None*, ***kwargs*)

Bases: *MarquardtInversion* (page 290)

Inversion of layered models (including layer thickness)

Variables
nLayers (*int*) –

__init__(*fop=None*, ***kwargs*)

fixLayers(*fixLayers*)

Fix layer thicknesses.

Parameters
fixLayers (*bool | [float]*) – Fix all layers to the last value or set the fix layer thickness for all layers

run(*dataVals*, *errorVals*, *nLayers=None*, *fixLayers=None*, *layerLimits=None*, *paraLimits=None*, ***kwargs*)

Parameters

- **nLayers** (*int [4]*) – Number of layers.
- **fixLayers** (*bool | [thicknesses]*) – See: pygimli.modelling.Block1DInversion.fixLayers For fixLayers=None, preset or defaults are uses.
- **layerLimits** (*[min, max]*) – Limits the thickness off all layers. For layerLimits=None, preset or defaults are uses.

- **paraLimits** (*[min, max] | [[min, max],...]*) – Limits the range of the model parameter. If you have multiple parameters you can set them with a list of limits.
- ****kwargs** – Forwarded to the parent class. See: pygimli.modelling.MarquardtInversion

setForwardOperator(*fop*)

setLayerLimits(*limits*)

Set min and max layer thickness.

Parameters

limits (*False | [min, max]*) –

setParaLimits(*limits*)

Set the limits for each parameter region.

class pygimli.frameworks.**Block1DModelling**(*nPara=1*, *nLayers=4*, ***kwargs*)

Bases: *Modelling* (page 296)

General forward operator for 1D layered models.

Model space: [thickness_i, parameter_jk], with i = 0 - nLayers-1, j = (0 .. nLayers), k=(0 .. nPara)

__init__(*nPara=1*, *nLayers=4*, ***kwargs*)

Constructor

Parameters

- **nLayers** (*int [4]*) – Number of layers.
- **nPara** (*int [1]*) – Number of parameters per layer (e.g. nPara=2 for resistivity and phase)

drawData(*ax*, *data*, *err=None*, *label=None*, ***kwargs*)

Default data view.

Modelling creates the data and should know best how to draw them.

Probably ugly and you should overwrite it in your derived forward operator.

drawModel(*ax*, *model*, ***kwargs*)

initModelSpace(*nLayers*)

Set number of layers for the 1D block model

property nLayers

property nPara

class pygimli.frameworks.**HarmFunctor**(*A*, *coeff*, *xmin*, *xSpan*)

Bases: object

Functor for harmonic functions plus offset and drift.

__init__(*A*, *coeff*, *xmin*, *xSpan*)

Initialize.

class pygimli.frameworks.**Inversion**(*fop=None*, *inv=None*, ***kwargs*)

Bases: object

Basic inversion framework.

Changes to prior Versions (remove me)

- holds the starting model itself, forward operator only provides a

method to create the starting model fop.createStartModel(dataValues)

Variables

- **verbose** (*bool*) – Give verbose output
- **debug** (*bool*) – Give debug output
- **startModel** (*float | array | None*) – Current starting model that can be set in the init or as property. If not set explicitly, it will be estimated from the forward operator methods. This property will be recalulated for every run call if not set explicitly with self.startModel = float|array, or None to reforce autogeneration. Note that the run call accepts a temporary startModel (for the current calculation only).
- **model** (*array*) – Holds the last active model
- **maxIter** (*int [20]*) – Maximal interation number.
- **stopAtChi1** (*bool [True]*) – Stop iteration when chi^2 is one. If set to False the iteration stops after maxIter or convergence reached (self.inv.deltaPhiAbortPercent())

__init__(*fop=None*, *inv=None*, ***kwargs*)

absrms()

Absolute root-mean-square misfit of the last run.

property blockyModel

chi2(*response=None*)

Chi-squared misfit (mean of squared error-weighted misfit).

convertStartModel(*model*)

Convert scalar or array into startmodel with valid range or
self.fop.parameterCount, if possible.

Variables
model (*float | int | array | None*) –

property dataErrs

property dataTrans

property dataVals

property debug

echoStatus()

Echo inversion status (model, response, rms, chi^2, phi).

property errorVals

property fop

property inv

property lam

property maxIter

property minDPhi

property model

The last active model.

property modelTrans

phi(*model=None*, *response=None*)

Total objective function (phiD + lambda * phiM)

phiData(*response=None*)

Data objective function (sum of suqred error-weighted misfit).

phiModel(*model=None*)

Model objective function (norm of regularization term).

relrms()

Relative root-mean-square misfit of the last run.

reset()

Reset function currently called at beginning of every inversion.

property response

property robustData

run(*dataVals*, *errorVals*, ***kwargs*)

Run inversion.

The inversion will always start from the starting model taken from the forward operator. If you want to run the inversion from a specified prior model, e.g., from a other run, set this model as starting model to the FOP (fop.setStartModel). Any self.inv.setModel() settings will be overwritten.

Parameters

- **dataVals** (*iterable*) – Data values
- **errorVals** (*iterable*) – Relative error values. dv / v

Keyword Arguments

- **maxIter** (*int*) – Overwrite class settings for maximal iterations number.
- **dPhi** (*float [1]*) – Overwrite class settings for delta data phi aborting criteria. Default is 1%

- **cType** (*int [1]*) – Temporary global contraint type for all regions.
- **startModel** (*array*) – Temporary starting model for the current inversion run.
- **lam** (*float*) – Temporary regularization parameter lambda.
- **lambdaFactor** (*float [1]*) – factor to change lam with every iteration
- **robustData** (*bool*) – robust (L1 norm mimicking) data reweighting
- **blockyModel** (*bool*) – robust (L1 norm mimicking) model roughness reweighting
- **isReference** (*bool [False]*) – starting model is also a reference to constrain against
- **showProgress** (*bool*) – show progress in form of updating models
- **verbose** (*bool*) – verbose output on the console
- **debug** (*bool*) – even verboser console and file output

setData(*data*)

Set data.

setDeltaChiStop(***kwargs*)

setDeltaPhiStop(*it*)

Define minimum relative decrease in objective function to stop.

setForwardOperator(*fop*)

setPostStep(*p*)

Set a function to be called after each iteration.

setPreStep(*p*)

Set a function to be called before each iteration.

setRegularization(**args*, ***kwargs*)

Set regularization properties for the inverse problem.

This can be for specific regions (args) or all regions (no args).

Parameters

- **regionNr** (*int, [ints], '*'*) – Region number, list of numbers, or wildcard "*" for all.
- **startModel** (*float*) – starting model value
- **limits** (*[float, float]*) – lower and upper limit for value using a barrier transform
- **trans** (*str*) – transformation for model barrier: "log", "cot", "lin"
- **cType** (*int*) – constraint (regularization) type
- **zWeight** (*float*) – relative weight for vertical boundaries
- **background** (*bool*) – exclude region from inversion completely (prolongation)

- **fix** (*float*) – exclude region from inversion completely (fix to value)
- **single** (*bool*) – reduce region to one unknown
- **correlationLengths** (*[floats]*) – correlation lengths for geostatistical inversion (x', y', z')
- **dip** (*float [0]*) – angle between x and x' (first correlation length)
- **strike** (*float [0]*) – angle between y and y' (second correlation length)

showProgress(*style='all'*)

Show the inversion progress after every iteration.

Can show models if *drawModel* method exists. The default fallback is plotting the χ^2 fit as a function of iterations. Called if *showProgress=True* is set for the inversion run.

property startModel

Gives the current default starting model.

Returns the current default starting model or calls *fop.createStartmodel()* if none is defined.

property stopAtChi1

property verbose

class pygimli.frameworks.**JointModelling**(*fopList*)

Bases: *MeshModelling* (page 291)

Cumulative (joint) forward operator.

__init__(*fopList*)

Initialize with lists of forward operators

createJacobian(*model*)

Fill the individual Jacobian matrices.

createStartModel(*data*)

Use inverse transformation to get m(p) for the starting model.

response(*model*)

Concatenate responses for all fops.

setData(*data*)

Distribute list of data to the forward operators.

setMesh(*mesh*, ***kwargs*)

Set the parameter mesh to all fops.

class pygimli.frameworks.**JointPetroInversionManager**(*petros*, *mgrs*)

Bases: *MeshMethodManager* (page 290)

Joint inversion targeting at the same parameter through petrophysics.

This is just syntactic sugar for the combination of *pygimli.frameworks.PetroModelling* (page 300) and *pygimli.frameworks.JointModelling* (page 287).

__init__(*petros*, *mgrs*)

Initialize with lists of managers and transformations

checkData(*data*)

Collect data values.

checkError(*err*, *data=None*)

Collect error values.

invert(*data*, ***kwargs*)

Run inversion

class pygimli.frameworks.**LCInversion**(*fop=None*, ***kwargs*)

Bases: *Inversion* (page 283)

Quasi-2D Laterally constrained inversion (LCI) framework.

__init__(*fop=None*, ***kwargs*)

prepare(*dataVals*, *errorVals*, *nLayers=4*, ***kwargs*)

run(*dataVals*, *errorVals*, *nLayers=4*, ***kwargs*)

Run inversion.

The inversion will always start from the starting model taken from the forward operator. If you want to run the inversion from a specified prior model, e.g., from a other run, set this model as starting model to the FOP (fop.setStartModel). Any self.inv.setModel() settings will be overwritten.

Parameters

- **dataVals** (*iterable*) – Data values
- **errorVals** (*iterable*) – Relative error values. dv / v

Keyword Arguments

- **maxIter** (*int*) – Overwrite class settings for maximal iterations number.
- **dPhi** (*float [1]*) – Overwrite class settings for delta data phi aborting criteria. Default is 1%
- **cType** (*int [1]*) – Temporary global contraint type for all regions.
- **startModel** (*array*) – Temporary starting model for the current inversion run.
- **lam** (*float*) – Temporary regularization parameter lambda.
- **lambdaFactor** (*float [1]*) – factor to change lam with every iteration
- **robustData** (*bool*) – robust (L1 norm mimicking) data reweighting
- **blockyModel** (*bool*) – robust (L1 norm mimicking) model roughness reweighting
- **isReference** (*bool [False]*) – starting model is also a reference to constrain against
- **showProgress** (*bool*) – show progress in form of updating models
- **verbose** (*bool*) – verbose output on the console

- **debug** (*bool*) – even verboser console and file output

class pygimli.frameworks.**LCModelling**(*fop*, ***kwargs*)

Bases: *Modelling* (page 296)

2D Laterally constrained (LC) modelling.

2D Laterally constrained (LC) modelling based on BlockMatrices.

__init__(*fop*, ***kwargs*)

Parameters: fop class .

createDefaultStartModel(*models*)

Create the default startmodel as the median of the data values.

createJacobian(*par*)

Create Jacobian matrix by creating individual Jacobians.

createParametrization(*nSoundings*, *nLayers=4*, *nPar=1*)

Create LCI mesh and suitable constraints informations.

Parameters

- **nLayers** (*int*) – Numbers of depth layers
- **nSoundings** (*int*) – Numbers of 1D measurements to laterally constrain
- **nPar** (*int*) – Numbers of independent parameter types, e.g., nPar = 1 for VES (invert for resisitivies), nPar = 2 for VESC (invert for resisitivies and phases)

drawModel(*ax*, *model*, ***kwargs*)

initJacobian(*dataVals*, *nLayers*, *nPar=None*)

Parameters

dataVals (*ndarray | RMatrix | list*) – Data values of size (nSounding x Data per sounding). All data per sounding need to be equal in length. If they don't fit into a matrix use list of sounding data.

initModelSpace(*nLayers*)

API

response(*par*)

Cut together forward responses of all soundings.

setDataBasis(***kwargs*)

Set homogeneous data basis.

Set a common data basis to all forward operators. If you want individual you need to set them manually.

class pygimli.frameworks.**LinearModelling**(*A*)

Bases: *Modelling* (page 296)

Modelling class for linearized problems with a given matrix.

__init__(*A*)

Initialize by storing the (reference to the) matrix.

createJacobian(*model*)

Do not compute a jacobian (linear).

property parameterCount

Define the number of parameters from the matrix size.

response(*model*)

Linearized forward modelling by matrix-vector product.

class pygimli.frameworks.**MarquardtInversion**(*fop=None*, ***kwargs*)

Bases: *Inversion* (page 283)

Marquardt scheme, i.e. local damping with decreasing strength.

__init__(*fop=None*, ***kwargs*)

run(*dataVals*, *errorVals*, ***kwargs*)

Parameters

****kwargs** – Forwarded to the parent class. See: pygimli.modelling.Inversion

class pygimli.frameworks.**MeshMethodManager**(***kwargs*)

Bases: *MethodManager* (page 292)

__init__(***kwargs*)

Constructor.

8.1.3.1 Attribute

mesh: GIMLI::Mesh

Copy of the main mesh to be distributed to inversion and the fop. You can overwrite it with invert(mesh=mesh).

applyMesh(*mesh*, *ignoreRegionManager=False*, ***kwargs*)

coverage()

Return coverage vector considering the logarithmic transformation.

createMesh(*data=None*, ***kwargs*)

API, implement in derived classes.

invert(*data=None*, *mesh=None*, *startModel=None*, ***kwargs*)

Run the full inversion.

Parameters

- **data** (*pg.DataContainer*) –
- **mesh** (GIMLI::Mesh [None]) –
- **startModel** (*float | iterable [None]*) – If set to None fop.createDefaultStartModel(dataValues) is called.

Keyword Arguments

- **zWeight** (*float [None]*) – Set zWeight or use defaults from regionManager.
- **correlationLengths** (*[float, float, float]*) – Correlation lengths for geostatistical regularization
- **dip** (*float*) – rotation axis between first and last dimension (x and z)
- **strike** (*float*) – rotation axis between first and second dimension (x and y)
- **limits** (*[float, float]*) – lower and upper value bounds for logarithmic transformation
- **Inversion.run** (*Al other are forwarded to*) –

Returns

model – Model mapped for match the paraDomain Cell markers. The calculated model vector (unmapped) is in self.fw.model.

Return type

array

property paraDomain

paraModel(*model=None*)

Give the model parameter regarding the parameter mesh.

setMesh(*mesh*, ***kwargs*)

Set a mesh and distribute it to the forward operator

showFit(*axs=None*, ***kwargs*)

Show data and the inversion result model response.

standardizedCoverage(*threshhold=0.01*)

Return standardized coverage vector (0|1) using thresholding.

class pygimli.frameworks.**MeshModelling**(***kwargs*)

Bases: *Modelling* (page 296)

Modelling class with a mesh discretization.

__init__(***kwargs*)

Variables

- **fop** (*pg.frameworks.Modelling*) –
- **data** (*pg.DataContainer*) –
- **modelTrans** (*[pg.trans.TransLog()]*) –

Parameters

****kwargs** – fop : Modelling

createConstraints()

Create constraint matrix.

createFwdMesh_()

createRefinedFwdMesh(*mesh*)

Refine the current mesh for higher accuracy.

This is called automatic when accessing self.mesh() so it ensures any effect of changing region properties (background, single).

drawModel(*ax*, *model*, ***kwargs*)

Draw the model as mesh-based distribution.

ensureContent()

Internal function to ensure there is a valid initialized mesh.

Initialization means the cell marker are recounted and/or there was a mesh refinement or boundary enlargement, all to fit the needs for the method-depending forward problem.

mesh()

Returns the current used mesh.

property paraDomain

Return parameter (inverse) mesh.

paraModel(*model*)

setDefaultBackground()

Set the lowest region to background if several exist.

setMesh(*mesh*, *ignoreRegionManager=False*)

Set mesh and specify whether region manager can be ignored.

setMeshPost(*data*)

Interface to be called when the mesh has been set successfully.

Might be overwritten by child classes.

class pygimli.frameworks.**MethodManager**(*fop=None*, *fw=None*, *data=None*, ***kwargs*)

Bases: object

General manager to maintenance a measurement method.

Method Manager are the interface to end-user interaction and can be seen as simple but complete application classes which manage all tasks of geophysical data processing.

The method manager holds one instance of a forward operator and an appropriate inversion framework to handle modelling and data inversion.

Method Manager also helps with data import and export, handle measurement data error estimation as well as model and data visualization.

Variables

- **verbose** (*bool*) – Give verbose output.
- **debug** (*bool*) – Give debug output.
- **fop** (*pygimli.frameworks.Modelling* (page 296)) – Forward Operator instance .. knows the physics. fop is initialized by pygimli.manager.MethodManager.initForwardOperator and calls a valid pygimli.manager.MethodManager.createForwardOperator method in any derived classes.

- **inv** (*pygimli.frameworks.Inversion* (page 283).) – Inversion framework instance .. knows the reconstruction approach. The attribute inv is initialized by default but can be changed overwriting `pygimli.manager.MethodManager.initInversionFramework`

__init__(*fop=None*, *fw=None*, *data=None*, ***kwargs*)

Constructor.

applyData(*data*)

checkData(*data*)

Overwrite for special checks to return data values

checkError(*err*, *dataVals=None*)

Return relative error. Default we assume 'err' are relative values. Overwrite is derived class if needed.

static createArgParser(*dataSuffix='dat'*)

Create default argument parser.

TODO move this to some kind of app class

Create default argument parser for the following options:

-Q, –quiet -R, –robustData: options.robustData -B, –blockyModel: options.blockyModel -l, –lambda: options.lam -i, –maxIter: options.maxIter –depth: options.depth

createForwardOperator(***kwargs*)

Mandatory interface for derived classes.

Here you need to specify which kind of forward operator FOP you want to use. This is called by any initForwardOperator() call.

Parameters

****kwargs** – Any arguments that are necessary for your FOP creation.

Returns

Instance of any kind of `pygimli.framework.Modelling`.

Return type

Modelling (page 296)

createInversionFramework(***kwargs*)

Create default Inversion framework.

Derived classes may overwrite this method.

Parameters

****kwargs** – Any arguments that are necessary for your creation.

Returns

Instance of any kind of `pygimli.framework.Inversion`.

Return type

Inversion (page 283)

property debug

estimateError(*data*, *errLevel=0.01*, *absError=None*)

Estimate data error.

Create an error of estimated measurement error. On default it returns an array of constant relative errors. More sophisticated error estimation should be done in specialized derived classes.

Parameters

- **data** (*iterable*) – Data values for which the errors should be estimated.
- **errLevel** (*float (0.01)*) – Error level in percent/100 (i.e., 3% = 0.03).
- **absError** (*float (None)*) – Absolute error in the unit of the data.

Returns

err – Returning array of size len(data)

Return type

array

property fop

property fw

property inv

invert(*data=None*, *err=None*, ***kwargs*)

Invert the data.

Invert the data by calling self.inv.run() with mandatory data and error values.

TODO

*need dataVals mandatory? what about already loaded data

Parameters

- **dataVals** (*iterable*) – Data values to be inverted.
- **errVals** (*iterable | float*) – Error value for the given data. If errVals is float we assume this means to be a global relative error and force self.estimateError to be called.

load(*fileName*)

API, overwrite in derived classes.

property model

postRun(**args*, ***kwargs*)

Called just after the inversion run.

preRun(**args*, ***kwargs*)

Called just before the inversion run starts.

reinitForwardOperator(***kwargs*)

Reinitialize the forward operator.

Sometimes it can be useful to reinitialize the forward operator. Keyword arguments will be forwarded to 'self.createForwardOperator'.

setData(*data*)

Set a data and distribute it to the forward operator

showData(*data=None*, *ax=None*, ***kwargs*)

Show the data.

Draw data values into a given axes or show the data values from the last run. Forwards on default to the self.fop.drawData function of the modelling operator. If there is no given function given, you have to override this method.

Parameters

- **ax** (*mpl axes*) – Axes object to draw into. Create a new if its not given.
- **data** (*iterable | pg.DataContainer*) – Data values to be draw.

Return type
ax, cbar

showFit(*ax=None*, ***kwargs*)

Show the last inversion data and response.

showModel(*model*, *ax=None*, ***kwargs*)

Show a model.

Draw model into a given axes or show inversion result from last run. Forwards on default to the self.fop.drawModel function of the modelling operator. If there is no function given, you have to override this method.

Parameters

- **ax** (*mpl axes*) – Axes object to draw into. Create a new if its not given.
- **model** (*iterable*) – Model data to be draw.

Return type
ax, cbar

showResult(*model=None*, *ax=None*, ***kwargs*)

Show the last inversion result.

Parameters

- **ax** (*mpl axes*) – Axes object to draw into. Create a new if its not given.
- **model** (*iterable [None]*) – Model values to be draw. Default is self.model from the last run

Return type
ax, cbar

showResultAndFit(***kwargs*)

Calls showResults and showFit.

8.1.3.2 Keyword Args

saveFig: str[None]
If not None save figure.

axs: [mpl.Axes]
Give 3 axes and its plotted into them instead of creating 3 new.

simulate(*model*, ***kwargs*)

property verbose

class pygimli.frameworks.**MethodManager1d**(*fop=None*, ***kwargs*)

Bases: *MethodManager* (page 292)

Method Manager base class for managers on a 1d discretization.

__init__(*fop=None*, ***kwargs*)

Constructor.

createInversionFramework(***kwargs*)

invert(*data=None*, *err=None*, ***kwargs*)

class pygimli.frameworks.**Modelling**(***kwargs*)

Bases: ModellingBaseMT__

Abstract Forward Operator.

Abstract Forward Operator that is or can use a Modelling instance. Can be seen as some kind of proxy Forward Operator.

__init__(***kwargs*)

Variables

- **fop** (*pg.frameworks.Modelling*) –
- **data** (*pg.DataContainer*) –
- **modelTrans** (*[pg.trans.TransLog()]*) –

Parameters
****kwargs** – fop : Modelling

clearRegionProperties()

Clear all region parameter.

createDefaultStartModel(*dataVals*)

Create the default startmodel as the median of the data values.

createStartModel(*dataVals=None*)

Create the default startmodel as the median of the data values.

Overwriting might be a good idea. Its used by inversion to create a valid startmodel if there are no starting values from the regions.

property data

Return the associated data container.

C++ signature :
GIMLI::DataContainer {lvalue} data(GIMLI::ModellingBase {lvalue})

drawData(*ax*, *data*, ***kwargs*)

drawModel(*ax*, *model*, ***kwargs*)

ensureContent()

estimateError(*data*, ***kwargs*)

Create data error fallback when the data error is not known. Should be implemented method depending.

property fop

initModelSpace(***kwargs*)

API

property modelTrans

property parameterCount

regionManager()

regionProperties(*regionNr=None*)

Return dictionary of all properties for region number regionNr.

setData(*data*)

setDataContainer(*data*)

setDataPost(*data*)

Called when the dataContainer has been set sucessfully.

setDataSpace(***kwargs*)

Set data space, e.g., DataContainer, times, coordinates.

setInterRegionCoupling(*region1*, *region2*, *weight=1.0*)

Set the weighting for constraints across regions.

setRegionProperties(*regionNr*, ***kwargs*)

Set region properties. regionNr can be '*' for all regions.

startModel=None, limits=None, trans=None, cType=None, zWeight=None, modelControl=None, background=None, fix=None, single=None, correlationLengths=None, dip=None, strike=None

Parameters

- **regionNr** (*int, [ints], '**') – Region number, list of numbers, or wildcard "*" for all.
- **startModel** (*float*) – starting model value
- **limits** (*[float, float]*) – lower and upper limit for value using a barrier transform

- **trans** (*str*) – transformation for model barrier: “log”, “cot”, “lin”
- **cType** (*int*) – constraint (regularization) type
- **zWeight** (*float*) – relative weight for vertical boundaries
- **background** (*bool*) – exclude region from inversion completely (prolongation)
- **fix** (*float*) – exclude region from inversion completely (fix to value)
- **single** (*bool*) – reduce region to one unknown
- **correlationLengths** (*[floats]*) – correlation lengths for geostatistical inversion (x’, y’, z’)
- **dip** (*float [0]*) – angle between x and x’ (first correlation length)
- **strike** (*float [0]*) – angle between y and y’ (second correlation length)

class pygimli.frameworks.**MultiFrameModelling**(*modellingOperator*, *scalef=1.0*)

Bases: *MeshModelling* (page 291)

Full frame (multiple fop parallel) forward modelling.

__init__(*modellingOperator*, *scalef=1.0*)

Init class and jacobian matrix.

createConstraints()

Create constraint matrix (special type for this).

createDefaultStartModel()

Create the default startmodel as the median of the data values.

createJacobian(*(object)arg1*, *(object)model*) → object :

C++ signature :
void* createJacobian(GIMLI::ModellingBase {lvalue},GIMLI::Vector<double>)

createJacobian((object)arg1, (object)model) -> object :

C++ signature :
void* createJacobian(ModellingBase_wrapper {lvalue},GIMLI::Vector<double>)

createJacobian((object)arg1, (object)model, (object)resp) -> object :

C++ signature :
void* createJacobian(GIMLI::ModellingBase {lvalue},GIMLI::Vector<double>,GIMLI::Vector<double>)

createJacobian((object)arg1, (object)model, (object)resp) -> object :

C++ signature :
void* createJacobian(ModellingBase_wrapper {lvalue},GIMLI::Vector<double>,GIMLI::Vector<double>)

createStartModel(*dataVals*)

Create the default startmodel as the median of the data values.

Overwriting might be a good idea. Its used by inversion to create a valid startmodel if there are no starting values from the regions.

property parameterCount

prepareJacobian()

Build up Jacobian block matrix (once the sizes are known).

response(*(object)arg1*, *(object)model*) → object :

C++ signature :
GIMLI::Vector<double> response(GIMLI::ModellingBase {lvalue},GIMLI::Vector<double>)

response((object)arg1, (object)model) -> object :

C++ signature :
GIMLI::Vector<double> response(ModellingBase_wrapper {lvalue},GIMLI::Vector<double>)

setData(*alldata*, *modellingOperator=None*)

Distribute the data containers amongst the fops.

setDefaultBackground()

Set the default background behaviour.

setMeshPost(*mesh*)

Interface to be called when the mesh has been set successfully.

Might be overwritten by child classes.

class pygimli.frameworks.**ParameterInversionManager**(*funct=None*, *fop=None*, ***kwargs*)

Bases: *MethodManager* (page 292)

Framework to invert unconstrained parameters.

__init__(*funct=None*, *fop=None*, ***kwargs*)

Constructor.

createInversionFramework(***kwargs*)

invert(*data=None*, *err=None*, ***kwargs*)

Parameters

- **limits** (*{str: [min, max]}*) – Set limits for parameter by parameter name.
- **startModel** (*{str: startModel}*) – Set the start value for parameter by parameter name.

class pygimli.frameworks.**ParameterModelling**(*funct=None*, ***kwargs*)

Bases: *Modelling* (page 296)

Model with symbolic parameter names instead of numbers

__init__(*funct=None*, ***kwargs*)

Variables

- **fop** (*pg.frameworks.Modelling*) –
- **data** (*pg.DataContainer*) –
- **modelTrans** (*[pg.trans.TransLog()]*) –

Parameters
****kwargs** – fop : Modelling

addParameter(*name*, *id=None*, ***kwargs*)

drawModel(*ax*, *model*)

property params

response(*(object)arg1*, *(object)model*) → object :

C++ signature :
GIMLI::Vector<double> response(GIMLI::ModellingBase {lvalue},GIMLI::Vector<double>)

response((object)arg1, (object)model) -> object :

C++ signature :
GIMLI::Vector<double> response(ModellingBase_wrapper {lvalue},GIMLI::Vector<double>)

setRegionProperties(*k*, ***kwargs*)

Set Region Properties by parameter name.

class pygimli.frameworks.**PetroInversionManager**(*petro*, *mgr=None*, ***kwargs*)

Bases: *MeshMethodManager* (page 290)

Class for petrophysical inversion (s. Rücker et al. 2017).

__init__(*petro*, *mgr=None*, ***kwargs*)

Initialize instance with manager and petrophysical relation.

class pygimli.frameworks.**PetroModelling**(*fop*, *petro*, ***kwargs*)

Bases: *MeshModelling* (page 291)

Combine petrophysical relation with the modelling class f(p).

Combine petrophysical relation $p(m)$ with a modelling class $f(p)$ to invert for the petrophysical model p instead of the geophysical model m.

p be the petrophysical model, e.g., porosity, saturation, ... m be the geophysical model, e.g., slowness, resistivity, ...

__init__(*fop*, *petro*, ***kwargs*)

Save forward class and transformation, create Jacobian matrix.

createJacobian(*model*)

Fill the individual jacobian matrices. J = dF(m) / dm = dF(m) / dp * dp / dm

createStartModel(*data*)

Use inverse transformation to get m(p) for the starting model.

property petro

response(*model*)

Use transformation to get p(m) and compute response f(p).

setDataPost(*data*)

setMeshPost(*mesh*)

class pygimli.frameworks.**PriorModelling**(*mesh=None*, *pos=None*, ***kwargs*)

Bases: *MeshModelling* (page 291)

Forward operator for grabbing values out of a mesh (prior data).

__init__(*mesh=None*, *pos=None*, ***kwargs*)

Init with mesh and some positions that are converted into ids.

createJacobian(*model*)

Do nothing (linear).

createRefinedFwdMesh(*mesh*)

Refine the current mesh for higher accuracy.

This is called automatic when accessing self.mesh() so it ensures any effect of changing region properties (background, single).

response(*model*)

Return values at the indexed cells.

setMesh(*mesh*)

Set mesh and specify whether region manager can be ignored.

8.2 pygimli.math

Math functions and matrices.

8.2.1 Overview

Functions

angle (page 302)(p1, p2, p3)	C++ signature :
besselI0 (page 302)(x)	Caluculate modified Bessel function of the first kind See Abramowitz: Handbook of math.
besselI1 (page 303)(x)	Caluculate modified Bessel function of the first kind See Abramowitz: Handbook of math.
besselK0 (page 303)(x)	Caluculate modified Bessel function of the second kind See Abramowitz: Handbook of math.

continues on next page

Table 1 – continued from previous page

besselK1 (page 303)(x)	Caluculate modified Bessel function of the second kind See Abramowitz: Handbook of math.
cos (page 303)(a)	C++ signature :
cot (page 303)(a)	C++ signature :
createCm05 (page 303)(*args, **kwargs)	
det (page 303)(A)	Return determinant for Matrix A.
dot (page 304)(A, B, c, ret)	C++ signature :
exp (page 305)(a)	C++ signature :
exp10 (page 305)(a)	C++ signature :
imag (page 305)(A)	C++ signature :
log (page 305)(a)	C++ signature :
log10 (page 306)(a)	C++ signature :
max (page 306)(v)	C++ signature :
median (page 306)(a)	C++ signature :
min (page 306)(v)	C++ signature :
pow (page 307)(v, int) is misinterpreted as pow)	so we need to fix this
rand (page 307)(vec [[, min, max])	C++ signature :
randn (page 307)(vec)	C++ signature :
real (page 307)(A)	C++ signature :
rms (page 308)(a)	C++ signature :
round (page 308)(v, tol)	C++ signature :
rrms (page 308)(a, b)	C++ signature :
sign (page 308)(a)	C++ signature :
sin (page 308)(a)	C++ signature :
sqrt (page 308)(a)	C++ signature :
sum (page 308)(c)	Templates argue with python bindings
toComplex (page 308)(re, im)	C++ signature :
unique (page 309)(a)	C++ signature :

8.2.2 Functions

pygimli.math.**angle**(*(object)p1*, *(object)p2*, *(object)p3*) → object :

C++ signature :
double angle(GIMLI::Pos,GIMLI::Pos,GIMLI::Pos)

angle((object)z) -> object :

C++ signature :
GIMLI::Vector<double> angle(GIMLI::Vector<std::complex<double> >)

angle((object)b, (object)a) -> object :

C++ signature :
GIMLI::Vector<double> angle(GIMLI::Vector<double>,GIMLI::Vector<double>)

pygimli.math.**besselI0**(*(object)x*) → object :

Caluculate modified Bessel function of the first kind See Abramowitz: Handbook of math. functions; DLLEXPORT double besselI0(double x);

C++ signature :
double besselI0(double)

pygimli.math.**besselI1**(*(object)x*) → object :

Caluculate modified Bessel function of the first kind See Abramowitz: Handbook of math. functions DLLEXPORT double besselI1(double x);

C++ signature :
double besselI1(double)

pygimli.math.**besselK0**(*(object)x*) → object :

Caluculate modified Bessel function of the second kind See Abramowitz: Handbook of math. functions DLLEXPORT double besselK0(double x);

C++ signature :
double besselK0(double)

Examples using `pygimli.math.besselK0`

- *Geoelectrics in 2.5D* (page 76)

pygimli.math.**besselK1**(*(object)x*) → object :

Caluculate modified Bessel function of the second kind See Abramowitz: Handbook of math. functions DLLEXPORT double besselK1(double x);

C++ signature :
double besselK1(double)

Examples using `pygimli.math.besselK1`

- *Geoelectrics in 2.5D* (page 76)

pygimli.math.**cos**(*(object)a*) → object :

C++ signature :
GIMLI::Vector<double> cos(GIMLI::Vector<double>)

pygimli.math.**cot**(*(object)a*) → object :

C++ signature :
GIMLI::Vector<double> cot(GIMLI::Vector<double>)

cot((object)a) -> object :

C++ signature :
double cot(double)

pygimli.math.**createCm05**(**args*, ***kwargs*)

pygimli.math.**det**(*(object)A*) → object :

Return determinant for Matrix A. This function is a stub. Only Matrix dimensions of 2 and 3 are considered.

C++ signature :
double det(GIMLI::Matrix3<double>)

det((object)A) -> object :
Return determinant for Matrix A. This function is a stub. Only Matrix dimensions of 2 and 3 are considered.

C++ signature :
double det(GIMLI::Matrix3<double>)

det((object)A) -> object :
Return determinant for Matrix A. This function is a stub. Only Matrix dimensions of 2 and 3 are considered.

C++ signature :
double det(GIMLI::Matrix<double>)

pygimli.math.**dot**(*(object)A*, *(object)B*, *(object)c*, *(object)ret*) → object :

C++ signature :
void* dot(GIMLI::ElementMatrix<double>,GIMLI::ElementMatrix<double>,double,GIMLI::ElementM
{lvalue})

dot((object)A, (object)B, (object)c, (object)ret) -> object :

C++ signature :
void* dot(GIMLI::ElementMatrix<double>,GIMLI::ElementMatrix<double>,GIMLI::Pos,GIMLI::Elem
{lvalue})

dot((object)A, (object)B, (object)c, (object)ret) -> object :

C++ signature :
void* dot(GIMLI::ElementMatrix<double>,GIMLI::ElementMatrix<double>,GIMLI::Matrix<double>,
{lvalue})

dot((object)A, (object)B, (object)c, (object)ret) -> object :

C++ signature :
void* dot(GIMLI::ElementMatrix<double>,GIMLI::ElementMatrix<double>,GIMLI::FEAFunction,GI
{lvalue})

dot((object)A, (object)B, (object)c) -> object :

C++ signature :
GIMLI::ElementMatrix<double> dot(GIMLI::ElementMatrix<double>,GIMLI::ElementMatrix<double

dot((object)A, (object)B, (object)c) -> object :

C++ signature :
GIMLI::ElementMatrix<double> dot(GIMLI::ElementMatrix<double>,GIMLI::ElementMatrix<double

dot((object)A, (object)B, (object)c) -> object :

C++ signature :
GIMLI::ElementMatrix<double> dot(GIMLI::ElementMatrix<double>,GIMLI::ElementMatrix<double

dot((object)A, (object)B [, (object)c]) -> object :

C++ signature :
GIMLI::ElementMatrix<double> dot(GIMLI::ElementMatrix<double>,GIMLI::ElementMatrix<double
[,GIMLI::FEAFunction])

dot((object)A, (object)B, (object)ret) -> object :

C++ signature :
void* dot(GIMLI::ElementMatrix<double>,GIMLI::ElementMatrix<double>,GIMLI::ElementMatrix<
{lvalue})

dot((object)v1, (object)v2) -> object :

C++ signature :
double dot(GIMLI::Vector<double>,GIMLI::Vector<double>)

pygimli.math.**exp**(*(object)a*) → object :

C++ signature :
GIMLI::Vector<double> exp(GIMLI::Vector<double>)

pygimli.math.**exp10**(*(object)a*) → object :

C++ signature :
GIMLI::Vector<double> exp10(GIMLI::Vector<double>)

exp10((object)a) -> object :

C++ signature :
double exp10(double)

pygimli.math.**imag**(*(object)A*) → object :

C++ signature :
GIMLI::SparseMapMatrix<double, unsigned long> imag(GIMLI::SparseMapMatrix<std::complex<double>, unsigned long>)

imag((object)A) -> object :

C++ signature :
GIMLI::SparseMatrix<double> imag(GIMLI::SparseMatrix<std::complex<double> >)

imag((object)cv) -> object :

C++ signature :
GIMLI::Matrix<double> imag(GIMLI::Matrix<std::complex<double> >)

imag((object)A) -> object :

C++ signature :
GIMLI::Matrix<double> imag(GIMLI::Matrix<std::complex<double> >)

imag((object)cv) -> object :

C++ signature :
GIMLI::Vector<double> imag(GIMLI::Vector<std::complex<double> >)

pygimli.math.**log**(*(object)a*) → object :

C++ signature :
GIMLI::Vector<double> log(GIMLI::Vector<double>)

log((object)type, (object)msg) -> object :

C++ signature :
void* log(GIMLI::LogType,std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >)

log((object)type, (object)vs) -> object :

C++ signature :
void* log(GIMLI::LogType,char const*)

log((object)type, (object)vs, (object)vs) -> object :

C++ signature :
void* log(GIMLI::LogType,char const*,char const*)

log((object)type, (object)vs) -> object :

C++ signature :
void* log(GIMLI::LogType,std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >)

pygimli.math.**log10**(*(object)a*) → object :

C++ signature :
GIMLI::Vector<double> log10(GIMLI::Vector<double>)

pygimli.math.**max**(*(object)v*) → object :

C++ signature :
int max(std::vector<int, std::allocator<int> >)

max((object)v) -> object :

C++ signature :
std::complex<double> max(GIMLI::Vector<std::complex<double> >)

max((object)v) -> object :

C++ signature :
unsigned long max(GIMLI::Vector<unsigned long>)

max((object)v) -> object :

C++ signature :
double max(GIMLI::Vector<double>)

max((object)a, (object)b) -> object :

C++ signature :
int max(int,unsigned long)

max((object)a, (object)b) -> object :

C++ signature :
unsigned long max(unsigned long,unsigned long)

max((object)a, (object)b) -> object :

C++ signature :
double max(double,double)

pygimli.math.**median**(*(object)a*) → object :

C++ signature :
double median(GIMLI::Vector<double>)

pygimli.math.**min**(*(object)v*) → object :

C++ signature :
std::complex<double> min(GIMLI::Vector<std::complex<double> >)

min((object)v) -> object :

C++ signature :
double min(GIMLI::Vector<double>)

min((object)a, (object)b) -> object :

C++ signature :
unsigned long min(unsigned long,unsigned long)

min((object)a, (object)b) -> object :

C++ signature :
double min(double,double)

pygimli.math.**pow**(*v*, *int) is misinterpreted as pow(v, rvec(int)*)

so we need to fix this

Examples using pygimli.math.pow

- *Polyfit* (page 229)

pygimli.math.**rand**(*(object)vec*[, *(object)min=0.0*[, *(object)max=1.0*]]) → object :

C++ signature :
void* rand(GIMLI::Vector<double> {lvalue} [,double=0.0 [,double=1.0]])

pygimli.math.**randn**(*(object)vec*) → object :

C++ signature :
void* randn(GIMLI::Vector<double> {lvalue})

randn((object)n) -> object :
Create a array of len n with normal distributed randomized values.

C++ signature :
GIMLI::Vector<double> randn(unsigned long)

pygimli.math.**real**(*(object)A*) → object :

C++ signature :
GIMLI::SparseMapMatrix<double, unsigned long> real(GIMLI::SparseMapMatrix<std::complex<double>, unsigned long>)

real((object)A) -> object :

C++ signature :
GIMLI::SparseMatrix<double> real(GIMLI::SparseMatrix<std::complex<double> >)

real((object)cv) -> object :

C++ signature :
GIMLI::Matrix<double> real(GIMLI::Matrix<std::complex<double> >)

real((object)A) -> object :

C++ signature :
GIMLI::Matrix<double> real(GIMLI::Matrix<std::complex<double> >)

real((object)cv) -> object :

C++ signature :
GIMLI::Vector<double> real(GIMLI::Vector<std::complex<double> >)

pygimli.math.**rms**(*(object)a*) → object :

C++ signature :
double rms(GIMLI::Vector<double>)

rms((object)a, (object)b) -> object :

C++ signature :
double rms(GIMLI::Vector<double>,GIMLI::Vector<double>)

pygimli.math.**round**(*(object)v*, *(object)tol*) → object :

C++ signature :
GIMLI::Vector<double> round(GIMLI::Vector<double>,double)

pygimli.math.**rrms**(*(object)a*, *(object)b*) → object :

C++ signature :
double rrms(GIMLI::Vector<double>,GIMLI::Vector<double>)

pygimli.math.**sign**(*(object)a*) → object :

C++ signature :
GIMLI::Vector<double> sign(GIMLI::Vector<double>)

sign((object)a) -> object :

C++ signature :
double sign(double)

pygimli.math.**sin**(*(object)a*) → object :

C++ signature :
GIMLI::Vector<double> sin(GIMLI::Vector<double>)

pygimli.math.**sqrt**(*(object)a*) → object :

C++ signature :
GIMLI::Vector<double> sqrt(GIMLI::Vector<double>)

pygimli.math.**sum**(*(object)c*) → object :

Templates argue with python bindings

C++ signature :
std::complex<double> sum(GIMLI::Vector<std::complex<double> >)

sum((object)r) -> object :

C++ signature :
double sum(GIMLI::Vector<double>)

sum((object)i) -> object :

C++ signature :
long sum(GIMLI::Vector<long>)

pygimli.math.**toComplex**(*(object)re*, *(object)im*) → object :

C++ signature :
GIMLI::Vector<std::complex<double> > toComplex(GIMLI::Vector<double>,GIMLI::Vector<double>)

toComplex((object)re [, (object)im=0.0]) -> object :

C++ signature :
GIMLI::Vector<std::complex<double> > toComplex(GIMLI::Vector<double> [,double=0.0])

toComplex((object)re, (object)im) -> object :

C++ signature :
GIMLI::Vector<std::complex<double> > toComplex(double,GIMLI::Vector<double>)

pygimli.math.**unique**(*(object)a*) → object :

C++ signature :
std::vector<long, std::allocator<long> > unique(std::vector<long, std::allocator<long> >)

unique((object)a) -> object :
Returning a copy of the vector and replacing all consecutive occurrences of a value by a single instance of that value. e.g. [0 1 1 2 1 1] -> [0 1 2 1]. To remove all double values from the vector use an additionally sorting. e.g. unique(sort(v)) gets you [0 1 2].

C++ signature :
GIMLI::Vector<double> unique(GIMLI::Vector<double>)

unique((object)a) -> object :
Returning a copy of the vector and replacing all consecutive occurrences of a value by a single instance of that value. e.g. [0 1 1 2 1 1] -> [0 1 2 1]. To remove all double values from the vector use an additionally sorting. e.g. unique(sort(v)) gets you [0 1 2].

C++ signature :
GIMLI::Vector<long> unique(GIMLI::Vector<long>)

8.3 pygimli.meshtools

Mesh generation and modification.

Note: Although we discriminate here between grids (structured meshes) and meshes (unstructured), both objects are treated the same internally.

8.3.1 Overview

Functions

appendBoundary (page 312)(mesh, **kwargs)	Append Boundary to a given mesh.
appendBoundaryGrid (page 313)(grid[, xbound, ybound, ...])	Return a copy of grid surrounded by a boundary grid.
appendTetrahedronBoundary (page 313)(mesh[, xbound, ...])	Return a copy of mesh surrounded by a tetrahedron mesh as boundary.
appendTriangleBoundary (page 315)(mesh[, xbound, ...])	Add a triangle mesh boundary to a given mesh.
cellDataToBoundaryData (page 317)(mesh, data)	TODO DOCUMENT_ME
cellDataToNodeData (page 317)(mesh, data[, style])	Convert cell data to node data.
convert (page 318)(mesh[, verbose])	Convert mesh from foreign formats.
convertHDF5Mesh (page 318)(h5Mesh[, group, indices, ...])	Converts instance of a hdf5 mesh to a GIMLI::Mesh.
convertMeshioMesh (page 318)(mesh[, verbose])	Convert mesh from meshio object.
createCircle (page 318)([pos, radius, nSegments, ...])	Create simple circle polygon.
createCube (page 320)([size, pos, start, end, rot, ...])	Create cube PLC as geometrie definition.
createCylinder (page 321)([radius, height, nSegments, ...])	Create PLC of a cylinder.
createFacet (page 321)(mesh[, boundaryMarker, verbose])	Create a coplanar PLC of a 2d mesh or poly
createGrid (page 321)([x, y, z])	Create grid style mesh.
createGridPieShaped (page 322)(x[, degree, h, marker])	Create a 2D pie shaped grid (segment from annulus or cirlce).
createLine (page 323)(start, end[, nSegments])	Create simple line polygon.
createMesh (page 325)(poly[, quality, area, smooth, ...])	Create a mesh for a given PLC or point list.
createMesh1D (page 328)(x)	Generate simple one dimensional mesh with nodes at position in RVector pos.
createMesh1DBlock (page 328)(nLayers [[, nProperties])	Generate 1D block model of thicknesses and properties
createMesh2D (page 329)(x, y [[, markerType])	Generate simple two dimensional mesh with nodes at position in RVector x and y.
createMesh3D (page 329)(x, y, z [[, markerType])	Generate simple three dimensional mesh with nodes at position in RVector x and y.
createMeshFromHull (page 329)(mesh[, fixNodes])	Create a new 2D triangular mesh from the boundaries of mesh.
createParaDomain2D (page 330)(*args, **kwargs)	API change here .

continues on next page

Table 2 – continued from previous page

createParaMesh (page 330)(data, **kwargs)	Create parameter mesh from list of sensor positions.
createParaMesh2DGrid (page 330)(sensors[, paraDX, ...])	Create a grid-style mesh for an inversion parameter mesh.
createParaMeshPLC (page 331)(sensors[, paraDX, ...])	Create a geometry (PLC) for an inversion parameter mesh.
createParaMeshPLC3D (page 333)(sensors[, paraDX, ...])	Create a geometry (PLC) for an 3D inversion parameter mesh.
createParaMeshSurface (page 334)(sensors[, ...])	Create surface mesh for an 3D inversion parameter mesh.
createPolygon (page 335)(verts[, isClosed, addNodes, ...])	Create a polygon from a list of vertices.
createRectangle (page 337)([start, end, pos, size])	Create rectangle polygon.
createSurface (page 339)(mesh[, boundaryMarker, verbose])	Convert a 2D mesh into a 3D surface mesh.
createWorld (page 339)(start, end[, marker, area, ...])	Create simple rectangular 2D or 3D world.
exportFenicsHDF5Mesh (page 340)(mesh, exportname)	Exports Gimli mesh in HDF5 format suitable for Fenics.
exportHDF5Mesh (page 341)(mesh, exportname[, group, ...])	Writes given GIMLI::Mesh in a hdf5 format file.
exportPLC (page 341)(poly, fname, **kwargs)	Export a piece-wise linear complex (PLC) to a .poly file (2D or 3D).
exportSTL (page 341)(mesh, fileName[, binary])	Write *STL* surface mesh and returns a GIMLI::Mesh.
extractUpperSurface2dMesh (page 341)(mesh[, zCut])	Extract 2d mesh from the upper surface of a 3D mesh.
extrude (page 342)(p2[, z, boundaryMarker])	Create 3D body by extruding a closed 2D poly into z direction
extrudeMesh (page 343)(mesh, a, **kwargs)	Extrude mesh to a higher dimension.
fillEmptyToCellArray (page 343)(mesh, vals[, slope])	Prolongate empty cell values to complete cell attributes.
fromSubsurface (page 345)(obj[, order, verbose])	Convert subsurface object to pygimli mesh.
interpolate (page 346)(*args, **kwargs)	Interpolation convinience function.
interpolateAlongCurve (page 348)(curve, t, **kwargs)	Interpolate along curve.
merge (page 349)(*args, **kwargs)	Little syntactic sugar to merge.
merge2Meshes (page 350)(m1, m2)	Merge two meshes into one new mesh and return the combined mesh.
mergeMeshes (page 350)(meshList[, verbose])	Merge several meshes into one new mesh and return the new mesh.
mergePLC (page 350)(plcs[, tol])	Merge multiply polygons.
mergePLC3D (page 351)(plcs[, tol])	Merge a list of 3D PLC into one
nodeDataToBoundaryData (page 352)(mesh, data)	Assuming [NodeCount, dim] data DOCUMENT_ME

continues on next page

Table 2 – continued from previous page

nodeDataToCellData (page 352)(mesh, data)	Convert node data to cell data.
quality (page 352)(mesh[, measure])	Return the quality of a given triangular mesh.
readFenicsHDF5Mesh (page 352)(fileName[, verbose])	Reads *FEniCS* mesh from file format .h5 and returns a GIMLI::Mesh.
readGmsh (page 352)(fName[, verbose, precision])	Read *Gmsh* ASCII file and return instance of GIMLI::Mesh class.
readHDF5Mesh (page 354)(fileName[, group, indices, ...])	Function for loading a mesh from HDF5 file format.
readHydrus2dMesh (page 355)([fileName])	Import mesh from Hydrus2D.
readHydrus3dMesh (page 356)([fileName])	Import mesh from Hydrus3D.
readMeshIO (page 356)(fileName[, verbose])	Generic mesh read using meshio.
readPLC (page 356)(filename[, comment])	Read in a piece-wise linear complex object (PLC) from .poly file.
readSTL (page 356)(fileName[, binary])	Read *STL* surface mesh and returns a GIMLI::Mesh.
readTetgen (page 356)(fName[, comment, verbose, ...])	Read and convert a mesh from the basic *Tetgen* output.
readTriangle (page 357)(fName[, verbose])	Read *Triangle* [**?**] mesh.
refineHex2Tet (page 357)(mesh[, style])	Refine mesh of hexahedra into a mesh of tetrahedra.
refineQuad2Tri (page 358)(mesh[, style])	Refine mesh of quadrangles into a mesh of triangle cells.
syscallTetgen (page 358)(filename[, quality, area, ...])	Create a mesh from a PLC by system-calling *Tetgen*.
tapeMeasureToCoordinates (page 360)(tape, pos)	Interpolate 2D tape measured topography to 2D Cartesian coordinates.
toSubsurface (page 361)(mesh[, verbose])	Create a subsurface object from pygimli mesh.

8.3.2 Functions

pygimli.meshtools.**appendBoundary**(*mesh*, ***kwargs*)

Append Boundary to a given mesh.

Syntactic sugar for *pygimli.meshtools.appendTriangleBoundary* (page 315) and *pygimli.meshtools.appendTetrahedronBoundary* (page 313).

Parameters

- **mesh** (GIMLI::Mesh) – "2d or 3d Mesh to which the boundary will be appended.
- **Args** (*Additional*) –
- **----------** –

:param **kwargs forwarded to *pygimli.meshtools.appendTriangleBoundary* (page 315): :param or *pygimli.meshtools.appendTetrahedronBoundary* (page 313).:

Returns

A new 2D or 3D mesh containing the original mesh and a boundary around.

Return type
GIMLI::Mesh

pygimli.meshtools.**appendBoundaryGrid**(*grid*, *xbound=None*, *ybound=None*, *zbound=None*, *marker=1*, *isSubSurface=True*, ***kwargs*)

Return a copy of grid surrounded by a boundary grid.

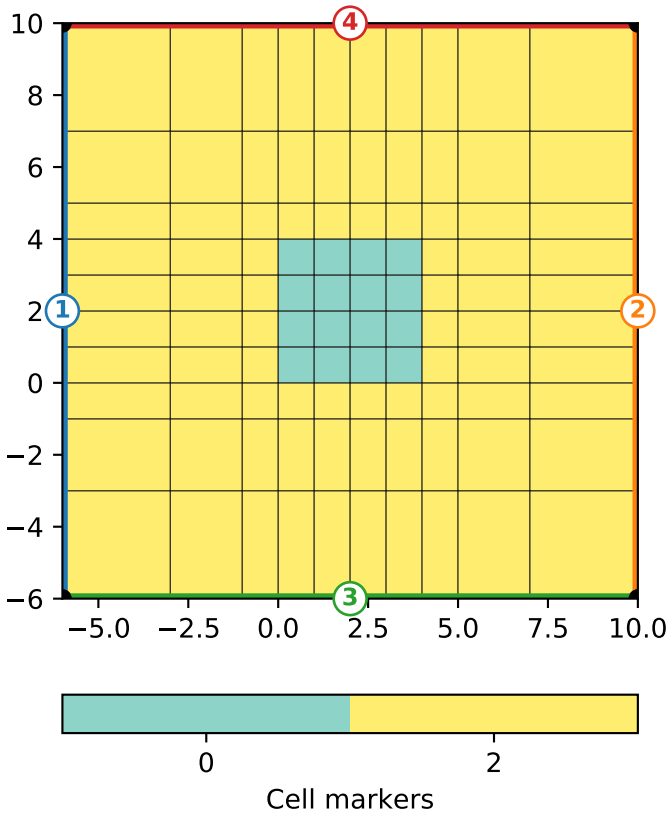
Note, the input grid needs to be a 2d or 3d grid with quad/hex cells.

Parameters

- **grid** (GIMLI::Mesh) – 2D or 3D Mesh that must contain structured quads or hex cells
- **xbound** (*iterable of type float [None]*) – Needed for 2D or 3D grid prolongation and will be added on the left side in opposit order and on the right side in normal order.
- **ybound** (*iterable of type float [None]*) – Needed for 2D or 3D grid prolongation and will be added (2D bottom, 3D fron) in opposit order and (2D top, 3D back) in normal order.
- **zbound** (*iterable of type float [None]*) – Needed for 3D grid prolongation and will be added the bottom side in opposit order on the top side in normal order.
- **marker** (*int [1]*) – Cellmarker for the cells in the boundary region
- **isSubSurface** (*boolean, optional*) – Apply boundary conditions suitable for geo-simulaion and prolongate mesh to the surface if necessary, e.i., no boundary on top of the grid.

```
>>> import pygimli as pg
>>> import pygimli.meshtools as mt
>>> grid = mt.createGrid(5,5)
...
>>> g1 = mt.appendBoundaryGrid(grid,
...                            xbound=[1, 3, 6],
...                            ybound=[1, 3, 6],
...                            marker=2,
...                            isSubSurface=False)
>>> ax,_ = pg.show(g1, markers=True, showMesh=True)
>>> grid = mt.createGrid(5,5,5)
...
>>> g2 = mt.appendBoundaryGrid(grid,
...                            xbound=[1, 3, 6],
...                            ybound=[1, 3, 6],
...                            zbound=[1, 3, 6],
...                            marker=2,
...                            isSubSurface=False)
>>> ax, _ = pg.show(g2, g2.cellMarkers(), showMesh=True,
...                 filter={'clip':{}});
```

pygimli.meshtools.**appendTetrahedronBoundary**(*mesh*, *xbound=10*, *ybound=10*, *zbound=10*, *marker=1*, *isSubSurface=True*, ***kwargs*)



Return a copy of mesh surrounded by a tetrahedron mesh as boundary.

Returns a new mesh that contains a tetrahedron mesh box around a given mesh suitable for geo-simulation (surface boundary with marker = -1 at top and marker = -2 in the inner subsurface). The old boundary marker from mesh will be preserved, except for marker == -2 which will be switched to 2 as we assume -2 is the world marker for outer boundaries in the subsurface.

Note: This method will only work stable if the mesh generator (Tetgen) preserves all input boundaries. This will lead to bad quality meshes for the boundary region so its a good idea to play with the addNodes keword argument to manually refine the newly created outer boundaries.

If the input mesh consists of hexahedrons a small inconsistency will arise because a quad boundary element will be split by 2 triangle boundaries from the boundary tetrahedrons. The effect of this hanging edges are unclear, also createNeighbourInfos may fail. We need to implement/test pyramid cells to handle this.

Parameters

- **mesh** (GIMLI::Mesh) – 3D Mesh to which the tetrahedron boundary should be appended.
- **xbound** (*float [10]*) – Horizontal prolongation distance in meter at x-direction. Need to be >= 0.
- **ybound** (*float [10]*) – Horizonal prolongation distance in meter at y-direction. Need to be greater 0.

- **zbound** (*float [10]*) – Vertical prolongation distance in meter at z-direction (>0).
- **marker** (*int, optional*) – Marker of new cells.
- **addNodes** (*float, optional*) – Triangle quality.
- **isSubSurface** (*boolean, optional*) – Apply boundary conditions suitable for geo-simulaion and prolongate mesh to the surface if necessary.
- **verbose** (*boolean, optional*) – Be verbose.

Returns
A new 3D mesh containing the original mesh and a boundary arround.

Return type
GIMLI::Mesh

See also:

pygimli.meshtools.appendBoundary (page 312), *pygimli.meshtools.appendTriangleBoundary* (page 315)

```
>>> import pygimli as pg
>>> import pygimli.meshtools as mt
>>> grid = mt.createGrid(5,5,5)
...
>>> mesh = mt.appendBoundary(grid, xbound=5, ybound=5, zbound=5,
...                          isSubSurface=False)
>>> ax, _ = pg.show(mesh, mesh.cellMarkers(), showMesh=True,
...                 filter={'clip':{}})
```

pygimli.meshtools.**appendTriangleBoundary**(*mesh*, *xbound=10*, *ybound=10*, *marker=1*, *isSubSurface=True*, ***kwargs*)

Add a triangle mesh boundary to a given mesh.

Returns a new mesh that contains a triangulated box around a given mesh suitable for geo-simulation (surface boundary with marker = -1 at top and marker = -2 in the inner subsurface). The old boundary marker from mesh will be preserved, except for marker == -2 which will be switched to 2 as we assume -2 is the world marker for outer boundaries in the subsurface.

Note that this all will only work stable if the mesh generator (triangle) preserve all input boundaries. This will lead to bad quality meshes for the boundary region so its a good idea to play with the addNodes keyword argument to manually refine the newly created outer boundaries.

Parameters

- **mesh** (GIMLI::Mesh) – Mesh to which the triangle boundary should be appended.
- **xbound** (*float, optional*) – Absolute horizontal prolongation distance.
- **ybound** (*float, optional*) – Absolute vertical prolongation distance.
- **marker** (*int, optional*) – Marker of new cells.
- **isSubSurface** (*boolean [True]*) – Apply boundary conditions suitable for geo-simulation and prolongate mesh to the surface if necessary.

- **Args** (*Additional*) –
- **----------** –
- **pg.createMesh** (*** kargs forwarded to*) –
- **quality** (*float, optional*) – Triangle quality.
- **area** (*float, optional*) – Triangle max size within the boundary.
- **smooth** (*boolean, optional*) – Apply mesh smoothing.
- **addNodes** (*int[5], iterable*) – Add additional nodes on the outer boundaries. Or for each boundary if given 5 values (isSubsurface=True) or 4 for isSubsurface=False

Returns

A new 2D mesh containing the original mesh and a boundary arround.

Return type

GIMLI::Mesh

See also:

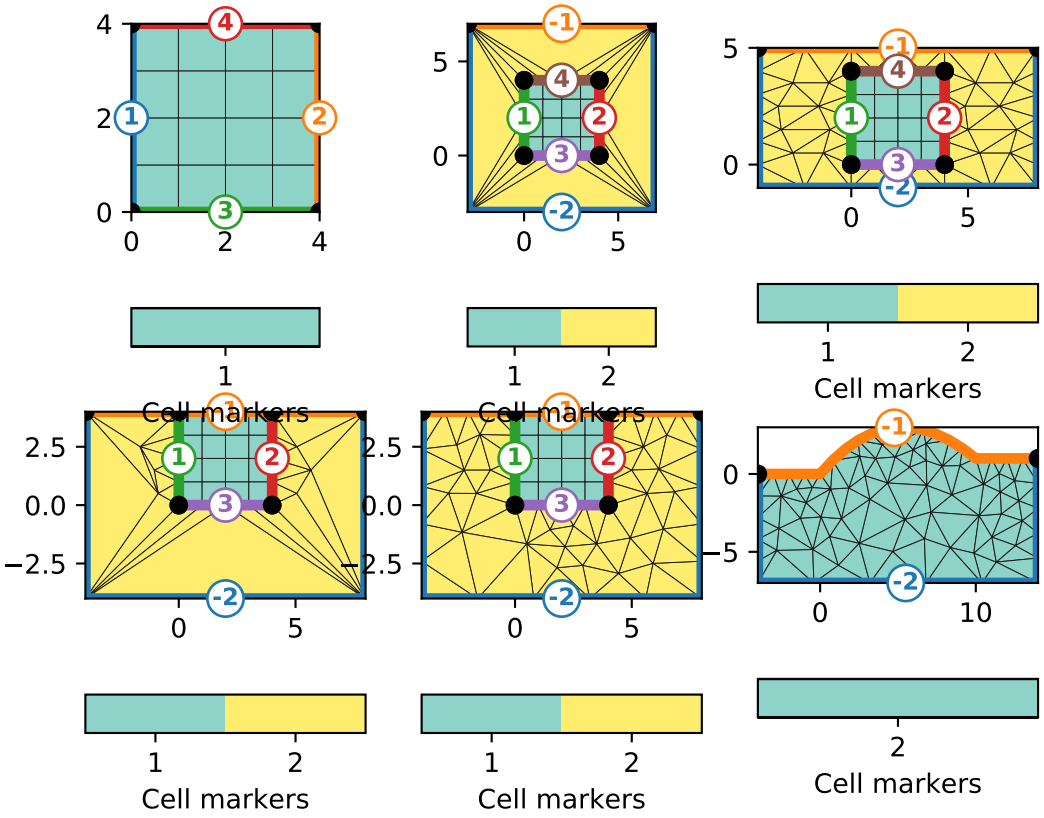
pygimli.meshtools.appendBoundary (page 312), *pygimli.meshtools.appendTetrahedronBoundary* (page 313)

```
>>> import matplotlib.pyplot as plt
>>> import pygimli as pg
>>> from pygimli.viewer.mpl import drawMesh, drawModel
>>> import pygimli.meshtools as mt
>>> inner = pg.createGrid(range(5), range(5), marker=1)
>>> fig, axs = plt.subplots(2,3)
>>> ax, _ = pg.show(inner, markers=True, showBoundaries=True,␣
↪showMesh=True, ax=axs[0][0])
>>> m = mt.appendTriangleBoundary(inner, xbound=3, ybound=3, marker=2,␣
↪addNodes=0, isSubSurface=False)
>>> ax, _ = pg.show(m, markers=True, showBoundaries=True, showMesh=True,␣
↪ax=axs[0][1])
>>> m = mt.appendTriangleBoundary(inner, xbound=4, ybound=1, marker=2,␣
↪addNodes=5, isSubSurface=False)
>>> ax, _ = pg.show(m, markers=True, showBoundaries=True, showMesh=True,␣
↪ax=axs[0][2])
>>> m = mt.appendTriangleBoundary(inner, xbound=4, ybound=4, marker=2,␣
↪addNodes=0, isSubSurface=True)
>>> ax, _ = pg.show(m, markers=True, showBoundaries=True, showMesh=True,␣
↪ax=axs[1][0])
>>> m = mt.appendTriangleBoundary(inner, xbound=4, ybound=4, marker=2,␣
↪addNodes=5, isSubSurface=True)
>>> ax, _ = pg.show(m, markers=True, showBoundaries=True, showMesh=True,␣
↪ax=axs[1][1])
>>> surf = mt.createPolygon([[0, 0],[5, 3], [10, 1]], boundaryMarker=-1,␣
↪addNodes=5, interpolate='spline')
>>> m = mt.appendTriangleBoundary(surf, xbound=4, ybound=4, marker=2,␣
↪addNodes=5, isSubSurface=True)
```

(continues on next page)

(continued from previous page)

```
>>> ax, _ = pg.show(m, markers=True, showBoundaries=True, showMesh=True,␣
↪ax=axs[1][2])
```



Examples using `pygimli.meshtools.appendTriangleBoundary`

- *Building a hybrid mesh in 2D* (page 33)
- *2D ERT modeling and inversion* (page 60)

`pygimli.meshtools.`**`cellDataToBoundaryData`**(*mesh*, *data*)

TODO DOCUMENT_ME

`pygimli.meshtools.`**`cellDataToNodeData`**(*mesh*, *data*, *style='mean'*)

Convert cell data to node data.

