```
from operator import itemgetter, attrgetter, mul
import random
import sys
import os
import math
import re
import csv
import time
import datetime
import numpy as np
from scipy.special import comb
"""Developped by Jing Deng, amended by Eric Delmelle (contact: delmelle@gmail.com)"""
"""(1)try to add a constraint of area for the selection of patches"""
"""(2)for the new generationselection, we always keep the best two chromosomes """
"""(3)output all the solutions and selected etlism solutions"""
"""(4)Record the time information"""
"""(6)This version is to test with real dataset, id actually starts from 0"""
"""(7)Read the solution, and output the list of the selected patches"""
"""(8)Change the first generation to select less patches"""
"""(9)While reading data the first time, index the area of the patches,"""
"""and change the area objective as a percentage of the total area """
"""(10)Improve the method of refining the chromosome, """
"""(11)Output the number of patches that are selected and total area """
"""(12)Add another input as the mark for whether the parcels are selected,
and read in another datatable should be included."""
"""(13)This version is to test new dataset, adding more types of distances"""
"""(14)Considering cluster (also called reserve group) before we evaluate the solution"
""
"""(15)Now the distance is updated to: min, max, avg and centroid distance"""
"""(16)Slightly change the method of identifying cluster, the minimum distance between
two patches should be larger than threshold"""
"""(17)This version do two-step optimization/randomization"""
"""(18)Change the random selection of cluster to use some algorithm to improve the sele
ction of clusters, tabu idea"""
"""(19)Change the area constraint to each cluster, divided by the number of clusters"""

"""(20)Modify the method of fixing area and distance constraint"""
"""(21)Slightly change the function for generate intial chrom and mutate"""
"""(22)While fixing the constraint issue, remove the smaller one first"""
"""(23)Add the prune area process"""
"""(24)In this version, we relax the probability surface more, and add prune area proce
ss before and after the checkChro function"""
"""(25)Change some fix area steps based on probability not just distances"""
"""(26)This version is based on ga05112016weare, change it back to 100 polygon example"
""

# SPECIFY CLUSTER LINE 283 (givenCluster) and line 1499
# GLOBAL VARIABLE
solution_found = False

"""GA parameters"""
popN = 75 # n number of chromos per each generation
genesPerCh = 409 # length of chromos
crossover_rate = 0.8
mutation_rate = 0.3
max_iterations2 =75#75  # the number of iterations for patch selection process


"""Application parameters"""
alpha = .8
```

```
distThreshold = 20000.0 # distance threshold for clustering, this will be used as interval for generating centroids
intervalFactor = 1 # when generate the cluster centroid points, calculate the interval by mutiply distance threshold and this factor
max_iterations = 100 #M: number of outer loop iterations
target = 1000000.0

#patchN = 2 # select a number of patches for land conservation
totalRecords = 0  # how many pairs, records
matrix = list() # to store all the data
matrix2 = list() # to store the mark table
distMatrixMin = np.zeros((genesPerCh,genesPerCh))
distMatrixMax = np.zeros((genesPerCh,genesPerCh))
distMatrixAvg = np.zeros((genesPerCh,genesPerCh))
bigDistMatrix = np.zeros((genesPerCh,genesPerCh))
bestN = 2 # output only the best two solutions
areaCon = 0.0
areaConP = 0.16
areaTConP = 0.31
areaTCon = 0.0
areaList = [0.0]*genesPerCh
indexList = [0]*genesPerCh # to mark whether it is available to change, preserved
# We may not use the centroid points below
cXList = [0.0]*genesPerCh # to store the x values for the patches' centroid points
cYList = [0.0]*genesPerCh # to store the y values for the patches' centroid points
markList = list() # to store the patches that are already preserved
areaTotal = 0
f1 = 0
#f2 = 0
#f3 = 0
#f4 = 0
outputFile = "C:\\GIS\\landconservation\\work\\weare\\generations.txt"
bestOutputFile = "C:\\GIS\\landconservation\\work\\weare\\bestChroms.txt"
timeFile = "C:\\GIS\\landconservation\\work\\weare\\time.txt"
clusterFile = "C:\\GIS\\landconservation\\work\\weare\\cluster.txt"
outputCentroidFile = "C:\\GIS\\landconservation\\work\\weare\\centroid.txt"
outFile = "C:\\GIS\\landconservation\\work\\weare\\out.txt"
fOut = open(outFile, 'w')


#For clusters:
enableOutput = False #whether output the centroid points
interval = 20000 # the interval distance between two cluster centroids
p = 3 #User define how many clusters they want
cInter = 5 #what is the interval between two clusters
clusterNum = 0 #how many clusters in total
cList = [] #store the cluster ids
#cCList = [] #store the number of patches for each cluster
noClusSet = set() #to mark out those bad cluster combinations
clusterList = [0]*genesPerCh #the final cluster result
cluXList = list() #to store the centroid points of the clusters
cluYList = list()
clusMark = [0]*genesPerCh #to mark the cluster number for the patches
distMask = [0.0]*genesPerCh #to store the minimum distance from patch to its closest cluster
distMask2 = [0.0]*genesPerCh #to recalculate the distance, standardized and consider biodiversity weight
distMask3 = [0.0]*genesPerCh #to store the distance biodiversity ration.
weiList = [0.0]*genesPerCh #to store the biodiversity weight for each patchset
#probList = [0.0]*genesPerCh #to store the probability for patch selection for each gene
```

```
Ccombs = 0 #given the number of total clusters and clusters to select, the number of al
    l the possible combinations
#The extent of the data
xMin = 0.0
yMin = 0.0
xMax = 0.0
yMax = 0.0
#To store the previous best selection of cluster
cpList=[]
#The best objective value for previous solution
opVal = -99999999999999999999.0

"""Based on the centroids of all the patches, get the extent"""
def getExtent(cXList, cYList):
  xMin = 10000000000000000
  yMin = 10000000000000000

  xMax = -1
  yMax = -1
  extentLst = []
  for i in cXList:
    if xMin > float(i):
      xMin = float(i)
    if xMax < float(i):
      xMax = float(i)

  for i in cYList:
    if yMin > float(i):
      yMin = float(i)
    if yMax < float(i):
      yMax = float(i)
  extentLst = list()
  extentLst.append(xMin)
  extentLst.append(yMin)
  extentLst.append(xMax)
  extentLst.append(yMax)
  return extentLst

"""Read in the data from a csv file"""
"""Also generate a number of cluster centroids"""
def init ():
  global totalRecords # need to modify the global copy of this variable
  global matrix
  global matrix2
  global areaTotal
  global areaList
  global indexList
  global cXList
  global cYList
  global markList
  global distMatrixMin
  global distMatrixMax
  global distMatrixAvg
  global interval
  global weiList

  parcelsDist = "C:\\GIS\\landconservation\\work\\weare\\data\\distance.csv"
  parcelsMark = "C:\\GIS\\landconservation\\work\\weare\\data\\info.csv"
  with open(parcelsDist,'rb') as file:
    contents = csv.DictReader(file)
    for row in contents:
```

```
171.        matrix.append(row)
172.        totalRecords = totalRecords + 1
173.# Store the information about whether the parcel is already conserved in matrix2
174.  with open(parcelsMark,'rb') as file:
175.    contents2 = csv.DictReader(file)
176.    for row in contents2:
177.        matrix2.append(row)
178.  #store the size of patches in a list and calculate the total area
179.  for i in range(genesPerCh):
180.    if int(matrix2[i]['CON']) == 1:
181.      indexList[i] = 1
182.      markList.append(i)
183.    else:
184.      indexList[i] = 0
185.    if areaList[i] == 0:
186.      areaList[i] = float(matrix2[i]['ACRES'])
187.    if cXList[i] == 0.0:
188.      cXList[i] = matrix2[i]['X']
189.    if cYList[i] == 0.0:
190.      cYList[i] = matrix2[i]['Y']
191.    if weiList[i] == 0.0:
192.      weiList[i] = float(matrix2[i]['W'])
193.  #standardize the weilist
194.  weiMax = max(weiList)
195.  print "weiMax=" + str(weiMax)
196.  weiMin = min(weiList)
197.  print "weiMin=" + str(weiMin)
198.  for i in range(genesPerCh):
199.    weiList[i] = float((weiList[i]-weiMin)/(weiMax-weiMin))+0.0001
200.  #print weiList
201.  for i in range(totalRecords):
202.    index = int(matrix[i]['PAID'])
203.    index2 = int(matrix[i]['PBID'])
204.    # Read the distance and store in the matrix
205.    distMatrixMin[index][index2]=float(matrix[i]['DISTMIN'])
206.    distMatrixMin[index2][index]=distMatrixMin[index][index2] # for later searching pur
    pose
207.    distMatrixMax[index][index2]=float(matrix[i]['DISTMAX'])
208.    distMatrixMax[index2][index]=float(matrix[i]['DISTMAX'])
209.    distMatrixAvg[index][index2]=float(matrix[i]['DISTAVG'])
210.    distMatrixAvg[index2][index]=float(matrix[i]['DISTAVG'])
211.  #We print out the data to check whether we read in the data correctly
212.  #print matrix2[0]['ACRES']
213.  #print matrix2[0]['X']
214.  #print matrix2[0]['Y']
215.  #print distMatrixMin[21][184]
216.  global areaCon
217.  global areaTCon
218.  for j in range(len(areaList)):
219.    areaTotal = areaTotal + float(areaList[j])
220.  areaCon = float(areaConP * areaTotal)
221.  areaTCon = areaTotal * areaTConP
222.  #print 'Total area of the patches is:%s' %(areaTotal)
223.  #print 'Constraint area for each cluster is:%s' %(areaCon)
224.  #print 'Constraint area for all the site selection is: %s' %(areaTCon)
225.
