tutorials Documentation

Release 0.0.1

SimPEG Developers

Jan 04, 2017

Contents

A set of tutorials for using SimPEG! SimPEG is a framework and toolbox for simulation and parameter estimation in geophysics.

Contents:

CHAPTER 1

Getting Started: for Developers

- **Purpose:** To download and set up your environment for using and developing within SimPEG.

1.1 Installing Python

SimPEG runs is written in Python! To install and maintain your Python 2.7 environment, Anaconda is a package manager that you can use. SimPEG requires Python 2.7. If you and Python are not yet acquainted, we highly recommend checking out Software Carpentry.

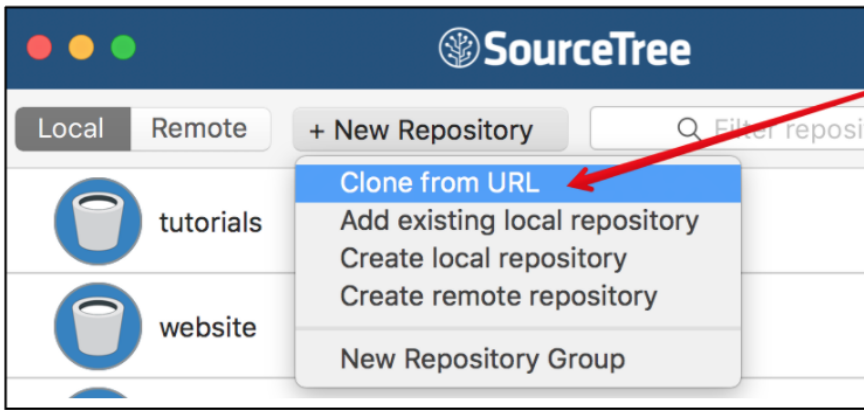
1.2 Working with Git and GitHub

To keep track of your code changes and contribute back to SimPEG, you will need a github account and fork the SimPEG repository (How to fork a repo). Software

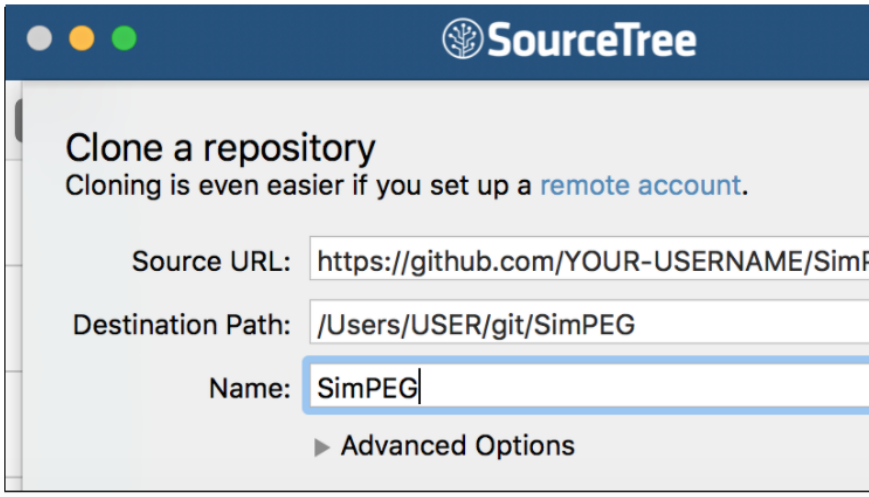
Next, clone your fork so that you have a local copy. We recommend setting up a directory called `git` in your home directory to put your version- controlled repositories. There are two ways you can clone a repository: (1) from a terminal (checkout: https://try.github.io for an tutorial):

```
git clone https://github.com/YOUR-USERNAME/SimPEG
```

1



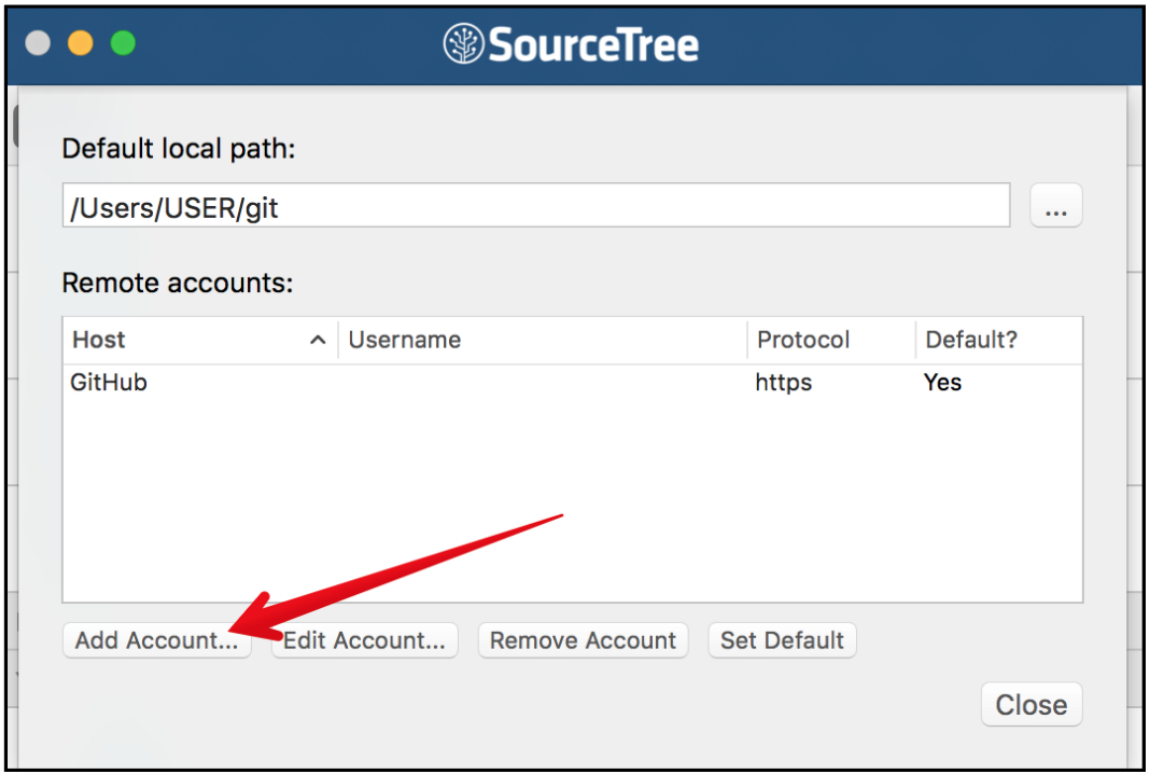
2



or (2) using a desktop client such as SourceTree.
If this is your first time managing a github repository through SourceTree, it is also handy to set up the remote account so it remembers your github user name and password

1

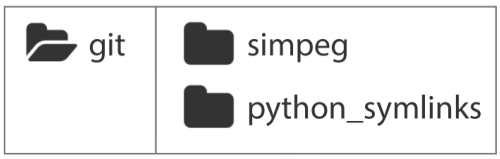
2



For managing your copy of SimPEG and contributing back to the main repository, have a look at the article: A successful git branching model

1.3 Setting up your environment

So that you can access SimPEG from anywhere on your computer, you need to add it to your path. This can be done using symlinks. In your `git` directory, create a directory called `python_symlinks`.



Open a terminal in this directory and create a symlink for SimPEG

```
ln -s ../SimPEG/SimPEG .
```

Then, in your shell, you need to add a PYTHONPATH variable. For Mac and Linux, if you are using Z shell (Oh My Zsh is used by a lot of SimPEG developers) or bash open the config in a text editor, ie:

```
nano ~/.zshrc
```

or:

```
nano ~/.bash_profile
```

and add a PYTHONPATH variable:

```
export PYTHONPATH="$PYTHONPATH:/Users/USER/git/python_symlinks"
```

and save and close. If you then restart the terminal, and run:

```
echo $PYTHONPATH
```

the output should be:

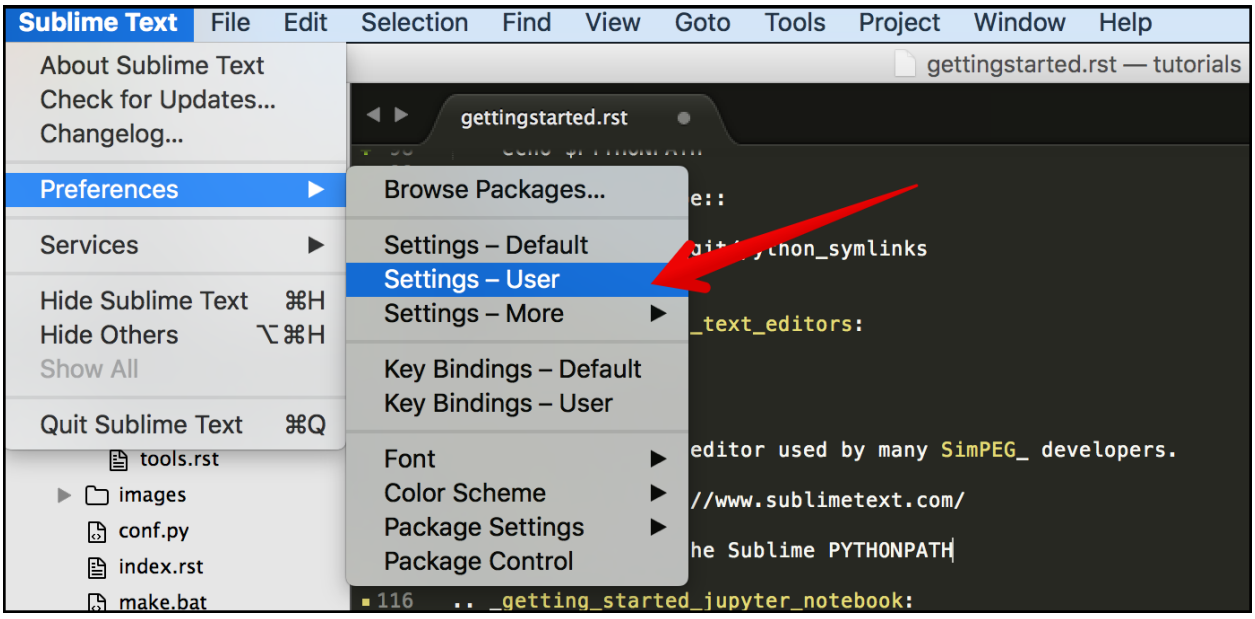
```
/Users/USER/git/python_symlinks
```

1.4 Text Editors

Sublime is a text editor used by many SimPEG developers.

