Synthetic model building for training neural networks in a Jupyter notebook

Robert G. Clapp

Abstract

Neural networks require tens of thousands of correctly labeled datasets, something that does not exist in reflection seismology. Synthetic data can be used to help fill this data gap. In this paper I describe an update to a synthetic model generator aimed at producing realistic labeled datasets for training neural nets.

Introduction

Synthetic data can be used to help train neural networks . In a previous work I described a simple basin modeling approach to create realistic synthetics . In this paper I extend that work, rewriting the modules in C++, binding to python with pybind , and using python to create the modules. I will be reviewing the basic synthetic generation modules. I then will describe how to use the python interface. I will finish by talking about the types of synthetic models I am generating and future plans for expansion of the code.

Running the code

The basic idea of the synthetic model generator is to describe a series of geologic events such as deposition, faulting, emplacement, and compression. Each event is described by a series of parameters that attempt to simulate the geologic event.
The original version of the synthetic model generator was written in Fortran. To build a model meant writing/generating parameters file hundreds of lines long. Changing a single parameter required regenerating the entire model.

This new version keeps the same idea of building a model from a series of geologic events. The code is rewritten in C++ and parallelized using Thread Building blocks . Each module is wrapped using pybind11 into the python module `pySyntheticGen`, which in turned is wrapped in the pure python module `syntheticGen`.

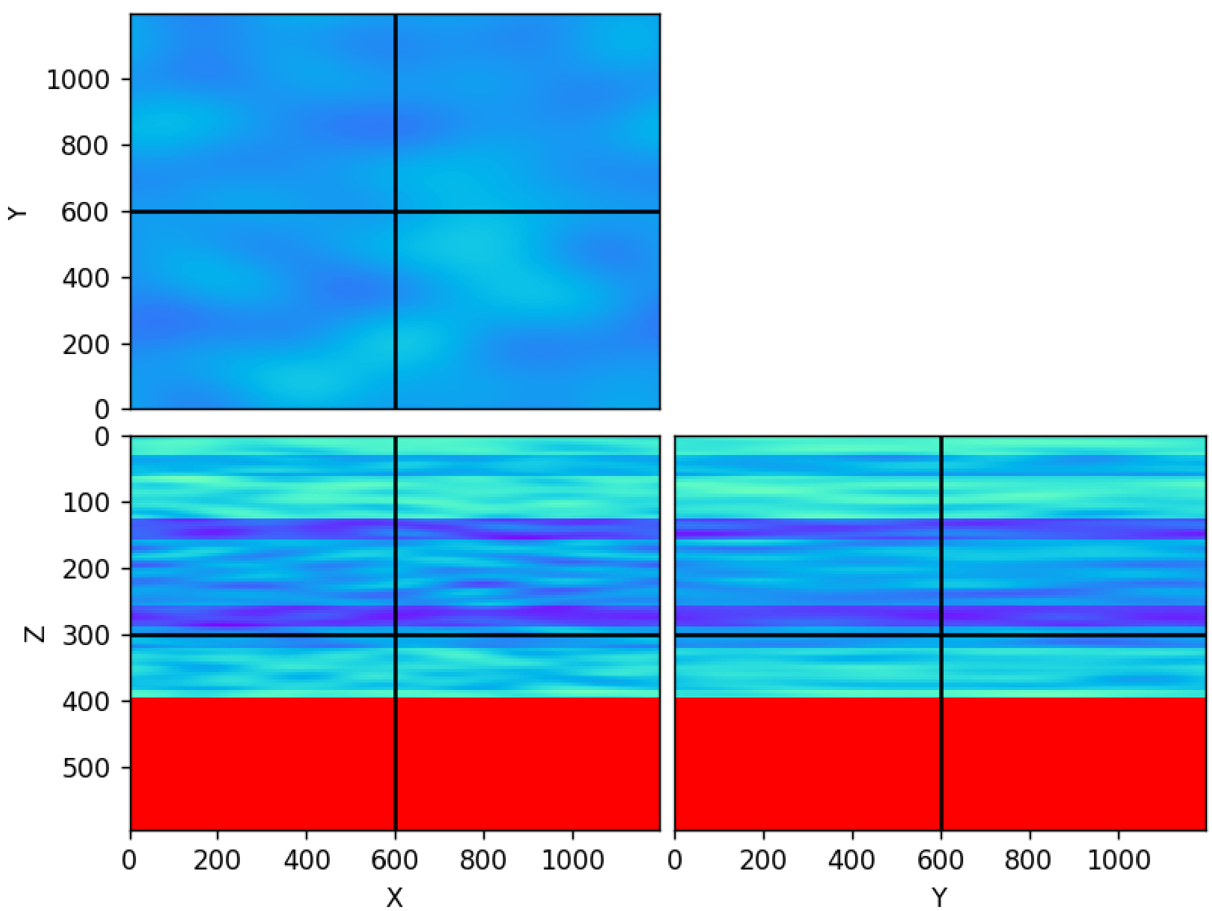
To start a new model we import the `syntheticGen` module, the full self-doc can be found in the appendix. We initialize the model by describing the size of the domain in x and y. For our first example we will create a small model with 300 samples in x and y.