226.  #Generate the clusters
227.  #below is using the predefined reserve centroid points
228.  lst = [947994,187782,991752, 237660]
229.  interval = 20000
230.  generateCluster(lst, interval)
```

```
231.  #lst = getExtent(cXList,cYList)
232.  #interval = int(lst[2] - lst[0])/5
233.  #generateCluster(lst,interval) #For the real data extent
234.
235.
236."""This function is to generate the list of clusters based on the extent of the real da
    ta, and calculate the centroid points"""
237."""Parameter is the extent of """
238.def generateCluster(extentLst,interval):
239.  # Get the extent and generate the list to store all the centroid points
240.  global xMin
241.  global xMax
242.  global yMin
243.  global yMax
244.  global Ccombs
245.  xMin = extentLst[0]
246.  yMin = extentLst[1]
247.  xMax = extentLst[2]
248.  yMax = extentLst[3]
249.
250.  global clusterNum
251.  global cInter
252.  global cluXList
253.  global cluYList
254.
255.  for i in range(int(xMin),int(xMax),interval):
256.    x = i
257.    for j in range(int(yMin),int(yMax),interval):
258.      y = j
259.      cluXList.append(x)
260.      cluYList.append(y)
261.      clusterNum = clusterNum + 1
262.  cInter = int(clusterNum / 10)
263.  #print "Generate "+str(clusterNum)+" cluster centroid points."
264.  #Ccombs = numPair(clusterNum - 1) #this can actually only handle two clusters selecti
    on, and the first one is not considered
265.  Ccombs = comb(clusterNum,p,exact=True) #this is not working correctly
266.  #print "Ccombs="
267.  #print Ccombs
268.  #Ccombs = 2925 #I give it a value directly C25,3
269.  #Ccombs = 300
270.  #print "There are "+str(Ccombs)+" combinations in total."
271.  if enableOutput == True:
272.    cenFile = open(outputCentroidFile,'w')
273.    for i in range(len(cluXList)):
274.      cenFile.write(str(cluXList[i])+"\t"+str(cluYList[i])+"\n")
275.    cenFile.close()
276.  global bigDistMatrix
277.  for i in range(genesPerCh):
278.    for j in range(len(cluXList)):
279.      bigDistMatrix[i][j] = distPatch(float(cXList[i]),float(cYList[i]),float(cluXList[
    j]),float(cluYList[j]))
280.
281."""c:the number of clusters in total, p: how many clusters to select"""
282."""Keep working until the clusters fit the requirments, the area balance"""
283.def selCluster(c,p,givenCluster):
284.  cList2 = givenCluster
285.  cList3 = tuple(cList2)
286.  #print cList2
287.  while cList3 in noClusSet or balanceClus(cList2) == False:
288.    #if it's not balanced, have to select again
```

```
289.    #print "not balance"
290.    cList2 = randomClu(c,p)
291.    cList3 = tuple(cList2)
292.  else:
293.    return cList2
294.
295."""This function is to do random selection of clusters"""
296.def randomClu(c,p):
297.  cl = set() #the id of selected cluster, can't have repeat id
298.  while len(cl) < p:
299.    # if p > 1:
300.    #   #calculate the interval
301.    #   inter1 = len(cl)*int(c/p)
302.    #   inter2 =(len(cl)+1)*int(c/p)
303.    #   if inter1 == 0:
304.    #     inter1 = 1
305.    #   if inter2 > c-1:
306.    #     inter2 = c-1
307.    #   cl.add(random.randint(inter1,inter2))
308.    # else:
309.    #   cl.add(random.randint(1,c-1))
310.
311.    #cl.add(random.randint(1,c-1))
312.    cl.add(random.randint(0,c-1))
313.  cl = list(cl)
314.  return cl
315.
316."""This function is to swap the cluster selection, random select one cluster number to
    swap"""
317."""The change is based on the existing list cList"""
318.def swapCluster(cl,c,p):
319.  cList2 = randomClu2(cl,c,p)
320.  cList3 = tuple(cList2)
321.  count = 0
322.  while cList3 in noClusSet or balanceClus(cList2) == False:
323.    #if it's not balanced, have to select again
324.    #print "swap clusters again"
325.    #print len(noClusSet)
326.    if count < p:
327.      cList2 = randomClu2(cl,c,p)
328.      count += 1
329.    else:
330.      cList2 = randomClu(c,p)
331.    cList3 = tuple(cList2)
332.  else:
333.    return cList2
334.
335."""This function is to implement the swap, it should be applied together with the funct
    ion swapCluster"""
336."""c:the number of clusters in total, p: how many clusters to select"""
337.def randomClu2(cl,c,p):
338.  #get a random position
339.  loc = random.randint(0,p-1)
340.  #print "the random location"
341.  #print loc
342.  #print cList
343.  list2 = list(cl) #don't want to change cl
344.  cID = list2[loc]
345.  del list2[loc] #remove the element at that position
346.  cList2 = set(list2)
347.  a = random.randint(0,1)
```

```
348.   while len(cList2)<p:
349.     # if a==0:
350.     #   cID += cInter
351.     # else:
352.     #   cID -= cInter
353.     # if cID > c-1:
354.     #   cID -= int(c/2)
355.     # if cID < 1:
356.     #   cID += int(c/2)
357.     # cList2.add(cID)
358.     cList2.add(random.randint(1,c-1)) #random change one cluster number
359.   cl = list(cList2)
360.   return cl
361.
362. """Random select clusters and identify cluster number for each patch"""
363. """The input parameter is the list of cluster id"""
364. """Generate a weight mask based on distance mask and weight"""
365. def defineClusterPatch(cl):
366.   #print cl
367.   global distMask
368.   global clusMask
369.   for i in range(genesPerCh):
370.     # For each patch get the minDist
371.     distL = {}
372.     if len(cl)==0:
373.       print "wrong!!!"
374.       print cl
375.
376.     for j in cl:
377.       x1 = float(cXList[i]) #patch centroid point
378.       y1 = float(cYList[i])
379.       x2 = float(cluXList[j]) #cluster centroid
380.       y2 = float(cluYList[j])
381.       dist = distPatch(x1,y1,x2,y2)
382.       key = str(j)
383.       distL[key] = dist
384.     minDist = sorted(distL.items(),key=itemgetter(1))
385.     clusMark[i] = int(minDist[0][0])
386.     distMask[i] = float(minDist[0][1])
387.
388.   reCalDist(alpha,bigDistMatrix)
389.
390. """This function is to recalculate the distmask"""
391. """It is the normalized distance"""
392. def reCalDist(alpha,bigDistMatrix):
393.   global distMask2
394.   global distMask3
395.   #distMin = float(min(distMax))
396.   #distMax = float(max(distMax))
397.   distMax = 61140.7880702
398.   distMin = 200.3293703
399.   distRg = float(distMax - distMin)
400.   # a = distPatch(float(cXList[11]),float(cYList[11]),float(cluXList[8]),float(cluYList
      [8]))
401.   # di = float((a-distMin)/distRg)+0.001
402.   # print di
403.   #print sum(weiList)
404.   for i in range(len(distMask)):
405.     distMask2[i] = (float(distMask[i])-float(distMin))/(float(distRg)+0.001)
406.     distMask3[i] = float(math.pow(math.pow(distMask2[i],1),alpha)/math.pow(weiList[i],(
      1-alpha)))
```

```
407.
408.
