Testing Eikonet on Cascadia Velocity Data

Lead: Barrett Johnson
Assist: Madeline Mamer

Project Motivation

- Fully and accurately resolving an earthquake's epicenter is important to...
  - Understand if earthquake was tectonic or volcanic in nature
  - Understand how Earth responds to stresses and strains
- A comparison between 1-dimensional velocity models and 3-dimensional…
  - Their ability to help pinpoint earthquake hypocenters
  - Their storage and model run time
- Test a novel neural-network based approach to solving the Eikonal equation on detecting earthquake location

Introduction to Eikonet[1]

- An approach to solving the factored Eikonal equation
- Physics-informed neural network
- Using neural networks, computes spatial gradient of the travel-time field
- Gives a continuous function output, “true 3D manner”, avoiding grids

Smith et al. (2020). Eikonet: Solving the Eikonal Equation with deep neural networks.
arXiv:2004.00361

Our Data

Cascadia 3-Dimensional Velocity Model

Free and available for download

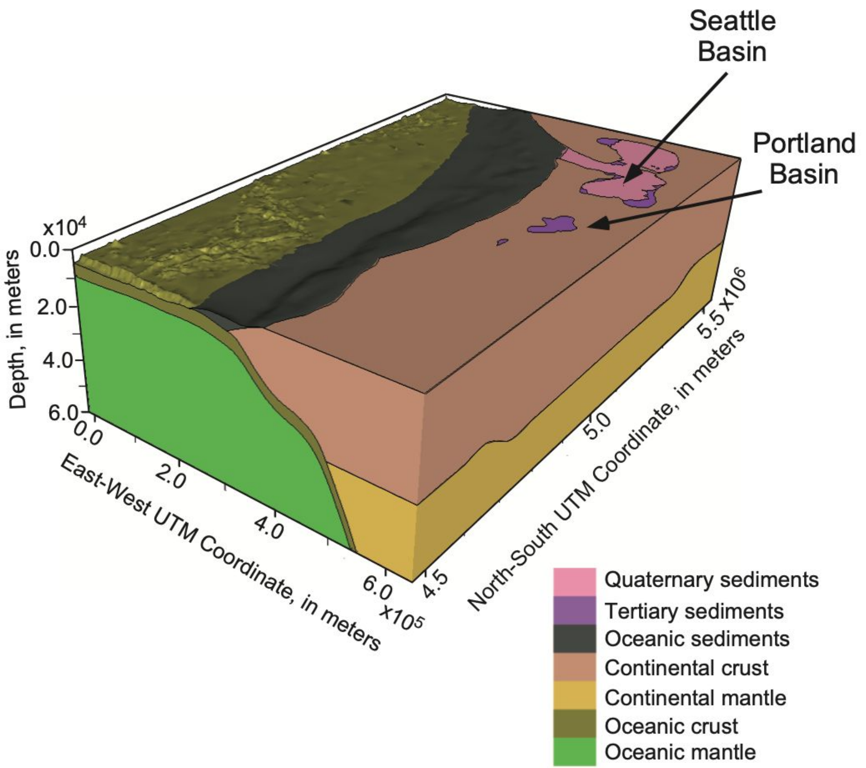
to the public

500 meter x 500 meter x 500 meter spacing

Latitude = 40.2 degrees to 50 degrees

Longitude = -129 degrees to -122 degrees

Depth 0 to 60 kilometers



Stephenson, W.J., 2017, Data for P- and S-wave Seismic Velocity Models Incorporating the Cascadia Subduction Zone for 3D Earthquake Ground Motion simulations- Update for Open-File Report 2007-1348: U.S. Geological Survey data release, https://doi.org/10.5066/F7NS0SWM.

Eikonet is contained in 3 particular scripts:

Database.py - Setting up the Model

Model.py - Trains the model (Where the CNN is located)

Plotting.py - Plotting results

A brief overview of how Eikonet works for 3-D models: database.py

Step 1: Define a three-dimensional model space using a minimum latitude, longitude and depth, and a maximum latitude, longitude and depth. Using these as boundaries, the code randomly generates 10,000 X_source,Y_source, and Z_source and 10,000 X_sreceiver,Y_receiver, and Z_receiver and pair them together.

Step 2: the code evaluates the velocity of the Source points with the provided 3-D velocity model.

Step 3: the model is trained on the Source points and using a Convolutional Neural Network, to calculate the travel-times from the source points to receiver points.

