Real-World Application of a Real-Valued Genetic Algorithm

Joshua Haas

Independent Study

Dr. Tinkham

2015-05-16

Motivation

This work is based on work I performed last semester implementing a real-valued genetic algorithm in python. While this was moderately successful, my implementation often failed to converge, even after many (i.e. 10,000) generations. Therefore, my main goal in this project was to improve the stopping criteria. In addition, I attempted to solve a real-world problem with the algorithm (all previous problems were artificially created test equations).

Background

Optimization consists of finding the inputs to a function that yield the smallest (or largest) output. Traditional methods such as gradient descent are highly prone to getting stuck in local optima. A local optimum has a gradient of zero, but there are other points in the search space that would yield a better answer. This can be seen below in Fig. 1 where the global minimum is at $x = 0$, but some algorithms may get "stuck" in the local minima at $x = \pm 2\pi$. Genetic algorithms have been proposed as a meta-heuristic approach to improve the odds of finding a global optimum in the search space.

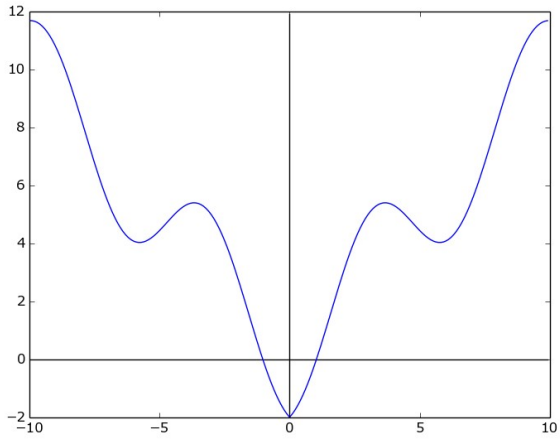

Figure 1. Graph showing global and local minima.

Genetic algorithms are based on the theories of evolution and natural selection, whereby on the fittest individuals in a population survive to reproduce. This project deals specifically with real-valued genetic algorithms, as opposed to discrete genetic algorithms. The algorithm tracks a population of points in the search space where each member of the population has a value for every variable in the optimization problem. For each generation, the members of the population are evaluated using a fitness function (the equation to be optimized) based on the variables of each member. The worst performing individuals are then removed from the population, and the remaining members are bred to create new children. This involves crossover, whereby children receive a random mixture of their parents' genes, and mutation, whereby children randomly mutate. The removal and breeding processes ensure some measure of convergence, and the mutation aspect continues searching the entire problem space for the duration of the algorithm. Given infinite time, a genetic algorithm should theoretically find the global optimum via mutation even if the population converges toward a local optimum.

Stopping Criteria

Deciding when to stop a genetic algorithm is not an easy task [1]. The simplest stopping criteria is to terminate after a set number of generations have passed. However, this is inaccurate and does not check for any form of convergence. The stopping criteria I settled on in the first iteration of this project was so-called genotypical because it was based on the variables of the population members. The algorithm stopped when the variance of the genes of the members passed below a threshold. In many cases, the algorithm never actually stopped under this criteria, instead running for the maximum 10,000 generations. In most of these cases, it found a global minimum earlier, but due to the slow speed at which genetic algorithms descend gradients, it failed to converge. Thus in this project I tried phenotypical stopping criteria instead [2]. In this method, the algorithm keeps track of the best fitness valued achieved in the past *n* generations. It terminates when the variance of this history falls below a threshold. This introduces a new user-defined parameter, the history window.

Previous Implementation

The previous implementation was derived from personal insight and some tips from reading pertinent literature. I settled on a crossover scheme of Gaussian combination, where the children's genes are pulled from a Gaussian distribution centered on the mean of the parents and with a standard deviation equal to half the distance between the parents. This promotes convergence since about 68% of values will fall between the parents, but also allows exploration of the local problem space since about 32% of values will fall outside the range of the parents. The algorithm was an elitist genetic algorithm, since the best performing parents are kept in the population to compete with the children [3]. However, my implementation implements population-wide elitism, rather than family-wide elitism. Mutation consisted of generating a completely random member to add to the population.

Methods and Changes for this Project

The main change is the implementation of the fitness variance stopping criteria. Although the previous algorithm was almost always successful in finding the global minimum in the test functions, it often never stopped until reaching its maximum number of generations. In many cases, these runs found the approximate global minimum in an early generation but then failed to converge as the algorithm performed steepest descent. In changing the stopping criteria, the algorithm would sometimes terminate after only one or two generations. To prevent this behavior, I created a new parameter that allows the user to set a minimum number of generations the algorithm must complete before stopping. All code is available in Appendix B.

The same test functions from the first iteration were tested again using the new stopping criteria [4]. As before, each function was tested using a population of 1000 members, tolerance of 10^{-12}, 100 minimum generations, and 10,000 max iterations. In addition, these tests were performed using a fitness history of the 100 most recent best fitness values. As before, the code relies on the SciPy libraries [5].

I chose to investigate a common problem facing many people and companies, namely the optimal placement of wireless access points. This problem and solution is implemented in the “genetic-wifi.py” file. The user first creates a Room (from “room.py”) by adding Walls (from “wall.py”) to take them into account in attenuation calculation. Walls are naively assumed to have an additive attenuation of 5dB [6,7,8,9]. Regular attenuation is calculated using the Free Space Path Loss formula shown below as Eq. 1, where d is the distance in meters, f is the frequency in hertz, and c is the speed of light

in meters per second. This problem is a good fit for a genetic algorithm because the introduction of the walls create discrete discontinuities that traditional algorithms tend to struggle with.

$$FSPL(dB) = 20\log_{10}\left(\frac{4\pi df}{c}\right) \qquad \text{Eq. 1}$$

Attenuation is naively calculated on the straight path from the potential access point to each user location, and the average of these calculations over all user locations is the fitness function. The user then supplies several locations where they want good reception, which can be weighted to place different priorities on different locations. I constructed an approximate blueprint for my apartment in the Whitney Center and used this to test the algorithm. The blueprint is shown below as Fig 2, with dimensions listed in feet. The access point is forbidden from being in the bathroom or the maintenance closet directly above it. The red circles indicate points where I would like a strong WiFi signal, and are explained to the right of the figure. The blue circles indicate the points found by the algorithm given different weights. The algorithm was tested once by weighting the locations, in order, as [1,1,1,1,1] and then also as [2,1,0.5,0.5,0.5]. The first run places equal priority on all locations. In the second case, I prioritize locations that do not have ethernet near them.