409."""This function is to check whether the selection of cluster make sense, the patches t
    hat belong to a cluster have enough sum area"""
410.def balanceClus(cl):
411.  #todo: for each cluster, calulate whether the patches area can be larger than the con
    straints
412.  defineClusterPatch(cl)
413.  global noClusSet
414.  #Check each cluster, if not working, return false
415.  #areaPerClu = float( areaTCon / p)
416.  areaPerClu = areaCon
417.  for eachClus in cl:
418.    # Each cluster
419.    areaTotal = 0.0
420.    for j in range(genesPerCh):
421.      #print eachClus
422.      if clusMark[j] == int(eachClus):
423.        areaTotal += float(areaList[j])
424.    if areaTotal < areaPerClu:
425.      #If it doesn't fit the solution, then this combination should be marked out
426.      #print "The solution doesn't balance, remove the solution."
427.      noClusSet.add(tuple(cl))
428.      #print areaTotal
429.      return False
430.  return True #in the end can return true
431.
432."""To set the probability for the second round, less strict probabilities"""
433.def setPatchesProb2(clusMark,distMask,cluList):
434.  probaPatches = [0.0]*genesPerCh
435.  listAll={}
436.  # For each cluster, sort the distance list and select the closest one
437.  for eachCluster in range(p):
438.    areaP = 0.0 #total area of the patch
439.    listC = {}
440.    listC2 = {} #To store all the preserved patches, they need to be add into cluster f
    irst
441.    for bit in range(genesPerCh):
442.      # add the distance and gene id into the dictionary, then sort the dictionary
443.      if clusMark[bit] == cluList[eachCluster]:
444.        key = str(bit)
445.        if int(indexList[bit]) == 1: #already preserved
446.          listC2[key] = distMask[bit]
447.          probaPatches[bit] = 1
448.        else:
449.          listC[key] = distMask[bit]
450.
451.    # The first round, add all the preserved patches
452.    for item in listC2:
453.      pIdx = int(item[0])
454.      areaP += float(areaList[pIdx])
455.      probaPatches[pIdx] = 1 #They have to be selected
456.
457.    # Then if we still have room, find the patches by order, smaller distance has more
    priority
458.    sorted_l = sorted(listC.items(),key=itemgetter(1))
459.    prob = 0.999 # the initial high probability
460.    while areaP < float(areaCon) and len(sorted_l) > 0:
461.    #while areaP < float(areaTCon/p) and len(sorted_l) > 0:
462.      pIdx = int(sorted_l[0][0])
463.      if probaPatches[pIdx] != 1:
```

```
464.        probaPatches[pIdx] = prob#These patches all have high probability to be selecte
    d
465.      areaP += float(areaList[pIdx])
466.      sorted_l.pop(0) #after sorting, it is a list, and remove the first element
467.
468.    # The last round, the patches left in the list will have very low probability to be
     selected
469.    probB = 0.01
470.    for item in sorted_l:
471.      pIdx = int(item[0])
472.      if probaPatches[pIdx] != 1:
473.        probaPatches[pIdx] = probB
474.  return probaPatches
475.
476.
477.
478."""To set the probability"""
479.def setPatchesProb(clusMark,distMask,cluList):
480.  probaPatches = [0.0]*genesPerCh
481.  listAll={}
482.  # For each cluster, sort the distance list and select the closest one
483.  for eachCluster in range(p):
484.    areaP = 0.0 #total area of the patch
485.    listC = {}
486.    listC2 = {} #To store all the preserved patches, they need to be add into cluster f
    irst
487.    for bit in range(genesPerCh):
488.      # add the distance and gene id into the dictionary, then sort the dictionary
489.      if clusMark[bit] == cluList[eachCluster]:
490.        key = str(bit)
491.        if int(indexList[bit]) == 1: #already preserved
492.          listC2[key] = distMask[bit]
493.          probaPatches[bit] = 1
494.        else:
495.          listC[key] = distMask[bit]
496.
497.    # The first round, add all the preserved patches
498.    for item in listC2:
499.      pIdx = int(item[0])
500.      areaP += float(areaList[pIdx])
501.      probaPatches[pIdx] = 1 #They have to be selected
502.
503.    # Then if we still have room, find the patches by order, smaller distance has more
    priority
504.    sorted_l = sorted(listC.items(),key=itemgetter(1))
505.    prob = 0.999 # the initial high probability
506.    while areaP < float(0.6*areaCon) and len(sorted_l) > 0:
507.    #while areaP < float(0.7*areaTCon/p) and len(sorted_l) > 0:
508.      pIdx = int(sorted_l[0][0])
509.      if probaPatches[pIdx] != 1:
510.        probaPatches[pIdx] = prob#These patches all have high probability to be selecte
    d
511.        prob = prob - 0.01
512.      areaP += float(areaList[pIdx])
513.      sorted_l.pop(0) #after sorting, it is a list, and remove the first
514.
515.    # The second round
516.    probA = 0.5 # the initial high probability
517.    while areaP < float(0.9*areaCon) and len(sorted_l) > 0:
518.    #while areaP < float(areaTCon/p) and len(sorted_l) > 0:
519.      pIdx = int(sorted_l[0][0])
```

```
      if probaPatches[pIdx] != 1:
        probaPatches[pIdx] = probA#These patches all have high probability to be select
    ed
      areaP += float(areaList[pIdx])
      sorted_l.pop(0) #after sorting, it is a list, and remove the first
      probA = probA - 0.01

    # The last round, the patches left in the list will have very low probability to be
     selected
    probB = 0.001
    for item in sorted_l:
      pIdx = int(item[0])
      if probaPatches[pIdx] != 1:
        probaPatches[pIdx] = probB

  #print probaPatches
  #print it out to a file, this is for testing purpose
  # outputProbFile = "C:\\GIS\\landconservation\\work\\simulateData100poly\\prob"+str(c
    luList)+".txt"
  # outputProbFile = "C:\\GIS\\landconservation\\work\\simulateData100poly\\prob.txt"
  # if cluList == [8,16,18]:
  #   probFile = open(outputProbFile,'w')
  #   print >> probFile, probaPatches
  #   probFile.close
  return probaPatches
"""To set the probability,the GA way"""
def setPatchesProb1(clusMark,distMask,cluList):
  probaPatches = [0.0]*genesPerCh
  listAll={}
  for i in range(genesPerCh):
    if indexList[i] == 1:
      probaPatches[i] = .7
    else:
      probaPatches[i] = .4  #the lower, the less likely it can be selected.

  #print probaPatches
  return probaPatches

"""This function is to generate a number of patches selection based on the probaPatches
    """
"""Input is the probability list, and output is the chromosomes"""
"""Apply the two constraints in the chromosome selection, totalArea and distance thresh
    old"""
"""Make sure for each cluster, at least certain area is selected"""
"""Also to ensure the performance, make sure one chromsome strictly follow the probabil
    ities"""
def generatePop(probaPatches, probaPatches2, probaPatches3, popN):
  #print "Probabilities for each patch:"
  #print probaPatches
  chromos, chromo = [], []
  #global cCList
  #add one chromos first, this one is very tight
  for bit in range(genesPerCh):
    if probaPatches2[bit] > 0.9:
      chromo.append(1)
    else:
      chromo.append(0)
  c1 = checkChro(chromo,probaPatches)
  if sum(c1) != genesPerCh:
    chromos.append(c1)

```

```
575.  # Add another one
576.  chromo = []
577.  for bit in range(genesPerCh):
578.    if probaPatches[bit] > 0.9:
579.      chromo.append(1)
580.    else:
581.      chromo.append(0)
582.  c1 = checkChro(chromo,probaPatches)
583.  if sum(c1) != genesPerCh:
584.    chromos.append(c1)
585.
586.  #over selecting more patches
587.  chromo = []
588.  for bit in range(genesPerCh):
589.    if probaPatches[bit] > 0.5:
590.      chromo.append(1)
591.    else:
592.      chromo.append(0)
593.  c1 = checkChro(chromo,probaPatches)
594.  if sum(c1) != genesPerCh:
595.    chromos.append(c1)
596.
597.  #add other chromos
598.  while len(chromos) < 2:
599.    chromo = []
600.    for bit in range(genesPerCh):
601.      if probaPatches3[bit] > 0.9:
602.        chromo.append(1) #The reserved ones
603.      else:
604.        if random.random() < probaPatches3[bit]:
605.          chromo.append(1)
606.        else:
607.          chromo.append(0)
608.
609.    c1 = checkChro(chromo,probaPatches3)
610.    if sum(c1)==0:
611.      print "check12"
612.    if sum(c1)>genesPerCh:
613.      print "check chromo here2"
614.    chromos.append(c1)
615.
616.  while len(chromos) < popN:
617.    chromo = []
618.    for bit in range(genesPerCh):
619.      if probaPatches[bit] > 0.9:
620.        chromo.append(1) #The reserved ones
621.      else:
622.        if random.random() < probaPatches[bit]:
623.          chromo.append(1)
624.        else:
625.          chromo.append(0)
626.
