Benjamin Asdell
CEE/MAE M20
UID: 005114992

Final Project

October 15, 2020

1 Problem Statement

1. Introduction

In this problem, the objective is to solve a ubiquitous scenario in engineering and robotics involving the joints of a robotic arm reaching a goal. If the robotic arm is split into four sections $L = [L_1, L_2, L_3, L_4]$ of decreasing length, certain angle values $\theta = [\theta_1, \theta_2, \theta_3, \theta_4]$ are required to reach the goal. In addition to these parameters, two obstacles that would block the simulated arm's path are also included to simulate the real-world application of the robotic arm.

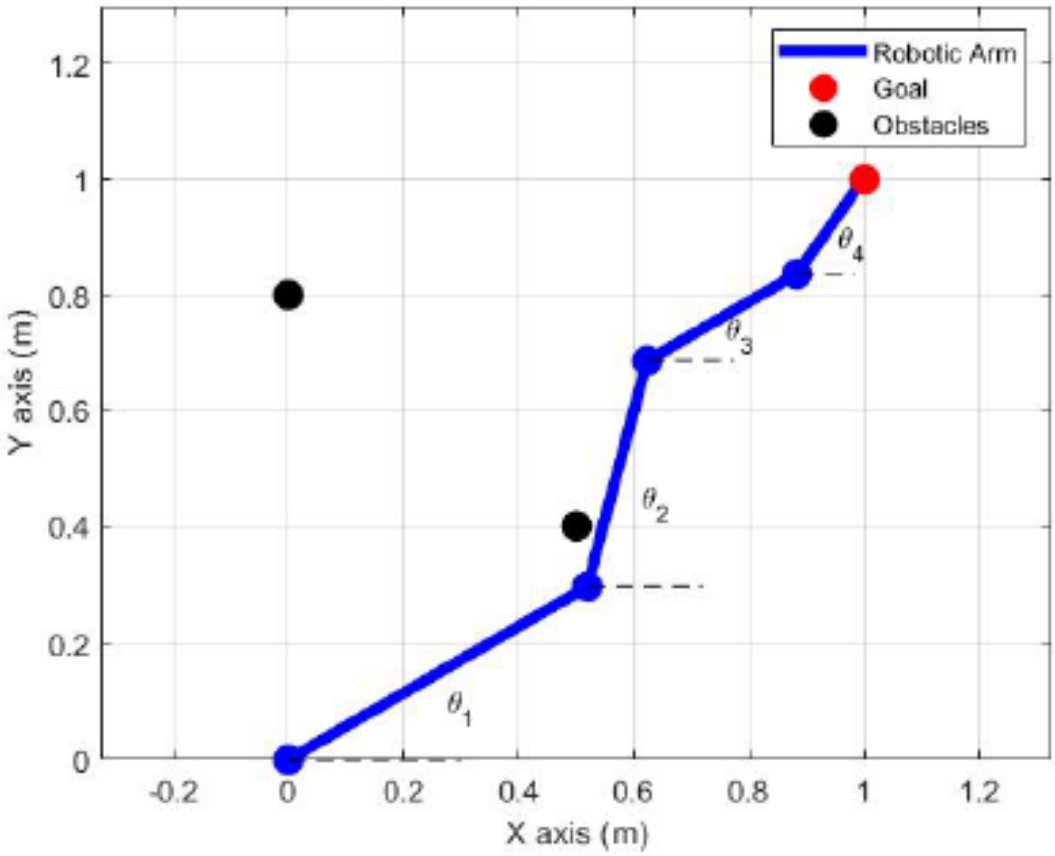

Figure 1: A depiction of a 4-joint robotic arm with obstacles and goal shown. Note this is not necessarily an example of an optimized solution. The angle θ values are defined relative to the horizontal.

In order to find the path that best minimizes the distance from the arm's endpoint to the goal and best maximizes the distance from each section of the arm to the obstacles, the program must use a type of optimizer.