You can configure the Sublime so that you can use the sublime build (Tools / Build) to run Python code.

Open your user settings



and edit them to include the path to your python_symlinks:

```
{
"added_words":
[
    "electromagnetics"
],
"ensure_newline_at_eof_on_save": true,
"extra_paths":
[
    "/Users/USER/git/python_symlinks/"
],
"font_size": 11,
"ignored_packages":
[
    "Vintage"
],
"translate_tabs_to_spaces": true,
"trim_trailing_white_space_on_save": true,
"word_wrap": false
}
```

There are a few other things configured here. In particular you will want to ensure that `"translate_tabs_to_spaces": true` is configured (Python is sensitive to tabs and spaces), that `"trim_trailing_white_space_on_save": true` so that your git flow does not get cluttered with extra spaces that are not actually changes to code and that `"ensure_newline_at_eof_on_save": true`, so that there is a blank line at the end of all saved documents. The rest are up to you.

1.5 Jupyter Notebook

The SimPEG team loves the Jupyter notebook. It is an interactive development environment. It is installed it you used Anaconda and can be launched from a terminal using:

```
jupyter notebook
```

1.6 If all is well ...

You should be able to open a terminal within SimPEG/SimPEG/Examples and run an example, ie.:

```
python Inversion_Linear.py
```

and open a Jupyter Notebook, and run the linear inversion

```
from SimPEG.Examples import Inversion_Linear
%matplotlib inline
```

```
Inversion_Linear.run()
```

and see

You are now set up to SimPEG!

Note: you likely got a message that said:

```
Efficiency Warning: Interpolation will be slow, use setup.py!
    python setup.py build_ext --inplace
```

This is because we use Cython to speed up interpolation. To set this up open up a command prompt in `git/simpeg`, and run:

```
python setup.py build_ext --inplace
```

Which might output a bunch of warnings, but so long as there are no errors, you should be good to go. To check, re-run the example and see if the efficiency warning still appears.

1.7 If all is not well ...

Submit an issue and change this file!

1.8 Advanced: Installing Mumps

Mumps is a direct solver that can be used for solving large(ish) [1] linear systems of equations. To install, follow the instructions to download and install pymatsolver.

- **Disclaimer for Windows users**: we have not figured out a stable way to install and connect Mumps for Windows Machines. If you have one, please change this file!

If you open a Jupyter notebook and are able to run:

```
from pymatsolver import MumpsSolver
```

then you have succeeded! Otherwise, make an issue in pymatsolver.

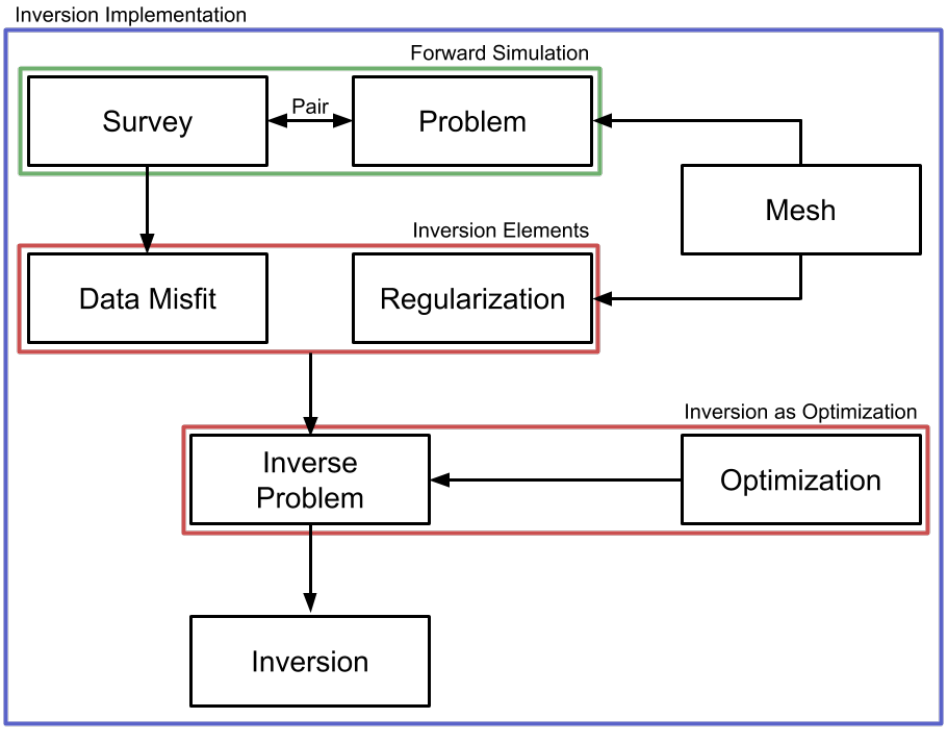
[1] These instructions are for serial (not parallel) Mumps installation. The definition of large also depends on the size of your computer

CHAPTER 2

Stories

Purpose

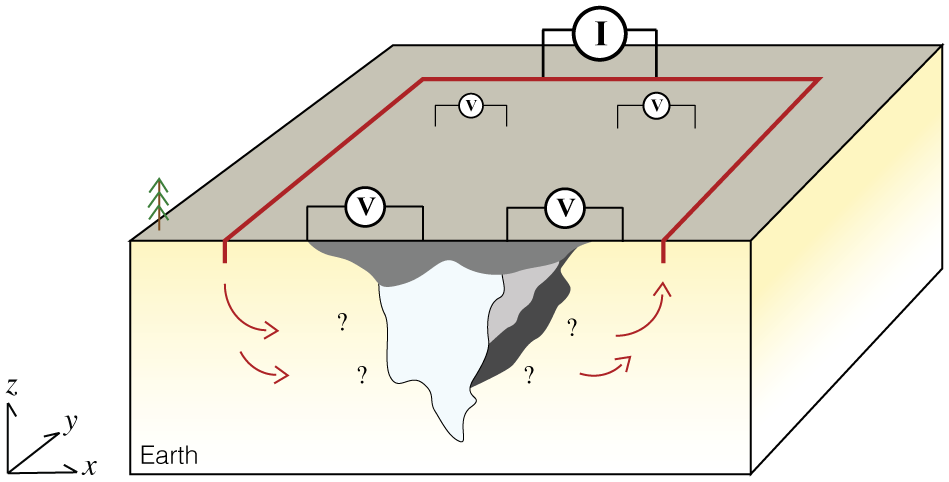
Using examples, we walk through how to use the SimPEG framework and implementation to solve geophysical forward and inverse problems



2.1 DC resistivity

Purpose

Understand basic setup and physics of a direct current (DC) resistivity survey within the context of a kimberlite exploration. Run DC forward modelling and inversion using SimPEG-Static package.



2.1.1 Set-up

The physical behavior of DC resistivity survey is governed by the steady-state Maxwell's equations:

$$\vec{j} = \sigma \vec{e}$$

$$\vec{e} = -\nabla \phi$$

$$\nabla \cdot \vec{j} = -\nabla \cdot \vec{j}_s = I_0(\delta(\vec{r} - \vec{r}_+) - \delta(\vec{r} - \vec{r}_-))$$

$$\vec{j} \cdot \hat{n} \Big|_{\partial\Omega} = 0$$

where: - $\vec{j}$: Current density (A/m ^{2})

- $\vec{e}$: Electric field (V/m)
- I_0: Current (A)
- δ: Volumetric delta function (m $^{-3}$)

Consider a simple gradient array having a pair of A (+) and B (+) current electrodes (Tx) with multiple M (+) and N (-) potential electrodes (Rx). Using giant battery (?), we setup a significant potential difference allowing electrical currents to flow between the A to B electrodes. If the earth includes conductors or resistors, these will distorts current flow, and measured potential differences on the surface electrodes (MN) will be reflective of those distortions. Typically kimberlitic pipes (including those containing diamonds!) will be more conductive than the background rock (granitic),

hence, the measured potential difference will be low. That is, contrasts in electrical conductivity between different rocks induce anomalous voltages. From the observed voltages, we want to estimate conductivity distribution of the earth. We use a geophysical inversion technique to do this procedure.

We work through each step of geophysical inversion using SimPEG-Static package under SimPEG's framework having two main items: a) Forward simulation and b) Inversion.

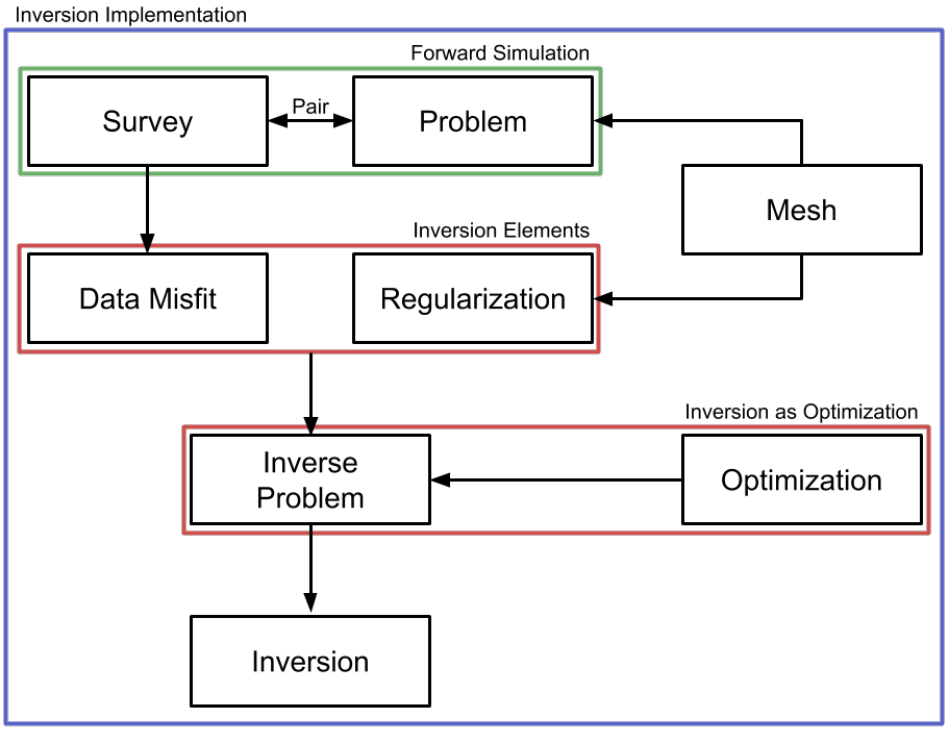


Fig. 2.1: SimPEG's framework

2.1.2 Forward simulation

A forward simulation of a DC experiment requires Survey and Problem classes. We need to the pass current and potential electrode locations to a DC survey class. The physical behavior of DC fields and fluxes are governed by the static Maxwell's equations. To numerically work with these equations, we use the DC problem class, which handles this by solving a corresponding partial different equation in a discrete space. For this, the earth needs to be discretized to solve corresponding partial differential equation. The Problem class computes fields in full discretized domain, and the Survey class evaluates data at potential electrodes using the fields. The Survey and Problem classes need to share information hence, we pair them.