In [1]:
```
import pySepVector
import syntheticModel
mod=syntheticModel.geoModel(nx=300,ny=300)
```

The most basic module is deposition. The deposition model adds a layer with a given parameter value (such as velocity). We can also decide to add spatial variations and interbed layers. In this case we will add two different layers. The first layer with an average velocity of 2700 m/s, a thickness of 100 samples. We will allow interbedding, where the velocity can vary by 30%. In addition we will allow variation as a function of space.

In [2]:
```
%matplotlib notebook
from latex_envs.latex_envs import figcaption
import Cubeplot
mod.deposit(prop=2700,thick=100,var=.3,dev_pos=.1,layer=25,dev_layer=.3,
layer_rand=.3,band2=.01, band3=.01)
figcaption("An example of using the deposition module.", label="fig:depo
sition")
b=Cubeplot.plot(mod.getProp("velocity"))
```

Caption: An example of using the deposition module.



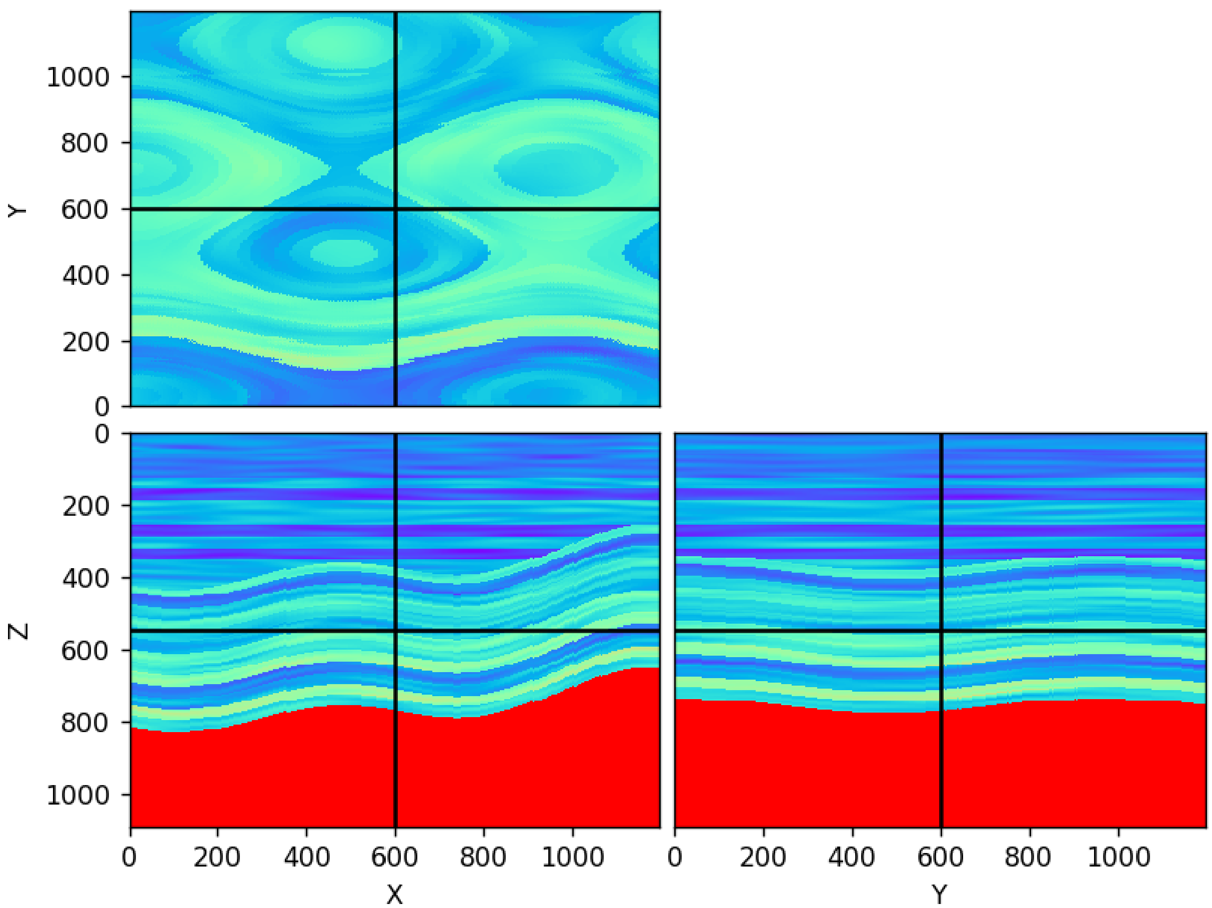
We can choose to introduce a compressional event. A compressional event produce an anticline-syncline pattern in the current model We can decide the angle of compression, the amount of uplift, and how much we want the pattern to vary spatially.