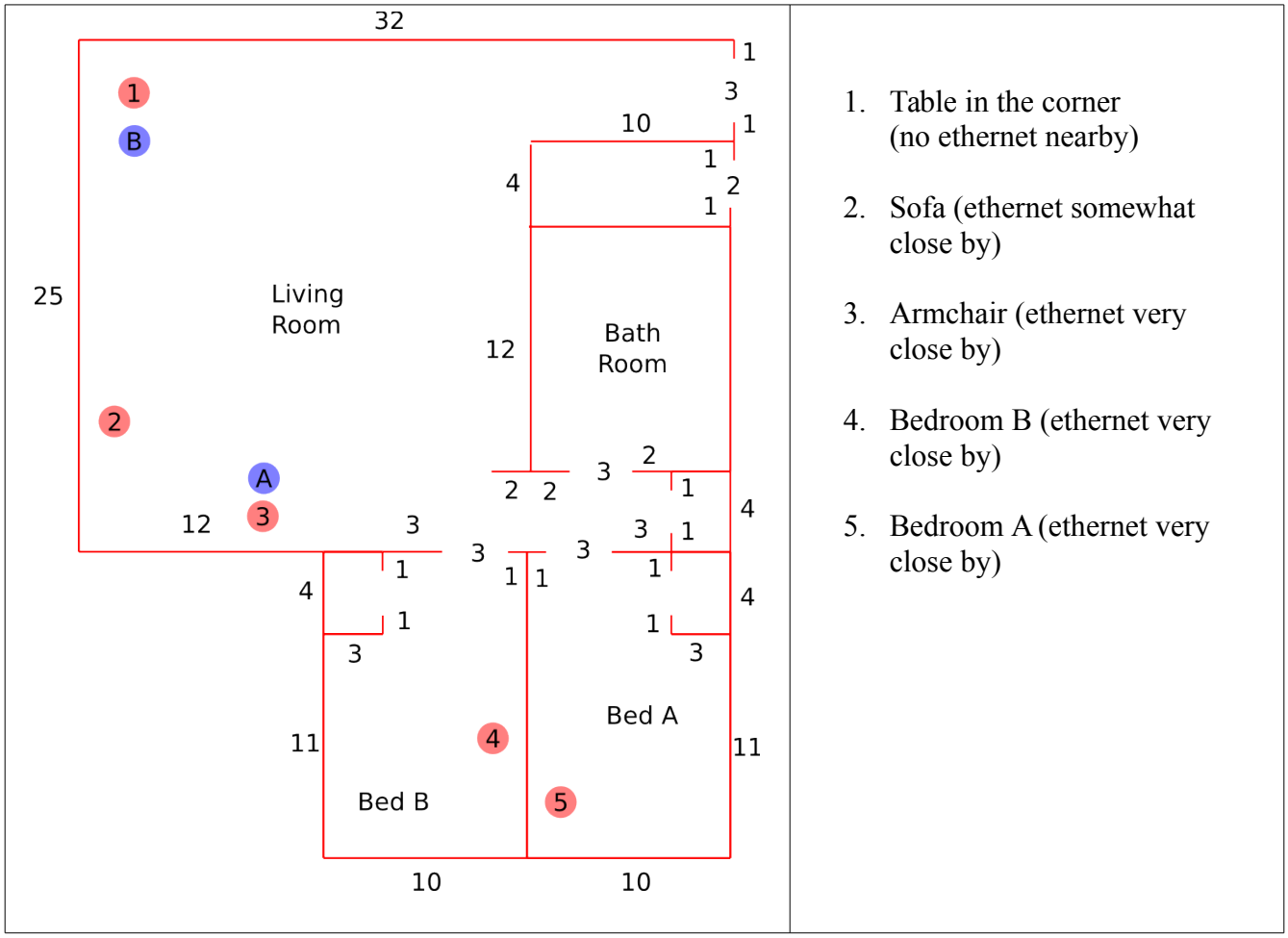

Figure 3. Approximate floor plan of my apartment.

Results

The results of running the modified algorithm on the test problems can be seen below in Table 1. The previous results are included in Appendix A. A comparison of the modified algorithm's performance to the original algorithm's performance is presented in Fig. 4 below. The blue bars indicate the generations required for the original algorithm, and the red bars indicate the generations required for the modified algorithm. Note that black bars indicate values of 10,000 for the original algorithm. These values were replace with "500" to make the smaller runs more visible.

Table 1. Results of the modified genetic algorithm for various test functions with 5 simulations each.

Function	Converged	Global	Min Gen	Mean Gen	Max Gen	Min Err	Mean Err	Max Err
ackley	5 / 5	5 / 5	133	135	136	0	0	0
sphere	5 / 5	5 / 5	149	151.6	153	0	0	0
rosenbrock	5 / 5	4 / 5	239	389.2	588	0	49.8	249
beale	5 / 5	5 / 5	118	121.4	125	0	0	0
goldstein	5 / 5	5 / 5	121	121	121	0	0	0
booth	5 / 5	5 / 5	114	117.6	121	0	0	0
bukin6	5 / 5	1 / 5	436	468	511	0.0016	0.0094	0.0239
matyas	5 / 5	5 / 5	113	115	118	0	0	0
levi13	5 / 5	5 / 5	118	120.4	121	0	0	0
threehump	5 / 5	5 / 5	112	113.2	115	0	0	0
easom	5 / 5	5 / 5	129	131.6	135	0	0	0
crosstray	5 / 5	5 / 5	107	112	119	0	0	0
eggholder	5 / 5	0 / 5	125	328.6	475	2.722	2.899	3.603
holdertable	5 / 5	5 / 5	129	132.8	137	0	0	0
mccormick	5 / 5	5 / 5	108	111.8	114	0	0	0
schaffer2	5 / 5	5 / 5	117	127.8	133	0	0	0
schaffer4	5 / 5	0 / 5	123	171.4	229	0.208	0.208	0.208
styblinski	5 / 5	5 / 5	114	117	119	0	0	0
simionescu	5 / 5	5 / 5	118	123	126	0	0	0