Mesh

We use a 3D tensor mesh to discretize the earth having 25x25x25 m core cell size. Smaller vertical size of the cell (dz) is used close to the topographic surface (12.5 m), and padding cells are used to satisfies the natural boundary condition imposed.

Survey

We use a simple gradient array having a pair of current electrodes (AB), and multiple potential electrodes (MN). The lengths of AB and MN electrodes are 1200 and 25 m, respectively.

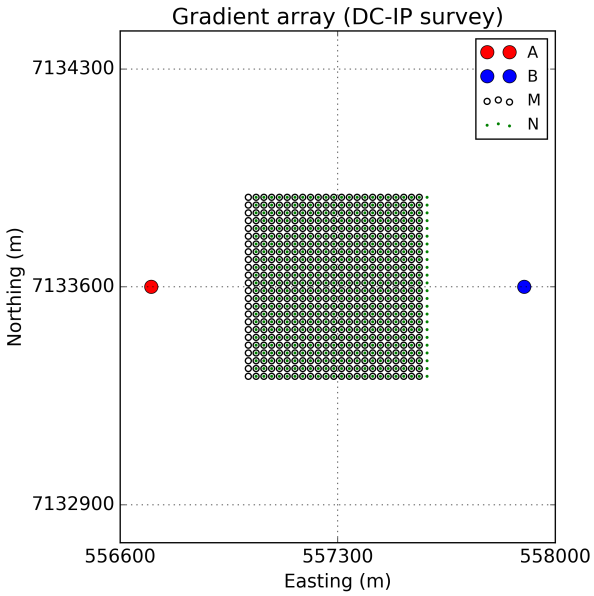


Fig. 2.2: Gradient array

Once we have obtained locations of AB (Src) and MN (Rx) electrodes, we can generate **Survey** class:

```
from SimPEG.EM.Static import DC

# Create Src and Rx classes for DC problem
Aloc1_x = np.r_[-600., 0, 0.] + np.r_[xc, yc, zc]
Bloc1_x = np.r_[600., 0, 0.] + np.r_[xc, yc, zc]

# Rx locations (M-N electrodes, x-direction)
x = mesh.vectorCCx[np.logical_and(mesh.vectorCCx>-300.+ xc, mesh.vectorCCx<300.+ xc)]
y = mesh.vectorCCy[np.logical_and(mesh.vectorCCy>-300.+ yc, mesh.vectorCCy<300.+ yc)]
# Grid selected cell centres to get M and N Rx electrode locations
Mx = Utils.ndgrid(x[:-1], y, np.r_[-12.5/2.])
Nx = Utils.ndgrid(x[1:], y, np.r_[-12.5/2.])

rx = DC.Rx.Dipole(Mx, Nx)
src = DC.Src.Dipole([rx], Aloc1_x, Bloc1_x)

# Form survey object using Srcs and Rxs that we have generated
survey = DC.Survey([src])
```

Fields and Data

By solving the DC equations, we compute electrical potential (ϕ) at every cell. The **Problem** class does this, but it still requires survey information hence we pair it to the **Survey** class:

```
# Define problem and set solver
problem = DC.Problem3D_CC(mesh)

problem.Solver = MumpsSolver
# Pair problem and survey
problem.pair(survey)
```

Here, we used `DC.Problem3D_CC`, which means 3D space and ϕ is defined at the cell center. Now, we are ready to run DC forward modelling! For this modelling, inside of the code, there are two steps:

1. Compute fields (ϕ at every cells)
2. Evaluate at Rx location (potential difference at MN electrodes)

Consider two conductivity models:

- Homogeneous background below topographic surface: `sigma0` (10^{-4} S/m)
- Includes diamond pipes: `sigma` (S/m)

```
# Read pre-generated conductivity model in UBC format
sigma = mesh.readModelUBC("VTKout_DC.dat")
# Identify air cells in the model
airind = sigma == 1e-8
# Generate background model (constant conductiivty below topography)
sigma0 = np.ones_like(sigma)*1e-4
sigma0[airind] = 1e-8
```

Then we compute fields for both conductivity models:

```
# Forward model fields due to the reference model and true model
f0 = problem.fields(sigma0)
f = problem.fields(sigma)
```

Now `f` and `f0` are **Field** objects including computed ϕ everywhere. However, this **Field** object know how to compute both $\vec{e}$, $\vec{j}$, and electrical charge, $\int_V \rho_v dV$ (ρ_v is volumetric charge density). Note that if we know ϕ, all of them can be computed for a corresponding source:

```
phi = f[src, 'phi']
e = f[src, 'e']
j = f[src, 'j']
charge = f[src, 'charge']
```

Since the field object for the background model is generic, we can obtain secondary potential:

```
# Secondary potential
phi0 = f0[src, 'phi']
phi_sec = phi - phi0
```

We present plan and section views of currents, charges, and secondary potentials in Fig. 2.3.

Current flows from A (+) to B (-) electrode (left to right). Kimberlite pipe should be more conductive than the background considering more currents are flowing through the pipe (See distortions of the current path in the left panel).

The distribution of electrical charges (the middle panel) supports that the pipe is conductive since left and right side of the pipe has negative and positive charges, respectively. In addition, charges only built on the boundary of the conductive pipe.

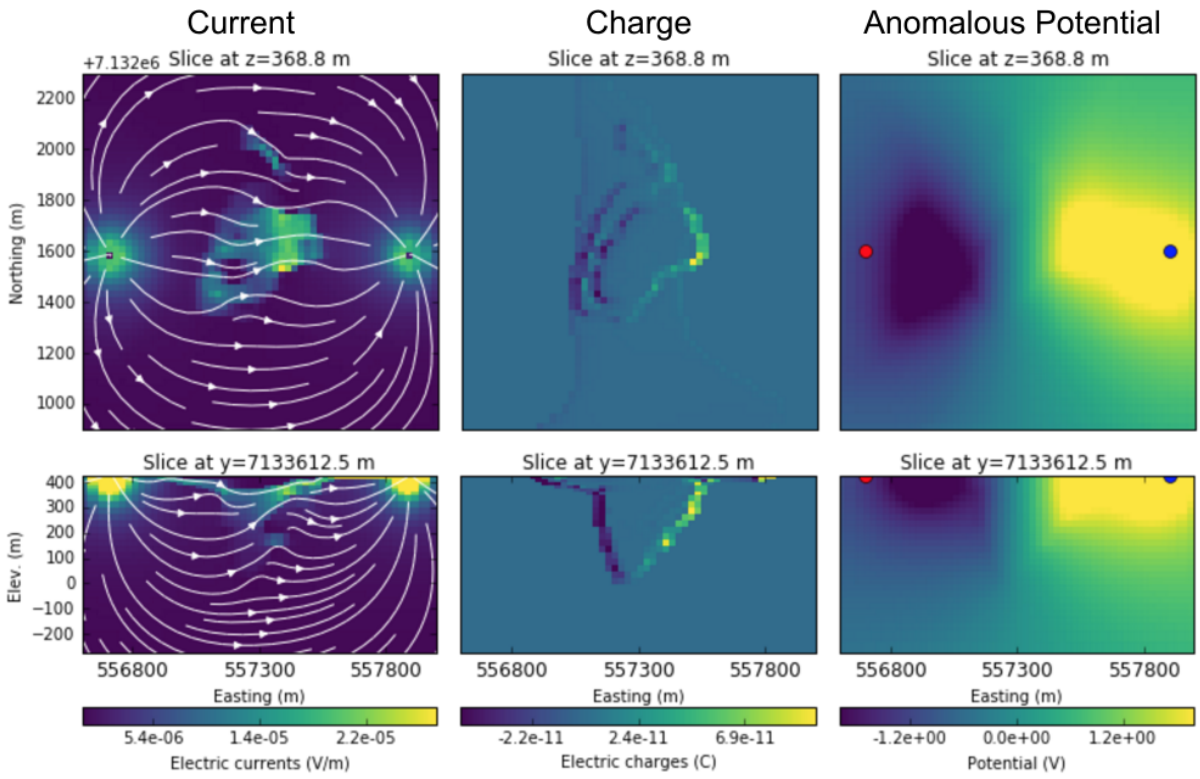


Fig. 2.3: DC fields. Left, middle, and right panels show currents, charges, and secondary potentials.

The secondary potential (the right panel) is important since it shows response from the kimberlite pipe, which often called "Anomalous potential". Usually, removing background response is a good way to see how much anomalous response could be obtained for the target.

On the other hand, we cannot measure those fields everywhere but measure potential differences at MN electrodes (Rx) hence we need to evaluate them from the fields:

```
# Get observed data
dobs = survey.dpred(sigma, f=f)
```

If the field has not been computed then we do:

```
# Get observed data
dobs = survey.dpred(sigma)
```

This will compute the field inside of the code then evaluate for data at Rx locations. Below image shows the computed DC data. Smaller potentials are obtained at the center locations, which implies the existence of conductive materials. Current easily flows with conductive materials, which means less potential is required to path through them, hence for resistive materials we get greater potential difference measured on the surface. The measured potential provides some idea of the earth; however, this is not enough, we want a 3D distribution of the conductivity!

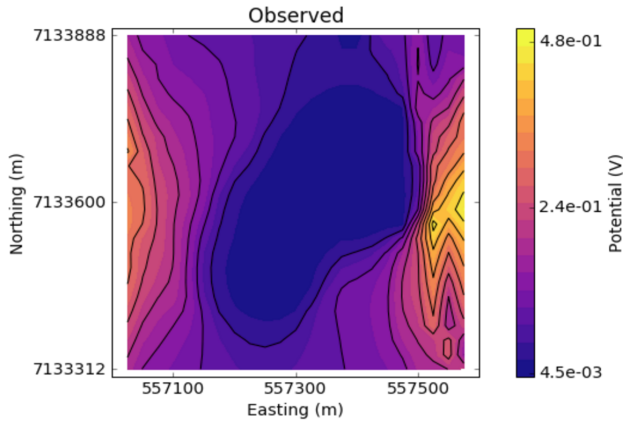


Fig. 2.4: DC data.

