Ouermi Timbwaoga Aime Judicael
Course: Phys 410 Scien Prog
Project 4
Collaborator : Samenthat Mellin
Date: 05/07/2015

## Introduction

For this assignment we have decided to use the terminal as interface first and once we have completed the assignment, we'll then build the graphical user interface. For this assignment I also collaborated with Sametha Mellin.

## Understanding the data

**There are six columns of information:**

- **Column 1 a technique descriptor which you do not need**
- **Column 2 year of occurrence**
- **Column 3,4 is lat/long**
- **Column 5 is depth under the surface; 0 indicates no data**
- **Column 6 richter scale magnitude of the quake**

We first looked at the data and noticed that we did not need column 1 and 2 for our calculation and in order to obtain the correct data we must remove the first 2 columns.
remove first column

```
awk '{$1=""; print $0;}' quake.txt > quake1.txt
```

Remove second column

```
sed s/[a-Z]//g quake1.txt > quake_data.txt
sed s/[A-Z]//g quake1.txt > quake_data.txt
```

## 1.Event Counter

**An event counter interface where the use enters latitude, longtiude, begin year, end year, and magnitude threshold and the counter returns the values. While you are strongly encouraged to make some kind of GUI inteface for these queries, you don't have too. As an alternative you can enter in parameters on a command line to some python program for sorting and printing. An example of this will be shown in class on Friday May 1.**

In here we want the number of earth given the parameters above and those parameters are provided by the user.

**Provide a sample output of your table that contains earthquake occurences within a radius of 100 miles from Seattle Washington. Note that negative longitudes (like Seattle would be) represent those longitudes WEST of Greenwich.**

In this portion of the problem we talked about different ways to approach it. First, we could approximate the area needed to a square of side 100 miles. Second, we could find the could find the equation of the circle of radius 100 miles and center seattle. Third we could calculate the distance between the point we looking at and seattle and if the that distance is less that 100 miles we consider it for our calculations.

I decided to go with the third option here

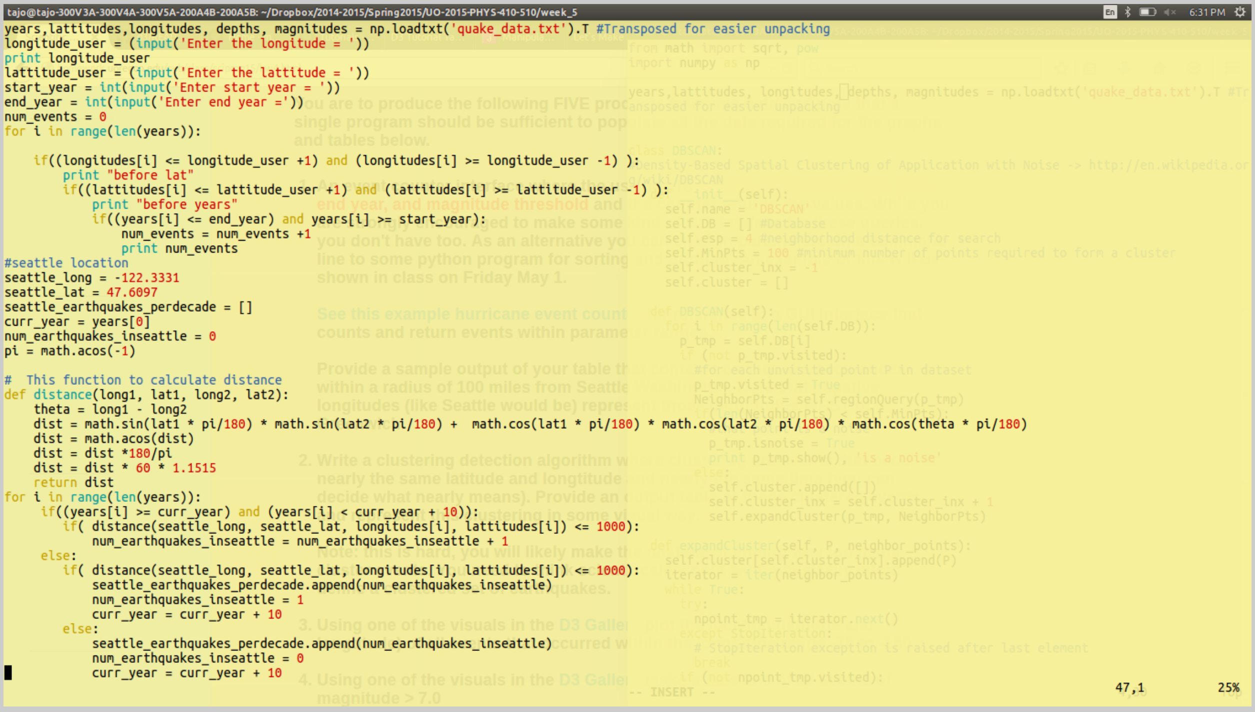
Seattle : 47.6097° N, 122.3331° W = 47.6097, -122.3331

Number of earthquakes per decade (1900 -2008) in Seattle

[ 3,  7,  6,  1,  3,  9,  12,  30,  31,  34]

Looking at this we can observe that the number of earthquakes per decade in Seattle has been increasing overall.