In [3]:
```
%matplotlib notebook

mod=syntheticModel.geoModel(nx=300,ny=300)
mod.deposit(prop=2700,thick=100,var=.3,dev_pos=.1,layer=25,dev_layer=.3,
layer_rand=.3,band2=.01, band3=.01)
mod.squish(max=150,random_inline=2.,random_crossline=3.,aziumth=40.,wave
length=.2)
mod.deposit(prop=2400,thick=50,var=.3,dev_pos=.1,layer=25,dev_layer=.2,l
ayer_rand=.3)
figcaption("An example of using the compressional module.", label="fig:c
ompress")

c=Cubeplot.plot(mod.getProp("velocity"))
```

Caption: An example of using the compressional module.

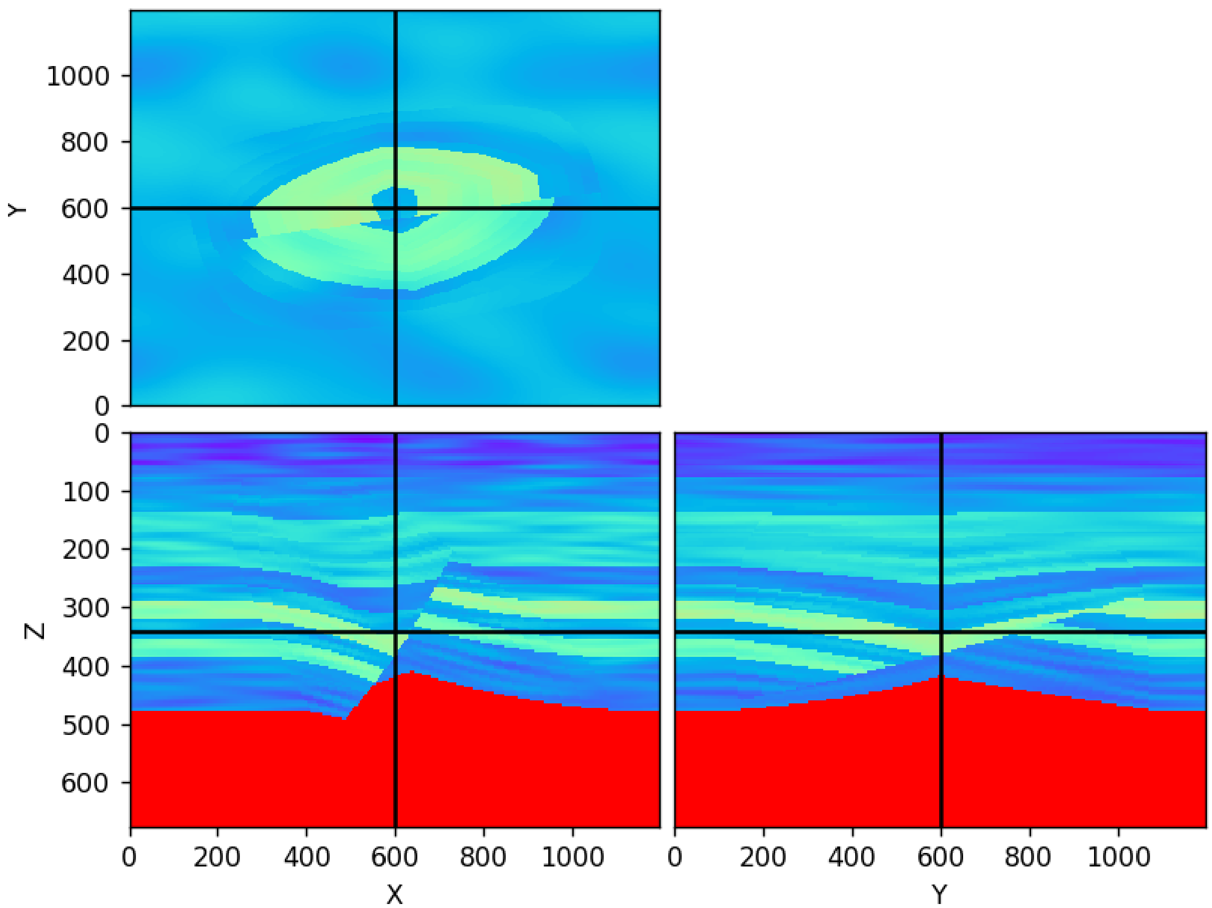


We can also add faults. Fault planes can be thought of as the surface of cylinder. Everything inside the cylinder rotates in one direction, everything outside the other direction. The further away the cylinder's focus, the more the fault looks line a plane. The bigger the angular rotation the more fault throw. The fault is centered at a given location, as we move away from that location along the cylinder the rotation lessens. We expect less rotation as we move away from the cylinder's edge.