2.1.3 Inversion Elements

Our goal here is finding a 3D conductivity model which explains the observed data shown in Fig. 2.4. Inversion elements (red box in Fig. 2.1) will handle this task with an ability to simulate forward problem. We go through each element and briefly explain.

Mapping

For the simulation, we used a 3D conductivity model, with a value defined in every cell center location. However, for the inversion, we may not want to estimate conductivity at every cell. For instance, our domain include some air cells, and we already know well about the conductivity of the air ($10^{-8} \approx 0$) hence, those air cell should be excluded from the inversion model, m. Accordingly, a mapping is required moving from the inversion model to conductivity model defined at whole discrete domain:

$$\sigma = \mathcal{M}(m)$$

In addition, conductivity is strictly positive and varies logarithmically, so we often use log conductivity as our inversion model ($m = log(\sigma)$). Our inversion model is log conductivity only defined below the subsurface cells, and this can be expressed as

$$\sigma = \mathcal{M}_{exp}\Big(\mathcal{M}_{act}(m)\Big),$$

where $\mathcal{M}_{act}(\cdot)$ is a `InjectActiveCells` map, which takes subsurface cell and surject to full domain including air cells, and $\mathcal{M}_{exp}(\cdot)$ is an `ExpMap` map takes log conductivity to conductivity. Combination of two maps are required to get σ from m, which can be codified as

```
# from log conductivity to conductivity
expmap = Maps.ExpMap(mesh)
# from subsurface cells to full 3D cells
actmap = Maps.InjectActiveCells(mesh, ~airind, np.log(1e-8))
mapping = expmap*actmap
```

Generated mapping should be passed to **Problem** class:

```
# Generate problem with mapping
problem = DC.Problem3D_CC(mesh, sigmaMap=mapping)
```

Data Misfit

Finding a model explaining the observed data requires a measure between observed ($\mathbf{d}^{obs}$) and predicted data ($\mathbf{d}^{dpred}$):

$$\phi_d = 0.5\|\mathbf{W}_d(\mathbf{d}^{pred} - \mathbf{d}^{obs})\|_2^2,$$

where $\mathbf{W}_d = \mathbf{diag}(\frac{1}{\%|\mathbf{d}^{obs}|+\epsilon})$ is the data weighting matrix. Uncertainty in the observed data is approximated as $\%|\mathbf{d}^{obs}| + \epsilon$.

```
# percentage and floor for uncertainty in the observed data
std, eps = 0.05, 1e-3
survey.std = std
survey.eps = eps
survey.dobs = dobs

# Define datamisfit portion of objective function
dmisfit = DataMisfit.l2_DataMisfit(survey)
```

Regularization

The objective function includes both data misfit and regularization terms, ϕ_m :

$$\phi = \phi_d + \beta\phi_m$$

We use Tikhonov-style regularization including both smoothness and smallness terms. For further details of this See XXX.

In addition, considering the geometry of the gradient array: a single source and distributed receivers, this specific DC survey may not have much depth resolution similar to magnetic and gravity data. Depth weighting ($\frac{1}{(z-z_0)^3}$) is often used to handle this. And with this weight we form **Regularization** class:

```
# Depth weighting
depth = 1./(abs(mesh.gridCC[:,2]-zc))**1.5
depth = depth/depth.max()

# Define regulatization (model objective function)
reg = Regularization.Simple(mesh, mapping=regmap, indActive=~airind)
reg.wght = depth[~airind]
reg.alpha_s = 1e-1
reg.alpha_x = 1.
reg.alpha_y = 1.
reg.alpha_z = 1.
```

Optimization

To minimize the objective function, an optimization scheme is required. The **Optimization** class handles this, and we use Inexact Gauss Newton Scheme CITExxx.

```
opt = Optimization.InexactGaussNewton(maxIter = 20)
```

InvProblem

Both **DatamMisfit** and **Regularization** classes are created, and an **Optimization** is chosen. To pose the inverse problem, they need to be declared as an optimization problem:

$$\text{minimize } \phi_d + \beta\phi_m$$
$$s.t. \text{ some constratins}$$

The **InvProblem** class can be set with **DatamMisfit**, **Regularization** and **Optimiztion** classes.

```
invProb = InvProblem.BaseInvProblem(dmisfit, reg, opt)
```

Inversion

We have stated our inverse problem, but a conductor is required, who directs our inverse problem. **Directives** conducts our **Inversion**. For instance, the trade-off parameter, β needs to be estimated, and sometimes cooled in the inversion iterations. A target misfit is need to be set usually upon discrepancy principle ($\phi_d^* = 0.5N_d$, where N_d is the number of data).

```
# Define Directives
betaest = Directives.BetaEstimate_ByEig(beta0_ratio=1e0)
beta = Directives.BetaSchedule(coolingFactor=5, coolingRate=2)
target = Directives.TargetMisfit()

# Define Inversion class
inv = Inversion.BaseInversion(invProb, directiveList=[beta, betaest, target])
```

Run

Now we all set. The initial model is assumed to be homogeneous.

```
# Create inital and reference model (1e-4 S/m)
m0 = np.ones(mesh.nC)[~airind]*np.log(1e-4)

# Run inversion
mopt = inv.run(m0)
```

The inversion runs and reaches to the target misfit, hence we fit the observed data.

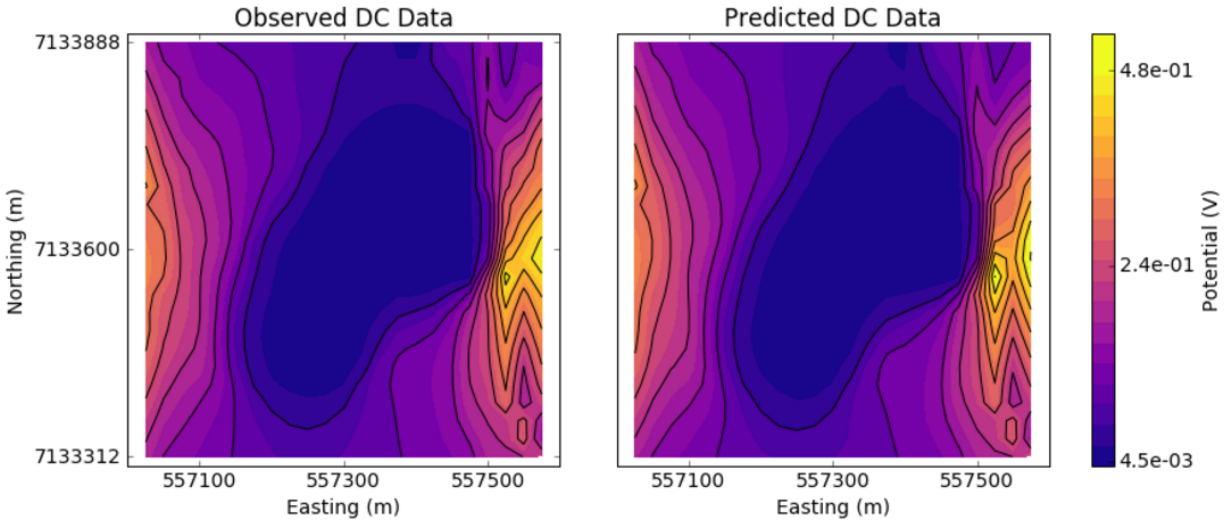


Fig. 2.5: Observed and Predicted DC data.

A 3D conductivity model is recovered and compared with the true conductivity model. A conductive pipe at depth is recovered!

2.2 Potential Fields (Magnetics)

Purpose

In this tutorial, we demonstrate how to invert magnetic field data in 3D using SimPEG.PF. It simulates a geophysical experiment over a synthetic kimberlite pipe model. The chosen geological model intends to replicate the Tli Kwi Cho kimberlite deposit, NWT. Using SimPEG Directives, we implement a sparse regularization function and recover both a smooth and a compact susceptibility model, which can used to infer geological information at depth. The same example was presented as a poster at Scipy 2016. There is also a notebook available for download here.

2.2.1 Setup

We begin this story with some physics background. We need to establish the connection between rocks magnetic properties and the associated geophysical experiment. Maxwell's equations for a static electric field and in the absence of free-currents can be written as:

$\nabla \cdot \mathbf{B} = 0$
$\nabla \times \mathbf{H} = 0$

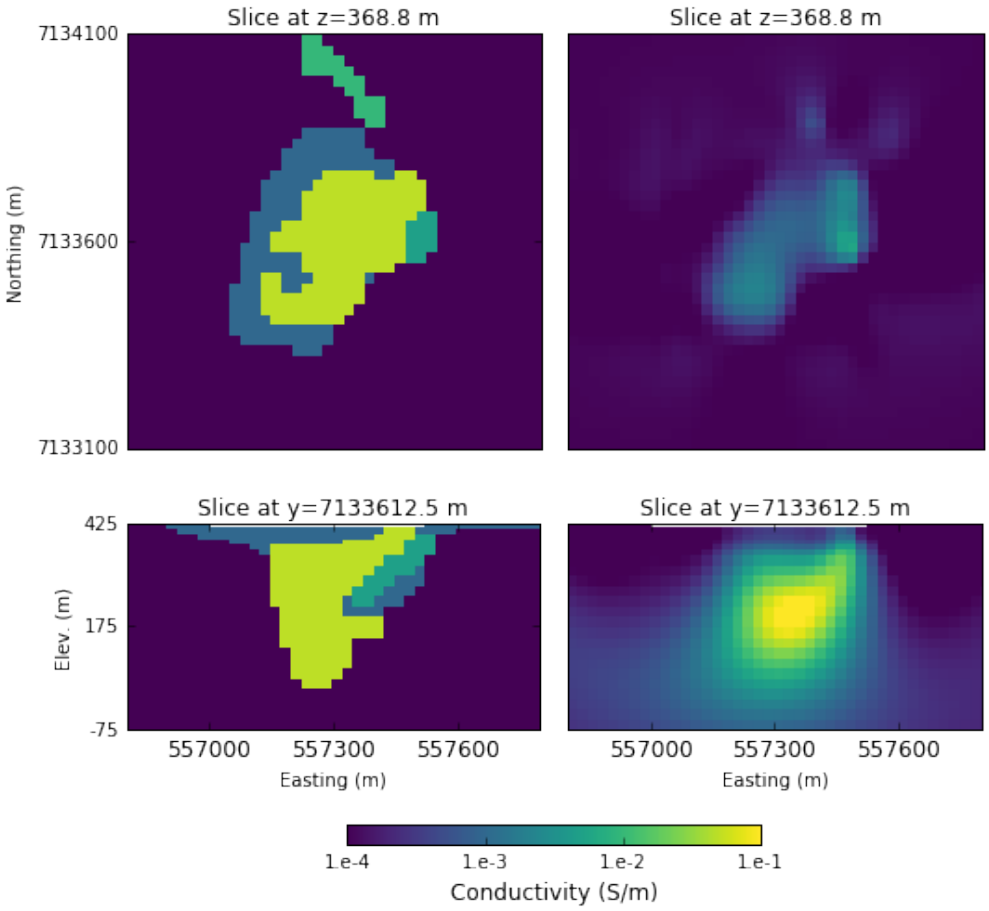
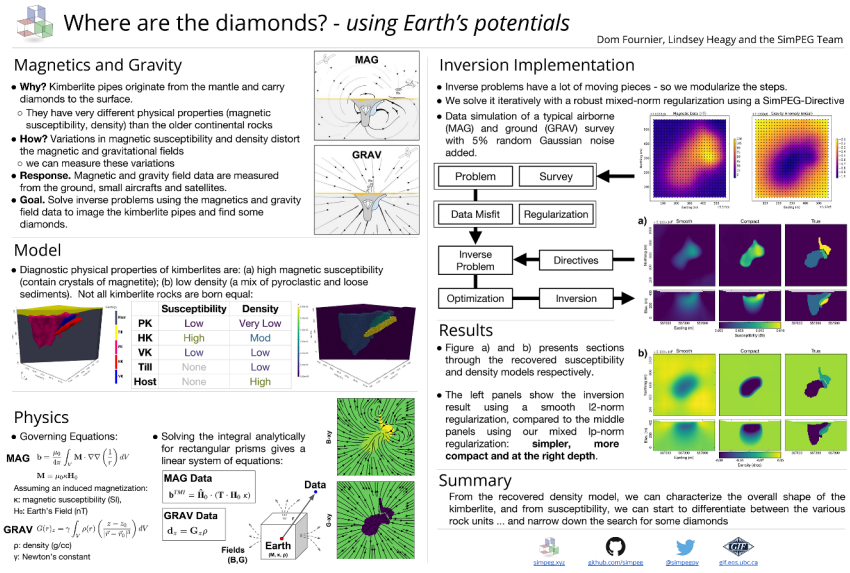