627.    c1 = checkChro(chromo,probaPatches)
628.    if sum(c1)==0:
629.      print "check12"
630.    if sum(c1)>genesPerCh:
631.      print "check chromo here2"
632.    chromos.append(c1)
633.  return chromos
634.
635.def generatePop2(probaPatches, popN):
```

```
636.   chromos, chromo = [], []
637.   #global cCList
638.   #add other chromos
639.   while len(chromos) < popN:
640.     chromo = []
641.     for bit in range(genesPerCh):
642.       if probaPatches[bit] > 0.9:
643.         chromo.append(1) #The reserved ones
644.       else:
645.         if random.random() < probaPatches[bit]:
646.           chromo.append(1)
647.         else:
648.           chromo.append(0)
649.
650.     c1 = checkChro(chromo,probaPatches)
651.     if sum(c1)==0:
652.       print "check12"
653.     if sum(c1)>genesPerCh:
654.       print "check chromo here2"
655.     chromos.append(c1)
656.   return chromos
657.
658. """This method will completely select all the patches"""
659. def relaxChro(ch):
660.   for i in range(genesPerCh):
661.     ch[i] = 1
662.   #return ch
663.
664. """This method is to check the area constraints and the fix the distance conflict"""
665. """It seems like here is the problem, we should repeatedly do checking distance and ara
     constraint until we reach a certain criteria, say 10 iterations"""
666. def checkChro(ch,probaPatches):
667.   fixed = False
668.   cd = checkDist(ch)
669.   if cd == False:
670.     #to fix the distance first if these is any problem
671.     ch = fixDist(ch,probaPatches)
672.     if sum(ch) == genesPerCh:
673.       return ch
674.   if checkDist(ch) == False:
675.     print "Distance problem is not fixed"
676.
677.   ca = checkArea(ch)
678.   if ca == "C100":
679.     #no need to fix anything
680.     ch1 = pruneArea(ch,2,probaPatches)
681.     if checkArea(ch1) !="C100":
682.       print "Area problem 1!"
683.     return ch1
684.   elif ca == "C99":
685.     #only need to fix the total area
686.     fixAreaTotal(ch, probaPatches)
687.     if sum(ch) == genesPerCh:
688.       return ch
689.     if checkArea(ch) != "C100" :
690.       #print ch
691.       print "Area problem 2!"
692.   else:
693.     #need to fix the cluster too
694.     fixArea(ch,probaPatches)
695.     if sum(ch) == genesPerCh:
```

```
      return ch
    if checkArea(ch)!="C100":
      print "Area problem 3!"

  if checkDist(ch) == True and checkArea(ch) == "C100":
    fixed = True
  else:
    print "here has problem again"
  return ch

"""This function is to check whether it has distance conflict"""
def checkDist(ch):
  if sum(ch)==genesPerCh:
    return True
  for bit in range(genesPerCh-1):
    if ch[bit] == 1:
      for bit2 in range(bit+1,genesPerCh):
        if ch[bit2] == 1:
          if clusMark[bit]!=clusMark[bit2] and distMatrixMin[bit][bit2] < distThreshold
:
            return False
  return True

"""This method is to check the distance conflict and fix it"""
"""All the patches in one cluster have to be far away from all the patches in another c
luster, using the threshold distance"""
"""While detecting conflict, identify the two conflicting patches and drop one of them"
""
def fixDist(ch,probaPatches):
  global noClusSet
  #First, check the distance conflicts, in this step, may remove several patches
  for bit in range(genesPerCh-1):
    if ch[bit] == 1:
      patch1 = bit
      for bit2 in range(bit+1,genesPerCh):
        if ch[bit2] == 1:
          patch2 = bit2
          if clusMark[patch1]!=clusMark[patch2] and distMatrixMin[patch1][patch2] < dis
tThreshold:
            #They are too close, need to fix it
            if probaPatches[patch1]==1 and probaPatches[patch2]==1:
              #print "There is distance conflict in the reserved patches: "+str(patch1)
 + " and "+str(patch2)+", please readjust the threshold distance!"
              #Can't use this cluster group, so next time it won't show up
              #noClusSet.add(tuple(cList))
              relaxChro(ch)
              return ch # directly finish this
            elif probaPatches[patch1] == 1:
              # can't change the patch1
              ch[patch2] = 0
              probaPatches[patch2] = 0 #can't add this patch, cause it will cause confl
ict with a reserved patch
              # clus = int(clusMark[patch2])
              # for a in range(len(cList)):
              #   if clus == cList[a]:
              #     cCList[a] = cCList[a] - 1
              #     break
            elif probaPatches[patch2] == 1:
              # can't change patch2
              ch[patch1] = 0
              probaPatches[patch1] = 0
```

```
751.              # clus = int(clusMark[patch1])
752.              # for a in range(len(cList)):
753.              #   if clus == cList[a]:
754.              #     cCList[a] = cCList[a] - 1
755.              #     break
756.            else:
757.              #if probaPatches[patch1] > probaPatches[patch2]:
758.              #instead of using the probability, using the distance
759.              if distMask3[patch1] < distMask3[patch2]:
760.                ch[patch2] = 0
761.                #instead of chaing the proability to 0, decrease it, lower the probabil
    ity of choosing this again
762.                # this penalty is higher because it's not random
763.                probaPatches[patch2] = probaPatches[patch2] - 0.5
764.                # clus = int(clusMark[patch2])
765.                # for a in range(len(cList)):
766.                #   if clus == cList[a]:
767.                #     cCList[a] = cCList[a] - 1
768.                #     break
769.              #elif probaPatches[patch1] < probaPatches[patch2]:
770.              elif distMask3[patch1] > distMask3[patch2]:
771.                ch[patch1] = 0
772.                probaPatches[patch1] = probaPatches[patch1] - 0.5
773.
774.              else:
775.                #03112016 can slightly improve this by choosing the one with bigger are
    a
776.                if float(areaList[patch1]) < float(areaList[patch2]):
777.                  ch[patch1] = 0
778.                  probaPatches[patch1] = probaPatches[patch1] - 0.4
779.
780.                else:
781.                  ch[patch2] = 0
782.                  probaPatches[patch2] = probaPatches[patch2] - 0.4
783.
784.  return ch
785.
786.
787."""This method is to refine the chromos applying the area constraints"""
788."""Add a new feature to keep area balance between different clusters"""
789."""The input probablity of the patches have already been changed"""
790."""A random patch will be selected to add the total area, but also based on the cluster
     requirments"""
791."""The previous name is refineChrom"""
792."""In stead of this random fix, maybe should do deterministic way, based on the probabi
    lity"""
793."""In this version, I will rank the probability and add the top ones into the list, if
    all the probability is less than 0, then return relaxed chrom"""
794."""Now since i added two fix area method, this one is the back up"""
795.def fixArea_bak(ch,probaPatches):
796.  # calculate the total area and sort the area of all the chromos
797.  area = 0.0
798.  #cAreas = {} # to store the total area for different clusters
799.  cArea = [0.0]*p
800.  #update the area list
801.  for i in range(p):
802.    cIndx = cList[i]
803.    for bit in range(genesPerCh):
804.      if ch[bit]==1 and clusMark[bit]==cIndx:
805.        cArea[i] += float(areaList[bit])
806.
```

```
  listArea={} #sort the probability
  listA_sort={}
  for bit in range(genesPerCh):
    if probaPatches[bit] > 0.1 and ch[bit]==0:
      # put the available patches into the list and sort by probabilities
      key = str(bit)
      #listArea[key] = probaPatches[bit]
      #listArea[key] = float(areaList[bit])*probaPatches[bit]
      listArea[key] = float(distMask3[bit])
      #should i use the probability or the area?