2 Optimization Specifics

1. Genetic Algorithm

   Previous optimization methods used in this class, such as gradient descent and Newton's method, are unequipped to deal with a non-convex three-dimensional optimization problem like this. Therefore, a heuristic genetic algorithm optimizer is used as an efficient alternative to those methods. A genetic algorithm works by operating on a generation to generation basis. Each generation is a set of different individuals or 'chromosomes' that represent different solutions to the problem. These chromosomes (and therefore, solutions) are not necessarily unique as the population is randomly generated at the start and the randomness of the different processes theoretically can produce the same chromosome in different individuals. The genetic algorithm applies a four step process to each generation before assessing the highest fitness of the population. Fitness is best described as a quantified measurement of how optimized the solution is, or rather how close to the goal and far away from the obstacles the specific set of angles generates the robotic arm. When the highest fitness reaches a certain value, the algorithm outputs the angles it has found to be most optimal; this event is called convergence.

2. Note: Algorithm Space and Solution Space

   In this problem, the space in which the solutions to the robotic arm scenario and the encoded data containing those solutions are different. The solution space, or the type of data containing the four angles corresponding to each piece of robotic arm, are simply decimal numbers. More specifically, they are MATLAB type 'double'. This data type is the input to the fitness.m function and can be interpreted by a human as angles in radians. However, to improve algorithmic efficiency that is offered by bitwise operations, each chromosome containing this angle data is encoded as unsigned 32-bit integer (MATLAB uint32). In this algorithm space, each sequence of 8 bits in the 32-bit integer corresponds to one of the angles in the solution space. Since the maximum value for such a data type is 255, there is a limit number of angles that can be represented by the 8 bits. The range (0, 255) in algorithm space is mapped to $(-\pi, \pi)$ in the solution space. To clarify for the rest of this report, the crossover.m and mutation.m functions operate on chromosomes using bitwise operations in the algorithm space.

3. Selection

   The first operation step involves selecting a subpopulation of individual chromosomes to be carried to the next generation. Simply selecting the fittest individuals limits the diversity of the population and actually slows down

convergence. Therefore, the Binary Tournament Selection method is used. This particular method selects two random individuals from the population of the current generation, then compares their fitness values. The fittest individual of the two is then selected for the next generation. Notice that this process has a (small) chance of selecting two very unfit chromosomes and passing on one of them, resulting in a less fit population that if only the fittest members were selected. However, this anomalous result of selection actually contributes to the optimization as more generations pass. This step produces a subpopulation that is one size less than the original population of the current generation.

4. Crossover

   The subpopulation for the next generation then undergoes crossover, a simulation of 'breeding' that allows the genetic algorithm to test various combinations of angles and actually change the population over time. Without crossover, each generation would be extremely similar to the previous, which would clearly slow down convergence significantly. The crossover process takes two consecutive chromosomes from the subpopulation produced by selection, chooses a bit index at which each chromosome is divided, then switches the pieces of the two chromosomes with their complement. This produces two 'daughter' chromosomes that each have one piece from each 'parent' chromosome. Note that crossover has a probability of occurrence, which means that if that probability is less than 1.0, there is a chance crossover does not happen and the daughters are identical copies of their parents. Lastly, the population of the next generation is then updated to replace the parent chromosomes with their daughters.

5. Mutation

   Chromosomal mutation is then implemented upon the subpopulation to introduce randomness into the next generation. This randomness further increases the genetic diversity, which once again improves the speed of convergence. Mutations also occur with a certain probability, meaning that some chromosomes will not be mutated at all. The mutation at the bit level occurs by selecting a random bit index and toggling that bit on or off depending on its original state. This change can have any magnitude of impact on the encoded angle data within the chromosome, but the key is that it allows the genetic algorithm to test a possibly new angle combination.

6. Elitism

   The next process, elitism, is designed to encourage a monotonic increase in the maximum fitness of each generation. The fittest individual from the current generation (the original population that is passed into selection) is chosen to be

appended to the subpopulation. Not only does this step ensure that a very fit individual survives from generation to generation, but also keeps the population size constant since the subpopulation at this point is one member less than the population of the current generation.