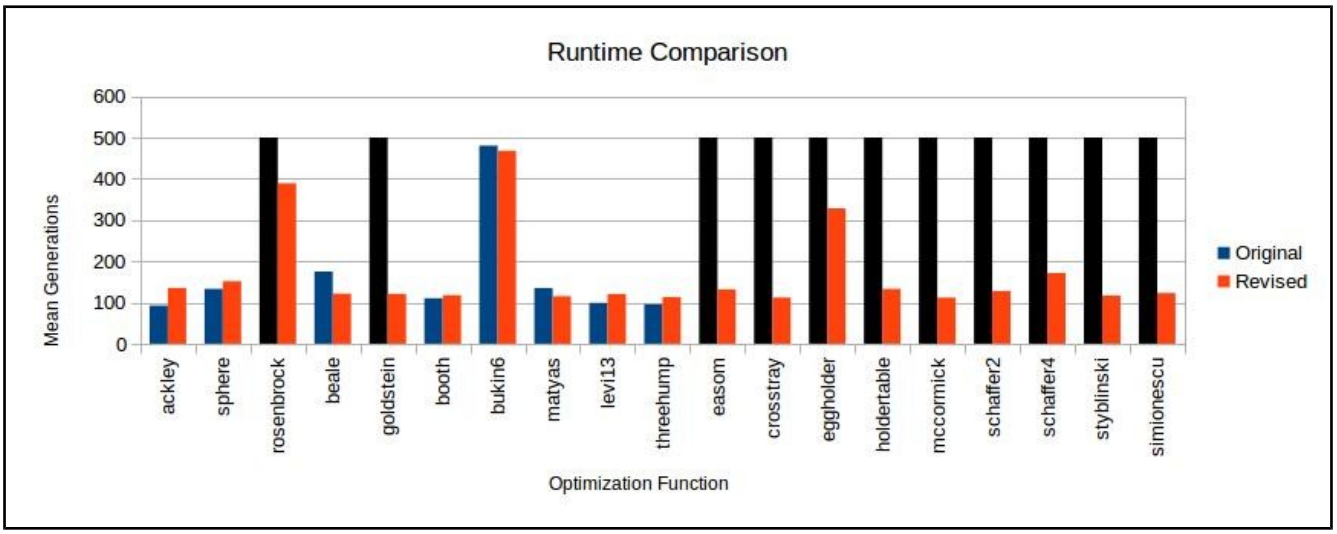

Figure 4. Comparison of performance on test problems.

The results from the WiFi access point optimization can be seen in Fig. 3. For equal weighted locations, the algorithm chose (8.9,18.9) as optimum in 149 generations. (The origin is taken from the bottom left of the bounding box of the floor plan.) For ethernet-weighted locations, the algorithm chose (4.1,35.0) as optimum in 128 generations. Both of these results make sense given the parameters.

Conclusion

The results on the test data showed almost no change in accuracy from the original algorithm. The only exception is on the eggholder function, for which the modified version never found the global minimum. In terms of speed and convergence, the modified algorithm is clearly superior. Using the new stopping criterion, it is almost always able to stop earlier than the original implementation while retaining just as much accuracy. The new phenotypical stopping criteria is obviously better, in general, than the genotypical one.

The WiFi access point results are reasonable. In fact, the actual access point in my apartment is located almost exactly in the place specified by the algorithm for the equal weights run. This supports the assertion that the algorithm is, in fact, solving the WiFi problem, since the building designers likely performed a similar survey. The result for ethernet-weighting makes sense since there is very little weight near the bottom of the apartment.

After initial testing, I added two additional constraints to the population members. They must not be closer than one foot to any user specified location. This is important because getting too close to an access point actually degrades the connection quality. In addition, I added the constraint that connections do not improve below 20dB attenuation. Neither of these significantly affected the results.

References

[1] M. Safe et al. “On Stopping Criteria for Genetic Algorithms.” Advances in Artificial Intelligence, 2004.

[2] D. Bhandari et al. “Variance as a Stopping Criterion for Genetic Algorithms with Elitist Model.” Fundamenta Informaticae, vol. 120, pg. 145-164.

[3] S. Mashohor et al. “Elitist Selection Schemes for Genetic Algorithm based Printed Circuit Board Inspection System.” IEEE Congress on Evolutionary Computation, vol. 2, 2005, pg. 974-978.

[4] R. Oldenhuis. “Test functions for global optimization algorithms.” Mathworks File Exchange, 2014.

[5] E. Jones et al. “SciPy: Open Source Scientific Tools for Python.” Online. 2001. http://www.scipy.org/

[6] Liveport. “WiFi Signal Attenuation.” Online. 2015.

[7] R. Wilson. “Propagation Losses Through Common Building Materials.” Magis Networks, Inc. 2002.

[8] D. Faria. “Modeling Sinal Attenuation in IEEE 802.11 Wireless LANs.” Stanford University. Department of Computer Science.

[9] “How to check if two given line segments intesect?” Online. http://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/.