  listA_sort = sorted(listArea.items(),key=itemgetter(1))

  if len(listA_sort) == 0:
    #nothing to add to the list
    relaxChro(ch)
    return ch
  #else:
    #print listA_sort[0]
  #check the cluster list
  for i in range(p):
    if cArea[i] < areaCon:
      #only do these when cluster doesn't not meet the minimum area requirement
      cIndx = cList[i]
      l = len(listA_sort)
      a = 0
      while a < l-1:
        if a == len(listA_sort) - 1:
          break
        #keep adding a patch into the list
        if cArea[i] < areaCon:
          idx = int(listA_sort[a][0]) #patch id
          if clusMark[idx] == cIndx:
            ch[idx] = 1
            cArea[i] += float(areaList[idx])
            listA_sort.pop(a) # i don't know whether should keep this
          else:
            a = a + 1
        else:
          break
    if cArea[i] < areaCon:
      #still don't reach the cluster requirement
      relaxChro(ch)
      return ch

  #check the complete probability list
  while sum(cArea) < areaTCon and len(listA_sort)>0:
    #If all the clusters meet the minimum requirement, but the total area dose not meet
the requirement
    idx2 = int(listA_sort[0][0]) #directly add the highest patch
    ch[idx2] = 1
    for i in range(p):
      if int(clusMark[idx2]) == cList[i]:
        cArea[i] += float(areaList[idx2])
    listA_sort.pop(0)

  if sum(cArea) < areaCon:
    #still didn't meet the requirements after adding all the patches
    relaxChro(ch)
  return ch

"""New version after 04182016"""
```

```
"""The function is to apply another two are fix process"""
def fixArea2(ch,probaPatches):
  #cAreas = {} # to store the total area for different clusters
  cArea = [0.0]*p
  #update the area list
  for i in range(p):
    for bit in range(genesPerCh):
      if ch[bit]==1 and int(clusMark[bit])==int(cList[i]):
        cArea[i] += float(areaList[bit])

  #for each cluster, fix the cluster area issue
  for i in range(p):
    if cArea[i] < areaCon:
      a = cList[0]
      ch = fixAreaClus(ch,a,probaPatches)
      if sum(ch)==genesPerCh:
        return ch

  check = checkArea(ch)
  if check != "C99" and check!="C100":
    print "something wrong, please check here"
    relaxChro(ch)
  return ch

"""When fixing area, need to make sure that not introducing new distance conflicts, so each new added one will be compared with others"""
"""If fail to fix the total area, return relaxed chromosome"""
def fixAreaTotal(ch,probaPatches):
  areaT = 0.0
  for bit in range(genesPerCh):
    if ch[bit]==1:
      areaT += float(areaList[bit])
  if areaT > areaTCon:
    return

  listArea={}
  listA_sort={}
  for bit in range(genesPerCh):
    if probaPatches[bit] > 0 and ch[bit]==0:
      # put the available patches into the list and sort by probability
      key = str(bit)
      #listArea[key] = float(probaPatches[bit])
      listArea[key] = float(-distMask3[bit])

  if len(listArea) == 0:
    relaxChro(ch)
    return
  # print "the length"
  # print len(listArea)
  #Use the probability, decending order
  listA_sort = sorted(listArea.items(),key=itemgetter(1),reverse=True)
  while len(listA_sort)>0:
    if areaT>areaTCon:
      break
    pId = int(listA_sort[0][0])
    #check all the genes in the chromosome, whether they have conflict with this patch

    addpId = True
    for bit in range(genesPerCh):
      if bit!=pId and ch[bit] == 1 and clusMark[bit]!=clusMark[pId]:
        if distMatrixMin[bit][pId] < distThreshold:
```

```
926.          #not able to add
927.          addpId = False
928.          break
929.    if addpId == True:
930.      ch[pId] = 1
931.      areaT += areaList[pId]
932.      #print "add" + str(pId)
933.    listA_sort.pop(0)
934.
935.  #print ch
936.  #After the fix process, see whether we can finally reach the total area constraint
937.  if areaT < areaTCon:
938.    #it still not enough area
939.    relaxChro(ch)
940.  return
941.  #return ch
942.
943."""The fix process for this function is to fix the single clusters first, then the tota
    l area"""
944.def fixArea(ch,probaPatches):
945.  cSList = [] # To store the cluster
946.  cArea = [0.0]*p
947.  #calculate the total area for the specific cluster
948.  for i in range(p):
949.    for bit in range(genesPerCh):
950.      if ch[bit]==1 and int(clusMark[bit])==cList[i]:
951.        cArea[i] += float(areaList[bit])
952.    if cArea[i] < areaCon:
953.      cSList.append(int(cList[i]))
954.
955.  while len(cSList)>0:
956.    cs = cSList[0]
957.    if sum(ch) != genesPerCh:
958.      fixAreaClus(ch,probaPatches,cs,1)
959.      del cSList[0]
960.    else:
961.      return
962.  fixAreaTotal(ch,probaPatches)
963.
964.def fixAreaClus(ch,probaPatches,cn,a):
965.  listArea={}
966.  listA_sort={}
967.  cArea = 0.0
968.  for bit in range(genesPerCh):
969.    if probaPatches[bit] > 0 and ch[bit]==0 and clusMark[bit]==int(cn):
970.      key = str(bit)
971.      #listArea[key] = float(probaPatches[bit])
972.      if a==1:
973.        listArea[key] = random.randint(0,99)
974.      #listArea[key] = float(-distMask3[bit])
975.    if ch[bit]==1 and clusMark[bit]==int(cn):
976.      cArea += areaList[bit]
977.  if len(listArea) == 0:
978.    relaxChro(ch)
979.    return ch
980.
981.  listA_sort = sorted(listArea.items(),key=itemgetter(1),reverse=True)
982.
983.  while len(listA_sort)>0 and cArea<areaCon:
984.    pId = int(listA_sort[0][0])
985.    addpId = True
```

```
    for bit in range(genesPerCh):
      if bit!=pId and ch[bit] == 1 and clusMark[bit]!=clusMark[pId] and distMatrixMin[bit][pId] < distThreshold:
        addpId = False
        break
    if addpId == True:
      ch[pId] = 1
      cArea += float(areaList[pId])
    listA_sort.pop(0)
  if cArea < areaCon:
    #can't fix this chromosome without violating distance threshold
    relaxChro(ch)


"""Add this function to prune the area a little bit after i get final solution"""
      """Try to identify which patches are removable, then based on their dstance, remove from the furthest one"""
      """Add some randomness in this function"""
      """Before running this function, we need to make sure the area fits the constraint first"""
      def pruneArea(ch,a, probList):
        if sum(ch) == genesPerCh:
          return ch
        #a = random.randint(0,2)
        # calculate the total area and sort the area of all the chromos
        area = 0.0
        areaT = 0.0
        cArea = [0.0]*p
        minP = [0.0]*p # to store the patch id of the smallest patch in each cluster, could be potentially removed
        listDrop={} #store the potential removable patches
        listDrop_sort={} #store the patches by distance
        #for each cluster, calculate area first
        for i in range(p):
          for bit in range(genesPerCh):
            if ch[bit]==1 and clusMark[bit]==cList[i]:
              cArea[i] += float(areaList[bit])
        areaT = sum(cArea)

        # Then in each cluster, identify the potential removable patches
        for i in range(p):
          listArea={} #store the area
          for bit in range(genesPerCh):
            if ch[bit]==1 and clusMark[bit]==cList[i] and probList[bit]<0.9:
              key = str(bit)
              listArea[key] = float(areaList[bit])
          # sort the area
          listA_sort = sorted(listArea.items(),key=itemgetter(1),reverse= True)
          #if len(listArea)==0:
            #print "No removable patches for cluster "+str(i)
          #print listA_sort
          for items in listA_sort:
            pArea = float(items[1])
            if areaT - pArea >= float(areaTCon) and cArea[i] - pArea >= float(areaCon):
              pIdx = int(items[0])
              if a==0:
              #   #using area
                listDrop[str(pIdx)]=float(-distMask2[pIdx])
                #listDrop[str(pIdx)]=float(areaList[pIdx])
                #listDrop[str(pIdx)] = float(areaList[pIdx]/distMask3[pIdx])
```

```
1042.               elif a==1:
1043.                 listDrop[str(pIdx)]=float(areaList[pIdx])
1044.                 #listDrop[str(pIdx)]=float(-distMask3[pIdx])
1045.                 #listDrop[str(pIdx)]=float(probList[pIdx])
1046.               else:
1047.                 #pure random
1048.                 listDrop[str(pIdx)]=random.randint(0,100)
1049.             else:
1050.               # Don't need to check further, no patch can be removed
1051.               break
1052.         if len(listDrop) == 0:
1053.           #print "no need to prune"
1054.           return ch
1055.
1056.         #by area or probability, ascending
1057.         #by distance, descending
1058.         listDrop_sort = sorted(listDrop.items(),key=itemgetter(1))
1059.
1060.         # Check the whole list of removable patches
1061.         while len(listDrop_sort) > 0:
1062.           pId = int(listDrop_sort[0][0])
1063.           pArea = areaList[pId]
1064.           clusId = clusMark[pId]
1065.           delpId = True #whether we can remove this patch
1066.           if areaT - pArea < float(areaTCon):
1067.             delpId = False
1068.           else:
1069.             for i in range(p):
1070.               if cList[i] == clusId:
1071.                 if cArea[i] - pArea < float(areaCon):
1072.                   delpId = False
1073.                 break
1074.           if delpId == True:
1075.             ch[pId] = 0
1076.             areaT = areaT - pArea
1077.             cArea[i] = cArea[i] - pArea
1078.           # Drop the option after we check it
1079.           listDrop_sort.pop(0)
1080.
1081.         #print ch
1082.         if sum(ch)==0:
1083.           print "check11"
1084.           print areaT
1085.         return ch
1086.