3 Pseudocode for Genetic Algorithm and its Steps

Algorithm 1: Genetic Algorithm(geneticAlgorithm.m)

Input:
fitness - function handle for fitness.m function
decode - function handle for decodeChromosome.m function
populationSize - length of population array (number of individuals in each generation)
pCrossover - probability of crossover occurrence
pMutation - probability of mutation occurrence
Output: xOpt - optimal solution in original solution space (angles, in radians)

Initialization:
Randomly populationSize uint32 values into column vector P
while *xOpt not converged* **do**
    Select $n-1$ chromosomes from P into P' using Binary Tournament selection
    Perform crossover on pairs of consecutive chromosomes in P' with probability pCrossover
    Perform mutation on each chromosome in P' with probability pMutation
    Select fittest chromosome, p^*, from P
    Update P to $P' \cup p^*$
Return: xOpt = decode(p^*)

Algorithm 2: Selection (selection.m)

Input:
P - population array of chromosomes representing current generation
fitness - function handle for fitness.m function
decode - function handle for decodeChromosome.m function
Output: P' - subpopulation of chromosomes representing the next generation

Initialization:
Create empty array P' of size one less than P
for *size of* P' **do**
    Select two random chromosomes (p_1, p_2) from P
    Compare fitness of p_1 and p_2 using fitness
    if *one chromosome is fitter than the other* **then**
        Place fitter of the two into P'
    else
        Place p_1 into P'
Return: P'

Algorithm 3: Mutation (mutation.m)

Input:
p_1 - one parent chromosome representing encoded angle values
p_2 - one parent chromosome representing encoded angle values
pCrossover - probability of crossover occurrence
Output: $[p_1', p_2']$ - array containing daughter chromosomes

if *randomly generated number less than pCrossover* **then**
    Randomly generate bit index where crossover will occur
    Create bitmask for all bits up to and including crossover index
    Use bitwise AND with bitmask to extract left hand bits of p_1 and p_2 into p_1^1 and p_2^1
    Reverse bitmask using bitcmp
    Use bitwise AND with bitmask to extract right hand bits of p_1 and p_2 into p_1^2 and p_2^2
    Return: $[p_1', p_2'] = [p_1^2$ bitwise AND p_2^1, p_1^1 bitwise AND p_2^2
else
    Return: $[p_1', p_2'] = [p_1, p_2]$

Algorithm 4: Crossover (crossover.m)

Input:
p - chromosome selected for mutation
pMutation - probability of mutation occurrence
Output: p' - chromosome to be returned to subpopulation

1 **if** *randomly generated number less than pMutation* **then**
2 Randomly generate bit index where mutation will occur
3 Create bitmask with only 1 at that bit index
4 Use bitwise XOR with the bitmask to create p_m
5 **Return:** $p' = p_m$
6 **else**
7 **Return:** $p' = p$

The pseudocode for the next algorithm is part of the extra credit for this project. This implementation of decodeChromosome.m is my interpretation of how to convert the encoded uint32 chromosomes in algorithm space to arrays of four θ values in solution space.

Algorithm 5: Extra Credit: Decode Chromosome (decodeChromosome.m)

Input:
p - chromosome to be decoded
Output: x = array containing four angles (in radians)

1 Create bitmask for each sequence of 8 bits in the uint32 (left to right)
2 Use bitwise AND with the chromosome and each mask (left to right)
3 Bitshift first three results of the previous step until the extracted bits are in the first 8 bits
4 Store each result in an array p (including the last result from step 2)
5 Cast the array p to type double
6 Map each element of the array p to an array θ in solution space
7 **Return:** $x = \theta$

To elaborate on line 6 of the pseudocode, once bit operations provided the four separate 8-bit values corresponding to each angle, the angles are calculated using the following equation.

$$\theta_k = -\pi + p_k * \frac{2\pi}{255} \tag{1}$$

4 Solution and Results

1. Parameters

   (a) The location of the goal is (1.0,1.0).

   (b) The locations of the two obstacles are (0.0, 0.8) and (0.5,0.4).

   (c) The lengths of each piece of the robotic arm are $L = [0.6, 0.4, 0.3, 0.2]$.