Convert cell data to node data via non-weighted averaging (mean) of common cell data.

Parameters

- **mesh** (GIMLI::Mesh) – 2D or 3D GIMLi mesh
- **data** (*iterable [float]*) – Data of len mesh.cellCount(). TODO complex, R3Vector, ndarray
- **style** (*str ['mean']*) – Interpolation style. * ‘mean’ : non-weighted averaging TODO harmonic averaging TODO weighted averaging (mean, harmonic) TODO interpolation via cell centered mesh

Examples

```
>>> import pygimli as pg
>>> grid = pg.createGrid(x=(1,2,3),y=(1,2,3))
>>> celldata = np.array([1, 2, 3, 4])
>>> nodedata = pg.meshtools.cellDataToNodeData(grid, celldata)
>>> print(nodedata.array())
[1.  1.5 2.  2.  2.5 3.  3.  3.5 4. ]
```

Examples using `pygimli.meshtools.cellDataToNodeData`

- *Hydrogeophysical modeling* (page 138)

`pygimli.meshtools.`**`convert`**(*mesh*, *verbose=False*)

Convert mesh from foreign formats.

`pygimli.meshtools.`**`convertHDF5Mesh`**(*h5Mesh*, *group='mesh'*, *indices='cell_indices'*, *pos='coordinates'*, *cells='topology'*, *marker='values'*, *marker_default=0*, *dimension=3*, *verbose=True*, *useFenicsIndices=False*)

Converts instance of a hdf5 mesh to a GIMLI::Mesh.

For full documentation please see `pygimli:meshtools:readHDF5Mesh`.

`pygimli.meshtools.`**`convertMeshioMesh`**(*mesh*, *verbose=False*)

Convert mesh from meshio object.

See https://pypi.org/project/meshio/1.8.9/

TODO

- test for 3D mesh
- test and improve if neeeded

`pygimli.meshtools.`**`createCircle`**(*pos=None*, *radius=1*, *nSegments=12*, *start=0*, *end=6.283185307179586*, ***kwargs*)

Create simple circle polygon.

Create simple circle polygon with given attributes.

Parameters

- **`pos`** (*`[x, y] [[0.0, 0.0]]`*) – Center position
- **`radius`** (*`float | [a,b] [1]`*) – radius or halfaxes of the circle
- **`nSegments`** (*`int [12]`*) – Discrete amount of segments for the circle.
- **`start`** (*`double [0]`*) – Starting angle in radians
- **`end`** (*`double [2*pi]`*) – Ending angle in radians
- **`**kwargs`** –

  marker: int [1]
  Marker for the resulting triangle cells after mesh generation

markerPosition
[floats [x, y] [0.0, 0.0]] Position of the marker (works for both regions and holes)

area: float [0]
Maximum cell size for resulting triangles after mesh generation

isHole: bool [False]
The polygon will become a hole instead of a triangulation

boundaryMarker: int [1]
Marker for the resulting boundary edges

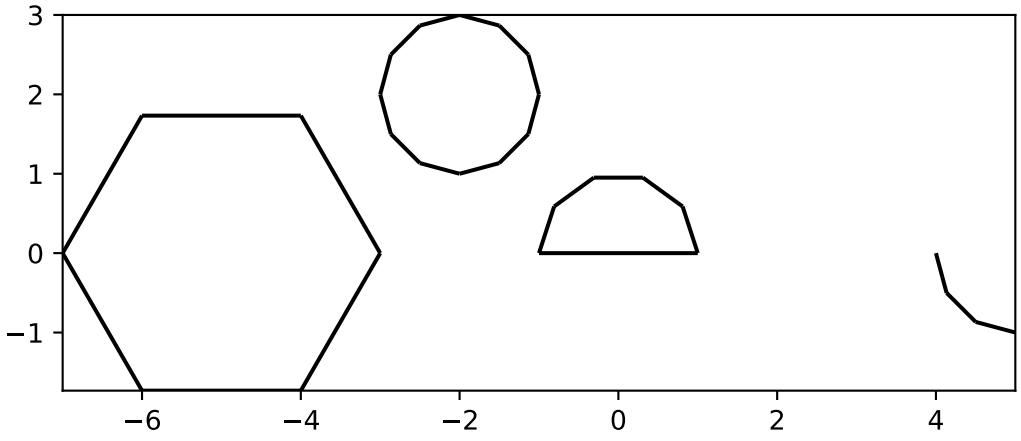
leftDirection: bool [True]
Rotational direction

isClosed: bool [True]
Add closing edge between last and first node.

Returns
poly – The resulting polygon is a GIMLI::Mesh.

Return type
GIMLI::Mesh

```
>>>  # no need to import matplotlib. pygimli's show does
>>> import math
>>> import pygimli as pg
>>> from pygimli.viewer.mpl import drawMesh
>>> import pygimli.meshtools as mt
>>> c0 = mt.createCircle(pos=(-5.0, 0.0), radius=2, nSegments=6)
>>> c1 = mt.createCircle(pos=(-2.0, 2.0), radius=1, area=0.01, marker=2)
>>> c2 = mt.createCircle(pos=(0.0, 0.0), nSegments=5, start=0, end=math.pi)
>>> c3 = mt.createCircle(pos=(5.0, 0.0), nSegments=3, start=math.pi,
...                      end=1.5*math.pi, isClosed=False)
>>> plc = mt.mergePLC([c0, c1, c2, c3])
>>> fig, ax = pg.plt.subplots()
>>> drawMesh(ax, plc, fillRegion=False)
>>> pg.wait()
```



Examples using `pygimli.meshtools.createCircle`

- *Extrude a 2D mesh to 3D* (page 25)
- *Crosshole traveltime tomography* (page 40)
- *2D ERT modeling and inversion* (page 60)
- *Complex-valued electrical modeling* (page 98)
- *Naive complex-valued electrical inversion* (page 104)
- *Gravimetry in 2D - Part I* (page 116)
- *Semianalytical Gravimetry and Geomagnetics in 2D* (page 118)
- *Petrophysical joint inversion* (page 153)
- *Quality of unstructured meshes* (page 202)
- *Region markers* (page 206)

pygimli.meshtools.**createCube**(*size=[1.0, 1.0, 1.0]*, *pos=None*, *start=None*, *end=None*, *rot=None*, *boundaryMarker=0*, ***kwargs*)

Create cube PLC as geometrie definition.

Create cube PLC as geometrie definition. You can either give size and center position or start and end position.

Parameters

- **size** (*[x, y, z]*) – x, y, and z-size of the cube. Default = [1.0, 1.0, 1.0] in m
- **pos** (*[x, y, z]*) – The center position, default is at the origin.
- **start** (*[x, y, z]*) – Left Front Bottom corner.
- **end** (*[x, y, z]*) – Right Back Top corner.
- **rot** (*pg.Pos [None]*) – Rotate on the center.
- **boundaryMarker** (*int [0]*) – Boundary marker for the resulting faces.
- **kwargs** (****) – Marker related arguments: See `pygimli.meshtools.polytools.setPolyRegionMarker`

Examples

```
>>> import pygimli.meshtools as mt
>>> cube = mt.createCube()
>>> print(cube)
Mesh: Nodes: 8 Cells: 0 Boundaries: 6
>>> cube = mt.createCube([10, 10, 1])
>>> print(cube.bb())
[RVector3: (-5.0, -5.0, -0.5), RVector3: (5.0, 5.0, 0.5)]
>>> cube = mt.createCube([10, 10, 1], pos=[-4.0, 0.0, 0.0])
>>> print(pg.center(cube.positions()))
RVector3: (-4.0, 0.0, 0.0)
```

Returns
poly – The resulting polygon is a GIMLI::Mesh.

Return type
GIMLI::Mesh

Examples using `pygimli.meshtools.createCube`

- *Refraction in 3D* (page 57)
- *3D modeling in a closed geometry* (page 85)
- *3D Darcy flow* (page 136)

`pygimli.meshtools.`**`createCylinder`**(*radius=1*, *height=1*, *nSegments=8*, *pos=None*, *rot=None*, *boundaryMarker=0*, ***kwargs*)

Create PLC of a cylinder.

Out of core wrapper for dcfemlib::polytools.

Note, there is a bug in the old polytools which ignores the area settings for marker == 0.

Parameters

- **`radius`** (*`float`*) – Radius of the cylinder.
- **`height`** (*`float`*) – Height of the cylinder
- **`nSegments`** (*`int [8]`*) – Number of segments of the cylinder.
- **`pos`** (*`pg.Pos [None]`*) – The center position, default is at the origin.

Keyword Arguments
`kwargs` (****) – Marker related arguments: See `pygimli.meshtools.polytools.setPolyRegionMarker`

Returns
poly – The resulting polygon is a GIMLI::Mesh.

Return type
GIMLI::Mesh

`pygimli.meshtools.`**`createFacet`**(*mesh*, *boundaryMarker=None*, *verbose=True*)

Create a coplanar PLC of a 2d mesh or poly

TODO: * mesh with cell into plc with boundaries * poly account for inner edges

`pygimli.meshtools.`**`createGrid`**(*x=None*, *y=None*, *z=None*, ***kwargs*)

Create grid style mesh.

Generate simple grid with defined node positions for each dimension. The resulting grid depends on the amount of given coordinate arguments and consists out of edges (1D - x), quads (2D- x and y), or hexahedrons(3D- x, y, and z).

Parameters
`kwargs` –

x: array
x-coordinates for all Nodes (1D, 2D, 3D)

y: array
y-coordinates for all Nodes (2D, 3D)

z: array
z-coordinates for all Nodes (3D)

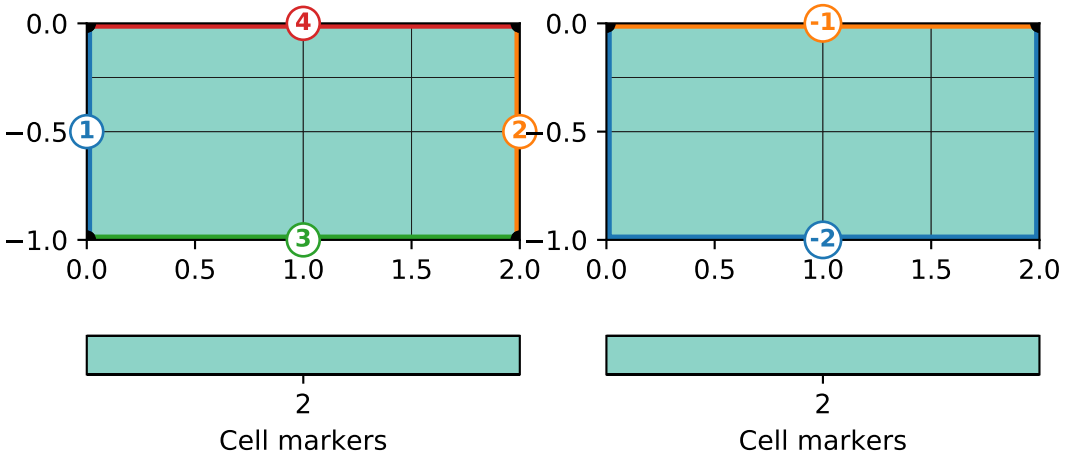
marker: int = 0
Marker for resulting cells.

worldBoundaryMarker
[bool = False] Boundaries are enumerated with world marker, i.e., Top = -1 All remaining = -2. Default markers: left=1, right=2, top=3, bottom=4, front=5, back=6

Returns
Either 1D, 2D or 3D mesh depending the input.

Return type
GIMLI::Mesh

```
>>> import pygimli as pg
>>> mesh = pg.meshtools.createGrid(x=[0, 1, 1.5, 2], y=[-1, -.5, -.25, 0],
...                                marker=2)
>>> print(mesh)
Mesh: Nodes: 16 Cells: 9 Boundaries: 24
>>> fig, axs = pg.plt.subplots(1, 2)
>>> _ = pg.show(mesh, markers=True, showMesh=True, ax=axs[0])
>>> mesh = pg.meshtools.createGrid(x=[0, 1, 1.5, 2], y=[-1, -.5, -0.25, 0],
...                                worldBoundaryMarker=True, marker=2)
>>> print(mesh)
Mesh: Nodes: 16 Cells: 9 Boundaries: 24
>>> _ = pg.show(mesh, markers=True, showBoundaries=True,
...             showMesh=True, ax=axs[1])
```



pygimli.meshtools.**createGridPieShaped**(*x*, *degree=10.0*, *h=2*, *marker=0*)

Create a 2D pie shaped grid (segment from annulus or cirlce).

8.3.2.1 TODO:

- degree: > 90 .. 360

param x
x-coordinates for all Nodes (2D). If you need it 3D, you can apply *pygimli.meshtools.extrudeMesh* (page 343) on it.

type x
array

param degree
Create a pie shaped grid for a value between 0 and 90. Creates an optional inner boundary (marker=2) for a annulus with x[0] > 0. Outer boundary marker is 1. Optional h refinement. Center node is the first for circle segment.

type degree
float [None]

param h
H-Refinement for degree option.

type h
int [2]

param marker
Marker for resulting cells.

type marker
int = 0

returns
mesh

rtype
GIMLI::Mesh

```
>>> import pygimli as pg
>>> mesh = pg.meshtools.createGridPieShaped(x=[0, 1, 3], degree=45, h=3)
>>> print(mesh)
Mesh: Nodes: 117 Cells: 128 Boundaries: 244
>>> _ = pg.show(mesh)
>>> mesh = pg.meshtools.createGridPieShaped(x=[1, 2, 3], degree=45, h=3)
>>> print(mesh)
Mesh: Nodes: 153 Cells: 128 Boundaries: 280
>>> _ = pg.show(mesh)
```

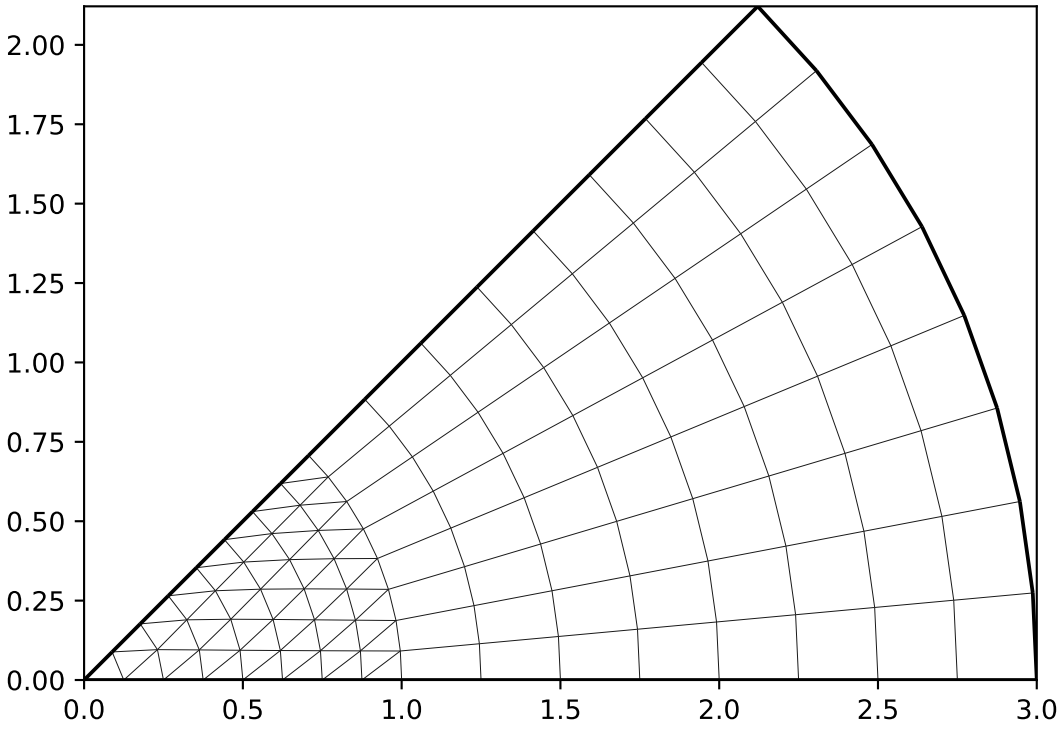
pygimli.meshtools.**createLine**(*start*, *end*, *nSegments=1*, ***kwargs*)

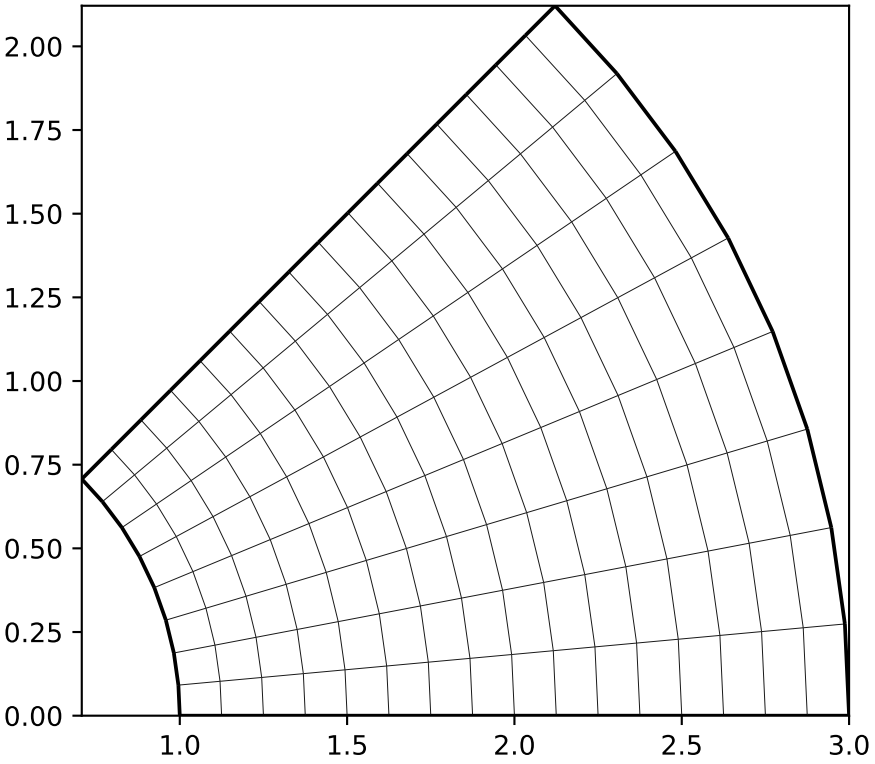
Create simple line polygon.

Create simple line polygon from start to end.

Parameters

- **start** (*[x, y]*) – start position
- **end** (*[x, y]*) – end position


2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0.0
0.5
1.0
1.5
2.0
2.5
3.0


2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
1.0
1.5
2.0
2.5
3.0

- **nSegments** (*int*) – Discrete amount of segments for the line

Keyword Arguments

- **boundaryMarker** (*int [1]*) – Marker for the resulting boundary edges
- **leftDirection** (*bool [True]*) – Rotational direction

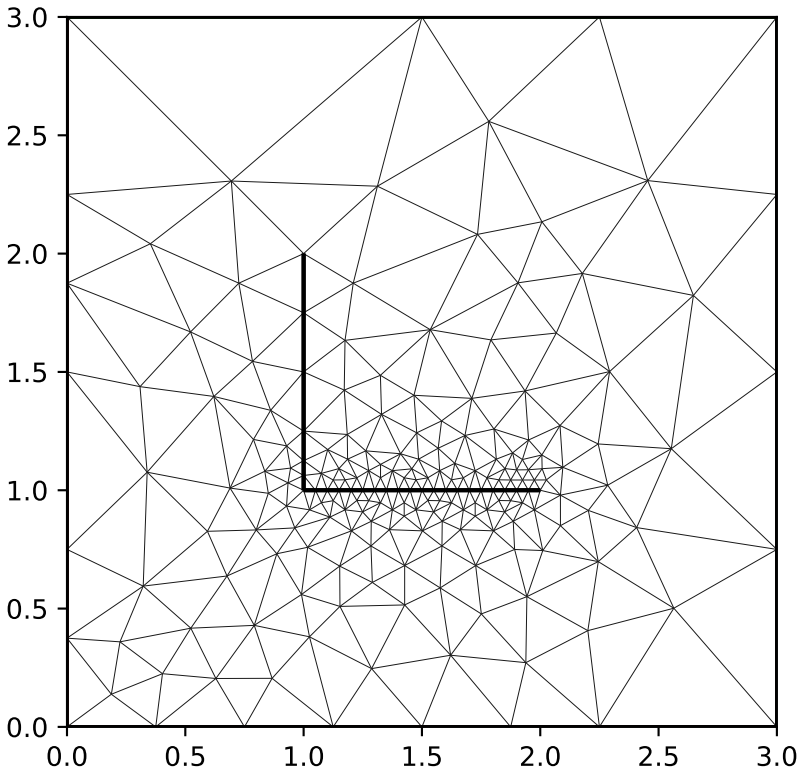
Returns

poly – The resulting polygon is a GIMLI::Mesh.

Return type

GIMLI::Mesh

```
>>>  # no need to import matplotlib. pygimli's show does
>>> import pygimli as pg
>>> import pygimli.meshtools as mt
>>>
>>> w = mt.createWorld(start=[0, 0], end=[3, 3])
>>> l1 = mt.createLine(start=[1, 1], end=[1, 2], nSegments=1,
...                    leftDirection=False)
>>> l2 = mt.createLine(start=[1, 1], end=[2, 1], nSegments=20,
...                    leftDirection=True)
>>>
>>> ax, _ = pg.show(mt.createMesh([w, l1, l2,]))
>>> ax, _ = pg.show([w, l1, l2,], ax=ax, fillRegion=False)
>>> pg.wait()
```



pygimli.meshtools.**createMesh**(*poly*, *quality=32*, *area=0.0*, *smooth=None*, *switches=None*, *verbose=False*, ***kwargs*)

Create a mesh for a given PLC or point list.

The mesh is created by *triangle* or *tetgen* if the pgGIMLi support for these mesh generators is installed. A PLC needs to contain nodes and boundaries and should be valid in the sense that the boundaries are non-intersecting.

If poly is a list of coordinates, a simple Delaunay mesh with a convex hull will be created. Quality and area arguments are ignored for this case to create a mesh with one node for each coordinate position.

Parameters

- **poly** (GIMLI::Mesh or list or ndarray) –
  - 2D or 3D gimli mesh that contains the PLC.
  - 2D mesh needs edges
  - 3D mesh needs a plc and tetgen as system component
  - List of x y pairs [[x0, y0], . . . ,[xN, yN]]
  - ndarray [x_i, y_i]
  - PLC or list of PLCs
- **quality** (*float*) – 2D triangle quality sets a minimum angle constraint. Be careful with values above 34 degrees. 3D tetgen quality. Be careful with values below 1.12.
- **area** (*float*) – Maximum element size (global). 2D maximum triangle size in m^{*2}, 3D maximum tetrahedral size in m^3.
- **smooth** (*tuple*) – [smoothing algorithm, number of iterations] 0: no smoothing 1: node center 2: weighted node center

  If smooth is just set to True then [1, 4] is choosen.
- **switches** (*str*) – Set additional triangle command switches. https://www.cs.cmu.edu/~quake/triangle.switch.html
- **Args** (*Additional*) –
- **----------** –
- **preserveBoundary** (*bool*) – Preserver boundary nodes, no more nodes on boundaries.

Returns

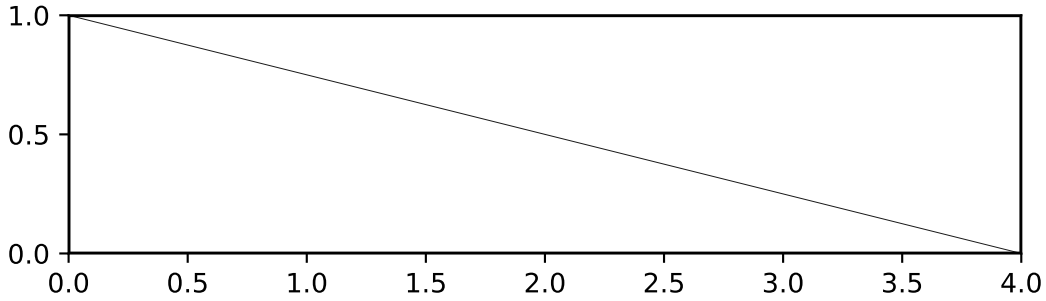
mesh

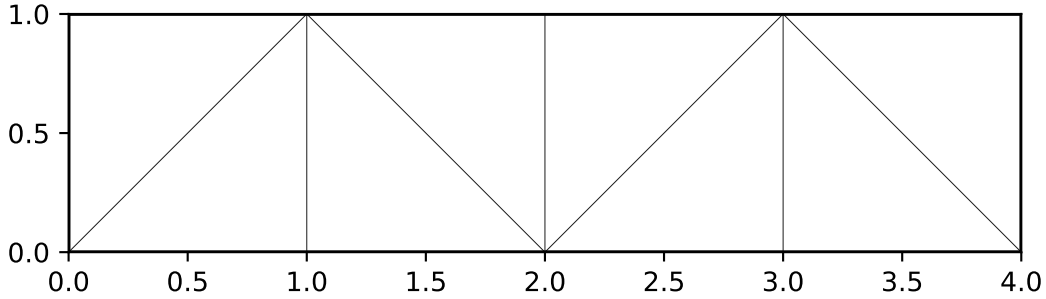
Return type

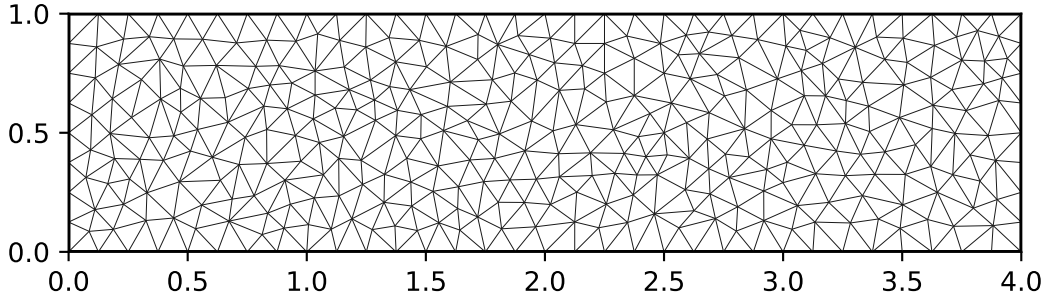
GIMLI::Mesh

```
>>> import pygimli as pg
>>> import pygimli.meshtools as mt
>>> rect = mt.createRectangle(start=[0, 0], end=[4, 1])
>>> ax, _ = pg.show(mt.createMesh(rect, quality=10))
>>> ax, _ = pg.show(mt.createMesh(rect, quality=33))
>>> ax, _ = pg.show(mt.createMesh(rect, quality=33, area=0.01))
>>> pg.wait()
```

Examples using `pygimli.meshtools.createMesh`


1.0
0.5
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0


1.0
0.5
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0


1.0
0.5
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

- *Meshing the Omega aka. BERT logo* (page 17)
- *Extrude a 2D mesh to 3D* (page 25)
- *Building a hybrid mesh in 2D* (page 33)
- *2D Refraction modeling and inversion* (page 34)
- *Crosshole traveltime tomography* (page 40)
- *Raypaths in layered and gradient models* (page 46)
- *Refraction in 3D* (page 57)
- *2D ERT modeling and inversion* (page 60)
- *2D FEM modelling on two-layer example* (page 73)
- *Four-point sensitivities* (page 81)
- *3D modeling in a closed geometry* (page 85)
- *Complex-valued electrical modeling* (page 98)
- *Naive complex-valued electrical inversion* (page 104)
- *Gravimetry in 2D - Part I* (page 116)
- *3D Darcy flow* (page 136)
- *Hydrogeophysical modeling* (page 138)
- *Petrophysical joint inversion* (page 153)
- *Quality of unstructured meshes* (page 202)
- *Heat equation in 2D* (page 223)
- *Regularization - concepts explained* (page 236)
- *Geostatistical regularization* (page 254)
- *Region-wise regularization* (page 262)

pygimli.meshtools.**createMesh1D**(*(object)x*) → object :

Generate simple one dimensional mesh with nodes at position in RVector pos.

C++ signature :
GIMLI::Mesh createMesh1D(GIMLI::Vector<double>)

createMesh1D((object)nCells [, (object)nProperties=1]) -> object :
Generate simple 1D mesh with nCells cells of length 1, and nCells + 1 nodes. In case of more than one property quasi-2d mesh with regions is generated.

C++ signature :
GIMLI::Mesh createMesh1D(unsigned long [,unsigned long=1])

pygimli.meshtools.**createMesh1DBlock**(*(object)nLayers*[, *(object)nProperties=1*]) → object :

Generate 1D block model of thicknesses and properties

C++ signature :
GIMLI::Mesh createMesh1DBlock(unsigned long [,unsigned long=1])

Examples using `pygimli.meshtools.createMesh1DBlock`

- *DC-EM Joint inversion* (page 149)

`pygimli.meshtools.`**`createMesh2D`**(*(object)x*, *(object)y*[, *(object)markerType=0*]) → object :

Generate simple two dimensional mesh with nodes at position in RVector x and y.

C++ signature :
GIMLI::Mesh createMesh2D(GIMLI::Vector<double>,GIMLI::Vector<double> [,int=0])

createMesh2D((object)mesh, (object)y [, (object)frontMarker=0 [, (object)backMarker=0 [, (object)leftMarker=0 [, (object)rightMarker=0 [, (object)adjustBack=False]]]]]) -> object :

Generate a simple 2D mesh by extruding a 1D polygone into RVector y using quads. We assume a 2D mesh here consisting on nodes and edge boundaries. Nodes with marker are extruded as edges with marker or set to front- and backMarker. Edges with marker are extruded as cells with marker. All back y-coordinates are adjusted if adjustBack is set.

C++ signature :
GIMLI::Mesh createMesh2D(GIMLI::Mesh,GIMLI::Vector<double> [,int=0 [,int=0 [,int=0 [,int=0 [,bool=False]]]]])

createMesh2D((object)xDim, (object)yDim [, (object)markerType=0]) -> object :

Generate simple two dimensional mesh with nRows x nCols cells with each length = 1.0

C++ signature :
GIMLI::Mesh createMesh2D(unsigned long,unsigned long [,int=0])

Examples using `pygimli.meshtools.createMesh2D`

- *Mesh interpolation* (page 200)

`pygimli.meshtools.`**`createMesh3D`**(*(object)x*, *(object)y*, *(object)z*[, *(object)markerType=0*]) → object :

Generate simple three dimensional mesh with nodes at position in RVector x and y.

C++ signature :
GIMLI::Mesh createMesh3D(GIMLI::Vector<double>,GIMLI::Vector<double>,GIMLI::Vector<double [,int=0])

createMesh3D((object)mesh, (object)z [, (object)topMarker=0 [, (object)bottomMarker=0]]) -> object :

Generate a simple three dimensional mesh by extruding a two dimensional mesh into RVector z using triangle prism or hexahedrons or both. 3D cell marker are set from 2D cell marker. The boundary marker for the side boundaries are set from edge marker in mesh. Top and bottomLayer boundary marker are set from parameter topMarker and bottomMarker.

C++ signature :
GIMLI::Mesh createMesh3D(GIMLI::Mesh,GIMLI::Vector<double> [,int=0 [,int=0]])

createMesh3D((object)xDim, (object)yDim, (object)zDim [, (object)markerType=0]) -> object :

Generate simple three dimensional mesh with nx x nx x nz cells with each length = 1.0

C++ signature :
GIMLI::Mesh createMesh3D(unsigned long,unsigned long,unsigned long [,int=0])

pygimli.meshtools.**createMeshFromHull**(*mesh*, *fixNodes=[]*, ***kwargs*)

Create a new 2D triangular mesh from the boundaries of mesh.

Parameters

- **mesh** (GIMLI::Mesh) – Input mesh.
- **fixNodes** (*iterable (int)*) – Nodes (IDs) from the input mesh that should be part of the new mesh.

Keyword Arguments

****kwargs** (*dict*) – Forwarded to pygimli:meshtools:createMesh.

Returns

mesh – Returning mesh. If fixed nodes are requested, a list of the new IDs are returned in advance.

Return type

GIMLI::Mesh, List of fixed nodes

pygimli.meshtools.**createParaDomain2D**(**args*, ***kwargs*)

API change here .. use createParaMeshPLC instead.

pygimli.meshtools.**createParaMesh**(*data*, ***kwargs*)

Create parameter mesh from list of sensor positions.

Create parameter mesh from list of sensor positions. Uses *:py:func:pygimli.meshtools.createParaMeshPLC* and *:py:func:pygimli.meshtools.createMesh* and forwards keyword arguments.

Parameters

data (*DataContainer*) – Data container to read sensors positions from.

Keyword Arguments

` (*Forwarded to*) –

Returns

poly

Return type

GIMLI::Mesh

Examples using pygimli.meshtools.createParaMesh

- *Hydrogeophysical modeling* (page 138)

pygimli.meshtools.**createParaMesh2DGrid**(*sensors*, *paraDX=1*, *paraDZ=1*, *paraDepth=0*, *nLayers=11*, *boundary=-1*, *paraBoundary=2*, ***kwargs*)

Create a grid-style mesh for an inversion parameter mesh.

Create a grid-style mesh for an inversion parameter mesh. Return parameter grid for a given list of sensor positions. Uses and forwards arguments to *pygimli.meshtools.appendTriangleBoundary* (page 315).

Parameters

- **sensors** (*list of RVector3 objects or data container with sensorPositions*) – Sensor positions. Must be sorted in positive x direction

- **paraDX** (*float, optional*) – Horizontal distance between sensors, relative regarding sensor distance. Value must be greater than 0 otherwise 1 is assumed.
- **paraDZ** (*float, optional*) – Vertical distance to the first depth layer, relative regarding sensor distance. Value must be greater than 0 otherwise 1 is assumed.
- **paraDepth** (*float, optional*) – Maximum depth for parametric domain, 0 (default) means 0.4 * maximum sensor range.
- **nLayers** (*int, optional [11]*) – Number of depth layers.
- **boundary** (*int, optional [-1]*) – Boundary width to be appended for domain prolongation in absolute para domain width. Values lower than 0 force the boundary to be 4 times para domain width.
- **paraBoundary** (*int, optional [2]*) – Offset to the parameter domain boundary in absolute sensor spacing.

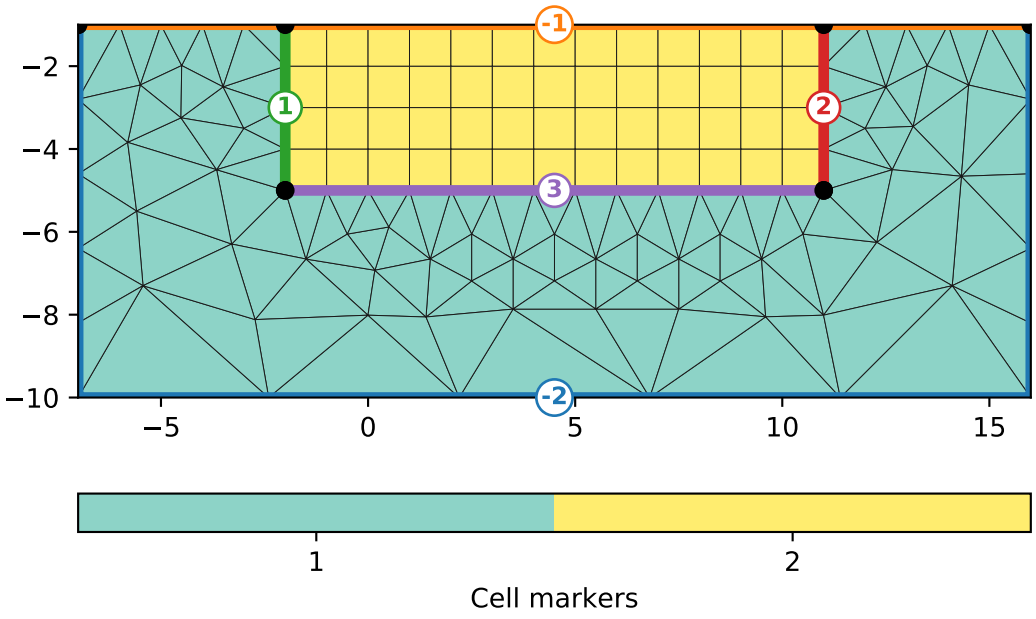
Returns

mesh

Return type

GIMLI::Mesh

```
>>> import pygimli as pg
>>> import matplotlib.pyplot as plt
>>>
>>> from pygimli.meshtools import createParaMesh2DGrid
>>> mesh = createParaMesh2DGrid(sensors=pg.Vector(range(10)),
...                             boundary=5, paraDX=1,
...                             paraDZ=1, paraDepth=5)
>>> ax, _ = pg.show(mesh, markers=True, showMesh=True)
```



pygimli.meshtools.**createParaMeshPLC**(*sensors*, *paraDX=1*, *paraDepth=-1*, *paraBoundary=2*, *paraMaxCellSize=0.0*, *boundary=-1*, *boundaryMaxCellSize=0*, *balanceDepth=True*, *isClosed=False*, *addNodes=1*, ***kwargs*)

Create a geometry (PLC) for an inversion parameter mesh.

Create an inversion mesh geometry (PLC) for a given list of sensor positions. Sensor positions are assumed to be on the surface and must be unique and sorted along x coordinate.

You can create a parameter mesh without sensors if you just set [xMin, xMax] as sensors.

The PLC is a GIMLI::Mesh and contain nodes, edges and two region markers, one for the parameters domain (marker=2) and a larger boundary around the outside (marker=1).

Parameters

- **sensors** (*[RVector3] | DataContainer with sensorPositions() | [xMin, xMax]*) – Sensor positions. Must be sorted and unique in positive x direction. Depth need to be y-coordinate.
- **paraDX** (*float [1]*) – Relative distance for refinement nodes between two sensors (1=none), e.g., 0.5 means 1 additional node between two neighboring sensors e.g., 0.33 means 2 additional equidistant nodes between two sensors
- **paraDepth** (*float[-1], optional*) – Maximum depth in m for parametric domain. Automatic (<=0) results in 0.4 * maximum sensor span range in m
- **balanceDepth** (*bool [True]*) – Equal depth for the parametric domain.
- **paraBoundary** (*float, optional*) – Margin for parameter domain in absolute sensor distances. 2 (default).
- **paraMaxCellSize** (*double, optional*) – Maximum cell size for parametric region in m^2
- **boundaryMaxCellSize** (*double, optional*) – Maximum cells size in the boundary region in m^2
- **boundary** (*float, optional*) – Boundary width to be appended for domain prolongation in absolute para domain width. Values lover 0 force the boundary to be 4 times para domain width.
- **isClosed** (*bool [False]*) – Create a closed geometry from sensor positions. Region marker is 1. Boundary marker is -1 (homogeneous Neumann)
- **addNodes** (*int [1]*) – Number of additional nodes to be added equidistant between sensors.

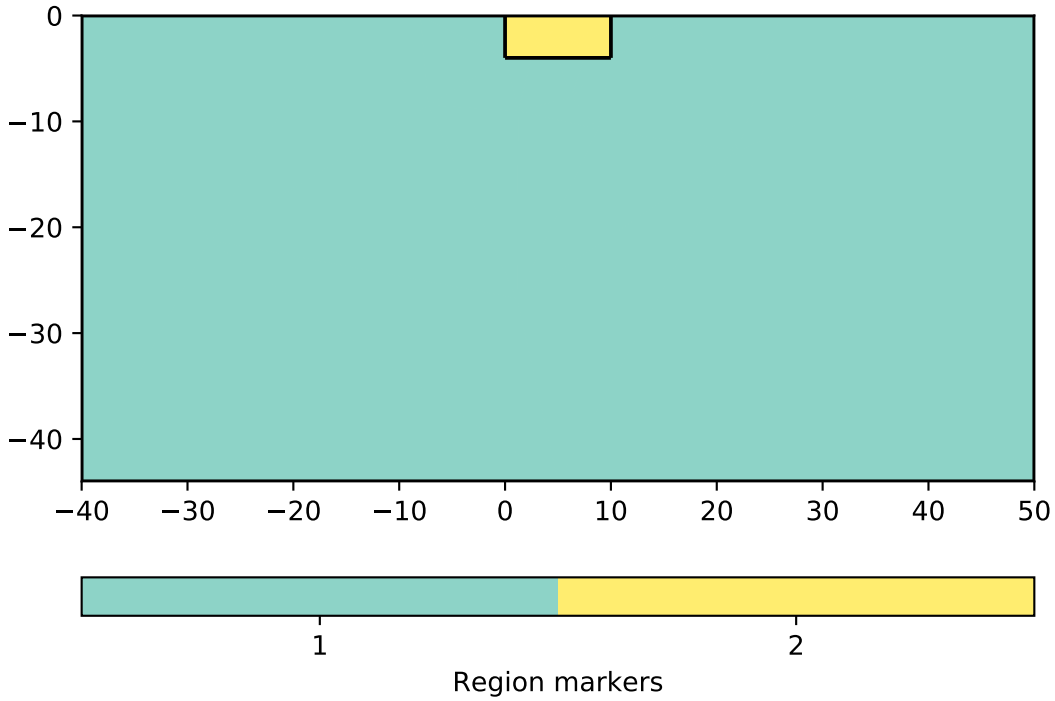
Returns

poly – Piecewise linear complex (PLC) containing nodes and edges

Return type

GIMLI::Mesh

```
>>> # no need to import matplotlib, pygimli show does.
>>> import pygimli as pg
>>> import pygimli.meshtools as mt
>>> # Create the simplest paramesh PLC with a para box of 10 m␣
↪without
>>> # sensors
>>> p = mt.createParaMeshPLC([0,10])
>>> # you can add subsurface sensors now with
>>> for z in range(1,4):
...     n = p.createNode((5,-z), -99)
>>> ax,_ = pg.show(p)
```



Examples using `pygimli.meshtools.createParaMeshPLC`

- *Region-wise regularization* (page 262)

`pygimli.meshtools.`**`createParaMeshPLC3D`**(*sensors*, *paraDX=0*, *paraDepth=-1*, *paraBoundary=None*, *paraMaxCellSize=0.0*, *boundary=None*, *boundaryMaxCellSize=0*, *surfaceMeshQuality=30*, *surfaceMeshArea=0*, *addTopo=None*, *isClosed=False*, ***kwargs*)

Create a geometry (PLC) for an 3D inversion parameter mesh.

Parameters

- **sensors** (*Sensor list or pg.DataContainer with .sensors()*) – Sensor positions.
- **paraDX** (*float [1]*) – Absolute distance for node refinement (0=none). Refinement node will be placed below the surface.
- **paraDepth** (*float[-1], optional*) – Maximum depth in m for para-

metric domain. Automatic (<=0) results in 0.4 * maximum sensor span range in m. Depth is set to median sensors depth + paraDepth.

- **paraBoundary** (*[float, float] [1.1, 1.1]*) – Margin for parameter domain in relative extend.
- **paraMaxCellSize** (*double, optional*) – Maximum cell size for parametric region in m^3
- **boundaryMaxCellSize** (*double, optional*) – Maximum cells size in the boundary region in m^3
- **boundary** (*[float, float] [10., 10.]*) – Boundary width to be appended for domain prolongation in relative para domain size.
- **surfaceMeshQuality** (*float [30]*) – Quality of the surface mesh.
- **surfaceMeshArea** (*float [0]*) – Max boundary size for surface area in parametric region.
- **addTopo** (*[[x,y,z],]*) – Number of additional nodes for topography.

Returns
poly – Piecewise linear complex (PLC) containing nodes and edges

Return type
GIMLI::Mesh

pygimli.meshtools.**createParaMeshSurface**(*sensors*, *paraBoundary=None*, *boundary=-1*, *surfaceMeshQuality=30*, *surfaceMeshArea=0*, *addTopo=None*)

Create surface mesh for an 3D inversion parameter mesh.

Topographic information (non-zero z-coodinate) can be from sensors together with addTopo, or in addTopo alone if provided. Outside boundary corners are set to median of all topography.

Parameters

- **sensors** (*DataContainer with sensorPositions()*) – Sensor positions.
- **paraBoundary** (*[float, float] [1.1, 1.1]*) – Margin for parameter domain in relative extend.
- **boundary** (*[float, float] [10., 10.]*) – Boundary width to be appended for domain prolongation in relative para domain size.
- **surfaceMeshQuality** (*float [30]*) – Quality of the surface mesh.
- **surfaceMeshArea** (*float [0]*) – Max cell Size for parametric domain.
- **addTopo** (*[[x,y,z],]*) – Number of additional nodes for topography.

Returns
surface – 3D Surface mesh

Return type
GIMLI::Mesh

```
>>> # no need to import matplotlib, pygimli show does.
>>> import numpy as np
>>> import pygimli as pg
>>> import pygimli.meshtools as mt
>>> # very simple design: 10 sensors on 1D profile in 3D␣
↪topography
>>> x = np.linspace(-10, 10, 10)
>>> topo = [[15, -15, 10], [-15, 15, -10]]
>>> surface = mt.createParaMeshSurface(np.asarray([x, x, x*0]).T,
...                                    paraBoundary=[1.2, 1.2],
...                                    boundary=[2, 2],
...                                    surfaceMeshQuality=30,
...                                    addTopo=topo)
>>> _ = pg.show(surface, showMesh=True, color='white')
```

pygimli.meshtools.**createPolygon**(*verts*, *isClosed=False*, *addNodes=0*, *interpolate='linear'*, ***kwargs*)

Create a polygon from a list of vertices.

All vertices need to be unique and duplicate vertices will be ignored. If you want the polygon be a closed region you can set the 'isClosed' flag. Closed region can be attributed by assigning a region marker. The automatic region marker is placed in the center of all vertices.

Parameters

- **verts** (*[]*) –
  - List of x y pairs [[x0, y0], . . . ,[xN, yN]]
- **isClosed** (*bool [True]*) – Add closing edge between last and first node.
- **addNodes** (*int [1], iterable*) – Constant or (for each) Number of additional nodes to be added, equidistant between sensors.
- **interpolate** (*str ['linear']*) – Interpolation rule for addNodes. 'linear' or 'spline'. TODO 'harmfit'
- ****kwargs** –

  marker
  [int [None]] Marker for the resulting triangle cells after mesh generation.

  markerPosition
  [floats [x, y] [0.0, 0.0]] Position (absolute) of the marker (works for both regions and holes)

  area
  [float [0]] Maximum cell size for resulting triangles after mesh generation

  isHole
  [bool [False]] The polygon will become a hole instead of a triangulation

  boundaryMarker
  [int [1]] Marker for the resulting boundary edges

  leftDirection
  [bool [True]] Rotational direction

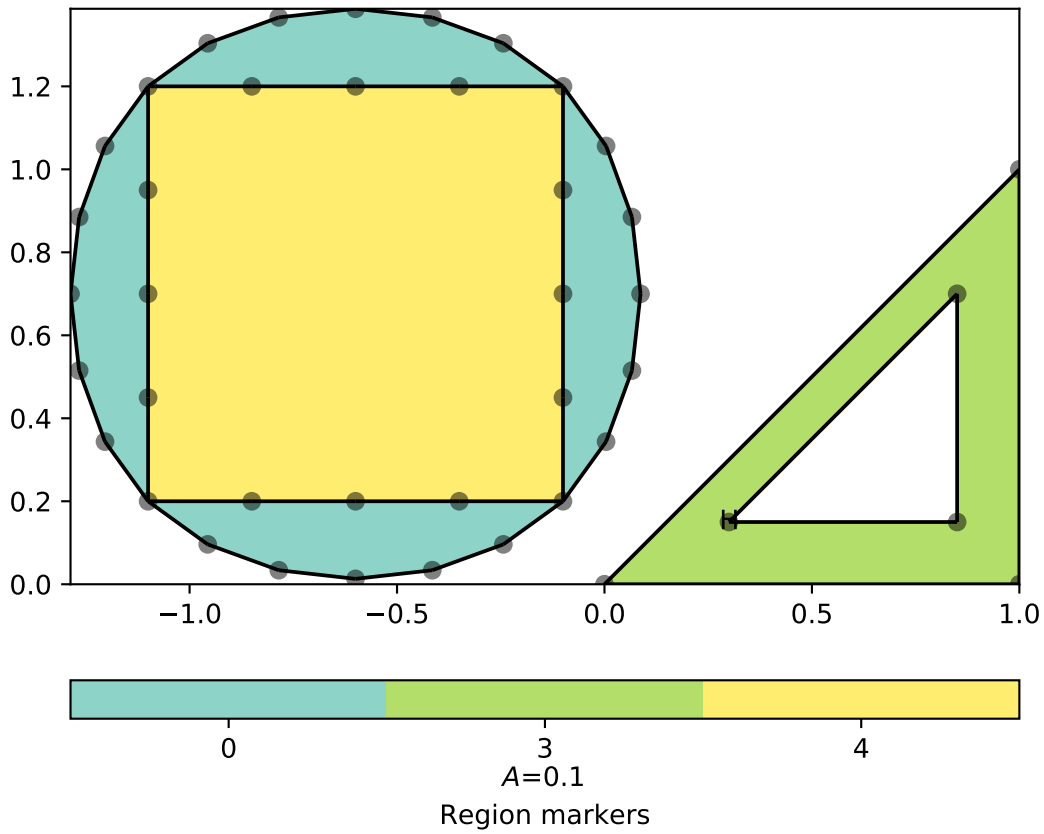
Returns

poly – The resulting polygon is a GIMLI::Mesh.

Return type

GIMLI::Mesh

```
>>> # no need to import matplotlib, pygimli show does.
>>> import pygimli as pg
>>> import pygimli.meshtools as mt
>>> p1 = mt.createPolygon([[0.0, 0.0], [1.0, 0.0], [1.0, 1.0]],
...                       isClosed=True, marker=3, area=0.1)
>>> p2 = mt.createPolygon([[0.3, 0.15], [0.85, 0.15], [0.85, 0.7]],
...                       isClosed=True, isHole=True)
>>> p3 = mt.createPolygon([[-0.1, 0.2], [-1.1, 0.2], [-1.1, 1.2], [-0.1, 1.
↪2]],
...                       isClosed=True, addNodes=3, marker=2)
>>> p4 = mt.createPolygon([[-0.1, 0.2], [-1.1, 0.2], [-1.1, 1.2], [-0.1, 1.
↪2]],
...                       isClosed=True, addNodes=5, interpolate='spline',
...                       marker=4)
>>> ax, _ = pg.show(mt.mergePLC([p1, p2, p3, p4]), showNodes=True)
>>> pg.wait()
```



Examples using `pygimli.meshtools.createPolygon`

- *2D Refraction modeling and inversion* (page 34)
- *2D ERT modeling and inversion* (page 60)

- *Petrophysical joint inversion* (page 153)
- *Region markers* (page 206)

pygimli.meshtools.**createRectangle**(*start=None*, *end=None*, *pos=None*, *size=None*, ***kwargs*)

Create rectangle polygon.

Create rectangle with start position and a given size. Give either start and end OR pos and size.

Parameters

- **start** (*[x, y]*) – Left upper corner. Default [-0.5, 0.5]
- **end** (*[x, y]*) – Right lower corner. Default [0.5, -0.5]
- **pos** (*[x, y]*) – Center position. The rectangle will be moved.
- **size** (*[x, y]*) – Factors for x and y by which the rectangle, defined by **start** and **width**, are scaled.

Keyword Arguments

****kwargs** – Additional kwargs

marker
[int [1]] Marker for the resulting triangle cells after mesh generation

markerPosition
[floats [x, y] [pos + (end - start) * 0.2]] Absolute position of the marker (works for both regions and holes).

area
[float [0]] Maximum cell size for resulting triangles after mesh generation

isHole
[bool [False]] The polygon will become a hole instead of a triangulation

boundaryMarker
[int [1]] Marker for the resulting boundary edges

leftDirection
[bool [True]] TODO Rotational direction

pnts: [[x, y],]
Return squared rectangle of origin-aligned boundingbox for pnts.

minBB: False
Return squared rectangle of non-origin-aligned minimum bounding box for pnts.

minBBOffset: [1.0, 1.0]
Offset for minimal boundingbox in x and y direction in relative extent .. whatever that means for non-aligned boxes.

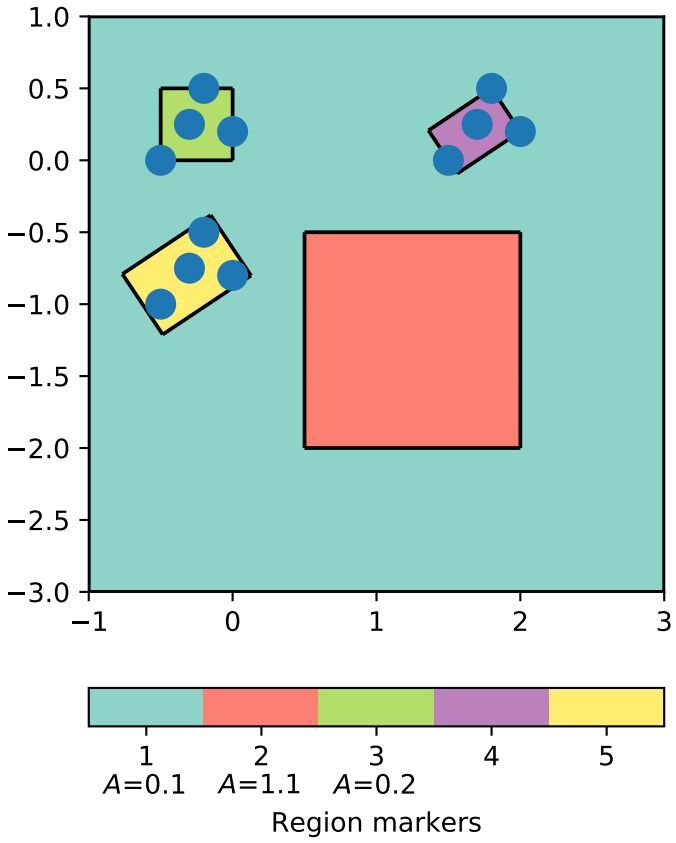
Returns

poly – The resulting polygon is a GIMLI::Mesh.

Return type

GIMLI::Mesh

```
>>> # no need to import matplotlib, pygimli show does.
>>> import pygimli as pg
>>> import pygimli.meshtools as mt
>>> r1 = mt.createRectangle(pos=[1, -1], size=[4.0, 4.0],
...                         marker=1, area=0.1, markerPosition=[0, -2])
>>> r2 = mt.createRectangle(start=[0.5, -0.5], end=[2, -2],
...                         marker=2, area=1.1)
>>> pnts3 = [[-0.5, 0], [0, 0.2], [-0.2, 0.5], [-0.3, 0.25]]
>>> r3 = mt.createRectangle(pnts=pnts3, marker=3, area=0.2)
>>> pnts4 = [[1.5, 0], [2.0, 0.2], [1.8, 0.5], [1.7, 0.25]]
>>> r4 = mt.createRectangle(pnts=pnts4, marker=4, minBB=True)
>>> pnts5 = [[-0.5, -1], [0, -0.8], [-0.2, -0.5], [-0.3, -0.75]]
>>> r5 = mt.createRectangle(pnts=pnts5, marker=5,
...                           minBB=True, minBBOffset=[1.2, 1.2])
>>> ax, _ = pg.show(mt.mergePLC([r1, r2, r3, r4, r5]))
>>> pg.viewer.mpl.drawSensors(ax, pnts3)
>>> pg.viewer.mpl.drawSensors(ax, pnts4)
>>> pg.viewer.mpl.drawSensors(ax, pnts5)
```



Examples using `pygimli.meshtools.createRectangle`

- *Crosshole traveltime tomography* (page 40)
- *Hydrogeophysical modeling* (page 138)
- *Region markers* (page 206)
- *Heat equation in 2D* (page 223)

- *Regularization - concepts explained* (page 236)
- *Geostatistical regularization* (page 254)

pygimli.meshtools.**createSurface**(*mesh*, *boundaryMarker=None*, *verbose=True*)

Convert a 2D mesh into a 3D surface mesh.

Parameters

- **mesh** (GIMLI::Mesh) – The 2D input mesh.
- **boundaryMarker** (*int [0]*) – Boundary marker for the resulting faces. If None the cell markers of the mesh are taken.

Returns

The 3D surface mesh.

Return type

GIMLI::Mesh

pygimli.meshtools.**createWorld**(*start*, *end*, *marker=1*, *area=0.0*, *layers=None*, *worldMarker=True*, ***kwargs*)

Create simple rectangular 2D or 3D world.

Create simple rectangular [hexagonal] world with appropriate boundary conditions. Surface boundary is set pg.core.MARKER_BOUND_HOMOGEN_NEUMANN, and inner subsurface is set to pg.core.MARKER_BOUND_MIXED, i.e., -2 OR Numbered: 1, 2, 3, 4, 5, 6 for left, right, bottom, top, front and back, if worldMarker is set to false and no layers are given. With layers, it is numbered in ascending order.

Parameters

- **start** (*[x, y, [z]]*) – Upper/Left/[Front] Corner
- **end** (*[x, y, [z]]*) – Lower/Right/[Back] Corner
- **marker** (*int*) – Marker for the resulting triangle cells after mesh generation.
- **area** (*float | list*) – Maximum cell size for resulting triangles after mesh generation. If area is a float set it global, if area is a list set it per layer.
- **layers** (*[float] [None]*) – List of depth coordinates for some layers.
- **worldMarker** (*bool [True]*) – Specify boundary markers: True: [-1, -2] for [surface, subsurface] boundaries False: ascending order [1, 2, 3, 4 ..]
- **createCube** (*Forwarded to*) –

Returns

poly – The resulting polygon is a GIMLI::Mesh.

Return type

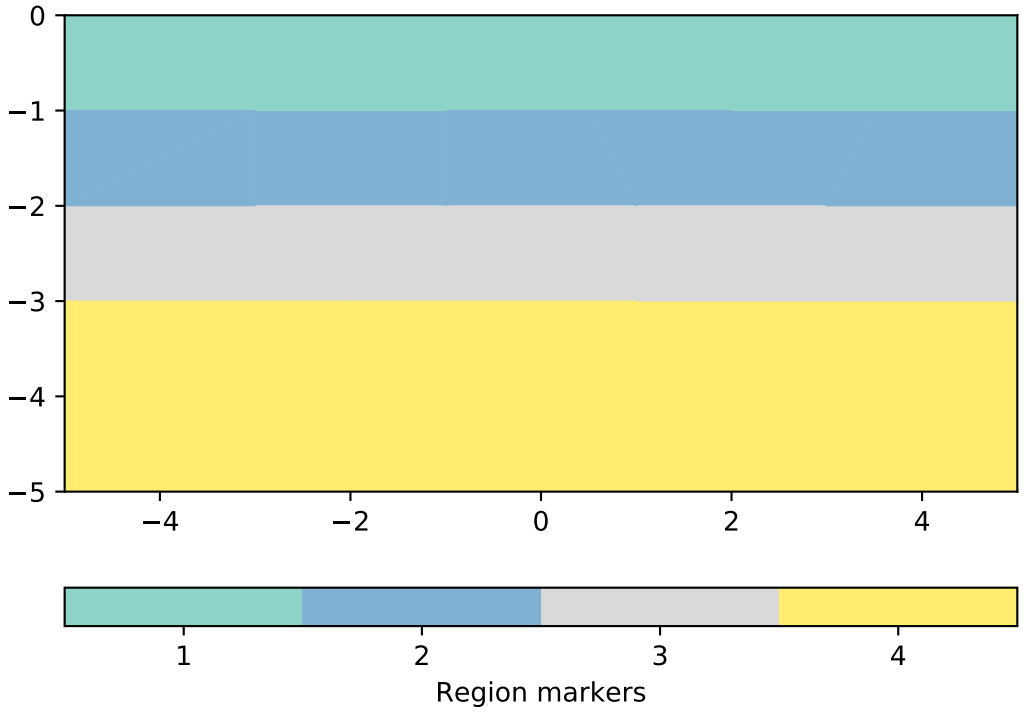
GIMLI::Mesh

```
>>> from pygimli.meshtools import createWorld
>>> from pygimli.viewer.mpl import drawMesh
>>> import matplotlib.pyplot as plt
>>> world = createWorld(start=[-5, 0], end=[5, -5], layers=[-1,-2,-3])
>>>
>>> fig, ax = plt.subplots()
```

(continues on next page)

(continued from previous page)

```
>>> drawMesh(ax, world)
>>> plt.show()
```



Examples using `pygimli.meshtools.createWorld`

- *Raypaths in layered and gradient models* (page 46)
- *2D ERT modeling and inversion* (page 60)
- *2D FEM modelling on two-layer example* (page 73)
- *Four-point sensitivities* (page 81)
- *Complex-valued electrical modeling* (page 98)
- *Naive complex-valued electrical inversion* (page 104)
- *Gravimetry in 2D - Part I* (page 116)
- *Hydrogeophysical modeling* (page 138)
- *Quality of unstructured meshes* (page 202)
- *Heat equation in 2D* (page 223)

`pygimli.meshtools.`**`exportFenicsHDF5Mesh`**(*mesh*, *exportname*)

Exports Gimli mesh in HDF5 format suitable for Fenics.

Equivalent to calling the function `pygimli.meshtools.exportHDF5Mesh(mesh, exportname, group=['mesh', 'domains'], indices='cell_indices', pos='coordinates', cells='topology', marker='values')`.

Parameters

- **mesh** (*:gimliapi:GIMLI::Mesh`*) – Mesh to be saved.

- **exportname** (*string*) – Name under which the mesh is saved.

pygimli.meshtools.**exportHDF5Mesh**(*mesh*, *exportname*, *group='mesh'*, *indices='cell_indices'*, *pos='coordinates'*, *cells='topology'*, *marker='values'*)

Writes given GIMLI::Mesh in a hdf5 format file.

3D tetrahedral meshes only! Boundary markers are ignored.

Keywords are explained in `pygimli.meshtools.readHDFS`

pygimli.meshtools.**exportPLC**(*poly*, *fname*, ***kwargs*)

Export a piece-wise linear complex (PLC) to a .poly file (2D or 3D).

Chooses from poly.dimension() and forwards accordingly to GIMLI::Mesh::exportAsTetgenPolyFile or `pygimli.meshtools.writeTrianglePoly`

Parameters

- **poly** (GIMLI::Mesh) – The polygon to be written.
- **fname** (*string*) – Filename of the file to write (*.n, *.e).