1087.       """This function is to check whether solution reach the area requirements"""
1088.       """In this version, we add also add check for the cluster, if all clusters and t
      he total area meet, return 100, either, if only total area doesn't meet, return99"""
1089.       """Else, return the clusters, 0:the first one; 1:the second one.... each time us
      e this function can only return one cluster id"""
1090.       def checkArea(ch):
1091.         totalArea = 0.0
1092.         cArea=[0.0]*p
1093.         for i in range(p):
1094.           for bit in range(genesPerCh):
1095.             if ch[bit]==1 and clusMark[bit]==cList[i]:
1096.               cArea[i] += float(areaList[bit])
1097.         totalArea = sum(cArea)
1098.
1099.         for i in range(p):
1100.           if cArea[i] < areaCon:
```

```
1101.             return cList[i]
1102.
1103.         if totalArea < areaTCon:
1104.           return "C99"
1105.         else:
1106.           return "C100"
1107.
1108.       """This function is to calculate the distance between two points"""
1109.       def distPatch(x1,y1,x2,y2):
1110.         return math.pow(math.sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2)),1)
1111.
1112.       """Define a cluster class"""
1113.       class Clus:
1114.         def __init__(self):
1115.           self.cNum = 0
1116.           self.XList = list()
1117.           self.YList = list()
1118.           self.cenX = 0.0
1119.           self.cenY = 0.0
1120.           self.idList = list()
1121.
1122.
1123.       """This function evaluates the sum distance from every patch centroid to cluster
      centroid"""
1124.       """The lower value, the better; which means more compact and less number of patc
     hes"""
1125.       """Instead of using the predefined cluster centroid, i recalculate the centroid
     points"""
1126.       def evaluate(chromo):
1127.         clusterList = [0]*genesPerCh
1128.         distList = [0.0]*genesPerCh #to get the update of the distance to centroid
1129.         for i in range(genesPerCh):
1130.           clusterList[i] = clusMark[i]*chromo[i] #it has to be selected first
1131.         #f4.write(str(clusterList)+"\n")
1132.
1133.         output = 10000.0
1134.         clusObjList = {} # to store all the cluster objects
1135.         #identify the clusters
1136.         for i in range(genesPerCh):
1137.           k = clusterList[i]
1138.           if k != 0:
1139.             key = str(k)
1140.             if key not in clusObjList:
1141.               clusObj = Clus()
1142.               clusObj.cNum = k
1143.               clusObj.XList.append(float(cXList[i]))
1144.               clusObj.YList.append(float(cYList[i]))
1145.               clusObj.idList.append(i)
1146.               clusObjList[key] = clusObj
1147.             else:
1148.               #only need to update the x, y list
1149.               clusObj = clusObjList[key]
1150.               clusObj.XList.append(float(cXList[i]))
1151.               clusObj.YList.append(float(cYList[i]))
1152.               clusObj.idList.append(i)
1153.         #Update the cluster centroid and calculate the distances of the other points t
     o the centroid points
1154.         for a,b in clusObjList.items():
1155.           b.cenX = sum(b.XList)/float(len(b.XList))
1156.           b.cenY = sum(b.YList)/float(len(b.YList))
1157.           for i in b.idList:
```

```
            #update the distance
            distList[i] = distPatch(float(cXList[i]),float(cYList[i]),b.cenX,b.cenY)
        for i in range(genesPerCh):
          output -
    = chromo[i]*distList[i] #Here it doesn't matter which cluster the patch belongs to, add
     them all together
        return output

      """This function evaluates the sum distance from every patch centroid to cluster
     centroid"""
      """The lower value, the better; which means more compact and less number of patc
    hes"""
      """Use the predefined cluster centroid"""
      """Include two objective function"""
      def evaluate2(chromo):
        F1=0.0 #distance
        F2=0.0 #weights
        #output = 10000.0
        #identify the clusters
        for i in range(genesPerCh):
          #cN = int(clusMark[i])
          #dist = distPatch(float(cluXList[cN]),float(cluYList[cN]),float(cXList[i]),f
    loat(cYList[i]))
          if distMask2[i]<0.000001:
              F1 += math.pow(0.0001,1) * chromo[i]
          else:
              F1 += math.pow(distMask2[i],1) * chromo[i]
          F2 += weiList[i]*chromo[i]
          #output -
    = chromo[i]*dist #Here it doesn't matter which cluster the patch belongs to, add them a
    ll together
          #output += chromo[i]*distMask[i]

        out = (1-alpha)*F2 - alpha*F1 #maximize this
        output = [F1,F2,out]
        return output


      """Based on the objective function, select the best chromosome"""
      """Not using this function"""
      def topChrom (chromos,iteration):
        outputs, F1, F2, errors = [], [], [], []
        i = 1
        for chromo in chromos:
          #f4.write(str(iteration)+"\t")
          output = evaluate2(chromo)
          F1.append(output[0])
          F2.append(output[1])
          outputs.append(output[2])
          error = target-output[2]
          #except ZeroDivisionError:
          if error <= 0:
            global solution_found
            solution_found = True
            error = 0
            #print '\nSOLUTION FOUND'
            #print '%s: \t%s=%s' %(str(i).rjust(5), chromo, str(output).rjust(10))
            break
          else:
            #error = 1/math.fabs(target-output)
```

```
1212.             errors.append(error)
1213.
1214.           i+=1
1215.         fitnessScores = calcFitness (errors) # calc fitness scores from the errors cal
    culated
1216.         pairedPop = zip ( chromos, outputs, F1, F2, fitnessScores) # pair each chromo
    with its ouput and fitness score
1217.         rankedPop = sorted ( pairedPop,key = itemgetter(-
    1),reverse=True) # sort the paired pop by descending fitness score
1218.         topChrom = rankedPop[0]
1219.         print "top"+str(rankedPop[0][1])+str(rankedPop[0][2])
1220.         return topChrom
1221.
1222.       """Calulates fitness as a fraction of the total fitness"""
1223.       """Since there are negative values, I need to take this into consideration"""
1224.       def calcFitness (errors):
1225.         fitnessScores = []
1226.         totalError = sum(errors)
1227.         i = 0
1228.         # fitness scores are a fraction of the total error
1229.         for error in errors:
1230.           fitnessScores.append (float(errors[i])/float(totalError))
1231.           i += 1
1232.         return fitnessScores
1233.
1234.       """Calulates fitness as a fraction of the total fitness"""
1235.       """Use model performance as the input"""
1236.       def calcFitness2 (outputs):
1237.         fitnessScores = []
1238.         totalError = sum(outputs)
1239.         i = 0
1240.         # fitness scores are a fraction of the total error
1241.         for output in outputs:
1242.           fitnessScores.append (float(output)/float(totalError))
1243.           i += 1
1244.         return fitnessScores
1245.
1246.       def displayFit (error):
1247.         bestFitDisplay = 100
1248.         dashesN = int(error * bestFitDisplay)
1249.         dashes = ''
1250.         for j in range(bestFitDisplay-dashesN):
1251.           dashes+=' '
1252.         for i in range(dashesN):
1253.           dashes+='+'
1254.         return dashes
1255.