Fig. 2.6: True and recovered conductivity models.



where $\mathbf{B}$ and $\mathbf{H}$ correspond to the magnetic flux density and magnetic field respectively. Both quantities are related by:

$\mathbf{B} = \mu\mathbf{H}$
$\mu = \mu_0(1 + \kappa)$,

where μ is the magnetic permeability. In free-space, both $\mathbf{B}$ and $\mathbf{H}$ are linearly related by the magnetic permealitity of free-space μ_0. In matter however, the magnetic flux can be increased proportionally on how easily magnetic material gets polarized, quantified by the magnetic susceptibility κ. In a macroscopic point of view, the magnetic property of matter are generally described in terms of magnetization per unit volume such that:

$\mathbf{M} = \kappa\mathbf{H_s} + \mathbf{H_0} + \mathbf{M_r}$,

where $\mathbf{M}$ can be oriented in any specific direction due to secondary local fields ($\mathbf{H_s}$) and/or due to permanent dipole moments ($\mathbf{M_r}$). For simplicity we will here assume a purely induced response due to the Earth's $\mathbf{H_0}$. Using a few vector identities, we can re-write the magnetic field due to magnetized material in terms of a scalar potential:

$\phi = \frac{1}{4\pi}\int_V \nabla\left(\frac{1}{r}\right)\cdot\mathbf{H}_0\kappa\, dV$,

where r defines the relative position between an observer and the magnetic source. Taking the divergence of this potential yields:

$\mathbf{b} = \frac{\mu_0}{4\pi}\int_V \nabla\nabla\left(\frac{1}{r}\right)\cdot\mathbf{H}_0\kappa\, dV$.

Great, we have a general expression relating any secondary magnetic flux due to magnetic material

2.2.2 Forward Problem

Assuming a purely induced response, we can solve the integral analytically. As derived by Sharma (1966), the integral can be evaluated for rectangular prisms such that:

$\mathbf{b} = \mathbf{T}\cdot\mathbf{H}_0\ \kappa$.

Where the tensor matrix $\mathbf{T}$ relates the vector magnetization $\mathbf{M}$ inside a single cell to the components of the field $\mathbf{b}$ observed at a given location:

$$\mathbf{T} = \begin{pmatrix} T_{xx} & T_{xy} & T_{xz} \\ T_{yx} & T_{yy} & T_{yz} \\ T_{zx} & T_{zy} & T_{zz} \end{pmatrix} .$$

In general, we discretize the earth into a collection of cells, each contributing to the magnetic data such that giving rise to a large and dense linear system of the form:

$\mathbf{b} = \sum_{j=1}^{nc}\mathbf{T}_j\cdot\mathbf{H}_0\ \kappa_j$.

In most geophysical surveys, we are not collecting all three components, but rather the magnitude of the field, or *Total Magnetic Intensity* (TMI) data. Because the inducing field is really large, we will assume that the anomalous fields are parallel to H_0:

$d^{TMI} = \hat{\mathbf{H}}_0\cdot\mathbf{b}$.

We then end up with a much smaller system:

$d^{TMI} = \mathbf{F}\,\boldsymbol{\kappa}$,

where $\mathbf{F}\in\mathbb{R}^{nd\times nc}$ is our *forward* operator and κ is the physical property describing the Earth.

Getting started

In order to define a geophysical experiment we need set several important parameters, such as a mesh, data location, inversion parameters and so on. While we could set all of those parameters manually, SimPEG.PF gives the option to

work with an input file, capturing all the necessary information to run the inversion. In preparation for this synthetic example, we put together all necessary files and added them to a working directory. The input file can then be loaded and easily accessed through the Driver class:

```
>> import SimPEG
>> import SimPEG.PF as PF
>> from SimPEG.Utils.io_utils import remoteDownload
>>
>> # Start by downloading files from the remote repository
>> url = 'https://storage.googleapis.com/simpeg/tkc_synthetic/potential_fields/'
>> cloudfiles = ['MagData.obs', 'Mesh.msh',
>>               'Initm.sus', 'SimPEG_PF_Input.inp']
>> driver = PF.MagneticsDriver.MagneticsDriver_Inv(input_file)
>>
>> # Objects loaded from the input file are then accessible like this
>> mesh = driver.mesh
>> initm = driver.m0
Download files from URL...
Retrieving: MagData.obs
Retrieving: Mesh.msh
Retrieving: Initm.sus
Retrieving: SimPEG_PF_Input.inp
Download completed!
```

The input file looks like this:

Line	Input	Description
1	Mesh.msh	Mesh file*
2	Data.obs	Data file*
3	VALUE -100	Topography file* | null (all included)
4	FILE Initm.mod	Starting model* | VALUE ##
5	VALUE 0	Reference model* | VALUE ##
6	DEFAULT	Magnetization file* | DEFAULT
7	DEFAULT	Cell weight file* | DEFAULT
8	DEFAULT	Target Chi factor VALUE | DEFAULT (1)
9	DEFAULT	Scaling parameters for regularization $(\alpha_s, \alpha_x, \alpha_y, \alpha_z)$
10	VALUE 0 1	Lower and upper bound values
11	VALUE 0 1 1 1	Lp-norms applied on model and model gradients (p, q_x, q_y, q_z)
12	DEFAULT	Treshold parameter for the norms (ϵ_p, ϵ_q) | DEFAULT
Note		• UBC file format

We will use each elements later, but for now, this how the inversion is initiated.

Model and Mapping

Since we have already loaded the model in a rectangular mesh, we can plot it with SimPEG's built-in functions.

Notice that some of the cells in the model are air and show as white. The code will detected the air cells from the VALUE specified on line 3 of the input file. These cells are ignored by the code. Alternatively, the user can input a topography file or an active model specifying the status of each cells (0:inactive, 1:active-dynamic, -1:active-static).

Data

Great, now that we have a mesh and a model, we only need to specify a survey (i.e. where is the data). Once again, an observation file is provided, as specified on Line 2 of the input file. We can now forward model some magnetic data above the synthetic kimberlite.

Error: Unable to execute python code at PF_MAG.rst:228:

no display name and no $DISPLAY environment variable

2.2.3 Inverse Problem

We have generated synthetic data, we now what to see if we can solve the inverse problem. Using the usual formulation, we seek a model that can reproduce the data, let's say a least-squares measure of the form:

$\phi_d = \|\mathbf{W}_d \left(\mathbf{F}\ \mathbf{m} - \mathbf{d}^{obs}\right)\|_2^2\ ,$

where $\mathbf{W}_d$ are estimated data uncertainties The inverse problem is hard because we don't have great data coverage, and the Earth is big, and there is usually noise in the data. So we need to add something to regularize it. The simplest way to do it is to penalize solutions that won't make sense geologically, for example to assume that the model is *small* and *smooth*. Most inversion codes use the l2-norm measure such that:

$\phi_m = \|\mathbf{W}_s\ (\mathbf{m} - \mathbf{m^{ref}})\|_2^2 + \sum_{i=x,y,z} \|\mathbf{W}_i\ \mathbf{G}_i\ \mathbf{m}\|_2^2$

where m^{ref} is any a priori knowledge that we might have about the solution and $\mathbf{G}_x, \mathbf{G}_y, \mathbf{G}_z$ are finite difference operators measuring the model spatial gradients along orthogonal directions. In a purely *unconstrained* case, m^{ref} is usually equal to some background value (i.e. zero susceptibility). The full objective function to be minimized can be written as:

$\phi(m) = \phi_d + \beta\phi_m$

which will yield our usual function that minimize the data error and model structure. The trade-off parameter β is adjusted in order to get a *good* balance between data misfit and model

We propose a fancier regularization function that can allow to recover *sparse* and *blocky* solutions. Starting with the well known Ekblom norm:

$\phi_m = \sum_{i=1}^{nc} (x_i^2 + \epsilon^2)^{p/2}$

where x_i denotes some function of the model parameter, and ϵ is a small value to avoid singularity as $m \to 0$.

For p=2, we get the usual least-squares measure and we recover the regularization presented above. For $p \leq 1$, the function becomes non-linear which requires some tweaking. We can linearize the function by updating the penality function iteratively, commonly known as an Iterative Re- weighted Least-Squares (IRLS) method. The regularization function becomes:

$\phi_m^{(k)} = \frac{1}{2} \sum_{i=1}^{nc} r_i\ x_i^2$

where we added the superscript $\square^{(k)}$ to denote the IRLS iterations. The weights $r(x)$ are computed from model values obtained at a previous iteration such that:

$r_i = \left({x_i}^{(k-1)} \right)^2 + \epsilon^2 \Big)^{p/2-1}$

where $r(x) \in \mathbb{R}^{nc}$.