Examples

```
>>> import pygimli as pg
>>> import tempfile, os
>>> fname = tempfile.mktemp() + '.poly' # Create temporary filename.
>>> world2d = pg.meshtools.createWorld(start=[-10, 0], end=[20, -10])
>>> pg.meshtools.exportPLC(world2d, fname)
>>> read2d = pg.meshtools.readPLC(fname)
>>> print(read2d)
Mesh: Nodes: 4 Cells: 0 Boundaries: 4
>>> world3d = pg.createGrid([0, 1], [0, 1], [-1, 0])
>>> pg.meshtools.exportPLC(world3d, fname)
>>> os.remove(fname)
```

See also:

readPLC (page 356)

pygimli.meshtools.**exportSTL**(*mesh*, *fileName*, *binary=False*)

Write *STL* surface mesh and returns a GIMLI::Mesh.

Export a three dimensional boundary GIMLI::Mesh into a *STL* surface mesh. Boundaries with different marker will be separated into different STL solids.

Parameters

- **mesh** (GIMLI::Mesh) – Mesh to be exported. Only Boundaries of type TriangleFace will be exported.
- **fileName** (*str*) – name of the .stl file containing the STL surface mesh
- **binary** (*bool [False]*) – Write STL binary format. TODO

pygimli.meshtools.**extractUpperSurface2dMesh**(*mesh*, *zCut=None*)

Extract 2d mesh from the upper surface of a 3D mesh.

Useful for showing a quick 2D plot of a 3D parameter distribution All cell-based parameters are copied to the new mesh

Parameters

- **mesh** (GIMLI::Mesh) – Input mesh (3D)
- **zCut** (*float*) – z value to distinguish between top and bottom

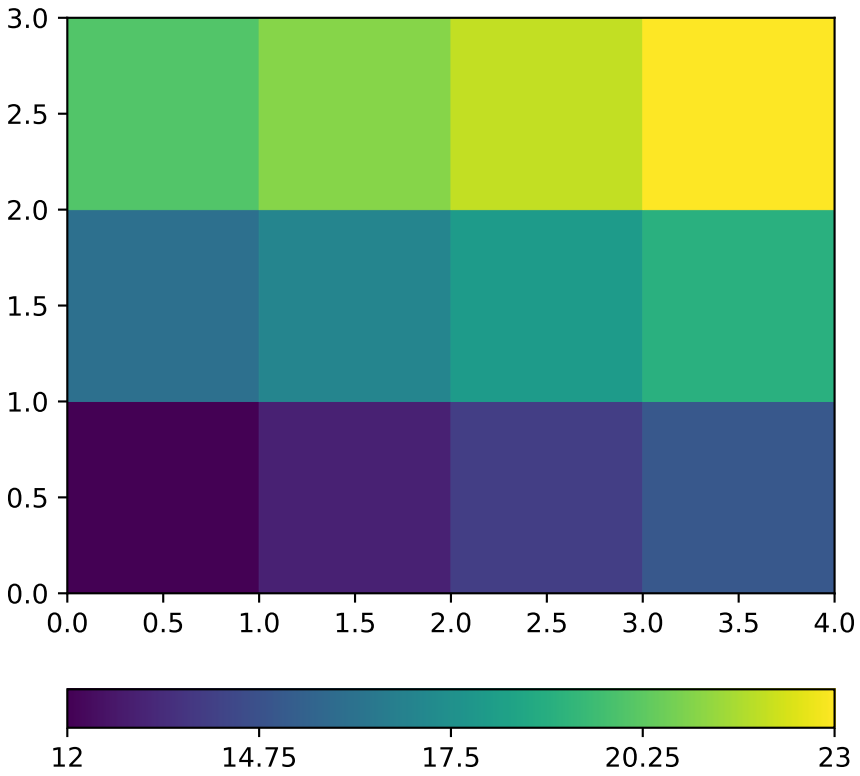
Returns

mesh2d – output 2D mesh consisting of triangles or quadrangles

Return type

GIMLI::Mesh

```
>>> import pygimli as pg
>>> from pygimli.meshtools import extractUpperSurface2dMesh
>>> mesh3d = pg.createGrid(5, 4, 3)
>>> mesh3d["val"] = pg.utils.grange(0, mesh3d.cellCount(), 1)
>>> mesh2d = extractUpperSurface2dMesh(mesh3d)
>>> ax, _ = pg.show(mesh2d, "val")
```



pygimli.meshtools.**extrude**(*p2*, *z=-1.0*, *boundaryMarker=0*, ***kwargs*)

Create 3D body by extruding a closed 2D poly into z direction

Parameters

- **p2** (GIMLI::Mesh) – 2D geometry

- **z** (*float [-1.0]*) – 2D geometry

Keyword Arguments

kwargs (****) – Marker related arguments: See pygimli.meshtools.polytools.setPolyRegionMarker

Returns

poly – The resulting polygon is a GIMLI::Mesh.

Return type

GIMLI::Mesh

pygimli.meshtools.**extrudeMesh**(*mesh*, *a*, ***kwargs*)

Extrude mesh to a higher dimension.

Generates a 2D mesh by extruding a 1D mesh along y-coordinate using quads. We assume a 2D mesh here consisting of nodes and edges. The marker of nodes are extruded as edges with the same marker. The marker of the edges are extruded as cells with same marker. Optionally all y-coordinates can be adjusted to become equal at the end

Generates a three-dimensional mesh by extruding a two-dimensional mesh along the z-coordinate transforming triangles into triangular prisms or quads into hexahedrons. 3D cell markers are set from 2D cell marker. The boundary marker for the side boundaries are set from edge markers.

Parameters

- **mesh** (GIMLI::Mesh) – Input mesh
- **a** (*iterable (float)*) – Additional coordinate to extrude into.

Keyword Arguments

adjustBottom (*bool [False]*) – Adjust all nodes such that the bottom of the mesh has a constant depth (only 2D)

Returns

mesh – Returning mesh of +1 dimension

Return type

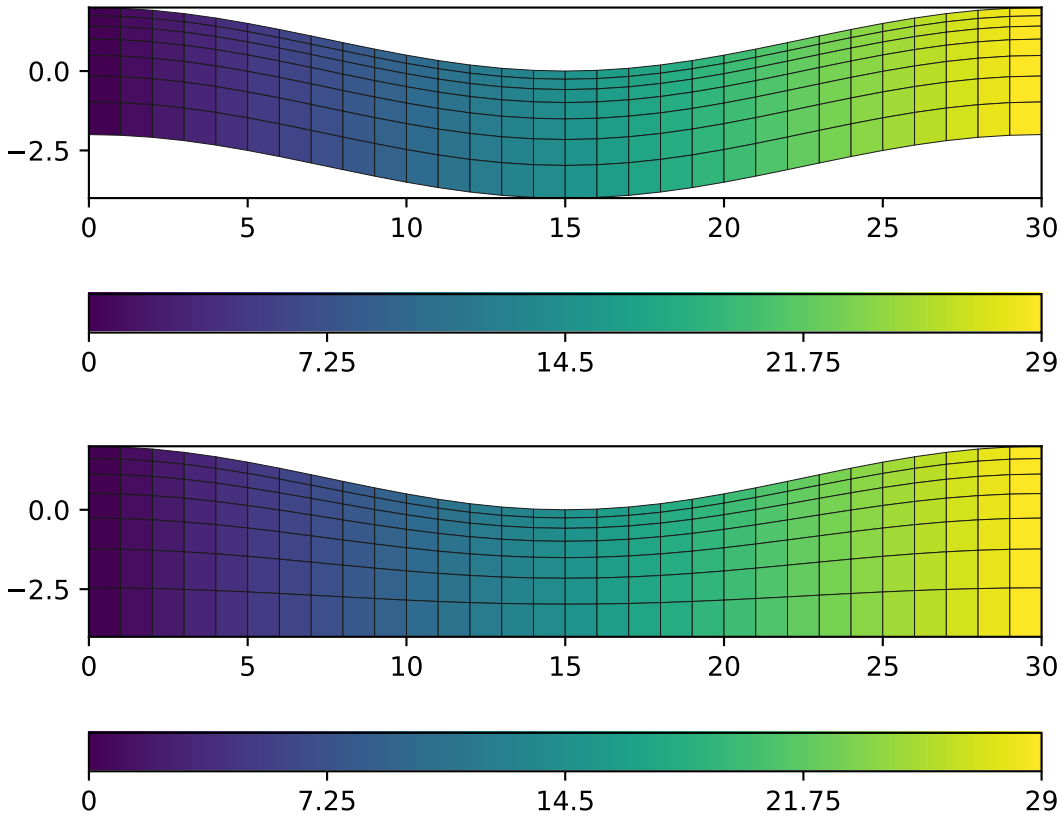
GIMLI::Mesh

```
>>> import numpy as np
>>> import pygimli as pg
>>> import pygimli.meshtools as mt
>>> topo = [[x, 1.0 + np.cos(2 * np.pi * 1/30 * x)] for x in range(31)]
>>> m1 = mt.createPolygon(topo)
>>> m1.setBoundaryMarkers(range(m1.boundaryCount()))
```

```
>>> m = mt.extrudeMesh(m1, a=-(np.geomspace(1, 5, 8)-1.0))
>>> _ = pg.show(m, m.cellMarkers(), showMesh=True)
>>> m = mt.extrudeMesh(m1, a=-(np.geomspace(1, 5, 8)-1.0),
...                    adjustBottom=True)
>>> _ = pg.show(m, m.cellMarkers(), showMesh=True)
```

Examples using pygimli.meshtools.extrudeMesh

- *Extrude a 2D mesh to 3D* (page 25)



pygimli.meshtools.**fillEmptyToCellArray**(*mesh*, *vals*, *slope=True*)

Prolongate empty cell values to complete cell attributes.

It is possible to have zero values that are filled with appropriate attributes. This function tries to fill empty values successively by prolongation of the non-zeros.

Parameters

- **mesh** (GIMLI::Mesh) – For each cell of mesh a value will be returned.
- **vals** (*array*) – Array of size cellCount().

Returns

atts – Array of length mesh.cellCount()

Return type

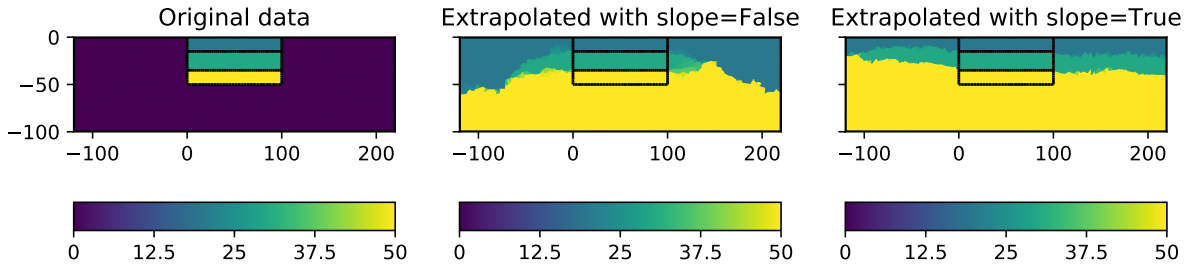
array

```
>>> import pygimli as pg
>>> import pygimli.meshtools as mt
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>>
>>> # Create a mesh with 3 layers and an outer region for␣
↪extrapolation
>>> layers = mt.createWorld([0,-50],[100,0], layers=[-15,-35])
>>> inner = mt.createMesh(layers, area=3)
>>> mesh = mt.appendTriangleBoundary(inner, xbound=120, ybound=50,
...                                  area=20, marker=0)
```

(continues on next page)

(continued from previous page)

```
>>>
>>> # Create data for the inner region only
>>> layer_vals = [20,30,50]
>>> data = np.array(layer_vals)[inner.cellMarkers() - 1]
>>>
>>> # The following fails since len(data) != mesh.cellCount(),␣
↪extrapolate
>>> # pg.show(mesh, data)
>>>
>>> # Create data vector, where zeros fill the outer region
>>> data_with_outer = np.array([0] + layer_vals)[mesh.cellMarkers()]
>>>
>>> # Actual extrapolation
>>> extrapolated_data = mt.fillEmptyToCellArray(mesh,
...                                     data_with_outer, slope=False)
>>> extrapolated_data_with_slope = mt.fillEmptyToCellArray(mesh,
...                                    data_with_outer, slope=True)
>>>
>>> # Visualization
>>> fig, (ax1, ax2, ax3) = plt.subplots(1,3, figsize=(10,8), sharey=True)
>>> _ = pg.show(mesh, data_with_outer, ax=ax1, cMin=0)
>>> _ = pg.show(mesh, extrapolated_data, ax=ax2, cMin=0)
>>> _ = pg.show(mesh, extrapolated_data_with_slope, ax=ax3, cMin=0)
>>> _ = ax1.set_title("Original data")
>>> _ = ax2.set_title("Extrapolated with slope=False")
>>> _ = ax3.set_title("Extrapolated with slope=True")
```



pygimli.meshtools.**fromSubsurface**(*obj*, *order='C'*, *verbose=False*)

Convert subsurface object to pygimli mesh.

See more: https://softwareunderground.github.io/subsurface/

Order refers to np.flatten(order) strategy for structured cell data arrangement, e.g., use 'F' (Fortran style) for gempy meshes. Default is 'C'-Style.

Testet objects so far:

- TriSurf
- UnstructuredData (3D Boundary from TriSurf)
- StructuredData (3D cell centered voxel)

Parameters

- **obj** (*obj*) – Subsurface obj, mesh object
- **order** (*str ['C']*) – Flatten style for structured data attributes. See above.
- **verbose** (*boolean [False]*) – Be verbose during import.

Returns
mesh

Return type
GIMLI::Mesh

pygimli.meshtools.**interpolate**(**args*, ***kwargs*)

Interpolation convinience function.

Convenience function to interpolate different kind of data. Currently supported interpolation schemes are:

- Interpolate mesh based data from one mesh to another

  (syntactic sugar for the core based interpolate (see below))

  Parameters:

  args: GIMLI::Mesh, GIMLI::Mesh, iterable
  outData = interpolate(outMesh, inMesh, vals) Interpolate values based on inMesh to outMesh. Values can be of length inMesh.cellCount() interpolated to outMesh.cellCenters() or inMesh.nodeCount() which are interpolated to outMesh.positions().

  Returns:
  Interpolated values.

- Mesh based values to arbitrary points, based on finite element interpolation (from gimli core).

  Parameters:

  args: GIMLI::Mesh, . . .
  Arguments forwarded to GIMLI::interpolate()

  kwargs:
  Arguments forwarded to GIMLI::interpolate()

  interpolate(srcMesh, destMesh)
  All data from inMesh are interpolated to outMesh

  Returns:
  Interpolated values

- Interpolate along curve. Forwarded to *pygimli.meshtools.interpolateAlongCurve* (page 348)

  Parameters:
  args: curve, t

  kwargs:
  Arguments forwarded to *pygimli.meshtools.interpolateAlongCurve* (page 348)

periodic
[bool [False]] Curve is periodic. Useful for closed parametric spline interpolation.

- 1D point set $u(x)$ for ascending x. Find interpolation function $I = u(x)$ and returns $u_i = I(x_i)$ (interpolation methods are [**linear** via matplotlib, cubic **spline** via scipy, fit **harmonic** functions' via pygimli]) Note, for 'linear' and 'spline' the interpolate contains all original coordinates while 'harmonic' returns an approximate best fit. The amount of harmonic coefficients can be specfied by the 'nc' keyword.

  Parameters:

  args: xi, x, u

  - x_i - target sample points
  - x - function sample points
  - u - function values

  kwargs:

  - **method**
    [string] Specify interpolation method 'linear, 'spline', 'harmonic'
  - **nc**
    [int] Number of harmonic coefficients for the 'harmonic' method.
  - **periodic**
    [bool [False]] Curve is periodic. Useful for closed parametric spline interpolation.

  Returns:

  ui: array of length xi
  $u_i = I(x_i)$, with $I = u(x)$

To use the core functions GIMLI::interpolate() start with a mesh instance as first argument or use the appropriate keyword arguments.

TODO

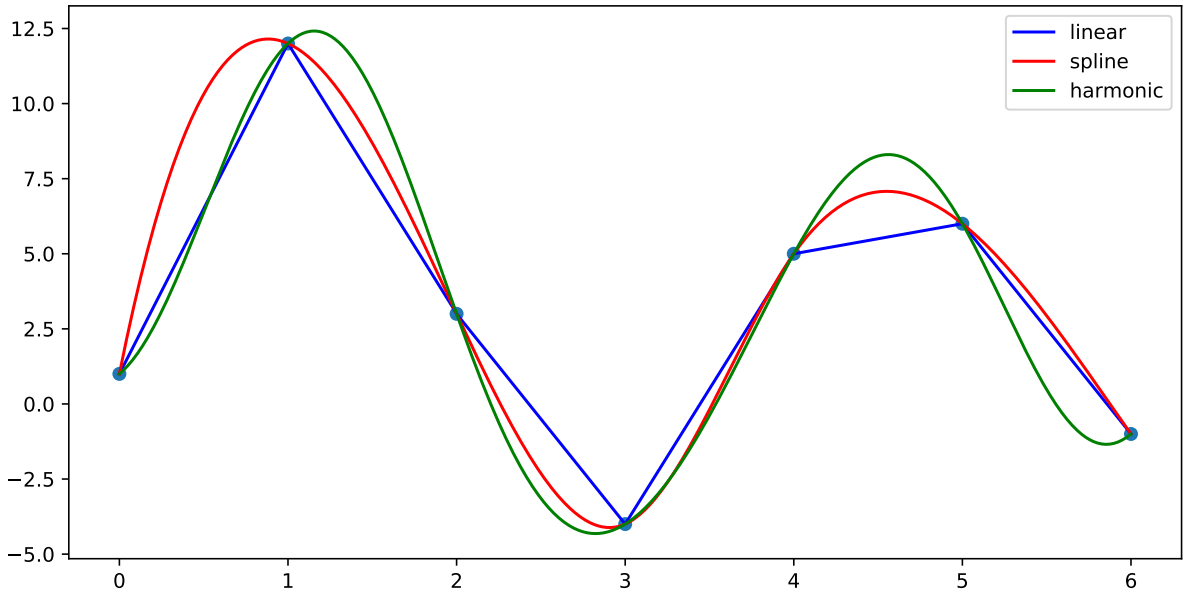
- 2D parametric to points (method=['linear, 'spline', 'harmonic'])
- **2D/3D point cloud to points/grids**
  ('Delauney', 'linear, 'spline', 'harmonic')
- Mesh to points based on nearest neighbor values (pg.core)

```
>>> import numpy as np
>>> import pygimli as pg
>>> fig, ax = pg.plt.subplots(1, 1, figsize=(10, 5))
>>> u = np.array([1.0, 12.0, 3.0, -4.0, 5.0, 6.0, -1.0])
>>> xu = np.array(range(len(u)))
>>> xi = np.linspace(xu[0], xu[-1], 1000)
>>> _= ax.plot(xu, u, 'o')
>>> _= ax.plot(xi, pg.interpolate(xi, xu, u, method='linear'),
...         color='blue', label='linear')
>>> _= ax.plot(xi, pg.interpolate(xi, xu, u, method='spline'),
```

(continues on next page)

(continued from previous page)

```
...             color='red', label='spline')
>>> _= ax.plot(xi, pg.interpolate(xi, xu, u, method='harmonic'),
...         color='green', label='harmonic')
>>> _= ax.legend()
>>>
>>> pg.plt.show()
```



pygimli.meshtools.**interpolateAlongCurve**(*curve*, *t*, ***kwargs*)

Interpolate along curve.

Return curve coordinates for a piecewise linear curve $C(t) = x_i, y_i, z_i$ at positions t. Curve and t values are expected to be sorted along distance from the origin of the curve.

Parameters

- **curve** ([[x,z]] | [[x,y,z]] | [GIMLI::RVector3] | GIMLI::R3Vector) – Discrete curve for 2D x, z curve=[[x,z]], 3D x, y, z
- **t** (*1D iterable*) – Query positions along the curve in absolute distance
- **kwargs** – If kwargs are given, an additional curve smoothing is applied using *pygimli.meshtools.interpolate* (page 346). The kwargs will be delegated.

  periodic
  [bool [False]] Curve is periodic. Usefull for closed parametric spline interpolation.

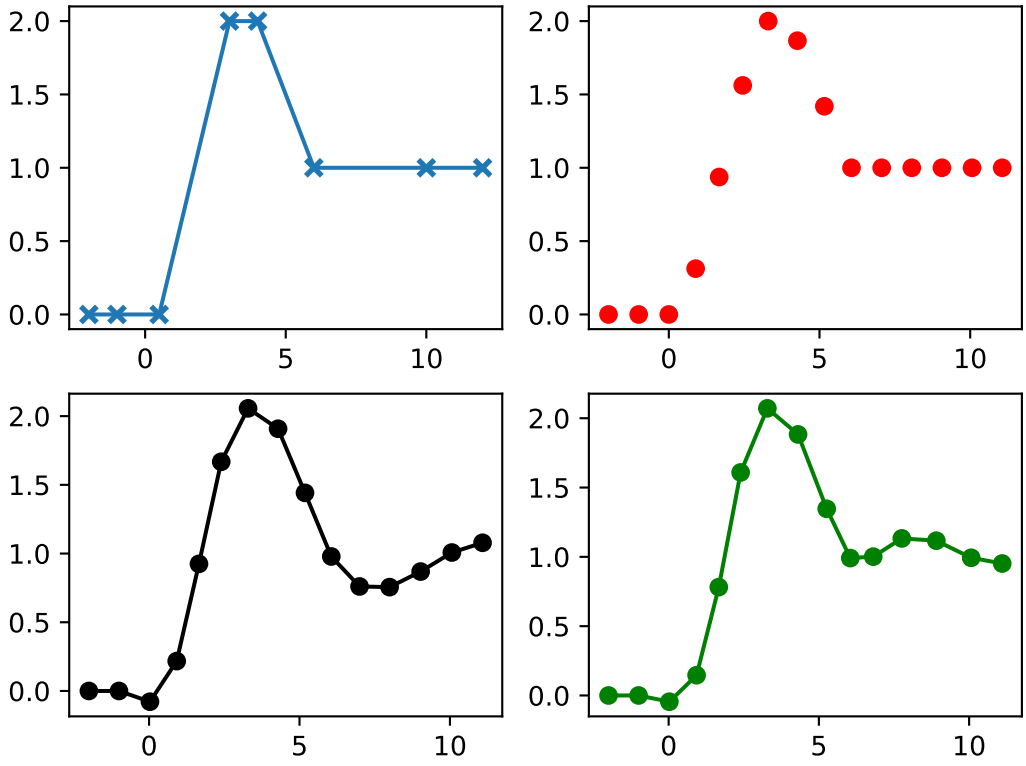
Returns

p – Curve positions at query points t. Dimension of p match the size of curve the coordinates.

Return type

np.array

```
>>> import numpy as np
>>> import pygimli as pg
>>> import pygimli.meshtools as mt
>>> fig, axs = pg.plt.subplots(2,2)
>>> topo = np.array([[-2., 0.], [-1., 0.], [0.5, 0.], [3., 2.], [4., 2.],␣
↪[6., 1.], [10., 1.], [12., 1.]])
>>> t = np.arange(15.0)
>>> p = mt.interpolateAlongCurve(topo, t)
>>> _= axs[0,0].plot(topo[:,0], topo[:,1], '-x', mew=2)
>>> _= axs[0,1].plot(p[:,0], p[:,1], 'o', color='red')
>>>
>>> p = mt.interpolateAlongCurve(topo, t, method='spline')
>>> _= axs[1,0].plot(p[:,0], p[:,1], '-o', color='black')
>>>
>>> p = mt.interpolateAlongCurve(topo, t, method='harmonic', nc=3)
>>> _= axs[1,1].plot(p[:,0], p[:,1], '-o', color='green')
>>>
>>> pg.plt.show()
```



pygimli.meshtools.**merge**(**args*, ***kwargs*)

Little syntactic sugar to merge.

All args are forwarded to mergeMeshes if isGeometry is not set. Otherwise it considers the mesh as PLC to merge.

Parameters

- **to** (*List of meshes or comma separated list of meshes that will be forwarded*) –

- **meshPLC.** (*mergeMeshes or*) –

pygimli.meshtools.**merge2Meshes**(*m1*, *m2*)

Merge two meshes into one new mesh and return the combined mesh.

Merge two meshes into a new mesh and return the combined mesh. Note that there is a duplicate check for all nodes which should reuse existing node but NO cells or boundaries.

Parameters

- **m1** (GIMLI::Mesh) – First mesh.
- **m2** (GIMLI::Mesh) – Second mesh.

Returns

mesh – Resulting mesh.

Return type

GIMLI::Mesh

pygimli.meshtools.**mergeMeshes**(*meshList*, *verbose=False*)

Merge several meshes into one new mesh and return the new mesh.

Merge several meshes into one new mesh and return the new mesh.

Parameters

- **meshList** ([GIMLI::Mesh, ...] | [str, ...]) – List of at least two meshes (or filenames to meshes) to be merged.
- **verbose** (*bool*) – Give some output

See also:

merge2Meshes (page 350)

Examples using pygimli.meshtools.mergeMeshes

- *Building a hybrid mesh in 2D* (page 33)

pygimli.meshtools.**mergePLC**(*plcs*, *tol=0.001*)

Merge multiply polygons.

Merge multiply polygons into a single polygon. Common nodes and common edges will be checked and removed. When a node touches an edge, the edge will be splited.

3D only OOC with polytools

Parameters

- **plcs** ([GIMLI::Mesh]) – List of PLC that want to be merged into one new PLC
- **tol** (*double*) – Tolerance to check for duplicated nodes. [1e-3]

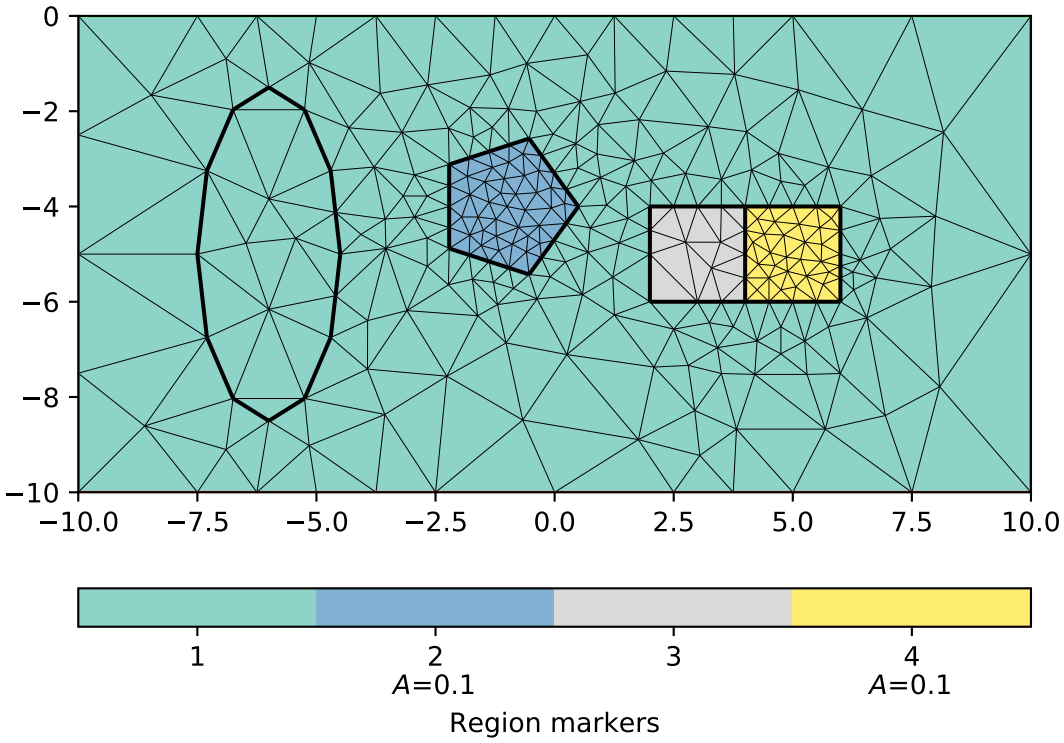
Returns

plc – The resulting polygon is a GIMLI::Mesh.

Return type

GIMLI::Mesh

```
>>> import pygimli as pg
>>> import pygimli.meshtools as mt
>>> from pygimli.viewer.mpl import drawMesh
>>> world = mt.createWorld(start=[-10, 0], end=[10, -10], marker=1)
>>> c1 = mt.createCircle([-1, -4], radius=1.5, area=0.1,
...                      marker=2, nSegments=5)
>>> c2 = mt.createCircle([-6, -5], radius=[1.5, 3.5], isHole=1)
>>> r1 = mt.createRectangle(pos=[3, -5], size=[2, 2], marker=3)
>>> r2 = mt.createRectangle(start=[4, -4], end=[6, -6],
...                         marker=4, area=0.1)
>>> plc = mt.mergePLC([world, c1, c2, r1, r2])
>>> fig, ax = pg.plt.subplots()
>>> drawMesh(ax, plc)
>>> drawMesh(ax, mt.createMesh(plc))
>>> pg.wait()
```



Examples using `pygimli.meshtools.mergePLC`

- *Naive complex-valued electrical inversion* (page 104)
- *Petrophysical joint inversion* (page 153)

`pygimli.meshtools.`**`mergePLC3D`**(*plcs*, *tol=0.001*)

Merge a list of 3D PLC into one

Experimental replacement for polyMerge. Don't expect too much.

Works if:

- all plcs are free and does not have any contact to each other
- contact of two facets if the second is completely within the first

pygimli.meshtools.**nodeDataToBoundaryData**(*mesh*, *data*)

Assuming [NodeCount, dim] data DOCUMENT_ME

pygimli.meshtools.**nodeDataToCellData**(*mesh*, *data*)

Convert node data to cell data.

Convert node data to cell data via interpolation to cell centers.

Parameters

- **mesh** (GIMLI::Mesh) – 2D or 3D GIMLi mesh
- **data** (*iterable [float]*) – Data of len mesh.nodeCount(). TODO complex, R3Vector, ndarray

Examples

Examples using pygimli.meshtools.nodeDataToCellData

- *Raypaths in layered and gradient models* (page 46)

pygimli.meshtools.**quality**(*mesh*, *measure='eta'*)

Return the quality of a given triangular mesh.

Parameters

- **mesh** (*mesh object*) – Mesh for which the quality is calculated.
- **measure** (*quality measure, str*) – Can be either “eta”, “nsr”, or “minimumAngle”.

```
>>> # no need to import matplotlib
>>> import pygimli as pg
>>> from pygimli.meshtools import polytools as plc
>>> from pygimli.meshtools import quality
>>> # Create Mesh
>>> world = plc.createWorld(start=[-10, 0], end=[10, -10],
...                         marker=1, worldMarker=False)
>>> c1 = plc.createCircle(pos=[0.0, -5.0], radius=3.0, area=.3)
>>> mesh = pg.meshtools.createMesh([world, c1], quality=21.3)
>>> # Compute and show quality
>>> q = quality(mesh, measure="nsr")
>>> ax, _ = pg.show(mesh, q, cMap="RdYlGn", showMesh=True, cMin=0.5,
...                 cMax=1.0, label="Normalized shape ratio")
```

See also:

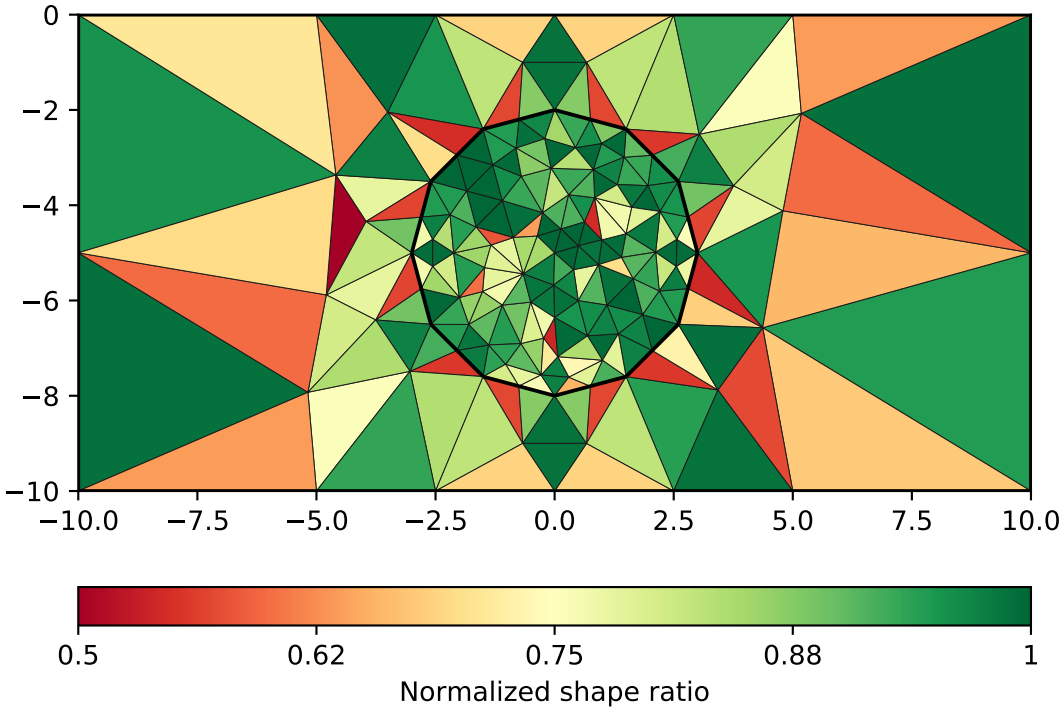
eta, nsr, minimumAngle

Examples using pygimli.meshtools.quality

- *Quality of unstructured meshes* (page 202)

pygimli.meshtools.**readFenicsHDF5Mesh**(*fileName*, *verbose=True*, ***kwargs*)

Reads *FEniCS* mesh from file format .h5 and returns a GIMLI::Mesh.



pygimli.meshtools.**readGmsh**(*fName*, *verbose=False*, *precision=None*)

Read *Gmsh* ASCII file and return instance of GIMLI::Mesh class.

Parameters

- **fName** (*string*) – Filename of the file to read (*.msh). The file must conform to the MSH ASCII file version 2 format
- **verbose** (*boolean, optional*) – Be verbose during import. Default: False
- **precision** (*None|int, optional*) – If not None, then round off node coordinates to the provided number of digits using numpy.round. This is useful in case that nodes are accessed using their coordinates, in which case numerical discrepancies can occur.

Notes

Physical groups specified in Gmsh are interpreted as follows:

- Points with the physical number 99 are interpreted as sensors. Note that physical point groups are ordered with respect to the node tag. E.g. “Physical Point (99) = {50, 34};” and “Physical Point (99) = {34, 50};” will yield the same mesh. This must be taken into account when defining measurement configurations using electrodes defined in GMSH using marker 99.
- ERT only: Points with markers 999 and 1000 are used to mark calibration and reference nodes.
- **Physical Lines and Surfaces define boundaries in 2D and 3D, respectively.**
  - Physical Number 1: Homogeneous Neumann condition

  - Physical Number 2: Mixed boundary condition
  - Physical Number 3: Homogeneous Dirichlet condition
  - Physical Number 4: Dirichlet condition
- **Physical Surfaces and Volumes define regions in 2D and 3D, respectively.**
  - Physical Number 1: No inversion region
  - Physical Number >= 2: Inversion region

Examples

```
>>> import tempfile, os
>>> from pygimli.meshtools import readGmsh
>>> gmsh = '''
... $MeshFormat
... 2.2 0 8
... $EndMeshFormat
... $Nodes
... 3
... 1 0 0 0
... 2 0 1 0
... 3 1 1 0
... $EndNodes
... $Elements
... 7
... 1 15 2 0 1 1
... 2 15 2 0 2 2
... 3 15 2 0 3 3
... 4 1 2 0 1 2 3
... 5 1 2 0 2 3 1
... 6 1 2 0 3 1 2
... 7 2 2 0 5 1 2 3
... $EndElements
... '''
>>> fName = tempfile.mktemp()
>>> with open(fName, "w") as f:
...     f.writelines(gmsh)
>>> mesh = readGmsh(fName)
>>> print(mesh)
Mesh: Nodes: 3 Cells: 1 Boundaries: 3
>>> os.remove(fName)
```

Examples using `pygimli.meshtools.readGmsh`

- *Flexible mesh generation using Gmsh* (page 26)

`pygimli.meshtools.`**`readHDF5Mesh`**(*fileName*, *group='mesh'*, *indices='cell_indices'*, *pos='coordinates'*, *cells='topology'*, *marker='values'*, *marker_default=0*, *dimension=3*, *verbose=True*, *useFenicsIndices=False*)

Function for loading a mesh from HDF5 file format.

Returns an instance of GIMLI::Mesh class. Default values for keywords are suited for *FEniCS* syntax .h5 meshes.

Requirements: h5py module

Parameters

- **fileName** (*string*) – Name of the mesh to be transformed into *pyGIMLi* format.
- **group** (*string ['domains']*) – hdf group that contains the mesh information (see other keyword arguments). Default is 'domains' for *FEniCS* compatibility.
- **indices** (*string ['cell_indices']*) – Key for the part of the hdf file containing the indices of the cells.
- **pos** (*string ['coordinates']*) – Key for the part of the hdf file containing the nodepositions.
- **cells** (*string ['topology']*) – Key for the part of the hdf file containing the array which defies the cells. Usually of shape (cellCount, 3) for 2D meshes or (cellCount, 4) for 3D tetrahedra meshes. For each cell the indices of the corresponding node indices is given.
- **marker** (*string ['values']*) – If marker is part of the hdf data container, the corresponding array is used as identifier for the cell markers. If not found, the cell markers will be set to marker_default.
- **marker_default** (*int or array [0]*) – Default marker if no markers are found in the hdf file. If array, size has to match the cellCount of the mesh.
- **dimension** (*int [3]*) – Dimension of the input/output mesh, no own check for dimensions yet. Fixed on 3 for now.

Returns

GIMLI::Mesh

Return type

mesh

pygimli.meshtools.**readHydrus2dMesh**(*fileName='MESHTRIA.TXT'*)

Import mesh from Hydrus2D.

Parameters

fName (*str, optional*) – Filename of Hydrus output file.

See also:

readHydrus3dMesh **(page 356)**

Similar routine for three-dimensional meshes.

References

pygimli.meshtools.**readHydrus3dMesh**(*fileName='MESHTRIA.TXT'*)

Import mesh from Hydrus3D.

Parameters
fName (*str, optional*) – Filename of Hydrus output file.

See also:

readHydrus2dMesh **(page 355)**
Similar routine for two-dimensional meshes.

References

pygimli.meshtools.**readMeshIO**(*fileName*, *verbose=False*)

Generic mesh read using meshio. (https://github.com/nschloe/meshio)

pygimli.meshtools.**readPLC**(*filename*, *comment='#'*)

Read in a piece-wise linear complex object (PLC) from .poly file.

A PLC is a pyGIMLi geometry, e.g., created using *mt.exportPLC*.

Read 2D *Triangle* or 3D *Tetgen* PLC files.

Parameters

- **filename** (*string*) – Filename *.poly
- **comment** (*string ('#')*) – String containing all characters that define a comment line. Identified lines will be ignored during import.

Returns
GIMLI::Mesh

Return type
poly

See also:

exportPLC (page 341)

pygimli.meshtools.**readSTL**(*fileName*, *binary=False*)

Read *STL* surface mesh and returns a GIMLI::Mesh.

Read *STL* surface mesh and returns a GIMLI::Mesh of triangle boundary faces. Multiple solids are supported with increasing boundary marker.

TODO: ASCII=False, read binary STL

Parameters

- **fileName** (*str*) – name of the .stl file containing the STL surface mesh
- **binary** (*bool [False]*) – STL Binary format

pygimli.meshtools.**readTetgen**(*fName*, *comment='#'*, *verbose=False*, *defaultCellMarker=0*, *loadFaces=True*, *quadratic=False*)

Read and convert a mesh from the basic *Tetgen* output.

Read *Tetgen* [?] ASCII files and return instance of GIMLI::Mesh class. See: http://tetgen.org/

Parameters

- **fName** (*str*) – Base name of the tetgen output, without ending. All additional files (.node, .ele and .face) need to have the same basename.
- **comment** (*str ('#')*) – String consisting of all symbols indicating a comment in the input files. Standard for tetgen files is the '#'.
- **verbose** (*boolean (True)*) – Enables console output during the import process.
- **defaultCellMarker** (*int (0)*) – *Tetgen* files can contain cell markers, but do not have to. If no markers are found, the given integer is used.
- **loadFaces** – Optional decision whether the faces of *Tetgen* output (.face) are loaded or not. Note that without the -f in during the tetgen call, the faces in the .face file will only contain the faces of the original input poly file and not all faces. If only a part of the faces are imported, a createNeighborInfos call of the mesh will fail.
- **quadratic** (*boolean (False)*) – Returns a P2 (quadratic) refined mesh when True (to be removed, as soon as direct import of quadratic meshes is possible).

Returns

mesh

Return type

GIMLI::Mesh

pygimli.meshtools.**readTriangle**(*fName*, *verbose=False*)

Read *Triangle* [?] mesh.

Read *Triangle* [?] ASCII mesh files and return an instance of GIMLI::Mesh class. See: ://www.cs.cmu.edu/~quake/triangle.html

Parameters

- **fName** (*string*) – Filename of the file to read (*.n, *.e)
- **verbose** (*boolean, optional*) – Be verbose during import.

pygimli.meshtools.**refineHex2Tet**(*mesh*, *style=1*)

Refine mesh of hexahedra into a mesh of tetrahedra.

Parameters

- **mesh** (GIMLI::Mesh) – Mesh containing hexrahedron cells, e.g., from a grid.
- **style** (*int [1]*) –
  - 1 bisect each hexahedron int 6 tetrahedrons (less numerical quality but no problems due to diagonal face split)

- 2 bisect each hexahedron int 5 tetrahedrons (leads to inconsistent meshes. Neighboring cell have different face

  split diagonal. Might be fixable by rotating the split order depending on coordinates for every 2nd split)

Returns

ret – Mesh containing tetrahedrons cells.

Return type

GIMLI::Mesh

Examples

```
>>> import pygimli as pg
>>> import pygimli.meshtools as mt
>>> hex = pg.createGrid(2, 2, 2)
>>> print(hex)
Mesh: Nodes: 8 Cells: 1 Boundaries: 6
>>> tet = mt.refineHex2Tet(hex, style=1)
>>> print(tet)
Mesh: Nodes: 8 Cells: 6 Boundaries: 12
>>> tet = mt.refineHex2Tet(hex, style=2)
>>> print(tet)
Mesh: Nodes: 8 Cells: 5 Boundaries: 12
```

pygimli.meshtools.**refineQuad2Tri**(*mesh*, *style=1*)

Refine mesh of quadrangles into a mesh of triangle cells.

TODO mixed meshes

Parameters

- **mesh** (GIMLI::Mesh) – Mesh containing quadrangle cells.
- **style** (*int [1]*) –
  - 1 bisect each quadrangle into 2 triangles
  - 2 cross-sect each quadrangle into 4 triangles

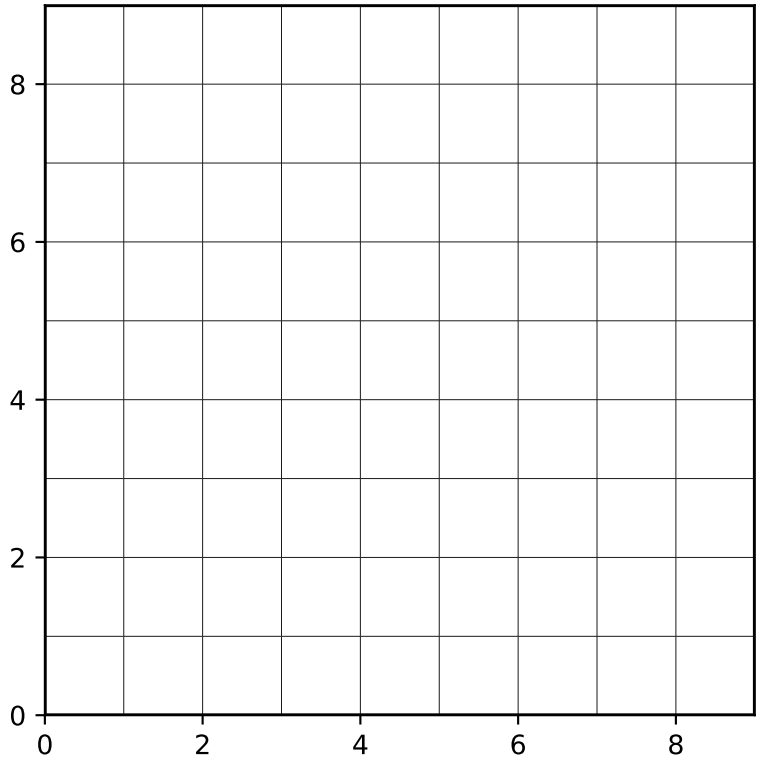
Returns

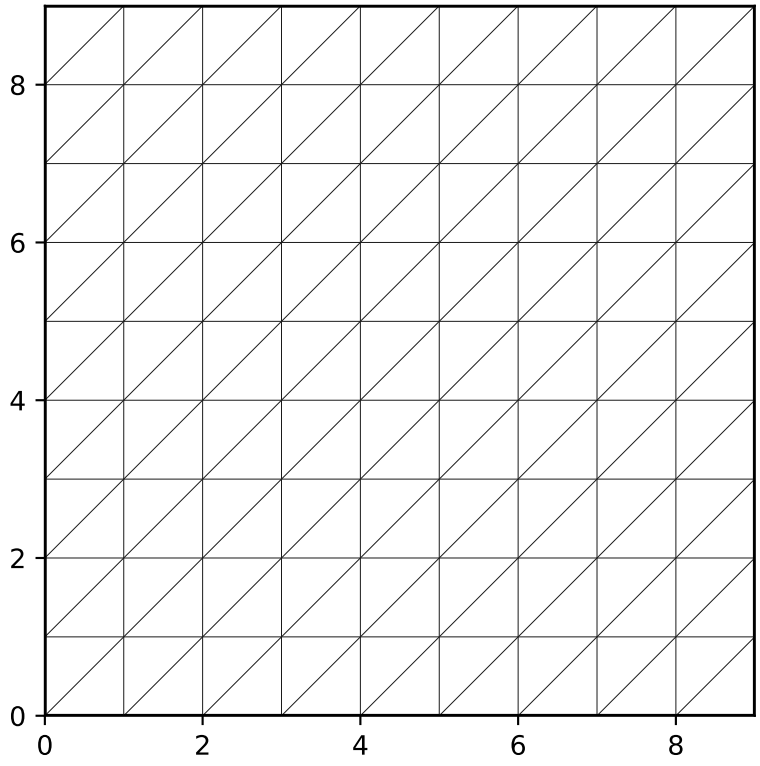
ret – Mesh containing triangle cells.

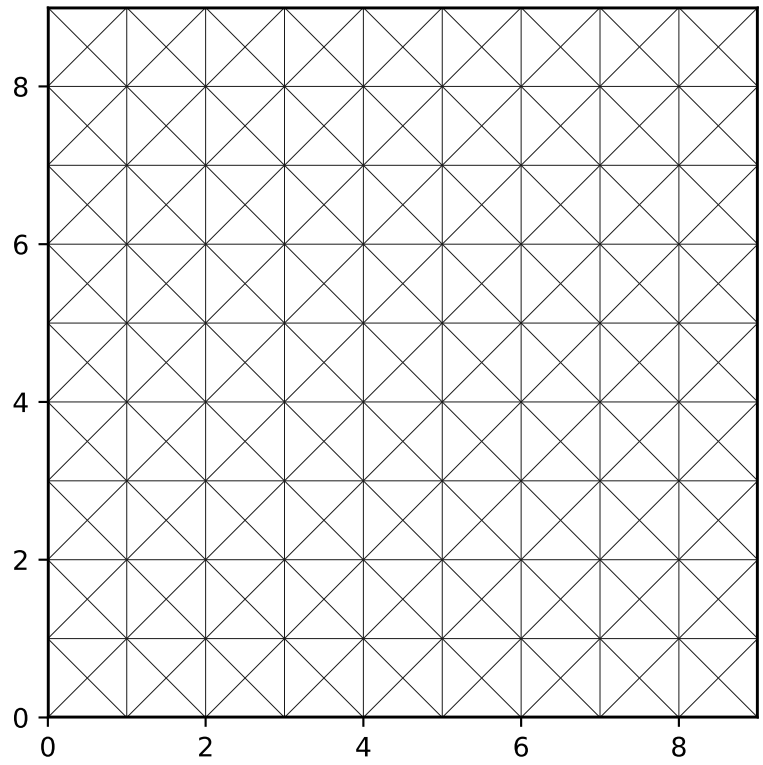
Return type

GIMLI::Mesh

```
>>> import pygimli as pg
>>> import pygimli.meshtools as mt
>>> quads = pg.createGrid(range(10), range(10))
>>> ax, _ = pg.show(quads)
>>> ax, _ = pg.show(mt.refineQuad2Tri(quads, style=1))
>>> ax, _ = pg.show(mt.refineQuad2Tri(quads, style=2))
>>> pg.wait()
```


0
2
4
6
8
0
2
4
6
8


0
2
4
6
8
0
2
4
6
8



pygimli.meshtools.**syscallTetgen**(*filename*, *quality=1.2*, *area=0*, *preserveBoundary=False*, *verbose=False*, *tetgen='tetgen'*)

Create a mesh from a PLC by system-calling *Tetgen*.

Create a *Tetgen* [**?**] mesh from a PLC.

Parameters

- **filename** (*str*) –
- **quality** (*float [1.2]*) – Refines mesh (to improve mesh quality). [1.1 ...]
- **area** (*float [0.0]*) – Maximum cell size (m^3)
- **preserveBoundary** (*bool [False]*) – Preserve PLC boundary mesh
- **verbose** (*bool [False]*) – be verbose
- **tetgen** (*str | path ['tetgen']*) – Binary for tetgen. Given as complete path or simple the binary name if its known in the system path.

Returns
mesh

Return type
GIMLI::Mesh

pygimli.meshtools.**tapeMeasureToCoordinates**(*tape*, *pos*)

Interpolate 2D tape measured topography to 2D Cartesian coordinates.

Tape and pos value are expected to be sorted along distance to the origin.

DEPRECATED will be removed, use *pygimli.meshtools.interpolateAlongCurve* (page 348) instead

TODO optional smooth curve with harmfit TODO parametric TODO parametric + Topo: 3d

Parameters

- **tape** (*[[x,z]] | [RVector3] | R3Vector*) – List of tape measured topography points with measured distance (x) from origin and height (z)
- **pos** (*iterable*) – Array of query positions along the tape measured profile t[0 ..

Returns

res – Same as pos but with interpolated height values. The Distance between pos points and res (along curve) points remains.

Return type

ndarray(N, 2)

pygimli.meshtools.**toSubsurface**(*mesh*, *verbose=False*)

Create a subsurface object from pygimli mesh.

Testet objects so far:

Creates Subsurface.TriSurf from 3D triangle boundaries

Parameters

- **mesh** (GIMLI::Mesh) –
- **verbose** (*boolean [False]*) – Be verbose during import.

Return type

Subsurface object depending on input mesh

8.4 pygimli.physics

Module containing submodules for various geophysical methods.

Module overview

em (page 362)	Frequency-domain (FD) or time-domain (TD) semi-analytical 1D solutions
ert (page 367)	Direct current electromagnetics
gravimetry (page 384)	Solve gravimetric and magneto static problems in 2D and 3D analytically
petro (page 390)	Various petrophysical models
seismics (page 395)	Full wave form seismics utilities and simulations
SIP (page 399)	Spectral induced polarization (SIP) measurements and fittings.
sNMR (page 414)	Surface nuclear magnetic resonance (NMR) data inversion
traveltime (page 419)	Refraction seismics or first arrival traveltime calculations.
petro (page 390)	Various petrophysical models

8.4.1 pygimli.physics.em

Frequency-domain (FD) or time-domain (TD) semi-analytical 1D solutions

8.4.1.1 Overview

Functions

importMaxminData (page 363)(filename[, verbose])	Import function reading in positions, data, frequencies, geometry.
readusffile (page 363)(filename[, data])	Read data from single USF (universal sounding file) file.
rhoafromB (page 363)(B, t, Tx[, current])	Apparent resistivity from B-field TEM
rhoafromU (page 363)(U, t, Tx[, current, Rx])	Apparent resistivity curve from classical TEM (U or dB/dt)

Classes

FDEM (page 363)([x, freqs, coilSpacing, inphase, ...])	Class for managing Frequency Domain EM data and their inversions.
HEMmodelling (page 365)(nlay, height[, f, r])	HEM Airborne modelling class based on the BGR RESOLVE system.
TDEM (page 366)([filename])	TEM class mainly for holding data etc.
VMDTimeDomainModelling (page 367)(times, txArea[, rxArea])	Vertical magnetic dipole (VMD) modelling.

8.4.1.2 Functions

pygimli.physics.em.**importMaxminData**(*filename*, *verbose=False*)

Import function reading in positions, data, frequencies, geometry.

pygimli.physics.em.**readusffile**(*filename*, *data=None*)

Read data from single USF (universal sounding file) file.

data = readusffile(filename) data = readusffile(filename, data) will append to data

pygimli.physics.em.**rhoafromB**(*B*, *t*, *Tx*, *current=1*)

Apparent resistivity from B-field TEM

$$\rho_a = ((A_{Tx} * I * \mu_0)/(30B))^2/3 * 4e-7/t$$

pygimli.physics.em.**rhoafromU**(*U*, *t*, *Tx*, *current=1.0*, *Rx=None*)

Apparent resistivity curve from classical TEM (U or dB/dt)

rhoafromU(U/I, t, TXarea[, RXarea])

$$\rho_a = (A_{Rx} * A_{Tx} * \mu_0/20/(U/I))^2/3 * t^{-5/3} * 4e-7$$

8.4.1.3 Classes

class pygimli.physics.em.**FDEM**(*x=None*, *freqs=None*, *coilSpacing=None*, *inphase=None*, *outphase=None*, *filename=None*, *scaleFreeAir=False*)

Bases: object

Class for managing Frequency Domain EM data and their inversions.

FOP(*nlay=2*, *useHEM=1*)

Forward modelling operator using a block discretization.

Parameters

nlay (*int*) – Number of blocks

FOP2d(*nlay*)

2d forward modelling operator.

FOPsmooth(*zvec*)

Forward modelling operator using fixed layers (smooth inversion)

Parameters

zvec (*array*) –

__init__(*x=None*, *freqs=None*, *coilSpacing=None*, *inphase=None*, *outphase=None*, *filename=None*, *scaleFreeAir=False*)

Initialize data class and load data. Provide filename or data.

If filename is given, data is loaded, overwriting settings.

Parameters

- **x** (*array*) – Array of measurement positions
- **freq** (*array*) – Measured frequencies

- **coilSpacing** (*float*) – Distance between 2 two coils
- **inphase** (*array*) – real part of $|amplitude| * \exp^{iphase}$
- **outphase** (*array*) – imaginary part of $|amplitude| * \exp^{iphase}$
- **filename** (*str*) – Filename to read from. Supported: .xyz (MaxMin), *.txt (Emsys)
- **scaleFreeAir** (*bool*) – Scale inphase and outphase data by free air (primary) solution

datavec(*xpos=0*)

Extract data vector (stack in and out phase) for given pos/no.

deactivate(*fr*)

Deactivate a single frequency.

error(*xpos=0*)

Return error as vector.

errorvec(*xpos=0*, *minvalue=0.0*)

Extract error vector for a give position or sounding number.

freq()

Return active (i.e., non-deactivated) frequencies.

importEmsysAsciiData(*filename*)

Import data from emsys text export file.

columns: no, pos(1-3), separation(4), frequency(6), error(8), inphase (9-11), outphase (12-14), reads: positions, data, frequencies, error and geometry

importIPXData(*filename*, *verbose=False*)

Import MaxMin IPX format with pos, data, frequencies & geometry.

importMaxMinData(*filename*, *verbose=False*)

Import MaxMin ASCII export (*.txt) data.

inv2D(*nlay*, *lam=100.0*, *resL=1.0*, *resU=1000.0*, *thkL=1.0*, *thkU=100.0*, *minErr=1.0*)

2d LCI inversion class.

invBlock(*xpos=0*, *nlay=2*, *noise=1.0*, *show=True*, *stmod=30.0*, *lam=1000.0*, *lBound=0.0*, *uBound=0.0*, *verbose=False*, ***kwargs*)

Create and return Gimli inversion instance for block inversion.

Parameters

- **xpos** (*array*) – position vector
- **nLay** (*int*) – Number of layers of the model to be determined OR vector of layer numbers OR forward operator
- **noise** (*float*) – Absolute data err in percent
- **stmod** (*float or pg.Vector*) – Starting model
- **lam** (*float*) – Global regularization parameter lambda.
- **lBound** (*float*) – Lower boundary for the model

- **uBound** (*float*) – Upper boundary for the model. 0 means no upper booundary
- **verbose** (*bool*) – Be verbose

plotAllData(*orientation='horizontal'*, *aspect=1000*, *outname=None*, *show=False*, *figsize=(11, 8)*, *everyx=None*)

Plot data along a profile as image plots for IP and OP.

plotData(*xpos=0*, *response=None*, *error=None*, *ax=None*, *marker='bo-'*, *rmarker='rx-'*, *clf=True*, *addlabel=''*, *nv=2*)

Plot data as curves at given position.

plotDataOld(*xpos=0*, *response=None*, *marker='bo-'*, *rmarker='rx-'*, *clf=True*)

Plot data as curves at given position.

plotModelAndData(*model*, *xpos*, *response*, *modelL=None*, *modelU=None*)

Plot both model and data in subfigures.

readHEMData(*filename*, *takeevery=1*, *choosevcp=True*)

Read RESOLVE type airborne EM data from .XYZ file.

selectData(*xpos=0*)

Select sounding at a specific position or by number.

Retrieve inphase, outphase and error(if exist) vector from index or near given position

Returns
array OP : array ERR : array or None (if no error is specified)

Return type
IP

showModelAndData(*model*, *xpos=0*, *response=None*, *figsize=(8, 6)*)

Show both model and data with response in subfigures.

class pygimli.physics.em.**HEMmodelling**(*nlay*, *height*, *f=None*, *r=None*, ***kwargs*)

Bases: *Modelling* (page 296)

HEM Airborne modelling class based on the BGR RESOLVE system.

__init__(*nlay*, *height*, *f=None*, *r=None*, ***kwargs*)

Initialize class with geometry

Parameters

- **nlay** (*int*) – number of layers
- **height** (*float*) – helicopter
- **f** (*array [BGR RESOLVE system 387Hz-133kHz]*) – frequency vector
- **r** (*array [BGR RESOLVE system 7.91-7.94]*) – distance vector
- **scaling** (*float*) – scaling factor or string (ppm=1e6, percent=1e2)

c0 = 299792251.7596404

calc_forward(*x*, *h*, *rho*, *d*, *epr*, *mur*, *quasistatic=False*)

Calculate forward response.

downward(*rho*, *d*, *z*, *epr*, *mur*, *lam*)

Downward continuation of fields.

ep0 = 8.8542e-12

fdefault = array([387., 1821., 8388., 41460., 133300.])

mu0 = 1.2566370614359173e-06

rdefault = array([7.94, 7.93, 7.93, 7.91, 7.92])

response(*par*)

Compute response vector by pasting in-phase and out-phase data.

scaling = 1000000.0

vmd_hem(*h*, *rho*, *d*, *epr=1.0*, *mur=1.0*, *quasistatic=False*)

Vertical magnetic dipole (VMD) response.

Parameters

- **h** (*float*) – flight height
- **rho** (*array*) – resistivity vector
- **d** (*array*) – thickness vector

vmd_total_Ef(*h*, *z*, *rho*, *d*, *epr*, *mur*, *tm*)

VMD E-phi field (not used actively).

class pygimli.physics.em.**TDEM**(*filename=None*)

Bases: object

TEM class mainly for holding data etc.

__init__(*filename=None*)

Initialize class and (optionally) load data

basename = 'new'

filterData(*token*, *vmin=0*, *vmax=9e+99*)

Filter all sounding data according to criterion.

filterSoundings(*token*, *value*)

Filter all values matching a certain token.

gather(*token*)

Collect item from all soundings.

invert(*nr=0*, *nlay=4*, *thickness=None*, *errorFloor=0.05*)

Do inversion.

load(*filename*)

Road data from usf, txt (siroTEM), tem (TEMfast) or UniK file.

plotRhoa(*ax=None*, *ploterror=False*, *corrramp=False*, ***kwargs*)

Plot all apparent resistivity curves into one window.

plotTransients(*ax=None*, ***kwargs*)

Plot all transients into one window

showInfos()

stackAll(*tmin=0*, *tmax=100*)

Stack all measurements yielding a new TDEM class instance.

class pygimli.physics.em.**VMDTimeDomainModelling**(*times*, *txArea*, *rxArea=None*, ***kwargs*)

Bases: VMDModelling

Vertical magnetic dipole (VMD) modelling.

__init__(*times*, *txArea*, *rxArea=None*, ***kwargs*)

calcEphiT(*tMin*, *tMax*, *rho*, *d*, *rMin*, *rMax*, *z*, *dipm*)

Compute radial electric field.

calcRhoa(*thk*, *res*)

Compute apparent resistivity response

createStartModel(*rhoa*, *nLayers=None*, *thickness=None*)

Create suitable starting model.

Create suitable starting model based on median apparent resistivity values and skin depth approximation.

response(*par*)

par = [thicknesses(nLay), res(nlay + 1)]

response_mt(*par*, *i=0*)

par = [thicknesses, res]

8.4.2 pygimli.physics.ert

Direct current electromagnetics

This package contains tools, modelling operators, and managers for:

- Electrical Resistivity Tomography (ERT) / Induced polarization (IP)
- Vertical Electric Sounding (VES)

8.4.2.1 Overview

Functions

createData (page 369)(elecs[, schemeName])	Utility one-liner to create a BERT datafile
createERTData (page 370)(*args, **kwargs)	
createGeometricFactors (page 370)(*args, **kwargs)	
createInversionMesh (page 370)(data, **kwargs)	Create default mesh for ERT inversion.
drawERTData (page 371)(ax, data[, vals])	Plot ERT data as pseudosection matrix (position over separation).
estimateError (page 371)(data[, absoluteError, ...])	Estimate error composed of an absolute and a relative part.
generateDataPDF (page 372)(data[, filename])	Generate a multi-page pdf showing all data properties.
geometricFactor (page 372)	geometricFactors((object)data [, (object)dim=3 [, (object)forceFlatEarth=False]]) -> object :
geometricFactors (page 372)(data [[, dim, forceFlatEarth])	Helper function to calculate configuration factors for a given DataContainerERT
load (page 372)(fileName[, verbose])	Shortcut to load ERT data.
show (page 372)(data[, vals])	Plot ERT data as pseudosection matrix (position over separation).
showData (page 373)(data[, vals])	Plot ERT data as pseudosection matrix (position over separation).
showERTData (page 373)(data[, vals])	Plot ERT data as pseudosection matrix (position over separation).
simulate (page 373)(mesh, scheme, res, **kwargs)	Simulate an ERT measurement.