Code for question 1 (question_1.py)



tajo@tajo-300V3A-300V4A-300V5A-200A4B-200A5B: ~/Dropbox/2014-2015/Spring2015/UO-2015-PHYS-410-510/week_5

```
years,lattitudes,longitudes, depths, magnitudes = np.loadtxt('quake_data.txt').T #Transposed for easier unpacking
longitude_user = (input('Enter the longitude = '))
print longitude_user
lattitude_user = (input('Enter the lattitude = '))
start_year = int(input('Enter start year = '))
end_year = int(input('Enter end year ='))
num_events = 0
for i in range(len(years)):

    if((longitudes[i] <= longitude_user +1) and (longitudes[i] >= longitude_user -1) ):
        print "before lat"
        if((lattitudes[i] <= lattitude_user +1) and (lattitudes[i] >= lattitude_user -1) ):
            print "before years"
            if((years[i] <= end_year) and years[i] >= start_year):
                num_events = num_events +1
                print num_events
#seattle location
seattle_long = -122.3331
seattle_lat = 47.6097
seattle_earthquakes_perdecade = []
curr_year = years[0]
num_earthquakes_inseattle = 0
pi = math.acos(-1)

#  This function to calculate distance
def distance(long1, lat1, long2, lat2):
    theta = long1 - long2
    dist = math.sin(lat1 * pi/180) * math.sin(lat2 * pi/180) +  math.cos(lat1 * pi/180) * math.cos(lat2 * pi/180) * math.cos(theta * pi/180)
    dist = math.acos(dist)
    dist = dist *180/pi
    dist = dist * 60 * 1.1515
    return dist
for i in range(len(years)):
     if((years[i] >= curr_year) and (years[i] < curr_year + 10)):
        if( distance(seattle_long, seattle_lat, longitudes[i], lattitudes[i]) <= 1000):
            num_earthquakes_inseattle = num_earthquakes_inseattle + 1
     else:
        if( distance(seattle_long, seattle_lat, longitudes[i], lattitudes[i]) <= 1000):
            seattle_earthquakes_perdecade.append(num_earthquakes_inseattle)
            num_earthquakes_inseattle = 1
            curr_year = curr_year + 10
        else:
            seattle_earthquakes_perdecade.append(num_earthquakes_inseattle)
            num_earthquakes_inseattle = 0
            curr_year = curr_year + 10
```

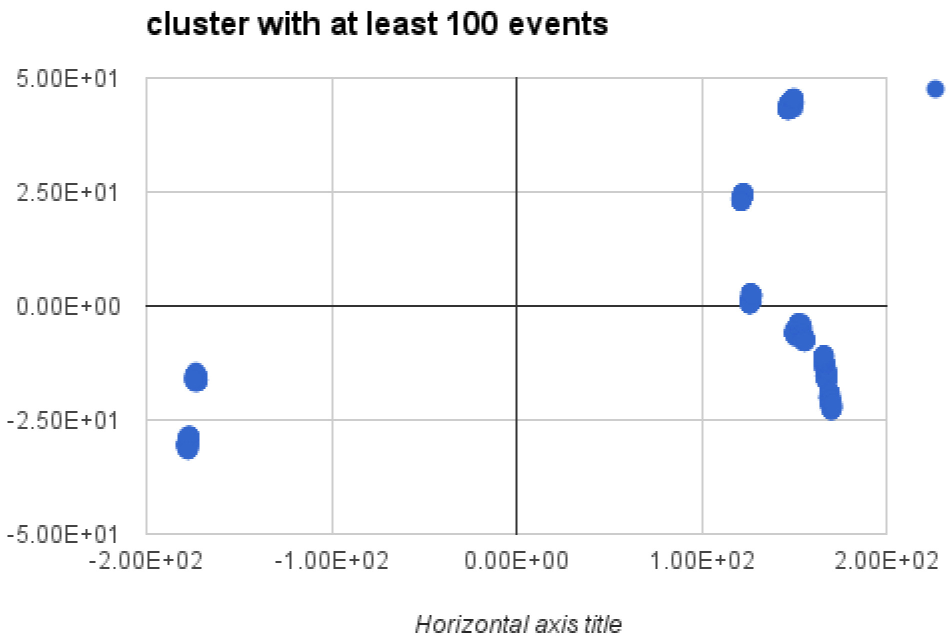
## 2. Cluster Detection

**Write a clustering detection algorithm where clustering are events that occur at nearly the same latitude and longtitude and nearly the same time (you can decide what nearly means). Provide an output table of those clustered events and represent this clustering in some visual way.**

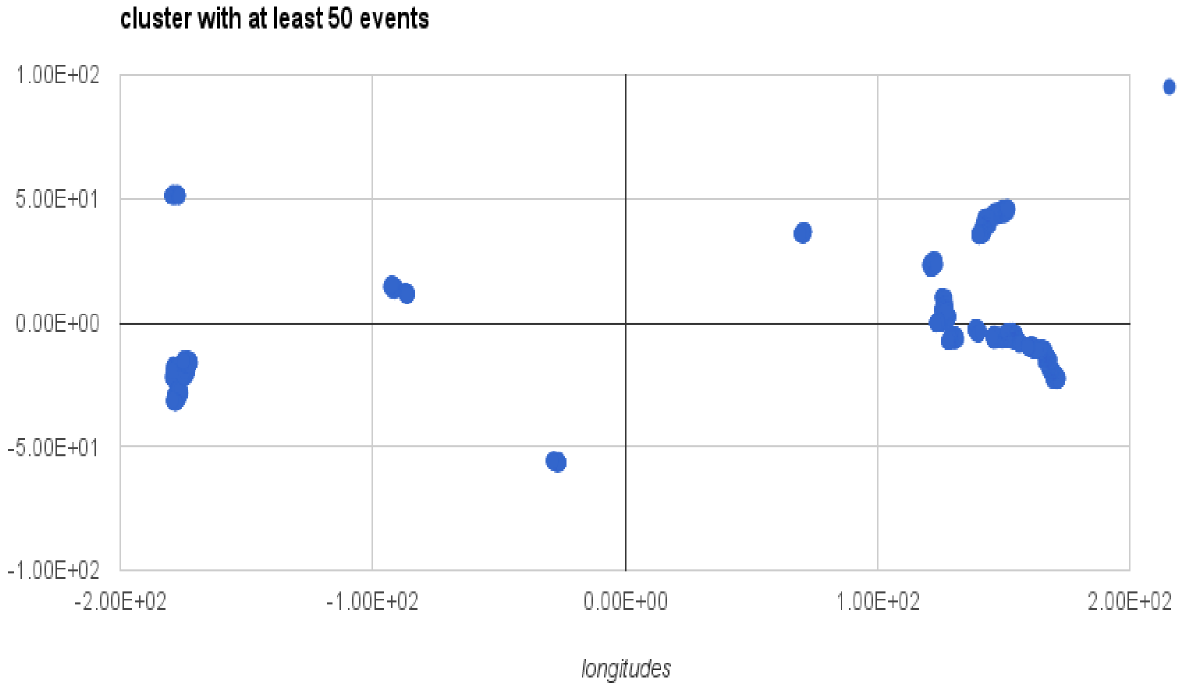
**Note: this is hard, you will likely make the mistake of finding way too many cluster events. You need to think scientifically about what physically might define a clustered set of earthquakes.**