In [4]:
```
%matplotlib notebook
mod=syntheticModel.geoModel(nx=300,ny=300)
mod.deposit(prop=2700,thick=100,var=.3,dev_pos=.1,layer=25,dev_layer=.3,
layer_rand=.3,band2=.01, band3=.01)
mod.fault(begx=.5,begy=.5,begz=.5,daz=800,dz=700,azimuth=10,theta_die=12
, theta_shit=7,dist_die=.4,perp_die=.4)
mod.deposit(prop=2400,thick=20,var=.3,dev_pos=.1,layer=25,dev_layer=.2,l
ayer_rand=.3)
figcaption("An example of using the fault  module.", label="fig:fault")

d=Cubeplot.plot(mod.getProp("velocity"))
```

Caption: An example of using the fault module.

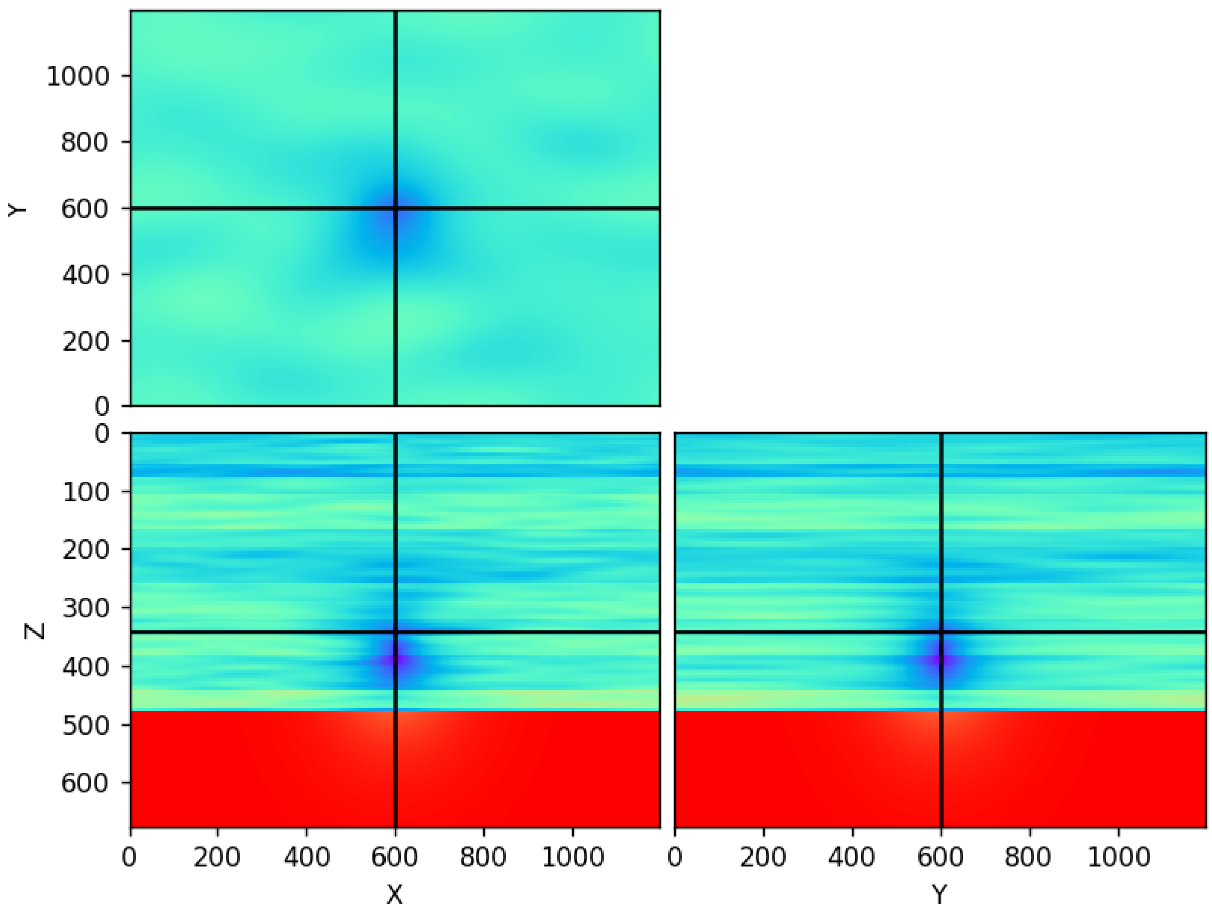


We can add Gaussian anomalies into our model by describing their location and amplitude.