Classes

DataContainer (page 375)	alias of DataContainerERT
ERTManager (page 375)([data])	ERT Manager.
ERTModelling (page 378)([sr, verbose])	Forward operator for Electrical Resistivity Tomography.
ERTModellingReference (page 379)(**kwargs)	Reference implementation for 2.5D Electrical Resistivity Tomography.
Manager (page 380)	alias of ERTManager (page 375)
VESCModelling (page 380)(**kwargs)	Vertical Electrical Sounding (VES) forward operator.
VESManager (page 381)(**kwargs)	Vertical electrical sounding (VES) manager class.
VESModelling (page 382)([ab2, mn2])	Vertical Electrical Sounding (VES) forward operator.

8.4.2.2 Functions

pygimli.physics.ert.**createData**(*elecs*, *schemeName='none'*, ***kwargs*)

Utility one-liner to create a BERT datafile

Parameters

- **elecs** (*int | list[pos] | array(x)*) – Number of electrodes or electrode positions or x-positions
- **schemeName** (*str ['none']*) – Name of the configuration. If you provide an unknown scheme name, all known schemes [‘wa’, ‘wb’, ‘pp’, ‘pd’, ‘dd’, ‘slm’, ‘hw’, ‘gr’] listed.
- ****kwargs** – Arguments that will be forwarded to the scheme generator.
  - **inverse**
    [bool] interchange AB MN with MN AB
  - **reciprocity**
    [bool] interchange AB MN with BA NM
  - **addInverse**
    [bool] add additional inverse measurements
  - **spacing**
    [float [1]] electrode spacing in meters
  - **closed**
    [bool] Close the chain. Measure from the end of the array to the first electrode.

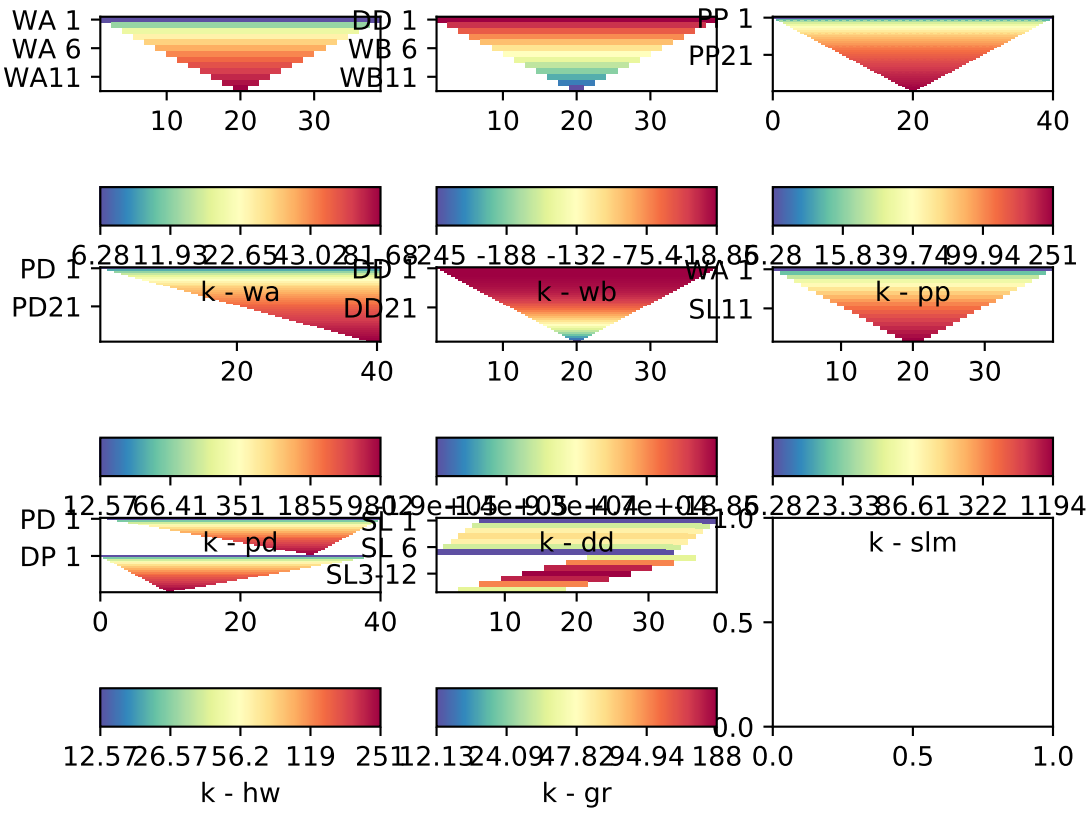
Returns
data

Return type
DataContainerERT

```
>>> import matplotlib.pyplot as plt
>>> from pygimli.physics import ert
>>>
>>> schemes = ['wa', 'wb', 'pp', 'pd', 'dd', 'slm', 'hw', 'gr']
>>> fig, ax = plt.subplots(3,3)
>>>
>>> for i, schemeName in enumerate(schemes):
...     s = ert.createData(elecs=41, schemeName=schemeName)
...     k = ert.geometricFactors(s)
...     _ = ert.show(s, vals=k, ax=ax.flat[i], label='k - ' + schemeName)
>>>
>>> plt.show()
```

Examples using pygimli.physics.ert.createData

- *2D ERT modeling and inversion* (page 60)
- *2D FEM modelling on two-layer example* (page 73)
- *Complex-valued electrical modeling* (page 98)



- *Naive complex-valued electrical inversion* (page 104)

pygimli.physics.ert.**createERTData**(**args*, ***kwargs*)

Examples using `pygimli.physics.ert.createERTData`

- *Hydrogeophysical modeling* (page 138)
- *Petrophysical joint inversion* (page 153)

pygimli.physics.ert.**createGeometricFactors**(**args*, ***kwargs*)

Examples using `pygimli.physics.ert.createGeometricFactors`

- *ERT field data with topography* (page 68)
- *Four-point sensitivities* (page 81)
- *Naive complex-valued electrical inversion* (page 104)

pygimli.physics.ert.**createInversionMesh**(*data*, ***kwargs*)

Create default mesh for ERT inversion.

Parameters

data (GIMLI::DataContainerERT) – Data Container needs at least sensors to define the geometry of the mesh.

:param Forwarded to *pygimli.meshtools.createParaMesh* (page 330):

Returns

mesh – Inversion mesh with default marker (1 for background, 2 parametric domain)

Return type
GIMLI::Mesh

pygimli.physics.ert.**drawERTData**(*ax*, *data*, *vals=None*, ***kwargs*)

Plot ERT data as pseudosection matrix (position over separation).

Parameters

- **data** (*DataContainerERT*) – data container with sensorPositions and a/b/m/n fields
- **vals** (*iterable of data.size() [data('rhoa')]*) – vector containing the vals to show
- **ax** (*mpl.axis*) – axis to plot, if not given a new figure is created
- **cMin/cMax** (*float*) – minimum/maximum color vals
- **logScale** (*bool*) – logarithmic colour scale [min(A)>0]
- **label** (*string*) – colorbar label
- ****kwargs** –
  - **dx**
    [float] x-width of individual rectangles
  - **ind**
    [integer iterable or IVector] indices to limit display
  - **circular**
    [bool] Plot in polar coordinates when plotting via patchValMap

Returns

- *ax* – The used Axes
- *cbar* – The used Colorbar or None

pygimli.physics.ert.**estimateError**(*data*, *absoluteError=0.001*, *relativeError=0.03*, *absoluteUError=None*, *absoluteCurrent=0.1*)

Estimate error composed of an absolute and a relative part.

Parameters

- **absoluteError** (*float [0.001]*) – Absolute data error in Ohm m. Need 'rhoa' values in data.
- **relativeError** (*float [0.03]*) – relative error level in %/100
- **absoluteUError** (*float [0.001]*) – Absolute potential error in V. Need 'u' values in data. Or calculate them from 'rhoa', 'k' and absoluteCurrent if no 'i' is given
- **absoluteCurrent** (*float [0.1]*) – Current level in A for reconstruction for absolute potential V

Returns
error

Return type
Array

Examples using `pygimli.physics.ert.estimateError`

- *ERT field data with topography* (page 68)
- *Incorporating prior data into ERT inversion* (page 162)
- *Region-wise regularization* (page 262)

`pygimli.physics.ert.`**`generateDataPDF`**(*data*, *filename='data.pdf'*)

Generate a multi-page pdf showing all data properties.

`pygimli.physics.ert.`**`geometricFactor`**()

geometricFactors((object)data [, (object)dim=3 [, (object)forceFlatEarth=False]]) -> object :

Helper function to calculate configuration factors for a given DataContainerERT

C++ signature :

GIMLI::Vector<double> geometricFactors(GIMLI::DataContainerERT [,int=3 [,bool=False]])

`pygimli.physics.ert.`**`geometricFactors`**(*(object)data*[, *(object)dim=3*[, *(object)forceFlatEarth=False*]]) → object :

Helper function to calculate configuration factors for a given DataContainerERT

C++ signature :

GIMLI::Vector<double> geometricFactors(GIMLI::DataContainerERT [,int=3 [,bool=False]])

Examples using `pygimli.physics.ert.geometricFactors`

- *Incorporating prior data into ERT inversion* (page 162)
- *Region-wise regularization* (page 262)

`pygimli.physics.ert.`**`load`**(*fileName*, *verbose=False*, ***kwargs*)

Shortcut to load ERT data.

Import Data and try to assume the file format. Additionally to unified data format we support the wide-spread res2dinv format as well as ASCII column files generated by the processing software of various instruments (ABEM LS, Syscal Pro, Resecs, ?)

If this fails, install pybert and use its auto importer pybert.importData.

Parameters

`fileName` (*str*) –

Returns

data

Return type

pg.DataContainer

`pygimli.physics.ert.`**`show`**(*data*, *vals=None*, ***kwargs*)

Plot ERT data as pseudosection matrix (position over separation).

Creates figure, axis and draw a pseudosection.

Parameters

- **`data`** (BERT::DataContainerERT) –

- ****kwargs** –
  - **axes**
    [matplotlib.axes] Axes to plot into. Default is None and a new figure and axes are created.
  - **vals**
    [Array[nData]] Values to be plotted. Default is data('rhoa').

Examples using `pygimli.physics.ert.show`

- *2D ERT modeling and inversion* (page 60)
- *ERT field data with topography* (page 68)
- *Incorporating prior data into ERT inversion* (page 162)
- *Region-wise regularization* (page 262)

`pygimli.physics.ert.`**`showData`**(*data*, *vals=None*, ***kwargs*)

Plot ERT data as pseudosection matrix (position over separation).

Creates figure, axis and draw a pseudosection.

Parameters

- **data** (BERT::DataContainerERT) –
- ****kwargs** –
  - **axes**
    [matplotlib.axes] Axes to plot into. Default is None and a new figure and axes are created.
  - **vals**
    [Array[nData]] Values to be plotted. Default is data('rhoa').

Examples using `pygimli.physics.ert.showData`

- *ERT field data with topography* (page 68)

`pygimli.physics.ert.`**`showERTData`**(*data*, *vals=None*, ***kwargs*)

Plot ERT data as pseudosection matrix (position over separation).

Creates figure, axis and draw a pseudosection.

Parameters

- **data** (BERT::DataContainerERT) –
- ****kwargs** –
  - **axes**
    [matplotlib.axes] Axes to plot into. Default is None and a new figure and axes are created.
  - **vals**
    [Array[nData]] Values to be plotted. Default is data('rhoa').

Examples using `pygimli.physics.ert.showERTData`

- *Complex-valued electrical modeling* (page 98)
- *Naive complex-valued electrical inversion* (page 104)

pygimli.physics.ert.**simulate**(*mesh*, *scheme*, *res*, ***kwargs*)

Simulate an ERT measurement.

Perform the forward task for a given mesh, resistivity distribution & measuring scheme and return data (apparent resistivity) or potentials.

For complex resistivity, the apparent resistivities is complex as well.

The forward operator itself only calculates potential values for the electrodes in the given data scheme. To calculate apparent resistivities, geometric factors (k) are needed. If there are no values k in the DataContainerERT scheme, the function tries to calculate them, either analytically or numerically by using a p2-refined version of the given mesh.

Parameters

- **mesh** (GIMLI::Mesh) – 2D or 3D Mesh to calculate for.
- **res** (*float, array(mesh.cellCount()) | array(N, mesh.cellCount()) |*) – list Resistivity distribution for the given mesh cells can be: . float for homogeneous resistivity (e.g. 1.0) . single array of length mesh.cellCount() . matrix of N resistivity distributions of length mesh.cellCount() . resistivity map as [[regionMarker0, res0],

  [regionMarker0, res1], . . .]
- **scheme** (GIMLI::DataContainerERT) – Data measurement scheme.

Keyword Arguments

- **verbose** (*bool [False]*) – Be verbose. Will override class settings.
- **calcOnly** (*bool [False]*) – Use fop.calculate instead of fop.response. Useful if you want to force the calculation of impedances for homogeneous models. No noise handling. Solution is put as token 'u' in the returned DataContainerERT.
- **noiseLevel** (*float [0.0]*) – add normally distributed noise based on scheme['err'] or on noiseLevel if error>0 is not contained
- **noiseAbs** (*float [0.0]*) – Absolute voltage error in V
- **returnArray** (*bool [False]*) – Returns an array of apparent resistivities instead of a DataContainerERT
- **returnFields** (*bool [False]*) – Returns a matrix of all potential values (per mesh nodes) for each injection electrodes.

Returns

- *DataContainerERT | array(data.size()) | array(N, data.size()) |*
- *array(N, mesh.nodeCount())* – Data container with resulting apparent resistivity data and errors (if noiseLevel or noiseAbs is set). Optional returns a Matrix of rhoa values (for returnArray==True forces noiseLevel=0). In case of a complex valued resistivity model, phase values are returned in the DataContainerERT (see example below), or as an additionally returned array.

Examples

>>> from pygimli.physics import ert # >>> import pygimli as pg # >>> import pygimli.meshtools as mt # >>> world = mt.createWorld(start=[-50, 0], end=[50, -50], # ... layers=[-1, -5], worldMarker=True) # >>> scheme = ert.createData(# ... elecs=pg.utils.grange(start=-10, end=10, n=21), # ... schemeName='dd') # >>> for pos in scheme.sensorPositions(): # ... _= world.createNode(pos) # ... _= world.createNode(pos + [0.0, -0.1]) # >>> mesh = mt.createMesh(world, quality=34) # >>> rhomap = [# ... [1, 100. + 0j], # ... [2, 50. + 0j], # ... [3, 10.+ 0j], # ...] # >>> ert = pg.ERTManager() # >>> data = ert.simulate(mesh, res=rhomap, scheme=scheme, verbose=True) # >>> rhoa = data.get('rhoa').array() # >>> phia = data.get('phia').array()

Examples using `pygimli.physics.ert.simulate`

- *2D ERT modeling and inversion* (page 60)
- *2D FEM modelling on two-layer example* (page 73)
- *3D modeling in a closed geometry* (page 85)
- *Complex-valued electrical modeling* (page 98)
- *Naive complex-valued electrical inversion* (page 104)
- *Petrophysical joint inversion* (page 153)

8.4.2.3 Classes

`pygimli.physics.ert.`**`DataContainer`**

alias of `DataContainerERT`

class `pygimli.physics.ert.`**`ERTManager`**(*data=None*, ***kwargs*)

Bases: *`MeshMethodManager`* (page 290)

ERT Manager.

Method Manager for Electrical Resistivity Tomography (ERT)

`__init__`(*data=None*, ***kwargs*)

Create ERT Manager instance.

Parameters

- **`data`** (GIMLI::DataContainerERT | str) – You can initialize the Manager with data or give them a dataset when calling the inversion.
- **`useBert`** (*⋆*) – Use Bert forward operator instead of the reference implementation.
- **`sr`** (*⋆*) – Calculate with singularity removal technique. Recommended but needs the primary potential. For flat earth cases the primary potential will be calculated analytical. For domains with topography the primary potential will be calculated numerical using a p2 refined mesh or you provide primary potentials with setPrimPot.

checkData(*data=None*)

Return data from container.

THINKABOUT: Data will be changed, or should the manager keep a copy?

checkErrors(*err*, *dataVals*)

Check (estimate) and return relative error.

By default we assume 'err' are relative values.

coverage()

Coverage vector considering the logarithmic transformation.

createForwardOperator(***kwargs*)

Create and choose forward operator.

createMesh(*data=None*, ***kwargs*)

Create default inversion mesh.

Forwarded to *pygimli.physics.ert.createInversionMesh* (page 370)

estimateError(*data=None*, ***kwargs*)

Estimate error composed of an absolute and a relative part.

Parameters

- **absoluteError** (*float [0.001]*) – Absolute data error in Ohm m. Need 'rhoa' values in data.
- **relativeError** (*float [0.03]*) – relative error level in %/100
- **absoluteUError** (*float [0.001]*) – Absolute potential error in V. Need 'u' values in data. Or calculate them from 'rhoa', 'k' and absoluteCurrent if no 'i' is given
- **absoluteCurrent** (*float [0.1]*) – Current level in A for reconstruction for absolute potential V

Returns
error

Return type
Array

load(*fileName*)

Load ERT data.

Forwarded to *pygimli.physics.ert.load* (page 372)

Parameters
fileName (*str*) – Filename for the data.

Returns
data

Return type
GIMLI::DataContainerERT

saveResult (*folder=None*, *size=(16, 10)*, ***kwargs*)

Save all results in the specified folder.

Saved items are:

Inverted profile Resistivity vector Coverage vector Standardized coverage vector Mesh (bms and vtk with results)

setPrimPot (*pot*)

Set primary potential from external is not supported anymore.

setSingularityRemoval (*sr=True*)

Turn singularity removal on or off.

showMisfit (*errorWeighted=False*, ***kwargs*)

Show relative or error-weighted data misfit.

showModel (*model=None*, *elecs=True*, *ax=None*, ***kwargs*)

Show the last inversion result.

Parameters

- **ax** (*mpl axes*) – Axes object to draw into. Create a new if its not given.
- **model** (*iterable [None]*) – Model values to be draw. Default is self.model from the last run

Return type

ax, cbar

simulate (*mesh*, *scheme*, *res*, ***kwargs*)

Simulate an ERT measurement.

Perform the forward task for a given mesh, resistivity distribution & measuring scheme and return data (apparent resistivity) or potentials.

For complex resistivity, the apparent resistivities is complex as well.

The forward operator itself only calculates potential values for the electrodes in the given data scheme. To calculate apparent resistivities, geometric factors (k) are needed. If there are no values k in the DataContainerERT scheme, the function tries to calculate them, either analytically or numerically by using a p2-refined version of the given mesh.

Parameters

- **mesh** (GIMLI::Mesh) – 2D or 3D Mesh to calculate for.
- **res** (*float, array(mesh.cellCount()) | array(N, mesh.cellCount()) |*) – list Resistivity distribution for the given mesh cells can be: . float for homogeneous resistivity (e.g. 1.0) . single array of length mesh.cellCount() . matrix of N resistivity distributions of length mesh.cellCount() . resistivity map as [[regionMarker0, res0], [regionMarker0, res1], . . .]
- **scheme** (GIMLI::DataContainerERT) – Data measurement scheme.

Keyword Arguments

- **verbose** (*bool[False]*) – Be verbose. Will override class settings.

- **calcOnly** (*bool [False]*) – Use fop.calculate instead of fop.response. Useful if you want to force the calculation of impedances for homogeneous models. No noise handling. Solution is put as token 'u' in the returned DataContainerERT.
- **noiseLevel** (*float [0.0]*) – add normally distributed noise based on scheme['err'] or on noiseLevel if error>0 is not contained
- **noiseAbs** (*float [0.0]*) – Absolute voltage error in V
- **returnArray** (*bool [False]*) – Returns an array of apparent resistivities instead of a DataContainerERT
- **returnFields** (*bool [False]*) – Returns a matrix of all potential values (per mesh nodes) for each injection electrodes.

Returns

- *DataContainerERT* | *array(data.size())* | *array(N, data.size())* |
- *array(N, mesh.nodeCount())* – Data container with resulting apparent resistivity data and errors (if noiseLevel or noiseAbs is set). Optional returns a Matrix of rhoa values (for returnArray==True forces noiseLevel=0). In case of a complex valued resistivity model, phase values are returned in the DataContainerERT (see example below), or as an additionally returned array.

Examples

>>> from pygimli.physics import ert # >>> import pygimli as pg # >>> import pygimli.meshtools as mt # >>> world = mt.createWorld(start=[-50, 0], end=[50, -50], # … layers=[-1, -5], worldMarker=True) # >>> scheme = ert.createData(# … elecs=pg.utils.grange(start=-10, end=10, n=21), # … schemeName='dd') # >>> for pos in scheme.sensorPositions(): # … _= world.createNode(pos) # … _= world.createNode(pos + [0.0, -0.1]) # >>> mesh = mt.createMesh(world, quality=34) # >>> rhomap = [# … [1, 100. + 0j], # … [2, 50. + 0j], # … [3, 10.+ 0j], # …] # >>> data = ert.simulate(mesh, res=rhomap, scheme=scheme, verbose=1) # >>> rhoa = data.get('rhoa').array() # >>> phia = data.get('phia').array()

standardizedCoverage(*threshold=0.01*)

Return standardized coverage vector (0|1) using thresholding.

class pygimli.physics.ert.**ERTModelling**(*sr=True*, *verbose=False*)

Bases: ERTModellingBase

Forward operator for Electrical Resistivity Tomography.

Note: Convention for complex resistiviy inversion: We want to use logarithm transformation for the imaginary part of model so we need the startmodel to have positive imaginary parts. The sign is flipped back to physical correct assumption before we call the response function. The Jacobian is calculated with negative imaginary parts and will be a conjugated complex block matrix for further calulations.

__init__(*sr=True*, *verbose=False*)

Variables

- **fop** (*pg.frameworks.Modelling*) –
- **data** (*pg.DataContainer*) –
- **modelTrans** (*[pg.trans.TransLog()]*) –

Parameters
****kwargs** – fop : Modelling

createJacobian(*mod*)
Compute Jacobian matrix and store but not return.

createStartModel(*dataVals*)
Create Starting model for ERT inversion.

flipImagPart(*v*)
Flip imaginary port (convention).

response(*mod*)
Forward response (apparent resistivity).

setDataPost(*data*)

setDefaultBackground()
Set the default background behaviour.

setMeshPost(*mesh*)

setVerbose(*(object)arg1*, *(object)verbose*) → object :
Set verbose state.

C++ signature :
void* setVerbose(GIMLI::ModellingBase {lvalue},bool)

class pygimli.physics.ert.**ERTModellingReference**(***kwargs*)
Bases: ERTModellingBase

Reference implementation for 2.5D Electrical Resistivity Tomography.

__init__(***kwargs*)

Variables

- **fop** (*pg.frameworks.Modelling*) –
- **data** (*pg.DataContainer*) –
- **modelTrans** (*[pg.trans.TransLog()]*) –

Parameters
****kwargs** – fop : Modelling

calcGeometricFactor(*data*)
Calculate geometry factors for a given dataset.

createJacobian(*model*)
TODO WRITEME.

createRHS(*mesh*, *elecs*)
Create right-hand-side vector.

getIntegrationWeights(*rMin*, *rMax*)

TODO WRITEME.

mixedBC(*boundary*, *userData*)

Apply mixed boundary conditions.

pointSource(*cell*, *f*, *userData*)

Define function for the current source term.

$\delta(x-pos), \int f(x)\delta(x-pos) = f(pos) = N(pos)$

Right hand side entries will be shape functions(pos)

response(*model*)

Solve forward task and return apparent resistivity for self.mesh.

uAnalytical(*p*, *sourcePos*, *k*)

Calculate analytical potential for homogeneous halfspace.

For sigma = 1 [S m]

pygimli.physics.ert.**Manager**

alias of *ERTManager* (page 375)

class pygimli.physics.ert.**VESCModelling**(***kwargs*)

Bases: *VESModelling* (page 382)

Vertical Electrical Sounding (VES) forward operator. (complex)

Vertical Electrical Sounding (VES) forward operator for complex resistivity values. see: *pygimli.physics.ert.VESModelling* (page 382)

__init__(***kwargs*)

Constructor

createStartModel(*rhoa*)

drawData(*ax*, *data*, *error=None*, *labels=None*, *ab2=None*, *mn2=None*, ***kwargs*)

Draw modeled apparent resistivity and apparent phase data.

Parameters

- **ax** (*axes*) – Matplotlib axes object to draw into.
- **data** (*iterable*) – Apparent resistivity values to draw. [rhoa phia].
- **error** (*iterable [None]*) – Rhoa in Ohm m and phia in radiand. Adds an error bar if you have error values. [err_rhoas err_phia] The error of amplitudes are assumed to be relative and the error of the phases is assumed to be absolute in mrad.
- **labels** (*str [r'$varrho_a$', r'$varphi_a$']*) – Set legend labels for amplitude and phase.
- **parameters** (*Other*) –
- **----------** –
- **ab2** (*iterable*) – Override ab2 that fits data size.
- **mn2** (*iterable*) – Override mn2 that fits data size.

- **plot** (*function name*) – Matplotlib plot function, e.g., plot, loglog, semilogx or semilogy

drawModel(*ax*, *model*, ***kwargs*)

Draw 1D VESC Modell.

phaseModel(*model*)

Return the current phase model values.

resModel(*model*)

Return the resistivity model values.

response_mt(*par*, *i=0*)

Multithread response for parametrization.

Returns [|rhoal, +phi(rad)] for [thicks, res, phi(rad)]

class pygimli.physics.ert.**VESManager**(***kwargs*)

Bases: *MethodManager1d* (page 296)

Vertical electrical sounding (VES) manager class.

```
>>> import numpy as np
>>> import pygimli as pg
>>> from pygimli.physics import VESManager
>>> ab2 = np.logspace(np.log10(1.5), np.log10(100), 32)
>>> mn2 = 1.0
>>> # 3 layer with 100, 500 and 20 Ohmm
>>> # and layer thickness of 4, 6, 10 m
>>> # over a Halfspace of 800 Ohmm
>>> synthModel = pg.cat([4., 6., 10.], [100., 5., 20., 800.])
>>> ves = VESManager()
>>> ra, err = ves.simulate(synthModel, ab2=ab2, mn2=mn2, noiseLevel=0.01)
>>> ax = ves.showData(ra, error=err)
>>> # _= ves.invert(ra, err, nLayer=4, showProgress=0,␣
↪verbose=0)
>>> # ax = ves.showModel(synthModel)
>>> # ax = ves.showResult(ax=ax)
>>> pg.wait()
```

__init__(***kwargs*)

Constructor

Parameters

complex (*bool*) – Accept complex resistivities.

Variables

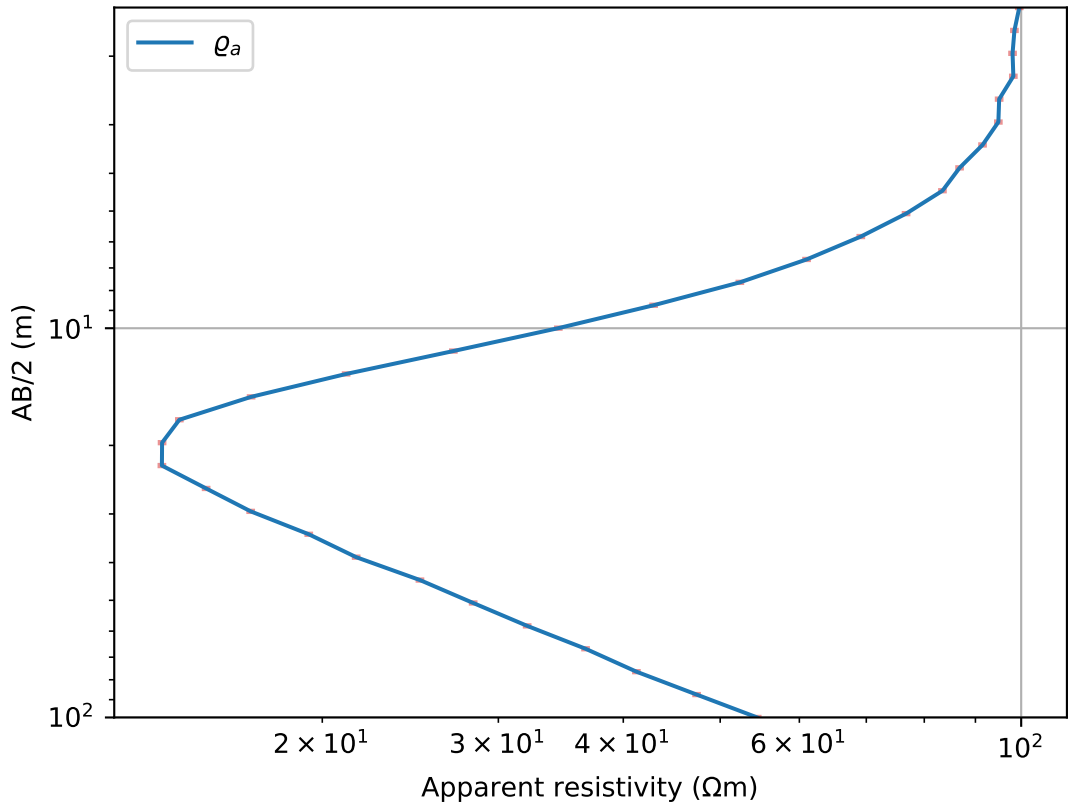
complex (*bool*) – Accept complex resistivities.

property complex

createForwardOperator(***kwargs*)

Create Forward Operator.

Create Forward Operator based on complex attribute.



exportData(*fileName*, *data=None*, *error=None*)

Export data into simple ascii matrix.

Usefull?

invert(*data=None*, *err=None*, *ab2=None*, *mn2=None*, ***kwargs*)

Invert measured data.

Keyword Arguments

****kwargs** – Additional kwargs inherited from %(MethodManager1d.invert) and %(Inversion.run)

Returns

model – inversion result

Return type

pg.Vector

loadData(*fileName*, ***kwargs*)

Load simple data matrix

preErrorCheck(*err*, *dataVals=None*)

Called before the validity check of the error values.

simulate(*model*, *ab2=None*, *mn2=None*, ***kwargs*)

Simulate measurement data.

class pygimli.physics.ert.**VESModelling**(*ab2=None*, *mn2=None*, ***kwargs*)

Bases: *Block1DModelling* (page 283)

Vertical Electrical Sounding (VES) forward operator.

Variables

- **am** – Part of data basis. Distances between A and M electrodes. A is first power, M is first potential electrode.
- **bm** – Part of data basis. Distances between B and M electrodes. B is second power, M is first potential electrode.
- **an** – Part of data basis. Distances between A and N electrodes. A is first power, N is second potential electrode.
- **bn** – Part of data basis. Distances between B and N electrodes. B is second power, N is second potential electrode.
- **ab2** – Half distance between the current electrodes A and B.
- **mn2** – Half distance between the potential electrodes M and N. Only used for input (feeding am etc.).

__init__(*ab2=None*, *mn2=None*, ***kwargs*)

Constructor

createStartModel(*rhoa*)

drawData(*ax*, *data*, *error=None*, *label=None*, ***kwargs*)

Draw modeled apparent resistivity data.

Parameters

- **ax** (*axes*) – Matplotlib axes object to draw into.
- **data** (*iterable*) – Apparent resistivity values to draw.
- **error** (*iterable [None]*) – Adds an error bar if you have error values.
- **label** (*str ['$varrho_a$']*) – Set legend label for the amplitude.
- **ab2** (*iterable*) – Override ab2 that fits data size.
- **mn2** (*iterable*) – Override mn2 that fits data size.
- **plot** (*function name*) – Matplotlib plot function, e.g., plot, loglog, semilogx or semilogy

drawModel(*ax*, *model*, ***kwargs*)

response(*(object)arg1*, *(object)model*) → object :

C++ signature :
GIMLI::Vector<double> response(GIMLI::ModellingBase {lvalue},GIMLI::Vector<double>)

response((object)arg1, (object)model) -> object :

C++ signature :
GIMLI::Vector<double> response(ModellingBase_wrapper {lvalue},GIMLI::Vector<double>)

response_mt(*(object)arg1*, *(object)model*[, *(object)i=0*]) → object :

C++ signature :
GIMLI::Vector<double> response_mt(GIMLI::ModellingBase {lvalue},GIMLI::Vector<double> [,unsigned long=0])

response_mt((object)arg1, (object)model [, (object)i=0]) -> object :

C++ signature :
GIMLI::Vector<double> response_mt(ModellingBase_wrapper {lvalue},GIMLI::Vector<double> [,unsigned long=0])

setDataSpace(*ab2=None*, *mn2=None*, *am=None*, *bm=None*, *an=None*, *bn=None*, ***kwargs*)

Set data basis, i.e., arrays for all am, an, bm, bn distances.

8.4.3 pygimli.physics.gravimetry

Solve gravimetric and magneto static problems in 2D and 3D analytically

8.4.3.1 Overview

Functions

BZPoly (page 385)(pnts, poly, mag[, openPoly])	TODO WRITEME.
BaZCylinderHoriz (page 385)(pnts, R, pos, M)	Magnetic anomaly for a horizontal cylinder.
BaZSphere (page 385)(pnts, R, pos, M)	Magnetic anomaly for a sphere.
SolveGravMagHolstein (page 385)(mesh, pnts, cmp, igrf)	Solve gravity and/or magnetics problem after Holstein (1997).
gradGZCylinderHoriz (page 386)(r, a, rho[, pos])	TODO WRITEME.
gradGZHalfPlateHoriz (page 386)(pnts, t, rho[, pos])	TODO WRITEME.
gradGZSphere (page 386)(r, rad, rho[, pos])	TODO WRITEME.
gradUCylinderHoriz (page 387)(r, a, rho[, pos])	2D Gradient of gravimetric potential of horizontal cylinder (in mGal at position *pos*).
gradUHalfPlateHoriz (page 387)(pnts, t, rho[, pos])	Gravitational field od a horizontal half plate.
gradUSphere (page 388)(r, rad, rho[, pos])	Gravitational field of a sphere.
solveGravimetry (page 388)(mesh[, dDensity, pnts, complete])	Solve gravimetric response.
uCylinderHoriz (page 388)(pnts, rad, rho[, pos])	Gravitational potential of horizonzal cylinder.
uSphere (page 388)(r, rad, rho[, pos])	Gravitational potential of a sphere.

Classes

MagneticsModelling (page 389)(mesh, points, cmp, igrf)	Magnetics modelling operator using Holstein (2007).

8.4.3.2 Functions

pygimli.physics.gravimetry.**BZPoly**(*pnts*, *poly*, *mag*, *openPoly=False*)

TODO WRITEME.

Parameters

- **pnts** (*list*) – Measurement points [[p1x, p1z], [p2x, p2z],. . .]
- **poly** (*list*) – Polygon [[p1x, p1z], [p2x, p2z],. . .]
- **mag** (*[M_x, M_y, M_z]*) – Magnetization = [M_x, M_y, M_z]

pygimli.physics.gravimetry.**BaZCylinderHoriz**(*pnts*, *R*, *pos*, *M*)

Magnetic anomaly for a horizontal cylinder.

Calculate the vertical component of the anomalous magnetic field Bz for a buried horizontal cylinder at position pos with radius R for a given magnetization M at measurement points pnts.

TODO .. only 2D atm

Parameters

- **pnts** (*[[x,z],]*) – measurement points – array[x,y,z]
- **R** (*float*) – radius
- **pos** (*[float, float]*) – [x,z] – sphere center
- **M** (*[float, float]*) – [Mx, Mz] – magnetization

pygimli.physics.gravimetry.**BaZSphere**(*pnts*, *R*, *pos*, *M*)

Magnetic anomaly for a sphere.

Calculate the vertical component of the anomalous magnetic field Bz for a buried sphere at position pos with radius R for a given magnetization M at measurement points pnts.

Parameters

- **pnts** (*[[x,y,z],]*) – measurement points – array[x,y,z]
- **R** (*float*) – radius
- **pos** (*[float, float, float]*) – [x,y,z] – sphere center
- **M** (*[float, float, float]*) – [Mx, My, Mz] – magnetization

pygimli.physics.gravimetry.**SolveGravMagHolstein**(*mesh*, *pnts*, *cmp*, *igrf*, *foot=inf*)

Solve gravity and/or magnetics problem after Holstein (1997).

Parameters

- **mesh** (*pygimli:mesh*) – tetrahedral or hexahedral mesh
- **pnts** (*list|array of (x, y, z)*) – measuring points

- **cmp** (*list of str*) – component list of: gx, gy, gz, TFA, Bx, By, Bz, Bxy, Bxz, Byy, Byz, Bzz
- **igrf** (*list|array of size 3 or 7*) – international geomagnetic reference field, either [D, I, H, X, Y, Z, F] - declination, inclination, horizontal field,

  X/Y/Z components, total field OR

  [X, Y, Z] - X/Y/Z components

Returns
out – kernel matrix to be multiplied with density or susceptibility

Return type
ndarray (nPoints x nComponents x nCells)

pygimli.physics.gravimetry.**gradGZCylinderHoriz**(*r*, *a*, *rho*, *pos=(0.0, 0.0)*)

TODO WRITEME.

$$g = -gradu(r), withr = [x,z], |r| = \sqrt{x^2+z^2}$$

Parameters

- **r** (*list[[x, z]]*) – Observation positions
- **a** (*float*) – Cylinder radius in [meter]
- **rho** – Density in [kg/m^3]

Return type
grad gz, [gz_x, gz_z]

Examples using pygimli.physics.gravimetry.gradGZCylinderHoriz

- *Semianalytical Gravimetry and Geomagnetics in 2D* (page 118)

pygimli.physics.gravimetry.**gradGZHalfPlateHoriz**(*pnts*, *t*, *rho*, *pos=(0.0, 0.0)*)

TODO WRITEME.

$$g = -\nabla u$$

Parameters

- **pnts** (array $(n \times 2)$) – n 2 dimensional measurement points
- **t** (*float*) – Plate thickness in [m]
- **rho** (*float*) – Density in $[\mathrm{kg/m^3}]$

Returns
gz – Gradient of z-component of g $\nabla(\frac{\partial u}{\partial \vec{r}_z})$

Return type
array

Examples using pygimli.physics.gravimetry.gradGZHalfPlateHoriz

- *Semianalytical Gravimetry and Geomagnetics in 2D* (page 118)

pygimli.physics.gravimetry.**gradGZSphere**(*r*, *rad*, *rho*, *pos=(0.0, 0.0, 0.0)*)

TODO WRITEME.

$$g = -\nabla u$$

Parameters

- **r** (*[float, float, float]*) – position vector
- **rad** (*float*) – radius of the sphere
- **rho** (*float*) – density in [kg/m^3]

Return type
[d g_z /dx, d g_z /dy, d g_z /dz]

pygimli.physics.gravimetry.**gradUCylinderHoriz**(*r*, *a*, *rho*, *pos=(0.0, 0.0)*)

2D Gradient of gravimetric potential of horizontal cylinder (in mGal at position *pos*).

$$g = -G[m^3/(kgs^2)] * dM[kg/m] * 1/r[1/m] * grad(r)[1/1] = [m^3/(kgs^2)] * [kg/m] * 1/m * [1/1] == m/s^2$$

Parameters

- **r** (*list[[x, z]]*) – Observation positions
- **a** (*float*) – Cylinder radius in [meter]
- **pos** (*[x,z]*) – Center position of cylinder.
- **rho** (*float*) – Delta density in [kg/m^3]

Returns
g – Gradient of gravimetry potential.

Return type
[dudx, dudz]

Examples using pygimli.physics.gravimetry.gradUCylinderHoriz

- *Gravimetry in 2D - Part I* (page 116)
- *Semianalytical Gravimetry and Geomagnetics in 2D* (page 118)

pygimli.physics.gravimetry.**gradUHalfPlateHoriz**(*pnts*, *t*, *rho*, *pos=(0.0, 0.0)*)

Gravitational field od a horizontal half plate.

$$g = -gradu,$$

Parameters

- **pnts** –
- **t** –
- **rho** – Density in [kg/m^3]

Returns
z-component of g .. math:: nabla(partial u/partialvec{r})_z

Return type
gz

Examples using `pygimli.physics.gravimetry.gradUHalfPlateHoriz`

- *Semianalytical Gravimetry and Geomagnetics in 2D* (page 118)

`pygimli.physics.gravimetry.`**`gradUSphere`**(*r*, *rad*, *rho*, *pos=(0.0, 0.0, 0.0)*)

Gravitational field of a sphere.

$$g = -G[m^3/(kgs^2)] * dM[kg] * 1/r^2 1/m^2] * \nabla(r)[1/1] = [m^3/(kgs^2)] * [kg] * 1/m^2 * [1/1] == m/s^2$$

Parameters

- **r** (*[float, float, float]*) – position vector
- **rad** (*float*) – radius of the sphere
- **rho** (*float*) – density in [kg/m^3]

Returns

[gx, gy, gz] – gravitational acceleration (note that gz points negative)

Return type

[float*3]

`pygimli.physics.gravimetry.`**`solveGravimetry`**(*mesh*, *dDensity=None*, *pnts=None*, *complete=False*)

Solve gravimetric response.

2D with `pygimli.physics.gravimetry.lineIntegralZ_WonBevis`

3D with `pygimli.physics.gravimetry.gravMagBoundarySinghGup`

TOWRITE

Parameters

- **mesh** (GIMLI::Mesh) – 2d or 3d mesh with or without cells.
- **dDensity** (*float | array*) – Density difference.
  - **float – solve for positive boundary marker only.**
    Assuming one inhomogeneity.
  - [[int, float]] – solve for multiple positive boundaries TOIMPL
  - array – solve for one delta density value per cell
  - None – return per cell kernel matrix G TOIMPL
- **pnts** (*[[x_i, y_i]]*) – List of measurement positions.
- **complete** (*bool [False]*) – If True return whole solution or matrix for [dgx, dgy, dgz] and . . . TODO

Examples using `pygimli.physics.gravimetry.solveGravimetry`

- *Gravimetry in 2D - Part I* (page 116)
- *Semianalytical Gravimetry and Geomagnetics in 2D* (page 118)

`pygimli.physics.gravimetry.`**`uCylinderHoriz`**(*pnts*, *rad*, *rho*, *pos=(0.0, 0.0)*)

Gravitational potential of horizonzal cylinder.

TODO

pygimli.physics.gravimetry.**uSphere**(*r*, *rad*, *rho*, *pos=None*)

Gravitational potential of a sphere.

Gravitational potential of a sphere with radius and density at a given position.

$$u = -G * dM * \frac{1}{r}$$

Parameters

- **r** (*[float, float, float]*) – position vector
- **rad** (*float*) – radius of the sphere
- **rho** (*float*) – density
- **pos** (*[float, float, float]*) – position of sphere (0.0, 0.0, 0.0)

8.4.3.3 Classes

class pygimli.physics.gravimetry.**MagneticsModelling**(*mesh*, *points*, *cmp*, *igrf*, *foot=None*)

Bases: *Modelling* (page 296)

Magnetics modelling operator using Holstein (2007).

__init__(*mesh*, *points*, *cmp*, *igrf*, *foot=None*)

Setup forward operator.

Parameters

- **mesh** (*pygimli:mesh*) – tetrahedral or hexahedral mesh
- **points** (*list|array of (x, y, z)*) – measuring points
- **cmp** (*list of str*) – component of: gx, gy, gz, TFA, Bx, By, Bz, Bxy, Bxz, Byy, Byz, Bzz
- **igrf** (*list|array of size 3 or 7*) – international geomagnetic reference field, either [D, I, H, X, Y, Z, F] - declination, inclination, horizontal field,

  X/Y/Z components, total field OR

  [X, Y, Z] - X/Y/Z components

createJacobian(*model*)

Do nothing as this is a linear problem.

response(*model*)

Compute forward response.

8.4.4 pygimli.physics.petro

Various petrophysical models

8.4.4.1 Overview

Functions

permeabilityEngelhardtPitter (page 391)(poro[, q, s, ...])	Empirical model for porosity to hydraulic permeability.
resistivityArchie (page 391)(rFluid, porosity[, a, m, ...])	Resistivity of rock for the petrophysical model from Archies law.
slownessWyllie (page 392)(phi[, sat, vm, vw, va, mesh, ...])	Return slowness *s* after Wyllie time-average equation.
transFwdArchiePhi (page 392)([rFluid, m])	Archies law transformation function for resistivity(porosity).
transFwdArchieS (page 393)([rFluid, phi, m, n])	Inverse Archie transformation function resistivity(saturation).
transFwdWylliePhi (page 393)([sat, vm, vw, va])	Wyllie transformation function porosity(slowness).
transFwdWyllieS (page 393)(phi[, vm, vw, va])	Wyllie transformation function slowness(saturation).
transInvArchiePhi (page 393)([rFluid, m])	Inverse Archie transformation function porosity(resistivity).
transInvArchieS (page 393)([rFluid, phi, m, n])	Inverse Archie transformation function saturation(resistivity).
transInvWylliePhi (page 393)([sat, vm, vw, va])	Inverse Wyllie transformation function porosity(slowness).
transInvWyllieS (page 393)(phi[, vm, vw, va])	Inverse Wyllie transformation function slowness(saturation).

Classes

JointPetroInversion (page 394)(managers, trans[, ...])	TODO.
PetroInversion (page 394)(manager, trans, **kwargs)	TODO.
PetroJointModelling (page 394)([f, p, mesh, verbose])	Cumulative (joint) forward operator for petrophysical inversions.
PetroModelling (page 395)(fop, trans[, mesh, verbose])	Combine petrophysical relation m(p) with modelling class f(p).

8.4.4.2 Functions

pygimli.physics.petro.**permeabilityEngelhardtPitter**(*poro*, *q=3.5*, *s=0.005*, *mesh=None*, *meshI=None*)

Empirical model for porosity to hydraulic permeability.

Postulated for sand and sandstones. [**?**]

$$k = 2 \cdot 10^7 \frac{\phi^2}{(1-\phi)^2} * \frac{1}{S^2}$$
$$S = q \cdot s$$
$$s = \sum_{i=1} (\frac{P_i}{r_i})$$

- ϕ - poro 0.0 –1.0
- q **- (3 for spheres, > 3 shape differ from sphere)**
  3.5 sand
- s - in cm^-1 (s = 1/r for particles with homogeneous radii r)
- P_i - Particle ration with radii r_i on 1cm^3 Sample
- S - in cm^-1 specific surface in cm^2/cm^3

Returns
in Darcy

Return type
k

pygimli.physics.petro.**resistivityArchie**(*rFluid*, *porosity*, *a=1.0*, *m=2.0*, *sat=1.0*, *n=2.0*, *mesh=None*, *meshI=None*, *fill=None*, *show=False*)

Resistivity of rock for the petrophysical model from Archies law.

Calculates resistivity of rock for the petrophysical model from Archie's law. [**?**]

$$\rho = a\rho_{fl}\phi^{-m}S^{-n}$$

- ρ - the electrical resistivity of the fluid saturated rock in Ωm
- ρ_{fl} - rFluid: electrical resistivity of the fluid in Ωm
- ϕ - porosity 0.0 –1.0
- S - fluid saturation 0.0 –1.0 [sat]
- a - Tortuosity factor. (common 1)
- m - Cementation exponent of the rock (usually in the range 1.3 – 2.5 for sandstones)
- n - is the saturation exponent (usually close to 2)

If mesh is not None the resulting values are calculated for each cell of the mesh. All parameter can be scalar, array of length mesh.cellCount() or callable(pg.cell). If rFluid is non-steady n-step distribution than rFluid can be a matrix of size(n, mesh.cellCount()) If meshI is not None the result is interpolated to meshI.cellCenters() and prolonged (if fill ==1).

Notes

We experience some unstable nonlinear behavior. Until this is clarified all results are rounded to the precision 1e-6.

Examples

```
>>> #
```

WRITEME

Examples using `pygimli.physics.petro.resistivityArchie`

- *Hydrogeophysical modeling* (page 138)

`pygimli.physics.petro.`**`slownessWyllie`**(*phi*, *sat=1*, *vm=4000*, *vw=1484*, *va=343*, *mesh=None*, *meshI=None*, *fill=None*)

Return slowness s after Wyllie time-average equation.

$$s = (1-\phi)\cdot\frac{1}{v_m} + \phi\cdot S\cdot\frac{1}{v_w} + \phi\cdot(1-S)\cdot\frac{1}{v_a}$$

- ϕ - porosity 0.0 –1.0
- S - fluid saturation 0.0 –1.0 [sat]
- v_m - velocity of matrix [4000 m/s]
- v_w - velocity of water [1484 m/s]
- v_a - velocity of air [343 m/s]

If mesh is not None the resulting values are calculated for each cell of the mesh. All parameter can be scalar, array of length mesh.cellCount() or callable(pg.cell). If meshI is not None the result is interpolated to meshI.cellCenters() and prolonged (if fill ==1).

Examples

WRITEME

`pygimli.physics.petro.`**`transFwdArchiePhi`**(*rFluid=20*, *m=2*)

Archies law transformation function for resistivity(porosity).

$$\rho = a\rho_{fl}\phi^{-m}\S_w^{-n}$$
$$\rho = \rho_{fl}\phi^{(}-m) = \left(\phi/\rho_{fl}^{-1/n}\right)^{-n}$$

See also *`pygimli.physics.petro.resistivityArchie`* (page 391)

Returns
trans – Transformation function

Return type
GIMLI::RTransPower

Examples

```
>>> from pygimli.physics.petro import *
>>> phi = 0.3
>>> tFAPhi = transFwdArchiePhi(rFluid=20)
>>> r1 = tFAPhi.trans(phi)
>>> r2 = resistivityArchie(rFluid=20.0, porosity=phi,
...                        a=1.0, m=2.0, sat=1.0, n=2.0)
>>> print(r1-r2 < 1e-12)
True
>>> phi = [0.3]
>>> tFAPhi = transFwdArchiePhi(rFluid=20)
>>> r1 = tFAPhi.trans(phi)
>>> r2 = resistivityArchie(rFluid=20.0, porosity=phi,
...                        a=1.0, m=2.0, sat=1.0, n=2.0)
>>> print((r1-r2 < 1e-12)[0])
True
```

pygimli.physics.petro.**transFwdArchieS**(*rFluid=20*, *phi=0.4*, *m=2*, *n=2*)

Inverse Archie transformation function resistivity(saturation).

Examples using `pygimli.physics.petro.transFwdArchieS`

- *Petrophysical joint inversion* (page 153)

pygimli.physics.petro.**transFwdWylliePhi**(*sat=1*, *vm=4000*, *vw=1600*, *va=330*)

Wyllie transformation function porosity(slowness).

pygimli.physics.petro.**transFwdWyllieS**(*phi*, *vm=4000*, *vw=1600*, *va=330*)

Wyllie transformation function slowness(saturation).

Examples using `pygimli.physics.petro.transFwdWyllieS`

- *Petrophysical joint inversion* (page 153)

pygimli.physics.petro.**transInvArchiePhi**(*rFluid=20*, *m=2*)

Inverse Archie transformation function porosity(resistivity).

rFluid/rho = phi^m ==> phi = (rFluid/rho)^(1/m) = (rho/rFluid)^(-1/m) See — *pygimli.physics.petro.transFwdArchiePhi* (page 392)

pygimli.physics.petro.**transInvArchieS**(*rFluid=20*, *phi=0.4*, *m=2*, *n=2*)

Inverse Archie transformation function saturation(resistivity).

pygimli.physics.petro.**transInvWylliePhi**(*sat=1*, *vm=4000*, *vw=1600*, *va=330*)

Inverse Wyllie transformation function porosity(slowness).

pygimli.physics.petro.**transInvWyllieS**(*phi*, *vm=4000*, *vw=1600*, *va=330*)

Inverse Wyllie transformation function slowness(saturation).

8.4.4.3 Classes

class pygimli.physics.petro.**JointPetroInversion**(*managers*, *trans*, *verbose=False*, *debug=False*, ***kwargs*)

Bases: *MethodManager* (page 292)

TODO.

__init__(*managers*, *trans*, *verbose=False*, *debug=False*, ***kwargs*)

TODO.

static createFOP(*verbose=False*)

Create forward operator.

createInv(*fop*, *verbose=True*, *doSave=False*)

TODO.

invert(*data=None*, *mesh=None*, *lam=20*, *limits=None*, ***kwargs*)

TODO.

model()

setData(*data*)

TODO.

setMesh(*mesh*)

TODO.

showModel(***showkwargs*)

TODO.

class pygimli.physics.petro.**PetroInversion**(*manager*, *trans*, ***kwargs*)

Bases: *JointPetroInversion* (page 394)

TODO.

__init__(*manager*, *trans*, ***kwargs*)

TODO.

invert(*data*, ***kwargs*)

TODO.

class pygimli.physics.petro.**PetroJointModelling**(*f=None*, *p=None*, *mesh=None*, *verbose=True*)

Bases: *Modelling* (page 296)

Cumulative (joint) forward operator for petrophysical inversions.

__init__(*f=None*, *p=None*, *mesh=None*, *verbose=True*)

Constructor.

createJacobian(*model*)

Creating individual Jacobian matrices.

initJacobian()

TODO.

response(*model*)

Create concatenated response for fop stack with model.

setData(*data*)

TODO.

setFopsAndTrans(*fops*, *trans*)

TODO.

setMesh(*mesh*)

TODO.

class pygimli.physics.petro.**PetroModelling**(*fop*, *trans*, *mesh=None*, *verbose=False*)

Bases: *Modelling* (page 296)

Combine petrophysical relation m(p) with modelling class f(p).

Combine petrophysical relation m(p) with modelling class f(p) to invert for m (or any inversion transformation) instead of p.

__init__(*fop*, *trans*, *mesh=None*, *verbose=False*)

Save forward class and transformation, create Jacobian matrix.

createJacobian(*model*)

Fill the individual jacobian matrices.

response(*model*)

Use inverse transformation to get p(m) and compute response.

setData(*data*)

TODO.

setMesh(*mesh*)

TODO.

8.4.5 pygimli.physics.seismics

Full wave form seismics utilities and simulations

8.4.5.1 Overview

Functions

drawSeismogram (page 396)(ax, mesh, u, dt[, ids, pos, i])	Extract and show time series from wave field
drawWiggle (page 396)(ax, x, t[, xoffset, posColor, ...])	Draw signal in wiggle style into a given ax.
ricker (page 396)(f, t[, t0])	Create Ricker wavelet.
solvePressureWave (page 397)(mesh, velocities, times, ...)	Solve pressure wave equation.

8.4.5.2 Functions

pygimli.physics.seismics.**drawSeismogram**(*ax*, *mesh*, *u*, *dt*, *ids=None*, *pos=None*, *i=None*)

Extract and show time series from wave field

Parameters

- **ids** (*list*) – List of node ids for the given mesh.
- **pos** (*list*) – List of positions for the given mesh. We will look for the nearest node.

pygimli.physics.seismics.**drawWiggle**(*ax*, *x*, *t*, *xoffset=0.0*, *posColor='red'*, *negColor='blue'*, *alpha=0.5*, ***kwargs*)

Draw signal in wiggle style into a given ax.

Parameters

- **ax** (*matplotlib ax*) – To plot into
- **x** (*array [float]*) – Signal.
- **t** (*array*) – Time base for x
- **xoffset** (*float*) – Move wiggle plot along x axis
- **posColor** (*str*) – Need to be convertible to matplotlib color. Fill positive areas with.
- **negColor** (*str*) – Need to be convertible to matplotlib color. Fill negative areas with.
- **alpha** (*float*) – Opacity for fill area.
- ****kwargs** (*dict()*) – Will be forwarded to matplotlib.axes.fill

```
>>> from pygimli.physics.seismics import ricker, drawWiggle
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> t = np.arange(0, 0.02, 1./5000)
>>> r = ricker(t, 100., 1./100)
>>> fig = plt.figure()
>>> ax = fig.add_subplot(1,1,1)
>>> drawWiggle(ax, r, t, xoffset=0, posColor='red', negColor='blue',
...            alpha=0.2)
>>> drawWiggle(ax, r, t, xoffset=1)
>>> drawWiggle(ax, r, t, xoffset=2, posColor='black', negColor='white',
...            alpha=1.0)
>>> ax.invert_yaxis()
>>> plt.show()
```

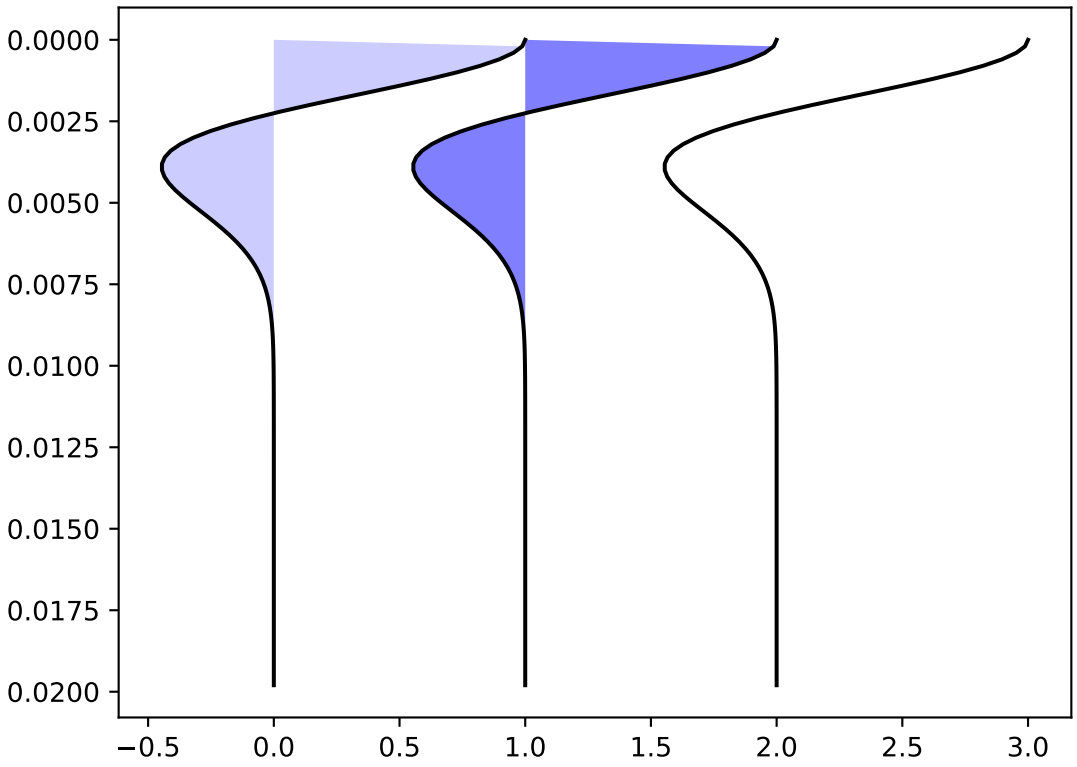
pygimli.physics.seismics.**ricker**(*f*, *t*, *t0=0.0*)

Create Ricker wavelet.

Create a Ricker wavelet with a desired frequency and signal length.

Parameters

- **f** (*float*) – Frequency of the wavelet in Hz



- **t** (*array [float]*) – Time base definition
- **t0** (*float*) – Offset time. Use 1/f to move the wavelet to start nearly from zero.

Returns
y – Signal

Return type
array_like

Create a 100 Hz Wavelet inside 1000 Hz sampled signal of length 0.1s.