Appendix A – Test problem results from original report

Function	Converged	Global	Min Gen.	Mean Gen.	Max Gen.	Min Err.	Mean Err.	Max Err.
ackley	5 / 5	5 / 5	91	92.4	94	0	0	0
sphere	5 / 5	5 / 5	132	132.8	134	0	0	0
rosenbrock	0 / 5	4 / 5	10000	10000	10000	0	5.28	26.4
beale	5 / 5	5 / 5	152	175	249	0	0	0
goldstein	0 / 5	5 / 5	10000	10000	10000	0	0	0
booth	5 / 5	5 / 5	108	110.2	112	0	0	0
bukin6	5 / 5	1 / 5	444	480.4	512	0.0018	0.0078	0.0138
matyas	5 / 5	5 / 5	129	135.2	144	0	0	0
levi13	5 / 5	5 / 5	98	99	101	0	0	0
threehump	5 / 5	5 / 5	94	95.6	97	0	0	0
easom	0 / 5	5 / 5	10000	10000	10000	0	0	0
crosstray	0 / 5	5 / 5	10000	10000	10000	0	0	0
eggholder	0 / 5	5 / 5	10000	10000	10000	0	0	0
holdertable	0 / 5	5 / 5	10000	10000	10000	0	0	0
mccormick	0 / 5	5 / 5	10000	10000	10000	0	0	0
schaffer2	0 / 5	5 / 5	10000	10000	10000	0	0	0
schaffer4	0 / 5	0 / 5	10000	10000	10000	0.208	0.208	0.208
styblinski	0 / 5	5 / 5	10000	10000	10000	0	0	0
simionescu	0 / 5	5 / 5	10000	10000	10000	0	0	0

Appendix B

Population.py

```
#!/usr/bin/env python
#
# The Population class implements a real-valued genetic algorithm
# for finding the minimum of a function.
#
# Author: Joshua A Haas
# Data: 2014-12-01

import time,random
import numpy as np

class Population:

  def __init__(self,fit,con,mins,maxs,
      pop=None,pop_size=1000,kill_rate=0.5,mut_rate=0.25):
    """Create a new population for genetic optimization"""

    # Set required args
    self.fit_fun = fit
    self.con_fun = con
    self.mins = mins
    self.maxs = maxs

    # Set optional args
    self.pop = pop
    self.pop_size = pop_size
    self.kill_rate = kill_rate
    self.mut_rate = mut_rate

    # Other setup
    self.genes = len(mins)
    self.hist = np.array([])

    if self.pop is None:
      self.pop = np.zeros((self.pop_size,self.genes))

    self.fitness = np.zeros(self.pop_size)

    self.checkparams()

    random.seed(time.time())

  def checkparams(self):
    """test all parameters for validity"""
```

```
  # Type assertions
  assert callable(self.fit_fun), 'Parameter fit must be function'
  assert callable(self.con_fun), 'Parameter con must be function'
  assert isinstance(self.mins,np.ndarray), 'Parameter mins must be
numpy.ndarray'
  assert isinstance(self.maxs,np.ndarray), 'Parameter maxs must be
numpy.ndarray'
  assert isinstance(self.pop,np.ndarray), 'Parameter pop must be
numpy.ndarray'
  assert isinstance(self.pop_size,int), 'Parameter pop_size must be
int'
  assert isinstance(self.kill_rate,(int,float)), 'Parameter
kill_rate must be int or float'
  assert isinstance(self.mut_rate,(int,float)), 'Parameter mut_rate
must be int or float'

  # Individual assertions
  assert len(self.mins.shape)==1, 'Parameter mins must be row
vector'
  assert len(self.maxs.shape)==1, 'Parameter maxs must be row
vector'
  assert len(self.pop)==self.pop_size, 'Parameter pop must be of
length pop_size'
  assert self.pop_size>0, 'Parameter pop_size must be greater than
0'
  assert (self.kill_rate>=0) and (self.kill_rate<1), 'Parameter
kill_rate must be in [0,1)'
  assert (self.mut_rate>=0) and (self.mut_rate<1), 'Parameter
mut_rate must be in [0,1)'

  # Dependant assertions
  assert self.pop.shape[0]==self.pop_size, 'Parameter pop must have
pop_size rows'
  assert self.pop.shape[1]==self.genes, 'Parameter pop must have
genes cols'
  assert self.mins.shape==self.maxs.shape, 'Parameters mins and
maxs must be same shape'
  assert all(self.maxs>self.mins), 'Parameter maxs must be greater
than mins'

def randpop(self):
  """generate a random population"""

  for r in xrange(0,self.pop_size):
    self.pop[r,:] = self.getrandmem()
  self.updatefit()

def getrandmem(self):
  """generate a random member using uniform dist"""
```

```
  feas = False
  mem = np.zeros(self.genes)
  while not feas:
    for g in xrange(0,self.genes):
      mem[g] = random.uniform(self.mins[g],self.maxs[g])
    feas = self.isfeasible(mem)
  return mem

def isfeasible(self,mem):
  """check if a member is feasible"""

  return ((self.con_fun(mem)) and
    all(mem>self.mins) and
    all(mem<self.maxs))

def updatefit(self,ind=None):
  """update fitness values"""

  if ind is None:
    ind = xrange(0,self.pop_size)
  for i in ind:
    self.fitness[i] = self.fit_fun(self.pop[i,:])

def evolve(self,maxiter=1000,converge=True,tol=1e-
 6,hist=100,mingen=1):
  """evolve the population until convergence or maxiter"""

  converged = False
  i = 0
  while ((not converged) or (not converge)) and (i<maxiter):
    i += 1
    dead = self.getkilled()
    self.breedpop(dead)
    self.updatefit(dead)
    if converge:
      converged = self.isconverged(tol,hist,mingen)
  return i

def getkilled(self):
  """return indices of randomly killed"""

  killnum = int(self.kill_rate*self.pop_size)
  return self.fitness.argsort()[-killnum:]

def getunfit(self,pcnt):
  """return the indices of the most unfit"""

  unfit = int(self.pop_size*pcnt)
```

```
  return self.fitness.argsort()[-unfit:]

def getfit(self,pcnt):
  """return the indices of the most fit"""

  fit = int(self.pop_size*pcnt)
  return self.fitness.argsort()[:-(fit+1)]

def breedpop(self,dead):
  """breed new members replacing the given indices"""

  alive = np.setdiff1d(xrange(0,self.pop_size),dead)
  for d in dead:
    self.pop[d] = self.breedmem(alive)

def breedmem(self,alive):
  """breed a new member from those that are alive"""

  parents = random.sample(alive,2)
  p1 = self.pop[parents[0]]
  p2 = self.pop[parents[1]]
  mu = (p1+p2)/2
  sigma = abs(p2-mu)
  mem = np.zeros(self.genes)
  feas = False
  while not feas:
    for g in xrange(0,self.genes):
      if random.uniform(0,1)<self.mut_rate:
        mem[g] = random.uniform(self.mins[g],self.maxs[g])
      else:
        mem[g] = random.gauss(mu[g],sigma[g])
    feas = self.isfeasible(mem)
  return mem

def isconverged(self,tol,hist,mingen):
  """check for convegence"""

  self.hist = np.append(self.hist,self.getmin()[1])

  if(len(self.hist)==1):
    return False
  if(len(self.hist)<mingen):
    return False
  if(len(self.hist)>hist):
    self.hist = self.hist[1:]

  fit = self.pop[self.getfit(0.5)]
  var = np.var(self.hist)
  return var<tol
```

```
def getmin(self):
  """return the best member as (variables,cost)"""

  minp = self.fitness.argsort()[0]
  return (self.pop[minp],self.fitness[minp])
```