I spent a lot of time learning the differences between the different clustering method and trying to identify which one is the most suited in our situation. The K-means would not work well because it assumes that you already know the number of cluster that exist. Here we are going to use **D**ensity-**B**ased **S**patial **C**lustering of **A**pplication with **N**oise also know as DBSCAN.
What makes a cluster? proximity and number of events that occurred within that proximity.

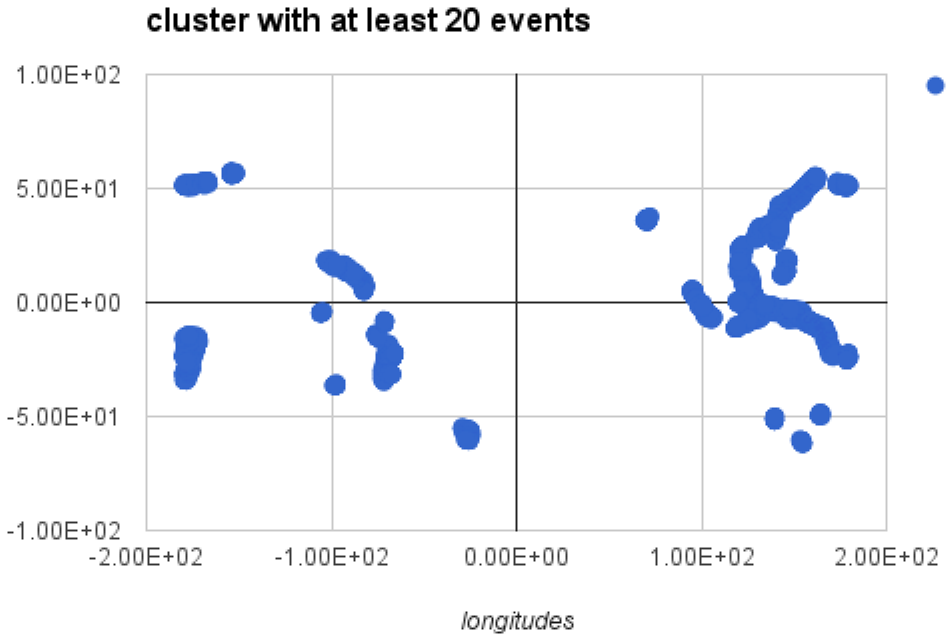
Cluster radius = 1 (used degrees for measurements)
Number of events = 100
With the setting above I get 7 clusters



Cluster radius = 1 (used degrees for measurements)
Number of events = 50
With this setting I get 21 clusters



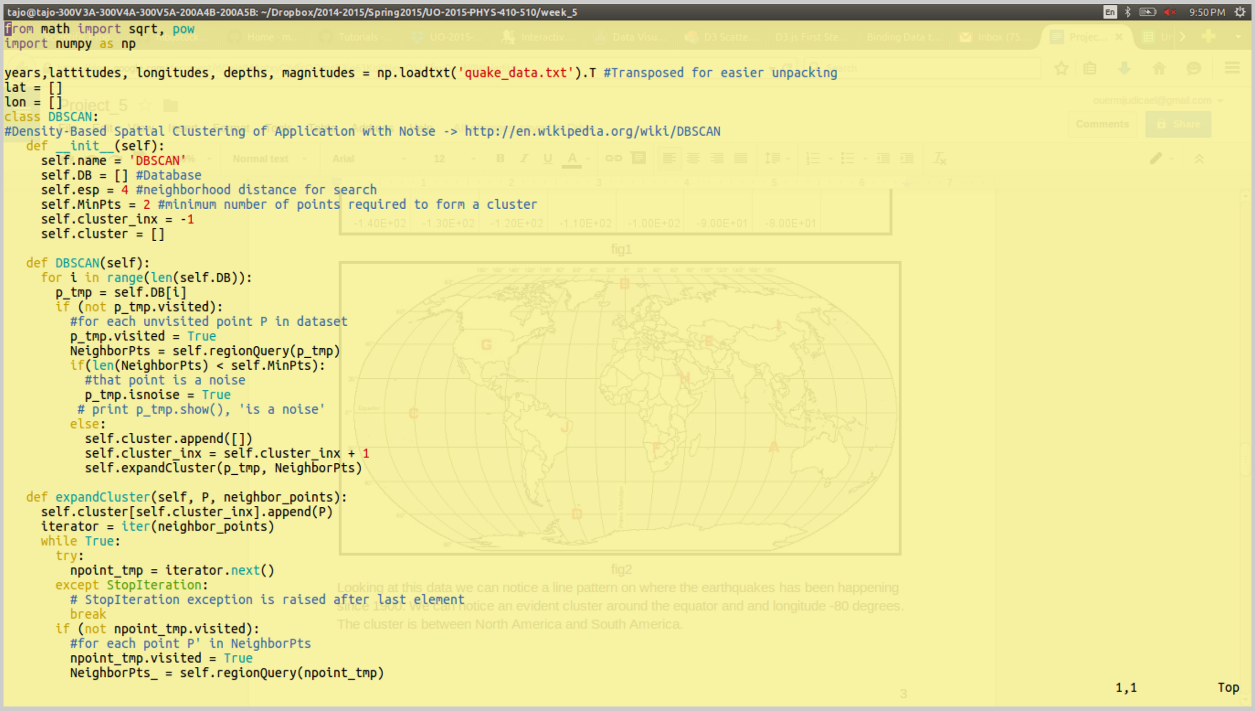
Cluster radius = 1 (used degrees for measurements)
Number of events = 50
we get 51 clusters



Our clustering algorithm is slow. it goes through the same data multiple time to figure out it belongs to a cluster or is just noise.

