Seismic Processing
Prac 1 - Building a Synthetic Record.

ERTH3021

October 6, 2014

This is the first of two pracs on seismic data processing. This prac is dedicated to building a synthetic 2D seismic dataset. The primary motivation for building a synthetic dataset for processing is to ensure we know what the answer is before we start, and thus can assess the effectiveness of our data processing.

The second prac will involve processing the dataset created in this prac.

1 Introduction

The core concept used to create this synthetic model is known as the convolutional model. The convolutional model states that a recorded signal is the convolution of a source wavelet with the earth's response, convolved with the recorder response, plus some additional noise, i.e.

$$Y(t) = S(t) * E(t) * R(t) + N(t)$$

where

- Y(t) is our recorded signal
- S(t) is the source wavelet
- E(t) is the earth's response
- R(t) is the recorder response
- N(t) is some noise
- * is the convolutional operator

We are going to break the earth's response $E(t)$ into 3 main components -

- A(t) - the direct wave
- B(t) - the refracted wave
- C(t) - the reflected wave

and we are going to ignore the recorder response for this prac. Thus our synthetic signal can be described as

$$Y(t) = [A(t) + B(t) + C(t)] * S(t) + N(t)$$

Each component described above will be addressed as a separate exercise.

This prac uses python as a teaching tool. The entire prac consists of several hundred lines of code. A significant proportion of this code has been supplied. This supplied code uses some advanced processing techniques, for example classes and decorators. The main reason for this is to reduce the amount of boilerplate code. Understanding this part of the code is not required for this prac.

The assessment of this prac will take the form of a brief report. This report should include a summary of the main components of the prac (suggested with an asterisk), as well as screen shots from each exercise. A brief paragraph and/or bullet points which shows understanding of the major concepts is sufficient.

Exercise 1 - Initial Setup

1. *Load and view earth model
2. Initialise parameter dictionary
3. Initialise data workspace
4. *Define survey geometry
5. Load initialisation values
6. *Write test signal
7. *View result.

Exercise 2: Direct Wave

The direct wave travels along the earth/air interface, and can thus be calculated from the velocity formula

$$v = \frac{s}{t}$$

where

- v = velocity
- s = displacement
- t = time

Spherical divergence is the idea that as a wave spreads out, the energy in the wave spreads out over the surface of the waveform. The amplitude of the wave is inversely proportional to the square of the distance traveled, i.e.

$$A = \frac{1}{distance^2}$$

1. *Create a function which calculates the direct travel time, given a velocity and distance
2. *Create a function which calculates the spherical divergence weighting, given a distance
3. Apply weights to traveltimes
4. Write valid results to workspace
5. Display Result
6. Apply AGC
7. *Display result with AGC

Exercise 3: Refracted Wave

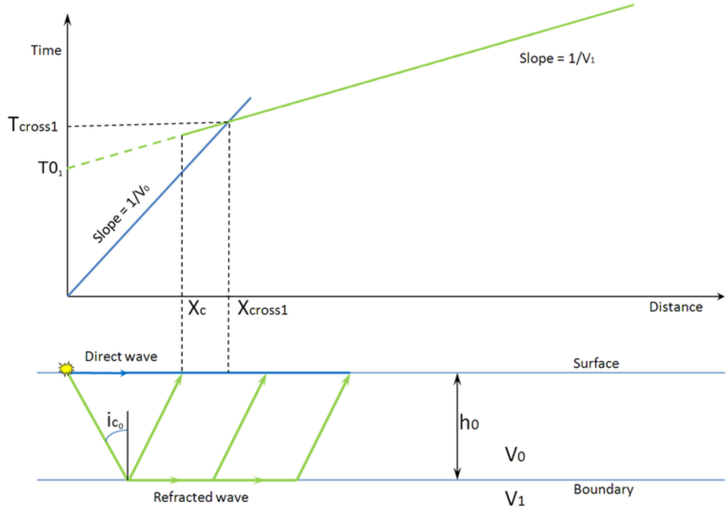

Figure 1: Calculating refraction travel-time, image from wikipedia

The formula to calculate the refracted travel time is

$$T = \frac{X}{V_1} + \frac{2z \cos i_c}{V_0}, \quad i_c = \sin^{-1} \frac{V_0}{V_1}$$

