

Homework 8 Advanced Analytics and Metaheuristics

Group 10: Nicholas Jacob

April 29, 2024

1. Genetic Algorithm

(a) Finalize Code

- i. To create the initial chromosomes, we kept it simple, generating a random number between -500 and 500. We used the global variable n as the number of dimensions.

```
def createChromosome(n):  
    x = []  
    for i in range(n):  
        x.append(myPRNG.uniform(-500,500))  
    return x
```

- ii. For mutation, we played with this quite a bit. Originally we randomly selected an index and the just reseeded it with a random. This seemed to work but we wanted more control over it so we introduced a new parameter that we could tweak, *mutationFactor*. We used this to change one of the indices by a random, $r \in [-1,1]$, with $x[i] = x[i] + r * \text{mutationFactor}$. We could then tweak this factor to get more or less mutation. We did check that it was inside the interval and if not, randomly reseed.

```
def mutate(x):  
  
    if mutationRate > myPRNG.random():  
        i = myPRNG.randint(0,n-1)  
        x[i] += mutationFactor*myPRNG.uniform(-1,1)  
        if (x[i] > 500) or (x[i]<-500):  
            x[i] = myPRNG.uniform(-500,500)
```

```
    return x
```

- iii. For cross over, we picked on index to swap start the swap. Everything after that index would be traded between the parents. This happened with a *crossOverRate*. Some times just the parents would be returned.

```
def crossover(x1,x2):
    p = myPRNG.random()
    if crossOverRate>p:
        z = myPRNG.randint(1,n-1)
        offspring1 = x1[:z] + x2[z:]
        offspring2 = x2[:z] + x1[z:]

    else:
        offspring1 = x1[:]
        offspring2 = x2[:]
```

```
    return offspring1, offspring2 #two offspring are returned
```

- iv. Elitism was achieved by keeping the best numbering to *eliteSolutions* from the parent generation. It was important to resort these as the children were often better. We also made sure to change it so the minimum was first as desired here.

```
def insert(pop,kids):

    newlist = []
    for i in range(eliteSolutions):
        newlist.append(pop[i])
    for kid in kids:
        newlist.append(kid)
    popVals = sorted(newlist, key=lambda newlist: newlist[1], reverse = 1)
    return popVals
```

- v. I want to share the code for computing the function. With *evaluate* we made sure to utilize the numpy speed in dealing with large dimensional data.

```
def evaluate(x):
    x = np.array(x)
    return 418.982887272443*n -sum(x*np.sin(np.sqrt(np.abs(x))))
```

- (b) For parameters, I played around with the code a lot. I noticed for the small dimensional and chromosome, it was important to

have higher mutation rate. I found very good results with the following parameters

```
#number of dimensions in a solution  
n = 2
```

```
Generations = 300    #number of GA generations
```

```
crossOverRate = 0.8  #whether the couple breeds or just continues into next  
mutationRate = 0.25  #rate at which mutation can occur  
eliteSolutions = 10   #number of elite solutions form previous generation tha  
mutationFactor = 50   # the largest number by which the mutation can change a  
populationSize = 80   #size of GA population
```