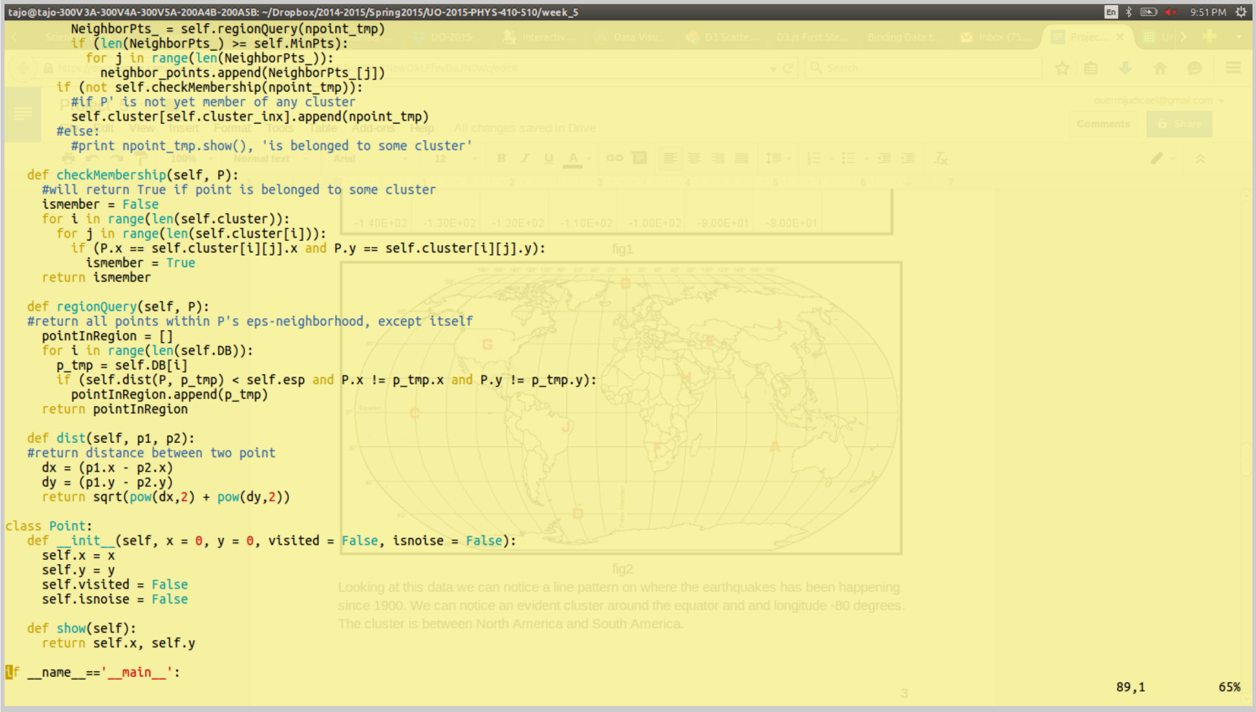
Looking at the data above which tells us where our defined clusters, we can say that it is definitely hard to decide on what makes a good definition of a cluster. We would hae to repeat the same scenario and limit ourselves to features of interest.

Code fro question 2 (question_2.py)