In [5]:
```
%matplotlib notebook
mod=syntheticModel.geoModel(nx=300,ny=300)
mod.deposit(prop=2700,thick=100,var=.3,dev_pos=.1,layer=25,dev_layer=.3,
layer_rand=.3,band2=.01, band3=.01)
mod.gaussian(vplus=-1000.,var=40.)
mod.deposit(prop=2400,thick=20,var=.1,dev_pos=.1,layer=25,dev_layer=.2,l
ayer_rand=.3)
figcaption("An example of using the Gaussian  module.", label="fig:gauss
ian")

d=Cubeplot.plot(mod.getProp("velocity"))
```

Caption: An example of using the Gaussian module.

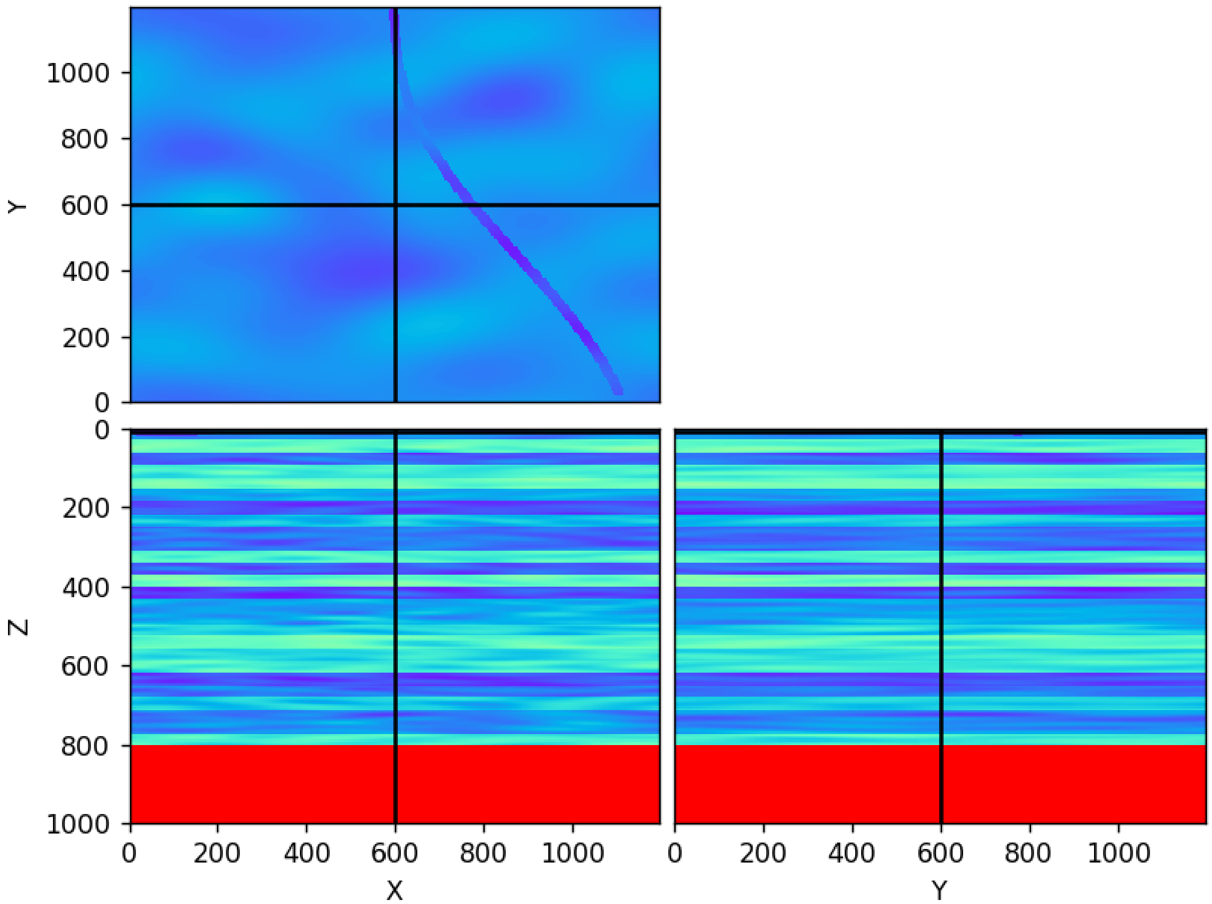


We can add river channels to our model by describing the beginning location and angle. We can add partial fill into the river channels and have their location move as a function of depth in a logical meandering stream pattern.