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> from pygimli.physics.seismics import ricker
>>> sampleFrequenz = 1000 #Hz
>>> t = np.arange(0, 0.1, 1./sampleFrequenz)
>>> r = ricker(100., t, 1./100)
>>> lines = plt.plot(t,r,'-x')
>>> plt.show()
```

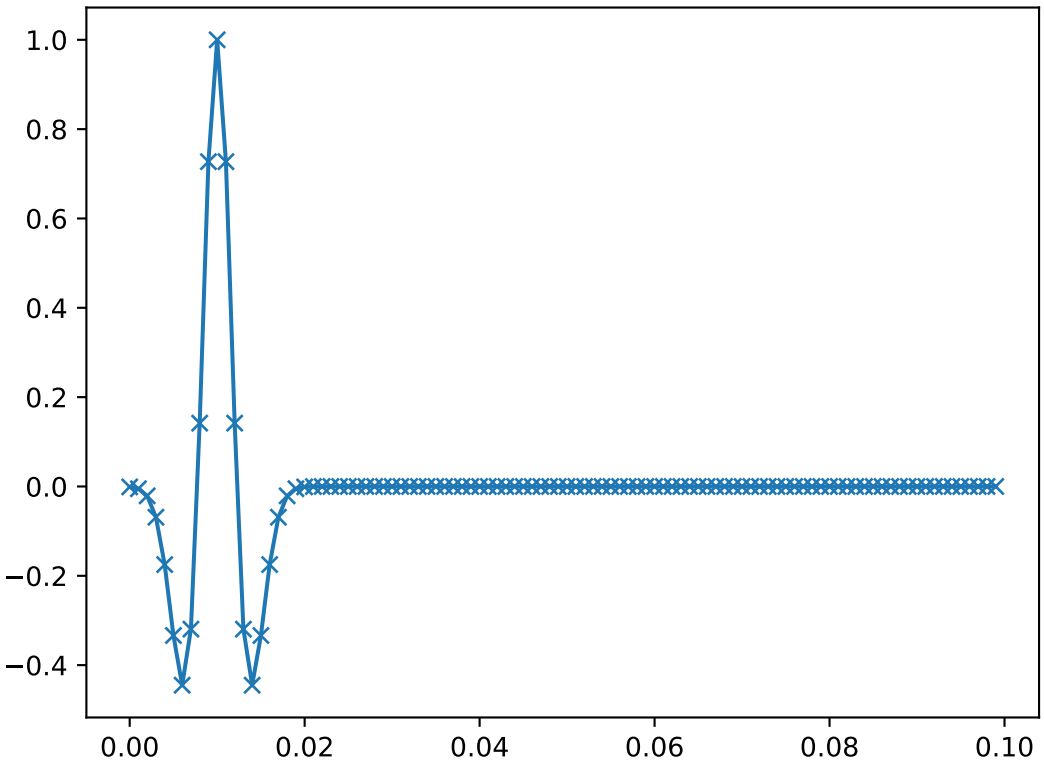
pygimli.physics.seismics.**solvePressureWave**(*mesh*, *velocities*, *times*, *sourcePos*, *uSource*, *verbose=False*)

Solve pressure wave equation.

Solve pressure wave for a given source function

$$\frac{\partial^2 u}{\partial t^2} = \nabla \cdot (a \nabla u) + f$$

finalizeequation



Parameters

- **mesh** (GIMLI::Mesh) – Mesh to solve on
- **velocities** (*array*) – velocities for each cell of the mesh
- **time** (*array*) – Time base definition
- **sourcePos** (*RVector3*) – Source position
- **uSource** (*array*) – u(t, sourcePos) source movement of length(times) Usually a Ricker wavelet of the desired seismic signal frequency.

Returns

u – Return

Return type

RMatrix

Examples

See TODO write example

8.4.6 pygimli.physics.SIP

Spectral induced polarization (SIP) measurements and fittings.

8.4.6.1 Overview

Functions

ColeCole (page 401)(f, R, m, tau, c[, a])	
ColeColeEpsilon (page 401)(f, e0, eInf, tau, alpha)	
ColeColeRho (page 401)(f, rho, m, tau, c[, a])	
ColeColeRhoDouble (page 401)(f, rho, m1, t1, c1, m2, t2, c2)	
ColeColeSigma (page 401)(f, sigma, m, tau, c[, a])	
ColeDavidson (page 401)(f, R, m, tau[, a])	
drawAmplitudeSpectrum (page 401)(ax, freq, amp[, ...])	Show amplitude spectrum (resistivity as a function of f).
drawPhaseSpectrum (page 401)(ax, freq, phi[, ylabel, ...])	Show phase spectrum (-phi as a function of f).
load (page 401)(fileName[, verbose])	Shortcut to load SIP spectral data.
modelColeColeEpsilon (page 402)(f, e0, eInf, tau, alpha)	Original complex-valued permittivity formulation (Cole&Cole, 1941).
modelColeColeRho (page 402)(f, rho, m, tau, c[, a])	Frequency-domain Cole-Cole impedance model after Pelton et al. (1978).
modelColeColeRhoDouble (page 403)(f, rho, m1, t1, c1, ...)	Frequency-domain double Cole-Cole resistivity (impedance) model.
modelColeColeSigma (page 403)(f, sigma, m, tau, c[, a])	Complex-valued conductivity (admittance) Cole-Cole model.
modelColeColeSigmaDouble (page 404)(f, sigma, m1, t1, ...)	Complex-valued double added conductivity (admittance) model.
modelColeDavidson (page 404)(f, R, m, tau[, a])	For backward compatibility.
showSpectrum (page 404)(freq, amp, phi[, nrows, ylog, axs])	Show amplitude and phase spectra in two subplots.
tauRhoToTauSigma (page 404)(tRho, m, c)	Convert τ_ρ to τ_σ Cole-Cole model.

Classes

ColeColeAbs (page 404)(f[, verbose])	Cole-Cole model with EM term after Pelton et al. (1978).
ColeColeComplex (page 405)(f[, verbose])	Cole-Cole model with EM term after Pelton et al. (1978).
ColeColeComplexSigma (page 405)(f[, verbose])	Cole-Cole model with EM term after Pelton et al. (1978).
ColeColePhi (page 406)(f[, verbose])	Cole-Cole model with EM term after Pelton et al. (1978).
DebyeComplex (page 406)(fvec, tvec[, verbose])	Debye decomposition (smooth Debye relaxations) of complex data
DebyePhi (page 406)(fvec, tvec[, verbose])	Debye decomposition (smooth Debye relaxations) phase only
DoubleColeColePhi (page 407)(f[, verbose])	Double Cole-Cole model after Pelton et al. (1978).
PeltonPhiEM (page 407)(f[, verbose])	Cole-Cole model with EM term after Pelton et al. (1978).
SIPSpectrum (page 407)([filename, unify, onlydown, f, ...])	SIP spectrum data analysis.
SpectrumManager (page 412)([fop])	Manager to work with spectra data.
SpectrumModelling (page 413)([funct])	Modelling framework with an array of freqencies as data space.

8.4.6.2 Functions

pygimli.physics.SIP.**ColeCole**(*f*, *R*, *m*, *tau*, *c*, *a=1*)

pygimli.physics.SIP.**ColeColeEpsilon**(*f*, *e0*, *eInf*, *tau*, *alpha*)

pygimli.physics.SIP.**ColeColeRho**(*f*, *rho*, *m*, *tau*, *c*, *a=1*)

pygimli.physics.SIP.**ColeColeRhoDouble**(*f*, *rho*, *m1*, *t1*, *c1*, *m2*, *t2*, *c2*)

pygimli.physics.SIP.**ColeColeSigma**(*f*, *sigma*, *m*, *tau*, *c*, *a=1*)

pygimli.physics.SIP.**ColeDavidson**(*f*, *R*, *m*, *tau*, *a=1*)

pygimli.physics.SIP.**drawAmplitudeSpectrum**(*ax*, *freq*, *amp*, *ylabel='$\\rho$ ($\\Omega$m)'*, *grid=True*, *marker='+'*, *ylog=True*, ***kwargs*)

Show amplitude spectrum (resistivity as a function of f).

pygimli.physics.SIP.**drawPhaseSpectrum**(*ax*, *freq*, *phi*, *ylabel='$-\\phi$ (mrad)'*, *grid=True*, *marker='+'*, *ylog=False*, ***kwargs*)

Show phase spectrum (-phi as a function of f).

pygimli.physics.SIP.**load**(*fileName*, *verbose=False*, ***kwargs*)

Shortcut to load SIP spectral data.

Import Data and try to assume the file format.

Parameters
fileName (*str*) –

Returns
freqs, amp, phi – Frequencies, amplitudes and phases phi in neg. radiant

Return type
np.array

pygimli.physics.SIP.**modelColeColeEpsilon**(*f*, *e0*, *eInf*, *tau*, *alpha*)

Original complex-valued permittivity formulation (Cole&Cole, 1941).

pygimli.physics.SIP.**modelColeColeRho**(*f*, *rho*, *m*, *tau*, *c*, *a=1*)

Frequency-domain Cole-Cole impedance model after Pelton et al. (1978)

Frequency-domain Cole-Cole impedance model after Pelton et al. (1978) [**?**]

$$Z(\omega) = \rho_0 \left[1 - m\left(1 - \frac{1}{1+(i\omega\tau)^c}\right)\right]$$
$$\text{with} \quad m = \frac{1}{1+\frac{\rho_0}{\rho_1}} \quad \text{and} \quad \omega = 2\pi f$$

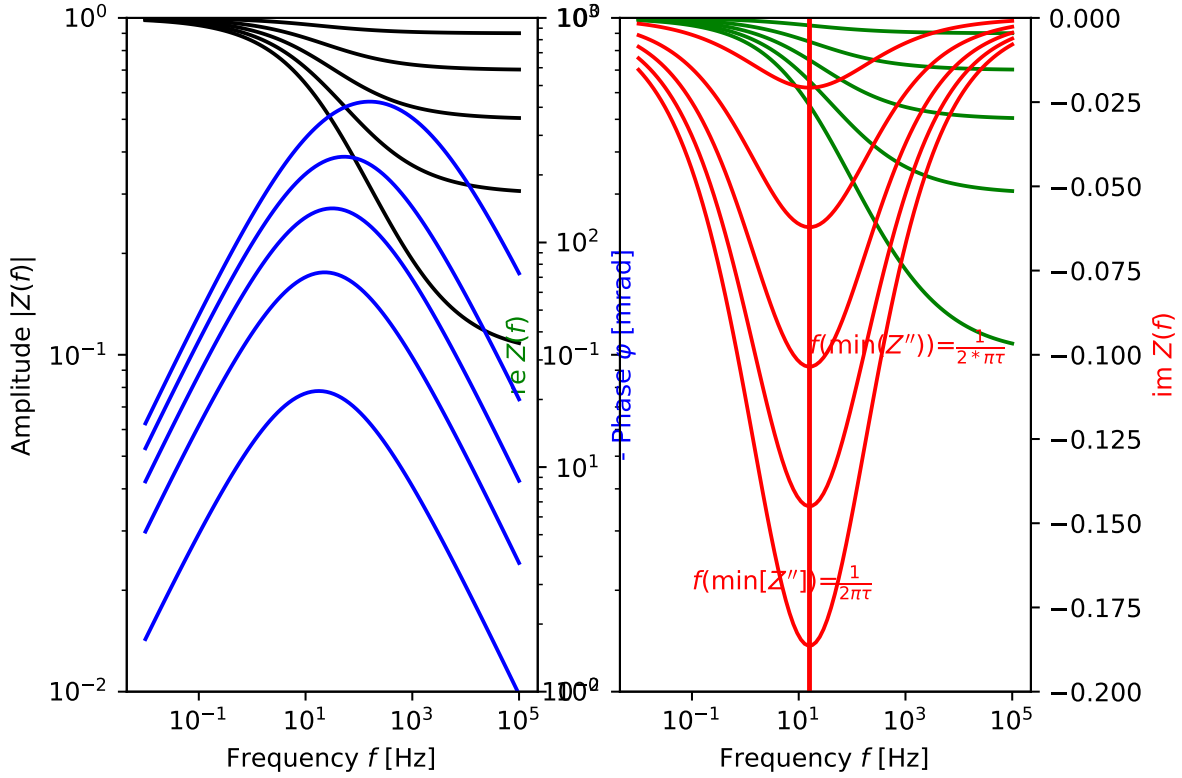
- $Z(\omega)$ - Complex impedance per 1A current injection
- f - Frequency
- ρ_0 – Background resistivity states the unblocked pore path
- ρ_1 – Resistance of the solution in the blocked pore passages
- m – Chargeability after Seigel (1959) [**?**] as being the ratio of voltage immediately after, to the voltage immediately before cessation of an infinitely long charging current.
- τ – 'Time constant' relaxation time [s] for 1/e decay
- c - Rate of charge accumulation. Cole-Cole exponent typically [0.1 .. 0.6]

```
>>> import numpy as np
>>> import pygimli as pg
>>> from pygimli.physics.SIP import modelColeColeRho
>>> f = np.logspace(-2, 5, 100)
>>> m = np.linspace(0.1, 0.9, 5)
>>> tau = 0.01
>>> fImMin = 1/(tau*2*np.pi)
>>> fig, axs = pg.plt.subplots(1, 2)
>>> ax1 = axs[0]
>>> ax2 = axs[0].twinx()
>>> ax3 = axs[1]
>>> ax4 = axs[1].twinx()
>>> for i in range(len(m)):
...     Z = modelColeColeRho(f, rho=1, m=m[i], tau=tau, c=0.5)
...     _= ax1.loglog(f, np.abs(Z), color='black')
...     _= ax2.loglog(f, -np.angle(Z)*1000, color='b')
...     _= ax3.loglog(f, Z.real, color='g')
...     _= ax4.semilogx(f, Z.imag, color='r')
...     _= ax4.plot([fImMin, fImMin], [-0.2, 0.], color='r')
```

(continues on next page)

(continued from previous page)

```
>>> _ = ax4.text(fImMin, -0.1, r"$f($min($Z''$))=$\frac{1}{2*\pi\tau}$",␣
↪color='r')
>>> _ = ax4.text(0.1, -0.17, r"$f($min[$Z''$])=$\frac{1}{2\pi\tau}$",␣
↪color='r')
>>> _ = ax1.set_ylabel('Amplitude $|Z(f)|$', color='black')
>>> _ = ax1.set_xlabel('Frequency $f$ [Hz]')
>>> _ = ax1.set_ylim(1e-2, 1)
>>> _ = ax2.set_ylabel(r'- Phase $\varphi$ [mrad]', color='b')
>>> _ = ax2.set_ylim(1, 1e3)
>>> _ = ax3.set_ylabel('re $Z(f)$', color='g')
>>> _ = ax4.set_ylabel('im $Z(f)$', color='r')
>>> _ = ax3.set_xlabel('Frequency $f$ [Hz]')
>>> _ = ax3.set_ylim(1e-2, 1)
>>> _ = ax4.set_ylim(-0.2, 0)
>>> pg.plt.show()
```



Examples using `pygimli.physics.SIP.modelColeColeRho`

- *Generating SIP signatures* (page 90)
- *Fitting SIP signatures* (page 92)

`pygimli.physics.SIP.`**`modelColeColeRhoDouble`**(*f*, *rho*, *m1*, *t1*, *c1*, *m2*, *t2*, *c2*, *a=1*, *mult=False*)

Frequency-domain double Cole-Cole resistivity (impedance) model.

Frequency-domain Double Cole-Cole impedance model returns the sum of two Cole-Cole Models with a common amplitude. Z = rho * (Z1(Cole-Cole) + Z2(Cole-Cole))

pygimli.physics.SIP.**modelColeColeSigma**(*f*, *sigma*, *m*, *tau*, *c*, *a=1*)

Complex-valued conductivity (admittance) Cole-Cole model.

pygimli.physics.SIP.**modelColeColeSigmaDouble**(*f*, *sigma*, *m1*, *t1*, *c1*, *m2*, *t2*, *c2*, *a=1*, *tauRho=True*)

Complex-valued double added conductivity (admittance) model.

pygimli.physics.SIP.**modelColeDavidson**(*f*, *R*, *m*, *tau*, *a=1*)

For backward compatibility.

pygimli.physics.SIP.**showSpectrum**(*freq*, *amp*, *phi*, *nrows=2*, *ylog=None*, *axs=None*, ***kwargs*)

Show amplitude and phase spectra in two subplots.

pygimli.physics.SIP.**tauRhoToTauSigma**(*tRho*, *m*, *c*)

Convert τ_ρ to τ_σ Cole-Cole model.

$$\tau_\sigma = \tau_\rho / (1-m)^{\frac{1}{c}}$$

Examples

```
>>> import numpy as np
>>> import pygimli as pg
>>> from pygimli.physics.SIP import modelColeColeRho,␣
↪modelColeColeSigma
>>> from pygimli.physics.SIP import tauRhoToTauSigma
>>> tr = 1.
>>> Z = modelColeColeRho(1e5, rho=10.0, m=0.5, tau=tr, c=0.5)
>>> ts = tauRhoToTauSigma(tr, m=0.5, c=0.5)
>>> S = modelColeColeSigma(1e5, sigma=0.1, m=0.5, tau=ts, c=0.5)
>>> abs(1.0/S - Z) < 1e-12
True
>>> np.angle(1.0/S / Z) < 1e-12
True
```

8.4.6.3 Classes

class pygimli.physics.SIP.**ColeColeAbs**(*f*, *verbose=False*)

Bases: ModellingBaseMT__

Cole-Cole model with EM term after Pelton et al. (1978)

__**init**__(*(object)arg1*[, *(object)verbose=False*]) → object :

C++ signature :
void* __init__(_object* [,bool=False])

__init__((object)arg1, (object)dataContainer [, (object)verbose=False]) -> object :

C++ signature :
void* __init__(_object*,GIMLI::DataContainer {lvalue} [,bool=False])

__init__((object)arg1, (object)mesh [, (object)verbose=False]) -> object :

C++ signature :
void* __init__(_object*,GIMLI::Mesh [,bool=False])

__init__((object)arg1, (object)mesh, (object)dataContainer [, (object)verbose=False]) -> object :

C++ signature :
void* __init__(_object*,GIMLI::Mesh,GIMLI::DataContainer {lvalue} [,bool=False])

response(*par*)

phase angle of the model

class pygimli.physics.SIP.**ColeColeComplex**(*f*, *verbose=False*)

Bases: ModellingBaseMT__

Cole-Cole model with EM term after Pelton et al. (1978)

__init__(*(object)arg1*[, *(object)verbose=False*]) → object :

C++ signature :
void* __init__(_object* [,bool=False])

__init__((object)arg1, (object)dataContainer [, (object)verbose=False]) -> object :

C++ signature :
void* __init__(_object*,GIMLI::DataContainer {lvalue} [,bool=False])

__init__((object)arg1, (object)mesh [, (object)verbose=False]) -> object :

C++ signature :
void* __init__(_object*,GIMLI::Mesh [,bool=False])

__init__((object)arg1, (object)mesh, (object)dataContainer [, (object)verbose=False]) -> object :

C++ signature :
void* __init__(_object*,GIMLI::Mesh,GIMLI::DataContainer {lvalue} [,bool=False])

response(*par*)

phase angle of the model

class pygimli.physics.SIP.**ColeColeComplexSigma**(*f*, *verbose=False*)

Bases: ModellingBaseMT__

Cole-Cole model with EM term after Pelton et al. (1978)

__init__(*(object)arg1*[, *(object)verbose=False*]) → object :

C++ signature :
void* __init__(_object* [,bool=False])

__init__((object)arg1, (object)dataContainer [, (object)verbose=False]) -> object :

C++ signature :
void* __init__(_object*,GIMLI::DataContainer {lvalue} [,bool=False])

__init__((object)arg1, (object)mesh [, (object)verbose=False]) -> object :

C++ signature :
void* __init__(_object*,GIMLI::Mesh [,bool=False])

__init__((object)arg1, (object)mesh, (object)dataContainer [, (object)verbose=False]) -> object :

C++ signature :
void* __init__(_object*,GIMLI::Mesh,GIMLI::DataContainer {lvalue} [,bool=False])

response(*par*)

phase angle of the model

class pygimli.physics.SIP.**ColeColePhi**(*f*, *verbose=False*)

Bases: ModellingBaseMT__

Cole-Cole model with EM term after Pelton et al. (1978)

Modelling operator for the Frequency Domain *Cole-Cole* (page 402) impedance model using *pygimli.physics.SIP.modelColeColeRho* (page 402) after Pelton et al. (1978) [**?**]

- $\mathbf{m} = \{m, \tau, c\}$

  Modelling parameter for the Cole-Cole model with $\rho_0 = 1$

- $\mathbf{d} = \{\varphi_i(f_i)\}$

  Modelling response for all given frequencies as negative phase angles $\varphi(f) = -tan^{-1}\frac{\mathrm{Im}Z(f)}{\mathrm{Re}Z(f)}$ and $Z(f, \rho_0 = 1, m, \tau, c) =$ Cole-Cole impedance.

__init__(*f*, *verbose=False*)

Setup class by specifying the frequency.

response(*par*)

Phase angle of the model.

class pygimli.physics.SIP.**DebyeComplex**(*fvec*, *tvec*, *verbose=False*)

Bases: ModellingBaseMT__

Debye decomposition (smooth Debye relaxations) of complex data

__init__(*fvec*, *tvec*, *verbose=False*)

constructor with frequecy and tau vector

createJacobian(*par*)

linear jacobian after Nordsiek&Weller (2008)

response(*par*)

amplitude/phase spectra as function of spectral chargeabilities

class pygimli.physics.SIP.**DebyePhi**(*fvec*, *tvec*, *verbose=False*)

Bases: ModellingBaseMT__

Debye decomposition (smooth Debye relaxations) phase only

__init__(*fvec*, *tvec*, *verbose=False*)

constructor with frequecy and tau vector

response(*par*)

amplitude/phase spectra as function of spectral chargeabilities

class pygimli.physics.SIP.**DoubleColeColePhi**(*f*, *verbose=False*)

Bases: ModellingBaseMT__

Double Cole-Cole model after Pelton et al. (1978)

Modelling operator for the Frequency Domain - phase only *Cole-Cole* (page 402) impedance model using *pygimli.physics.SIP.modelColeColeRho* (page 402) after Pelton et al. (1978) [**?**]

- $\mathbf{m} = \{m_1, \tau_1, c_1, m_2, \tau_2, c_2\}$

  Modelling parameter for the Cole-Cole model with $\rho_0 = 1$

- $\mathbf{d} = \{\varphi_i(f_i)\}$

  Modelling Response for all given frequencies as negative phase angles $\varphi(f) = \varphi_1(Z_1(f)) + \varphi_2(Z_2(f)) = -tan^{-1}\frac{\mathrm{Im}(Z_1 Z_2)}{\mathrm{Re}(Z_1 Z_2)}$ and $Z_1(f, \rho_0 = 1, m_1, \tau_1, c_1)$ and $Z_2(f, \rho_0 = 1, m_2, \tau_2, c_2)$ ColeCole impedances.

__init__(*f*, *verbose=False*)

Setup class by specifying the frequency.

response(*par*)

phase angle of the model

class pygimli.physics.SIP.**PeltonPhiEM**(*f*, *verbose=False*)

Bases: ModellingBaseMT__

Cole-Cole model with EM term after Pelton et al. (1978)

__init__(*(object)arg1*[, *(object)verbose=False*]) → object :

C++ signature :
void* __init__(_object* [,bool=False])

__init__((object)arg1, (object)dataContainer [, (object)verbose=False]) -> object :

C++ signature :
void* __init__(_object*,GIMLI::DataContainer {lvalue} [,bool=False])

__init__((object)arg1, (object)mesh [, (object)verbose=False]) -> object :

C++ signature :
void* __init__(_object*,GIMLI::Mesh [,bool=False])

__init__((object)arg1, (object)mesh, (object)dataContainer [, (object)verbose=False]) -> object :

C++ signature :
void* __init__(_object*,GIMLI::Mesh,GIMLI::DataContainer {lvalue} [,bool=False])

response(*par*)

phase angle of the model

class pygimli.physics.SIP.**SIPSpectrum**(*filename=None*, *unify=False*, *onlydown=True*, *f=None*, *amp=None*, *phi=None*, *k=1*, *sort=True*, *basename='new'*)

Bases: object

SIP spectrum data analysis.

__init__(*filename=None*, *unify=False*, *onlydown=True*, *f=None*, *amp=None*, *phi=None*, *k=1*, *sort=True*, *basename='new'*)

Init SIP class with either filename to read or data vectors.

Examples

```
>>> #sip = SIPSpectrum('sipexample.txt', unify=True) #␣
↪unique f values
>>> #sip = SIPSpectrum(f=f, amp=R, phi=phase, basename='new')
```

checkCRKK(*useEps=False*, *use0=False*, *ax=None*)

Check coupling removal (CR) by Kramers-Kronig (KK) relation

cutF(*fcut=1e+99*, *down=False*)

Cut (delete) frequencies above a certain value fcut.

determineEpsilon(*mode=0*, *sigmaR=None*, *sigmaI=None*)

Retrieve frequency-independent epsilon for f->Inf.

Parameters

- **mode** (*int*) –

  Operation mode:
  =0 - extrapolate using two highest frequencies (default) <0 - take last -n frequencies >0 - take n-th frequency

- **sigmaR/sigmaI** (*float*) – real and imaginary conductivity (if not given take data)

Returns
er – relative permittivity (epsilon) value (dimensionless)

Return type
float

epsilonR()

Calculate relative permittivity from imaginary conductivity

fit2CCPhi(*ePhi=0.001*, *lam=1000.0*, *mpar=(0, 0, 1)*, *taupar1=(0, 1e-05, 1)*, *taupar2=(0, 0.1, 1000)*, *cpar=(0.5, 0, 1)*, *verbose=False*)

Fit two Cole-Cole terms (to phase only).

Parameters

- **ePhi** (*float*) – absolute error of phase angle
- **lam** (*float*) – regularization parameter

- **mpar** (*list[3]*) – starting value, lower bound, upper bound for chargeability
- **taupar2** (*taupar1 /*) – starting value, lower bound, upper bound for 2 time constants
- **cpar2** (*cpar1 /*) – starting value, lower bound, upper bound for 2 relaxation exponents

fitCCEM(*ePhi=0.001*, *lam=1000.0*, *remove=True*, *mpar=(0.2, 0, 1)*, *taupar=(0.01, 1e-05, 100)*, *cpar=(0.25, 0, 1)*, *empar=(1e-07, 1e-09, 1e-05)*, *verbose=False*)

Fit a Cole-Cole term with additional EM term to phase

Parameters

- **ePhi** (*float*) – absolute error of phase angle
- **lam** (*float*) – regularization parameter
- **remove** (*bool*) – remove EM term from data
- **mpar** (*list[3]*) – inversion parameters (starting value, lower bound, upper bound) for Cole-Cole parameters (m, tau, c) and EM relaxation time (em)
- **taupar** (*list[3]*) – inversion parameters (starting value, lower bound, upper bound) for Cole-Cole parameters (m, tau, c) and EM relaxation time (em)
- **cpar** (*list[3]*) – inversion parameters (starting value, lower bound, upper bound) for Cole-Cole parameters (m, tau, c) and EM relaxation time (em)
- **empar** (*list[3]*) – inversion parameters (starting value, lower bound, upper bound) for Cole-Cole parameters (m, tau, c) and EM relaxation time (em)

fitCCPhi(*ePhi=0.001*, *lam=1000.0*, *mpar=(0, 0, 1)*, *taupar=(0, 1e-05, 100)*, *cpar=(0.3, 0, 1)*, *verbose=False*)

Fit a Cole-Cole term (to phase only).

Parameters

- **ePhi** (*float*) – absolute error of phase angle
- **lam** (*float*) – regularization parameter
- **mpar** (*list[3]*) – inversion parameters (starting value, lower bound, upper bound) for Cole-Cole parameters (m, tau, c) and EM relaxation time (em)
- **taupar** (*list[3]*) – inversion parameters (starting value, lower bound, upper bound) for Cole-Cole parameters (m, tau, c) and EM relaxation time (em)
- **cpar** (*list[3]*) – inversion parameters (starting value, lower bound, upper bound) for Cole-Cole parameters (m, tau, c) and EM relaxation time (em)

fitColeCole(*useCond=False*, ***kwargs*)

Fit a Cole-Cole model to the phase data

Parameters

- **useCond** (*bool*) – use conductivity form of Cole-Cole model instead of resistivity
- **error** (*float [0.01]*) – absolute phase error
- **lam** (*float [1000]*) – initial regularization parameter
- **mpar** (*tuple/list (0, 0, 1)*) – inversion parameters for chargeability: start, lower, upper bound
- **taupar** (*tuple/list (1e-2, 1e-5, 100)*) – inversion parameters for time constant: start, lower, upper bound
- **cpar** (*tuple/list (0.25, 0, 1)*) – inversion parameters for Cole exponent: start, lower, upper bound

fitDebyeModel(*ePhi=0.001*, *lam=1000.0*, *lamFactor=0.8*, *mint=None*, *maxt=None*, *nt=None*, *new=True*, *showFit=False*, *cType=1*, *verbose=False*)

Fit a (smooth) continuous Debye model (Debye decomposition).

Parameters

- **ePhi** (*float*) – absolute error of phase angle
- **lam** (*float*) – regularization parameter
- **lamFactor** (*float*) – regularization factor for subsequent iterations
- **mint/maxt** (*float*) – minimum/maximum tau values to use (else automatically from f)
- **nt** (*int*) – number of tau values (default number of frequencies * 2)
- **new** (*bool*) – new implementation (experimental)
- **showFit** (*bool*) – show fit
- **cType** (*int*) – constraint type (1/2=smoothness 1st/2nd order, 0=minimum norm)

fitDoubleColeCole(*ePhi=0.001*, *eAmp=0.01*, *lam=1000.0*, *robust=False*, *verbose=True*, *useRho=True*, *useMult=False*, *aphi=True*, *mpar1=(0.2, 0, 1)*, *mpar2=(0.2, 0, 1)*, *tauRho=True*, *taupar1=(0.01, 1e-05, 100)*, *taupar2=(0.0001, 1e-05, 100)*, *cpar1=(0.5, 0, 1)*, *cpar2=(0.5, 0, 1)*)

Fit double Cole-Cole term to complex resistivity or phase.

Parameters

- **ePhi** (*float [0.001]*) – absolute error of phase angle in rad
- **eAmp** (*float [0.01 = 1%]*) – absolute error of phase angle
- **lam** (*float*) – regularization parameter
- **robust** (*bool [False]*) – use robustData (L1 norm on data side)
- **useRho** (*bool [True]*) – Cole-Cole defined for impedance/resistivity, otherwise conductivity

- **useMult** (*bool [False]*) – the two terms are combined as product (otherwise sum)
- **tauRho** (*bool [False]*) – in case of useRho=False the time constant is defined like for rho
- **mpar1/2** (*list[3]*) – inversion parameters (starting value, lower bound, upper bound) for Cole-Cole parameters (m, tau, c)
- **taupar1/2** (*list[3]*) – inversion parameters (starting value, lower bound, upper bound) for Cole-Cole parameters (m, tau, c)
- **cpar1/2** (*list[3]*) – inversion parameters (starting value, lower bound, upper bound) for Cole-Cole parameters (m, tau, c)

getKK(*use0=False*)

Compute Kramers-Kronig impedance values (re->im and im->re).

getPhiKK(*use0=False*)

Compute phase from Kramers-Kronig quantities.

loadData(*filename*, ***kwargs*)

Import spectral data.

Import Data and try to assume the file format.

logMeanTau()

Mean logarithmic relaxation time (50% cumulative log curve).

omega()

Angular frequency.

realimag(*cond=False*)

Real and imaginary part of resistivity/conductivity (cond=True).

removeEpsilonEffect(*er=None*, *mode=0*)

remove effect of (constant high-frequency) epsilon from sigma

Parameters

- **er** (*float*) – relative epsilon to correct for (else automatically determined)
- **mode** (*int*) – automatic epsilon determination mode (see determineEpsilon)

Returns

er – determined permittivity (see determineEpsilon)

Return type

float

saveFigures(*name=None*, *ext='pdf'*)

Save all existing figures to files using file basename.

showAll(*save=False*, *ax=None*)

Plot spectrum, Cole-Cole fit and Debye distribution

showData(*reim=False*, *znorm=False*, *cond=False*, *nrows=2*, *ax=None*, ***kwargs*)

Show amplitude and phase spectrum in two subplots

Parameters

- **reim** (*bool*) – show real/imaginary part instead of amplitude/phase
- **znorm** (*bool (true forces reim)*) – use normalized real/imag parts after Nordsiek&Weller (2008)
- **(default=2)** (*nrows - use nrows subplots*) –

Returns

fig, ax

Return type

mpl.figure, mpl.axes array

showDataKK(*use0=False*)

Show data as real/imag subplots along with Kramers-Kronig curves

showPhase(*ax=None*, ***kwargs*)

Plot phase spectrum (-phi over frequency).

showPolarPlot(*cond=False*)

Show data in a polar plot (imaginary vs. real parts).

sortData()

Sort data along increasing frequency (e.g. useful for KK).

totalChargeability()

Total chargeability (sum) from Debye curve.

unifyData(*onlydown=False*)

Unify data (only one value per frequency) by mean or selection.

zNorm()

Normalized real (difference) and imag. z [**?**]

class pygimli.physics.SIP.**SpectrumManager**(*fop=None*, ***kwargs*)

Bases: *MethodManager* (page 292)

Manager to work with spectra data.

__init__(*fop=None*, ***kwargs*)

Set up spectrum manager

Parameters

- **fop** (*pg.Modelling operator, optional*) – Forward operator. The default is None.
- ****kwargs** (*TYPE*) – passed to SpectrumManager.

createForwardOperator(***kwargs*)

Create a Forward operator.

createInversionFramework(***kwargs*)

Create inversion framework.

invert(*data=None*, *f=None*, ***kwargs*)

setData(*freqs=None*, *amp=None*, *phi=None*, *eAmp=0.03*, *ePhi=0.001*)

Set data for chosen sip model.

Parameters

- **freqs** (*iterable*) – Array-like frequencies.
- **amp** (*iterable*) – Array-like amplitudes to work with.
- **phi** (*iterable*) – Array-like phase angles to work with.
- **eAmp** (*float|iterable*) – Relative error for amplitudes.
- **ePhi** (*float|iterable*) – Absolute error for phase angles.

setFunct(*fop*, ***kwargs*)

Set forward modelling function

showResult()

simulate()

class pygimli.physics.SIP.**SpectrumModelling**(*funct=None*, ***kwargs*)

Bases: *ParameterModelling* (page 299)

Modelling framework with an array of freqencies as data space.

Variables

- **params** (*dict*) –
- **function** (*callable*) –
- **complex** (*bool*) –

__init__(*funct=None*, ***kwargs*)

Variables

- **fop** (*pg.frameworks.Modelling*) –
- **data** (*pg.DataContainer*) –
- **modelTrans** (*[pg.trans.TransLog()]*) –

Parameters

****kwargs** – fop : Modelling

property complex

drawData(*ax*, *data*, *err=None*, ***kwargs*)

property freqs

response(*params*)

Model response.

Parameters

params (*iterable*) – model

Returns
model response vector

Return type
array_like

8.4.7 pygimli.physics.sNMR

Surface nuclear magnetic resonance (NMR) data inversion

8.4.7.1 Overview

Classes

MRS (page 414)([name, verbose])	Magnetic resonance sounding (MRS) manager class.
MRS1dBlockQTModelling (page 417)(nlay, K, zvec, t[, ...])	MRS1dBlockQTModelling - pygimli modelling class for block-mono QT inversion
MRSprofile (page 417)([filename, x, dx, x0])	manager class for several MRS data along a profile for joint inversion

8.4.7.2 Classes

class pygimli.physics.sNMR.**MRS**(*name=None*, *verbose=True*, ***kwargs*)

Bases: object

Magnetic resonance sounding (MRS) manager class.

Variables

- **q**(*t,*) –
- **error**(*data,*) –
- **z**(*K,*) –
- **modelU**(*model, modelL,*) –

loadMRSI - load MRSI (MRSmatlab format) data

showCube - show any data/error/
misfit as data cube (over q and t)

showDataAndError - show data and error cubes

showKernel - show Kernel matrix

createFOP - create forward operator

createInv - create pygimli Inversion instance

run - run block-mono (alternatively smooth-mono) inversion (with bootstrap)

calcMCM – compute model covariance matrix and thus uncertainties

**splitModel – return thickness,
water content and T2* time from vector**

showResult/showResultAndFit – show inversion result (with fit)

runEA – run evolutionary algorithm (GA, PSO etc.) using inspyred

plotPopulation – plot final population of an EA run

__init__(*name=None*, *verbose=True*, ***kwargs*)

MRS init with optional data load from mrsi file

Parameters

- **name** (*string*) – Filename with load data and kernel (*.mrsi) or just data (*.mrsd)
- **verbose** (*bool*) – be verbose

:param kwargs - see *MRS.loadMRSI()* (page 416).:

calcMCM()

Compute linear model covariance matrix.

calcMCMbounds()

Compute model bounds using covariance matrix diagonals.

checkData(***kwargs*)

Check data and retrieve data and error vector.

static createFOP(*nlay*, *K*, *z*, *t*)

Create forward operator instance.

createInv(*nlay=3*, *lam=100.0*, *verbose=True*, ***kwargs*)

Create inversion instance (and fop if necessary with nlay).

genMod(*individual*)

Generate (GA) model from random vector (0-1) using model bounds.

invert(*nlay=3*, *lam=100.0*, *startvec=None*, *verbose=True*, *uncertainty=False*, ***kwargs*)

Easiest variant doing all (create fop and inv) in one call.

loadDataCube(*filename='datacube.dat'*)

Load data cube from single ascii file (old stuff)

loadDataNPZ(*filename*, ***kwargs*)

Load data and kernel from numpy gzip packed file.

The npz file contains the fields: q, t, D, (E), z, K

loadDir(*dirname*)

Load several standard files from dir (old Borkum stage).

loadErrorCube(*filename='errorcube.dat'*)

Load error cube from a single ascii file (old stuff).

loadKernel(*name=''*)

Load kernel matrix from mrsk or two bmat files.

loadKernelNPZ(*filename*, ***kwargs*)

Load data and kernel from numpy gzip packed file.

The npz file contains the fields: q, t, D, (E), z, K

loadMRSD(*filename*, *usereal=False*, *mint=0.0*, *maxt=2.0*)

Load mrsd (MRS data) file: not really used as in MRSD.

loadMRSI(*filename*, ***kwargs*)

Load data, error and kernel from mrsi or mrsd file

Parameters

- **usereal** (*bool [False]*) – use real parts (after data rotation) instead of amplitudes
- **mint/maxt** (*float [0.0/2.0]*) – minimum/maximum time to restrict time series

loadResult(*filename*)

Load inversion result from column file.

loadZVector(*filename='zkernel.vec'*)

Load the kernel vertical discretisation (z) vector.

plotEAstatistics(*fname=None*)

Plot EA statistics (best, worst, ...) over time.

plotPopulation(*maxfitness=None*, *fitratio=1.05*, *savefile=True*)

Plot fittest individuals (fitness<maxfitness) as 1d models

Parameters

- **maxfitness** (*float*) – maximum fitness value (absolute) OR
- **fitratio** (*float [1.05]*) – maximum ratio to minimum fitness

result()

Return block model results (thk, wc and T2 vectors).

run(*verbose=True*, *uncertainty=False*, ***kwargs*)

Easiest variant doing all (create fop and inv) in one call.

runEA(*nlay=None*, *eatype='GA'*, *pop_size=100*, *num_gen=100*, *runs=1*, *mp_num_cpus=8*, ***kwargs*)

Run evolutionary algorithm using the inspyred library

Parameters

- **nlay** (*int [taken from classic fop if not given]*) – number of layers
- **pop_size** (*int [100]*) – population size
- **num_gen** (*int [100]*) – number of generations

- **runs** (*int [pop_size*num_gen]*) – number of independent runs (with random population)
- **eatype** (*string ['GA']*) –

  algorithm, choose among:
  'GA' - Genetic Algorithm [default] 'SA' - Simulated Annealing 'DEA' - Discrete Evolutionary Algorithm 'PSO' - Particle Swarm Optimization 'ACS' - Ant Colony Strategy 'ES' - Evolutionary Strategy

saveFigs(*basename=None*, *extension='pdf'*)

Save all figures to (pdf) files.

saveResult(*filename*)

Save inversion result to column text file for later use.

setBoundaries()

Set parameter boundaries for inversion.

showCube(*ax=None*, *vec=None*, *islog=None*, *clim=None*, *clab=None*)

Plot any data (or response, error, misfit) cube nicely.

showDataAndError(*figsize=(10, 8)*, *show=False*)

Show data cube along with error cube.

showKernel(*ax=None*)

Show the kernel as matrix (Q over z).

showResult(*figsize=(10, 8)*, *save=''*, *fig=None*, *ax=None*)

Show theta(z) and T2*(z) (+uncertainties if there).

showResultAndFit(*figsize=(12, 10)*, *save=''*, *plotmisfit=False*, *maxdep=0*, *clim=None*)

Show ec(z), T2*(z), data and model response.

static simulate(*model*, *K*, *z*, *t*)

Do synthetic modelling.

splitModel(*model=None*)

Split model vector into d, theta and T2*.

class pygimli.physics.sNMR.**MRS1dBlockQTModelling**(*nlay*, *K*, *zvec*, *t*, *verbose=False*)

Bases: ModellingBaseMT__

MRS1dBlockQTModelling - pygimli modelling class for block-mono QT inversion

f=MRS1dBlockQTModelling(lay, KR, KI, zvec, t, verbose = False)

__init__(*nlay*, *K*, *zvec*, *t*, *verbose=False*)

Constructor with number of layers, kernel, z and t vectors.

response(*par*)

Yield model response cube as vector.

class pygimli.physics.sNMR.**MRSprofile**(*filename=None*, *x=None*, *dx=1*, *x0=0*, ***kwargs*)

Bases: object

manager class for several MRS data along a profile for joint inversion

Variables

- **mrs** (*list of MRS objects (single soundings)*) –
- **x** (*list of positions for the soundings*) –

load - load mrs files from a directory

set X - set x vector

showData - show MRS data

independentBlock1dInversion - perform independent 1D block inversion

block1dInversion - 1D block inversion of all data sets together

blockLCInversion - 1D block laterally constrained inversion of all data

printFits - print total misfit (chi^2, rms) and individual values

showModel - show LCI model

__init__(*filename=None*, *x=None*, *dx=1*, *x0=0*, ***kwargs*)

Initialize profile object by mrs objects and optional positions.

Parameters

- **filename** (*list of str | str*) – list of files OR filenames(with *) OR directory to load
- **x** (*iterable*) – position vector of individual soundings
- **x0** (*float [0]*) – starting position
- **dx** (*position [1]*) – position increment

block1dInversion(*nlay=2*, *lam=100.0*, *show=False*, *verbose=True*, *uncertainty=False*)

Invert all data together by a 1D model (more general solution).

block1dInversionOld(*nlay=2*, *startModel=None*, *verbose=True*, *uncertainty=False*, ***kwargs*)

Invert all data together by one 1D model (variant 1 - all equal).

blockLCInversion(*nlay=2*, *startModel=None*, ***kwargs*)

Laterally constrained (piece-wise 1D) block inversion.

independentBlock1dInversion(*nlay=2*, *lam=100*, *startModel=None*)

Independent inversion of all soundings.

Parameters

- **nlay** (*int [2]*) – number of layers
- **lam** (*float*) – regularization parameter
- **startModel** (*array/vector*) – starting model (see MRS.run parameters)

load(*filenames*, ***kwargs*)

load mrs files in a list of (single) MRS handlers filename can be a list of mrsi files or a directory to search Additional parameters: usereal, mint, maxt (see MRS.load)

loadKernel(*kernelfile*)

load one kernel file for all soundings

printFits()

Show single fits and total fit.

saveFigs(*basename='out'*, *extension='pdf'*)

Save all figures to (pdf) files.

setX(*x=None*, *x0=0*, *dx=1*)

define positions for soundings and sort accordingly

show1dModel()

Show 1D model (e.g. of joint block inversion).

showData(*figsize=(15, 10)*, *nc=0*, *nr=0*, *clim=None*)

show all data cubes in subplots

showFits(*ax=None*)

Show chi-square and rms fits of individual soundings.

showInitialValues()

show initial values of whole profile

showModel(*showFit=0*, *cmap='Spectral'*, *figsize=(13, 12)*, *wlim=(0, 0.5)*, *tlim=(0.05, 0.5)*)

Show 2d model as stitched 1d models along with fit.

showT2(*tlim=(0.05, 0.5)*, *ax=None*, *cmap='Spectral'*, *title='T_2^* (s)'*)

Show relaxation time distribution as stitched model section.

showWC(*wlim=(0, 0.5)*, *ax=None*, *cmap='Spectral'*, *title='$\\theta$ (-)'*)

Show water content distribution as stitched model section.

8.4.8 pygimli.physics.traveltime

Refraction seismics or first arrival traveltime calculations.

8.4.8.1 Overview

Functions

createCrossholeData (page 420)(sensors)	Create crosshole scheme assuming two boreholes with equal sensor numbers.
createGradientModel2D (page 421)(data, mesh, vTop, vBot)	Create 2D velocity gradient model.
createRAData (page 421)(sensors[, shotDistance])	Create a refraction data container.
drawFirstPicks (page 421)(ax, data[, tt, plotva])	Plot first arrivals as lines.
drawTravelTimeData (page 421)(ax, data[, t])	Draw first arrival traveltime data into mpl ax a.
drawVA (page 422)(ax, data[, vals, usePos, pseudosection])	Draw apparent velocities as matrix into an axis.
load (page 422)(fileName[, verbose])	Shortcut to load TravelTime data.
shotReceiverDistances (page 422)(data[, full])	Return vector of all distances (in m) between shot and receiver.
showVA (page 422)(data[, usePos, ax])	Show apparent velocity as image plot.
simulate (page 423)(mesh, scheme[, slowness, vel])	Manager for refraction seismics (traveltime tomography).

Classes

DataContainerTT (page 423)([data])	Data Container for traveltime.
Manager (page 423)	alias of *TravelTimeManager* (page 425)
RefractionNLayer (page 423)([offset, nlay, vbase, verbose])	Forward operator for 1D Refraction seismic with layered model.
RefractionNLayerFix1stLayer (page 423)([offset, nlay, ...])	FOP for 1D Refraction seismic with layered model (e.g.
TravelTimeDijkstraModelling (page 424)(**kwargs)	Forward modelling class for traveltime using Dijsktras method.
TravelTimeManager (page 425)([data])	Manager for refraction seismics (traveltime tomography).

8.4.8.2 Functions

pygimli.physics.traveltime.**createCrossholeData**(*sensors*)

Create crosshole scheme assuming two boreholes with equal sensor numbers.

Parameters

sensors (*array (Nx2)*) – Array with position of sensors.

Returns

scheme – Data container with *sensors* predefined sensor indices ‘s’ and ‘g’ for shot and receiver numbers.

Return type

DataContainer

Examples using `pygimli.physics.traveltime.createCrossholeData`

- *Crosshole traveltime tomography* (page 40)

`pygimli.physics.traveltime.`**`createGradientModel2D`**(*data*, *mesh*, *vTop*, *vBot*, *flat=False*)

Create 2D velocity gradient model.

Creates a smooth, linear, starting model that takes the slope of the topography into account. This is done by fitting a straight line and using the distance to that as the depth value. Known as "The Marcus method"

Parameters

- **data** (*pygimli DataContainer*) – The topography list is in here.
- **mesh** (*pygimli.Mesh*) – The parametric mesh used for the inversion
- **vTop** (*float*) – The velocity at the surface of the mesh
- **vBot** (*float*) – The velocity at the bottom of the mesh

Returns

model – A numpy array with slowness values that can be used to start the inversion.

Return type

pygimli Vector, length M

`pygimli.physics.traveltime.`**`createRAData`**(*sensors*, *shotDistance=1*)

Create a refraction data container.

Default data container for shot and geophon at every sensor position. Predefined sensor indices's 's' as shot position and 'g' as geophon position.

Parameters

- **sensors** (*ndarray | R3Vector*) – Geophon and shot positions (same)
- **shotDistances** (*int [1]*) – Distance between shot indices.

Returns

data – Data container with predefined sensor indices 's' and 'g' for shot and receiver numbers.

Return type

DataContainer

Examples using `pygimli.physics.traveltime.createRAData`

- *2D Refraction modeling and inversion* (page 34)
- *Refraction in 3D* (page 57)
- *Petrophysical joint inversion* (page 153)

`pygimli.physics.traveltime.`**`drawFirstPicks`**(*ax*, *data*, *tt=None*, *plotva=False*, ***kwargs*)

Plot first arrivals as lines.

Examples using `pygimli.physics.traveltime.drawFirstPicks`

- *Field data inversion ("Koenigsee")* (page 53)

pygimli.physics.traveltime.**drawTravelTimeData**(*ax*, *data*, *t=None*)

Draw first arrival traveltime data into mpl ax a.

data of type pg.DataContainer must contain sensorIdx 's' and 'g' and thus being numbered internally [0..n)

pygimli.physics.traveltime.**drawVA**(*ax*, *data*, *vals=None*, *usePos=True*, *pseudosection=False*, ***kwargs*)

Draw apparent velocities as matrix into an axis.

Parameters

- **ax** (*mpl.Axes*) –
- **data** (*pg.DataContainer()*) – Datacontainer with 's' and 'g' Sensorindieces and 't' traveltimes.
- **usePos** (*bool [True]*) – Use sensor positions for axes tick labels
- **pseudosection** (*bool [False]*) – Show in pseudosection style.
- **vals** (*iterable*) – Traveltimes, if None data need to contain 't' values.

pygimli.physics.traveltime.**load**(*fileName*, *verbose=False*, ***kwargs*)

Shortcut to load TravelTime data.

Import Data and try to assume the file format.

TODO

- add RHL class importer

Parameters
fileName (*str*) –

Returns
data

Return type
pg.DataContainer

pygimli.physics.traveltime.**shotReceiverDistances**(*data*, *full=True*)

Return vector of all distances (in m) between shot and receiver. for each 's' and 'g' in data.

Parameters

- **data** (*pg.DataContainerERT*) –
- **full** (*bool [False]*) – Get distances between shot and receiver position when full is True or only form x coordinate if full is False

Returns
dists – Array of distances

Return type
array

pygimli.physics.traveltime.**showVA**(*data*, *usePos=True*, *ax=None*, ***kwargs*)

Show apparent velocity as image plot.

Parameters

data (*pg.DataContainer()*) – Datacontainer with 's' and 'g' Sensorindieces and 't' traveltimes.

Examples using `pygimli.physics.traveltime.showVA`

- *Crosshole traveltime tomography* (page 40)

`pygimli.physics.traveltime.`**simulate**(*mesh*, *scheme*, *slowness=None*, *vel=None*, ***kwargs*)

Manager for refraction seismics (traveltime tomography).

TODO Document main members and use default MethodManager interface e.g., self.inv, self.fop, self.paraDomain, self.mesh, self.data

Examples using `pygimli.physics.traveltime.simulate`

- *Crosshole traveltime tomography* (page 40)
- *Petrophysical joint inversion* (page 153)

8.4.8.3 Classes

class `pygimli.physics.traveltime.`**DataContainerTT**(*data=None*, ***kwargs*)

Bases: `DataContainer`

Data Container for traveltime.

__init__(*data=None*, ***kwargs*)

Initialize empty data container, load or copy existing.

`pygimli.physics.traveltime.`**Manager**

alias of *TravelTimeManager* (page 425)

class `pygimli.physics.traveltime.`**RefractionNLayer**(*offset=0*, *nlay=3*, *vbase=1100*, *verbose=True*)

Bases: `ModellingBaseMT__`

Forward operator for 1D Refraction seismic with layered model.

__init__(*offset=0*, *nlay=3*, *vbase=1100*, *verbose=True*)

Init forward operator. Model are velocity increases.

Parameters

- **offset** (*iterable*) – vector of offsets between shot and geophone
- **nlay** (*int*) – number of layers
- **vbase** (*float*) – base velocity (all values are above)

response(*model*)

Return forward response f(m).

thkVel(*model*)

Return thickness and velocity vectors from model.

class pygimli.physics.traveltime.**RefractionNLayerFix1stLayer**(*offset=0*, *nlay=3*, *v0=1465*, *d0=200*, *muteDirect=False*, *verbose=True*)

Bases: ModellingBaseMT__

FOP for 1D Refraction seismic with layered model (e.g. water layer).

__init__(*offset=0*, *nlay=3*, *v0=1465*, *d0=200*, *muteDirect=False*, *verbose=True*)

Init forward operator for velocity increases with fixed 1st layer.

Parameters

- **offset** (*iterable*) – vector of offsets between shot and geophone
- **nlay** (*int*) – number of layers
- **v0** (*float*) – First layer velocity (at the same time base velocity)
- **d0** (*float*) – Depth of first layer in meter.
- **muteDirect** (*bool [False]*) – Mute the direct arrivels fron the first layer.

response(*model*)

Return forward response f(m).

thkVel(*model*)

Return thickness and velocity vectors from model.

class pygimli.physics.traveltime.**TravelTimeDijkstraModelling**(***kwargs*)

Bases: *MeshModelling* (page 291)

Forward modelling class for traveltime using Dijsktras method.

__init__(***kwargs*)

Variables

- **fop** (*pg.frameworks.Modelling*) –
- **data** (*pg.DataContainer*) –
- **modelTrans** (*[pg.trans.TransLog()]*) –

Parameters

****kwargs** – fop : Modelling

createJacobian(*par*)

Create Jacobian (way matrix).

createRefinedFwdMesh(*mesh*)

Refine the current mesh for higher accuracy.

This is called automatic when accesing self.mesh() so it ensures any effect of changing region properties (background, single).

createStartModel(*dataVals*)

Create a starting model from data values (gradient or constant).

property dijkstra

Return current Dijkstra graph associated to mesh and model.

drawData(*ax*, *data=None*, ***kwargs*)

Draw the data (as apparent velocity crossplot by default).

Parameters

data (*pg.DataContainer*) –

drawModel(*ax*, *model*, ***kwargs*)

Draw the model.

regionManagerRef()

Region manager reference (core Dijkstra has an own!).

response(*par*)

Return forward response (simulated traveltimes).

setDataPost(*data*)

Set data after forward operator has been initalized.

setMeshPost(*mesh*)

Set mesh after forward operator has been initalized.

way(*s*, *g*)

Return node indices for the way from the shot to the receiver.

The index is based on the given data, mesh and last known model.

class pygimli.physics.traveltime.**TravelTimeManager**(*data=None*, ***kwargs*)

Bases: *MeshMethodManager* (page 290)

Manager for refraction seismics (traveltime tomography).

TODO Document main members and use default MethodManager interface e.g., self.inv, self.fop, self.paraDomain, self.mesh, self.data

__init__(*data=None*, ***kwargs*)

Create an instance of the Traveltime manager.

Parameters

data (GIMLI::DataContainer | str) – You can initialize the Manager with data or give them a dataset when calling the inversion.

applyMesh(*mesh*, *secNodes=None*, *ignoreRegionManager=False*)

Apply mesh, i.e. set mesh in the forward operator class.

checkData(*data*)

Return data from container.

checkError(*err*, *dataVals*)

Return relative error.

createForwardOperator(***kwargs*)

Create default forward operator for Traveltime modelling.

Your want your Manager use a special forward operator you can add them here on default Dijkstra is used.

createMesh(*data=None*, ***kwargs*)

Create default inversion mesh.

Inversion mesh for traveltime inversion does not need boundary region.

Parameters

data (*DataContainer*) – Data container to read sensors from.

Keyword Arguments

` (*Forwarded to*) –

drawRayPaths(*ax*, *model=None*, *rayPaths=None*, ***kwargs*)

Draw the the ray paths for model or last model.

If model is not specified, the last calculated Jacobian is used.

Parameters

- **rayPaths** (*list of np.array*) – x/y column array with ray point positions
- **model** (*array*) – Velocity model for which to calculate and visualize ray paths (the default is model for last Jacobian calculation in self.velocity).
- **ax** (*matplotlib.axes object*) – To draw the model and the path into.
- ****kwargs** (*type*) – Additional arguments passed to LineCollection (alpha, linewidths, color, linestyles).

Returns

lc

Return type

matplotlib.LineCollection

getRayPaths(*model=None*)

Compute ray paths.

If model is not specified, the last calculated Jacobian is used.

Parameters

model (*array*) – Velocity model for which to calculate and visualize ray paths (the default is model for last Jacobian calculation in self.velocity).

Return type

list of two-column array holding x and y positions

invert(*data=None*, *useGradient=True*, *vTop=500*, *vBottom=5000*, *secNodes=2*, ***kwargs*)

Invert data.

Parameters

- **data** (*pg.DataContainer()*) – Data container with at least SensorIndieces ‘s g’ and data values ‘t’ (traveltime in ms) and ‘err’ (absolute error in ms)
- **useGradient** (*bool [True]*) – Use a gradient like starting model suited for standard flat earth cases. [Default] For cross tomography geometry you should set this to False for a non gradient startind model.

- **vTop** (*float*) – Top velocity for gradient stating model.
- **vBottom** (*float*) – Bottom velocity for gradient stating model.
- **secNodes** (*int [2]*) – Amount of secondary nodes used for ensure accuracy of the forward operator.

Keyword Arguments

kwargs (****) – Inversion related arguments: See *pygimli.frameworks.MeshMethodManager.invert* (page 290)

load(*fileName*)

Load any supported data file.

rayCoverage()

Ray coverage, i.e. summed raypath lengths.

saveResult(*folder=None*, *size=(16, 10)*, *verbose=False*, ***kwargs*)

Save the results in a specified (or date-time derived) folder.

Saved items are:

- Resulting inversion model
- Velocity vector
- Coverage vector
- Standardized coverage vector
- Mesh (bms and vtk with results)

Parameters

- **path** (*str[None]*) – Path to save into. If not set the name is automatically created
- **size** (*(float, float) (16,10)*) – Figure size.

Keyword Arguments

showResults (*Will be forwarded to*) –

Returns

Name of the result path.

Return type

str

showCoverage(*ax=None*, *name='coverage'*, ***kwargs*)

Show the ray coverage in log-scale.

showRayPaths(*model=None*, *ax=None*, ***kwargs*)

Show the model with ray paths for given model.

If not model specified, the last calculated Jacobian is taken.

Parameters

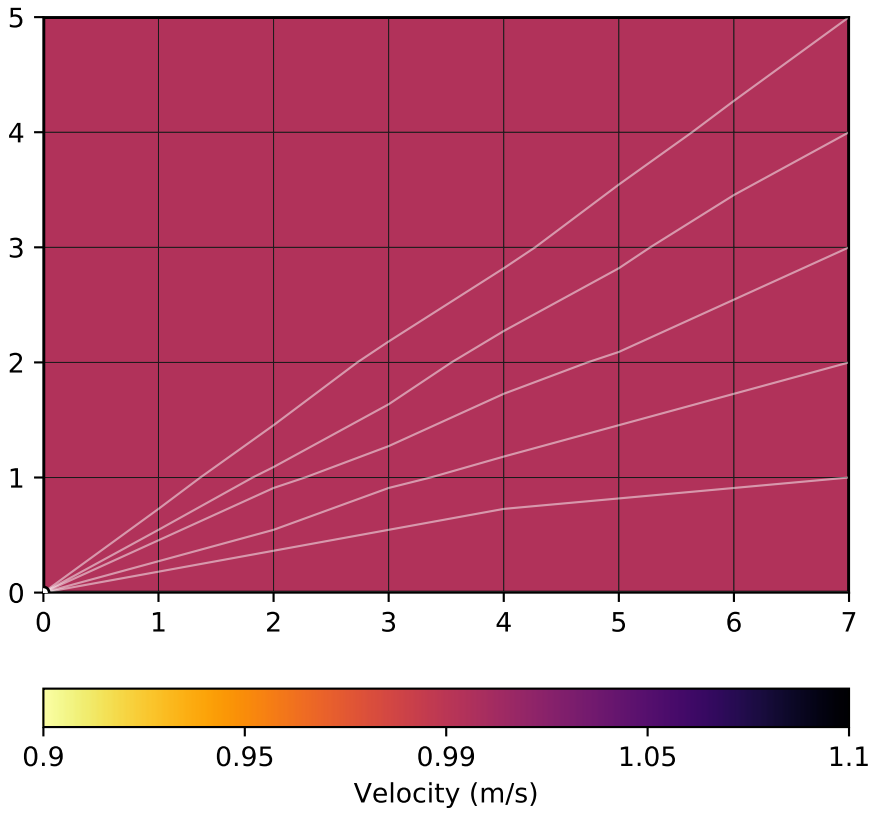
- **model** (*array*) – Velocity model for which to calculate and visualize ray paths (the default is model for last Jacobian calculation in self.velocity).

- **ax** (*matplotlib.axes object*) – To draw the model and the path into.
- ****kwargs** (*type*) – forward to drawRayPaths

Returns

- **ax** (*matplotlib.axes object*)
- **cb** (*matplotlib.colorbar object (only if model is provided)*)

```
>>> # No reason to import matplotlib
>>> import pygimli as pg
>>> from pygimli.physics import TravelTimeManager
>>> from pygimli.physics.traveltime import createRAData
>>>
>>> x, y = 8, 6
>>> mesh = pg.createGrid(x, y)
>>> data = createRAData([(0,0)] + [(x, i) for i in range(y)],
...                     shotDistance=y+1)
>>> data.set("t", pg.Vector(data.size(), 1.0))
>>> tt = TravelTimeManager()
>>> tt.fop.setData(data)
>>> tt.applyMesh(mesh, secNodes=10)
>>> ax, cb = tt.showRayPaths(showMesh=True, diam=0.1)
```



simulate(*mesh*, *scheme*, *slowness=None*, *vel=None*, *seed=None*, *secNodes=2*, *noiseLevel=0.0*, *noiseAbs=0.0*, ***kwargs*)

Simulate traveltime measurements.

Perform the forward task for a given mesh, a slowness distribution (per cell) and return data (traveltime) for a measurement scheme.

Parameters

- **mesh** (GIMLI::Mesh) – Mesh to calculate for or use the last known mesh.
- **scheme** (GIMLI::DataContainer) – Data measurement scheme needs 's' for shot and 'g' for geophone data token.
- **slowness** (*array(mesh.cellCount()) | array(N, mesh.cellCount())*) – Slowness distribution for the given mesh cells can be:
  - a single array of len mesh.cellCount()
  - a matrix of N slowness distributions of len mesh.cellCount()
  - a res map as [[marker0, res0], [marker1, res1], ...]
- **vel** (*array(mesh.cellCount()) | array(N, mesh.cellCount())*) – Velocity distribution for the given mesh cells. Will overwrite given slowness.
- **secNodes** (*int [2]*) – Number of refinement nodes to increase accuracy of the forward calculation.
- **noiseLevel** (*float [0.0]*) – Add relative noise to the simulated data. noiseLevel*100 in %
- **noiseAbs** (*float [0.0]*) – Add absolute noise to the simulated data in ms.
- **seed** (*int [None]*) – Seed the random generator for the noise.

Keyword Arguments

- **returnArray** (*[False]*) – Return only the calculated times.
- **verbose** (*[self.verbose]*) – Overwrite verbose level.
- ****kwargs** – Additional kwargs ...

Returns

t – The resulting simulated travel time values. Either one column array or matrix in case of slowness matrix.

Return type

array(N, data.size()) | DataContainer

standardizedCoverage()

Standardized coverage vector (0|1) using neighbor info.

property velocity

Return velocity vector (the inversion model).

8.5 pygimli.solver

General physics independent solver interface.

8.5.1 Overview

Functions

anisotropyMatrix (page 432)(*args, **kwargs)	
applyDirichlet (page 432)(mat, rhs, uDirIndex, uDirichlet)	This should be moved directly into the core
assembleBC (page 432)(bc, mesh, mat, rhs[, time, ...])	Shortcut to apply all boundary conditions.
assembleDirichletBC (page 432)(mat, boundaryPairs[, ...])	Apply Dirichlet boundary condition.
assembleLoadVector (page 432)(mesh, f[, userData])	Assemble the load vector.
assembleNeumannBC (page 432)(rhs, boundaryPairs[, ...])	Apply Neumann condition to the system matrix S.
assembleRobinBC (page 433)(mat, boundaryPairs[, rhs, ...])	Apply Robin boundary condition.
boundaryIdsFromDictKey (page 433)(mesh, key[, outside])	Find all boundaries matching a dictionary key.
cellValues (page 434)(mesh, arg, **kwargs)	Get a value for each cell.
checkCFL (page 435)(times, mesh, vMax[, verbose])	Check Courant-Friedrichs-Lewy condition.
constitutiveMatrix (page 435)(*args, **kwargs)	
crankNicolson (page 435)(times, S, I[, f, u0, theta, ...])	Generic Crank Nicolson solver for time dependend problems.
createAnisotropyMatrix (page 436)(lon, trans, theta)	Create anisotropy matrix with desired properties.
createConstitutiveMatrix (page 436)([lam, mu, E, nu, ...])	Create constitutive matrix for 2 or 3D isotropic media.
createForceVector (page 437)(mesh, f[, userData])	Create a right hand side vector for vector valued solutions.
createLoadVector (page 437)(mesh[, f, userData])	Create right hand side vector based on the given mesh and load values (scalar solution) or force vectors (vector value solution).
createMassMatrix (page 437)(mesh[, b])	Create the mass matrix.
createStiffnessMatrix (page 438)(mesh[, a, isVector])	Create the Stiffness matrix.
deepcopy (page 438)(x[, memo, _nil])	Deep copy operation on arbitrary Python objects.
div (page 438)(mesh, v)	Return the discrete interpolated divergence field.
divergence (page 439)(mesh[, func, normMap, order])	Divergence for callable function func((x,y,z)).

continues on next page

Table 3 – continued from previous page

generateBoundaryValue (page 440)(boundary, arg[, time, ...])	Generate a value for the given Boundary.
getDirichletMap (page 440)(mat, boundaryPairs[, time, ...])	Get map of index: dirichlet value
grad (page 441)(mesh, u[, r])	Return the discrete interpolated gradient **v** for a given scalar field **u**.
greenDiffusion1D (page 442)(x[, t, a, dim])	Greens function for diffusion operator.
identity (page 443)(dom[, start, end, scale])	Create identity matrix.
intDomain (page 443)(u[, mesh])	Return integral over nodal solution u.
linSolve (page 444)(mat, b[, solver, verbose])	Direct linear solution after **x** using core LinSolver.
normH1 (page 444)(u[, mat, mesh])	Create (H1) norm for the finite element space.
normL2 (page 444)(u[, mat, mesh])	Create Lebesgue (L2) norm for finite element space.
parseArgPairToBoundaryArray (page 445)(pair, mesh)	Parse boundary related pair argument to create a list of [GIMLI::Boundary, value|callable].
parseArgToArray (page 446)(arg, nDof[, mesh, userData])	Parse array related arguments to create a valid value array.
parseArgToBoundaries (page 446)(args, mesh)	Parse boundary related arguments to create a valid boundary value list: [GIMLI::Boundary, value|callable]
*parseDictKey_* (page 448)(key, markers)	
parseMapToCellArray (page 448)(attributeMap, mesh[, ...])	Parse a value map to cell attributes.
parseMarkersDictKey (page 449)(key, markers)	Parse dictionary key of type str to marker list.
showSparseMatrix (page 449)(mat[, full])	Show the content of a sparse matrix.
solve (page 449)(mesh, **kwargs)	Solve partial differential equation.
solveFiniteElements (page 450)(mesh[, a, b, f, bc, ...])	Solve partial differential equation with Finite Elements.
solveFiniteVolume (page 452)(mesh[, a, b, f, fn, vel, ...])	Solve partial differential equation with Finite Volumes.
triDiagToeplitz (page 453)(dom, a, l, r[, start, end])	Create tri-diagonal Toeplitz matrix.

Classes

LinSolver (page 454)([mat, solver, verbose])	Proxy class for the solution of linear systems of equations.
RungeKutta (page 454)(solver[, verbose])	TODO DOCUMENT ME
WorkSpace (page 454)()	

8.5.2 Functions

pygimli.solver.**anisotropyMatrix**(**args*, ***kwargs*)

pygimli.solver.**applyDirichlet**(*mat*, *rhs*, *uDirIndex*, *uDirichlet*)

This should be moved directly into the core

pygimli.solver.**assembleBC**(*bc*, *mesh*, *mat*, *rhs*, *time=None*, *userData={}*, *dofOffset=0*, *nCoeff=1*)

Shortcut to apply all boundary conditions.

Shortcut to apply all boundary conditions will only forward to appropriate assemble functions.

Returns

- **map{id** (*uDirichlet}: Map of index to Dirichlet value.*)
- *None*

pygimli.solver.**assembleDirichletBC**(*mat*, *boundaryPairs*, *rhs=None*, *time=0.0*, *userData={}*, *nodePairs=None*, *dofOffset=0*, *nCoeff=1*, *dofPerCoeff=None*)

Apply Dirichlet boundary condition.

Parameters

rhs (Vector) – Right hand side vector of the system equation will bet set to u_{D}

Examples using pygimli.solver.assembleDirichletBC

- *Basics of Finite Element Analysis* (page 214)

pygimli.solver.**assembleLoadVector**(*mesh*, *f*, *userData={}*)

Assemble the load vector. See createLoadVector.

pygimli.solver.**assembleNeumannBC**(*rhs*, *boundaryPairs*, *nDim=1*, *time=0.0*, *userData={}*, *dofOffset=0*, *nCoeff=1*, *dofPerCoeff=None*)

Apply Neumann condition to the system matrix S.

Apply Neumann condition to the system matrix S. The right hand side values for g can be given for each boundary element individually by setting proper boundary pair arguments.

$$\frac{\partial u(\mathbf{r},t)}{\partial \mathbf{n}} = \mathbf{n}\nabla u(\mathbf{r},t) = g \quad \text{for} \quad \mathbf{r} \quad \text{on} \quad \delta\Omega = \Gamma_{\text{Neumann}}$$

Parameters

- **rhs** (Vector) – Right hand side vector of length node count.
- **boundaryPair** (*list()*) – List of pairs [GIMLI::Boundary, g]. The value *g* will assigned to the nodes of the boundaries. Later assignment overwrites prior.

  g need to be a scalar value (float or int) or a value generator function that will be executed at run time.

  See pygimli.solver.solver.parseArgToBoundaries and *Modelling with Boundary Conditions* (page 219) for example syntax,

- **nDim** (*int [1]*) – Number of dimensions for vector valued problems. The rhs array need to have the correct size, i.e., number of Nodes * mesh.dimension()
- **time** (*float*) – Will be forwarded to value generator.
- **userData** (*class*) – Will be forwarded to value generator.
- **dofOffset** (*int[0]*) – Offset for matrix index.

pygimli.solver.**assembleRobinBC**(*mat*, *boundaryPairs*, *rhs=None*, *time=0.0*, *userData={}*, *dofOffset=0*, *nCoeff=1*, *dofPerCoeff=None*)

Apply Robin boundary condition.

Apply Robin boundary condition to the system matrix and the rhs vector:

$$\frac{\partial u(\mathbf{r},t)}{\partial \mathbf{n}} = \alpha(u_0 - u) \quad \text{or}$$
$$\beta\frac{\partial u(\mathbf{r},t)}{\partial \mathbf{n}} + \alpha u = \gamma$$
$$\text{for} \quad \mathbf{r} \quad \text{on} \quad \delta\Omega = \Gamma_{\text{Robin}}$$

Parameters

- **mat** (GIMLI::SparseMatrix) – System matrix of the system equation.
- **boundaryPair** (*list*) –

  List of pairs [GIMLI::Boundary, a, u_0 **|** α, β, γ**].**

  The values will assigned to the nodes of the boundaries. Later assignment overwrites prior.

  Values can be a single value for α or a, two values will be interpreted as a, u_0, and three values will be α, β, γ. Also generator (callable) is possible which will be executed at runtime See `pygimli.solver.solver.parseArgToBoundaries` *Modelling with Boundary Conditions* (page 219) or testing/test_FEM.py for example syntax.
- **time** (*float*) – Will be forwarded to value generator.
- **userData** (*dict*) – Will be forwarded to value generator.
- **dofOffset** (*int[0]*) – Offset for matrix index.

pygimli.solver.**boundaryIdsFromDictKey**(*mesh*, *key*, *outside=True*)

Find all boundaries matching a dictionary key.

Variables

- **mesh** (GIMLI::Mesh) –
- **key** (*str|int*) – Representation for boundary marker. Will be parsed by `pygimli.solver.solver.parseMarkersDictKey`
- **outside** (*bool [True]*) – Only select outside boundaries.

Returns

dict

Return type
{marker, [boundary.id()]}

pygimli.solver.**cellValues**(*mesh*, *arg*, ***kwargs*)

Get a value for each cell.

Returns a array or vector of length mesh.cellCount() based on arg. The preferable arg is a dictionary for the cell marker and the appropriate cell value. The designated value can be calculated using a callable(cell, **kwargs), which is called on demand.

Variables

- **mesh** (GIMLI::Mesh) – Used if arg is callable
- **arg** (*float | int | complex | ndarray | iterable | callable | dict*) – Argument to be parsed as cell data. If arg is a dictionary, its key will be interpreted as cell marker:

  Dictionary is key: value. Value can be float, int, complex or ndarray. The last for anistropic or elastic tensors.

  Key can be integer for cell marker or str, which will be interpreted as splice or list. See examples or *py:mod:pygimli.solver.parseMarkersDictKey*.

  Iterable of length mesh.nodeCount() to be interpolated to cell centers.
- **userData** (*class*) – Used if arg is callable

Returns
ret – Array of desired length filled with the appropriate values.

Return type
GIMLI::RVector | ndarray(mesh.cellCount(), xx)

Examples

```
>>> import pygimli as pg
>>> mesh = pg.createGrid(x=range(5))
>>> mesh.setCellMarkers([1, 1, 2, 2])
>>> print(mesh.cellCount())
4
>>> print(pg.solver.cellValues(mesh, [1, 2, 3, 4]))
[1, 2, 3, 4]
>>> print(pg.solver.cellValues(mesh, {1:1.0, 2:10}))
[1.0, 1.0, 10, 10]
>>> print(pg.solver.cellValues(mesh, {':':2.0}))
[2.0, 2.0, 2.0, 2.0]
>>> print(pg.solver.cellValues(mesh, {'0:2':3.0}))
[3.0, 3.0, None, None]
>>> print(pg.solver.cellValues(mesh, np.ones(mesh.nodeCount())))
4 [1.0, 1.0, 1.0, 1.0]
>>> print(np.array(pg.solver.cellValues(mesh, {'1:3' : np.diag([1.0, 2.0])}
↪)))
[[[1. 0.]
  [0. 2.]]
```

(continues on next page)

(continued from previous page)

```
 [[1. 0.]
  [0. 2.]]

