```
##############################################################################################
##### RADIATION MODEL CODE ########## Taylor Oshan ###############October 22nd 2013 ####
##############################################################################################
###### Recommended: Copy and Paste into text editor and save as .txt .or .py ###########
##############################################################################################
#### Python package Dependencies: Numpy, Networkx, and Scipy.spatial.KDTree ############
##############################################################################################
## The code found in RadiationModel.py within this directory is written according to: ##
##                                                                                    ##
##      Simini, F., Gonzalez, M. C., Maritan, A., & Barabasi, A.-L. (2012). A         ##
##      universal model for mobility and migration patterns. Nature, 484 , 96{        ##
##      100.                                                                          ##
##############################################################################################
## It is employed within my thesis as one of three differing methods for simulating   ##
## spatial interaction networks in order to study the spatial distribution of         ##
## arhcaeological sites and their sizes within the region of ancient Etruria          ##
##############################################################################################

import numpy as np
import matplotlib as plt
import math
import networkx as nx
from scipy.spatial import KDTree

#To analytically predict the the commuting fluxes we consider locations i and j with
#population m(i) and n(j) respecitively, at ditance r(ij) from each other, and we
#denote with s(ij) the total population in the circle of radius r(ij) centered at i
#(exluding the source and destination population). The average flux T(ij) from
# i to j as predicted by the radiation model is :
# T(ij) = (m(i)*n(j))/((m(i) + s(ij))*(m(i) + n(i) + s(ij)))


#Function to calculate distance between two sites ***This distance will eventually
#be the length of Least Accumulated Cost paths between origins and destinations****

def get_dist(lat1, long1, lat2, long2):
    #degrees_to_radians = math.pi/180.0
    #phi1 = (90.0 - lat1)*degrees_to_radians
    #phi2 = (90.0 - lat2)*degrees_to_radians
    #theta1 = long1*degrees_to_radians
    #theta2 = long2*degrees_to_radians
    #cos = (math.sin(phi1)*math.sin(phi2)*math.cos(theta1 - theta2) + math.cos(phi1)*math.cos(phi2))
    #arc = math.acos( cos )
    #return arc
    return math.sqrt((long1-long2)**2+(lat1-lat2)**2)
    #For now Im using euclidian distance since it seems thats what the kdtree uses

#Function to load data (site size, and position) and process it into format for input to model
```

```
#Create kd tree
def get_data(file_path):
    data = np.loadtxt(open(file_path),delimiter=",",skiprows=1, dtype = str) #Load in CSV file
    #Drop the sites name(easier to have no string types in the numpy array for the moment)
    data = np.array(data[:,1:], dtype = float)
    x,y = data[:,2], data[:,3] #Isolate the long/lat coordinates for kdtree
    tree = KDTree(zip(x.ravel(), y.ravel())) #Create KDtree
    num_sites = np.size(data[:,1]) #Count number of sites in the data
    pos = {} #Tuple to store coordinates for plotting results
    for site in range(num_sites):
        pos[str(site)] =(float(data[site][3]), float(data[site][2])) #Add each sites coordinates
    dists = np.zeros((num_sites,num_sites)) #Array to hold calculated distance for each pair of sites
    rows,cols = np.shape(dists)
    for r in range(rows): #calculate each distance; distance from a site to itself is always 0
        for c in range(cols):
            if r == c:
                dists[r,c] = 0
            else:
                dists[r,c] =
get_dist(float(data[r][2]),float(data[r][3]),float(data[c][2]),float(data[c][3]))

    return data, tree, num_sites, dists, pos

#Function to calculate total estimated area within distance r from an origin site
def s(data,tree,i,j, dist):
    s_total = 0
    comp_sites = tree.query_ball_point(data[i,2:4], dist) #Query kdtree to find all sites within distance
    if len(comp_sites) > 1: #If there is more then one competing site
        for each in comp_sites:
            if each != i and each != j: #Exclude M and N from the total
                s_total += data[each,1] #Add the estimated area (population) for each competing site
    else:
        s_total = data[comp_sites,1] #If there is only one then its estimated area is the total
    return s_total

#Function to calculate the flow from each origin site to each destination site using radiation model
def rad_model(data, tree, num_sites, dists, tot_ratio=1.0):
    # tot_ratio comes from Ti from Eqn (2) of the Radiation Model paper
    # In the paper, they define Ti as equivalent to \sum_{j != i}{Tij} or more simply:
    # Ti = Mi * (Nc/N) where Nc is total # of 'commuters' and N is total population
    # For our purposes, this acts like a 'scaling' factor, and can be set to 1 by default
    flows = np.zeros((num_sites,num_sites)) #Create an array to hold calculations from site pairs
    rows,cols = np.shape(flows)
    for i in range(rows): # Calculate each flow flows from a site to iself is always 0
        for j in range(cols):
            if i == j:
                flows[i,j] = 0 #Flows from a site to iself is always 0
            else:
```

```
                Mi = data[i,1] #Estimated area (population) of origin site
                Nj = data[j,1] #Estimated area (population) of destination site
                Sij = s(data, tree, i, j, dists[i,j]) #Total estimated area of all sites r distance
                Ti = Mi*tot_ratio                     # (or less) from M; not including M or N.
                Tij = (Ti*Mi*Nj)/((Mi + Sij)*(Mi + Nj + Sij))
                flows[i,j] = Tij #Flow calculation
    return flows


# Preprocess data
data, tree, num_sites, dists,pos = get_data("DATA IN CSV FILE GOES HERE")

# Run Model
flows = rad_model(data, tree, num_sites, dists, tot_ratio=1.0)

#Graph results and save outputs
print "Graphing proposed network"
cutoff = np.max(flows) * 0 #Cutoff used to optionally filter flows
G=nx.DiGraph()
links = []
widths = []
justLinks = []
rows,cols = np.shape(flows)
out = file('flows.csv', 'w') # create output file
out.write("Origin, Destination, Flow\n")
for row in range(rows):
    for col in range(cols):
        if flows[row,col] > cutoff:
            # write flows to output file
            out.writelines("%s\t%s\t%s\n" %  (str(row) + "," , str(col) + "," , flows[row, col]))
            links.append([str(row),str(col), (flows[row,col]*1.0)])
            widths.append(flows[row,col]*0.01)
G.add_weighted_edges_from(links)



nx.draw_networkx(G, pos, width = widths)
#plt.pyplot.show()
plt.pyplot.savefig('flows.png')
out.close()


nx.draw_networkx(G,pos, width = widths, alpha = .7 )
plt.pyplot.show()
```