poptest.py

```
#!/usr/bin/env python
#
# This script can be used to test the Population class with various
# optimization test functions.
#
# Author: Joshua A Haas
# Data: 2014-12-01

import math,time
import numpy as np
from population import Population

VALID = (['ackley','sphere','rosenbrock','beale','goldstein',
          'booth','bukin6','matyas','levi13','threehump',
          'easom','crosstray','eggholder','holdertable','mccormick',
          'schaffer2','schaffer4','styblinski','simionescu'])

FUNCTION = 'simionescu'

def main():
  """run the algorithm on the selected FUNCTION"""

  # get the parameters for the chosen FUNCTION
  (fitness,cons,mins,maxs,globmin) = getfunc(FUNCTION)

  # setup the Population
  p = Population(fitness,cons,mins,maxs,pop_size=1000)
  p.randpop()

  # keep track of iterations and time the algorithm
  iters = 0
  t1 = time.time()

  # run for up to 10000 generations, or until convergence
  # print current best every 10 generations
  for i in xrange(0,int(1e3)):
    add = p.evolve(maxiter=10,tol=1e-12,hist=100,mingen=100)
    iters += add
    m = p.getmin()
    x1 = m[0][0]
    x2 = m[0][1]
    y = m[1]
    print str(iters).zfill(5)+': ['+str(x1)+','+str(x2)+'] = '+str(y)
    if add<10:
      break

  # print final results
```

```
  print '\n'+str(iters)+' generations in '+str(time.time()-t1)+'
   secs'
  print '\nCalced: ['+str(round(x1,6))+','+str(round(x2,6))+'] =
   '+str(round(y,6))
  x1 = globmin[0][0]
  x2 = globmin[0][1]
  y = globmin[1]
  print 'Actual: ['+str(round(x1,6))+','+str(round(x2,6))+'] =
   '+str(round(y,6))

def getfunc(fun):
  """return the cost function, constraint function, mins, maxs, and
  globalmin information for the given string fun or raise an error
  if the given string is invalid"""

  if fun=='ackley':
    fun = ackley
    cons = nocons
    mins = np.array([-5,-5])
    maxs = np.array([5,5])
    globmin = (np.array([0,0]),0)
  elif fun=='sphere':
    fun = sphere
    cons = nocons
    mins = np.array([-1000000,-1000000])
    maxs = np.array([1000000,1000000])
    globmin = (np.array([0,0]),0)
  elif fun=='rosenbrock':
    fun = rosenbrock
    cons = nocons
    mins = np.array([-1000000,-1000000])
    maxs = np.array([1000000,1000000])
    globmin = (np.array([1,1]),0)
  elif fun=='beale':
    fun = beale
    cons = nocons
    mins = np.array([-4.5,-4.5])
    maxs = np.array([4.5,4.5])
    globmin = (np.array([3,0.5]),0)
  elif fun=='goldstein':
    fun = goldstein
    cons = nocons
    mins = np.array([-2,-2])
    maxs = np.array([2,2])
    globmin = (np.array([0,-1]),3)
  elif fun=='booth':
    fun = booth
    cons = nocons
    mins = np.array([-10,-10])
```

```
  maxs = np.array([10,10])
  globmin = (np.array([1,3]),0)
elif fun=='bukin6':
  fun = bukin6
  cons = nocons
  mins = np.array([-15,-3])
  maxs = np.array([-5,3])
  globmin = (np.array([-10,1]),0)
elif fun=='matyas':
  fun = matyas
  cons = nocons
  mins = np.array([-10,-10])
  maxs = np.array([10,10])
  globmin = (np.array([0,0]),0)
elif fun=='levi13':
  fun = levi13
  cons = nocons
  mins = np.array([-10,-10])
  maxs = np.array([10,10])
  globmin = (np.array([1,1]),0)
elif fun=='threehump':
  fun = threehump
  cons = nocons
  mins = np.array([-5,-5])
  maxs = np.array([5,5])
  globmin = (np.array([0,0]),0)
elif fun=='easom':
  fun = easom
  cons = nocons
  mins = np.array([-100,-100])
  maxs = np.array([100,100])
  globmin = (np.array([math.pi,math.pi]),-1)
elif fun=='crosstray':
  fun = crosstray
  cons = nocons
  mins = np.array([-10,-10])
  maxs = np.array([10,10])
  globmin = (np.array([1.34941,1.34941]),-2.06261)
elif fun=='eggholder':
  fun = eggholder
  cons = nocons
  mins = np.array([-512,-512])
  maxs = np.array([512,512])
  globmin = (np.array([512,404.2319]),-959.6407)
elif fun=='holdertable':
  fun = holdertable
  cons = nocons
  mins = np.array([-10,-10])
  maxs = np.array([10,10])
```

```
    globmin = (np.array([8.05502,9.66459]),-19.2085)
  elif fun=='mccormick':
    fun = mccormick
    cons = nocons
    mins = np.array([-1.5,-3])
    maxs = np.array([4,4])
    globmin = (np.array([-0.54719,-1.54719]),-1.9133)
  elif fun=='schaffer2':
    fun = schaffer2
    cons = nocons
    mins = np.array([-100,-100])
    maxs = np.array([100,100])
    globmin = (np.array([0,0]),0)
  elif fun=='schaffer4':
    fun = schaffer4
    cons = nocons
    mins = np.array([-100,-100])
    maxs = np.array([100,100])
    globmin = (np.array([0,1.25313]),0.292579)
  elif fun=='styblinski':
    fun = styblinski
    cons = nocons
    mins = np.array([-5,-5])
    maxs = np.array([5,5])
    globmin = (np.array([-2.903534,-2.903534]),-78.33198)
  elif fun=='simionescu':
    fun = simionescu
    cons = simioinescucons
    mins = np.array([-1.25,-1.25])
    maxs = np.array([1.25,1.25])
    globmin = (np.array([0.84852813,-0.84852813]),-0.072)
  else:
    raise ValueError('Function "'+str(fun)+'" is not a valid
   function')
  return (fun,cons,mins,maxs,globmin)

# Minimum at f(0,0)=0 in x1:[-5,5] x2:[-5,5]
# Converged to f(0,0)=0 in [136,135,135,136,133]
def ackley(genes):
  """cost function for the ackley function"""

  x1 = genes[0]
  x2 = genes[1]
  return (-20*math.exp(-0.2*pow(0.5*(x1**2+x2**2),0.5))-
    math.exp(0.5*(math.cos(2*math.pi*x1)+math.cos(2*math.pi*x2)))+
    math.e+20)

# Minimum at f(0,0)=0 in x1:[-inf,inf] x2:[-inf,inf]
# Converged to f(0,0)=0 in [152,152,149,152,153]
```

```
def sphere(genes):
  """cost function for the sphere function"""

  x1 = genes[0]
  x2 = genes[1]
  return x1**2+x2**2

# Minimum at f(1,1)=0 in x1:[-inf,inf] x2:[-inf,inf]
# Found f(12.179655,149.456782)=248.815412 in 239
# Found f(0.999779,0.999548)=0 in 299
# Found f(1.000038,1.000022)=0 in 588
# Found f(0.999897,0.999798)=0 in 543
# Found f(1.003006,1.006102)=0 in 277
def rosenbrock(genes):
  """cost function for the rosenbrock function"""

  x1 = genes[0]
  x2 = genes[1]
  return (100*pow(x2-x1**2,2)+pow(x1-1,2))

# Minimum at f(3,0.5)=0 in x1:[-4.5,4.5] x2:[-4.5,4.5]
# Converged to f(3,0.5)=0 in [121,121,118,125,122]
def beale(genes):
  """cost function for the beale function"""

  x1 = genes[0]
  x2 = genes[1]
  return (pow(1.5-x1+x1*x2,2)+
    pow(2.25-x1+x1*pow(x2,2),2)+