 [[1. 0.]
  [0. 2.]]

 [[1. 0.]
  [0. 2.]]]
>>> print(np.array(pg.solver.cellValues(mesh, {':' : pg.core.CMatrix(2, 2)}
↪)))
[[[0.+0.j 0.+0.j]
  [0.+0.j 0.+0.j]]

 [[0.+0.j 0.+0.j]
  [0.+0.j 0.+0.j]]

 [[0.+0.j 0.+0.j]
  [0.+0.j 0.+0.j]]

 [[0.+0.j 0.+0.j]
  [0.+0.j 0.+0.j]]]
>>> print(pg.solver.cellValues(mesh, {'1,2':1 + 1j*2.0}))
[(1+2j), (1+2j), (1+2j), (1+2j)]
>>> def cellVal(c, b=1):
...     return c.center()[0]*b
>>> t = pg.solver.cellValues(mesh, {':' : cellVal})
>>> print([t[c.id()](c) for c in mesh.cells()])
[0.5, 1.5, 2.5, 3.5]
```

pygimli.solver.**checkCFL**(*times*, *mesh*, *vMax*, *verbose=False*)

Check Courant-Friedrichs-Lewy condition.

For advection and flow problems. CFL Number should be lower then 1 to ensure stability.

pygimli.solver.**constitutiveMatrix**(**args*, ***kwargs*)

pygimli.solver.**crankNicolson**(*times*, *S*, *I*, *f=None*, *u0=None*, *theta=1.0*, *dirichlet=None*, *solver=None*, *progress=None*)

Generic Crank Nicolson solver for time dependend problems.

Limitations so far:

S = Needs to be constant over time (i.e. no change in coefficients) f = constant over time (would need assembling in every step)

Parameters

- **times** (*iterable(float)*) – Timeteps to solve for. Give at least 2.
- **S** (*Matrix*) – Systemmatrix holds your discrete equations and boundary conditions

- **I** (*Matrix*) – Identity matrix (FD, FV) or Masselementmatrix (FE) to handle solution vector
- **u0** (*iterable [None]*) – Starting condition. zero if not given
- **f** (*iterable (float) [None]*) – External forces. Note f might also contain compensation values due to algebraic Dirichlet correction of S
- **theta** (*float [1.0]*) –
  - 0: Backward difference scheme (implicit)
  - 1: Forward difference scheme (explicit)

  strong time steps dependency .. will be unstable for to small values * 0.5: probably best tradeoff but can also be unstable
- **dirichlet** (*dirichlet generator*) – Genertor object to applay dirichlet boundary conditions
- **solver** (LinSolver (page 454) *[None]*) – Provide a pre configured solver if you want some special.
- **progress** (*Progress [None]*) – Provide progress object if you want to see some.

Returns
Solution for each time steps

Return type
np.ndarray

pygimli.solver.**createAnisotropyMatrix**(*lon*, *trans*, *theta*)

Create anisotropy matrix with desired properties.

Anistropy tensor from longitudinal value lon, transverse value trans and the angle theta of the symmetry axis relative to the vertical after cite:WieseGreZho2015 https://www.researchgate.net/publication/249866312_Explicit_expressions_for_the_Frechet_derivatives_in_3D_a

Examples using `pygimli.solver.createAnisotropyMatrix`

- *Basics of Finite Element Analysis* (page 214)

pygimli.solver.**createConstitutiveMatrix**(*lam=None*, *mu=None*, *E=None*, *nu=None*, *dim=2*, *voigtNotation=False*)

Create constitutive matrix for 2 or 3D isotropic media.

Either give lam and mu or E and nu.

Parameters

- **lam** (*float [None]*) –
  1. Lame' constant
- **mu** (*float [None]*) –
  2. Lame' constant = G = Schubmodul
- **E** (*float [None]*) – Young's Modulus
- **nu** (*float [None]*) – Poisson's ratio

- **voigtNotation** (*bool [False]*) – Return in Voigt's notation instead of Kelvin's notation [default].

Returns
C – Either 3x3 or 6x6 matrix depending on the dimension

Return type
mat

pygimli.solver.**createForceVector**(*mesh*, *f*, *userData={}*)

Create a right hand side vector for vector valued solutions.

Parameters

- **f** (*[convertable]*) – List of rhs side options. Must be convertable to createLoadVector. See `createLoadVector`
- **rhs** (*np.array()*) – Squeezed vector of length mesh.nodeCount() * mesh.dimensions()

pygimli.solver.**createLoadVector**(*mesh*, *f=1.0*, *userData={}*)

Create right hand side vector based on the given mesh and load values (scalar solution) or force vectors (vector value solution).

Create right hand side based on the given mesh and load or force values.

Parameters
f (*float[1.0], array, callable(cell, [userData]), [f_x, f_y, f_z]*) –

- **float will be assumed as constant for all cells**
  like rhs = rhs(np.ones(mesh.cellCount() * f),
- **array of length mesh.cellCount() will be processed as load value for**
  each cell: rhs = rhs(f),
- **array of length mesh.nodeCount() is assumed to be already processed**
  correct: rhs = f
- **callable is evaluated on once for each cell and need to return a load**
  value for each cell and can have optional a userData dictionary: *f_cell = f(cell, [userData={}])* rhs = rhs(f(c, userData) for c in mesh.cells())
- **list with length of mesh.dimension() of float or array entries will**
  create a squeezed rhs for vector valued problems rhs = squeeze([rhs(f[0]), rhs(f[1]), rhs(f[2])])

Returns
rhs – Right-hand side load vector for scalar values or squeezed vector values

Return type
pg.Vector(mesh.nodeCount())

Examples using `pygimli.solver.createLoadVector`

- *Basics of Finite Element Analysis* (page 214)

pygimli.solver.**createMassMatrix**(*mesh*, *b=None*)

Create the mass matrix.

Calculates the Mass matrix (Finite element identity matrix) the given mesh.

..math::

...

Parameters

- **mesh** (GIMLI::Mesh) – Arbitrary mesh to calculate the mass element matrix. Type of base and shape functions depends on the cell types.
- **b** (*array*) – Per cell values. If None given default is 1.

Returns
A – Mass element matrix

Return type
GIMLI::RSparseMatrix

pygimli.solver.**createStiffnessMatrix**(*mesh*, *a=None*, *isVector=False*)

Create the Stiffness matrix.

Calculates the Stiffness matrix **S** for the given mesh scaled with the per cell values a.

..math::

...

Parameters

- **mesh** (GIMLI::Mesh) – Arbitrary mesh to calculate the stiffness for. Type of base and shape functions depends on the cell types.
- **a** (*iterable of type float, int, complex, RMatrix, CMatrix*) – Per cell values., e.g., physical parameter. Length of a need to be mesh.cellCount(). If None given default is 1.
- **isVector** (*bool [False]*) – We want to solve for vector valued problems. Resulting SparseMatrix is a SparseMapMatrix and have the dimension (nNodes * nDims, nNodes * nDims) with nNodes = mesh.nodeCount() and nDims = mesh.dimension().

Returns
A – Stiffness matrix, with real or complex values.

Return type
GIMLI::[C]SparseMatrix | [C]SparseMapMatrix

Examples using `pygimli.solver.createStiffnessMatrix`

- *Basics of Finite Element Analysis* (page 214)

pygimli.solver.**deepcopy**(*x*, *memo=None*, *_nil=[]*)

Deep copy operation on arbitrary Python objects.

See the module's __doc__ string for more info.

pygimli.solver.**div**(*mesh*, *v*)

Return the discrete interpolated divergence field.

Return the discrete interpolated divergence field. **u** for each cell for a given vector field **v**. First order integration via boundary center.

$$d(cells) = \nabla \cdot \vec{v}$$

$$d(c_i) = \sum_{j=0}^{N_B} \vec{v}_{B_j} \cdot \vec{n}_{B_j}$$

Parameters

- **mesh** (GIMLI::Mesh) – Discretization base, interpolation will be performed via finite element base shape functions.
- **v** (*array(N,3) | R3Vector*) – Vector field at cell centers or boundary centers

Returns

d – Array of divergence values for each cell in the given mesh.

Return type

array(M)

Examples

```
>>> import pygimli as pg
>>> import numpy as np
>>> v = lambda p: p
>>> mesh = pg.createGrid(x=np.linspace(0, 1, 4))
>>> print(pg.math.round(pg.solver.div(mesh, v(mesh.boundaryCenters())), 1e-
↪5))
3 [1.0, 1.0, 1.0]
>>> print(pg.math.round(pg.solver.div(mesh, v(mesh.cellCenters())), 1e-5))
3 [0.5, 1.0, 0.5]
>>> mesh = pg.createGrid(x=np.linspace(0, 1, 4),
...                      y=np.linspace(0, 1, 4))
>>> print(pg.math.round(pg.solver.div(mesh, v(mesh.boundaryCenters())), 1e-
↪5))
9 [2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0]
>>> divCells = pg.solver.div(mesh, v(mesh.cellCenters()))
>>> # divergence from boundary values are exact where the␣
↪divergence from
>>> # interpolated cell center values wrong due to␣
↪interpolation to boundary
>>> print(sum(divCells))
12.0
>>> mesh = pg.createGrid(x=np.linspace(0, 1, 4),
...                      y=np.linspace(0, 1, 4),
...                      z=np.linspace(0, 1, 4))
>>> print(sum(pg.solver.div(mesh, v(mesh.boundaryCenters()))))
81.0
>>> divCells = pg.solver.div(mesh, v(mesh.cellCenters()))
>>> print(sum(divCells))
54.0
```

pygimli.solver.**divergence**(*mesh*, *func=None*, *normMap=None*, *order=1*)

Divergence for callable function func((x,y,z)).

MOVE THIS to a better place

Divergence for callable function func((x,y,z)). Return sum div over boundary.

pygimli.solver.**generateBoundaryValue**(*boundary*, *arg*, *time=0.0*, *userData={}*, *expectList=False*, *nCoeff=1*)

Generate a value for the given Boundary.

Parameters

- **boundary** (GIMLI::Boundary or list of ..) – The related boundary.
- **expectList** (*bool[False]*) – Allow list values for Robin BC.
- **arg** (*convertible | iterable | callable or list of ..*) –
  - convertible into float
  - iterable of minimum length = boundary.id()
  - callable generator function

  If arg is a callable it must fulfill:

  :: arg(boundary=:gimliapi:*GIMLI::Boundary*, time=0.0, userData={})

  The callable function arg have to return appropriate values for all nodes of the boundary or one value for all nodes (scalar field only). Value can be scalar or vector field value, e.g., return force values for all nodes at a boundary to return an ndarray((nodes, dims)), e.g. 'lambda _b: np.array([[forc_x, forc_y, forc_z] for n in _b.nodes()]).T'

Returns

val – Value for all nodes of the boundary.

Return type

[float]

pygimli.solver.**getDirichletMap**(*mat*, *boundaryPairs*, *time=0.0*, *userData={}*, *nodePairs=None*, *dofOffset=0*, *nCoeff=1*, *dofPerCoeff=None*)

Get map of index: dirichlet value

Apply Dirichlet boundary condition to the system matrix S and rhs vector. The right hand side values for h can be given for each boundary element individually by setting proper boundary pair arguments.

$$u(\mathbf{r},t) = h \quad \text{for} \quad \mathbf{r} \quad \text{on} \quad \delta\Omega = \Gamma_{\text{Dirichlet}}$$

Parameters

- **mat** (GIMLI::RSparseMatrix) – System matrix of the system equation.
- **boundaryPair** (*list()*) – List of pairs [GIMLI::Boundary, h]. The value *h* will assigned to the nodes of the boundaries. Later assignment overwrites prior.

  h need to be a scalar value (float or int) or a value generator function that will be executed at runtime. See `pygimli.solver.`

solver.parseArgToBoundaries and *Modelling with Boundary Conditions* (page 219) for example syntax,

- **nodePairs** (*list() | callable*) – List of pairs [nodeID, uD]. The value uD will assigned to the nodes given there ids. This node value settings will overwrite any prior settings due to boundaryPair.
- **time** (*float*) – Will be forwarded to value generator.
- **userData** (*class*) – Will be forwarded to value generator.
- **dofOffset** (*int[0]*) – Offset for matrix index.

pygimli.solver.**grad**(*mesh*, *u*, *r=None*)

Return the discrete interpolated gradient **v** for a given scalar field **u**.

$$\mathbf{v}(\mathbf{r}_\mathcal{C}) = \nabla u(\mathbf{r}_\mathcal{N})$$

$$(\mathbf{v_x}(\mathbf{r}_\mathcal{C}), \mathbf{v_y}(\mathbf{r}_\mathcal{C}), \mathbf{v_z}(\mathbf{r}_\mathcal{C}))^\mathrm{T} = \left(\frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}, \frac{\partial u}{\partial z}\right)^\mathrm{T}$$

With $\mathcal{N} = \cup_{i=0}^{N}\mathrm{Node}_i$, $\mathcal{C} = \cup_{j=0}^{M}\mathrm{Cell}_j$, $\mathbf{u} = \{u(\mathbf{r}_i)\} \in I\!R$ and $\mathbf{r}_i = (x_i, y_i, z_i)^\mathrm{T}$

The discrete scalar field $\mathbf{u}(\mathbf{r}_\mathcal{N})$ need to be defined for each node position $\mathbf{r}_\mathcal{N}$. The resulting vector field $\mathbf{v}(\mathbf{r}_\mathcal{C})$ is defined for each cell center position $\mathbf{r}_\mathcal{C}$. If you need other positions than the cell center, provide an appropriate array of coordinates **r**.

Parameters

- **mesh** (GIMLI::Mesh) – Discretization base, interpolation will be performed via finite element base shape functions.
- **u** (*array | callable*) – Scalar field per mesh node position or an appropriate callable([[x,y,z]])
- **r** (*ndarray((M, 3)) [mesh.cellCenter()]*) – Alternative target coordinates :math:`mathbf{r} for the resulting gradient field. i.e., the positions where the vector field is defined. Default are all cell centers.

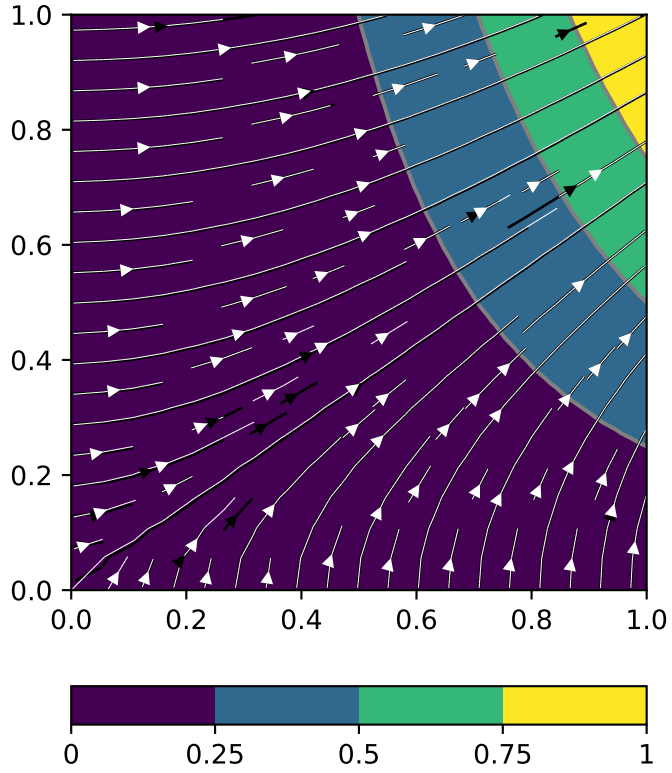
Returns

v – Resulting vector field defined on $\mathbf{v}(\mathbf{r}_\mathcal{C})$. M is number of cells or length of given alternative coordinates r.

Return type

ndarray((M, 3))

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import pygimli as pg
>>> fig, ax = plt.subplots()
>>> mesh = pg.createGrid(x=np.linspace(0, 1, 20), y=np.linspace(0, 1, 20))
>>> u = lambda p: pg.x(p)**2 * pg.y(p)
>>> _ = pg.show(mesh, u(mesh.positions()), ax=ax)
>>> _ = pg.show(mesh, [2.*pg.y(mesh.cellCenters())*pg.x(mesh.
↪cellCenters()),
...                    pg.x(mesh.cellCenters())**2], ax=ax)
>>> _ = pg.show(mesh, pg.solver.grad(mesh, u), ax=ax, color='w',
...                    linewidth=0.4)
>>> plt.show()
```



Examples using `pygimli.solver.grad`

- *3D Darcy flow* (page 136)
- *Hydrogeophysical modeling* (page 138)

`pygimli.solver.`**`greenDiffusion1D`**(*x*, *t=0*, *a=1*, *dim=1*)

Greens function for diffusion operator.

Provides the elementary solution for:

$$g(x,t) = \partial t + a\Delta$$

To find a solution for:

$$u(x,t) \quad \text{for} \quad \frac{\partial u(x,t)}{\partial t} + a\Delta u(x,t) = f(x)$$

$$x = [-x, 0, x]$$

$$u(x,t) = g(x,t) * f(x)$$

Parameters

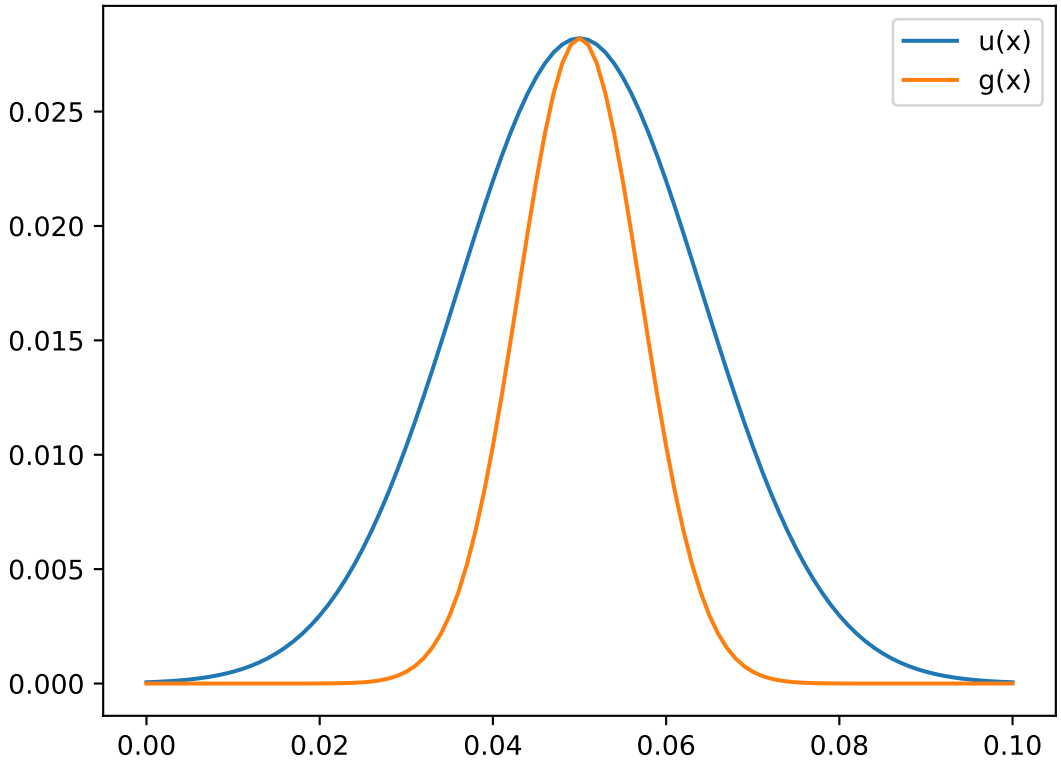
- **x** (*array_like*) – Discrete spatial coordinates. Should better be symmetric [-x, 0, x].
- **t** (*float*) – Time
- **a** (*float, optional*) – Scale for Laplacian operator
- **dim** (*int, optional*) – Spatial dimension [1]

Returns
g – Discrete Greens'function

Return type
array_like

```
>>> import numpy as np
>>> from pygimli.solver import greenDiffusion1D
>>> dx = 0.001
>>> x = np.arange(0, 0.1+dx, dx)
>>> g = greenDiffusion1D(np.hstack((-x[:0:-1], x)), t=1.0, a=1e-4)
>>> g *= dx
>>> f = np.zeros(len(x))
>>> f[int(len(x)/2)] = 1.
>>> u = np.convolve(g, f)[len(x)-1:2*len(x)-1]
```

```
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots()
>>> _ = ax.plot(x, u, label="u(x)")
>>> _ = ax.plot(x, g[::2], label="g(x)")
>>> _ = ax.legend()
>>> fig.show()
```



pygimli.solver.**identity**(*dom*, *start=0*, *end=-1*, *scale=1*)

Create identity matrix.

pygimli.solver.**intDomain**(*u*, *mesh=None*)

Return integral over nodal solution u.

$$\int_{\Omega} u$$

pygimli.solver.**linSolve**(*mat*, *b*, *solver=None*, *verbose=False*, ***kwargs*)

Direct linear solution after **x** using core LinSolver.

$$\mathbf{Ax} = \mathbf{b}$$

If **A** is symmetric, sparse and positive definite.

Parameters

- **mat** (GIMLI::RSparseMatrix, GIMLI::RSparseMapMatrix,) – numpy.array

  System matrix. Need to be symmetric, sparse and positive definite.
- **b** (*iterable array*) – Right hand side of the equation.
- **solver** (*str [None]*) – Try to choose a solver, 'pg' for pygimli core cholmod or umfpack. 'np' for numpy linalg or scipy.sparse.linalg. Automatic choosing if solver is None depending on matrixtype.
- **verbose** (*bool [False]*) – Be verbose.

Returns

x – Solution vector.

Return type

GIMLI::Vector

Examples using `pygimli.solver.linSolve`

- *Basics of Finite Element Analysis* (page 214)

pygimli.solver.**normH1**(*u*, *mat=None*, *mesh=None*)

Create (H1) norm for the finite element space.

Parameters

- **u** (*iterable*) – Node based value to compute the H1 norm for.
- **mat** (*Matrix*) – Stiffness matrix.
- **mesh** (GIMLI::Mesh) – Mesh with the FE space to generate S if necessary.

Returns

ret – $H1(u)$ norm.

Return type

float

Examples using `pygimli.solver.normH1`

- *Geoelectrics in 2.5D* (page 76)

pygimli.solver.**normL2**(*u*, *mat=None*, *mesh=None*)

Create Lebesgue (L2) norm for finite element space.

Find the L2 Norm for a solution for the finite element space. u exact solution $\mathbf{M}$ Mass matrix, i.e., Finite element identity matrix.

$$
\begin{aligned}
L2(f(x)) = ||f(x)||_{L^2} &= (\int |f(x)|^2 \mathrm{d}x)^{1/2} \\
&\approx h(\sum |f(x)|^2)^{1/2} \\
L2(u) = ||u||_{L^2} &= (\int |u|^2 \mathrm{d}x)^{1/2} \\
&\approx (\sum M(u))^{1/2} \\
e_{L2_{rel}} = \frac{L2(u)}{L2(u)} &= \frac{(\sum M(u))^{1/2}}{(\sum Mu)^{1/2}}
\end{aligned}
$$

The error for any approximated solution u_h correlates to the L2 norm of 'L2Norm(u - u_h, M)'. If you like relative values, you can also normalize this error with 'L2Norm(u - u_h, M) / L2Norm(u, M)*100'.

Parameters

- **u** (*iterable*) – Node based value to compute the L2 norm for.
- **mat** (*Matrix*) – Mass element matrix.
- **mesh** (GIMLI::Mesh) – Mesh with the FE space to generate M if necessary.

Returns

ret – $L2(u)$ norm.

Return type

float

Examples using `pygimli.solver.normL2`

- *Geoelectrics in 2.5D* (page 76)
- *Basics of Finite Element Analysis* (page 214)

`pygimli.solver.`**`parseArgPairToBoundaryArray`**(*pair*, *mesh*)

Parse boundary related pair argument to create a list of [GIMLI::Boundary, value|callable].

Parameters

- **pair** (*tuple*) –
  - [marker, arg]
  - [marker, [callable, *kwargs]]
  - [marker, [arg_x, arg_y, arg_z]]
  - [boundary, arg]
  - ['*', arg]
  - [node, arg]
  - [[marker, . . .], arg] (REMOVE ME because of bad design)
  - [[boundary,. . .], arg] (REMOVE ME because of bad design)
  - [marker, callable, *kwargs] (REMOVE ME because of bad design)
  - [[marker, . . .], callable, *kwargs] (REMOVE ME because of bad design)

arg will be parsed by `pygimli.solver.solver.generateBoundaryValue` and distributed to each boundary. Callable functions will be executed at run time. '*' is interpreted as all boundary elements with one neighboring cell

- **mesh** (GIMLI::Mesh) – Used to find boundaries by marker.

Returns
bc – [GIMLI::Boundary, value|callable]

Return type
list()

`pygimli.solver.`**parseArgToArray**(*arg*, *nDof*, *mesh=None*, *userData={}*)

Parse array related arguments to create a valid value array.

Parameters

- **arg** (*float | int | iterable | callable*) – The target array value that will be converted to an array.

  If arg is a callable with it must fulfill:

  :: arg(cell|node|boundary, userData={})

  Where MeshEntity is one of GIMLI::Cell , GIMLI::Node or GIMLI::Boundary depending on nDof, where nDof is mesh.cellCount(), mesh.nodeCount() or mesh.boundaryCount(), respectively.

- **nDof** (*int | [int]*) – Desired array size.
- **mesh** (GIMLI::Mesh) – Used if arg is callable
- **userData** (*class*) – Used if arg is callable

Returns
ret – Array of desired length filled with the appropriate values.

Return type
GIMLI::RVector

Examples using `pygimli.solver.parseArgToArray`

- *Complex-valued electrical modeling* (page 98)
- *Naive complex-valued electrical inversion* (page 104)
- *Petrophysical joint inversion* (page 153)

`pygimli.solver.`**parseArgToBoundaries**(*args*, *mesh*)

Parse boundary related arguments to create a valid boundary value list: [GIMLI::Boundary, value|callable]

Parameters

- **args** (*dict, float, callable*) – Dictionary is preferred (key=value|callable). If args is just a callable or float every outer boundary is processed with args.

  List pairs will be removed or not correct parsed for vector valued problems. Callable will be evaluated at runtime. See examples. Else see `pygimli.solver.solver.parseArgPairToBoundaryArray`

- **mesh** (GIMLI::Mesh) – Used to find boundaries by marker

Returns

boundaries – [GIMLI::Boundary, value|callable]

Return type

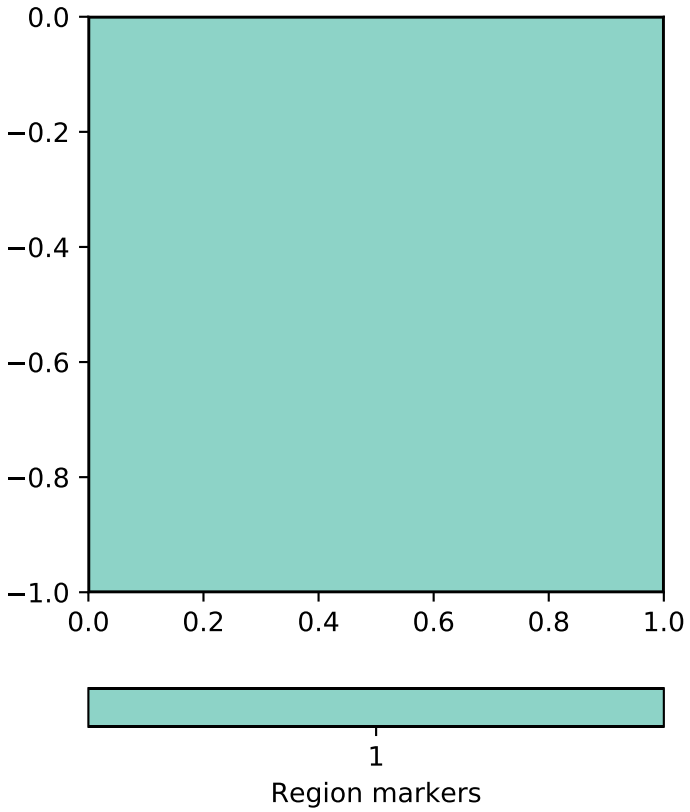
list()

```
>>> # no need to import matplotlib. pygimli show does
>>> import pygimli as pg
>>> import pygimli.meshtools as mt
>>> plc = mt.createWorld([0, 0], [1, -1], worldMarker=0)
>>> ax, _ = pg.show(plc, boundaryMarker=True)
>>> mesh = mt.createMesh(plc)
>>> # all four outer boundaries get value = 1.0
>>> b = pg.solver.parseArgToBoundaries(1.0, mesh)
>>> print(len(b))
4
>>> # all edges with marker 1 get value = 1.0
>>> b = pg.solver.parseArgToBoundaries({1: 1.0}, mesh)
>>> print(len(b))
1
>>> # same as above with marker 2 get value 2
>>> b = pg.solver.parseArgToBoundaries({'1': 1.0, 2 : 2.0}, mesh)
>>> print(len(b))
2
>>> # same as above with marker 3 get value 3
>>> b = pg.solver.parseArgToBoundaries({1:1., 2:2., 3:3.}, mesh)
>>> print(len(b))
3
>>> # Boundary values for vector valued problem
>>> b = pg.solver.parseArgToBoundaries({1:[1.0, 1.0]}, mesh)
>>> print(len(b), b[0][1])
1 [1.0, 1.0]
>>> # edges with marker 1 and 2 get value 1
>>> b = pg.solver.parseArgToBoundaries({'1,2':1.0}, mesh)
>>> print(len(b))
2
>>> b = pg.solver.parseArgToBoundaries({'1, 2, 3': 1.0}, mesh)
>>> print(len(b))
3
>>> b = pg.solver.parseArgToBoundaries({'1:4':1.0, 4:4.0}, mesh)
>>> print(len(b))
4
>>> b = pg.solver.parseArgToBoundaries({mesh.node(0):0.0}, mesh)
>>> print(len(b))
1
>>> def bCall(boundary):
...     u = []
...     for i, n in enumerate(boundary.nodes()):
...         u.append(i)
...     return u
```

(continues on next page)

(continued from previous page)

```
>>> b = pg.solver.parseArgToBoundaries({1:bCall}, mesh)
>>> print(len(b),b[0][1](b[0][0]))
1 [0, 1]
>>> def bCall(boundary, a1, a2):
...     return a1 + a2
>>> b = pg.solver.parseArgToBoundaries({1: [bCall, {'a1':2, 'a2':3}]},␣
↪mesh)
>>> print(len(b), b[0][1][0](b[0][0], **b[0][1][1]))
1 5
>>> b = pg.solver.parseArgToBoundaries({1: [bCall, {'a1':1, 'a2':2}],
...                                     2: [bCall, {'a1':3, 'a2':4}]},␣
↪mesh)
>>> print(len(b), b[0][1][0](b[0][0], **b[0][1][1]))
2 3
>>> b = pg.solver.parseArgToBoundaries({'1,2': [bCall, {'a1':4, 'a2':5}]},
↪ mesh)
>>> print(len(b), b[1][1][0](b[1][0], **b[1][1][1]))
2 9
>>> pg.wait()
```



Examples using `pygimli.solver.parseArgToBoundaries`

- *Basics of Finite Element Analysis* (page 214)

`pygimli.solver.`**`parseDictKey_`**(*key*, *markers*)

pygimli.solver.**parseMapToCellArray**(*attributeMap*, *mesh*, *default=0.0*)

Parse a value map to cell attributes.

A map should consist of pairs of marker and value. A marker is an integer and corresponds to the cell.marker().

Parameters

- **mesh** (GIMLI::Mesh) – For each cell of mesh a value will be returned.
- **attributeMap** (*list | dict*) – List of pairs [marker, value]] || [[marker, value]], or dictionary with marker keys
- **default** (*float [0.0]*) – Fill all unmapped attributes to this default.

Returns

att – Array of length mesh.cellCount()

Return type

array

Examples using pygimli.solver.parseMapToCellArray

- *Gravimetry in 2D - Part I* (page 116)
- *3D Darcy flow* (page 136)
- *Hydrogeophysical modeling* (page 138)

pygimli.solver.**parseMarkersDictKey**(*key*, *markers*)

Parse dictionary key of type str to marker list.

Utility function to parse a dictionary key string into a valid list of markers containing in a given markers list.

Parameters

- **key** (*str | int*) – Supported are - int: single markers - '*': all markers - 'm1': Single marker - 'm1,m2': Comma separated list - ':': Slice wildcard - 'start:stop:step': Slice like syntax
- **markers** (*[int]*) – List of integers, e.g., cell or boundary markers

Returns

mas – List of integers described by key

Return type

[int]

pygimli.solver.**showSparseMatrix**(*mat*, *full=False*)

Show the content of a sparse matrix.

Parameters

- **mat** (GIMLI::SparseMatrix | GIMLI::SparseMapMatrix) – Matrix to be shown.
- **full** (*bool [False]*) – Show as dense matrix.

pygimli.solver.**solve**(*mesh*, ***kwargs*)

Solve partial differential equation.

This is a syntactic sugar convenience function for solving partial differential equation on a given mesh. Using the best guess method for the given parameter. Currently only Finite Element calculation using *pygimli.solver.solveFiniteElements* (page 450)

TODO *pygimli.solver.solveFiniteVolume* (page 452)

pygimli.solver.**solveFiniteElements**(*mesh*, *a=1.0*, *b=None*, *f=0.0*, *bc=None*, *times=None*, *c=1.0*, *userData={}*, *verbose=False*, ***kwargs*)

Solve partial differential equation with Finite Elements.

This is a syntactic sugar convenience function for using the Finite Element functionality of the library core to solve partial differential equation (PDE) that match the following form:

$$
\begin{aligned}
c\frac{\partial u}{\partial t} &= \nabla\cdot(a\nabla u) + bu + f(\mathbf{r},t) \;\mid\; \Omega_{\text{Mesh}} \\
u &= h \;\mid\; \Gamma_{\text{Dirichlet}} \\
\frac{\partial u}{\partial \mathbf{n}} &= g \;\mid\; \Gamma_{\text{Neumann}} \\
\alpha u + \beta\frac{\partial u}{\partial \mathbf{n}} &= \gamma \;\mid\; \Gamma_{\text{Robin}} \\
\frac{\partial u}{\partial \mathbf{n}} &= \alpha(u_0 - u) \;\mid\; \Gamma_{\text{Robin}}
\end{aligned}
$$

for the scalar $u(\mathbf{r},t)$ or vector $(u)(\mathbf{r},t)$ solution at each node of a given mesh. The Domain Ω and the Boundary Γ are defined through the mesh with appropriate boundary marker.

Note, to ensure vector solution either set vector forces or at least on vector component boundary condition.

Parameters

- **mesh** (GIMLI::Mesh) – Mesh represents spatial discretization of the calculation domain
- **a** (*value | array | callable(cell, userData)*) – Cell values of type float or complex can be scalar, anisotropy matrix or elastic tensor.
- **b** (*value | array | callable(cell, userData) [None]*) – Cell values. None means the term is unused.
- **c** (*value | array | callable(cell, userData) [None]*) – Scale the unsteady term, only for times is not None.
- **f** (*value | array(cells) | array(nodes) | callable(args, kwargs)*) – force values, for vector fields use (n x dim) values.
- **bc** (*dict()*) – Dictionary of boundary conditions. Current supported boundary conditions by dictionary keys: ‘Dirichlet’, ‘Neumann’, ‘Robin’, ‘Node’.

  The dictionary can contain multiple “key: Arg” Arg will be parsed by `pygimli.solver.solver.parseArgPairToBoundaryArray`

If the dictionary key is 'Node' then fixed values for single node indices can by be given. e.g., bc={'Node': [nodeID, value]}. Note this is only a shortcut for bc={'Dirichlet': [mesh.node(nodeID), value]}.

- **times** (*array [None]*) – Solve as time dependent problem for the given times.

Keyword Arguments

****kwargs** –

u0: value | array | callable(pos, userData)
Node values

theta: float [1]

- $theta = 0$ means explicit Euler, maybe stable for

Δt near $h * theta = 0.5$, Crank-Nicolson scheme, maybe instable * $theta = 2/3$, Galerkin scheme * $theta = 1$, implicit Euler

If unsure choose $\theta = 0.5 + \varepsilon$ (probably stable).

dynamic: bool [False]
Boundary conditions for time depending problems will be considered dynamic for each time step.

stats: bool
Give some statistics.

progress: bool
Give some calculation progress.

assembleOnly: bool
Stops after matrix asssemblation. Returns the system matrix A and the rhs vector.

fixPureNeumann: bool [auto]
If set or detected automatic, we add the additional condition: $\int_d omainudv = 0$ making elliptic problems well-posed.

rhs: iterable
Pre assembled rhs. Will preferred on any f settings.

ws: dict
The WorkSpace is a dictionary that will get some temporary data during the calculation. Any keyvalue 'u' in the dictionary is used for the resulting array.

vectorValued: bool (False)
Solution forced to vector valued, in case the auto detection fails

Returns

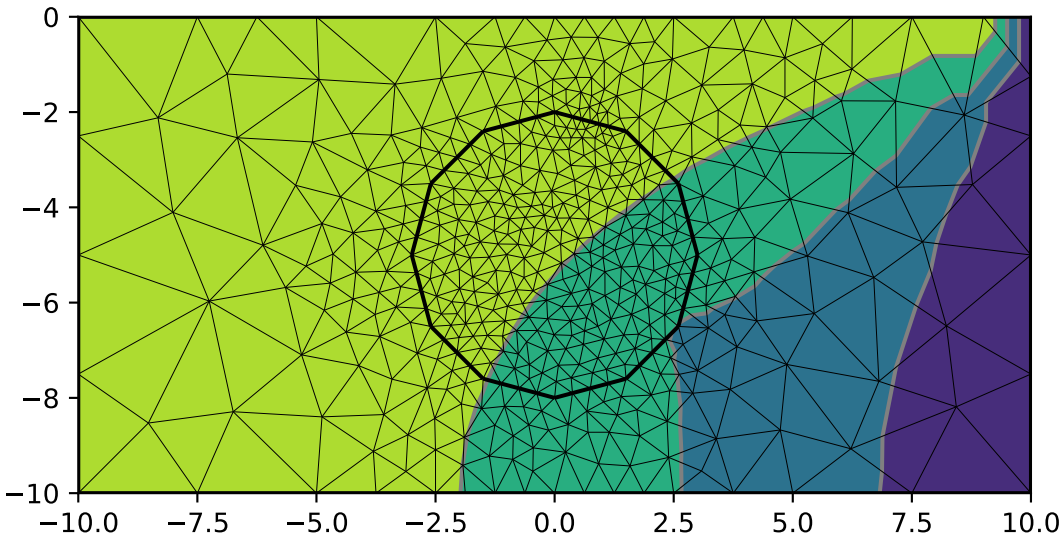
u – Returns the solution u either 1,n array for stationary problems or for m,n array for m time steps

Return type
array

See also:

Modelling (page 214) and *pygimli.solver.solve* (page 449)

```
>>> import pygimli as pg
>>> from pygimli.meshtools import polytools as plc
>>> from pygimli.viewer.mpl import drawField, drawMesh
>>> import matplotlib.pyplot as plt
>>> world = plc.createWorld(start=[-10, 0], end=[10, -10],
...                         marker=1, worldMarker=False)
>>> c1 = plc.createCircle(pos=[0.0, -5.0], radius=3.0, area=.1, marker=2)
>>> mesh = pg.meshtools.createMesh([world, c1], quality=34.3)
>>> u = pg.solver.solveFiniteElements(mesh, a={1: 100.0, 2: 1.0},
...                                   bc={'Dirichlet':{4: 1.0, 2: 0.0}})
>>> fig, ax = plt.subplots()
>>> pc = drawField(ax, mesh, u)
>>> drawMesh(ax, mesh)
>>> plt.show()
```



Examples using `pygimli.solver.solveFiniteElements`

- *3D Darcy flow* (page 136)
- *Hydrogeophysical modeling* (page 138)
- *Heat equation in 2D* (page 223)

`pygimli.solver.`**`solveFiniteVolume`**(*mesh*, *a=1.0*, *b=0.0*, *f=0.0*, *fn=0.0*, *vel=None*, *u0=0.0*, *times=None*, *uL=None*, *relax=1.0*, *ws=None*, *scheme='CDS'*, ***kwargs*)

Solve partial differential equation with Finite Volumes.

This function is a syntactic sugar proxy for using the Finite Volume functionality of the library core to solve elliptic and parabolic partial differential of the following type:

$$\frac{\partial u}{\partial t} + \mathbf{v}\cdot\nabla u = \nabla\cdot(a\nabla u) + bu + f(\mathbf{r},t)$$
$$u(\mathbf{r},t) = u_B \quad \mathbf{r}\in\Gamma_{\text{Dirichlet}}$$
$$\frac{\partial u(\mathbf{r},t)}{\partial \mathbf{n}} = u_{\partial B} \quad \mathbf{r}\in\Gamma_{\text{Neumann}}$$
$$u(\mathbf{r},t=0) = u_0 \quad \text{with} \quad \mathbf{r}\in\Omega$$

The Domain Ω and the Boundary Γ are defined through the given mesh with appropriate boundary marker.

The solution $u(\mathbf{r},t)$ is given for each cell in the mesh.

Parameters

- **mesh** (GIMLI::Mesh) – Mesh represents spatial discretization of the calculation domain
- **a** (*value | array | callable(cell, userData)*) – Stiffness weighting per cell values.
- **b** (*value | array | callable(cell, userData)*) – Scale for mass values b
- **f** (*iterable(cell)*) – Load vector
- **fn** (*iterable(cell)*) – TODO What is fn
- **vel** (*ndarray (N,dim) | RMatrix(N,dim)*) – Velocity field $\mathbf{v}(\mathbf{r},t=\text{const}) = \{[v_i]_j,\}$ with $i = [1\ldots3]$ for the mesh dimension and $j = [0\ldots N-1]$ with N either the amount of cells, nodes, or boundaries. Velocities per boundary are preferred and will be interpolated on demand.
- **u0** (*value | array | callable(cell, userData)*) – Starting field
- **times** (*iterable*) – Time steps to calculate for.
- **Workspace** (*ws*) – This can be an empty class that will used as an Workspace to store and cache data.

  If ws is given: The system matrix is taken from ws or calculated once and stored in ws for further usage.

  The system matrix is cached in this Workspace as ws.S The LinearSolver with the factorized matrix is cached in this Workspace as ws.solver The rhs vector is only stored in this Workspace as ws.rhs
- **scheme** (*str [CDS]*) – Finite volume scheme: `pygimli.solver.diffusionConvectionKernel`
- ****kwargs** –
  - bc : Boundary Conditions dictionary, see pg.solver
  - uB : Dirichlet boundary conditions DEPRECATED
  - duB : Neumann boundary conditions DEPRECATED

Returns

u – Solution field for all time steps.

Return type

ndarray(nTimes, nCells)

Examples using `pygimli.solver.solveFiniteVolume`

- *Hydrogeophysical modeling* (page 138)

pygimli.solver.**triDiagToeplitz**(*dom*, *a*, *l*, *r*, *start=0*, *end=-1*)

Create tri-diagonal Toeplitz matrix.

8.5.3 Classes

class pygimli.solver.**LinSolver**(*mat=None*, *solver=None*, *verbose=False*, ***kwargs*)

Bases: object

Proxy class for the solution of linear systems of equations.

__init__(*mat=None*, *solver=None*, *verbose=False*, ***kwargs*)

Init the solver proxy class with a matrix and start factorization.

Parameters

solver (*str [None]*) – Name for the used solver (pg (umfpack or cholmod), scipy). If solver is none decide from matrix type.

factorize(*mat*)

factorizePG(*mat*)

factorizeSciPy(*mat*)

isFactorized()

solve(*b*)

class pygimli.solver.**RungeKutta**(*solver*, *verbose=False*)

Bases: object

TODO DOCUMENT ME

__init__(*solver*, *verbose=False*)

TODO DOCUMENT_ME

rk4a = [0.0, -0.41789047449985195, -1.192151694642677, -1.6977846924715279, -1.5141834442571558]

rk4b = [0.14965902199922912, 0.37921031299962726, 0.8229550293869817, 0.6994504559491221, 0.15305724796815198]

rk4c = [0.0, 0.14965902199922912, 0.37040095736420475, 0.6222557631344432, 0.9582821306746903]

run(*u0*, *dt*, *tMax=1*)

TODO DOCUMENT_ME

start(*u0*, *dt*, *tMax=1*)

TODO DOCUMENT_ME

step()

TODO DOCUMENT ME

class pygimli.solver.**WorkSpace**

Bases: object

8.6 pygimli.testing

Testing utilities

In Python you can call *pygimli.test()* to run all docstring examples.

8.6.1 Writing tests for pygimli

Please check: https://docs.pytest.org/en/latest/

8.6.2 Overview

Functions

join (page 455)(a, *p)	Join two or more pathname components, inserting '/' as needed.
realpath (page 455)(filename)	Return the canonical path of the specified filename, eliminating any symbolic links encountered in the path.
test (page 455)([target, show, onlydoctests, coverage, ...])	Run docstring examples and additional tests.

8.6.3 Functions

pygimli.testing.**join**(*a*, **p*)

Join two or more pathname components, inserting '/' as needed. If any component is an absolute path, all previous path components will be discarded. An empty last part will result in a path that ends with a separator.

pygimli.testing.**realpath**(*filename*)

Return the canonical path of the specified filename, eliminating any symbolic links encountered in the path.

pygimli.testing.**test**(*target=None*, *show=False*, *onlydoctests=False*, *coverage=False*, *htmlreport=False*, *abort=False*, *verbose=True*)

Run docstring examples and additional tests.

Examples

```
>>> import pygimli as pg
>>> # You can test everything with pg.test() or test a single␣
↪function:
>>> pg.test("pg.utils.boxprint", verbose=False)
>>> # The target argument can also be the function directly
>>> from pygimli.utils import boxprint
>>> pg.test(boxprint, verbose=False)
```

Parameters

- **target** (*function or string, optional*) – Function or method to test. By default everything is tested.
- **show** (*boolean, optional*) – Show matplotlib windows during test run. They will be closed automatically.
- **onlydoctests** (*boolean, optional*) – Run test files in ../tests as well.
- **coverage** (*boolean, optional*) – Create a coverage report. Requires the pytest-cov plugin.
- **htmlreport** (*str, optional*) – Filename for HTML report such as www.pygimli.org/build_tests.html. Requires pytest-html plugin.
- **abort** (*boolean, optional*) – Return correct exit code, e.g. abort documentation build when a test fails.

8.7 pygimli.utils

Useful utility functions.

8.7.1 Overview

Functions

GKtoUTM (page 459)(ea[, no, zone, gk, gkzone])	Transform any Gauss-Krueger to UTM autodetect GK zone from offset.
KramersKronig (page 459)(f, re, im[, usezero])	Return real/imaginary parts retrieved by Kramers-Kronig relations.
boxprint (page 459)(s[, width, sym])	Print string centered in a box.
cMap (page 459)(name)	Return default colormap for physical quantity name.
cache (page 459)(funct)	Cache decorator.
chi2 (page 459)(a, b, err[, trans])	Return chi square value.
cmap (page 459)(name)	Return default colormap for physical quantity name.
computeInverseRootMatrix (page 460)(CM[, thrsh, verbose])	Compute inverse square root (C^{-0.5} of matrix.
convertCRSIndex2Map (page 460)(rowIdx, colPtr)	Converts CRS indices to uncompressed indices (row, col).
covarianceMatrix (page 460)(mesh[, nodes])	Geostatistical covariance matrix (cell or node) for given mesh.
createDateTimeString (page 460)([now])	Return datetime as string (e.g.
createFolders (page 460)(*args, **kwargs)	
createPath (page 460)(pathList)	Create the path structure specified by list.

continues on next page

Table 4 – continued from previous page

createResultFolder (page 460)(*args, **kwargs)	
createResultPath (page 460)(subfolder[, now])	Create a result Folder.
createfolders (page 460)(*args, **kwargs)	
cumDist (page 460)(p)	The progressive, i.e., cumulative length for a path p.
cut (page 461)(v[, n])	Cuts the array v into n parts
diff (page 461)(v)	Calculate approximate derivative.
dist (page 462)(p[, c])	Calculate the distance for each position in p relative to pos c(x,y,z).
filterIndex (page 462)(seq, idx)	TODO DOCUMENTME.
filterLinesByCommentStr (page 463)(lines[, comment_str])	Filter all lines from a file.readlines output which begins with one of the symbols in the comment_str.
findNearest (page 463)(x, y, xp, yp[, radius])	TODO DOCUMENTME.
findUTMZone (page 463)(lon, lat)	Find utm zone for lon and lat values.
generateGeostatisticalModel (page 463)(mesh[, nodes, seed])	Generate geostatistical model (cell or node) for given mesh.
getIndex (page 464)(seq, f)	TODO DOCUMENTME.
getProjection (page 464)(name[, ref])	Syntactic sugar to get some default Projections.
getSavePath (page 464)([folder, subfolder, now])	TODO.
getUTMProjection (page 464)(zone[, ellps])	Create and return the current coordinate projection.
gmat2numpy (page 464)(mat)	Convert pygimli matrix into numpy.array.
grange (page 464)(start, end[, dx, n, log])	Create array with possible increasing spacing.
hankelFC (page 465)(order)	Filter coefficients for Hankel transformation.
interpExtrap (page 465)(x, xp, yp)	numpy.interp interpolation function extended by linear extrapolation.
interperc (page 465)(a[, trimval, islog, bins])	Return symmetric interpercentiles for alpha-trim outliers.
inthist (page 465)(a, vals[, bins, islog])	Return point of integral (cumulative) histogram.
isComplex (page 465)(vals)	Check numpy or pg.Vector if have complex data type
iterateBounds (page 465)(inv[, dchi2, maxiter, change])	Find parameter bounds by iterating model parameter.
logDropTol (page 466)(p[, dropTol])	Create logarithmic scaled copy of p.
modCovar (page 466)(inv)	Formal model covariance matrix (MCM) from inversion.
nanrms (page 466)(v[, axis])	Compute the root mean square excluding nan values.
niceLogspace (page 466)(vMin, vMax[, nDec])	Create nice logarithmic space from the next decade lower to vMin to decade larger then vMax.
noCache (page 467)(c)	

continues on next page

Table 4 – continued from previous page

num2str (page 467)(a[, fmtstr])	List of strings (deprecated, for backward-compatibility).
numpy2gmat (page 467)(nmat)	Convert numpy.array into pygimli RMatrix.
prettify (page 467)(value[, roundValue])	Return prettified string for value .
prettyFloat (page 467)(value[, roundValue])	Return prettified string for a float value.
prettyTime (page 467)(t)	Return prettified time in seconds as string.
rand (page 468)(n[, minVal, maxVal, seed])	Create RVector of length n with normally distributed random numbers.
randn (page 468)(n[, seed])	Create n normally distributed random numbers with optional seed.
readGPX (page 469)(fileName)	Extract GPS Waypoint from GPS Exchange Format (GPX).
rms (page 469)(v[, axis])	Compute the root mean square.
rmsWithErr (page 469)(a, b, err[, errtol])	Compute (abs-)root-mean-square of values with error above threshold.
rmswitherr (page 469)(a, b, err[, errtol])	Compute (abs-)root-mean-square of values with error above threshold.
rndig (page 469)(a[, ndig])	Round float using a number of counting digits.
rrms (page 469)(a, b[, axis])	Compute the relative (regarding a) root mean square.
saveResult (page 469)(fname, data[, rrms, chi2, mode])	Save rms/chi2 results into filename.
sparseMatrix2Array (page 469)(matrix[, indices, getInCRS])	Extract indices and value from sparse matrix (SparseMap or CRS)
sparseMatrix2Dense (page 469)(matrix)	Convert sparse matrix to dense ndarray
sparseMatrix2coo (page 470)(A[, rowOffset, colOffset])	Convert SparseMatrix to scipy.coo_matrix.
sparseMatrix2csr (page 470)(A)	Convert SparseMatrix to scipy.csr_matrix.
squeezeComplex (page 470)(z[, polar, conj])	Squeeze complex valued array into [real, imag] or [amp, phase(rad)]
strHash (page 470)(string)	
streamline (page 470)(mesh, field, startCoord, dLengthSteps)	Create a streamline from start coordinate and following a vector field in up and down direction.
streamlineDir (page 470)(mesh, field, startCoord, ...)	down = -1, up = 1, both = 0
toCOO (page 470)(A)	
toCSR (page 470)(A)	
toComplex (page 470)(amp[, phi])	Convert real values into complex (z = a + ib) valued array.
toPolar (page 471)(z)	Convert complex (z = a + ib) values array into amplitude and phase in radiant
toRealMatrix (page 471)(C[, conj])	Convert complex valued matrix into a real valued Blockmatrix
toSparseMapMatrix (page 471)(A)	Convert any matrix type to pg.SparseMatrix and return copy of it.

continues on next page

Table 4 – continued from previous page

toSparseMatrix (page 472)(A)	Convert any matrix type to pg.SparseMatrix and return copy of it.
trimDocString (page 472)(docstring)	Return properly formatted docstring.
unicodeToAscii (page 472)(text)	TODO DOCUMENTME.
unique (page 472)(a)	Return list of unique elements ever seen with preserving order.
uniqueAndSum (page 472)(indices, to_sum[, ...])	Sum double values found by indices in a various number of arrays.
unique_everseen (page 473)(iterable[, key])	Return iterator of unique elements ever seen with preserving order.
unique_rows (page 474)(array)	Return unique rows in a 2D array.
unit (page 474)(name[, unit])	Return the name of a physical quantity with its unit.

Classes

ProgressBar (page 474)(its[, width, sign])	Animated text-based progressbar.

8.7.2 Functions

pygimli.utils.**GKtoUTM**(*ea*, *no=None*, *zone=32*, *gk=None*, *gkzone=None*)

Transform any Gauss-Krueger to UTM autodetect GK zone from offset.

pygimli.utils.**KramersKronig**(*f*, *re*, *im*, *usezero=False*)

Return real/imaginary parts retrieved by Kramers-Kronig relations.

formulas including singularity removal according to Boukamp (1993)

pygimli.utils.**boxprint**(*s*, *width=80*, *sym='#'*)

Print string centered in a box.

Examples

```
>>> from pygimli.utils import boxprint
>>> boxprint("This is centered in a box.", width=40, sym='+')
++++++++++++++++++++++++++++++++++++++++
+      This is centered in a box.      +
++++++++++++++++++++++++++++++++++++++++
```

pygimli.utils.**cMap**(*name*)

Return default colormap for physical quantity name.

pygimli.utils.**cache**(*funct*)

Cache decorator.

pygimli.utils.**chi2**(*a*, *b*, *err*, *trans=None*)

Return chi square value.

pygimli.utils.**cmap**(*name*)

Return default colormap for physical quantity name.

pygimli.utils.**computeInverseRootMatrix**(*CM*, *thrsh=0.001*, *verbose=False*)

Compute inverse square root (C^{-0.5} of matrix.

pygimli.utils.**convertCRSIndex2Map**(*rowIdx*, *colPtr*)

Converts CRS indices to uncompressed indices (row, col).

pygimli.utils.**covarianceMatrix**(*mesh*, *nodes=False*, ***kwargs*)

Geostatistical covariance matrix (cell or node) for given mesh.

Parameters

- **mesh** (gimliapi:*GIMLI::Mesh*) – Mesh
- **nodes** (*bool [False]*) – use node positions, otherwise (default) cell centers are used
- ****kwargs** –

  I
  [float or list of floats] correlation lengths (range) in individual directions

  dip
  [float] dip angle (in degree) of major axis (I[0])

  strike
  [float] strike angle (for 3D)

Returns

Cm – covariance matrix

Return type

np.array (square matrix of size cellCount/nodeCount)

Examples using `pygimli.utils.covarianceMatrix`

- *Geostatistical regularization* (page 254)

pygimli.utils.**createDateTimeString**(*now=None*)

Return datetime as string (e.g. for saving results).

pygimli.utils.**createFolders**(**args*, ***kwargs*)

pygimli.utils.**createPath**(*pathList*)

Create the path structure specified by list.