** From the source code, they do not provide a general method to define the 3-D volume and evaluate the velocity of randomly generated points. Rather, they call an outside Fortran code to interpolate their SCEC 3-D velocity model.

Our Model Runs

1. Installation of EikoNet and all of the dependencies
2. Run the examples from the Google Colab Notebook they provided.
3. Prepare the CVM - creating a submodel for testing
4. Read in the CVM to EikoNet. → Not as simple as it seems!*
5. Train the EikoNet model, with the CVM dataset.
6. Assess the performance of the algorithm.

*This required us to write a new class to be able to read in our dataset and prevented us from finishing the model runs.

```python
class SCEC_CVMH:
    def __init__(self,xmin=None,xmax=None,projection=None,phase='VP',cvm_host=None):
        self.xmin         = xmin
        self.xmax         = xmax
        self.projection   = projection

        self.FileTemp = os.getcwd()

        self.cvm_host = cvm_host
        if type(self.cvm_host) == type(None):
          print('Please specify a path to the CVM-H Fortran code')

        self.phase    = phase

    def eval(self,Xp):
        Yp = np.zeros((Xp.shape[0],2))

        print('Compute CVM-H at point locations {}'.format(len(Yp)))

        # converting back into LatLong and flattening
        proj               = Proj(self.projection)
        long_flat,lat_flat = proj(Xp[:,3],Xp[:,4],inverse=True)

        # Creating a grid of points and saving to a temp file
        Locations = pd.DataFrame({'X':long_flat,'Y':lat_flat,'Z':-Xp[:,5]*1000})
        Locations.to_csv('{}/tmp_events'.format(self.FileTemp),header=False,index=False,sep=' ')

        # Running CVM-H
        call('../src/vx < {}/tmp_events > {}/tmp_vpvs'.format(self.FileTemp,self.FileTemp),cwd='{}/model'.format(self.cvm_host),shell=True)
        VPVS = pd.read_csv('tmp_vpvs',
                                    names=['Long','Lat','Z','UTM_X','UTM_Y','UTM_elv_X','UTM_elv_Y',
                                           'topo','mtop','base','moho','flg','cellX','cellY','cellZ',
                                           'tag','VP','VS','RHO'],sep=r'\s+')

        if self.phase == 'VP':
            Yp[:,1] = VPVS['VP']
        if self.phase == 'VS':
            Yp[:,1] = VPVS['VS']

        # Removing unknown velocities
        Yp[Yp[:,1]==-99999.0000] = np.nan
        Yp[Yp[:,1]==0.0] = np.nan

        # Velocity in km/s
        Yp = Yp/1000

        return Yp
```

How we confronted this issue:

Similar to the original format, we decided it would be best to write our own class, calling it PNSN_CVM.

This was an important step in coding for us, as neither of us had written a class before this course.

The PNSN_CVM class evaluates the randomly generated points for the velocity by reading a simple .csv file where the columns are Latitude, Longitude, UTME, UTMW, Depth, P-wave velocity and S-wave velocity, for each node, or point of the 3D velocity model.

The function eval() reads the .csv as a pandas dataframe and finds the two closest nodes and takes the average velocity between the two nearest points.