In [6]:
```
%matplotlib notebook
mod=syntheticModel.geoModel(nx=300,ny=300)
mod.deposit(prop=2700,thick=200,var=.3,dev_pos=.1,layer=25,dev_layer=.3,
layer_rand=.3,band2=.01, band3=.01)
mod.erodeRiver()
mod.deposit(prop=2400,thick=1,var=.1,dev_pos=.1,layer=25,dev_layer=.2,la
yer_rand=.3)
figcaption("An example of using the erode river  module.", label="fig:ri
ver")

d=Cubeplot.plot(mod.getProp("velocity"),slice1=2)
```

Caption: An example of using the erode river module.

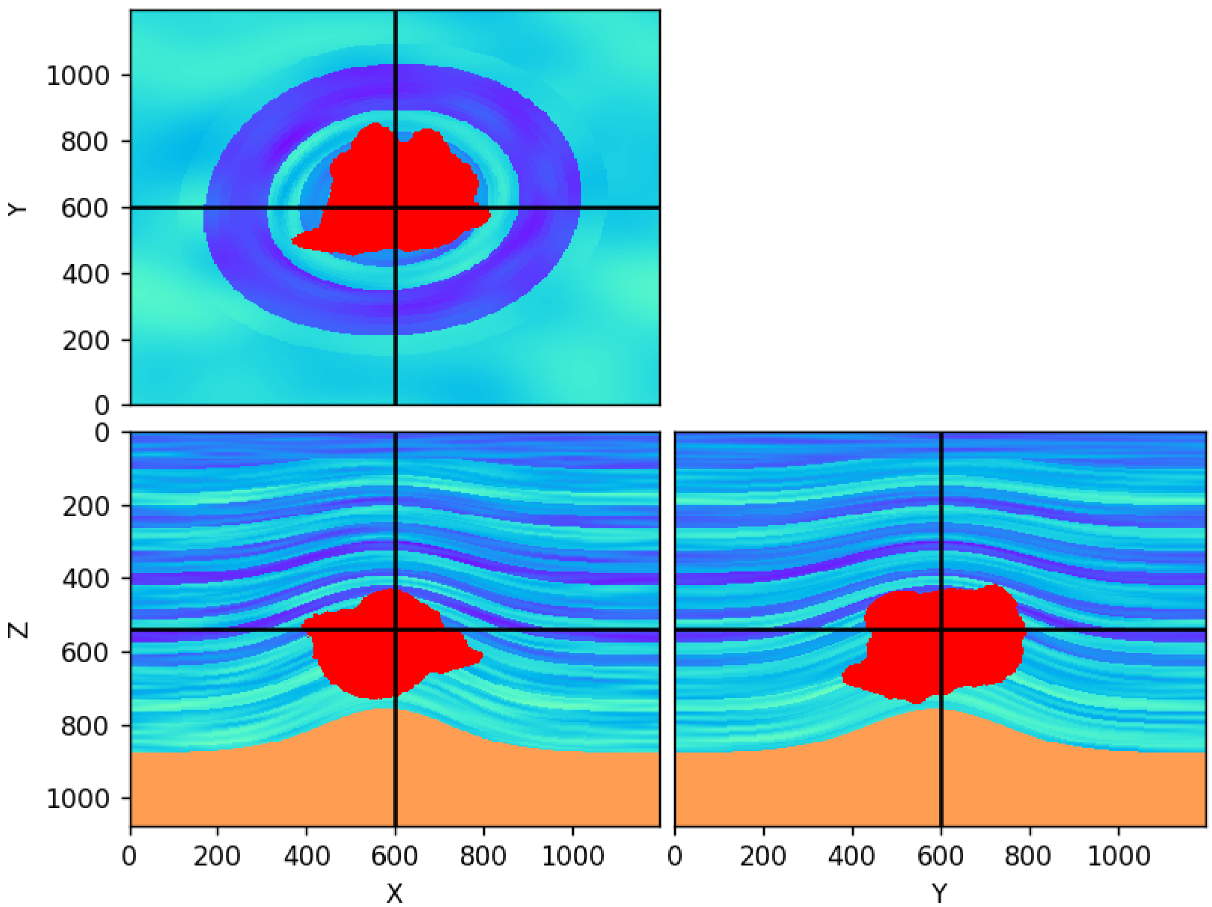


We can emplace salt. The salt is created by putting random sized perturbations within in a given area. Random, relatively low wavenumber, conductivity is then assigned. The heat equation is used to merge the perturbations.