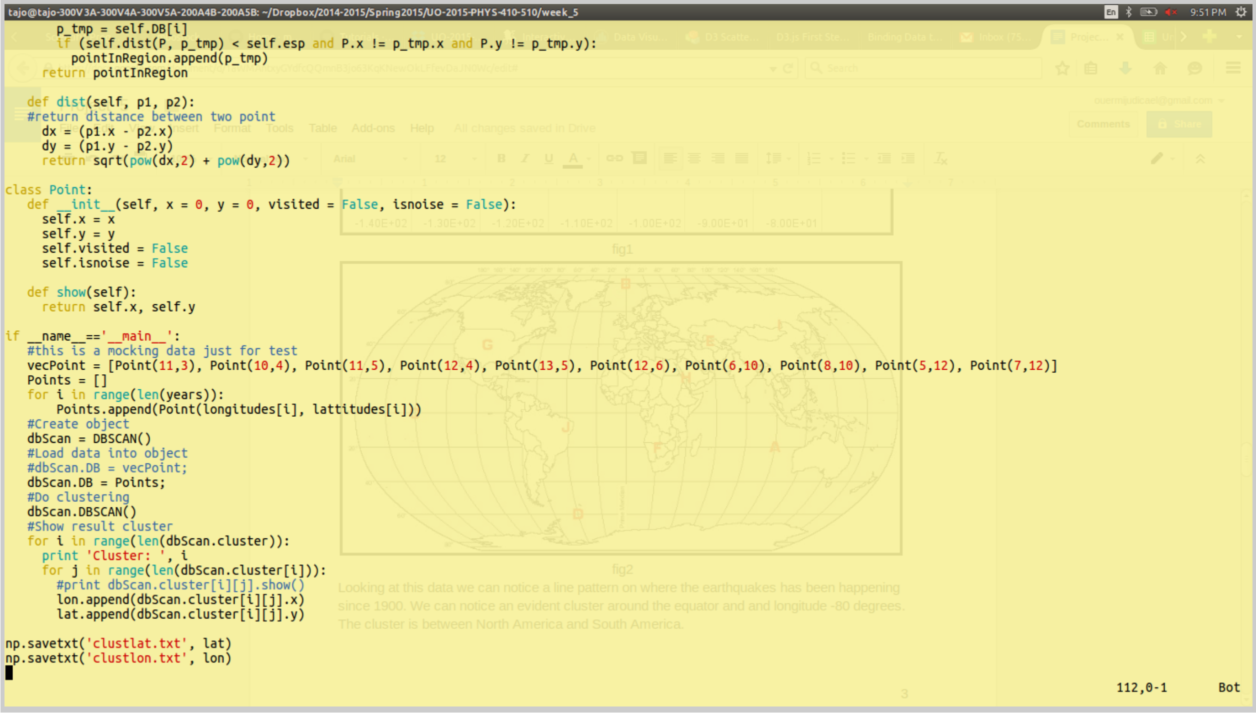


tajo@tajo-300V3A-300V4A-300V5A-200A4B-200A5B: ~/Dropbox/2014-2015/Spring2015/UO-2015-PHYS-410-510/week_5

```
from math import sqrt, pow
import numpy as np

years,lattitudes, longitudes, depths, magnitudes = np.loadtxt('quake_data.txt').T #Transposed for easier unpacking
lat = []
lon = []
class DBSCAN:
#Density-Based Spatial Clustering of Application with Noise -> http://en.wikipedia.org/wiki/DBSCAN
   def __init__(self):
     self.name = 'DBSCAN'
     self.DB = [] #Database
     self.esp = 4 #neighborhood distance for search
     self.MinPts = 2 #minimum number of points required to form a cluster
     self.cluster_inx = -1
     self.cluster = []

   def DBSCAN(self):
     for i in range(len(self.DB)):
       p_tmp = self.DB[i]
       if (not p_tmp.visited):
         #for each unvisited point P in dataset
         p_tmp.visited = True
         NeighborPts = self.regionQuery(p_tmp)
         if(len(NeighborPts) < self.MinPts):
           #that point is a noise
           p_tmp.isnoise = True
          # print p_tmp.show(), 'is a noise'
         else:
           self.cluster.append([])
           self.cluster_inx = self.cluster_inx + 1
           self.expandCluster(p_tmp, NeighborPts)

   def expandCluster(self, P, neighbor_points):
     self.cluster[self.cluster_inx].append(P)
     iterator = iter(neighbor_points)
     while True:
       try:
         npoint_tmp = iterator.next()
       except StopIteration:
         # StopIteration exception is raised after last element
         break
       if (not npoint_tmp.visited):
         #for each point P' in NeighborPts
         npoint_tmp.visited = True
         NeighborPts_ = self.regionQuery(npoint_tmp)
```



```
tajo@tajo-300V3A-300V4A-300V5A-200A4B-200A5B: ~/Dropbox/2014-2015/Spring2015/UO-2015-PHYS-410-510/week_5
        NeighborPts_ = self.regionQuery(npoint_tmp)
        if (len(NeighborPts_) >= self.MinPts):
          for j in range(len(NeighborPts_)):
            neighbor_points.append(NeighborPts_[j])
      if (not self.checkMembership(npoint_tmp)):
        #if P' is not yet member of any cluster
        self.cluster[self.cluster_inx].append(npoint_tmp)
      #else:
        #print npoint_tmp.show(), 'is belonged to some cluster'

  def checkMembership(self, P):
    #will return True if point is belonged to some cluster
    ismember = False
    for i in range(len(self.cluster)):
      for j in range(len(self.cluster[i])):
        if (P.x == self.cluster[i][j].x and P.y == self.cluster[i][j].y):
          ismember = True
    return ismember

  def regionQuery(self, P):
  #return all points within P's eps-neighborhood, except itself
    pointInRegion = []
    for i in range(len(self.DB)):
      p_tmp = self.DB[i]
      if (self.dist(P, p_tmp) < self.esp and P.x != p_tmp.x and P.y != p_tmp.y):
        pointInRegion.append(p_tmp)
    return pointInRegion

  def dist(self, p1, p2):
  #return distance between two point
    dx = (p1.x - p2.x)
    dy = (p1.y - p2.y)
    return sqrt(pow(dx,2) + pow(dy,2))

class Point:
  def __init__(self, x = 0, y = 0, visited = False, isnoise = False):
    self.x = x
    self.y = y
    self.visited = False
    self.isnoise = False

  def show(self):
    return self.x, self.y

if __name__=='__main__':
                                                                                  89,1          65%
```



```
tajo@tajo-300V3A-300V4A-300V5A-200A4B-200A5B: ~/Dropbox/2014-2015/Spring2015/UO-2015-PHYS-410-510/week_5
      p_tmp = self.DB[i]
      if (self.dist(P, p_tmp) < self.esp and P.x != p_tmp.x and P.y != p_tmp.y):
        pointInRegion.append(p_tmp)
    return pointInRegion

  def dist(self, p1, p2):
  #return distance between two point
    dx = (p1.x - p2.x)
    dy = (p1.y - p2.y)
    return sqrt(pow(dx,2) + pow(dy,2))

class Point:
  def __init__(self, x = 0, y = 0, visited = False, isnoise = False):
    self.x = x
    self.y = y
    self.visited = False
    self.isnoise = False

  def show(self):
    return self.x, self.y

if __name__=='__main__':
  #this is a mocking data just for test
  vecPoint = [Point(11,3), Point(10,4), Point(11,5), Point(12,4), Point(13,5), Point(12,6), Point(6,10), Point(8,10), Point(5,12), Point(7,12)]
  Points = []
  for i in range(len(years)):
      Points.append(Point(longitudes[i], lattitudes[i]))
  #Create object
  dbScan = DBSCAN()
  #Load data into object
  #dbScan.DB = vecPoint;
  dbScan.DB = Points;
  #Do clustering
  dbScan.DBSCAN()
  #Show result cluster
  for i in range(len(dbScan.cluster)):
    print 'Cluster: ', i
    for j in range(len(dbScan.cluster[i])):
      #print dbScan.cluster[i][j].show()
      lon.append(dbScan.cluster[i][j].x)
      lat.append(dbScan.cluster[i][j].y)

np.savetxt('clustlat.txt', lat)
np.savetxt('clustlon.txt', lon)
                                                                                  112,0-1       Bot
```

## 3. Event Counting with longitude range

**Using one of the visuals in the D3 Gallery plot the location (latitude and longtitude) of all events that occurred within the longtitude range -75 to -150.**

For this section we'll just reuse the code from question 1 and modify it slightly to get the information we need. Once we acquired the correct data we can use the D3 Gallery to plot our result.