In matrix form, our objective function simply becomes:

$\phi(m) = \|\mathbf{W}_d \left(\mathbf{F}\ \mathbf{m} - \mathbf{d}^{obs}\right)\|_2^2 + \beta \Big[\|\mathbf{W}_s\ \mathbf{R}_s\ (\mathbf{m} - \mathbf{m^{ref}})\|_2^2 + \sum_{i=x,y,z} \|\mathbf{W}_i\ \mathbf{R}_i\ \mathbf{G}_i\ \mathbf{m}\|_2^2 \Big]$

where the IRLS weights $\mathbf{R}_s$ and $\mathbf{R}_i$ are diagonal matrices defined as:

$$R_{s_{jj}} = \sqrt{\eta_p}\Big[{m_j}^{(k-1)}\Big)^2 + \epsilon_p^2\Big]^{(p/2-1)/2}$$

$$R_{i_{jj}} = \sqrt{\eta_q}\Big[\Big((G_i\ m^{(k-1)})_j\Big)^2 + \epsilon_q^2\Big]^{(q/2-1)/2}$$

$$\eta_p = {\epsilon_p}^{(1-p/2)}\ ,\ \eta_q = {\epsilon_q}^{(1-q/2)}$$

we added two scaling parameters η_p and η_q for reasons that we won't dicuss here, but turn out to be important to get stable solves.

In order to initialize the IRLS and get an estimate for the stabilizing parameters ϵ_p and ϵ_q, we first invert with the smooth l_2-norm. Once the target data misfit has been achieved, the inversion switches to the sparse regularization. This way we get a good starting point, hopefully close enough to the true solution. The whole IRLS process is implemented with a directive added to the inversion workflow.

Error: Unable to execute python code at PF_MAG.rst:380:

Now we can plot sections and compare the smooth and compact models with the true solution.

Summary

We have inverted magnetic field data over a synthetic kimberlite pipe, using both a smooth and compact penalty. The smooth model gives a conservative and robust estimate of the kimberlite pipe location, as well as providing an excellent starting point for the sparse regularization. The compact model on the other hand gives a much closer estimate of susceptibility values and shape of the magnetic anomaly. More details about the scaled IRLS method can be found in this thesis.

CHAPTER 3

Tools

- **Purpose:** To introduce and provide resources for the tools used to build and work with SimPEG

SimPEG is implemented in Python 2.7, a high-level, object oriented language and has core dependencies on three packages standard to scientific computing in Python.

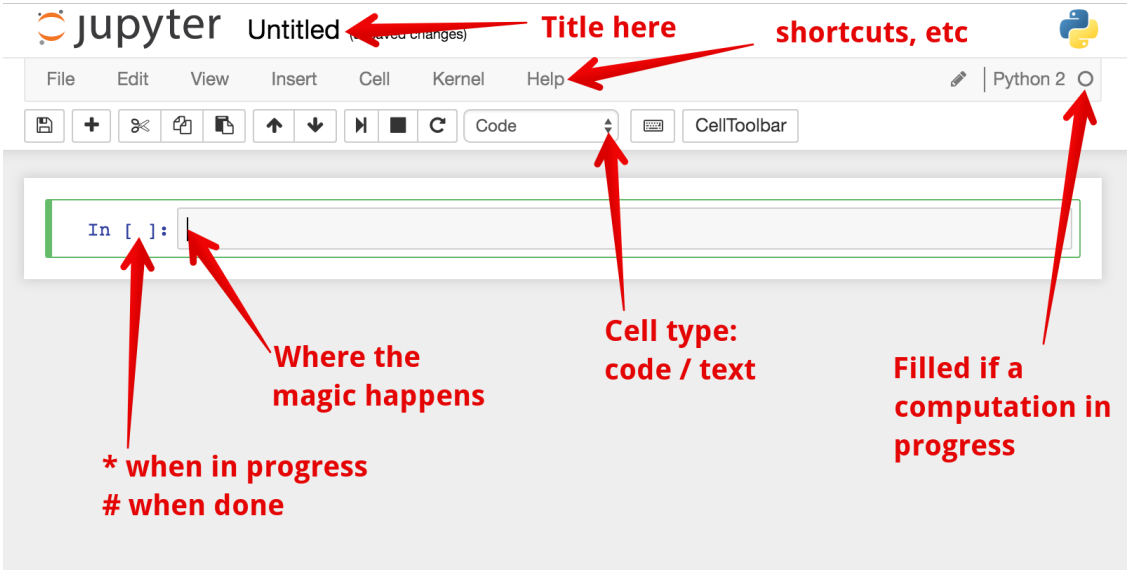
- NumPy: n-dimensional array package
- SciPy: scientific computing including: sparse matrices, numerical solvers, optimization routines, etc.
- Matplotlib: 2D plotting library

3.1 Jupyter Notebook

A notebook containing the following examples is available for you to download and follow along. In the directory where you downloaded the notebook, open up a Jupyter Notebook from a terminal:

```
jupyter notebook
```

and open `tools.ipynb`. A few things to note



- To execute a cell is **Shift + Enter**

- To restart the kernel (clean your slate) is **Esc + 00**

Throughout this tutorial, we will show a few tips for working with the notebook.

3.2 Python

Python is a high-level interpreted computing language. Here we outline a few of the basics and common trip-ups. For more information and tutorials, check out the Python Documentation. Note that at the moment, we are using Python 2.7, so those are the docs to follow. In particular, up to chapter 5 at this stage of the tutorials.

3.2.1 Types

Python makes a distinction on types: *int*, *float*, and *complex*:

```
>>> type(1) == int
True
>>> type(1.) == float
True
>>> type(1j) == complex
True
```

This is particularly important when doing division:

```
>>> 1/2
0
```

Note: If that scares you, you can always use division from the future! `from __future__ import division` see PEP0238

---

is integer division, while:

```
>>> 1./2.
0.5
```

is floating point division. This is only the case in Python 2, in Python 3, division will return a floating point number.

3.2.2 Counting and Lists

Python uses zero-based indexing:

```
>>> mylist = [6, 5, 4, 3]
>>> len(mylist)
4
>>> mylist[0]
6
```

There are a few handy indexing tricks:

```
>>> mylist[:2] # counting up
[6, 5]
>>> mylist[2:] # starting from
[4, 3]
```

```
>>> mylist[-1] # going backwards
3
```

3.2.3 Loops and List Comprehension

A `for` loop to append `10` values to a list looks like:

```
>>> n = 10
>>> a = []
>>> for i in range(n):
... a.append(i)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

or using list comprehension

```
>>> n = 10
>>> b = [i for i in range(n)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Try running these in the notebook and compare the times. To get a better picture, increase `n`.

Note: In the notebook, we use the cell magic `%%time` to track the amount of time it takes to execute cell

A handy tool for looping over lists is `enumerate`:

```
>>> mylist = ['Monty', 'Python', 'Flying', 'Circus'] # python was named after the␣
↪movie!
>>> for i, val in enumerate(mylist):
... print i, val
0 Monty
1 Python
2 Flying
3 Circus
```

This is a flavor of some of the flow control for lists in Python, for more details, check out chapters 4, 5 in the Python Tutorial.

3.2.4 If, elif, else

Conditionals in Python are implemented using `if`, `elif`, `else`

```
>>> # Pick a random number between 0 and 100
>>> import numpy as np
>>> number = (100.*np.random.rand(1)).round() # make it an integer
>>> if number > 42:
...     print '%i is too high'%number
... elif number < 42:
...     print '%i is too low'%number
... else:
...     print 'you found the secret to life. %i'%number
```

Note that the indentation level matters in python. Logical operators, `or`, `and` are also handy for constructing conditionals

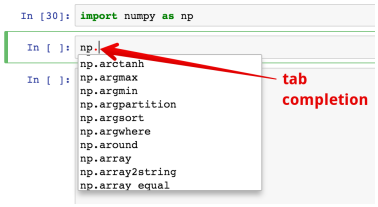
3.2.5 Functions

3.3 NumPy

NumPy contains the n-dimensional array machinery for storing and working with matrices and vectors. To use NumPy, it must first be imported. It is standard practice to import is as shorthand `np`.

```
>>> import numpy as np
```

Note: You can use tab completion to look at the attributes of an object



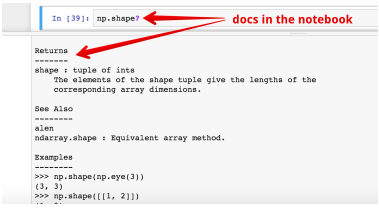
3.3.1 How many dimensions?

NumPy makes a distinction between scalars, vectors and arrays

```
>>> a = np.array(1) # scalar
>>> print a.shape # has no dimensions
()
>>> b = np.array([1]) # vector
>>> print b.shape # has one dimension
(1,)
>>> c = np.array([[1]]) # array
>>> print c.shape # has two (or more) dimensions
(1, 1)
```

The `shape` gives the length of each array dimension. (`size` gives you the number of elements)

Note: In the notebook, you can query documentation using a `?`



This distinction is particularly important when performing linear algebra operations. For instance, if we create a vector:

```
>>> v = np.random.rand(10)
>>> v.shape
(10,)
```

and then want to take an inner product, which we expect to output a scalar, one way you might think to do this (spoiler alert: it's wrong!):

```
>>> a = v.T * v
>>> a.shape
(10,)
```

turns out, the concept of a transpose doesn't matter for vectors, they only have one dimension, so there is no way to exchange dimensions. What we just did was a Hadamard product (element-wise multiplication), so

```
>>> v.T * v == v * v
True
```

So how do we take an inner product? (`dot`)

```
>>> b = v.dot(v)
>>> b.shape
()
```

Success! `b` is a scalar.

What happens if we work with arrays instead?

```
>>> w = np.random.rand(10,1)
>>> w.shape
(10,1)
```

3.4 SciPy

Note: This page is under construction. If you have suggestions, please make some noise on github!

3.4.1 Thinking in sparse

```
import scipy.sparse as sp
```

3.4.2 How do I solve you?

3.5 Matplotlib

3.6 Object Oriented Programming in Python

Also need functions

Class, Inheritance, Properties, Wrappers, and Self

3.7 Pointers

- Software Carpentry
- Python Tutorial

CHAPTER 4

Practices

- **Purpose**: In the development of SimPEG, we strive to follow best practices. Here, we provide an overview of those practices and some tools we use to support them.