In [7]:
```
%matplotlib notebook
mod=syntheticModel.geoModel(nx=300,ny=300)
mod.deposit(prop=2700,thick=200,var=.3,dev_pos=.1,layer=25,dev_layer=.3,
layer_rand=.3,band2=.01, band3=.01)
mod.implace(ntSteps=50)
mod.deposit(prop=2400,thick=20,var=.1,dev_pos=.1,layer=25,dev_layer=.2,l
ayer_rand=.3)
figcaption("An example of using the implace  module.", label="fig:implac
e")

d=Cubeplot.plot(mod.getProp("velocity"))
```

Caption: An example of using the implace module.



How good do the synthetics need to be?

Training a neural network at some level is just solving a non-linear inversion problem. The key to solving a non-linear problem is finding the neighborhood the solution lives in. One way to think about pre-training and transferring learning is that it is attempting to get the network into generally the correct neighborhood. This hypothesis can lead to several interesting questions.

- Can we build an initial network that generally understands seismic migrated volumes?
- Do we need to have different networks for different geologic basins?
- Can we slowly navigate into the correct neighborhood by training with a series of more realistic synthetic datasets?

This third question leads us to a potential solution for our lack of data problem. Creating geologic models is relatively cheap (minutes on a single machine). Finite difference modeling/migration is expensive hours/days on 10s of machines for a single dataset. A possible approach is to create hundreds of synthetic `migrated` volumes by calculating impedances and then convolving with a wavelet. A smaller set of modeled/migrated datasets can then be used to further improve the network. Such an approach would also help with the first two questions. For example, synthetics mimicking different basins can be generated to create basin specific networks.

Packaging

There are several ways to use the `syntheticGen` code. You can clone the code from Stanford's School of Earth and Environment's gitlab site . Building the code requires several other packages that also publicly available from that website. You can find all of the dependencies by looking in the `docker` sub-folder. You can download the latest version inside a docker from `rgc007/synthetic-gen`. This docker is accessible through a Jupyter notebook . Finally, there should be a link off the sep website where you found this paper to bring up an interactive document.

Future plans

There are several different modules that could be improved upon. The salt generation still does not consistently provide realistic salt geometries. The deposition model could benefit from using a similar heat equation approach to output different sediments rather than smoothing random numbers. Adding the abilities to do turbidites, emplacement, and other geologic features would also be useful improvements.

Finally, it might be useful to rewrite some of the code to run on GPUs. The shear number of models that will be needed for some machine learning problems will make speed of model generation even more important.

Conclusion

Synthetic data can be used to help train neural networks. Simplified basin modeling is one approach to creating synthetics. To improve the codes eases of use the C++ base code is wrapped in python interfaces. A Jupyter notebook is used to further enhance the code's accessibility.

Bibliography

Clapp, Robert. 2018. Geologic Synthetic Model Generator. http://zapad.stanford.edu/SEP-external/syntheticModel (http://zapad.stanford.edu/SEP-external/syntheticModel).

Clapp, Robert G. 2014. Synthetic Model Building Using a Simplified Basin Modeling Approach. SEP. http://sepwww.stanford.edu/public/docs/sep155 (http://sepwww.stanford.edu/public/docs/sep155).

Jakob, Wenzel, Jason Rhinelander, and Dean Moldovan. 2016. "pybind11 — Seamless Operability between C++11 and Python." https://github.com/pybind/pybind11 (https://github.com/pybind/pybind11).

Kluyver, Thomas, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, et al. 2016. "Jupyter Notebooks-a Publishing Format for Reproducible Computational Workflows." In ELPUB, 87–90.

Le, Tuan Anh, Atilim Giineş Baydin, Robert Zinkov, and Frank Wood. 2017. "Using Synthetic Data to Train Neural Networks Is Model-Based Reasoning." In Neural Networks (IJCNN), 2017 International Joint Conference on, 3514–21. IEEE.

Merkel, Dirk. 2014. "Docker: Lightweight Linux Containers for Consistent Development and Deployment." Linux Journal 2014 (239): 2.

Reinders, James. 2007. Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism. O'Reilly Media, Inc.

Van Baarsen, Jeroen. 2014. GitLab Cookbook. Packt Publishing Ltd.