1256.       """Takes a population of chromosomes and returns a list of tuples where each chr
    omo is paired to its fitness scores and ranked accroding to its fitness"""
1257.       def rankPopbak (chromos,iteration):
1258.         outputs, F1, F2, errors = [], [], [], []
1259.         i = 1
1260.         # evaluate the output of the chromosome,
1261.         # calculate the inverse error of the output
1262.         #print '%s: \t%s=%s \t%s ' %('n'.rjust(5), 'CHROMOSOME'.rjust(15), 'OUTPUT'.rj
    ust(10), 'DISTANCE FROM TARGET'.rjust(17))
1263.         for chromo in chromos:
1264.           #f4.write(str(iteration)+"\t")
1265.           #if sum(chromo)>genesPerCh:
1266.           #  print "chromo problem at iteration "+str(iteration)
1267.           if sum(chromo)==genesPerCh:
```

```
            #not a feasible solution
            output = -9999999999999999999
            F1.append(-999)
            F2.append(-999)
            outputs.append(output)
          else:
            out = evaluate2(chromo)
            F1.append(out[0])
            F2.append(out[1])
            outputs.append(out[2])
        # if they are all the same, then change all their values to a very low number

        # count = 0
        # for i in range(popN):
        #   if outputs[i] == outputs[0]:
        #     count += 1
        # if count == popN:
        #   for i in range(popN):
        #     outputs[i] = -99999999999
        for i in range(popN):
          #try:
            #error = 1/math.fabs(target-output)
          error = target-outputs[i]
          #except ZeroDivisionError:
          if error <= 0:
            global solution_found
            solution_found = True
            error = 0
            #print '\nSOLUTION FOUND'
            #print '%s: \t%s=%s' %(str(i).rjust(5), chromos[i], str(outputs[i]).rjust(10))
            break
          else:
            #error = 1/math.fabs(target-output)
            errors.append(error)
            #print '%s: \t%s=%s\t%s' %(str(i).rjust(5), chromo, str(output).rjust(10), str(error).rjust(15))
            print >> f1, '%s\t%s\t%s\t%s' %(iteration, chromo, output, error)
          i+=1
        fitnessScores = calcFitness2 (outputs) # calc fitness scores from the erros calculated
        pairedPop = zip ( chromos, outputs, F1, F2, fitnessScores) # pair each chromo with its ouput and fitness score
        # if count == popN:
        #   rankedPop = pairedPop
        # else:
        rankedPop = sorted ( pairedPop,key = itemgetter(-1),reverse=True ) # sort the paired pop by descending fitness score
        return rankedPop

      """Takes a population of chromosomes and returns a list of tuples where each chromo is paired to its fitness scores and ranked accroding to its fitness"""
      def rankPop (chromos,iteration):
        outputs, F1, F2 = [], [], []
        i = 1
        for chromo in chromos:
          if sum(chromo)==genesPerCh:
            #not a feasible solution
            output = -9999999999999999999
            F1.append(-999)
            F2.append(-999)
```

```
            outputs.append(output)
          else:
            out = evaluate2(chromo)
            F1.append(out[0])
            F2.append(out[1])
            outputs.append(out[2])

        fitnessScores = calcFitness2 (outputs) # calc fitness scores from the erros ca
    lculated
        pairedPop = zip ( chromos, outputs, F1, F2, fitnessScores) # pair each chromo
    with its ouput and fitness score
        rankedPop = sorted ( pairedPop,key = itemgetter(-
    1) ) # sort the paired pop by descending fitness score
        return rankedPop

      """ taking a ranked population selects two of the fittest members using roulette
     method"""
      """In this way, the better solution has higher possibility to be chosen"""
      def selectFittest (fitnessScores, rankedChromos):
        #print fitnessScores
        count = 0
        while count < 5: # ensure that the chromosomes selected for breeding are have
    different indexes in the population
          index1 = roulette (fitnessScores)
          index2 = roulette (fitnessScores)
          if index1 == index2:
            count +=1
            continue
          else:
            break
        if index1 == index2 and index1-1>-1:
          index2 = index1 - 1
        elif index1 == index2 and index1+1<len(fitnessScores):
          index2 = index1+1

        ch1 = rankedChromos[index1] # select and return chromosomes for breeding
        ch2 = rankedChromos[index2]
        if len(ch1)>genesPerCh:
          print len(ch1)
          print index1
        return ch1, ch2

      """Fitness scores are fractions, their sum = 1. Fitter chromosomes have a larger
     fraction.  """
      def roulette (fitnessScores):
        index = 0
        cumalativeFitness = 0.0
        r = random.random()

        for i in range(len(fitnessScores)): # for each chromosome's fitness score
          cumalativeFitness += fitnessScores[i] # add each chromosome's fitness score
    to cumalative fitness
          if cumalativeFitness > r: # in the event of cumalative fitness becoming grea
    ter than r, return index of that chromo
            #print cumalativeFitness
            #print r
            return i

      """Modify this function to enable the crossover for specific cluster"""
      def crossover (ch1, ch2):
        # at a random chiasma
```

```
1375.         r = random.randint(1,genesPerCh-1)
1376.         ch3 = ch1[:r]+ch2[r:]
1377.         ch4 = ch2[:r]+ch1[r:]
1378.         #print len(ch2)
1379.         #print len(ch4)
1380.         return ch3, ch4
1381.
1382.       """In this version, the mutate function is to randomly change """
1383.       """Modify this function to give the higher probability patch less likely to be c
    hanged"""
1384.       def mutate (ch,probList):
1385.         # if len(ch)>genesPerCh:
1386.         #   print "mutate ch problem!"
1387.         global mutation_rate
1388.         mutatedCh = []
1389.         for i in range(genesPerCh):
1390.           #print i
1391.           if probList[i] > 0.8 or probList[i] < 0.2:
1392.             mutation_rate = mutation_rate * 0.5
1393.
1394.           if random.random() < mutation_rate:
1395.             mutatedCh.append(1-ch[i])
1396.           else:
1397.             mutatedCh.append(ch[i])
1398.
1399.         #assert mutatedCh != ch
1400.         #after mutate, we need to make sure the total area is still under control
1401.         #also the distance thing is under control too
1402.         mutatedCh2 = checkChro(mutatedCh,probList)
1403.         # if sum(mutatedCh2) == 0:
1404.         #   print "check2"
1405.         #   relaxChro(mutatedCh2)
1406.         return mutatedCh2
1407.
1408.       """Using breed and mutate it generates two new chromos from the selected pair"""

1409.       def breed (ch1, ch2,probList):
1410.         newCh1, newCh2 = [], []
1411.         if random.random() < crossover_rate: # rate dependent crossover of selected ch
    romosomes
1412.           newCh1, newCh2 = crossover(ch1, ch2)
1413.         else:
1414.           newCh1, newCh2 = ch1, ch2
1415.         #random select one to mutate
1416.         if random.randint(0,1) == 0:
1417.           newnewCh1 = mutate (newCh1,probList) # mutate crossovered chromos
1418.           # if sum(newnewCh1)==0:
1419.           #   print "check7"
1420.           return newnewCh1
1421.         else:
1422.           newnewCh2 = mutate (newCh2,probList)
1423.           # if sum(newnewCh2)==0:
1424.           #   print "check7"
1425.           return newnewCh2
1426.
1427.       """ Taking a ranked population return a new population by breeding the ranked on
    e"""
1428.       def iteratePop (rankedPop,probList):
1429.         fitnessScores = [ item[-
    1] for item in rankedPop ] # extract fitness scores from ranked population
```

```
1430.         rankedChromos = [ item[0] for item in rankedPop ] # extract chromosomes from r
    anked population
1431.         #print '%s'%(rankedChromos)
1432.         newpop = []
1433.         #newpop.extend(rankedChromos[:popN/15]) # known as elitism, conserve the best
    solutions to new population
1434.         #Reserve the best two chromosomes
1435.         for aa in range(bestN):
1436.           newpop.append(rankedChromos[aa])
1437.
1438.         a = bestN
1439.         # Add one chromosome with random change
1440.         if len(newpop) != popN:
1441.           ch1 = rankedChromos[a]
1442.           ch2 = mutate(ch1,probList)
1443.           newpop.append(ch2)
1444.         while len(newpop) != popN:
1445.           ch1, ch2 = [], []
1446.           ch1, ch2 = selectFittest (fitnessScores, rankedChromos) # select two of the
    fittest chromos
1447.           ch3 = breed (ch1, ch2,probList) # breed them to create one new chromosome
1448.           if sum(ch3)==0:
1449.             print "check6"
1450.           if sum(ch3)>genesPerCh:
1451.             print "problem for breed"
1452.           newpop.append(ch3) # and append to new population
1453.         return newpop
1454.
1455.       def iteratePop2 (rankedPop,probList):
1456.         fitnessScores = [ item[-
    1] for item in rankedPop ] # extract fitness scores from ranked population
1457.         rankedChromos = [ item[0] for item in rankedPop ] # extract chromosomes from r
    anked population
1458.         #print '%s'%(rankedChromos)
1459.         newpop = []
1460.         #newpop.extend(rankedChromos[:popN/15]) # known as elitism, conserve the best
    solutions to new population
1461.         #Reserve the best two chromosomes
1462.         for aa in range(bestN):
1463.           newpop.append(rankedChromos[aa])
1464.
1465.         a = bestN
1466.         while len(newpop) != popN:
1467.           chromo = []
1468.           for bit in range(genesPerCh):
1469.             if probList[bit] > 0.9:
1470.               chromo.append(1) #The reserved ones
1471.             else:
1472.               if random.random() < probList[bit]:
1473.                 chromo.append(1)
1474.               else:
1475.                 chromo.append(0)
1476.
1477.           c1 = checkChro(chromo,probList)
1478.           if sum(c1)==0:
1479.             print "check12"
1480.           if sum(c1)>genesPerCh:
1481.             print "check chromo here2"
1482.           newpop.append(c1)
1483.         return newpop
1484.
```

```
1485.       def configureSettings ():
1486.         configure = raw_input ('T - Enter Target Number \tD - Default settings: ')
1487.         match1 = re.search( 't',configure, re.IGNORECASE )
1488.         if match1:
1489.           global target
1490.           target = input('Target int: ' )
1491.
1492.