Here we cover

- *Testing*
- *Style*
- *Pull Requests*
- *Licensing*

4.1 Testing

On each update, SimPEG is tested using the continuous integration service Travis CI. We use Codecov to check and provide stats on how much of the code base is covered by tests. This tells which lines of code have been run in the test suite. It does not tell you about the quality of the tests run! In order to assess that, have a look at the tests we are running - they tell you the assumptions that we do not want to break within the code base.

Within the repository, the tests are located in the top-level **tests** directory. Tests are organized similar to the structure of the repository. There are several types of tests we employ, this is not an exhaustive list, but meant to provide a few places to look when you are developing and would like to check that the code you wrote satisfies the assumptions you think it should.

All tests inherit from `unittest` which is a part of core python. Checkout the docs on unittest.

4.1.1 Compare with known values

In a simple case, you might now the exact value of what the output should be and you can `assert` that this is in fact the case. For example, in test_basemesh.py, we setup a 3D `BaseRectangularMesh` and assert that it has 3 dimensions.

```
import unittest
import sys
from SimPEG.Mesh.BaseMesh import BaseRectangularMesh
import numpy as np

class TestBaseMesh(unittest.TestCase):

    def setUp(self):
```

```
        self.mesh = BaseRectangularMesh([6, 2, 3])

    def test_meshDimensions(self):
        self.assertTrue(self.mesh.dim, 3)
```

The class inherits from `unittest.TestCase`. When running the tests, the `setUp` is run first, in this case we attach a mesh to the instance of this class, and then all functions with the naming convention `test_XXX` are run. Here we check that the dimensions are correct for the 3D mesh.

If the value is not an integer, you can be subject to floating point errors, so `assertTrue` might be too harsh. In this case, you will want to use a tolerance. For instance in test_maps.py

```
class MapTests(unittest.TestCase):

    # method setUp is used to create meshes

    def test_mapMultiplication(self):
        M = Mesh.TensorMesh([2,3])
        expMap = Maps.ExpMap(M)
        vertMap = Maps.SurjectVertical1D(M)
        combo = expMap*vertMap
        m = np.arange(3.0)
        t_true = np.exp(np.r_[0,0,1,1,2,2.])
        self.assertLess(np.linalg.norm((combo * m)-t_true,np.inf),TOL)
```

These are rather simple examples, more advanced tests might include solving an electromagnetic problem numerically and comparing it to an analytical solution , or performing an adjoint test to test `Jvec` and `Jtvec`.

4.1.2 Order and Derivative Tests

Order tests can be used when you are testing differential operators (we are using a second-order, staggered grid discretization for our operators). For example, testing a 2D curl operator in test_operators.py

```
import numpy as np
import unittest
from SimPEG.Tests import OrderTest

class TestCurl2D(OrderTest):
    name = "Cell Grad 2D - Dirichlet"
    meshTypes = ['uniformTensorMesh']
    meshDimension = 2
    meshSizes = [8, 16, 32, 64]

    def getError(self):
        # Test function
        ex = lambda x, y: np.cos(y)
        ey = lambda x, y: np.cos(x)
        sol = lambda x, y: -np.sin(x)+np.sin(y)

        sol_curl2d = call2(sol, self.M.gridCC)
        Ec = cartE2(self.M, ex, ey)
        sol_ana = self.M.edgeCurl*self.M.projectFaceVector(Ec)
        err = np.linalg.norm((sol_curl2d-sol_ana), np.inf)

        return err
```

```
    def test_order(self):
        self.orderTest()
```

Derivative tests are a particular type or *Order and Derivative Tests*, and since they are used so extensively, SimPEG includes a `checkDerivative` method.

In the case of testing a derivative, we consider a Taylor expansion of a function about x. For a small perturbation Δx,

$$f(x + \Delta x) \simeq f(x) + J(x)\Delta x + \mathcal{O}(h^2)$$

As Δx decreases, we expect $\|f(x) - f(x + \Delta x)\|$ to have first order convergence (e.g. the improvement in the approximation is directly related to how small Δx is, while if we include the first derivative in our approximation, we expect that $\|f(x) + J(x)\Delta x - f(x + \Delta x)\|$ to converge at a second-order rate. For example, all maps have an associated derivative test . An example from test_FDEM_derivs.py

```
def derivTest(fdemType, comp):

    # setup problem, survey

    def fun(x):
        return survey.dpred(x), lambda x: prb.Jvec(x0, x)
    return Tests.checkDerivative(fun, x0, num=2, plotIt=False, eps=FLR)
```

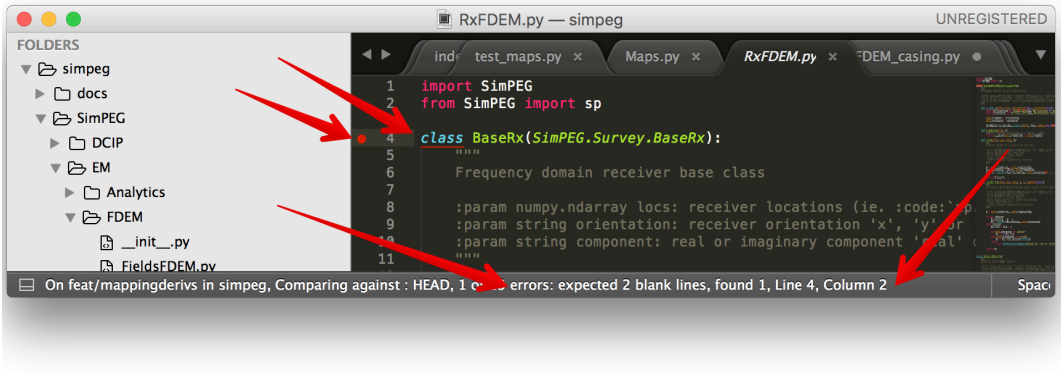
4.2 Style

Consistency make code more readable and easier for collaborators to jump in. PEP 8 provides conventions for coding in Python. SimPEG is currently not PEP 8 compliant, but we are working towards it and would appreciate contributions that do too!

There are a few resources we use to promote these practices: the service Quantified Code to check for consistency (... we have some work to do. Pull requests are welcome!)

Sublime has PEP 8 linter packages that you can use. I use SublimeLinter-pep8. You can install it by going to your package manager (*cmd* + *shift* + *p*), install package and search for SublimeLinter-pep8. Installation instructions are available at https://github.com/SublimeLinter/SublimeLinter-pep8.

This highlights and gives you tips on how to correct the code.



Below is a sample user-settings configuration for the SublimeLinter (Sublime Text > Preferences > Package Settings > SublimeLinter > Settings-User)

```
{
    "user": {
        "debug": false,
        "delay": 0.25,
        "error_color": "D02000",
        "gutter_theme": "Packages/SublimeLinter/gutter-themes/Default/Default.gutter-
↪theme",
        "gutter_theme_excludes": [],
        "lint_mode": "background",
        "linters": {
            "pep8": {
                "@disable": false,
                "args": [],
                "excludes": [],
                "ignore": "",
                "max-line-length": null,
                "select": ""
            },
            "proselint": {
                "@disable": false,
                "args": [],
                "excludes": []
            }
        },
        "mark_style": "solid underline",
        "no_column_highlights_line": false,
        "passive_warnings": false,
        "paths": {
            "linux": [],
            "osx": [
                "/anaconda/bin"
            ],
            "windows": []
        },
        "python_paths": {
            "linux": [],
            "osx": [],
            "windows": []
        },
        "rc_search_limit": 3,
        "shell_timeout": 10,
        "show_errors_on_save": false,
        "show_marks_in_minimap": true,
        "syntax_map": {
            "html (django)": "html",
            "html (rails)": "html",
            "html 5": "html",
            "javascript (babel)": "javascript",
            "magicpython": "python",
            "php": "html",
            "python django": "python",
            "pythonimproved": "python"
        },
        "warning_color": "DDB700",
        "wrap_find": true
    }
}
```

4.3 Pull Requests

Pull requests are a chance to get peer review on your code. For the git flow, we do all pull requests onto **dev** before merging to **master**. If you are working on a specific geophysical application, e.g. electromagnetics, pull requests should first go through that method's **dev** branch, in this case, **em/dev**. This way, we make sure that new changes are up-to date with the given method, and there is a chance to catch bugs before putting changes onto **master**. We do code reviews on pull requests, with the aim of promoting best practices and ensuring that new contributions can be built upon by the SimPEG community. For more info on best practices for version control and git flow, check out the article A successful git branching model

4.4 Licensing

We want SimPEG to be a useful resource for the geoscience community and believe that following open development practices is the best way to do that. SimPEG is licensed under the MIT license which is allows open and commercial use and extension of SimPEG. It does not force packages that use SimPEG to be open source nor does it restrict commercial use.

CHAPTER 5

For Example: There and Back Again

Note: This page is under construction. If you have suggestions, please make some noise on github!

- **Purpose** The goal of this page is to walk through a simple inversion using SimPEG.

CHAPTER 6

Pixels and Their Neighbors

Note: This article was originally published in the Leading Edge:

Rowan Cockett, Lindsey J. Heagy, and Douglas W. Oldenburg (2016). "Pixels and their neighbors: Finite volume." The Leading Edge, 35(8), 703–706. doi: 10.1190/tle35080703.1

Associated notebooks are available on github and can be run online with binders

We take you on the journey from continuous equations to their discrete matrix representations using the finite volume method for the Direct Current (DC) resistivity problem. These techniques are widely applicable across geophysical simulation types and have their parallels in finite element and finite difference. We show derivations visually, as you would on a whiteboard, and have provided an accompanying notebook to explore the numerical results using SimPEG *(Cockett et al. 2015)*.

6.1 DC Resistivity

Fig. 6.1: Setup of a DC resistivity survey.

DC resistivity surveys obtain information about subsurface electrical conductivity, σ. This physical property is often diagnostic in mineral exploration, geotechnical, environmental and hydrogeologic problems, where the target of interest has a significant electrical conductivity contrast from the background. In a DC resistivity survey, steady state currents are set up in the subsurface by injecting current through a positive electrode and completing the circuit with a return electrode (Fig. 6.1). The equations for DC resistivity are derived in Fig. 6.2. Conservation of charge (which can be derived by taking the divergence of Ampere's law at steady state) connects the divergence of the current density everywhere in space to the source term which consists of two point sources, one positive and one negative. The flow of current sets up electric fields according to Ohm's law, which relates current density to electric fields through the electrical conductivity. From Faraday's law for steady state fields, we can describe the electric field in terms of a scalar potential, ϕ, which we sample at potential electrodes to obtain data in the form of potential differences.

Fig. 6.2: Derivation of the DC resistivity equations.