    pow(2.625-x1+x1*pow(x2,3),2))

# Minimum at f(0,-1)=3 in x1:[-2,2] x2:[-2,2]
# Found f(0,-1)=3 in [121,121,121,121,121]
def goldstein(genes):
  """cost function for the goldstein function"""

  x1 = genes[0]
  x2 = genes[1]
  return ((1+pow(x1+x2+1,2)*(19-14*x1+3*x1**2-
   14*x2+6*x1*x2+3*x2**2))*
    (30+pow(2*x1-3*x2,2)*(18-32*x1+12*x1**2+48*x2-
   36*x1*x2+27*x2**2)))

# Minimum at f(1,3)=0 in x1:[-10,10] x2:[-10,10]
# Converged to f(1,3)=0 in [116,114,119,121,118]
def booth(genes):
  """cost function for the booth function"""

  x1 = genes[0]
```

```
  x2 = genes[1]
  return (pow(x1+2*x2-7,2)+pow(2*x1+x2-5,2))

# Minimum at f(-10,1)=0 in x1:[-15,-5] x2:[-3,3]
# Converged to f(-9.841941,0.968638)=0.001581 in 487
# Converged to f(-9.498291,0.902175)=0.005017 in 443
# Converged to f(-9.045733,0.818253)=0.009543 in 511
# Converged to f(-9.329035,0.870309)=0.00671 in 463
# Converged to f(-7.609899,0.579106)=0.023901 in 436
def bukin6(genes):
  """cost function for the bukin6 function"""

  x1 = genes[0]
  x2 = genes[1]
  return (100*pow(abs(x2-0.01*x1**2),0.5)+0.01*abs(x1+10))

# Minimum at f(0,0)=0 in x1:[-10,10] x2:[-10,10]
# Converged to f(0,0)=0 in [115,114,115,118,113]
def matyas(genes):
  """cost function for the matyas function"""

  x1 = genes[0]
  x2 = genes[1]
  return (0.26*(x1**2+x2**2)-0.48*x1*x2)

# Minimum at f(1,1)=0 in x1:[-10,10] x2:[-10,10]
# Converged to f(1,1)=0 in [121,121,121,121,118]
def levi13(genes):
  """cost function for the levi13 function"""

  x1 = genes[0]
  x2 = genes[1]
  return (math.sin(3*math.pi*x1)**2+((x1-1)**2)*
    (1+math.sin(3*math.pi*x2)**2)+((x2-1)**2)*
    (1+math.sin(2*math.pi*x2)**2))

# Minimum at f(0,0)=0 in x1:[-5,5] x2:[-5,5]
# Converged to f(0,0)=0 in [115,114,112,113,112]
def threehump(genes):
  """cost function for the threehump function"""

  x1 = genes[0]
  x2 = genes[1]
  return (2*x1**2-1.05*x1**4+(x1**6)/6+x1*x2+x2**2)

# Minimum at f(pi,pi)=-1 in x1:[-100,100] x2:[-100,100]
# Found f(pi,pi)=-1 in [129,132,131,131,135]
def easom(genes):
  """cost function for the easom function"""
```

```
  x1 = genes[0]
  x2 = genes[1]
  return (-math.cos(x1)*math.cos(x2)*
    math.exp(-((x1-math.pi)**2+(x2-math.pi)**2)))

# Minimum at f(+-1.34941,+-1.34941) in x1:[-10,10] x2:[-10,10]
# Found f(-1.349407,1.349407)=-2.062612 in 113
# Found f(1.349407,-1.349407)=-2.062612 in 107
# Found f(1.349407,1.349407)=-2.062612 in 108
# Found f(1.349407,1.349407)=-2.062612 in 113
# Found f(1.349407,-1.349407)=-2.062612 in 119
# Note that this function has 4 global minima
def crosstray(genes):
  """cost function for the crosstray function"""

  x1 = genes[0]
  x2 = genes[1]
  return (-0.0001*(abs(math.sin(x1)*math.sin(x2)*
    math.exp(abs(100-math.sqrt(x1**2+x2**2)/math.pi)))+1)**0.1)

# Minimum at f(512,404.2319)=-959.6407 in x1:[-512,512] x2:[-512,512]
# Found f(482.353276,432.878972)=-956.918232 in 388
# Found f(482.352961,432.878628)=-956.918232 in 475
# Found f(482.353278,432.878967)=-956.918232 in 384
# Found f(480.755988,431.159998)=-956.03721 in 125
# Found f(482.35331,432.878999)=-956.918232 in 271
def eggholder(genes):
  """cost function for the eggholder function"""

  x1 = genes[0]
  x2 = genes[1]
  return (-(x2+47)*math.sin(math.sqrt(abs(x2+x1/2+47)))-
    x1*math.sin(math.sqrt(abs(x1-(x2+47)))))

# Minimum at f(+-8.05502,+-9.66459)=-19.2085 in x1:[-10,10] x2:[-
   10,10]
# Found f(-8.055023,-9.66459)=-19.208503 in 133
# Found f(-8.055023,9.66459)=-19.208503 in 131
# Found f(-8.055023,9.66459)=-19.208503 in 137
# Found f(8.055023,-9.66459)=-19.208503 in 134
# Found f(8.055023,9.66459)=-19.208503 in 129
# Note that this function has 4 global minima
def holdertable(genes):
  """cost function for the holdertable function"""

  x1 = genes[0]
  x2 = genes[1]
  return (-abs(math.sin(x1)*math.cos(x2)*
```

```
    math.exp(abs(1-math.sqrt(x1**2+x2**2)/math.pi))))

# Minimum at f(-0.54719,-1.54719)=-1.9133 in x1:[-1.5,4] x2:[-3,4]
# Found f(-0.547198,-1.547198)=-1.913223 in 113
# Found f(-0.547198,-1.547198)=-1.913223 in 114
# Found f(-0.547198,-1.547198)=-1.913223 in 111
# Found f(-0.547198,-1.547198)=-1.913223 in 113
# Found f(-0.547198,-1.547198)=-1.913223 in 108
def mccormick(genes):
  """cost function for the mccormick function"""

  x1 = genes[0]
  x2 = genes[1]
  return (math.sin(x1+x2)+(x1-x2)**2-1.5*x1+2.5*x2+1)

# Minimum at f(0,0)=0 in x1:[-100,100] x2:[-100,100]
# Found f(0,0)=0 in [131,131,127,133,117]
def schaffer2(genes):
  """cost function for the schaffer2 function"""

  x1 = genes[0]
  x2 = genes[1]
  return (0.5+((math.sin(x1**2-x2**2)**2-0.5)/
    (1+0.001*(x1**2+x2**2))**2))

# Minimum at f(0,1.25313)=0.292579 in x1:[-100,100] x2:[-100,100]
# Found f(99.514633,-97.706601)=0.500097 in 123
# Found f(99.992107,99.999962)= 0.500095 in 155
# Found f(99.337472,98.360113)=0.500096 in 189
# Found f(-99.671221,99.553099)=0.500094 in 161
# Found f(99.99199,99.021126)=0.500093 in 229
def schaffer4(genes):
  """cost function for the schaffer4 function"""

  x1 = genes[0]
  x2 = genes[1]
  return (0.5+((math.cos(math.sin(abs(x1**2-x2**2)))-0.5)/
    (1+0.001*(x1**2+x2**2))**2))

# Minimum at f(-2.903534,-2.903534)=-78.33198 in x1:[-5,5] x2:[-5,5]
# Found f(-2.903534,-2.903534)=-78.332331 in [119,114,117,118,117]
def styblinski(genes):
  """cost function for the styblinski function"""

  x1 = genes[0]
  x2 = genes[1]
  return (((x1**4-16*x1**2+5*x1)+(x2**4-16*x2**2+5*x2))/2)

# Minimum at f(+-0.84852813,-+0.84852813)=-0.072 in x1:[-1.25,1.25]
```

```
   x2:[-1.25,1.25]
# Found f(-0.848592,0.848464)=-0.072 in 121
# Found f(0.848469,-0.848587)=-0.072 in 118
# Found f(-0.848477,0.848579)=-0.072 in 126
# Found f(0.848626,-0.84843)=-0.072 in 124
# Found f(-0.84852,0.848535)=-0.072 in 126
def simionescu(genes):
  """cost function for the simionescu function"""

  x1 = genes[0]
  x2 = genes[1]
  return (0.1*x1*x2)


def simioinescucons(genes):
  """constraint function for the simionescu function"""

  x1 = genes[0]
  x2 = genes[1]
  return x1**2+x2**2<=(1+0.2*math.cos(8*math.atan(x1/x2)))**2

def nocons(genes):
  """constraint function for functions with only mins and maxs"""

  return True

if __name__ == '__main__':
  main()
```