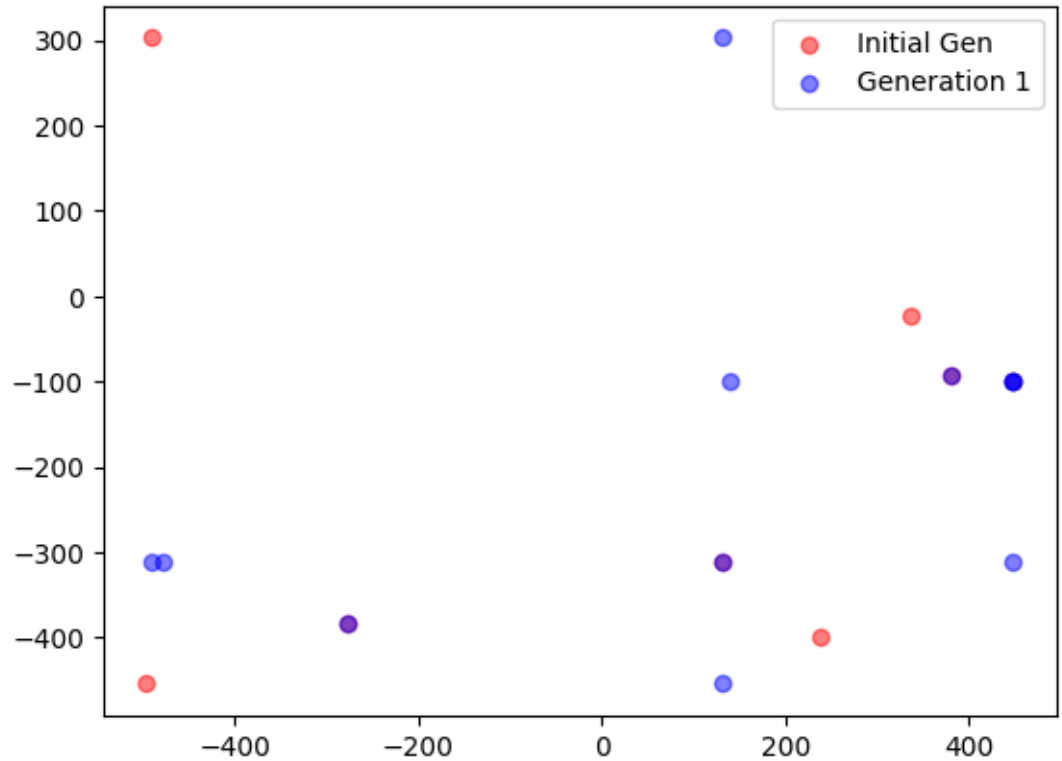
For the higher dimensions, we used

```
#number of dimensions in a solution  
n = 200
```

```
Generations = 1000   #number of GA generations
```

```
crossOverRate = 0.8  #whether the couple breeds or just continues into next  
mutationRate = 0.25  #rate at which mutation can occur  
eliteSolutions = 100  #number of elite solutions form previous generation th  
mutationFactor = 100  # the largest number by which the mutation can change a  
populationSize = 1000 #size of GA population
```

- i. Here is my visualization of 8 chromosomes. We see some genes splicing and we see some repeats. It is also possible we see some mutations.



- ii. This had Best solution: [420.9337473832483, 421.0086992961223]
as the best solution, accurate to three decimal places, 0.0003559856297670194

- (c) This was quite challenging. At 1000 generations, we were still seeing slight improvements. This is with an initial population of 1000 and keeping 100 elite solutions. We saw the value go as low as 18666. While this is nowhere as good as the 2D solution, we do see many entries with a value near the 420 optimal. The execution of this code was slow and with the generations not showing much improvement, do not think expanding beyond 1000 generations will improve the result by much. This problem is hard!

2. Particle Swarm Optimization

(a) Code Basics

i. Particle Tracking:

```
def pbestUpdate(): #update the best lists
    particleUpdate = 0 #how many particles have been updated?
    globalUpdate = 0 #did the global best update
```

```

global pbestestVal, pbestest #get these variables from global environm
for i in range(swarmSize): #iterate through swarm
    if curValue[i]< pbestVal[i]: #is the current better than current best
        pbest[i] = pos[i][:] #since yes, change it up
        pbestVal[i] = curValue[i]
        particleUpdate +=1 #increment the counter
        if curValue[i]<pbestestVal: #since this is better than before, che
            pbestestVal = curValue[i] #since it is, update
            pbestest = pos[i][:]
            globalUpdate +=1
    return particleUpdate, globalUpdate #return counts for stopping criteri
ii. Velocity and Position updates