Where

- X = lateral distance
- V_0 = velocity of weathering layer
- V_1 = velocity of sub-weathering layer
- z = thickness of weathering layer
- i_c = critical angle

The exercise consists of the following steps:

1. *Create a function which calculates the refracted travel time
2. Apply spherical divergence to travel times
3. Write valid results to dataset
4. *Display result

Exercise 4: Reflected Wave

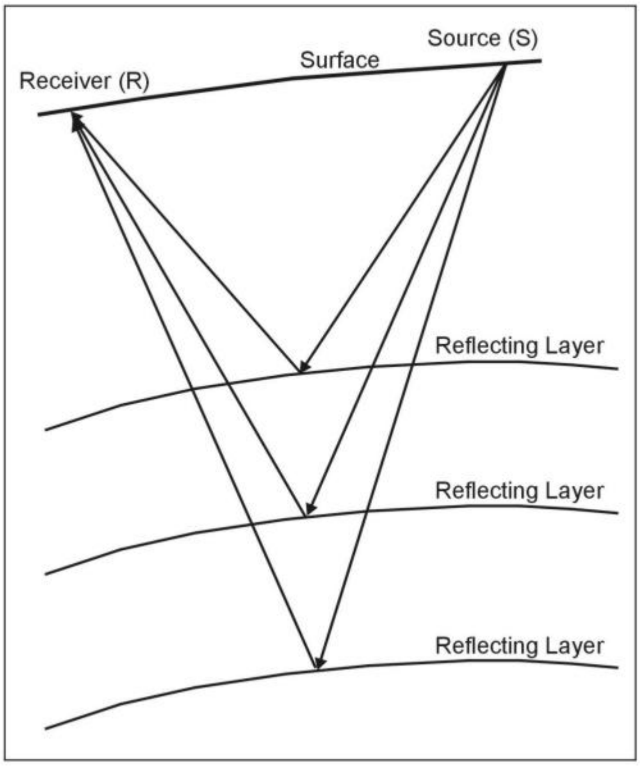

Figure 2: Calculating reflection travel-time, image from the U.S. EPA website

Calculating the reflection times in a homogeneous earth is relatively simple. Calculating the travel time in an in-homogeneous earth is less simple. Functions have been provided which will perform most of the hard lifting for this.

Calculating amplitudes can also be complex. A relatively accurate approximation might involve the Aki-Richards equations seen on the Crewes Zeoppritz Explorer. Instead, for this exercise, we will use the zero-offset reflection and transmission coefficients.

$$R_r = \frac{z_1 - z_0}{z_1 + z_0}$$

$$R_t = \frac{2 * z_0}{z_1 + z_0}$$

where

- z_0 = acoustic contrast in layer 0, i.e. $\rho_0 v_0$
- z_1 = acoustic contrast in layer 1, i.e. $\rho_1 v_1$

This exercise will include the following steps:

1. *Implement functions to calculate reflection and transmission coefficients
2. *Discuss geometry of calculations
3. Calculate reflection traveltimes (using supplied point extrapolation routine)
4. Calculate transmission amplitudes (using supplied point extrapolation routine)
5. Calculate reflection amplitudes
6. Write result to dataset
7. *Display Result

Exercise 5: Build Combined Shot-record

1. *Write function which combines exercises 2, 3 & 4.
2. *Convolve with wavelet
3. *Display result
4. *Add noise
5. *Display result