genetic-wifi.py

```
#!/usr/bin/env python
#
# This script optimizes the placement of a Wireless Access point in a
# specific room.
#
# Author: Joshua A Haas
# Date: 2015-04-28

import math,time
import numpy as np
from population import Population
from room import Room
from wall import Wall

WALL_ATTEN = 5
FREQ = 2400000000
LIGHT = 300000000

# table in the corner
LOC1 = (4,36,1)
# sofa farthest from ethernet
LOC2 = (2,23,1)
# armchair next to ethernet
LOC3 = (8,18,1)
# bedroom 1 desk
LOC4 = (21,6,1)
# bedroom 2 desk
LOC5 = (23,3,1)

locs = [LOC1,LOC2,LOC3,LOC4,LOC5]

r = Room()
r.addwall(0,40,32,40)
r.addwall(0,15,0,40)
r.addwall(0,15,18,15)
r.addwall(12,0,12,15)
r.addwall(12,0,32,0)
r.addwall(32,0,32,32)
r.addwall(32,34,32,36)
r.addwall(32,39,32,40)
r.addwall(22,35,32,35)
r.addwall(22,19,22,35)
r.addwall(22,31,32,31)
r.addwall(22,0,22,15)
r.addwall(21,15,23,15)
r.addwall(20,19,24,19)
r.addwall(27,19,32,19)
```

```
r.addwall(26,15,32,15)
r.addwall(29,18,29,19)
r.addwall(29,14,29,16)
r.addwall(29,11,32,11)
r.addwall(29,11,29,12)
r.addwall(12,11,15,11)
r.addwall(15,11,15,12)
r.addwall(15,14,15,15)

def main():

  # Define mins and maxs

  # Setup population
  mins = np.array([0,0])
  maxs = np.array([32,40])
  p = Population(fitness,cons,mins,maxs,pop_size=1000)
  p.randpop()

  iters = 0
  t1 = time.time()

  # run for up to 10000 generations, or until convergence
  # print current best every 10 generations
  for i in xrange(0,int(1e3)):
    add = p.evolve(maxiter=10,tol=1e-12,hist=100,mingen=100)
    iters += add
    m = p.getmin()
    x1 = m[0][0]
    x2 = m[0][1]
    y = m[1]
    print str(iters).zfill(5)+': ['+str(x1)+','+str(x2)+'] = '+str(y)
    if add<10:
      break

  # print final results
  print '\n'+str(iters)+' generations in '+str(time.time()-t1)+'
   secs'
  print '\nCalced: ['+str(round(x1,6))+','+str(round(x2,6))+'] =
   '+str(round(y,6))

# Constraint function to keep members inside the house
def cons(genes):

  x = genes[0]
  y = genes[1]

  # Bounding box of the layout
  if(x<=0 or x>=32 or y<=0 or y>=40):
```

```
    return False

  # Do not include blank area in bottom left
  if(x<=12 and y<=15):
    return False

  # Do not include bathroom or utility room
  if(x>=22 and x<=32 and y>=19 and y<=31):
    return False

  # Do not allow the point to be closer than 1 ft to any loc
  for loc in locs:
    wall = Wall(x,y,loc[0],loc[1])
    if(wall.length<1):
      return False

  return True

# Fitness function to determine optimal placement given weighted
   locations
def fitness(genes):

  total = 0
  for loc in locs:
    wall = Wall(genes[0],genes[1],loc[0],loc[1])

    fspl = 20*math.log(4*math.pi*wall.length*0.3048*FREQ/LIGHT,10)
    amp = fspl + WALL_ATTEN*r.num_intersects(wall)

    # below 20 db gives no improvement
    amp = max(amp,20)

    total += loc[2]*amp

  return total/len(locs)

if __name__ == '__main__':
  main()
```