Parameters

pathList (*str | list(str)*) – Create Path with option subpaths

pygimli.utils.**createResultFolder**(**args*, ***kwargs*)

pygimli.utils.**createResultPath**(*subfolder*, *now=None*)

Create a result Folder.

pygimli.utils.**createfolders**(**args*, ***kwargs*)

pygimli.utils.**cumDist**(*p*)

The progressive, i.e., cumulative length for a path p.

d = [0.0, d[0]+ | p[1]-p[0] |, d[1] + | p[2]-p[1] | + . . .]

Parameters

p (*ndarray(N,2) | ndarray(N,3) | pg.core.R3Vector*) – Position array

Returns

d – Distance array

Return type

ndarray(N)

Examples

```
>>> import pygimli as pg
>>> from pygimli.utils import cumDist
>>> import numpy as np
>>> p = pg.core.R3Vector(4)
>>> p[0] = [0.0, 0.0]
>>> p[1] = [0.0, 1.0]
>>> p[2] = [0.0, 1.0]
>>> p[3] = [0.0, 0.0]
>>> print(cumDist(p))
[0. 1. 1. 2.]
```

pygimli.utils.**cut**(*v*, *n=2*)

Cuts the array v into n parts

pygimli.utils.**diff**(*v*)

Calculate approximate derivative.

Calculate approximate derivative from v as d = [v_1-v_0, v2-v_1, . . .]

Parameters

v (*array(N) | pg.core.R3Vector(N)*) – Array of double values or positions

Returns

d – derivative array

Return type

[type(v)](N-1) |

Examples

```
>>> import pygimli as pg
>>> from pygimli.utils import diff
>>> p = pg.core.R3Vector(4)
>>> p[0] = [0.0, 0.0]
>>> p[1] = [0.0, 1.0]
>>> print(diff(p)[0])
RVector3: (0.0, 1.0, 0.0)
>>> print(diff(p)[1])
RVector3: (0.0, -1.0, 0.0)
>>> print(diff(p)[2])
RVector3: (0.0, 0.0, 0.0)
>>> p = pg.Vector(3)
>>> p[0] = 0.0
>>> p[1] = 1.0
>>> p[2] = 2.0
>>> print(diff(p))
2 [1.0, 1.0]
```

pygimli.utils.**dist**(*p*, *c=None*)

Calculate the distance for each position in p relative to pos c(x,y,z).

Parameters

- **p** (*ndarray(N,2) | ndarray(N,3) | pg.core.R3Vector*) – Position array
- **c** (*[x,y,z] [None]*) – relative origin. default = [0, 0, 0]

Returns

d – Distance array

Return type

ndarray(N)

Examples

```
>>> import pygimli as pg
>>> from pygimli.utils import dist
>>> import numpy as np
>>> p = pg.core.R3Vector(4)
>>> p[0] = [0.0, 0.0]
>>> p[1] = [0.0, 1.0]
>>> print(dist(p))
[0. 1. 0. 0.]
>>> x = pg.Vector(4, 0)
>>> y = pg.Vector(4, 1)
>>> print(dist(np.array([x, y]).T))
[1. 1. 1. 1.]
```

pygimli.utils.**filterIndex**(*seq*, *idx*)

TODO DOCUMENTME.

pygimli.utils.**filterLinesByCommentStr**(*lines*, *comment_str='#'*)

Filter all lines from a file.readlines output which begins with one of the symbols in the comment_str.

pygimli.utils.**findNearest**(*x*, *y*, *xp*, *yp*, *radius=-1*)

TODO DOCUMENTME.

pygimli.utils.**findUTMZone**(*lon*, *lat*)

Find utm zone for lon and lat values.

lon -180 – -174 -> 1 . . . 174 – 180 -> 60 lat < 0 hemisphere = S, > 0 hemisphere = N

Parameters

- **lon** (*float*) –
- **lat** (*float*) –

Returns

zone + hemisphere

Return type

str

pygimli.utils.**generateGeostatisticalModel**(*mesh*, *nodes=False*, *seed=None*, ***kwargs*)

Generate geostatistical model (cell or node) for given mesh.

Parameters

- **mesh** (gimliapi:*GIMLI::Mesh*) – Mesh
- **nodes** (*bool [False]*) – use node positions, otherwise (default) cell centers are used
- **seed** (*int, array_like[ints], SeedSequence, BitGenerator, Generator}, optional*) – A seed to initialize the BitGenerator. If None, then fresh, unpredictable entropy will be used. The *seed* variable is passed to `numpy.random.default_rng()`
- ****kwargs** –

  I

  [float or list of floats] correlation lengths (range) in individual directions

  dip

  [float] dip angle of major axis (I[0])

  strike

  [float] strike angle (for 3D)

Returns

res

Return type

np.array of size cellCount or nodeCount (nodes=True)

Examples using `pygimli.utils.generateGeostatisticalModel`

- *Geostatistical regularization* (page 254)

pygimli.utils.**getIndex**(*seq*, *f*)

TODO DOCUMENTME.

pygimli.utils.**getProjection**(*name*, *ref=None*, ***kwargs*)

Syntactic sugar to get some default Projections.

pygimli.utils.**getSavePath**(*folder=None*, *subfolder=''*, *now=None*)

TODO.

pygimli.utils.**getUTMProjection**(*zone*, *ellps='WGS84'*)

Create and return the current coordinate projection.

This is a proxy for pyproj.

Parameters

- **utmZone** (*str*) – Zone for for UTM
- **ellipsoid** (*str*) – ellipsoid based on [‘wgs84’]

Return type
Pyproj Projection

pygimli.utils.**gmat2numpy**(*mat*)

Convert pygimli matrix into numpy.array.

TODO implement correct rval

pygimli.utils.**grange**(*start*, *end*, *dx=0*, *n=0*, *log=False*)

Create array with possible increasing spacing.

Create either array from start step-wise filled with dx until end reached [start, end] (like np.array with defined end). Fill the array from start to end with n steps. [start, end] (like np.linespace) Fill the array from start to end with n steps but logarithmic increasing, dx will be ignored.

Parameters

- **start** (*float*) – First value of the resulting array
- **end** (*float*) – Last value of the resulting array
- **dx** (*float*) – Linear step length, n will be ignored
- **n** (*int*) – Amount of steps
- **log** (*bool*) – Logarithmic increasing range of length = n from start to end. dx will be ignored.

Examples

```
>>> from pygimli.utils import grange
>>> v1 = grange(start=0, end=10, dx=3)
>>> v2 = grange(start=0, end=10, n=3)
>>> print(v1)
4 [0.0, 3.0, 6.0, 9.0]
>>> print(v2)
3 [0.0, 5.0, 10.0]
```

Returns
ret – Return resulting array

Return type
GIMLI::RVector

Examples using `pygimli.utils.grange`

- *2D ERT modeling and inversion* (page 60)
- *2D FEM modelling on two-layer example* (page 73)
- *Hydrogeophysical modeling* (page 138)

`pygimli.utils.`**`hankelFC`**(*order*)

Filter coefficients for Hankel transformation.

10 data points per decade.

DOCUMENTME .. Author RUB?

Parameters
order (*int*) – order=1: NY=+0.5 (SIN) order=2: NY=-0.5 (COS) order=3: NY=0.0 (J0) order=4: NY=1.0 (J1)

Returns

- **fc** (*np.array()*) – Filter coefficients
- **nc0** (*int*) – fc[nc0] refers to zero argument

`pygimli.utils.`**`interpExtrap`**(*x*, *xp*, *yp*)

numpy.interp interpolation function extended by linear extrapolation.

`pygimli.utils.`**`interperc`**(*a*, *trimval=3.0*, *islog=False*, *bins=None*)

Return symmetric interpercentiles for alpha-trim outliers.

E.g. interperc(a, 3) returns range of inner 94% (3 to 97%) which is particularly useful for colorscales).

`pygimli.utils.`**`inthist`**(*a*, *vals*, *bins=None*, *islog=False*)

Return point of integral (cumulative) histogram.

E.g. inthist(a, [25, 50, 75]) provides quartiles and median of an array

`pygimli.utils.`**`isComplex`**(*vals*)

Check numpy or pg.Vector if have complex data type

pygimli.utils.**iterateBounds**(*inv*, *dchi2=0.5*, *maxiter=100*, *change=1.02*)

Find parameter bounds by iterating model parameter.

Find parameter bounds by iterating model parameter until error bound is reached

Parameters

- **inv** – gimli inversion object
- **dchi2** – allowed variation of chi^2 values [0.5]
- **maxiter** – maximum iteration number for parameter iteration [100]
- **change** – changing factor of parameters [1.02, i.e. 2%]

pygimli.utils.**logDropTol**(*p*, *dropTol=0.001*)

Create logarithmic scaled copy of p.

Examples

```
>>> from pygimli.utils import logDropTol
>>> x = logDropTol((-10, -1, 0, 1, 100))
>>> print(x.array())
[-4. -3.  0.  3.  5.]
```

Examples using pygimli.utils.logDropTol

- *Four-point sensitivities* (page 81)

pygimli.utils.**modCovar**(*inv*)

Formal model covariance matrix (MCM) from inversion.

var, MCMs = modCovar(inv)

Parameters

inv (*pygimli inversion object*) –

Returns

- **var** (*variances (inverse square roots of MCM matrix)*)
- **MCMs** (*scaled MCM (such that diagonals are 1.0)*)

```
>>> # import pygimli as pg
>>> # import matplotlib.pyplot as plt
>>> # from matplotlib.cm import bwr
>>> # INV = pg.Inversion(data, f)
>>> # par = INV.run()
>>> # var, MCM = modCovar(INV)
>>> # i = plt.imshow(MCM, interpolation='nearest',
>>> #                 cmap=bwr, vmin=-1, vmax=1)
>>> # plt.colorbar(i)
```

pygimli.utils.**nanrms**(*v*, *axis=None*)

Compute the root mean square excluding nan values.

pygimli.utils.**niceLogspace**(*vMin*, *vMax*, *nDec=10*)

Create nice logarithmic space from the next decade lower to vMin to decade larger then vMax.

Parameters

- **vMin** (*float*) – lower limit need to be > 0
- **vMax** (*float*) – upper limit need to be >= vMin
- **nDec** (*int*) – Amount of logarithmic equidistant steps for one decade

Examples

```
>>> from pygimli.utils import niceLogspace
>>> v1 = niceLogspace(vMin=0.1, vMax=0.1, nDec=1)
>>> print(v1)
[0.1 1. ]
>>> v1 = niceLogspace(vMin=0.09, vMax=0.11, nDec=1)
>>> print(v1)
[0.01 0.1  1.  ]
>>> v1 = niceLogspace(vMin=0.09, vMax=0.11, nDec=10)
>>> print(len(v1))
21
>>> print(v1)
[0.01       0.01258925 0.01584893 0.01995262 0.02511886 0.03162278
 0.03981072 0.05011872 0.06309573 0.07943282 0.1        0.12589254
 0.15848932 0.19952623 0.25118864 0.31622777 0.39810717 0.50118723
 0.63095734 0.79432823 1.        ]
```

pygimli.utils.**noCache**(*c*)

pygimli.utils.**num2str**(*a*, *fmtstr='%g'*)

List of strings (deprecated, for backward-compatibility).

pygimli.utils.**numpy2gmat**(*nmat*)

Convert numpy.array into pygimli RMatrix.

TODO implement correct rval

pygimli.utils.**prettify**(*value*, *roundValue=False*)

Return prettified string for value .. if possible.

pygimli.utils.**prettyFloat**(*value*, *roundValue=None*)

Return prettified string for a float value.

pygimli.utils.**prettyTime**(*t*)

Return prettified time in seconds as string.

No months, no leap year.

Parameters

t (*float*) – Time in seconds, should be > 0

Examples

```
>>> from pygimli.utils import prettyTime
>>> print(prettyTime(1))
1 s
>>> print(prettyTime(3600*24))
1 day
>>> print(prettyTime(2*3600*24))
2 days
>>> print(prettyTime(365*3600*24))
1 year
>>> print(prettyTime(3600))
1 hour
>>> print(prettyTime(2*3600))
2 hours
>>> print(prettyTime(3660))
1h1m
>>> print(prettyTime(1e-3))
1 ms
>>> print(prettyTime(1e-6))
1 µs
>>> print(prettyTime(1e-9))
1 ns
```

pygimli.utils.**rand**(*n*, *minVal=0.0*, *maxVal=1.0*, *seed=None*)

Create RVector of length n with normally distributed random numbers.

pygimli.utils.**randn**(*n*, *seed=None*)

Create n normally distributed random numbers with optional seed.

Parameters

- **n** (*long*) – length of random numbers array.
- **seed** (*int [None]*) – Optional seed for random number generator

Returns

r – Random numbers.

Return type

np.array

Examples

```
>>> import numpy as np
>>> from pygimli.utils import randn
>>> a = randn(5, seed=1337)
>>> b = randn(5)
>>> c = randn(5, seed=1337)
>>> print(np.array_equal(a, b))
False
```

(continues on next page)

(continued from previous page)

```
>>> print(np.array_equal(a, c))
True
```

pygimli.utils.**readGPX**(*fileName*)

Extract GPS Waypoint from GPS Exchange Format (GPX).

Currently only simple waypoint extraction is supported.

<gpx version="1.0" creator="creator">

<metadata> <name>Name</name> </metadata> <wpt lat="51." lon="11."> <name>optional</name> <time>optional</time> <description>optional</description> </wpt>

</gpx>

pygimli.utils.**rms**(*v*, *axis=None*)

Compute the root mean square.

pygimli.utils.**rmsWithErr**(*a*, *b*, *err*, *errtol=1*)

Compute (abs-)root-mean-square of values with error above threshold.

pygimli.utils.**rmswitherr**(*a*, *b*, *err*, *errtol=1*)

Compute (abs-)root-mean-square of values with error above threshold.

pygimli.utils.**rndig**(*a*, *ndig=3*)

Round float using a number of counting digits.

pygimli.utils.**rrms**(*a*, *b*, *axis=None*)

Compute the relative (regarding a) root mean square.

pygimli.utils.**saveResult**(*fname*, *data*, *rrms=None*, *chi2=None*, *mode='w'*)

Save rms/chi2 results into filename.

pygimli.utils.**sparseMatrix2Array**(*matrix*, *indices=True*, *getInCRS=True*)

Extract indices and value from sparse matrix (SparseMap or CRS)

Get python Arrays from SparseMatrix or SparseMapMatrix in either CRS convention (row index, column Start_End, values) or row index, column index, values.

Parameters

- **matrix** (*pg.matrix.SparseMapMatrix or pg.matrix.SparseMatrix*) – Input matrix to be transformed to numpy arrays.
- **indices** (*boolean (True)*) – Decides weather the indices of the matrix will be returned or not.
- **getInCSR** (*boolean (True)*) – If returned, the indices can have the format of a compressed row storage (CSR), the default or uncompressed lists with column and row indices.

Returns

- **vals** (*numpy.ndarray*) – Entries of the matrix.
- **indices** (*list, list*) – Optional. Returns additional array with the indices for reconstructing the matrix in the defined format.

pygimli.utils.**sparseMatrix2Dense**(*matrix*)

Convert sparse matrix to dense ndarray

pygimli.utils.**sparseMatrix2coo**(*A*, *rowOffset=0*, *colOffset=0*)

Convert SparseMatrix to scipy.coo_matrix.

Parameters
A (*pg.matrix.SparseMapMatrix | pg.matrix.SparseMatrix*) – Matrix to convert from.

Returns
mat – Matrix to convert into.

Return type
scipy.coo_matrix

Examples using `pygimli.utils.sparseMatrix2coo`

- *Naive complex-valued electrical inversion* (page 104)

pygimli.utils.**sparseMatrix2csr**(*A*)

Convert SparseMatrix to scipy.csr_matrix.

Compressed Sparse Row matrix, i.e., Compressed Row Storage (CRS)

Parameters
A (*pg.matrix.SparseMapMatrix | pg.matrix.SparseMatrix*) – Matrix to convert from.

Returns
mat – Matrix to convert into.

Return type
scipy.csr_matrix

pygimli.utils.**squeezeComplex**(*z*, *polar=False*, *conj=False*)

Squeeze complex valued array into [real, imag] or [amp, phase(rad)]

Examples using `pygimli.utils.squeezeComplex`

- *Naive complex-valued electrical inversion* (page 104)

pygimli.utils.**strHash**(*string*)

pygimli.utils.**streamline**(*mesh*, *field*, *startCoord*, *dLengthSteps*, *dataMesh=None*, *maxSteps=1000*, *verbose=False*, *coords=(0, 1)*)

Create a streamline from start coordinate and following a vector field in up and down direction.

pygimli.utils.**streamlineDir**(*mesh*, *field*, *startCoord*, *dLengthSteps*, *dataMesh=None*, *maxSteps=150*, *down=True*, *verbose=False*, *coords=(0, 1)*)

down = -1, up = 1, both = 0

pygimli.utils.**toCOO**(*A*)

pygimli.utils.**toCSR**(*A*)

pygimli.utils.**toComplex**(*amp*, *phi=None*)

Convert real values into complex (z = a + ib) valued array.

If no phases phi are given, assuming z = amp[0:N] + i amp[N:2N].

If phi is given in (rad) complex values are generated: z = amp*(cos(phi) + i sin(phi))

Parameters

- **amp** (*iterable (float)*) – Amplitudes or real unsqueezed real valued array.
- **phi** (*iterable (float)*) – Phases in neg radiant

Returns

z – Complex values

Return type

ndarray(dtype=np.complex)

Examples using `pygimli.utils.toComplex`

- *Complex-valued electrical modeling* (page 98)
- *Naive complex-valued electrical inversion* (page 104)

pygimli.utils.**toPolar**(*z*)

Convert complex (z = a + ib) values array into amplitude and phase in radiant

If z is real valued we assume its squeezed.

Parameters

z (*iterable (floats, complex)*) – If z contains of floats and squeezed-Complex is assumed [real, imag]

Returns

amp, phi – Amplitude amp and phase angle phi in radiant.

Return type

ndarray

pygimli.utils.**toRealMatrix**(*C*, *conj=False*)

Convert complex valued matrix into a real valued Blockmatrix

Parameters

- **C** (*CMatrix*) – Complex valued matrix
- **conj** (*bool [False]*) – Fill the matrix as complex conjugated matrix

Returns

R

Return type

pg.matrix.BlockMatrix()

pygimli.utils.**toSparseMapMatrix**(*A*)

Convert any matrix type to pg.SparseMatrix and return copy of it.

Parameters

A (*pg or scipy matrix*) –

Return type
pg.SparseMatrix

pygimli.utils.**toSparseMatrix**(*A*)

Convert any matrix type to pg.SparseMatrix and return copy of it.

No conversion if A is a SparseMatrix already :param A: :type A: pg or scipy matrix

Return type
pg.SparseMatrix

pygimli.utils.**trimDocString**(*docstring*)

Return properly formatted docstring.

From: https://www.python.org/dev/peps/pep-0257/

Examples

```
>>> from pygimli.utils import trimDocString
>>> docstring = '    This is a string with indention and whitespace.    '
>>> trimDocString(docstring).replace('with', 'without')
'This is a string without indention and whitespace.'
```

pygimli.utils.**unicodeToAscii**(*text*)

TODO DOCUMENTME.

pygimli.utils.**unique**(*a*)

Return list of unique elements ever seen with preserving order.

Examples

```
>>> from pygimli.utils import unique
>>> unique((1,1,2,2,3,1))
[1, 2, 3]
```

See also:

unique_everseen (page 473), *unique_rows* (page 474)

pygimli.utils.**uniqueAndSum**(*indices*, *to_sum*, *return_index=False*, *verbose=False*)

Sum double values found by indices in a various number of arrays.

Returns the sorted unique elements of a column_stacked array of indices. Another column_stacked array is returned with values at the unique indices, while values at double indices are properly summed.

Parameters

- **ar** (*array_like*) – Input array. This will be flattened if it is not already 1-D.
- **to_sum** (*array_like*) – Input array to be summed over axis 0. Other existing axes will be broadcasted remain untouched.

- **return_index** (*bool, optional*) – If True, also return the indices of *ar* (along the specified axis, if provided, or in the flattened array) that result in the unique array.

Returns

- **unique** (*ndarray*) – The sorted unique values.
- **summed_array** (*ndarray*) – The summed array, whereas all values for a specific index is the sum over all corresponding nonunique values.
- **unique_indices** (*ndarray, optional*) – The indices of the first occurrences of the unique values in the original array. Only provided if *return_index* is True.

Examples

```
>>> import numpy as np
>>> from pygimli.utils import uniqueAndSum
>>> idx1 = np.array([0, 0, 1, 1, 2, 2])
>>> idx2 = np.array([0, 0, 1, 2, 3, 3])
>>> # indices at positions 0 and 1 and at positions 5 and 6 are␣
↪not unique
>>> to_sort = np.column_stack((idx1, idx2))
>>> # its possible to stack more than two array
>>> # you need for example 3 array to find unique node␣
↪positions in a mesh
>>> values = np.arange(0.1, 0.7, 0.1)
>>> print(values)
[0.1 0.2 0.3 0.4 0.5 0.6]
>>> # some values to be summed together (for example attributes␣
↪of nodes)
>>> unique_idx, summed_vals = uniqueAndSum(to_sort, values)
>>> print(unique_idx)
[[0 0]
 [1 1]
 [1 2]
 [2 3]]
>>> print(summed_vals)
[0.3 0.3 0.4 1.1]
```

pygimli.utils.**unique_everseen**(*iterable*, *key=None*)

Return iterator of unique elements ever seen with preserving order.

Return iterator of unique elements ever seen with preserving order.

From: https://docs.python.org/3/library/itertools.html#itertools-recipes

Examples

```
>>> from pygimli.utils import unique_everseen
>>> s1 = 'AAAABBBCCDAABBB'
>>> s2 = 'ABBCcAD'
>>> list(unique_everseen(s1))
['A', 'B', 'C', 'D']
>>> list(unique_everseen(s2, key=str.lower))
['A', 'B', 'C', 'D']
```

See also:

unique (page 472), *unique_rows* (page 474)

pygimli.utils.**unique_rows**(*array*)

Return unique rows in a 2D array.

Examples

```
>>> from pygimli.utils import unique_rows
>>> import numpy as np
>>> A = np.array([[1,2,3],[3,2,1],[1,2,3]])
>>> unique_rows(A)
array([[1, 2, 3],
       [3, 2, 1]])
```

pygimli.utils.**unit**(*name*, *unit='auto'*)

Return the name of a physical quantity with its unit.

8.7.3 Classes

class pygimli.utils.**ProgressBar**(*its*, *width=80*, *sign=':'*, ***kwargs*)

Bases: object

Animated text-based progressbar.

Animated text-based progressbar for intensive loops. Should work in the console. In IPython Notebooks a 'tqdm' progressbar instance is created and can be configured with appropriate keyword arguments.

__init__(*its*, *width=80*, *sign=':'*, ***kwargs*)

Create animated text-based progressbar.

- optional: 'estimated time' instead of 'x of y complete'

its
[int] Number of iterations of the process.

width
[int] Width of the ProgressBar, default is 80.

sign
[str] Sign used to fill the bar.

Forwarded to create the tqdm progressbar instance. See https://tqdm.github.io/docs/tqdm/

```
>>> from pygimli.utils import ProgressBar
>>> pBar = ProgressBar(its=20, width=40, sign='+')
>>> pBar.update(5)
```

[++++++++++ 30%] 6 of 20 complete

update(*iteration*, *msg=''*)

Update ProgressBar by iteration number starting at 0 with optional message.

8.8 pygimli.viewer

Note: For 2D and 3D visualizations, we rely on matplotlib (www.matplotlib.org) and pyvista (www.pyvista.org), respectively.

Interface for 2D and 3D visualizations.

Module overview

mpl (page 475)	Drawing functions using Matplotlib.
pv (page 505)	Pyvista based drawing functions used by pygimli.viewer.

8.8.1 pygimli.viewer.mpl

Drawing functions using Matplotlib.

8.8.1.1 Overview

Functions

addCoverageAlpha (page 478)(patches, coverage[, ...])	Add alpha values to the colors of a polygon collection.
adjustWorldAxes (page 479)(ax)	Set some common default properties for an axe.
autolevel (page 479)(z, nLevs[, logScale, zMin, zMax])	Create nLevs bins for the data array z based on matplotlib ticker.
cacheFileName (page 479)(fullname, vendor)	Createfilename and path to cache download data.
cmapFromName (page 479)([cmapname, ncols, bad])	Get a colormap either from name or from keyworld list.
createAnimation (page 479)(fig, animate, nFrames, dpi, out)	Create animation for the content of a given matplotlib figure.

continues on next page

Table 5 – continued from previous page

createColorBar (page 479)(gci[, orientation, size, pad])	Create a Colorbar.
createColorBarOnly (page 480)([cMin, cMax, logScale, ...])	Create figure with a colorbar.
createMeshPatches (page 480)(ax, mesh[, rasterized, ...])	Utility function to create 2d mesh patches within a given ax.
createTriangles (page 480)(mesh)	Generate triangle objects for later drawing.
createTwinX (page 481)(ax)	Utility function to create (or return existing) twin x axes for ax.
createTwinY (page 481)(ax)	Utility function to create (or return existing) twin x axes for ax.
create_legend (page 481)(ax, cmap, ids, classes)	Create a list of patch objects that can be used for borehole legends.
deg2MapTile (page 481)(lon_deg, lat_deg, zoom)	TODO Documentme.
draw1DColumn (page 481)(ax, x, val, thk[, width, ...])	Draw a 1D column (e.g., from a 1D inversion) on a given ax.
draw1dmodel (page 481)(x[, thk, xlab, zlab, islog, z0])	DEPRECATED.
draw1dmodelErr (page 481)(x, xL[, xU, thk, xcol, ycol])	TODO.
draw1dmodelLU (page 481)(x, xL, xU[, thk])	Draw 1d model with lower and upper bounds.
drawBlockMatrix (page 481)(ax, mat, **kwargs)	Draw a view of a matrix into the axes.
drawBoundaryMarkers (page 483)(ax, mesh[, ...])	Draw boundary markers for mesh.boundaries with marker != 0
drawDataMatrix (page 484)(ax, mat[, xmap, ymap, cMin, ...])	Draw previously generated (generateVecMatrix) matrix.
drawField (page 484)(ax, mesh[, data, levels, nLevs, ...])	Draw mesh with scalar field data.
drawMesh (page 485)(ax, mesh[, fitView])	Draw a 2d mesh into a given ax.
drawMeshBoundaries (page 486)(ax, mesh[, hideMesh, ...])	Draw mesh on ax with boundary conditions colorized.
drawModel (page 487)(ax, mesh[, data, tri, rasterized, ...])	Draw a 2d mesh and color the cell by the data.
drawModel1D (page 489)(ax[, thickness, values, model, ...])	Draw 1d block model into axis ax.
drawPLC (page 490)(ax, mesh[, fillRegion, ...])	Draw 2D PLC into given axes.
drawParameterConstraints (page 491)(ax, mesh, cMat[, ...])	Draw inter parameter constraints between cells.
drawSelectedMeshBoundaries (page 492)(ax, boundaries[, ...])	Draw mesh boundaries into a given axes.
drawSelectedMeshBoundariesShadow (page 492)(ax, boundaries)	Draw mesh boundaries as shadows into a given axes.
drawSensorAsMarker (page 493)(ax, data)	Draw Sensor marker, these marker are pickable.
drawSensors (page 493)(ax, sensors[, diam, coords])	Draw sensor positions as black dots with a given diameter.

continues on next page

Table 5 – continued from previous page

drawSparseMatrix (page 493)(ax, mat, **kwargs)	Draw a view of a matrix into the axes.
drawStreamLines (page 494)(ax, mesh, u[, nx, ny])	Draw streamlines for the gradients of field values u on a mesh.
drawStreams (page 495)(ax, mesh, data[, startStream, ...])	Draw streamlines based on an unstructured mesh.
drawValMapPatches (page 496)(ax, vals[, xVec, yVec, dx, dy])	Show values as patches over x and y vector.
drawVecMatrix (page 497)(ax, xvec, yvec, vals[, full])	
findAndMaskBestClim (page 497)(dataIn[, cMin, cMax, ...])	TODO Documentme.
generateMatrix (page 497)(xvec, yvec, vals, **kwargs)	Generate a data matrix from x/y and value vectors.
getMapTile (page 497)(xtile, ytile, zoom[, vendor, verbose])	Get a map tile from public mapping server.
hold (page 497)([val])	TODO WRITEME.
insertUnitAtNextLastTick (page 497)(ax, unit[, xlabel, ...])	Replace the last-but-one tick label by unit symbol.
isInteractive (page 497)()	Returns False if a non-interactive backend is used, e.g.
mapTile2deg (page 498)(xtile, ytile, zoom)	Calculate the NW-corner of the square.
noShow (page 498)([on])	Toggle quiet mode to avoid popping figures.
patchMatrix (page 498)(mat[, xmap, ymap, ax, cMin, ...])	Plot previously generated (generateVecMatrix) matrix.
patchValMap (page 498)(vals[, xvec, yvec, ax, cMin, ...])	Plot previously generated (generateVecMatrix) y map (category).
plotDataContainerAsMatrix (page 499)(*args, **kwargs)	DEPRECATED naming scheme
plotLines (page 499)(ax, line_filename[, linewidth, step])	Read lines from file and plot over model.
plotMatrix (page 499)(mat, *args, **kwargs)	Naming conventions.
plotVecMatrix (page 499)(xvec, yvec, vals[, full])	DEPRECATED for nameing
registerShowPendingFigsAtExit (page 499)()	If called it register a closing function that will ensure all pending MPL figures are shown.
renameDepthTicks (page 499)(ax)	Switch signs of depth ticks to be positive
saveAnimation (page 499)(mesh, data, out[, vData, plc, ...])	Create and save an animation for a given mesh with a set of field data.
saveAxes (page 499)(ax, filename[, adjust])	Save axes as pdf.
saveFigure (page 499)(fig, filename[, pdfTrim])	Save figure as pdf.
setCbarLevels (page 499)(cbar[, cMin, cMax, nLevs, levels])	Set colorbar levels given a number of levels and min/max values.
setMappableData (page 499)(mappable, dataIn[, cMin, ...])	Change the data values for a given mappable.
setOutputStyle (page 499)([dim, paperMargin, xScale, ...])	Set preferred output style.

continues on next page

Table 5 – continued from previous page

setPlotStuff (page 499)([fontsize, dpi])	TODO merge with setOutputStyle.
show1dmodel (page 499)(x[, thk, xlab, zlab, islog, z0])	Show 1d block model defined by value and thickness vectors.
showDataContainerAsMatrix (page 499)(data[, x, y, v])	Plot data container as matrix (cross-plot).
showDataMatrix (page 500)(mat[, xmap, ymap])	Show value map as matrix.
showStitchedModels (page 500)(models[, ax, x, cMin, ...])	Show several 1d block models as (stitched) section.
showValMapPatches (page 500)(vals[, xVec, yVec, dx, dy])	Show values as patches over x and y vector.
showVecMatrix (page 501)(xvec, yvec, vals[, full])	Plot three vectors as matrix.
showfdemsounding (page 501)(freq, inphase, quadrat[, ...])	Show FDEM sounding as real(inphase) and imaginary (quadrature) fields.
showmymatrix (page 501)(mat, x, y[, dx, dy, xlab, ...])	What is this good for?.
twin (page 501)(ax)	Return the twin of ax if exist.
underlayBKGMap (page 501)(ax[, mode, utmzone, epsg, ...])	Underlay digital orthophoto or topographic (mode='DTK') map under axes.
underlayMap (page 502)(ax, proj[, vendor, zoom, ...])	Get a map from public mapping server and underlay it on the given ax.
updateAxes (page 502)(ax[, force])	For internal use.
updateColorBar (page 502)(cbar[, gci, cMin, cMax, ...])	Update colorbar values.
updateFig (page 503)(fig[, force, sleep])	For internal use.
wait (page 503)(**kwargs)	TODO WRITEME.

Classes

BoreHole (page 503)(fname)	Class for handling (structural) borehole data for inclusion in plots.
BoreHoles (page 503)(fnames)	Class to load and handle several boreholes belonging to one profile.
CellBrowser (page 503)(mesh[, data, ax])	Interactive cell browser on current or specified ax for a given mesh.

8.8.1.2 Functions

pygimli.viewer.mpl.**addCoverageAlpha**(*patches*, *coverage*, *dropThreshold=0.4*)

Add alpha values to the colors of a polygon collection.

Parameters

- **patches** (*2D mpl mappable*) –
- **coverage** (*array*) – coverage values. Maximum coverage mean no opaqueness.

- **dropThreshold** (*float*) – relative minimum coverage

pygimli.viewer.mpl.**adjustWorldAxes**(*ax*)

Set some common default properties for an axe.

pygimli.viewer.mpl.**autolevel**(*z*, *nLevs*, *logScale=None*, *zMin=None*, *zMax=None*)

Create nLevs bins for the data array z based on matplotlib ticker.

Examples

```
>>> import numpy as np
>>> from pygimli.viewer.mpl import autolevel
>>> x = np.linspace(1, 10, 100)
>>> autolevel(x, 3)
array([ 1. ,  5.5, 10. ])
>>> x = np.linspace(1, 1000, 100)
>>> autolevel(x, 4, logScale=True)
array([   1.,   10.,  100., 1000.])
```

pygimli.viewer.mpl.**cacheFileName**(*fullname*, *vendor*)

Createfilename and path to cache download data.

pygimli.viewer.mpl.**cmapFromName**(*cmapname='jet'*, *ncols=256*, *bad=None*, ***kwargs*)

Get a colormap either from name or from keyworld list.

See http://matplotlib.org/examples/color/colormaps_reference.html

Parameters

- **cmapname** (*str*) – Name for the colormap.
- **ncols** (*int*) – Amount of colors.
- **bad** (*[r,g,b,a]*) – Default color for bad values [nan, inf] [white]
- **kwargs** (****) –

  cMap
  [str] Name for the colormap

  cmap
  [str] colormap name (old)

Returns
matplotlib Colormap

Return type
cMap

pygimli.viewer.mpl.**createAnimation**(*fig*, *animate*, *nFrames*, *dpi*, *out*)

Create animation for the content of a given matplotlib figure.

Until I know a better place.

pygimli.viewer.mpl.**createColorBar**(*gci*, *orientation='horizontal'*, *size=0.2*, *pad=None*, ***kwargs*)

Create a Colorbar.

Shortcut to create a matplotlib colorbar within the ax for a given patchset. The colorbar is stored in the axes object as __cBar__ to avoid duplicates.

Parameters

- **gci** (*matplotlib graphical instance*) –
- **orientation** (*string*) –
- **size** (*float*) –
- **pad** (*float*) –
- ****kwargs** –

  onlyColorSet: bool (False)
  If set to true, only the gci aka mappable values are changed and no colorBar will be created.

  Forwarded to updateColorBar

pygimli.viewer.mpl.**createColorBarOnly**(*cMin=1*, *cMax=100*, *logScale=False*, *cMap=None*, *nLevs=5*, *label=None*, *orientation='horizontal'*, *savefig=None*, *ax=None*, ***kwargs*)

Create figure with a colorbar.

Create figure with a colorbar.

Parameters
****kwargs** – Forwarded to mpl.colorbar.ColorbarBase.

Returns
The created figure.

Return type
fig

Examples

```
>>> # import pygimli as pg
>>> # from pygimli.viewer.mpl import createColorBarOnly
>>> # createColorBarOnly(cMin=0.2, cMax=5, logScale=False,
>>> #                    cMap='b2r',
>>> #                    nLevs=7,
>>> #                    label=r'Ratio',
>>> #                    orientation='horizontal')
>>> # pg.wait()
```

pygimli.viewer.mpl.**createMeshPatches**(*ax*, *mesh*, *rasterized=False*, *verbose=True*)

Utility function to create 2d mesh patches within a given ax.

pygimli.viewer.mpl.**createTriangles**(*mesh*)

Generate triangle objects for later drawing.

Creates triangle for each 2D triangle cell or 3D boundary. Quads will be split into two triangles. Result will be cached into mesh._triData.

Parameters

mesh (GIMLI::Mesh) – 2D mesh or 3D mesh

Returns

- **x** (*numpy array*) – x position of nodes
- **y** (*numpy array*) – x position of nodes
- **triangles** (*numpy array Cx3*) – cell indices for each triangle, quad or boundary face
- **z** (*numpy array*) – z position for given indices
- **dataIdx** (*list of int*) – List of indices for a data array

pygimli.viewer.mpl.**createTwinX**(*ax*)

Utility function to create (or return existing) twin x axes for ax.

pygimli.viewer.mpl.**createTwinY**(*ax*)

Utility function to create (or return existing) twin x axes for ax.

pygimli.viewer.mpl.**create_legend**(*ax*, *cmap*, *ids*, *classes*)

Create a list of patch objects that can be used for borehole legends.

pygimli.viewer.mpl.**deg2MapTile**(*lon_deg*, *lat_deg*, *zoom*)

TODO Documentme.

pygimli.viewer.mpl.**draw1DColumn**(*ax*, *x*, *val*, *thk*, *width=30*, *ztopo=0*, *cmin=1*, *cmax=1000*, *cmap=None*, *name=None*, *textoffset=0.0*)

Draw a 1D column (e.g., from a 1D inversion) on a given ax.

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from pygimli.viewer.mpl import draw1DColumn
>>> thk = [1, 2, 3, 4]
>>> val = thk
>>> fig, ax = plt.subplots()
>>> draw1DColumn(ax, 0.5, val, thk, width=0.1, cmin=1, cmax=4, name="VES")
<matplotlib.collections.PatchCollection object at ...>
>>> _ = ax.set_ylim(-np.sum(thk), 0)
```

pygimli.viewer.mpl.**draw1dmodel**(*x*, *thk=None*, *xlab=None*, *zlab='z in m'*, *islog=True*, *z0=0*)

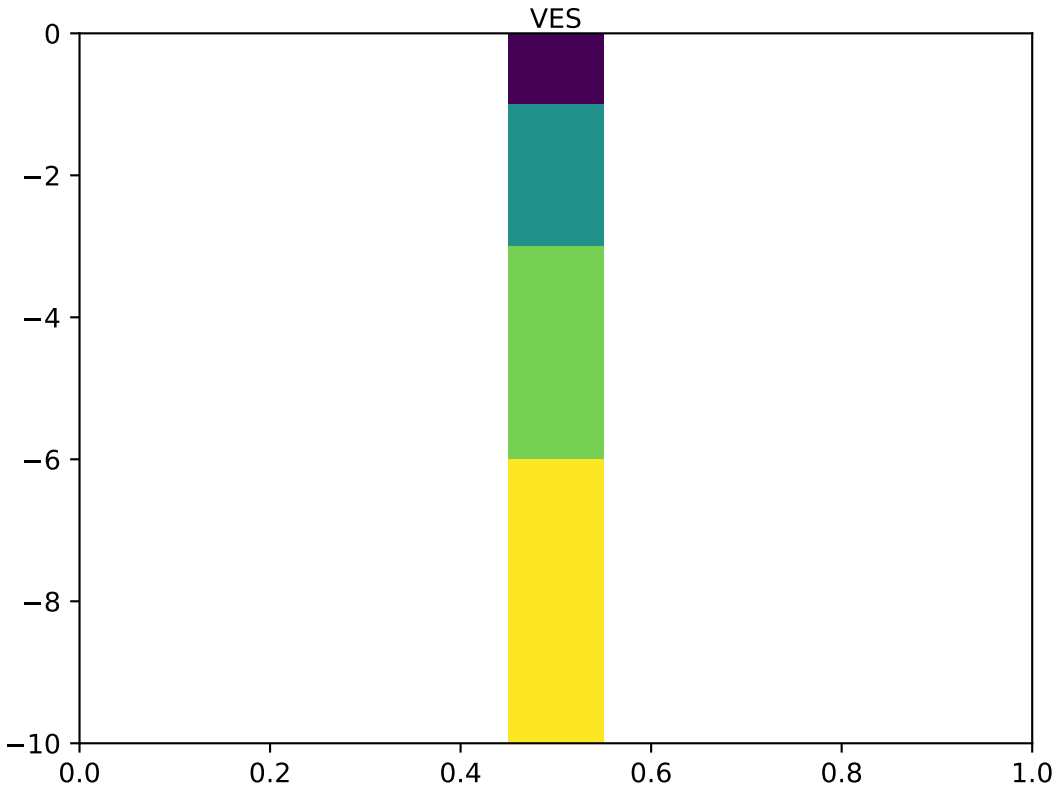
DEPRECATED.

pygimli.viewer.mpl.**draw1dmodelErr**(*x*, *xL*, *xU=None*, *thk=None*, *xcol='g'*, *ycol='r'*, ***kwargs*)

TODO.

pygimli.viewer.mpl.**draw1dmodelLU**(*x*, *xL*, *xU*, *thk=None*, ***kwargs*)

Draw 1d model with lower and upper bounds.



pygimli.viewer.mpl.**drawBlockMatrix**(*ax*, *mat*, ***kwargs*)

Draw a view of a matrix into the axes.

Parameters

- **ax** (*mpl axis instance, optional*) – Axis instance where the matrix will be plotted.
- **mat** (*pg.Matrix.BlockMatrix*) –

Keyword Arguments

spy (*bool [False]*) – Draw all matrix entries instead of colored blocks

Return type

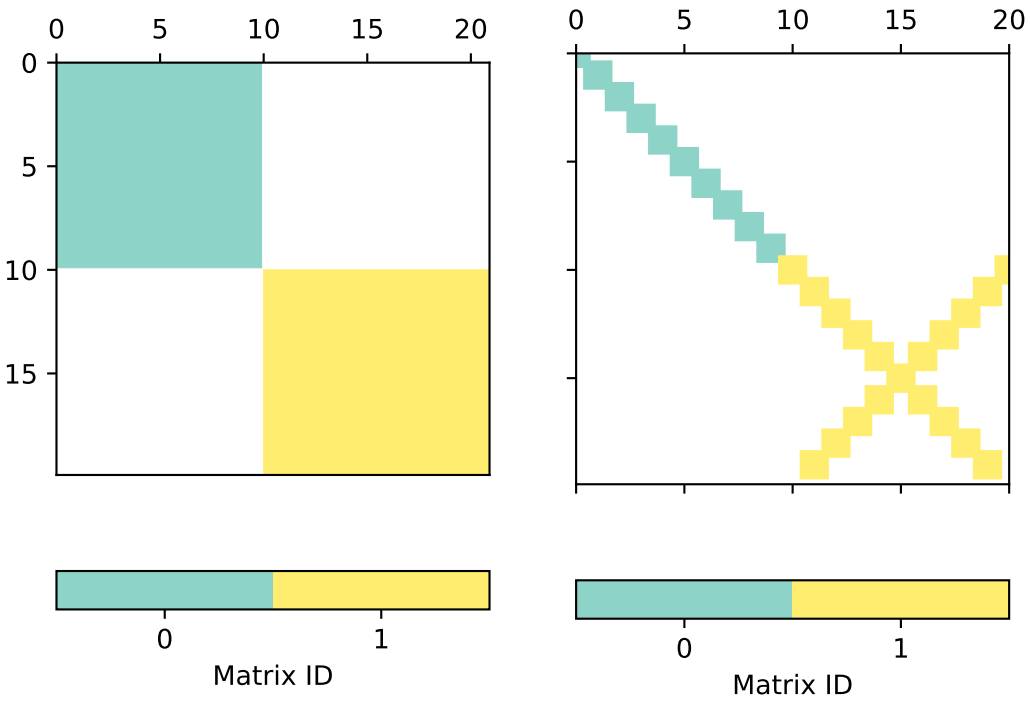
ax

```
>>> import numpy as np
>>> import pygimli as pg
>>> I = pg.matrix.IdentityMatrix(10)
>>> SM = pg.matrix.SparseMapMatrix()
>>> for i in range(10):
...     SM.setVal(i, 10 - i, 5.0)
...     SM.setVal(i,  i, 5.0)
>>> B = pg.matrix.BlockMatrix()
>>> B.add(I, 0, 0)
0
>>> B.add(SM, 10, 10)
1
>>> print(B)
```

(continues on next page)

(continued from previous page)

```
pg.matrix.BlockMatrix of size 20 x 21 consisting of 2 submatrices.
>>> fig, (ax1, ax2) = pg.plt.subplots(1, 2, sharey=True)
>>> _ = pg.show(B, ax=ax1)
>>> _ = pg.show(B, spy=True, ax=ax2)
```



pygimli.viewer.mpl.**drawBoundaryMarkers**(*ax*, *mesh*, *clipBoundaryMarkers=False*, ***kwargs*)

Draw boundary markers for mesh.boundaries with marker != 0

Parameters

- **mesh** (GIMLI::Mesh) – Mesh that have the boundary markers.
- **clipBoundaryMarkers** (*bool [False]*) – Clip boundary marker to the axes limits if needed.

Keyword Arguments

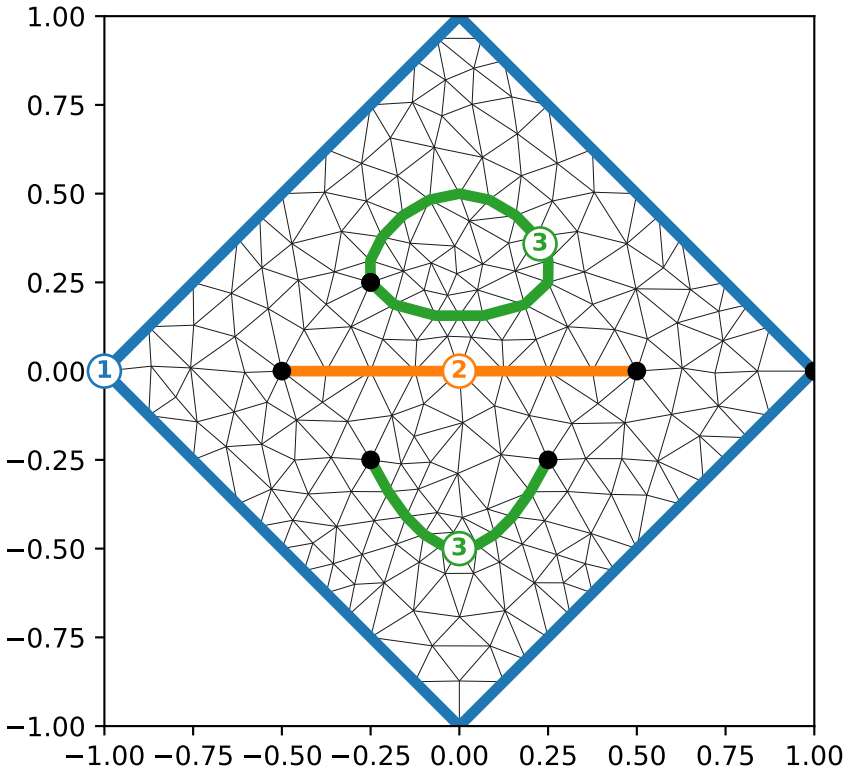
****kwargs** – Forwarded to plot

```
>>> import pygimli as pg
>>> import pygimli.meshtools as mt
>>> c0 = mt.createCircle(pos=(0.0, 0.0), radius=1, nSegments=4)
>>> l0 = mt.createPolygon([[-0.5, 0.0], [.5, 0.0]], boundaryMarker=2)
>>> l1 = mt.createPolygon([[-0.25, -0.25], [0.0, -0.5], [0.25, -0.25]],
...                       interpolate='spline', addNodes=4,
...                       boundaryMarker=3)
>>> l2 = mt.createPolygon([[-0.25, 0.25], [0.0, 0.5], [0.25, 0.25]],
...                       interpolate='spline', addNodes=4,
...                       isClosed=True, boundaryMarker=3)
>>> mesh = mt.createMesh([c0, l0, l1, l2], area=0.01)
```

(continues on next page)

(continued from previous page)

```
>>> ax, _ = pg.show(mesh)
>>> pg.viewer.mpl.drawBoundaryMarkers(ax, mesh)
```



pygimli.viewer.mpl.**drawDataMatrix**(*ax*, *mat*, *xmap=None*, *ymap=None*, *cMin=None*, *cMax=None*, *logScale=None*, *label=None*, ***kwargs*)

Draw previously generated (generateVecMatrix) matrix.

Parameters

- **ax** (*mpl.axis*) – axis to plot, if not given a new figure is created
- **mat** (*numpy.array2d*) – matrix to show
- **xmap** (*dict {i:num}*) – dict (must match A.shape[0])
- **ymap** (*iterable*) – vector for x axis (must match A.shape[0])
- **cMin/cMax** (*float*) – minimum/maximum color values
- **logScale** (*bool*) – logarithmic colour scale [min(A)>0]
- **label** (*string*) – colorbar label

pygimli.viewer.mpl.**drawField**(*ax*, *mesh*, *data=None*, *levels=None*, *nLevs=5*, *cMin=None*, *cMax=None*, *nCols=None*, *logScale=False*, *fitView=True*, ***kwargs*)

Draw mesh with scalar field data.

Draw scalar field into MPL axes using matplotlib triplot. Only for triangle/quadrangle meshes currently

Parameters

- **ax** (*mpl axe*) –
- **mesh** (GIMLI::Mesh) – 2D mesh
- **data** (*iterable*) – Scalar field values. Can be of length mesh.cellCount() or mesh.nodeCount().
- **levels** (*iterable of type float*) – Values for contour lines. If empty auto generated from nLevs.
- **nLevs** (*int*) – Number of contour levels based on cMin, cMax and logScale.
- **cMin** (*float [None]*) – Minimal contour value. If None min(data).
- **cMax** (*float [None]*) – Maximal contour value. If None max(data).
- **logScale** (*bool [False]*) – Levels and colors distributes with logarithmic scale.
- **fitView** (*bool [True]*) – Adjust ax limits to mesh bounding box.

Keyword Arguments

- **shading** (*'flat' | 'gouraud'*) –
- **fillContour** (*[True]*) –
- **contourLines** (*[True]*) –
- ****kwargs** – Additional kwargs forwarded to ax.tripcolor, ax.tricontour, ax.tricontourf

Returns

gci – The current image object useful for post color scaling

Return type

image object

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import pygimli as pg
>>> from pygimli.viewer.mpl import drawField
>>> n = np.linspace(0, -2, 11)
>>> mesh = pg.createGrid(x=n, y=n)
>>> nx = pg.x(mesh.positions())
>>> ny = pg.y(mesh.positions())
>>> data = np.cos(1.5 * nx) * np.sin(1.5 * ny)
>>> fig, ax = plt.subplots()
>>> drawField(ax, mesh, data)
<matplotlib.tri.tricontour.TriContourSet ...>
```

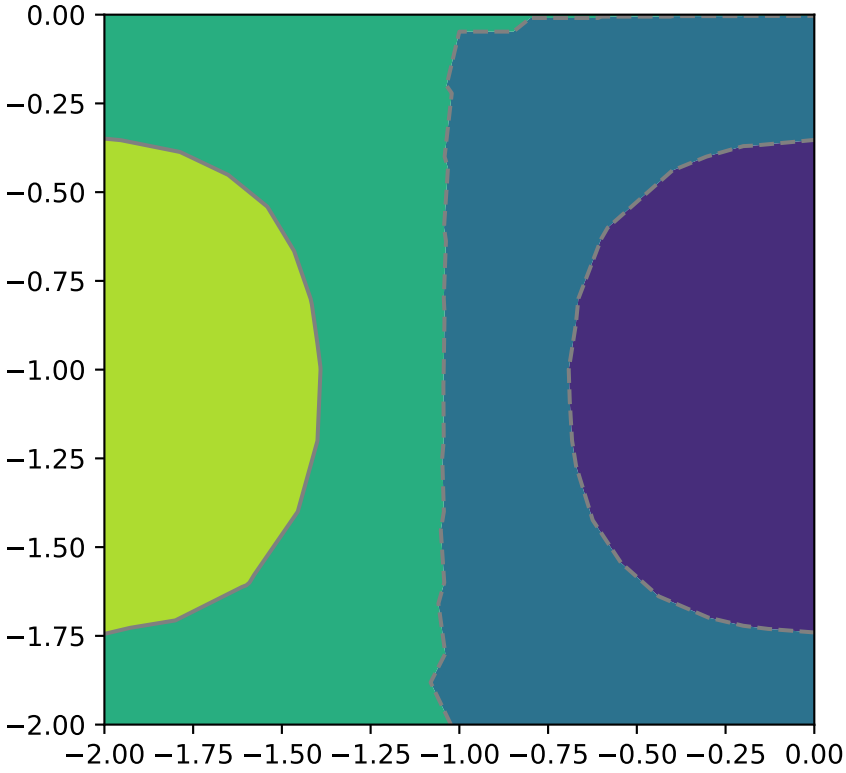
pygimli.viewer.mpl.**drawMesh**(*ax*, *mesh*, *fitView=True*, ***kwargs*)

Draw a 2d mesh into a given ax.

Set the limits of the ax tor the mesh extent.

Parameters

- **ax** (*mpl axe instance*) – Axis instance where the mesh is plotted.
- **mesh** (GIMLI::Mesh) – The 2D mesh which will be drawn.



- **fitView** (*bool [True]*) – Adjust ax limits to mesh bounding box.

Keyword Arguments

- ****kwargs** – Additional kwargs forward to drawPLC or drawMeshBoundaries.
- **%(drawMeshBoundaries)** –
- **%(drawPLC)** –

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import pygimli as pg
>>> from pygimli.viewer.mpl import drawMesh
>>> n = np.linspace(1, 2, 10)
>>> mesh = pg.createGrid(x=n, y=n)
>>> fig, ax = plt.subplots()
>>> drawMesh(ax, mesh)
>>> plt.show()
```

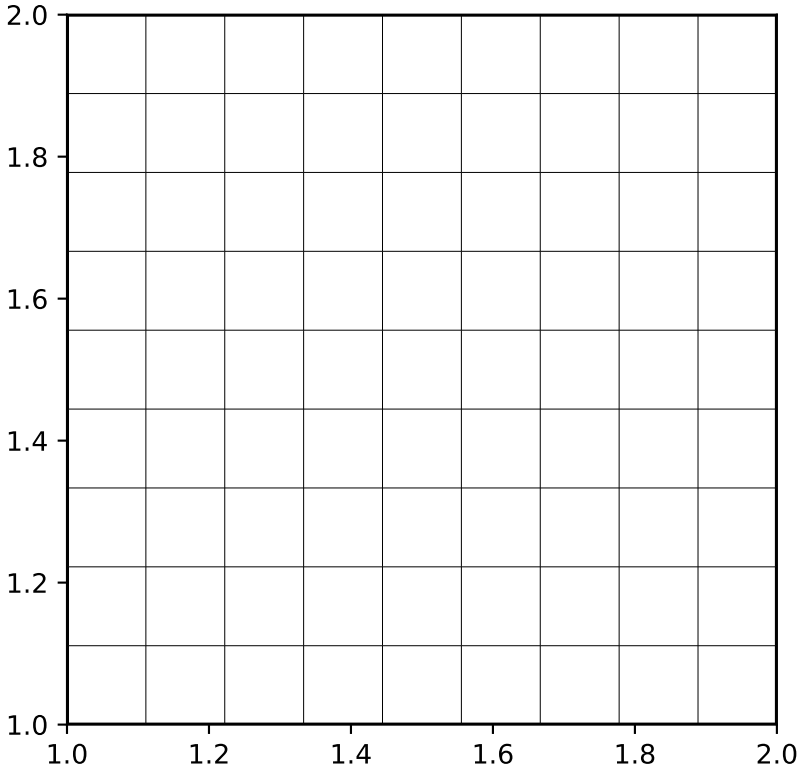
Examples using `pygimli.viewer.mpl.drawMesh`

- *Raypaths in layered and gradient models* (page 46)
- *Mesh interpolation* (page 200)

`pygimli.viewer.mpl.`**`drawMeshBoundaries`**(*ax*, *mesh*, *hideMesh=False*, *useColorMap=False*, *fitView=True*, *lw=None*, *color=None*, ***kwargs*)

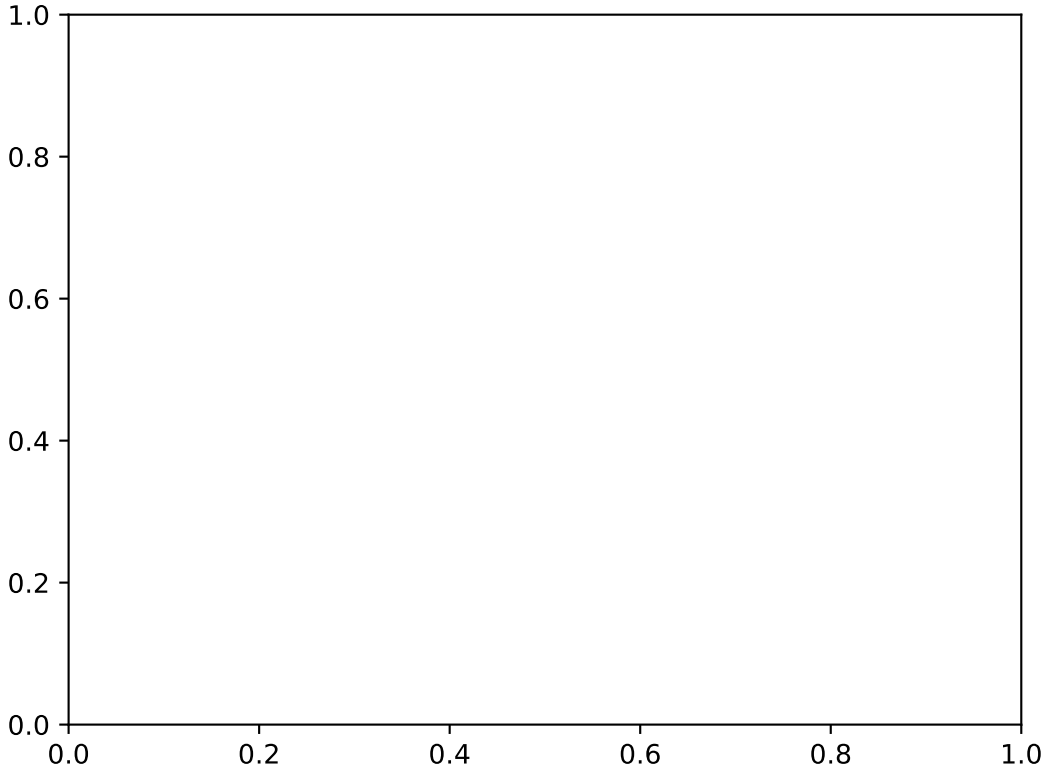
Draw mesh on ax with boundary conditions colorized.

Parameters



- **mesh** (GIMLI::Mesh) –
- **hideMesh** (*bool [False]*) – Show only the boundary of the mesh and omit inner edges that separate the cells.
- **useColorMap** (*bool[False]*) – Apply the default colormap to boundaries with marker values > 0
- **fitView** (*bool [True]*) – Adjust ax limits to mesh bounding box.
- **lw** (*float [None]*) – Linewidth. When set to None then lw depends on boundary marker. Linewidth [0.3] for edges with marker == 0 if hideMesh is False.
- **color** (*None*) – Color for special lines. If set to None automatic "black".

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import pygimli as pg
>>> from pygimli.viewer.mpl import drawMeshBoundaries
>>> n = np.linspace(0, -2, 11)
>>> mesh = pg.createGrid(x=n, y=n)
>>> for bound in mesh.boundaries():
...     if not bound.rightCell():
...         bound.setMarker(pg.core.MARKER_BOUND_MIXED)
...     if bound.center().y() == 0:
...         bound.setMarker(pg.core.MARKER_BOUND_HOMOGEN_NEUMANN)
>>> fig, ax = plt.subplots()
>>> drawMeshBoundaries(ax, mesh)
```



pygimli.viewer.mpl.**drawModel**(*ax*, *mesh*, *data=None*, *tri=False*, *rasterized=False*, *cMin=None*, *cMax=None*, *logScale=False*, *xlabel=None*, *ylabel=None*, *fitView=True*, *verbose=False*, ***kwargs*)

Draw a 2d mesh and color the cell by the data.

Parameters

- **ax** (*mpl axis instance, optional*) – Axis instance where the mesh is plotted (default is current axis).
- **mesh** (GIMLI::Mesh) – The plotted mesh to browse through.
- **data** (*array, optional*) – Data to draw. Should either equal numbers of cells or nodes of the corresponding *mesh*.
- **tri** (*boolean, optional*) – use MPL tripcolor (experimental)
- **rasterized** (*boolean, optional*) – Rasterize mesh patches to reduce file size and avoid zooming artifacts in some PDF viewers.
- **fitView** (*bool [True]*) – Adjust ax limits to mesh bounding box.