Exercise 6: Build Complete Survey

Using the tools created in exercises 1 to 5, build a complete synthetic seismic survey

```python
from toolbox import io
import toolbox
import numpy as np

#-----------------------------------------------------------------------
#                useful functions
#-----------------------------------------------------------------------

@io
def spike(dataset, **kwargs):
  dataset['trace'][:,500] = 1
  return dataset

#-----------------------------------------------------------------------
#                main functions
#-----------------------------------------------------------------------

def initialise():
  parameters = {}
  #build our model, which is pre-defined in the toolbox
  parameters['model'] = toolbox.build_model()
  #have a look at it - it has a build in display routine
  #~ parameters['model'].display()

  #initialise data workspace
  parameters['sutype'] = toolbox.typeSU(1000)
  workspace = np.zeros(500, dtype=parameters['sutype'])

  #define survey geometry, ie shot and reciever points
  parameters['sx_coords'] = np.arange(500.0)[::2] + 1
  workspace['gx'] = np.arange(500.0)+1
  parameters['gx'] = np.arange(500.0)

  #add some more useful stuff
  workspace['ns'] = 1000
  workspace['dt'] = 1000 #* 1e-6
  parameters['dt'] = 1e-3
  parameters['sz'] = 0
  parameters['gz'] = 0
  parameters['nx'] = workspace['gx'].size
  return workspace, parameters

if __name__ == '__main__':
  workspace, param = initialise()
  spike(workspace, None, **param)
  toolbox.display(workspace, None, **param)
```

../exersize1.py

```python
# in prac 1 we will build a synthetic shot record.
# it will compose of 3 separate components
# direct wave
# refracted wave
# reflected wave
# based up on a predefined model.

from toolbox import io
import toolbox
import numpy as np
import matplotlib.pyplot as pylab
from exersize1 import initialise

#-----------------------------------------------------------------------
#                useful functions
#-----------------------------------------------------------------------
def diverge(distance, coefficient=3.0):
  '''spherical divergence correction'''
  r = np.abs(1.0/(distance**coefficient))
  return r

def direct(distance, velocity):
  time = distance/velocity
  return time

```

```
#-----------------------------------------------------------------------
#                main functions
#-----------------------------------------------------------------------

@io
def build_direct(dataset, **kwargs):
    '''
    calculates direct wave arrival time and
    imposes it upon an array. assumes 330 m/s
    surface velocity
    '''

    directv = 330.0 #m/s
    direct_times = direct(kwargs['aoffsets'], directv)

    #set base amplitude (from testing)
    direct_amps = np.ones_like(kwargs['gx']) * 0.005
    #calculate the spherical divergence correction
    direct_correction = diverge(kwargs['aoffsets'], 2.0)
    #apply correction
    direct_amps *= direct_correction
    direct_amps[~np.isfinite(direct_amps)] = 0.01

    #we are not interested in anything after 1 second
    limits = [direct_times < 1]
    x = kwargs['gx'][limits]
    t = direct_times[limits]
    direct_amps = direct_amps[limits]



    #convert to coordinates
    t *= 1000 # milliseconds
    x = np.floor(x).astype(np.int)
    t = np.floor(t).astype(np.int)

    dataset['trace'][x, t] += direct_amps
    return dataset



if __name__ == '__main__':
    workspace, param = initialise()

    sx = 100

    workspace['sx'] = sx
    workspace['offset'] = workspace['gx'] - workspace['sx']
    param['aoffsets'] = np.abs(workspace['offset'])


    #lets set up for calculating direct wave
    build_direct(workspace, None, **param)
    toolbox.agc(workspace, None, **param)
    toolbox.display(workspace, None, **param)
```