1493.       def main():
1494.         #print sys.argv
1495.         #S = sys.argv[1]
1496.         #K = sys.argv[2]
1497.         #A = sys.argv[3]
1498.         #al = sys.argv[4]
1499.         givenCluster = [8,3,1]
1500.         klt = 0
1501.         S = 10000
1502.         K = 3
1503.         A = 0.5
1504.         al = 1
1505.         F1 = 0
1506.         F2 = 0
1507.         global areaConP
1508.         areaConP = float(A)/float(K)
1509.         #Just modify these three parameters
1510.         global distThreshold
1511.         distThreshold = int(S)
1512.         global p
1513.         p = int(K)
1514.         global areaTConP
1515.         areaTConP = float(A)
1516.         global alpha
1517.         alpha = float(al)
1518.         global bestOutputFile
1519.         bestOutputFile = "C:\\GIS\\landconservation\\work\\simulateData100poly\\bestCh
    roms"+str(alpha)+".txt"
1520.         beginTime = datetime.datetime.now()
1521.         init ()
1522.         global f1
1523.         f1 = open(outputFile,'w')
1524.         #global f2
1525.         f2 = open(bestOutputFile,'w')
1526.         #global f3
1527.         #f3 = open(timeFile,'w')
1528.         #global f4
1529.         #f4 = open(clusterFile,'w')
1530.         print >> f1, 'Iteration\tGA Generation\tPerformance\tDistance\tBiodiversity'
1531.         print >> f2, 'Iteration\tChromosome\tPerformance\tF1\tF2\tNumber of new\tTotal
    Area of new\tNumber of all\tTotalArea of all'
1532.         #print >> f3, 'Iteration\tTime'
1533.         #print >> f4, 'Iteration\tCluster'
1534.
1535.         iterations = 0
1536.         outStr = "" #the output result
1537.         out = 0.0 #the performance
1538.         global noClusSet
1539.
1540.       #  while iterations != max_iterations and solution_found != True and len(noClusS
    et)<Ccombs:
1541.         while iterations <= max_iterations:
1542.           print "iteration=" + str(iterations)
```

```
1543.           print "maxIterations=" + str(max_iterations)
1544.
1545.           #print "aa" + str(iterations)
1546.           iteration2 = 0 # the loop for local improvement
1547.           global cList
1548.           global cpList
1549.           global opVal
1550.
1551.           #print '\nCurrent iterations:', iterations+1
1552.           if iterations == 0:
1553.             cList = selCluster(clusterNum,p, givenCluster)
1554.
1555.           #global probList
1556.           probList = setPatchesProb1(clusMark,distMask3,cList)
1557.           #probList2 = setPatchesProb2(clusMark,distMask3,cList)
1558.           #probList3 = setPatchesProb3(clusMark,distMask3,cList)
1559.           #chromos = generatePop(probList,probList2,probList3,popN)
1560.           chromos = generatePop2(probList,popN)
1561.           #maybe at the first step, try to see whether we need to further conduct GA
1562.           rankedPop = rankPop(chromos,0)
1563.           topPop = rankedPop[0]
1564.           if sum(topPop[0])==genesPerCh:
1565.             # no need to do GA anymore
1566.             #print "No GA for cluster "+str(cList)
1567.             iteration2 = max_iterations2
1568.           if sum(topPop[0])>genesPerCh:
1569.             print "topPop problem"
1570.       #    global mutation_rate
1571.       #    mutation_rate = 0.9
1572.           while iteration2 < max_iterations2:
1573.             rankedPop = rankPop(chromos,iteration2)
1574.             topPop = rankedPop[0]
1575.
1576.             print >> f1, '%s\t%s\t%s\t%s\t%s' %(iterations,iteration2,-
    topPop[1],topPop[2],topPop[3])
1577.             # we need to consider the situation that all the solutions are the same, t
    hat means they all don't fit the constraints
1578.             if sum(topPop[0]) == 0:
1579.               print "something is wrong"
1580.             elif sum(topPop[0]) > genesPerCh:
1581.               print "problem rankedpop"
1582.             else:
1583.               chromos = iteratePop(rankedPop,probList)
1584.             print str(iteration2)
1585.             if topPop[1] > -999:
1586.               print str(klt) + "\t" + str(-float(topPop[1])) + "\t" + str(-
    float(topPop[2])) + "\t" + str(-float(topPop[3]))
1587.               fOut.write(str(klt) + "\t" + str(-float(topPop[1])) +"\n")
1588.               klt +=1
1589.       #      print "iteration2=" + str(iteration2)
1590.             iteration2 += 1
1591.       #      print "maxIterations2=" + str(max_iterations2)
1592.
1593.
1594.           # After certain number of iterations, get the best one
1595.           rankedPop = rankPop(chromos,iterations+1)
1596.           topPop = rankedPop[0]
1597.           #print "Below is the best one: solution, output, fitness"
1598.           #print out the best solution
1599.           solution=[] #the final cluster result
1600.           areaT = 0 #The total area of selected patches
```

```
1601.           areaTN = 0 # The total area of newly selected patches
1602.           countP = 0 # The number of patches that are selected
1603.           countPN = 0 # The total number of newly selected patches
1604.           for aa in range(genesPerCh):
1605.             solution.append(topPop[0][aa]*clusMark[aa])
1606.             if topPop[0][aa]==1:
1607.               areaT += float(areaList[aa])
1608.               countP += 1
1609.               if int(indexList[aa]) != 1:
1610.                 areaTN += float(areaList[aa])
1611.                 countPN += 1
1612.           #Print to screen
1613.       #    if topPop[1] > -99:
1614.       #        print "iterations = "
1615.       #        print str(iterations) + "\t" + str(-float(topPop[1]))
1616.       #        fOut.write(str(iterations) + "\t" + str(-float(topPop[1])) +"\n")
1617.
1618.       #    print str(solution)+"\t"+str(topPop[1])+"\t"+str(topPop[2])
1619.           if iterations!=0:
1620.             if float(topPop[1]) > opVal:
1621.       #        print "Cluster selection is better than last one!"
1622.               #found a better one, swap the cluster solution based on this one
1623.               opVal = float(topPop[1])
1624.               F1 = float(topPop[2])
1625.               F2 = float(topPop[3])
1626.               cpList = list(cList)
1627.               noClusSet.add(tuple(cList))
1628.               #cList = swapCluster(cList,clusterNum,p)
1629.
1630.               # only print it out when it's improved
1631.               print >> f2, '%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s' %(iterations+1,solutio
    n,-topPop[1],topPop[2],topPop[3],countPN,areaTN,countP,areaT)
1632.               #print solution
1633.               #print distMask2
1634.               #iterations = iterations+1
1635.               #out = 10000 - opVal
1636.               out = -opVal
1637.             else:
1638.               #print "Cluster selection is worse than last one!"
1639.               #If it's not better, go back to the old list, and swap based on that one

1640.               cList = swapCluster(cpList,clusterNum,p)
1641.               #Put this one into the tabu one
1642.               noClusSet.add(tuple(cList))
1643.
1644.               #print "The length of the no cluster set is: "+ str(len(noClusSet))
1645.           else:
1646.             opVal = float(topPop[1])
1647.             F1 = float(topPop[2])
1648.             F2 = float(topPop[3])
1649.             cpList = list(cList)
1650.             print >> f2, '%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s' %(iterations+1,solution,
    -topPop[1],topPop[2],topPop[3],countPN,areaTN,countP,areaT)
1651.             #print solution
1652.             #print distMask2
1653.             iterations = iterations+1
1654.           iterations += 1
1655.           #noClusSet.add(tuple(cList))
1656.           #print >> f2, '%s\t%s\t%s\t%s\t%s\t%s\t%s' %(iteratio.total_seconds()ns+1,so
    lution,topPop[1],countPN,areaTN,countP,areaT)
1657.
```

```
1658.         #print >> f3, '%s\t%s' %(iterations,duration)
1659.
1660.         f1.close()
1661.         f2.close()
1662.         #f3.close()
1663.         #f4.close()
1664.         #print str(Ccombs)
1665.
1666.         #Below is to print out the excluded combinations to see how many we tried
1667.         #outf = open("C:\\GIS\\landconservation\\work\\simulateData100poly\\noClus.txt
    ",'w')
1668.         #for a in noClusSet:
1669.         #  print >> outf, a
1670.         #outf.close()
1671.
1672.         currentTime = datetime.datetime.now()
1673.         duration = (currentTime - beginTime).total_seconds()
1674.         outStr = str(S)+"\t"+str(K)+"\t"+str(A)+"\t"+str(out)+"\t"+str(F1)+"\t"+str(F2
    )+"\t"+str(duration)
1675.         print outStr
1676.       if __name__ == "__main__":
1677.           main()
1678.       fOut.close()
```