room.py

```
#!/usr/bin/env python
#
# Room class to keep track of walls and calculate intersections
#
# Author: Joshua Haas

from wall import Wall

class Room:

  def __init__(self):

    self.walls = []

  def num_intersects(self,wall):
    """count number of intersects between the given wall and all
   walls in this room"""

    count = 0
    for w in self.walls:
      if wall.intersects(w):
        count += 1

    return count

  def addwall(self,x1,y1,x2,y2):
    """add the wall checking for duplicates"""

    wall = Wall(x1,y1,x2,y2)
    for w in self.walls:
      if wall==w:
        return

    self.walls.append(wall)

  def ft2m(self):
    """convert all walls from ft to meters"""

    for w in self.walls:
      w.ft2m()
```

wall.py

```
#!/usr/bin/env python
#
# Wall class to hold points and perform operations
#
# Author: Joshua Haas

import math

HORIZ = 0
VERT = 1

class Wall:

  def __init__(self,x1=None,y1=None,x2=None,y2=None):
    """Create a new wall"""

    self.x1 = x1
    self.y1 = y1
    self.x2 = x2
    self.y2 = y2

    if(self.x1==self.x2 or self.y1==self.y2):
      self.length = abs(self.x1-self.x2)+abs(self.y1-self.y2)
    else:
      self.length = math.sqrt((self.x1-self.x2)**2+(self.y1-
   self.y2)**2)

    #assert self.length>0, 'Walls must have greater than 0 length'

  def __eq__(self,obj):
    """override equals operator"""

    if not isinstance(obj,Wall):
      return False

    if(self.x1==obj.x1 and self.y1==obj.y1 and self.x2==obj.x2 and
   self.y2==obj.y2):
      return True
    if(self.x1==obj.x2 and self.y1==obj.y2 and self.x2==obj.x1 and
   self.y2==obj.y1):
      return True

    return False

  def __ne__(self,obj):
    """override not equals operator"""
```

```
  return not self==obj


def intersects(self,other):
  """return true if this wall intersects the other"""

  # they do intersect if they share vertices
  if((self.x1==other.x1 and self.y1==other.y1) or
 (self.x2==other.x2 and self.y2==other.y2)):
    return True

  # don't intersect if bounding boxes don't
  if(max([self.x1,self.x2])<min([other.x1,other.x2])):
    return False
  if(max([self.y1,self.y2])<min([other.x1,other.x2])):
    return False

  # check if points are on left and right of other line
  return
 ((self.ccw(self.x1,self.y1,other.x1,other.y1,other.x2,other.y2)
      !=
 self.ccw(self.x2,self.y2,other.x1,other.y1,other.x2,other.y2)) and
      (self.ccw(self.x1,self.y1,self.x2,self.y2,other.x1,other.y1)
      !=
 self.ccw(self.x1,self.y1,self.x2,self.y2,other.x2,other.y2)))

def ccw(self,x1,y1,x2,y2,x3,y3):
  """determine orientation of points"""

  return (y3-y1)*(x2-x1) > (y2-y1)*(x3-x1)

def ft2m(self):
  """convert the values from feet to meters"""

  self.x1 *= 0.3048
  self.y1 *= 0.3048
  self.x2 *= 0.3048
  self.y2 *= 0.3048
```