../exersize2.py

```
# in prac 1 we will build a synthetic shot record.
# it will compose of 3 separate components
# direct wave
# refracted wave
# reflected wave
# based up on a predefined model.

from toolbox import io
import toolbox
import numpy as np
import matplotlib.pyplot as pylab
from exersize1 import initialise
from exersize2 import diverge


#-----------------------------------------------------------------------
#                useful functions
#-----------------------------------------------------------------------
```

```python

def refract(x, v0, v1, z0):
    ic = np.arcsin(v0/v1)
    t0 = 2.0*z0*np.cos(ic)/v0
    t = t0 + x/v1
    return t

#-----------------------------------------------------------------------------
#                main functions
#-----------------------------------------------------------------------------

@io
def build_refractor(dataset, **kwargs):
    '''
    builds refractor
    '''

    #some shortcuts
    v0 = kwargs['model']['vp'][0]
    v1 = kwargs['model']['vp'][1]
    z0 = kwargs['model']['dz'][0]

    refraction_times = refract(kwargs['aoffsets'], v0, v1, z0)

    #create amplitude array
    refract_amps = np.ones_like(kwargs['gx']) * 0.01
    #calculate the spherical divergence correction
    refract_correction = diverge(kwargs['aoffsets'], 2.0)
    #apply correction
    refract_amps *= refract_correction
    refract_amps[~np.isfinite(refract_amps)] = 0.01

    #it probably wont exceed 1s, but to make it look right we
    #need to limit it so that it doesnt cross over the direct
    directv = 330.0 #m/s
    direct_times = kwargs['aoffsets']/directv
    limits = [refraction_times < direct_times]
    x = kwargs['gx'][limits]
    t = refraction_times[limits]
    refract_amps = refract_amps[limits]

    #convert coordinates to integers
    x = np.floor(x).astype(np.int)
    t *= 1000 # milliseconds
    t = np.floor(t).astype(np.int)

    dataset['trace'][x, t] += refract_amps
    return dataset



if __name__ == '__main__':
    workspace, param = initialise()

    sx = 100

    workspace['sx'] = sx
    workspace['offset'] = workspace['gx'] - workspace['sx']
    param['aoffsets'] = np.abs(workspace['offset'])


    build_refractor(workspace, 'refractor.su', **param)
    tmp = toolbox.agc('refractor.su', None, **param)
    toolbox.display(tmp, None, **param)
```

../exersize3.py

```python
# in prac 1 we will build a synthetic shot record.
# it will compose of 3 separate components
# direct wave
# refracted wave
# reflected wave
# based up on a predefined model.

from toolbox import io
import toolbox
```

```python
import numpy as np
import matplotlib.pyplot as pylab
from exersize1 import initialise
from exersize2 import diverge


#------------------------------------------------------------------------
#               useful functions
#------------------------------------------------------------------------

def reflection_coefficient(z0, z1):
    z = ((z1 - z0))/(z1+z0)
    return z

def transmission_coefficient(z0, z1):
    r = (2.0*z0)/((z1+z0))
    return r

#------------------------------------------------------------------------
#               main functions
#------------------------------------------------------------------------

@io
def build_reflector(dataset, **kwargs):
  '''
  builds reflector
  '''

  #some shortcuts
  vp = kwargs['model']['model']['vp']
  rho = kwargs['model']['model']['rho']
  R = kwargs['model']['model']['R']
  sz = kwargs['sz']
  gz = kwargs['gz']
  sx = kwargs['sx']

  numpoints = 100 #used for interpolating through the model
  for gx in dataset['gx']:
    cmpx = np.floor((gx + sx)/2.).astype(np.int) # nearest midpoint
    h = cmpx - sx #half offset
    #the next line extracts the non-zero reflection points at this midpoint
    #and iterates over them

    for cmpz in (np.nonzero(R[cmpx,:])[0]):
      ds = np.sqrt(cmpz**2 + (h)**2)/float(numpoints) # line step distance
      #predefine outputs
      amp = 1.0
      time = 0.0

      #traveltime from source to cdp
      vp_down = toolbox.find_points(sx, sz, cmpx, cmpz, numpoints, vp)
      time += np.sum(ds/vp_down)

      #traveltime from cdp to geophone
      vp_up = toolbox.find_points(cmpx, cmpz, gx-1, gz, numpoints, vp)
      time += np.sum(ds/vp_up)

      #loss due to spherical divergence
      amp *= diverge(ds*numpoints, 3)#two way

      #~ #transmission losses from source to cdp
      rho_down = toolbox.find_points(sx, sz, cmpx, cmpz, numpoints, rho)
      z0s = rho_down * vp_down
      z1s = toolbox.roll(z0s, 1)
      correction = np.cumprod(transmission_coefficient(z0s, z1s) )[-1]
      amp *= correction

      #amplitude loss at reflection point
      correction = R[cmpx,cmpz]
      amp *= correction
      #transmission loss from cdp to source
      rho_up = toolbox.find_points(cmpx, cmpz, gx-1, gz, numpoints, rho)
      z0s = rho_up * vp_up
      z1s = toolbox.roll(z0s, 1)
      correction = np.cumprod(transmission_coefficient(z0s, z1s))[-1]
      amp *= correction
```

```
        x = np.floor(gx).astype(np.int) -1
        t = np.floor(time*1000).astype(np.int)

        dataset['trace'][x, t] += amp
    return dataset



if __name__ == '__main__':
    workspace, param = initialise()

    sx = 100

    workspace['sx'] = sx
    param['sx'] = sx
    workspace['offset'] = workspace['gx'] - workspace['sx']
    param['aoffsets'] = np.abs(workspace['offset'])


    build_reflector(workspace, 'reflector.su', **param)
    tmp = toolbox.agc('reflector.su', None, **param)
    toolbox.display(tmp, None, **param)
```