```

```

def velocityUpdate(): #update velocities
    for i in range(swarmSize): #iterate through swarm
        for j in range(dimensions): #iterate through dimensions
            vel[i][j] = w*vel[i][j]+phi1*myPRNG.random()*(pbest[i][j]-pos[i][j])
            if vel[i][j] >maxVelocity: #check if outside of allowed values
                vel[i][j] = maxVelocity #just return max/min allowed
            if vel[i][j] <-1*maxVelocity:
                vel[i][j] = -1*maxVelocity

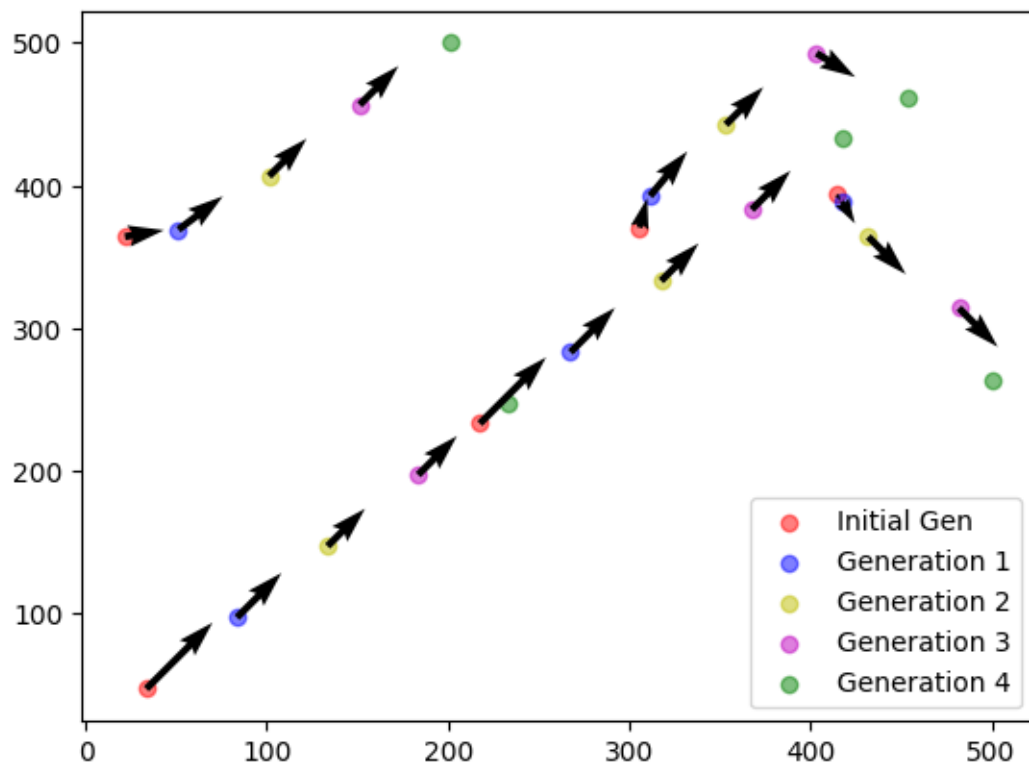
def positionUpdate():#must be called after velocity update
    for i in range(swarmSize): #iterate through swarm
        for j in range(dimensions): #update all dimensions
            pos[i][j] = pos[i][j] + vel[i][j] #follow basic PSO

            if pos[i][j] > upperBound: #check if outside of feasible
                pos[i][j] = upperBound #if so, put on edge
            if pos[i][j] < lowerBound: #outside feasible?
                pos[i][j] = lowerBound
            curValue[i] = evaluate(pos[i]) #update currentvalue too
iii. Limits on particle positions are included in the position up-
    date. I only allow getting on the boundary
iv. Limits on velocity are included in velocityUpdate. Max ve-
    locity is a parameter that can be tweaked later.

```

(b) 2D Schwefel

i. Here is a plot with just positions and velocities



ii. Here is a plot that includes the global best on each iteration