I spent a considerable amount of time trying to figure out D3 works. Due to the time constraint I was facing I decided to use different tools for the moment and learn how to use D3 Gallery for next assignment. Once the data collected we can produce the following plots of **latitudes vs longitudes**

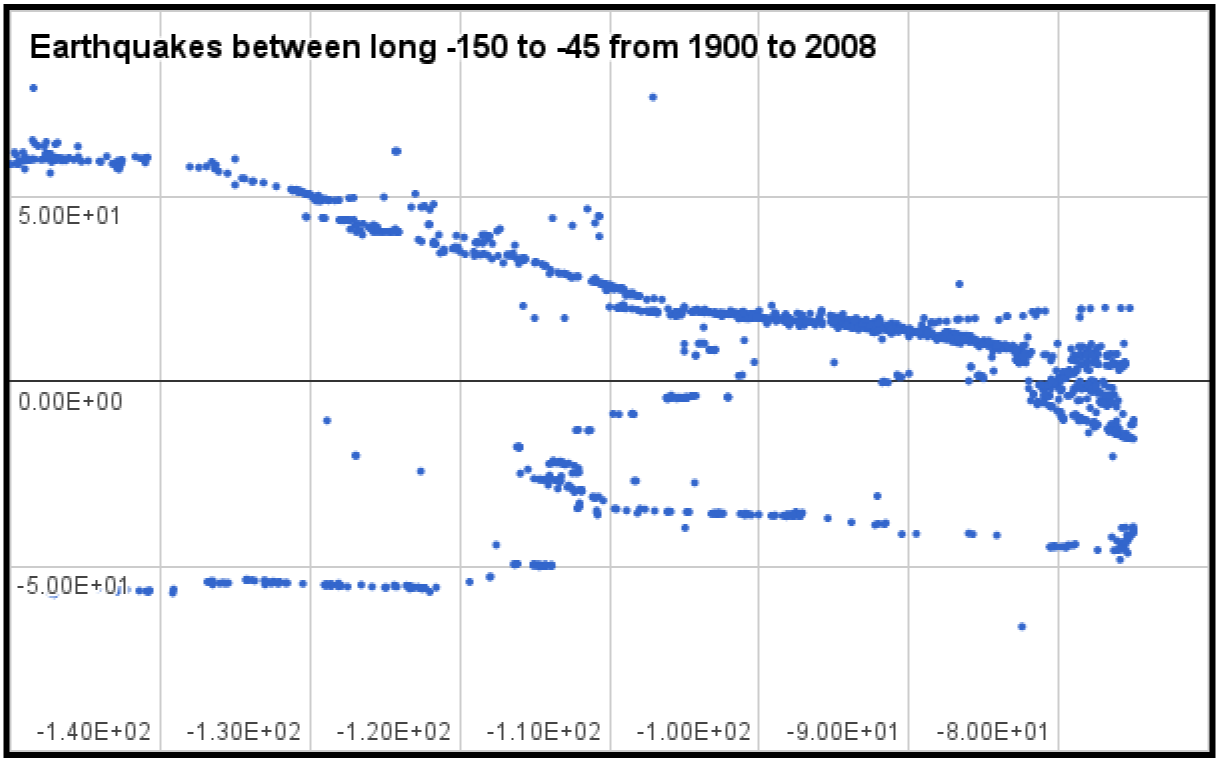


fig1

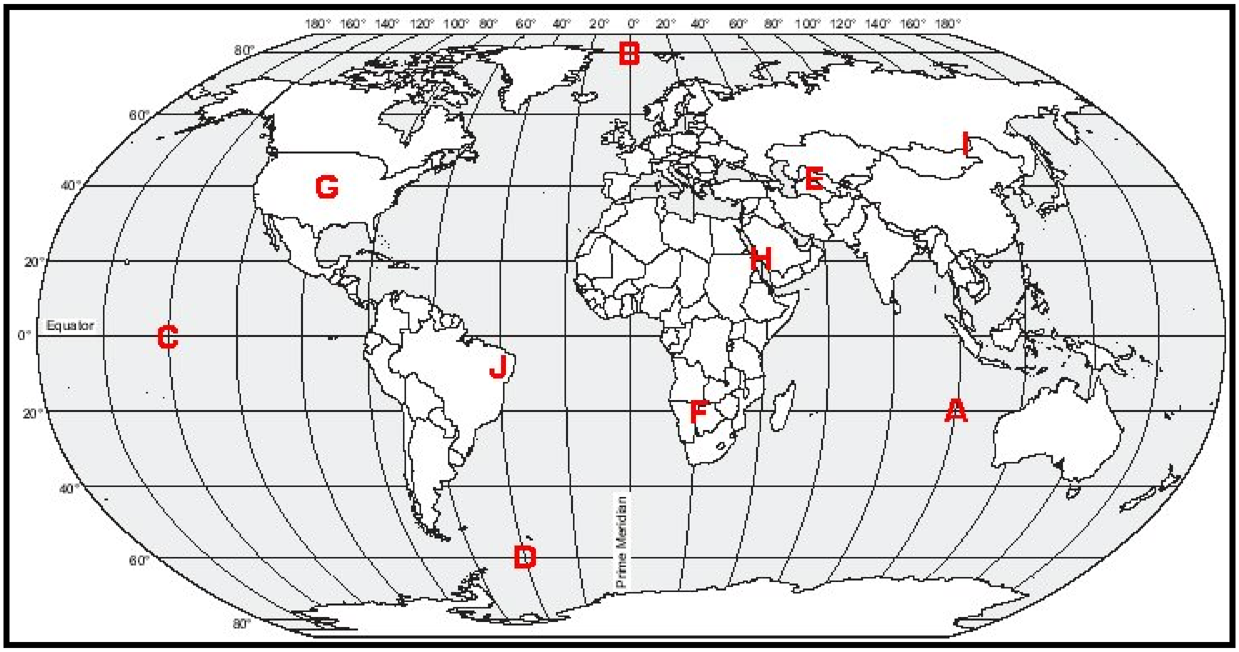
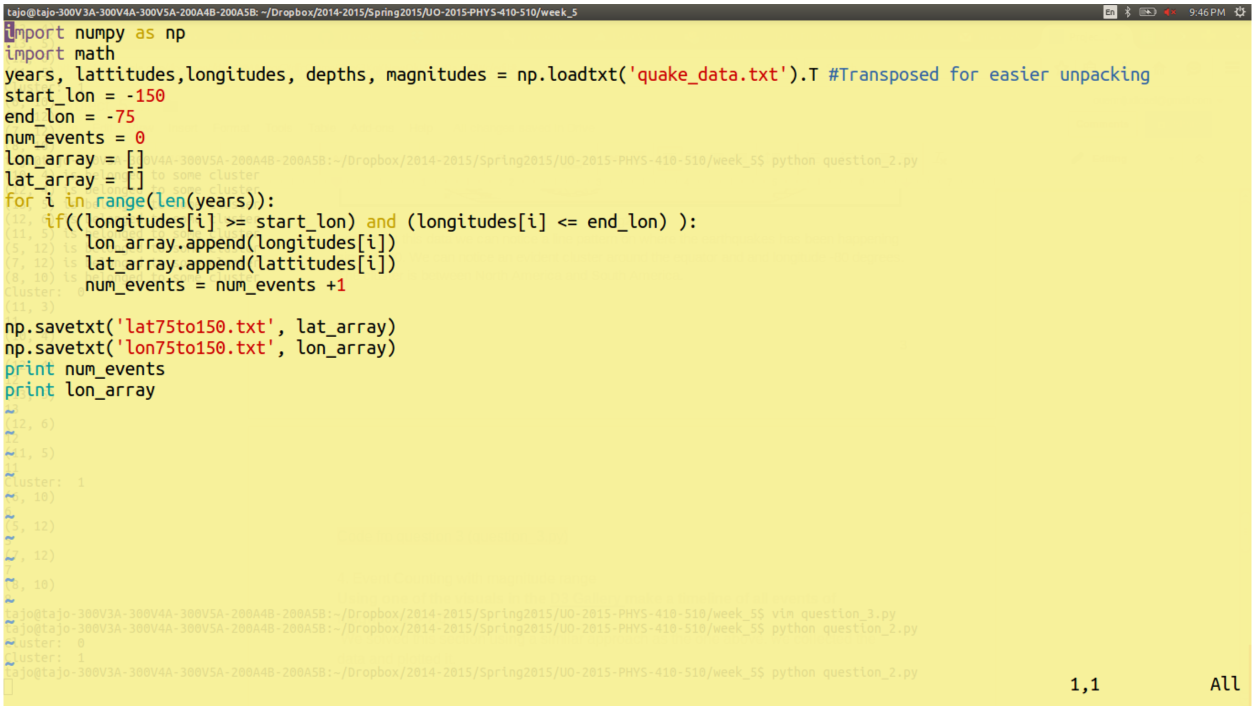


fig2

Looking at this data we can notice a line pattern on where the earthquakes has been happening since 1900. We can notice an evident cluster around the equator and and longitude -80 degrees. The cluster is between North America and South America.

Code fro question 3 (question_3.py)


tajo@tajo-300V3A-300V4A-300V5A-200A4B-200A5B: ~/Dropbox/2014-2015/Spring2015/UO-2015-PHYS-410-510/week_5
9:46 PM

```
import numpy as np
import math
years, lattitudes,longitudes, depths, magnitudes = np.loadtxt('quake_data.txt').T #Transposed for easier unpacking
start_lon = -150
end_lon = -75
num_events = 0
lon_array = []
lat_array = []
for i in range(len(years)):
    if((longitudes[i] >= start_lon) and (longitudes[i] <= end_lon) ):
        lon_array.append(longitudes[i])
        lat_array.append(lattitudes[i])
        num_events = num_events +1

np.savetxt('lat75to150.txt', lat_array)
np.savetxt('lon75to150.txt', lon_array)
print num_events
print lon_array
```

4. Event Counting with magnitude range

**Using one of the visuals in the D3 Gallery make a timeline of all events of magnitude > 7.0**

We solved this section using a similar approach as the one above. we collected the data and plotted it.