../exersize4.py

```
from toolbox import io
import toolbox
import numpy as np
import matplotlib.pyplot as pylab
from exersize1 import initialise
from exersize2 import  build_direct
from exersize3 import build_refractor
from exersize4 import build_reflector

#-----------------------------------------------------------------------
#               useful functions
#-----------------------------------------------------------------------

@io
def build_combined(dataset, **kwargs):
    dataset = build_direct(dataset, None, **kwargs)
    dataset = build_refractor(dataset, None, **kwargs)
    dataset = build_reflector(dataset, None, **kwargs)
    return dataset

@io
def add_noise(dataset, **kwargs):
    noise = np.random.normal(0.0, 1e-8, size=(dataset['trace'].shape))
    dataset['trace'] += noise
    return dataset

@io
def convolve_wavelet(dataset, **kwargs):
    wavelet = toolbox.ricker(60)
    dataset =  toolbox.conv(dataset, wavelet)
    return dataset

if __name__ == '__main__':
    workspace, param = initialise()

    sx = 100

    workspace['sx'] = sx
    param['sx'] = sx
    workspace['offset'] = workspace['gx'] - workspace['sx']
    param['aoffsets'] = np.abs(workspace['offset'])

    #build record
    workspace = build_combined(workspace, None, **param)

    #build wavelet
    convolve_wavelet(workspace, 'test.su', **param)

    workspace = toolbox.agc(workspace, None, **param)
    #~ workspace = add_noise(workspace, 'record.su', **param)
```

```
  toolbox.display(workspace, None, **param)
```

../exersize5.py

```
from toolbox import io
import toolbox
import numpy as np
import matplotlib.pyplot as pylab
from exersize1 import initialise
from exersize5 import build_combined, add_noise, convolve_wavelet

if __name__ == "__main__":
  workspace, param = initialise()

  for sx in param['sx_coords']:
    print sx
    workspace['sx'] = sx
    param['sx'] = sx
    workspace['offset'] = workspace['gx'] - workspace['sx']
    param['aoffsets'] = np.abs(workspace['offset'])
    workspace['trace'].fill(0)
    workspace = build_combined(workspace, None, **param)
    workspace = convolve_wavelet(workspace, None, **param)
    workspace = add_noise(workspace, None, **param)
    #need to add cdp locations to workspace
    toolbox.cp(workspace, 'shot%d.su' %sx, **param)
```

../exersize6.py