Keyword Arguments

****kwargs** – Additional kwargs forwarded to the draw functions and mpl methods, respectively.

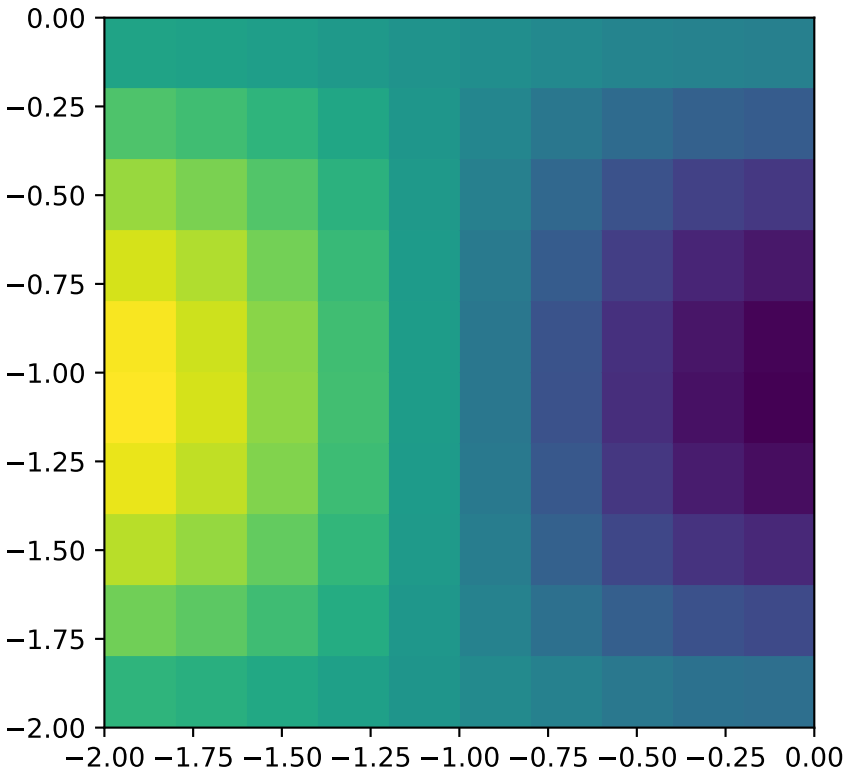
Returns

gci

Return type

matplotlib graphics object

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import pygimli as pg
>>> from pygimli.viewer.mpl import drawModel
>>> n = np.linspace(0, -2, 11)
>>> mesh = pg.createGrid(x=n, y=n)
>>> mx = pg.x(mesh.cellCenter())
>>> my = pg.y(mesh.cellCenter())
>>> data = np.cos(1.5 * mx) * np.sin(1.5 * my)
>>> fig, ax = plt.subplots()
>>> drawModel(ax, mesh, data)
<matplotlib.collections.PolyCollection object at ...>
```



Examples using `pygimli.viewer.mpl.drawModel`

- *Mesh interpolation* (page 200)

`pygimli.viewer.mpl.`**`drawModel1D`**(*ax*, *thickness=None*, *values=None*, *model=None*, *depths=None*, *plot='plot'*, *xlabel='Resistivity $(\\Omega$m$)$'*, *zlabel='Depth (m)'*, *z0=0*, ***kwargs*)

Draw 1d block model into axis ax.

Draw 1d block model into axis ax defined by values and thickness vectors using plot function. For log y cases, z0 should be set > 0 so that the default becomes 1.

Parameters

- **ax** (*mpl axes*) – Matplotlib Axes object to plot into.
- **values** (*iterable [float]*) – [N] Values for each layer plus lower

background.

- **thickness** (*iterable [float]*) – [N-1] thickness for each layer. Either thickness or depths must be set.
- **depths** (*iterable [float]*) – [N-1] Values for layer depths (positive z-coordinates). Either thickness or depths must be set.
- **model** (*iterable [float]*) – Shortcut to use default model definition. thks = model[0:nLay] values = model[nLay:]
- **plot** (*string*) – Matplotlib plotting function. 'plot', 'semilogx', 'semilogy', 'loglog'
- **xlabel** (*str*) – Label for x axis.
- **ylabel** (*str*) – Label for y axis.
- **z0** (*float*) – Starting depth in m
- ****kwargs** (*dict()*) – Forwarded to the plot routine

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> import pygimli as pg
>>> # plt.style.use('ggplot')
>>> thk = [1, 4, 4]
>>> res = np.array([10., 5, 15, 50])
>>> fig, ax = plt.subplots()
>>> pg.viewer.mpl.drawModel1D(ax, values=res*5, depths=np.cumsum(thk),
...                           plot='semilogx', color='blue')
>>> pg.viewer.mpl.drawModel1D(ax, values=res, thickness=thk, z0=1,
...                           plot='semilogx', color='red')
>>> pg.wait()
```

Examples using `pygimli.viewer.mpl.drawModel1D`

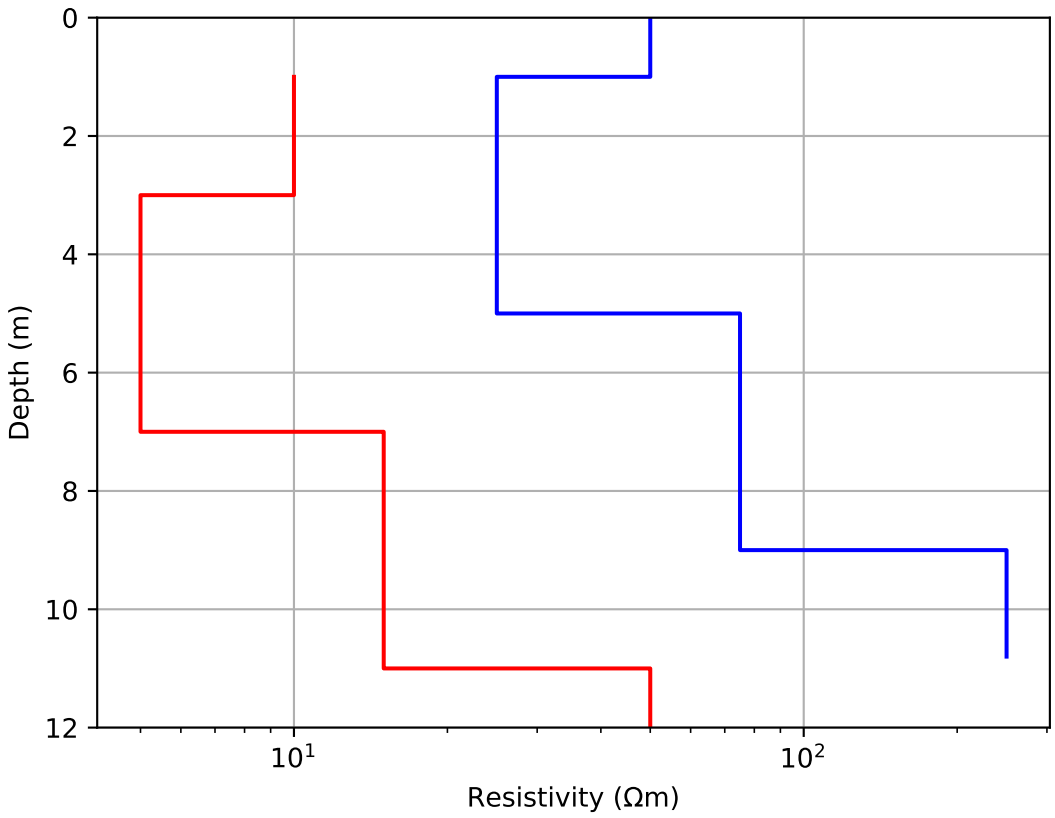
- *DC-EM Joint inversion* (page 149)
- *VES inversion for a blocky model* (page 231)
- *VES inversion for a smooth model* (page 234)

pygimli.viewer.mpl.**drawPLC**(*ax*, *mesh*, *fillRegion=True*, *regionMarker=True*, *boundaryMarkers=False*, *showNodes=False*, *fitView=True*, ***kwargs*)

Draw 2D PLC into given axes.

Parameters

- **ax** (*mpl axe*) –
- **mesh** (GIMLI::Mesh) –
- **fillRegion** (*bool [True]*) – Fill the regions with default colormap.
- **regionMarker** (*bool [True]*) – Show region marker.
- **boundaryMarkers** (*bool [False]*) – Show boundary marker.
- **showNodes** (*bool [False]*) – Draw all nodes as little dots.



- **fitView** (*bool [True]*) – Adjust ax limits to mesh bounding box.

Keyword Arguments

****kwargs** – Additional kwargs forwarded to the draw functions and mpl methods, respectively.

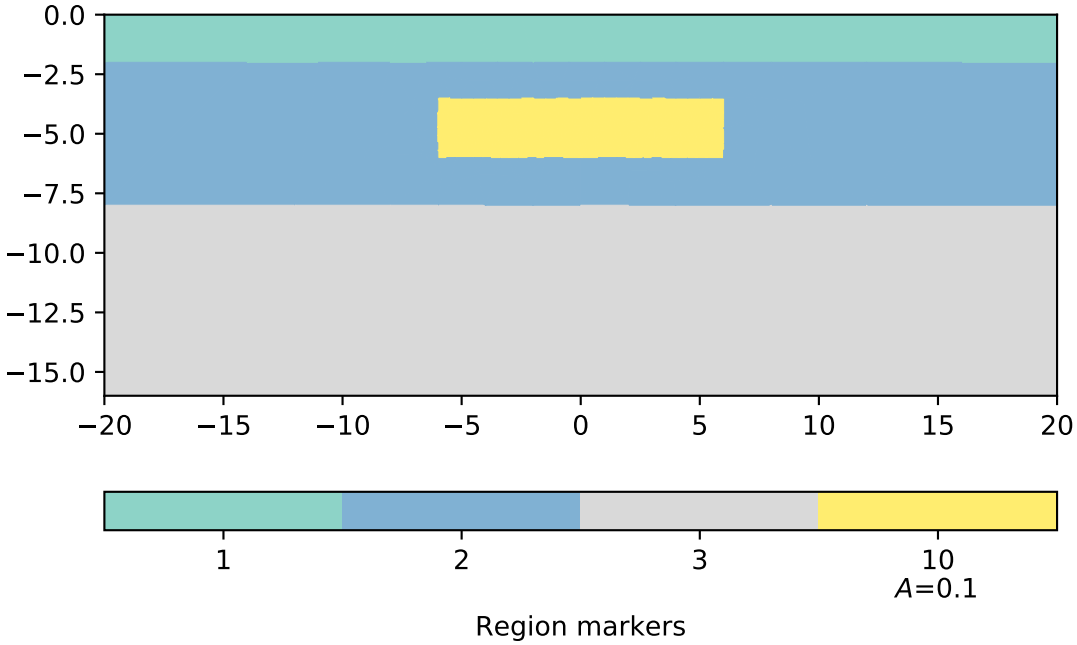
```
>>> import matplotlib.pyplot as plt
>>> import pygimli as pg
>>> import pygimli.meshtools as mt
>>> # Create geometry definition for the modelling domain
>>> world = mt.createWorld(start=[-20, 0], end=[20, -16],
...                        layers=[-2, -8], worldMarker=False)
>>> # Create a heterogeneous block
>>> block = mt.createRectangle(start=[-6, -3.5], end=[6, -6.0],
...                            marker=10,  boundaryMarker=10, area=0.1)
>>> fig, ax = plt.subplots()
>>> geom = world + block
>>> _ = pg.viewer.mpl.drawPLC(ax, geom)
```

pygimli.viewer.mpl.**drawParameterConstraints**(*ax*, *mesh*, *cMat*, *cWeights=None*)

Draw inter parameter constraints between cells.

Parameters

- **ax** (*MPL axes*) –
- **mesh** (GIMLI::Mesh) – 2d mesh
- **cMat** (GIMLI::SparseMapMatrix) – ConstraintsMatrix



- **cWeights** (*iterable float*) – Weights for all constraints. Need to have a lengths == cMat.rows()

pygimli.viewer.mpl.**drawSelectedMeshBoundaries**(*ax*, *boundaries*, *color=None*, *linewidth=1.0*, *linestyles='-'*, ***kwargs*)

Draw mesh boundaries into a given axes.

Parameters

- **ax** (*matplotlib axes*) – axes to plot into
- **boundaries** (GIMLI::Mesh boundary vector) – collection of boundaries to plot
- **color** (matplotlib color |str [None]) – matching color or string, else colors are according to markers
- **linewidth** (*float [1.0]*) – line width
- **linestyles** (*linestyle for line collection, i.e. solid or dashed*) –

Returns

lco

Return type

matplotlib line collection object

Examples using `pygimli.viewer.mpl.drawSelectedMeshBoundaries`

- *Petrophysical joint inversion* (page 153)

pygimli.viewer.mpl.**drawSelectedMeshBoundariesShadow**(*ax*, *boundaries*, *first='x'*, *second='y'*, *color=(0.3, 0.3, 0.3, 1.0)*)

Draw mesh boundaries as shadows into a given axes.

Parameters

- **ax** (*matplotlib axes*) – axes to plot into
- **boundaries** (GIMLI::Mesh boundary vector) – collection of boundaries to plot
- **second** (*first /*) – attribute names to retrieve from nodes
- **color** (matplotlib color |str [None]) – matching color or string, else colors are according to markers
- **linewidth** (*float [1.0]*) – line width

Returns

lco

Return type

matplotlib line collection object

pygimli.viewer.mpl.**drawSensorAsMarker**(*ax*, *data*)

Draw Sensor marker, these marker are pickable.

pygimli.viewer.mpl.**drawSensors**(*ax*, *sensors*, *diam=None*, *coords=None*, ***kwargs*)

Draw sensor positions as black dots with a given diameter.

Parameters

- **ax** (*mpl axe instance*) –
- **sensors** (*vector or list of RVector3*) – List of positions to plot.
- **diam** (*float [None]*) – Diameter of circles (None leads to point distance by 4).
- **coords** (*(int, int) [0, 1]*) – Coordinates to take (usually x and y).

Keyword Arguments

****kwargs** – Additional kwargs forwarded to mpl.PatchCollection, mpl.Circle

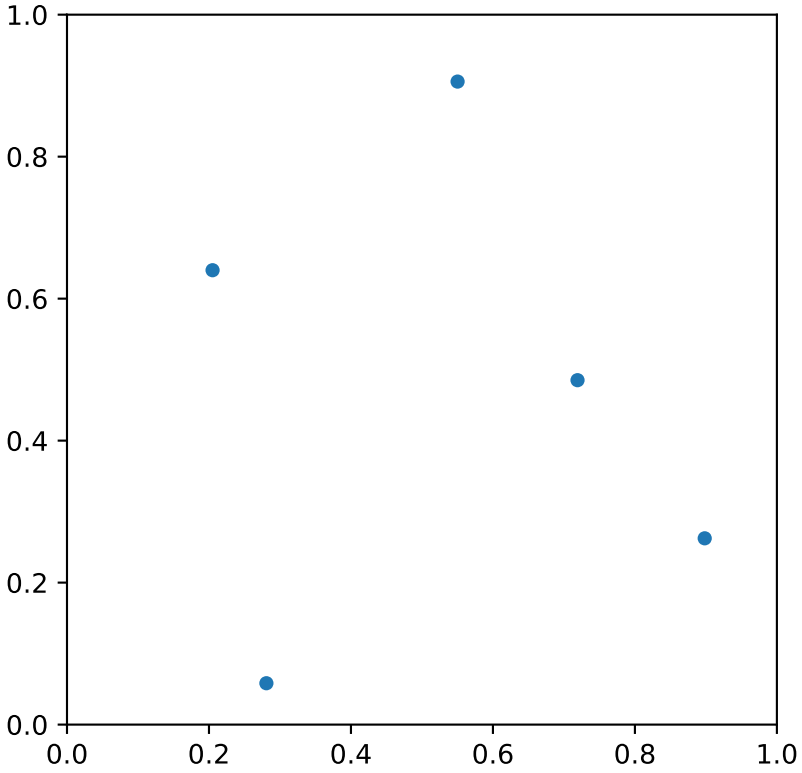
```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import pygimli as pg
>>> from pygimli.viewer.mpl import drawSensors
>>> sensors = np.random.rand(5, 2)
>>> fig, ax = pg.plt.subplots()
>>> drawSensors(ax, sensors, diam=0.02, coords=[0, 1])
>>> ax.set_aspect('equal')
>>> pg.wait()
```

Examples using pygimli.viewer.mpl.drawSensors

- *2D Refraction modeling and inversion* (page 34)
- *Petrophysical joint inversion* (page 153)

pygimli.viewer.mpl.**drawSparseMatrix**(*ax*, *mat*, ***kwargs*)

Draw a view of a matrix into the axes.



Parameters

- **ax** (*mpl axis instance, optional*) – Axis instance where the matrix will be plotted.
- **mat** (*pg.matrix.SparseMatrix or pg.matrix.SparseMapMatrix*) –

Return type

mpl.lines.line2d

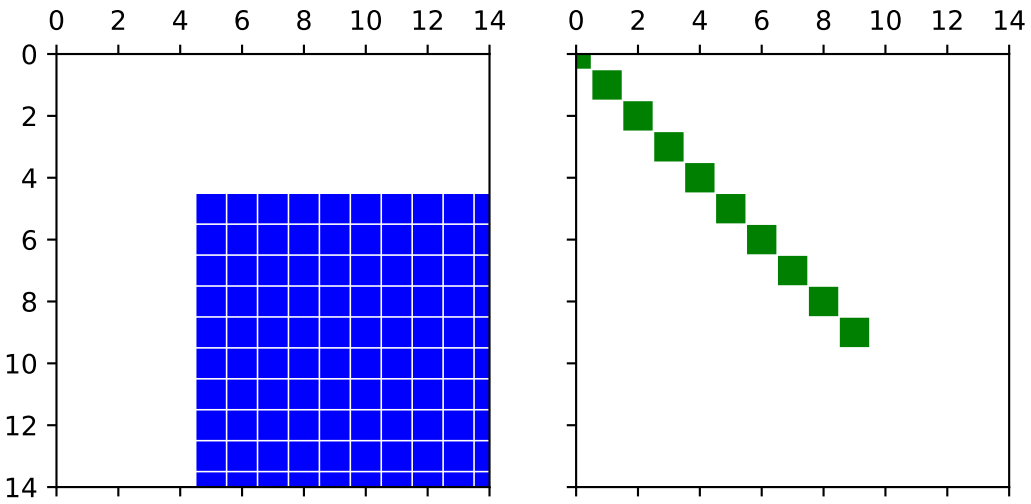
```
>>> import numpy as np
>>> import pygimli as pg
>>> from pygimli.viewer.mpl import drawSparseMatrix
>>> A = pg.randn((10,10), seed=0)
>>> SM = pg.core.SparseMapMatrix()
>>> for i in range(10):
...     SM.setVal(i, i, 5.0)
>>> fig, (ax1, ax2) = pg.plt.subplots(1, 2, sharey=True, sharex=True)
>>> _ = drawSparseMatrix(ax1, A, colOffset=5, rowOffset=5, color='blue')
>>> _ = drawSparseMatrix(ax2, SM, color='green')
```

pygimli.viewer.mpl.**drawStreamLines**(*ax*, *mesh*, *u*, *nx=25*, *ny=25*, ***kwargs*)

Draw streamlines for the gradients of field values u on a mesh.

The matplotlib routine streamplot needs equidistant spacings so we interpolate first on a grid defined by nx and ny nodes. Additionally arguments are piped to streamplot.

This works only for rectangular regions. You should use pg.viewer.mpl.drawStreams, which is more comfortable and more flexible.



Parameters

- **ax** (*mpl axe*) –
- **mesh** (GIMLI::Mesh) – 2D mesh
- **u** (*iterable float*) – Scalar data field.

pygimli.viewer.mpl.**drawStreams**(*ax*, *mesh*, *data*, *startStream=3*, *coarseMesh=None*, *quiver=False*, ***kwargs*)

Draw streamlines based on an unstructured mesh.

Every cell contains only one streamline and every new stream line starts in the center of a cell. You can alternatively provide a second mesh with coarser mesh to draw streams for.

Parameters

- **ax** (*matplotlib.ax*) – ax to draw into
- **mesh** (GIMLI::Mesh) – 2d mesh
- **data** (*iterable float | [float, float] | pg.core.R3Vector*) – If data is an array (per cell or node) gradients are calculated otherwise the data will be interpreted as vector field per nodes or cell centers.
- **startStream** (*int*) – variate the start stream drawing, try values from 1 to 3 what every you like more.
- **coarseMesh** (GIMLI::Mesh) – Instead of draw a stream for every cell in mesh, draw a streamline segment for each cell in coarseMesh.
- **quiver** (*bool [False]*) – Draw arrows instead of streamlines.

Keyword Arguments

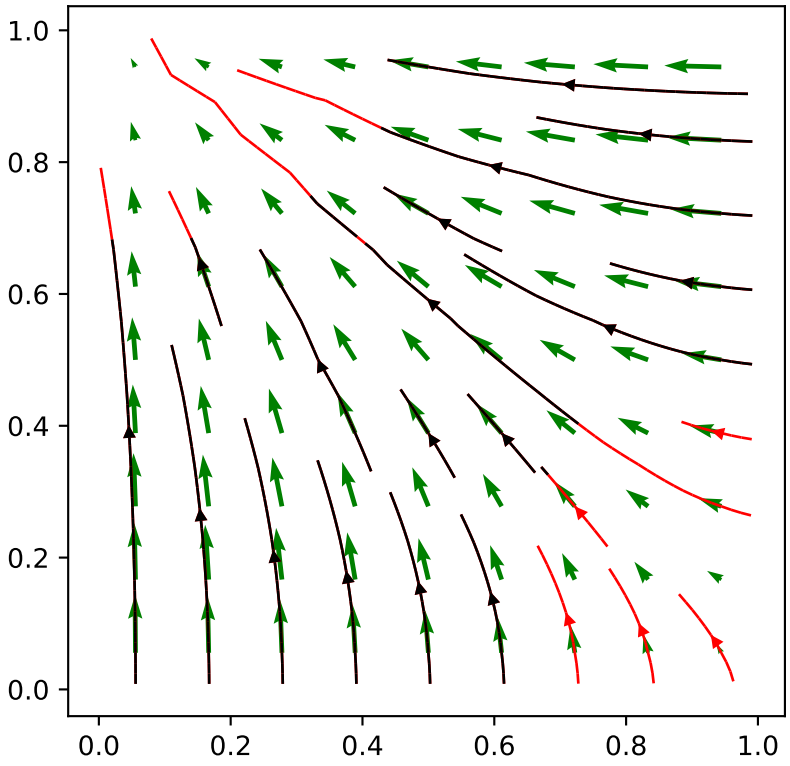
****kwargs** – Additional kwargs forwarded to axe.quiver, drawStreamLine

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import pygimli as pg
>>> from pygimli.viewer.mpl import drawStreams
>>> n = np.linspace(0, 1, 10)
```

(continues on next page)

(continued from previous page)

```
>>> mesh = pg.createGrid(x=n, y=n)
>>> nx = pg.x(mesh.positions())
>>> ny = pg.y(mesh.positions())
>>> data = np.cos(1.5 * nx) * np.sin(1.5 * ny)
>>> fig, ax = plt.subplots()
>>> drawStreams(ax, mesh, data, color='red')
>>> drawStreams(ax, mesh, data, dropTol=0.9)
>>> drawStreams(ax, mesh, pg.solver.grad(mesh, data),
...             color='green', quiver=True)
>>> ax.set_aspect('equal')
>>> pg.wait()
```



Examples using `pygimli.viewer.mpl.drawStreams`

- *Geoelectrics in 2.5D* (page 76)
- *Modelling with Boundary Conditions* (page 219)

`pygimli.viewer.mpl.`**`drawValMapPatches`**(*ax*, *vals*, *xVec=None*, *yVec=None*, *dx=1*, *dy=None*, ***kwargs*)

Show values as patches over x and y vector.

Parameters

- **`vals`** (*`iterable`*) – values to plot
- **`xVec/yVec`** (*`iterable`*) – x/y axis values
- **`dx/dy`** (*`float`*) – patch width
- **`circular`** (*`bool`*) – assume circular (cyclic) positions

pygimli.viewer.mpl.**drawVecMatrix**(*ax*, *xvec*, *yvec*, *vals*, *full=False*, ***kwargs*)

pygimli.viewer.mpl.**findAndMaskBestClim**(*dataIn*, *cMin=None*, *cMax=None*, *dropColLimitsPerc=5*, *logScale=False*)

TODO Documentme.

pygimli.viewer.mpl.**generateMatrix**(*xvec*, *yvec*, *vals*, ***kwargs*)

Generate a data matrix from x/y and value vectors.

Parameters

- **xvec** (*iterables (list, np.array, pg.Vector) of same length*) –
- **yvec** (*iterables (list, np.array, pg.Vector) of same length*) –
- **vals** (*iterables (list, np.array, pg.Vector) of same length*) –
- **full** (*bool [False]*) – generate a fully symmetric matrix containing all unique xvec+yvec otherwise A is squeezed to the individual unique vectors

Returns

- **A** (*np.ndarray(2d)*) – matrix containing the values sorted according to unique xvec/yvec
- **xmap/ymap** (*dict {key: num}*) – dictionaries for accessing matrix position (row/col number from x/y[i])

pygimli.viewer.mpl.**getMapTile**(*xtile*, *ytile*, *zoom*, *vendor='OSM'*, *verbose=False*)

Get a map tile from public mapping server.

Its not recommended to use the google maps tile server without the google maps api so if you use it to often your IP will be blacklisted

TODO: look here for more vendors https://github.com/Leaflet/Leaflet TODO: Try http://scitools.org.uk/cartopy/docs/v0.14/index.html TODO: Try https://github.com/jwass/mplleaflet

Parameters

- **xtile** (*int*) –
- **ytile** (*int*) –
- **zoom** (*int*) –
- **vendor** (*str*) – . ‘OSM’ or ‘Open Street Map’ (tile.openstreetmap.org) . ‘GM’ or ‘Google Maps’ (mt.google.com) (do not use it to much)
- **verbose** (*bool [false]*) – be verbose

pygimli.viewer.mpl.**hold**(*val=True*)

TODO WRITEME.

pygimli.viewer.mpl.**insertUnitAtNextLastTick**(*ax*, *unit*, *xlabel=True*, *position=-2*)

Replace the last-but-one tick label by unit symbol.

pygimli.viewer.mpl.**isInteractive**()

Returns False if a non-interactive backend is used, e.g. for Jupyter Notebooks and sphinx builds.

pygimli.viewer.mpl.**mapTile2deg**(*xtile*, *ytile*, *zoom*)

Calculate the NW-corner of the square.

Use the function with xtile+1 and/or ytile+1 to get the other corners. With xtile+0.5 ytile+0.5 it will return the center of the tile.

pygimli.viewer.mpl.**noShow**(*on=True*)

Toggle quiet mode to avoid popping figures.

Parameters

on (*bool [True]*) – Set Matplotlib backend to 'agg' and restore old backend if set to False.

pygimli.viewer.mpl.**patchMatrix**(*mat*, *xmap=None*, *ymap=None*, *ax=None*, *cMin=None*, *cMax=None*, *logScale=None*, *label=None*, *dx=1*, ***kwargs*)

Plot previously generated (generateVecMatrix) matrix.

Parameters

- **mat** (*numpy.array2d*) – matrix to show
- **xmap** (*dict {i:num}*) – dict (must match A.shape[0])
- **ymap** (*iterable*) – vector for x axis (must match A.shape[0])
- **ax** (*mpl.axis*) – axis to plot, if not given a new figure is created
- **cMin/cMax** (*float*) – minimum/maximum color values
- **logScale** (*bool*) – logarithmic colour scale [min(A)>0]
- **label** (*string*) – colorbar label
- **dx** (*float*) – width of the matrix elements (by default 1)

pygimli.viewer.mpl.**patchValMap**(*vals*, *xvec=None*, *yvec=None*, *ax=None*, *cMin=None*, *cMax=None*, *logScale=None*, *label=None*, *dx=1*, *dy=None*, *cTrim=0*, ***kwargs*)

Plot previously generated (generateVecMatrix) y map (category).

Parameters

- **vals** (*iterable*) – Data values to show.
- **xvec** (*dict {i:num}*) – dict (must match vals.shape[0])
- **ymap** (*iterable*) – vector for x axis (must match vals.shape[0])
- **ax** (*mpl.axis*) – axis to plot, if not given a new figure is created
- **cMin/cMax** (*float*) – minimum/maximum color values
- **cTrim** (*float [0]*) – use trim value to exclude outer cTrim percent of data from color scale
- **logScale** (*bool*) – logarithmic colour scale [min(vals)>0]
- **label** (*string*) – colorbar label
- **kwargs** (****) –

– **circular**
[bool] Plot in polar coordinates.

pygimli.viewer.mpl.**plotDataContainerAsMatrix**(**args*, ***kwargs*)

DEPRECATED naming scheme

pygimli.viewer.mpl.**plotLines**(*ax*, *line_filename*, *linewidth=1.0*, *step=1*)

Read lines from file and plot over model.

pygimli.viewer.mpl.**plotMatrix**(*mat*, **args*, ***kwargs*)

Naming conventions. Use drawDataMatrix or showDataMatrix

pygimli.viewer.mpl.**plotVecMatrix**(*xvec*, *yvec*, *vals*, *full=False*, ***kwargs*)

DEPRECATED for nameing

pygimli.viewer.mpl.**registerShowPendingFigsAtExit**()

If called it register a closing function that will ensure all pending MPL figures are shown.

Its only set on demand by pg.show() since we only need it if matplotlib is used.

pygimli.viewer.mpl.**renameDepthTicks**(*ax*)

Switch signs of depth ticks to be positive

pygimli.viewer.mpl.**saveAnimation**(*mesh*, *data*, *out*, *vData=None*, *plc=None*, *label=''*, *cMin=None*, *cMax=None*, *logScale=False*, *cmap=None*, ***kwargs*)

Create and save an animation for a given mesh with a set of field data.

Until I know a better place.

pygimli.viewer.mpl.**saveAxes**(*ax*, *filename*, *adjust=False*)

Save axes as pdf.

pygimli.viewer.mpl.**saveFigure**(*fig*, *filename*, *pdfTrim=False*)

Save figure as pdf.

pygimli.viewer.mpl.**setCbarLevels**(*cbar*, *cMin=None*, *cMax=None*, *nLevs=5*, *levels=None*)

Set colorbar levels given a number of levels and min/max values.

pygimli.viewer.mpl.**setMappableData**(*mappable*, *dataIn*, *cMin=None*, *cMax=None*, *logScale=None*, ***kwargs*)

Change the data values for a given mappable.

pygimli.viewer.mpl.**setOutputStyle**(*dim='w'*, *paperMargin=5*, *xScale=1.0*, *yScale=1.0*, *fontsize=9*, *scale=1*, *usetex=True*)

Set preferred output style.

pygimli.viewer.mpl.**setPlotStuff**(*fontsize=7*, *dpi=None*)

TODO merge with setOutputStyle.

Change ugly name.

pygimli.viewer.mpl.**show1dmodel**(*x*, *thk=None*, *xlab=None*, *zlab='z in m'*, *islog=True*, *z0=0*, ***kwargs*)

Show 1d block model defined by value and thickness vectors.

pygimli.viewer.mpl.**showDataContainerAsMatrix**(*data*, *x=None*, *y=None*, *v=None*, ***kwargs*)

Plot data container as matrix (cross-plot).

for each x, y and v token strings or vectors should be given

Examples using `pygimli.viewer.mpl.showDataContainerAsMatrix`

- *The DataContainer class* (page 180)

pygimli.viewer.mpl.**showDataMatrix**(*mat*, *xmap=None*, *ymap=None*, ***kwargs*)

Show value map as matrix.

Returns

- **ax** (*matplotlib axes object*) – axes object
- **cb** (*matplotlib colorbar*) – colorbar object

pygimli.viewer.mpl.**showStitchedModels**(*models*, *ax=None*, *x=None*, *cMin=None*, *cMax=None*, *thk=None*, *logScale=True*, *title=None*, *zMin=0*, *zMax=0*, *zLog=False*, ***kwargs*)

Show several 1d block models as (stitched) section.

Parameters

- **model** (*iterable of iterable (np.ndarray or list of np.array)*) – 1D models (consisting of thicknesses and values) to plot
- **ax** (*matplotlib axes [None - create new]*) – axes object to plot in
- **x** (*iterable*) – positions of individual models
- **cMin/cMax** (*float [None - autodetection from range]*) – minimum and maximum colorscale range
- **logScale** (*bool [True]*) – use logarithmic color scaling
- **zMin/zMax** (*float [0 - automatic]*) – range for z (y axis) limits
- **zLog** (*bool*) – use logarithmic z (y axis) instead of linear
- **topo** (*iterable*) – vector of elevation for shifting
- **thk** (*iterable*) – vector of layer thicknesses for all models

Returns

ax – axes object to plot in

Return type

matplotlib axes [None - create new]

pygimli.viewer.mpl.**showValMapPatches**(*vals*, *xVec=None*, *yVec=None*, *dx=1*, *dy=None*, ***kwargs*)

Show values as patches over x and y vector.

Parameters

- **vals** (*iterable*) – values to plot

- **xVec/yVec** (*iterable*) – x/y axis values
- **dx/dy** (*float*) – patch width
- **circular** (*bool*) – assume circular (cyclic) positions

pygimli.viewer.mpl.**showVecMatrix**(*xvec*, *yvec*, *vals*, *full=False*, ***kwargs*)

Plot three vectors as matrix.

Parameters

- **xvec** (*iterable (e.g. list, np.array, pg.Vector) of identical length*) – vectors defining the indices into the matrix
- **yvec** (*iterable (e.g. list, np.array, pg.Vector) of identical length*) – vectors defining the indices into the matrix
- **vals** (*iterable of same length as xvec/yvec*) – vector containing the values to show
- **full** (*bool [False]*) – use a fully symmetric matrix containing all unique xvec+yvec otherwise A is squeezed to the individual unique xvec/yvec values
- ****kwargs** (*forwarded to plotMatrix*) –
  - **ax**
    [mpl.axis] Axis to plot, if not given a new figure is created
  - **cMin/cMax**
    [float] Minimum/maximum color values
  - **logScale**
    [bool] Lgarithmic colour scale [min(A)>0]
  - **label**
    [string] Colorbar label

Returns

- **ax** (*matplotlib axes object*) – axes object
- **cb** (*matplotlib colorbar*) – colorbar object

pygimli.viewer.mpl.**showfdemsounding**(*freq*, *inphase*, *quadrat*, *response=None*, *npl=2*)

Show FDEM sounding as real(inphase) and imaginary (quadrature) fields.

Show FDEM sounding as real(inphase) and imaginary (quadrature) fields
normalized by the (purely real) free air solution.

pygimli.viewer.mpl.**showmymatrix**(*mat*, *x*, *y*, *dx=2*, *dy=1*, *xlab=None*, *ylab=None*, *cbar=None*)

What is this good for?.

pygimli.viewer.mpl.**twin**(*ax*)

Return the twin of ax if exist.

pygimli.viewer.mpl.**underlayBKGMap**(*ax*, *mode='DOP'*, *utmzone=32*, *epsg=0*, *imsize=2500*, *uuid=''*, *usetls=False*, *origin=None*)

Underlay digital orthophoto or topographic (mode='DTK') map under axes.

First accessed, the image is obtained from BKG, saved and later loaded.

Parameters

- **mode** (*str*) – 'DOP' (digital orthophoto 40cm) or 'DTK' (digital topo map 1:25000)
- **imsize** (*int*) – image width in pixels (height will be automatically determined

pygimli.viewer.mpl.**underlayMap**(*ax*, *proj*, *vendor='OSM'*, *zoom=-1*, *pixelLimit=None*, *verbose=False*, *fitMap=False*)

Get a map from public mapping server and underlay it on the given ax.

Parameters

- **ax** (*matplotlib.ax*) –
- **proj** (*pyproy*) – Proj Projection
- **vendor** (*str*) – . 'OSM' or 'Open Street Map' (tile.openstreetmap.org) . 'GM' or 'Google Maps' (mt.google.com)
- **zoom** (*int [-1]*) – Zoom level. If zoom is set to -1, the pixel size of the resulting image is lower than pixelLimit.
- **pixelLimit** (*[int, int]*) –
- **verbose** (*bool [false]*) – be verbose
- **fitMap** (*bool*) – The ax is resized to fit the whole map.

pygimli.viewer.mpl.**updateAxes**(*ax*, *force=True*)

For internal use.

pygimli.viewer.mpl.**updateColorBar**(*cbar*, *gci=None*, *cMin=None*, *cMax=None*, *cMap=None*, *logScale=None*, *nCols=256*, *nLevs=5*, *levels=None*, *label=None*, ***kwargs*)

Update colorbar values.

Update limits and label of a given colorbar.

Parameters

- **cbar** (*matplotlib colorbar*) –
- **gci** (*matplotlib graphical instance*) –
- **cMin** (*float*) –
- **cMax** (*float*) –
- **cLog** (*bool*) –
- **cMap** (*matplotlib colormap*) –
- **nCols** (*int [None]*) – Number of colors. If not set its number of levels.
- **nLevs** (*int*) – Number of color levels for the colorbar, can be different from the number of colors.
- **levels** (*iterable*) – Levels for the colorbar, overwrite nLevs.

- **label** (*str*) – Colorbar name.

pygimli.viewer.mpl.**updateFig**(*fig*, *force=True*, *sleep=0.001*)

For internal use.

pygimli.viewer.mpl.**wait**(***kwargs*)

TODO WRITEME.

8.8.1.3 Classes

class pygimli.viewer.mpl.**BoreHole**(*fname*)

Bases: object

Class for handling (structural) borehole data for inclusion in plots.

Each row in the data file must contain a start depth [m], end depth and a classification. The values should be separated by whitespace. The first line should contain the inline position (x and z), text ID and an offset for the text (for plotting).

The format is very simple, and a sample file could look like this:

16.0 0.0 BOREHOLEID_TEXTOFFSET n 0.0 1.6 clti n 1.6 10.0 shale

__init__(*fname*)

add_legend(*ax*, *cmap*, ***legend_kwargs*)

Add a legend to the plot.

plot(*ax*, *plot_thickness=1.0*, *cmin=None*, *cmax=None*, *cm=None*, *do_legend=True*, ***legend_kwargs*)

Plots the data on the specified axis.

class pygimli.viewer.mpl.**BoreHoles**(*fnames*)

Bases: object

Class to load and handle several boreholes belonging to one profile.

It makes the color coding consistent across boreholes.

__init__(*fnames*)

Load a list of bore hole from filenames.

add_legend(*ax*, *cmap*, ***legend_kwargs*)

Add a legend to the plot.

plot(*ax*, *plot_thickness=1.0*, *do_legend=True*, ***legend_kwargs*)

Plot the boreholes on the specified axis.

class pygimli.viewer.mpl.**CellBrowser**(*mesh*, *data=None*, *ax=None*)

Bases: object

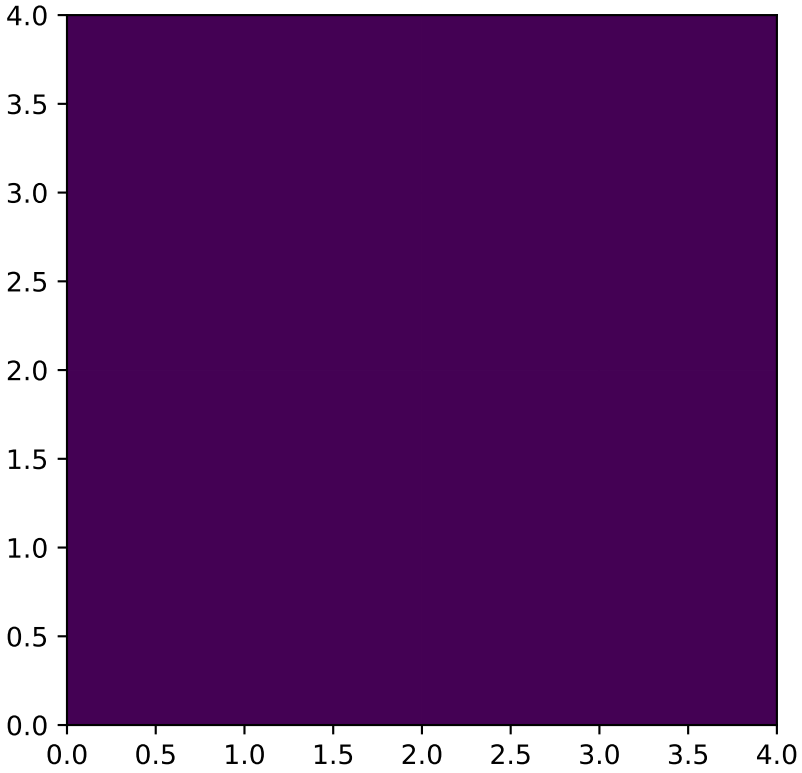
Interactive cell browser on current or specified ax for a given mesh.

Cell information can be displayed by mouse picking. Arrow keys up and down can be used to scroll through the cells, while ESC closes the cell information window.

Parameters

- **mesh** (GIMLI::Mesh) – The plotted 2D mesh to browse through.
- **data** (*iterable*) – Cell data.
- **ax** (*mpl axis instance, optional*) – Axis instance where the mesh is plotted (default is current axis).

```
>>> import matplotlib.pyplot as plt
>>> import pygimli as pg
>>> from pygimli.viewer.mpl import drawModel
>>> from pygimli.viewer.mpl import CellBrowser
>>>
>>> mesh = pg.createGrid(range(5), range(5))
>>> fig, ax = plt.subplots()
>>> plc = drawModel(ax, mesh, mesh.cellMarkers())
>>> browser = CellBrowser(mesh)
>>> browser.connect()
```



__init__(*mesh*, *data=None*, *ax=None*)

Construct CellBrowser on a specific *mesh*.

connect()

Connect to matplotlib figure canvas.

disconnect()

Disconnect from matplotlib figure canvas.

hide()

Hide info window.

highlightCell(*cell*)

Highlight selected cell.

initText()

onPick(*event*)

Call *self.update()* on mouse pick event.

onPress(*event*)

Call *self.update()* if up, down, or escape keys are pressed.

removeHighlightCell()

Remove cell highlights.

setData(*data=None*)

Set data, if not set look for the artist array data.

setMesh(*mesh*)

update()

Update the information window. Hide the information window for self.cellID == -1

8.8.2 pygimli.viewer.pv

Pyvista based drawing functions used by pygimli.viewer.

8.8.2.1 Overview

Functions

drawMesh (page 506)(ax, mesh[, notebook])	Draw a mesh into a given plotter.
drawModel (page 506)([ax, mesh, data])	Draw a mesh with given data.
drawSensors (page 506)(ax, sensors[, diam, color])	Draw the sensor positions to given mesh or the the one in given plotter.
drawSlice (page 507)(ax, mesh[, normal])	**param ax** The Plotter to draw on.
drawStreamLines (page 507)(ax, mesh, data[, label, radius])	Draw streamlines of given data.
pgMesh2pvMesh (page 508)(mesh[, data, label, boundaries])	pyGIMLi's mesh format is different from pyvista's needs, some preparation is necessary.
showMesh3D (page 508)(mesh, data, **kwargs)	Calling the defined function to show the 3D object.
toPVMesh (page 508)(mesh[, data, label, boundaries])	pyGIMLi's mesh format is different from pyvista's needs, some preparation is necessary.

8.8.2.2 Functions

pygimli.viewer.pv.**drawMesh**(*ax*, *mesh*, *notebook=False*, ***kwargs*)

Draw a mesh into a given plotter.

Parameters

- **ax** (*pyvista.Plotter [optional]*) – The plotter to draw everything. If none is given, one will be created.
- **mesh** (*pg.Mesh*) – The mesh to show.
- **notebook** (*bool [False]*) – Sets the plotter up for jupyter notebook/lab.
- **cMap** (*str ['viridis']*) – The colormap string.
- **bc** (*pyvista color ['#EEEEEE']*) – Background color.
- **style** (*str['surface']*) – Possible options:"surface","wireframe","points"
- **label** (*str*) – Data to be plottet. If None the first is taken.

Returns

ax – The plotter

Return type

pyvista.Plotter [optional]

Examples using pygimli.viewer.pv.drawMesh

- *3D magnetics modelling and inversion* (page 121)

pygimli.viewer.pv.**drawModel**(*ax=None*, *mesh=None*, *data=None*, ***kwargs*)

Draw a mesh with given data.

Parameters

- **ax** (*pyvista.Plotter [None]*) – Pyvista's basic Plotter to add the mesh to.
- **mesh** (*pg.Mesh*) – The Mesh to plot.
- **data** (*iterable*) – Data that should be displayed with the mesh.

Returns

ax – The plotter

Return type

pyvista.Plotter [optional]

pygimli.viewer.pv.**drawSensors**(*ax*, *sensors*, *diam=0.01*, *color='grey'*, ***kwargs*)

Draw the sensor positions to given mesh or the the one in given plotter.

Parameters

- **ax** (*pyvista.Plotter*) – The plotter to draw everything. If none is given, one will be created.
- **sensors** (*iterable*) – Array-like object containing tuple-like (x, y, z) positions.
- **diam** (*float [0.01]*) – Radius of sphere markers.

- **color** (*str ['grey']*) – Color of sphere markers.

Returns
ax – The plotter containing the mesh and drawn electrode positions.

Return type
pyvista.Plotter

Examples using `pygimli.viewer.pv.drawSensors`

- *Refraction in 3D* (page 57)

`pygimli.viewer.pv.`**drawSlice**(*ax*, *mesh*, *normal=[1, 0, 0]*, ***kwargs*)

Parameters

- **ax** (*pyvista.Plotter*) – The Plotter to draw on.
- **mesh** (*pg.Mesh*) – The mesh to take the slice out of.
- **normal** (*list [[1, 0, 0]]*) – Coordinates to orientate the slice.

Returns
ax – The plotter containing the mesh and drawn electrode positions.

Return type
pyvista.Plotter

Note: Possible kwargs are: normal: tuple(float), str origin: tuple(float) generate_triangles: bool, optional contour: bool, optional

They can be found at https://docs.pyvista.org/api/core/filters.html

---

Examples using `pygimli.viewer.pv.drawSlice`

- *3D Darcy flow* (page 136)

`pygimli.viewer.pv.`**drawStreamLines**(*ax*, *mesh*, *data*, *label=None*, *radius=0.01*, ***kwargs*)

Draw streamlines of given data.

PyVista streamline needs a vector field of gradient data per cell.

Parameters

- **ax** (*pyvista.Plotter [None]*) – The plotter that should be used for visualization.
- **mesh** (*pyvista.UnstructuredGrid|pg.Mesh [None]*) – Structure to plot the streamlines in to. If pv grid a check is performed if the data set is already contained.
- **data** (*iterable [None]*) – Values used for streamlining.
- **label** (*str*) – Label for the data set. Will be searched for within the data.
- **radius** (*float [0.01]*) – Radius for the streamline tubes.

Note: All kwargs will be forwarded to pyvistas streamline filter: https://docs.pyvista.org/api/core/_autosummary/pyvista.DataSetFilters.streamlines.html

---

Examples using `pygimli.viewer.pv.drawStreamLines`

- *3D Darcy flow* (page 136)

`pygimli.viewer.pv.`**`pgMesh2pvMesh`**(*mesh*, *data=None*, *label=None*, *boundaries=False*)

pyGIMLi's mesh format is different from pyvista's needs, some preparation is necessary.

Parameters

- **`mesh`** (*`pg.Mesh`*) – Structure generated by pyGIMLi to display.
- **`data`** (*`iterable`*) – Parameter to distribute to cells/nodes.

`pygimli.viewer.pv.`**`showMesh3D`**(*mesh*, *data*, ***kwargs*)

Calling the defined function to show the 3D object.

`pygimli.viewer.pv.`**`toPVMesh`**(*mesh*, *data=None*, *label=None*, *boundaries=False*)

pyGIMLi's mesh format is different from pyvista's needs, some preparation is necessary.

Parameters

- **`mesh`** (*`pg.Mesh`*) – Structure generated by pyGIMLi to display.
- **`data`** (*`iterable`*) – Parameter to distribute to cells/nodes.

8.8.3 Functions

`pygimli.viewer.`**`show`**(*obj=None*, *data=None*, ***kwargs*)

Mesh and model visualization.

Syntactic sugar to show a obj with data. Forwards to a known visualization for obj. Typical is *`pygimli.viewer.showMesh`* (page 509) or `pygimli.viewer.mayaview.showMesh3D` to show most of the typical 2D and 3D content. See tutorials and examples for usage hints. An empty show call creates an empty ax window.

Parameters

- **`obj`** (*`obj`*) – obj can be so far. * GIMLI::Mesh or list of meshes * DataContainer * pg.core.Sparse[Map]Matrix
- **`data`** (*`iterable`*) – Optionally data to visualize. See appropriate show function.

Keyword Arguments

`**kwargs` – Additional kwargs forward to appropriate show functions.

- **ax**
  [axe [None]] Matplotlib axes object. Create a new if necessary.
- **fitView**
  [bool [True]] Scale x and y limits to match the view.

Returns

- *Return the results from the showMesh* functions. Usually the axe object*
- *and a colorbar.*

See also:

`showMesh` (page 509)

pygimli.viewer.**showMesh**(*mesh*, *data=None*, *block=False*, *colorBar=None*, *label=None*, *coverage=None*, *ax=None*, *savefig=None*, *showMesh=False*, *showBoundary=None*, *markers=False*, ***kwargs*)

2D Mesh visualization.

Create an axis object and plot a 2D mesh with given node or cell data. Returns the axis and the color bar. The type of data determines the appropriate draw method.

Parameters

- **mesh** (GIMLI::Mesh) – 2D or 3D GIMLi mesh
- **data** (*iterable [None]*) – Optionally data to visualize.

  . None (draw mesh only)
  forward to *pygimli.viewer.mpl.drawMesh* (page 485) or if no cells are given: forward to *pygimli.viewer.mpl.drawPLC* (page 490)

  . [[marker, value], . . .]
  List of Cellvalues per cell marker forward to *pygimli.viewer.mpl.drawModel* (page 487)

  . float per cell – model, patch
  forward to *pygimli.viewer.mpl.drawModel* (page 487)

  . float per node – scalar field
  forward to *pygimli.viewer.mpl.drawField* (page 484)

  . iterable of type [float, float] – vector field
  forward to *pygimli.viewer.mpl.drawStreams* (page 495)

  . pg.core.R3Vector – vector field
  forward to *pygimli.viewer.mpl.drawStreams* (page 495)

  . pg.core.stdVectorRVector3 – sensor positions
  forward to *pygimli.viewer.mpl.drawSensors* (page 493)
- **block** (*bool [False]*) – Force to open the Figure of your content and blocks the script until you close the current figure. Same like pg.show(); pg.wait()
- **colorBar** (*bool [None], Colorbar*) – Create and show a colorbar. If colorBar is a valid colorbar then only its values will be updated.
- **label** (*str*) – Set colorbar label. If set colorbar is toggled to True. [None]
- **coverage** (*iterable [None]*) – Weight data by the given coverage array and fadeout the color.
- **ax** (*matplotlib.Axes [None]*) – Instead of creating a new and empty ax, just draw into the given one. Useful to combine multiple plots into one figure.
- **savefig** (*string*) – Filename for a direct save to disc. The matplotlib pdf-output is a little bit big so we try an epstopdf if the .eps suffix is found in savefig
- **showMesh** (*bool [False]*) – Shows the mesh itself additional.

- **showBoundary** (*bool [None]*) – Highlight all boundaries with marker != 0. None means automatic. True for cell data and False for node data.
- **marker** (*bool [False]*) – Show cell markers and boundary marker.
- **boundaryMarkers** (*bool [False]*) – Highlight boundaries with marker !=0 and add Marker annotation. Applies *pygimli.viewer.mpl.drawBoundaryMarkers* (page 483). Dictionary "boundaryProps" can be added and will be forwarded to *pygimli.viewer.mpl.drawBoundaryMarkers* (page 483).

Keyword Arguments

- **xlabel** (*str [None]*) – Add label to the x axis
- **ylabel** (*str [None]*) – Add label to the y axis
- **fitView** (*bool*) – Fit the axes limits to the all content of the axes. Default True.
- **boundaryProps** (*dict*) – Arguments for plotboundar
- **hold** (*bool [pg.hold()]*) – Holds back the opening of the Figure. If set to True [default] nothing happens until you either force another show with hold=False or block=True or call pg.wait() or pg.plt.show(). If hold is set to False your script will open the figure and continue working. You can change global hold with pg.hold(bool).
- **functions** (*All remaining will be forwarded to the draw*) –
- **respectively.** (*and matplotlib methods,*) –

```
>>> import pygimli as pg
>>> import pygimli.meshtools as mt
>>> world = mt.createWorld(start=[-10, 0], end=[10, -10],
...                        layers=[-3, -7], worldMarker=False)
>>> mesh = mt.createMesh(world, quality=32, area=0.2, smooth=[1, 10])
>>> _ = pg.viewer.showMesh(mesh, markers=True)
```

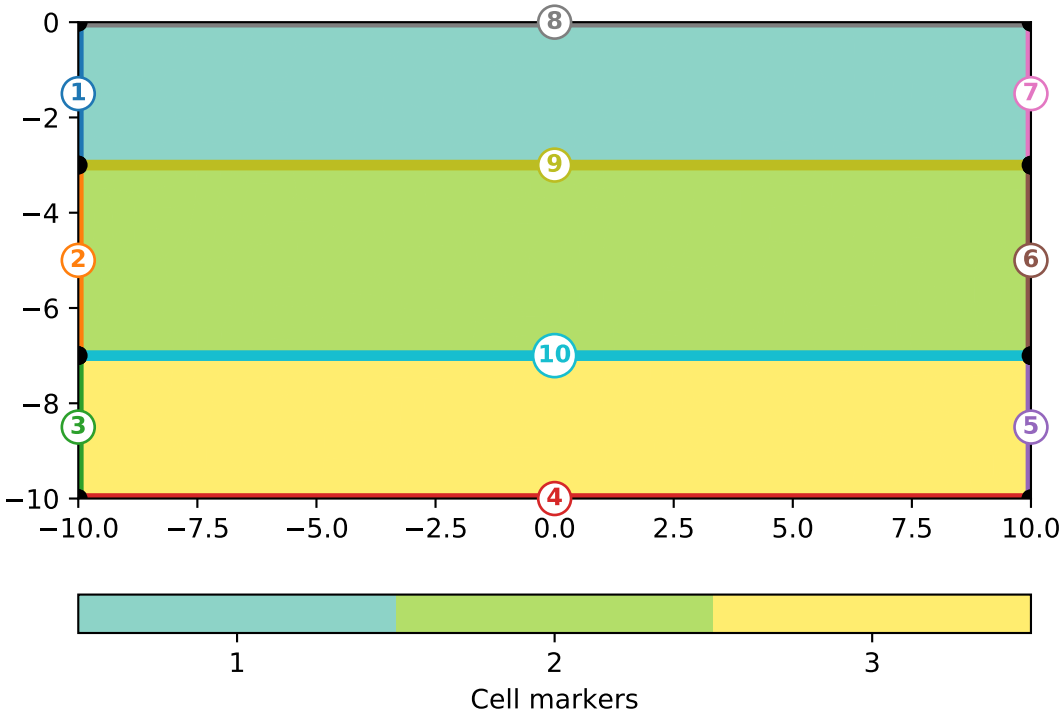
Returns

- **ax** (*matplotlib.axes*)
- **cBar** (*matplotlib.colorbar*)

Note: This documentation is valid for **|version|**. Check your installed version with

```
import pygimli as pg
print(pg.__version__)
```

and consider updating (e.g., via the conda package manager). If you have a newer version, the current documentation of the development version (*dev* branch) before the next release is available at: https://dev.pygimli.org.


0
−2
−4
−6
−8
−10
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
1
2
3
4
5
6
7
8
9
10
1
2
3
Cell markers

CHAPTER

NINE

CONTRIBUTING

As an open-source project, pyGIMLi always welcomes contributions from the community. Here, we offer guidance for 4 different ways of contributing with increasing levels of required coding proficiency (A-D).

9.1 A. Submit a bug report

If you experience issues with pyGIMLi or miss a certain feature, please open a new issue on GitHub. To do so, you need to create a GitHub account.

9.2 B. Send us your example

We are constantly looking for interesting usage examples of pyGIMLi. If you have used the package and would like to contribute your work to the *Examples* (page 17), you can send us your example. Make sure that your Python script is documented according to the sphinx-gallery syntax.

9.3 C. Quickly edit the documentation

If you would like to edit a typo or improve the documentation on our website, you can submit fixes to the documentation page by clicking on "Improve this page" under the quick links menu on every page in the right sidebar. This will direct you to the Github page to make the edits directly on the ".rst" using Github (you will need a Github account, see step A). Once you make your edits, fill out the fields under "Commit changes" (make sure to provide a good overview of the changes you have made). Note that you will make changes to a new branch under your fork since you will not have access to make changes in the existing branch. Once you click "Commit changes" this automatically creates a pull request which we will review and then merge, if everything is good to go!

If you would like to contribute to the API or make changes offline to the files under *doc*, see following *Contribute to Code* and *Documentation* for more information.

9.4 D. Contribute to code

Note: To avoid redundant work, please contact us or check the current issues before you start working on a non-trivial feature.

The preferred way to contribute to the pygimli code base is via a pull request (PR) on GitHub. The general concept is explained here and involves the following steps:

9.4.1 1. Fork and clone the repository

If you are a first-time contributor, you need a GitHub account and your own copy ("fork") of the code. To do so, go to https://github.com/gimli-org/gimli and click the "Fork button" in the upper right corner. This will create an identical copy of the complete code base under your username on the GitHub server. You can navigate to this repository and clone it to your local disk:

```
git clone https://github.com/YOUR_USERNAME/gimli
```

9.4.2 2. Prepare your environment

To install the neccesary Python requirements on your computer, we recommend the Anaconda Python distribution and the conda package manager. The cloned and forked repository contains an *environment.yml* file with the specification for all package requirements to build and test pyGIMLi. This file should exist in the directory where you have cloned the repository. In order to create a separate environment where you will work, run the following command in the same directory of the repository in your terminal:

```
conda env create # Creates the environment (only once)
conda activate pgdev # Activates it (everytime you want to work in it)
conda develop . # Installs your git-version of pygimli (only once)
python -c "import pygimli; pygimli.test()" # Make sure that everything␣
↪works (also after adding new code)
```

You will need to do the last step (`conda activate pgdev`) everytime you start a terminal or put it in your `.bashrc`. `(pgdev)` should appear before your terminal location.

9.4.3 3. Create a feature branch

Go to the source folder and create a feature branch to hold your changes. It is advisable to give it a sensible name such as `adaptive_meshes`.

```
cd gimli
git checkout -b adaptive_meshes
```

9.4.4 4. Start making your changes

Go nuts! Add and modify files and regularly commit your changes with meaningful commit messages. Remember that you are working in your own personal copy and in case you break something, you can always go back. While coding, we encourage you to follow a few sec:Coding_rules.

```
git add new_file1 new_file2 modified_file1
git commit -m "Implemented adaptive meshes."
```

9.4.5 5. Test your code

Make sure that everything works as expected. New functions should always contain a docstring with a test:

```
def sum(a, b):
    """Return the sum of `a` and `b`.

    Examples
    --------
    >>> a = 1
    >>> b = 2
    >>> sum(a,b)
    3
    """
    return a + b
```

When you run `pg.test()` the docstring test will be evaluated. See also the section on sec:testing.

9.4.6 6. Documentation

We use sphinx to build the web pages from these sources. To edit HTML file, navigate to the doc folder in the repository. Once you have made edits. Run the following commands on your terminal.

```
make html
```

The process should build all of the documentation so it may take some time to complete. When it finishes, the last line should state:

```
Build finished. The HTML pages are in _build/html.
```

9.4.7 7. Submit a pull request

Once you implemented a functioning new feature, make sure your GitHub repository contains all your commits:

```
git push origin adaptive_meshes
```

After pushing, you can go to GitHub and you will see a green PR button. Describe your changes in more detail. Once reviewed by the core developers, your PR will be merged to the main repository.

Note: Please see https://pygimli.org/dev for additional notes on coding rules, testing, etc.

CHAPTER

TEN

GLOSSARY

GCC
: The GNU Compiler Collection (GCC). Default compiler for most linux systems. The Windows port is *MinGW* See: http://gcc.gnu.org/

GIMLi
: The eponymous software package. See *About pyGIMLi* (page 1)

pyGIMLi
: The eponymous software package (python bindings). See *About pyGIMLi* (page 1)

libGIMLi
: The eponymous software package (c++ core). See *About pyGIMLi* (page 1)

FEniCS
: The FEniCSx computing platform. https://fenicsproject.org/

Gmsh
: Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities [**?**] http://gmsh.info/. See: `pygimli.meshtools.mesh.readGmsh()`

BERT
: Boundless electrical resistivity tomography http://www.resistivity.net/

IPython
: An improved *Python* shell that integrates nicely with *Matplotlib*. See http://ipython.org/.

Matplotlib
: Matplotlib *Python* package displays publication quality results. It displays both 1D X-Y type plots and 2D contour plots for structured and unstructured data. It works on all common platforms and produces publication quality hard copies. http://matplotlib.org

MinGW
: MinGW, a contraction of "Minimalist GNU for Windows", is a minimalist development environment for native Microsoft Windows applications. See: http://www.mingw.org/

MSYS
: MSYS, a contraction of "Minimal SYStem", is a Bourne Shell command line interpreter system. Offered as an alternative to Microsoft's cmd.exe, this provides a general purpose command line environment, which is particularly suited to use with MinGW, for porting of many Open Source applications to the MS-Windows platform. See: http://www.mingw.org/

NumPy
: NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object and various functionalities related to it. https://docs.scipy.org/doc/numpy/

Paraview
: Is an open-source, multi-platform data analysis and visualization application. See: http://paraview.org/

Python
: The programming language that *pyGIMLi* (and your scripts) are written in. See: https://www.python.org/

Pylab
: Meta-package importing several core packages such as *NumPy*, *SciPy*, *Matplotlib*, etc. into a single namespace. This is usually not recommended due to possible name conflicts but provides a quick way to get MATLAB-like functionality.

PyVista
: 3D visualization tool based on VTK: https://www.pyvista.org

SciPy
: Scientific Computing Tools for Python - Open-source library with many numerical routines but the term is often used as a synonym for the scientific python community, several conferences, and the *SciPy Stack*, i.e. a set of core packages. https://scipy.org/about.html

Sphinx
: The tools used to generate the *GIMLi* documentation. See: http://sphinx-doc.org

STL
: Unstructured triangulated surface file format native to the "stereolithography" CAD software created by 3D Systems. https://en.wikipedia.org/wiki/STL_%28file_format%29

SuiteSparse
: SuiteSparse is a single archive that contains packages for solving large sparse problems using Sparse Cholesky factorization. http://faculty.cse.tamu.edu/davis/suitesparse.html

Triangle
: A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator. [**?**] http://www.cs.cmu.edu/~quake/triangle.html See: `pygimli.meshtools.mesh.readTriangle()`

Tetgen
: A Quality Tetrahedral Mesh Generator and a 3D Delaunay Triangulator. [**?**] http://tetgen.org/ See: `pygimli.meshtools.mesh.readTetgen()`

BIBLIOGRAPHY

[CRucker15] C. Rücker, T. Günther, F. Wagner. PyGIMLi - Eine Open Source Python Bibliothek zur Inversion und Modellierung in der Geophysik. In *75. Jahrestagung der Deutschen Geophysikalischen Gesellschaft (DGG), Hannover 2015*. 2015.

[CRucker16] C. Rücker, T. Günther, F. Wagner. PyGIMLi - An Open Source Python Library for Inversion and Modelling in Geophysics. In *78th EAGE Conference and Exhibition 2016, WS08-Open Source Software in Applied Geosciences*. 2016. doi:10.3997/2214-4609.201601651.

[GuntherDHY10] Günther, T., Dlugosch, R., Holland, R., and Yaramanci, U. Aquifer characterization using coupled inversion of MRS & DC/IP data on a hydrogeophysical test-site. In *Ext. Abstract, 23. EEGS annual meeting (SAGEEP), April 11-14, 2010; Keystone, CO.*, volume 23, 302–307. 2010. doi:10.4133/1.3445447.

[GuntherRucker06] Günther, T. and Rücker, C. A new joint inversion approach applied to the combined tomography of dc resistivity and seismic refraction data. In *Ext. Abstract, 19. EEGS annual meeting (SAGEEP), 02.-06.04.2006; Seattle, USA*. 2006. doi:10.4133/1.2923578.

[Rucker11] Rücker, C. *Advanced Electrical Resistivity Modelling and Inversion using Unstructured Discretization*. PhD thesis, University of Leipzig, 2011. URL: https://nbn-resolving.de/urn:nbn:de:bsz:15-qucosa-69066.

[WRuckerGunther17] Wagner, F., Rücker, C., and Günther, T. Reproducible hydrogeophysical inversions through the open-source library pygimli. In *AGU Fall Meeting, New Orleans, 11-15 Dec 2017, Open-Source Software in the Geosciences (NS41B-0016)*. 2017. invited. URL: https://agu17.pygimli.org, doi:10.5281/zenodo.1095621.

[WRuckerGunther+20] Wagner, F., Rücker, C., Günther, T., Dinsel, F., Skibbe, N., Weigand, M., and Hase, J. Open-source hydrogeophysical modeling and inversion with pygimli 1.1: recent advances and examples in research and education. In *EGU General Assembly 2020, Online Meeting*. 2020. invited. doi:10.5194/egusphere-egu2020-18751.

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

N

O

P

Q

R

S

T

U

V

W

Z