Apppendix

In [8]:
```
help(syntheticModel.geoModel)
```

```
Help on class geoModel in module syntheticModel:

class geoModel(builtins.object)
 |  Methods defined here:
 |
 |  __init__(self, **kw)
 |      Create a new geomodel
 |      nx - (100) Number of samples in x
 |      ox - (0.)  First sample in x
 |      dx - (4.)  Sampling in x
 |      ny - (100) Number of samples in y
 |      oy - (0.)  First sample in y
 |      dy - (4.)  Sampling in y
 |      dz - (4.)  Sampling in z
 |      basement - (4000.) Basement property
 |      nbasement - (50) Initial number of samples in the basement
 |      properties - (['velocity']) Properties to model
 |
 |  compact(self, **kw)
 |      Compact layers
 |      compact - [0.] Compact layers
 |
 |  deposit(self, **kw)
 |      Deposit a layer
 |      base_param - ["velocity"] Base param to base all other properti
es
 |      band1  - [.60] Bandpass parameter axis 1 property dependent vs.
band1=
 |      band2  - [.05] Bandpass parameter axis 2 property dependent
 |      band3  - [.05] Bandpass parameter axis 3 property dependent
 |      ratio  - [.4]  Base ratio of property to main property
 |      var    - [.0]  Variance from main parameter
 |      layer_rand - [.5] Randomness variation within layer
 |      layer  - [9999.] Layer Base value
 |      prop  - [1.4]
 |      dev_layer - [0.]
 |      dev_pos - [0.]
 |      thick - [0.]
 |
 |  erodeBowl(self, **kw)
 |      Erode a bowl shape
 |      center2 - [.5] Create a bowl fractional amount into model2
 |      center3 - [.5] Create a bowl fractional amount into model3
 |      width2  - [.01] Width of bowl fractional to length of axis 2
 |      width3  - [.01] Width of bowl fractional to length of axis 3
 |      depth   - [.01] Depth of bowl fractional  to length of axis 1
 |      fill_depth - [.01] Fill depth of bowl fractional to length of a
xis 1
 |      fill_prop - [.3] Fill value, dependent on model parameter
 |
 |  erodeFlat(self, **kw)
 |      Erode a flat surface
 |      depth [.1] Fractional depth (axis 1) to slice off
 |
 |  erodeRiver(self, **kw)
 |      Erode a river shape
 |      start2 - [.5] Position (relative to axis length) to start river
```

```
|       start3 - [.0] Position (relative) to start river
|       dist   - [1.4] Length (relative) of river
|       azimuth - [0.] Angle for river
|       fill_prop - [0.] Fill value for deposition for river chanel
|       fill_depth - [0.] Fill dpeth for river chanel
|       nlevels - [1] Number of river chanel bends to layout
|       wavelength - [.01] Wavelenth multiplier for random river path
|       waveamp - [.01] Wave ampitude multiplier
|       thick - [.3] Thicknewss of river chanel
|
|   fault(self, **kw)
|       Fault model
|       azimuth - [0.] Azimuth of fault
|       begx    - [.5] Relative location of the begining of fault x
|       begy    - [.5] Relative location of the begining of fault y
|       begz    - [.5] Relative location of the begining of fault z
|       dz      - [0.] Distance away for the center of a circle in z
|       daz     - [.01] Distance away in azimuth
|       perp_die- [0.1] Dieoff of fault in in perpdincular distance
|       deltaTheta-[.1] Dieoff in theta away from the fault
|       dist_die-  [0.] Distance dieoff of fault
|       theta_die- [0.01] Distance dieoff in thetat
|       theta_shift-[.1] Shift in thetat for fault
|       dir - [.1] Direction of fault movement
|
|   gaussian(self, **kw)
|       Add a gaussian anomaly
|       center2 - [.5] Relative position of anomaly axis2
|       center1 - [.5] Relative position of anomaly axis1
|       center3 - [.5] Relative position of anomaly axis3
|       vplus   - [1.] Value of anomaly to add
|       var     - [.1] Relative variance of anomaly
|
|   getHyper(self)
|
|   getMinMax(self, prop)
|
|   getProp(self, prop)
|       Get model propertt
|
|   implace(self, **kw)
|       Add feature to model
|       emplace - [True] Whether or not emplace a body into the model
|       prop    - [4500.] Value to set body
|       center1,center2,center3 [.5] Relativel location of center of an
omaly
|       axis1,axis2,axis3 - [.3] Relative axes for anomaly
|       azimuth - [0.] Rotation azimuth for body
|       pctRemove - [30.] Percentage of points to remove
|       conform - [True] Conform model arroudn shape introduced
|       down_decrease - [True] Decrease below anomaly
|       down_dist - [0.] Distance down to change model
|       ntSteps- [50] Number of time steps
|       down_amount [0.] Down amount
|
|   parseParams(self, ks, typ, intM, floatM, stringM, boolM)
|       Internal function to parse parameters
```

```
 |
 |  squish(self, **kw)
 |      Squish a model
 |      aziumth - [0.] Azimuth for squishing
 |      max - [50.] Maximum shift in z
 |      wavelength- - [1.] Wavlength scaling
 |      random_inline - [.5] Random inline
 |      random_crossline - [.5] random crossline
 |
 |  ------------------------------------------------------------------
---
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```