To set up a solvable system of equations, we need the same number of unknowns as equations, in this case two unknowns (one scalar, ϕ, and one vector $\vec{j}$) and two first-order equations (one scalar, one vector).

In this tutorial, we walk through setting up these first order equations in finite volume in three steps: (1) defining where the variables live on the mesh; (2) looking at a single cell to define the discrete divergence and the weak formulation;

and (3) moving from a cell based view to the entire mesh to construct and solve the resulting matrix system. The notebooks included with this tutorial leverage the SimPEG package (http://simpeg.xyz, *Cockett et al. 2015*), which extends the methods discussed here to various mesh types.

6.2 Where do things live?

To bring our continuous equations into the computer, we need to discretize the earth and represent it using a finite(!) set of numbers. In this tutorial we will explain the discretization in 2D and generalize to 3D in the notebooks. A 2D (or 3D!) mesh is used to divide up space, and we can represent functions (fields, parameters, etc.) on this mesh at a few discrete places: the nodes, edges, faces, or cell centers. For consistency between 2D and 3D we refer to faces having area and cells having volume, regardless of their dimensionality. Nodes and cell centers naturally hold scalar quantities while edges and faces have implied directionality and therefore naturally describe vectors. The conductivity, σ, changes as a function of space, and is likely to have discontinuities (e.g. if we cross a geologic boundary). As such, we will represent the conductivity as a constant over each cell, and discretize it at the center of the cell. The electrical current density, $\vec{j}$, will be continuous across conductivity interfaces, and therefore, we will represent it on the faces of each cell. Remember that $\vec{j}$ is a vector; the direction of it is implied by the mesh definition (i.e. in x, y or z), so we can store the array $\mathbf{j}$ as *scalars* that live on the face and inherit the face's normal. When $\vec{j}$ is defined on the faces of a cell the potential, $\vec{\phi}$, will be put on the cell centers (since $\vec{j}$ is related to ϕ through spatial derivatives, it allows us to approximate centered derivatives leading to a staggered, second-order discretization). Once we have the functions placed on our mesh, we look at a single cell to discretize each first order equation. For simplicity in this tutorial we will choose to have all of the faces of our mesh be aligned with our spatial axes (x, y, and z), the extension to curvilinear meshes will be presented in the supporting notebooks.

Fig. 6.3: Anatomy of a finite volume cell.

6.3 One cell at a time

To discretize the first order differential equations we consider a single cell in the mesh and we will work through the discrete description of equations (1) and (2) over that cell.

6.3.1 (1) In and out

Equation (a) relates the divergence of the current density to a source term. To discretize using finite volume, we will look at the divergence geometrically. The divergence is the integral of a flux through a closed surface as that enclosed volume shrinks to a point. Since we have discretized and no longer have continuous functions, we cannot fully take the limit to a point; instead, we approximate it around some (finite!) volume: *a cell*. The flux out of the surface ($\vec{j} \cdot \vec{n}$) is actually how we discretized $\vec{j}$ onto our mesh (i.e. $\mathbf{j}$) except that the face normal points out of the cell (rather than in the axes direction). After fixing the direction of the face normal (multiplying by ± 1), we only need to calculate the face areas and cell volume to create the discrete divergence matrix.

Fig. 6.4: Geometrical definition of the divergence and the discretization.

So we have half of the equation discretized — the left hand side. Now we need to take care of the source: it contains two dirac delta functions — these are infinite at their origins, r_{s^+} and r_{s^-}, (infinity is not exactly something a computer does well with!). However, the volume integral of a delta function *is* well defined: it is *unity* if the volume contains the origin of the delta function otherwise it is *zero*. As such, we can integrate both sides of the equation over the volume enclosed by the cell. Since $\mathbf{Dj}$ is constant over the cell, the integral is simply a multiplication by the volume of the

cell $\mathrm{v}\mathbf{Dj}$. The integral of the source is zero unless one of the source electrodes is located inside the cell, in which case it is $q = \pm I$. Now we have a discrete description of equation 1 over a single cell:

$$\mathrm{v}\mathbf{Dj} = q$$

6.3.2 (2) Scalar equations only, please

Equation (b) is a vector equation, so really it is two or three equations involving multiple components of $\vec{j}$. We want to work with a single scalar equation, allow for anisotropic physical properties, and potentially work with non-axis-aligned meshes — how do we do this?! We can use the **weak formulation** where we take the inner product ($\int \vec{a} \cdot \vec{b} dv$) of the equation with a generic face function, $\vec{f}$. This reduces requirements of differentiability on the original equation and also allows us to consider tensor anisotropy or curvilinear meshes *(Haber 2014)*.

In Fig. 6.5, we visually walk through the discretization of equation (b). On the left hand side, a dot product requires a *single* cartesian vector, $[\mathbf{j_x}, \mathbf{j_y}]$. However, we have a j defined on each face (2 j_x and 2 j_y in 2D!). There are many different ways to evaluate this inner product: we could approximate the integral using trapezoidal, midpoint or higher order approximations. A simple method is to break the integral into four sections (or 8 in 3D) and apply the midpoint rule for each section using the closest $\mathbf{j}$ components to compose a cartesian vector. A $\mathbf{P}_i$ matrix (size 2 × 4) is used to pick out the appropriate faces and compose the corresponding vector (these matrices are shown with colors corresponding to the appropriate face in the figure). On the right hand side, we use a vector identity to integrate by parts. The second term will cancel over the entire mesh (as the normals of adjacent cell faces point in opposite directions) and ϕ on mesh boundary faces are zero by the Dirichlet boundary condition footnote{We are using Dirichlet for simplicity in this example, in practice, Neumann conditions are often used. This is because "infinity" needs to be further away if applying Dirichlet boundary conditions since potential falls off as $1/r^2$ and current density as $1/r^3$.}. This leaves us with the divergence, which we already know how to do!

Fig. 6.5: Discretization using the weak formulation and inner products.

The final step is to recognize that, now discretized, we can cancel the general face function $\mathbf{f}$ and transpose the result (for convention's sake):

$$\frac{1}{4}\sum_{i=1}^{4} \mathbf{P}_i^\top \sqrt{v}\boldsymbol{\Sigma}^{-1}\sqrt{v}\mathbf{P}_i\mathbf{j} = \mathbf{D}^\top v\phi$$

6.4 All together now

We have now discretized the two first order equations over a single cell. What is left is to assemble and solve the DC system over the entire mesh. To implement the divergence on the full mesh, the stencil of ± 1's must index into $\mathbf{j}$ on the entire mesh (instead of four elements). Although this can be done in a `for-loop`, it is conceptually, and often computationally, easier to create this stencil using nested kronecker products (see notebook). The volume and area terms in the divergence get expanded to diagonal matrices, and we multiply them together to get the discrete divergence operator. The discretization of the *face* inner product can be abstracted to a function, $\mathbf{M}_f(\sigma^{-1})$, that completes the inner product on the entire mesh at once. The main difference when implementing this is the $\mathbf{P}$ matrices, which must index into the entire mesh.

With the necessary operators defined for both equations on the entire mesh, we are left with two discrete equations

$$\begin{aligned} \mathrm{diag}(\mathbf{v})\mathbf{Dj} &= \mathbf{q} \\ \mathbf{M}_f(\sigma^{-1})\mathbf{j} &= \mathbf{D}^\top \mathrm{diag}(\mathbf{v})\phi. \end{aligned}$$

Note that now all variables are defined over the entire mesh. We could solve this coupled system or we could eliminate $\mathbf{j}$ and solve for ϕ directly (a smaller, second-order system).

$$\text{diag}(\mathbf{v})\mathbf{D}\mathbf{M}_f(\sigma^{-1})^{-1}\mathbf{D}^\top\text{diag}(\mathbf{v})\boldsymbol{\phi} = \mathbf{q}.$$

By solving this system matrix, we obtain a solution for the electric potential ϕ everywhere in the domain. Creating predicted data from this requires an interpolation to the electrode locations and subtraction to obtain potential differences!

Fig. 6.6: Electric potential on (a) Tensor and (b) Curvilinear meshes.

Moving from continuous equations to their discrete analogues is fundamental in geophysical simulations. In this tutorial, we have started from a continuous description of the governing equations for the DC resistivity problem, selected locations on the mesh to discretize the continuous functions, constructed differential operators by considering one cell at a time, assembled and solved the discrete DC equations. Composing the finite volume system in this way allows us to move to different meshes and incorporate various types of boundary conditions that are often necessary when solving these equations in practice.

6.5 References

- Cockett, Rowan, Seogi Kang, Lindsey J. Heagy, Adam Pidlisecky, and Douglas W. Oldenburg. "SimPEG: An Open Source Framework for Simulation and Gradient Based Parameter Estimation in Geophysical Applications." Computers & Geosciences 85 (2015): 142–54. doi:10.1016/j.cageo.2015.09.015.
- Haber, E. Computational Methods in Geophysical Electromagnetics: Mathematics in Industry. Society for Industrial and Applied Mathematics, 2014. https://books.google.ca/books?id=favjoQEACAAJ.

CHAPTER 7

Fields, Fluxes, Physics

Note: This page is under construction. If you have suggestions, please make some noise on github!

CHAPTER 8

Guess, but first tell me what you know

Note: This page is under construction. If you have suggestions, please make some noise on github!

CHAPTER 9

Stroll, Preferably downhill

Note: This page is under construction. If you have suggestions, please make some noise on github!

CHAPTER 10

Which way is down?

Note: This page is under construction. If you have suggestions, please make some noise on github!

CHAPTER 11

Spices - Embedding knowledge

Note: This page is under construction. If you have suggestions, please make some noise on github!

CHAPTER 12

Resources

- **Purpose:** A collection of resources we find useful.

This is far from complete, so if you have additional resources, please edit this page

12.1 SimPEG

- SimPEG: homepage for SimPEG!
- SimPEG docs: SimPEG documentation
- SimPEG repo GitHub repository with SimPEG code

12.2 Setting up your python work environment

12.2.1 Package managers

- Anaconda
- Enthought Canopy

12.2.2 Git and GitHub

- GitHub: version controlling and collaborating on software
- GitHub Help: resources for learning Git and GitHub
- A successful git branching model
- SourceTree: desktop app for managing version controlled repositories

12.2.3 Tools for Developing

- Sublime Text
- Jupyter

12.3 Scientific Computing with Python

- Python: homepage for python
- Software Carpentry: tutorials for scientific computing

12.4 Tools for collaborating

- Gitter: instant messaging with SimPEG developers
- Slack: instant messaging for teams

12.5 Resources for Geophysical Inversions

- GeoSci.xyz: resources for geophysics
- Inversion Tutorials: article and associated Jupyter