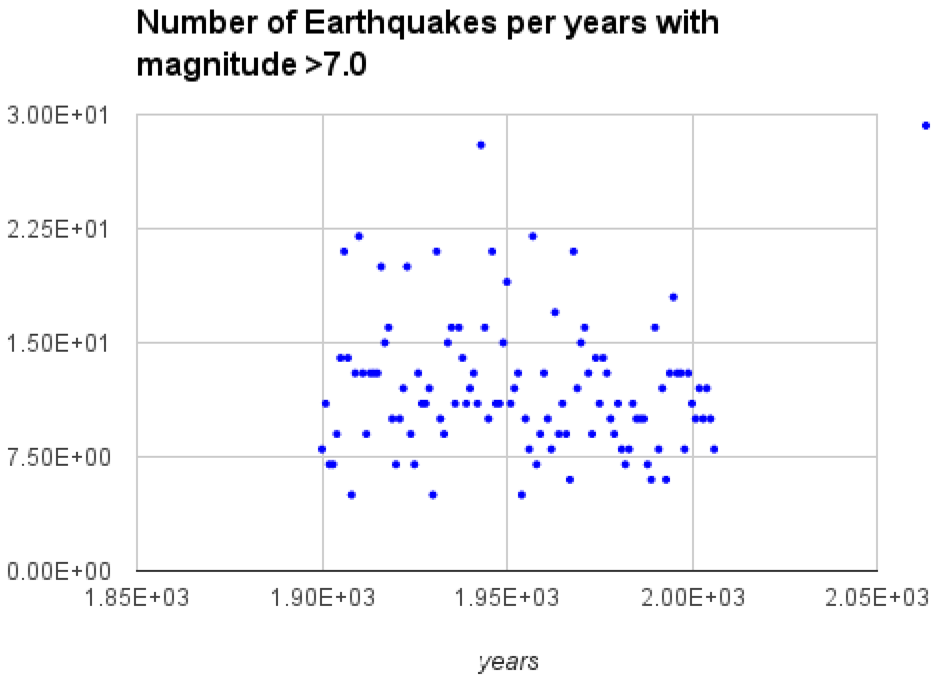
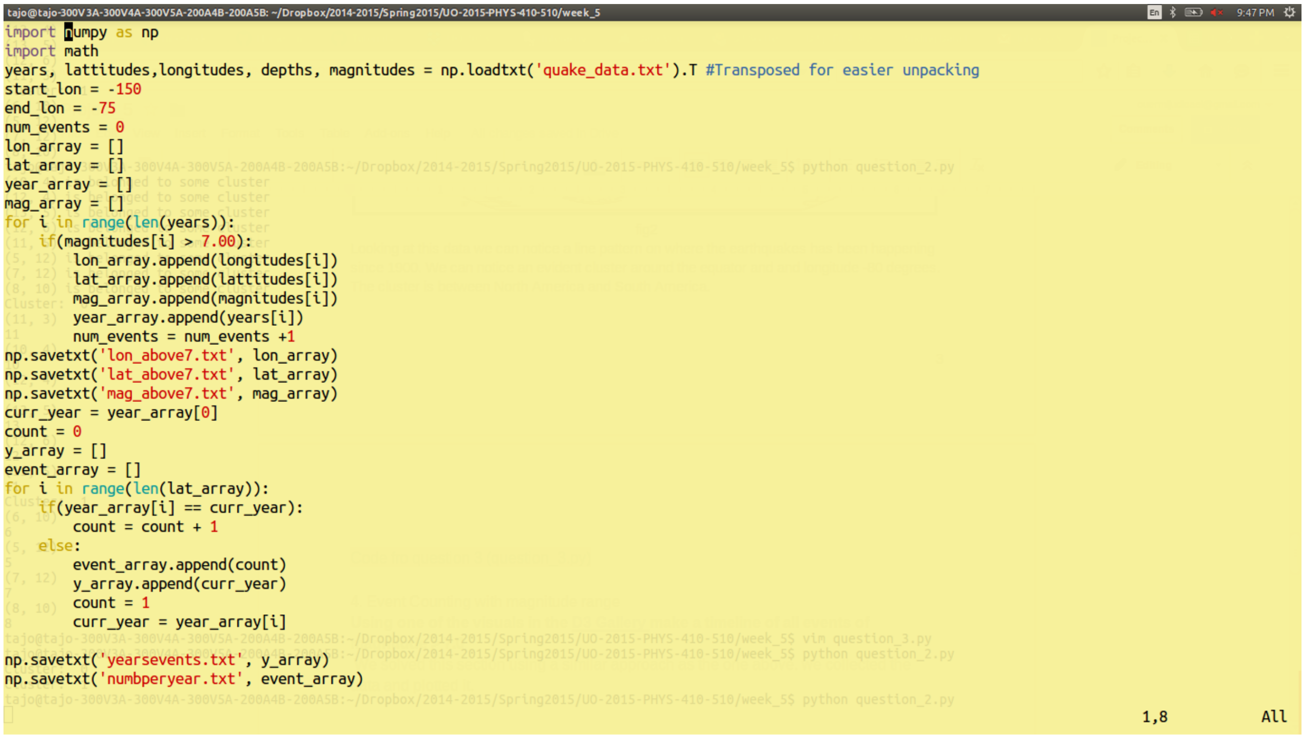


fig3

This shows us the number of earthquakes with magnitudes above 7.0 per year. It is very hard to get much information from the produced. A different approach would provide us with more meaningful information.

For instance I could use the approach in the previous question and plot latitudes vs longitudes to see what we get.

Code for question 4 (question_4.py)



tajo@tajo-300V3A-300V4A-300V5A-200A4B-200A5B: ~/Dropbox/2014-2015/Spring2015/UO-2015-PHYS-410-510/week_5

```
import numpy as np
import math
years, lattitudes,longitudes, depths, magnitudes = np.loadtxt('quake_data.txt').T #Transposed for easier unpacking
start_lon = -150
end_lon = -75
num_events = 0
lon_array = []
lat_array = []
year_array = []
mag_array = []
for i in range(len(years)):
    if(magnitudes[i] > 7.00):
        lon_array.append(longitudes[i])
        lat_array.append(lattitudes[i])
        mag_array.append(magnitudes[i])
        year_array.append(years[i])
        num_events = num_events +1
np.savetxt('lon_above7.txt', lon_array)
np.savetxt('lat_above7.txt', lat_array)
np.savetxt('mag_above7.txt', mag_array)
curr_year = year_array[0]
count = 0
y_array = []
event_array = []
for i in range(len(lat_array)):
    if(year_array[i] == curr_year):
        count = count + 1
    else:
        event_array.append(count)
        y_array.append(curr_year)
        count = 1
        curr_year = year_array[i]

np.savetxt('yearsevents.txt', y_array)
np.savetxt('numbperyear.txt', event_array)
                                                                       1,8           All
```

**5.**

**Using the "What Makes People Happy" multiple axis graph in the D3 library, set it up with the following variables that could populate either the X or Y axis. Note there are now many alternatives to the multiple axis plot, this is just one example.**

1.
   - **Longtitude**
   - **Depth**
   - **Magnitude**
   - **Year**

I run out time so I did not get chance to solve this section. If i did have time I would first collect the needed data in a CVS or txt files separetly and then transport it to D3 for visualisation. I will also try to group the data multiple way to see what information I can get out of it.