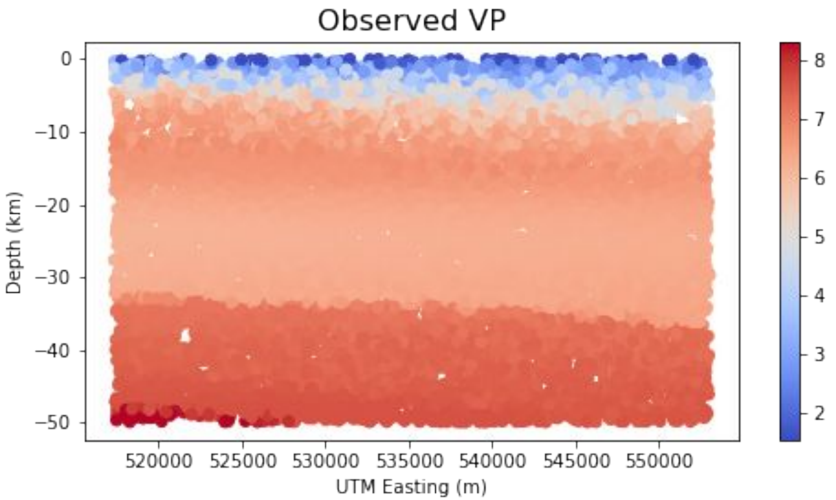
```python
class PNSN_CVM:
    def __init__(self,path,file,xmin=None,xmax=None,projection=None,phase='VP'):
        self.file       = file
        self.path       = path
        self.xmin       = xmin
        self.xmax       = xmax
        self.projection = projection
        self.phase      = phase

    def eval(self,Xp):
        df = pd.read_csv(self.path+"/"+self.file)
        df['z'] = df['z']/1000
        velocity = []
        Yp = []
        for i in range(len(Xp)):
            t = Xp[i]

            x_dep = t[5]
            sub = df.iloc[(df['z']-x_dep).abs().argsort()[:2]]
            sub_z = np.unique(sub['z'].values)
            zdf = df.iloc[(df['z'].values) == sub_z]

            x_utme = t[3]
            x_utmn = t[4]

            sub = df.iloc[(df['utmn']-x_utmn).abs().argsort()[:2]]
            sub_n = np.unique(sub['utmn'].values)

            sub = df.iloc[(df['utme']-x_utme).abs().argsort()[:2]]
            sub_e = np.unique(sub['utme'].values)

            row = zdf.loc[(zdf['utme'].values == sub_e) & (zdf['utmn'].values == sub_n)]
            if self.phase == 'VP':
                vp = row['vp'].values[0]
                velocity.append(vp)
            if self.phase == 'VS':
                vp = row['vs'].values[0]
                velocity.append(vp)
            vp = row['vp'].values[0]
            Yp.append([0,vp])
        Yp = np.asarray(Yp)

        if self.phase == 'VP':
            Yp[:,1] = VPVS['VP']
        if self.phase == 'VS':
            Yp[:,1] = VPVS['VS']

        Yp = Yp/1000

        return Yp
```

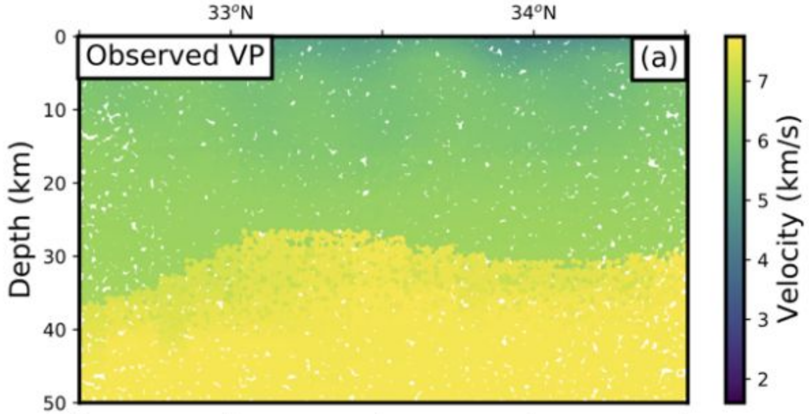
Results

The figure on the top is the result from PNSN_CVM and the figure on the bottom is the output from SCEC_CVMH.

We were able to produce a similar figure with a similar resolution, but the clarity near the top of our model isn't as well defined.

This is a success as we didn't have any insight into how they interpolate the values in their Fortran code.





Discussion

We attempted to complete the training of the model, but the amount of time it took was too long. In the text, they provided a brief explanation for the amount fo time we could expect, approximately ~16 hours. This was next to impossible on a google Colab notebook. We explored AWS, but there was a breakdown in how we set up the environment. We weren’t able to resolve it within the allocated time of the project.

Our approach is more generalizable, compared the SCEC_CVMH class, as PNSN_CVM only requires a .csv file that describes the 3-D velocity model of your choice.

Our program does take ~20 minutes to evaluate the velocities for the randomly generated points, but we can’t compare this time to their times as they didn’t explicitly state the amount of time required for their Fortran code to run.

Reproducibility and Adaptability of EikoNet

- Their model’s colab is documented well enough specifically for their own data
  - Running and getting an idea of what the model does is simple
- 3 main scripts used for running the model are not well commented - at all
  - manipulating/making tweaks to the database/type of velocity model used is difficult
- Their example script downloads the model output, rather than running the model and displaying the output
  - no idea what the time budget is for training and predicting on some data
  - Don’t actually get to see the workflow of the model itself, just what it produces