   (d) The probability of crossover occurrence, pCrossover, is 0.8.

   (e) The probability of mutation, pMutation, is 0.3.

   (f) The size of the population, populationSize, in each generation is 1001.

   (g) The desired minimum fitness value is 0.99 to reach convergence.

   (h) As written, the program does not take much longer than 20 generations (in most cases) to converge, therefore no generation limit is included.

2. Program Output

   To demonstrate the functionality of the program, the program (main.m) script will be executed with the above parameters. The rest of the results shown are from one example run, and successive calls of the program will not produce these exact results. When the script is run, the following is printed to the command window.

```
The final optimized angles (in radians) are:
0.455839, 0.751518, 1.047198, and 1.466077
```

3. Video Link

   The video of the histogram plot with each successive generation can be found at https://youtu.be/WuOuzyAcrFI. The video itself can also be found in the submitted .zip directory.

4. Figures and Plot

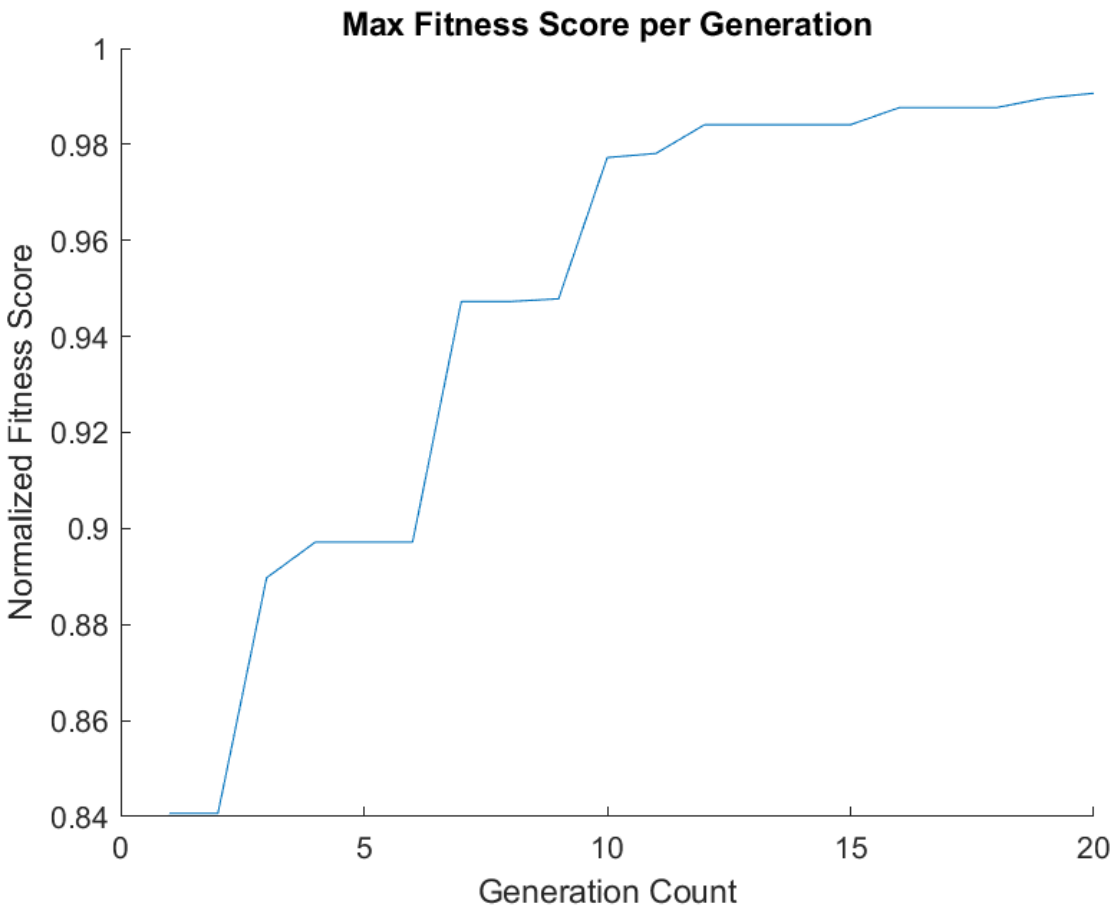

Figure 2: Figure 2 shows the variance of the normalized fitness score of the fittest individual in each generation with the generation number. As desired, the trend of the graph is monotonically increasing, meaning that the algorithm always produces either a more fit or equally fit member in one generation compared to the previous. This particular example converges at generation 20, or rather the first individual to reach a fitness higher than 0.99 is 'born' in the 20th generation. In fact, the 20th generation has a maximum fitness of 0.9906.

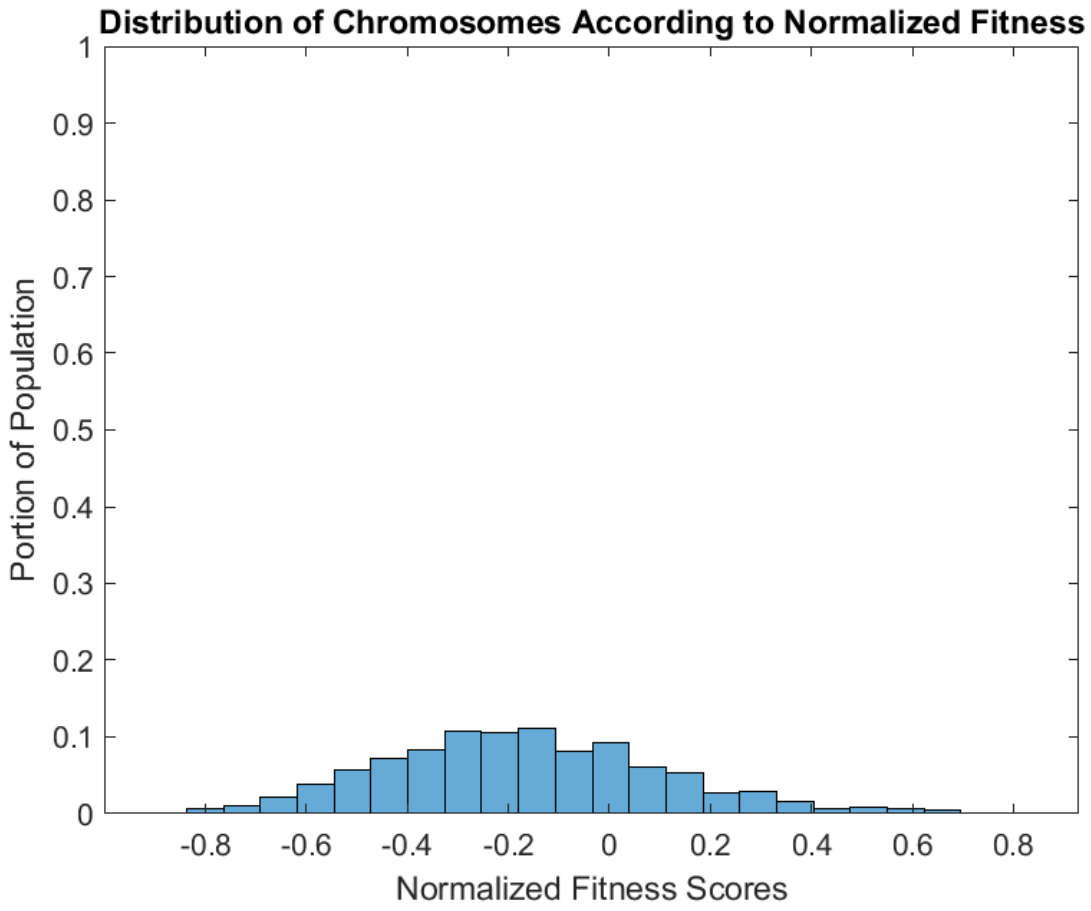

Figure 3: A histogram depicting the distribution of fitness among the entire initial population. Since this population is generated entirely randomly, it is expected that this distribution is weighted towards unfit values.

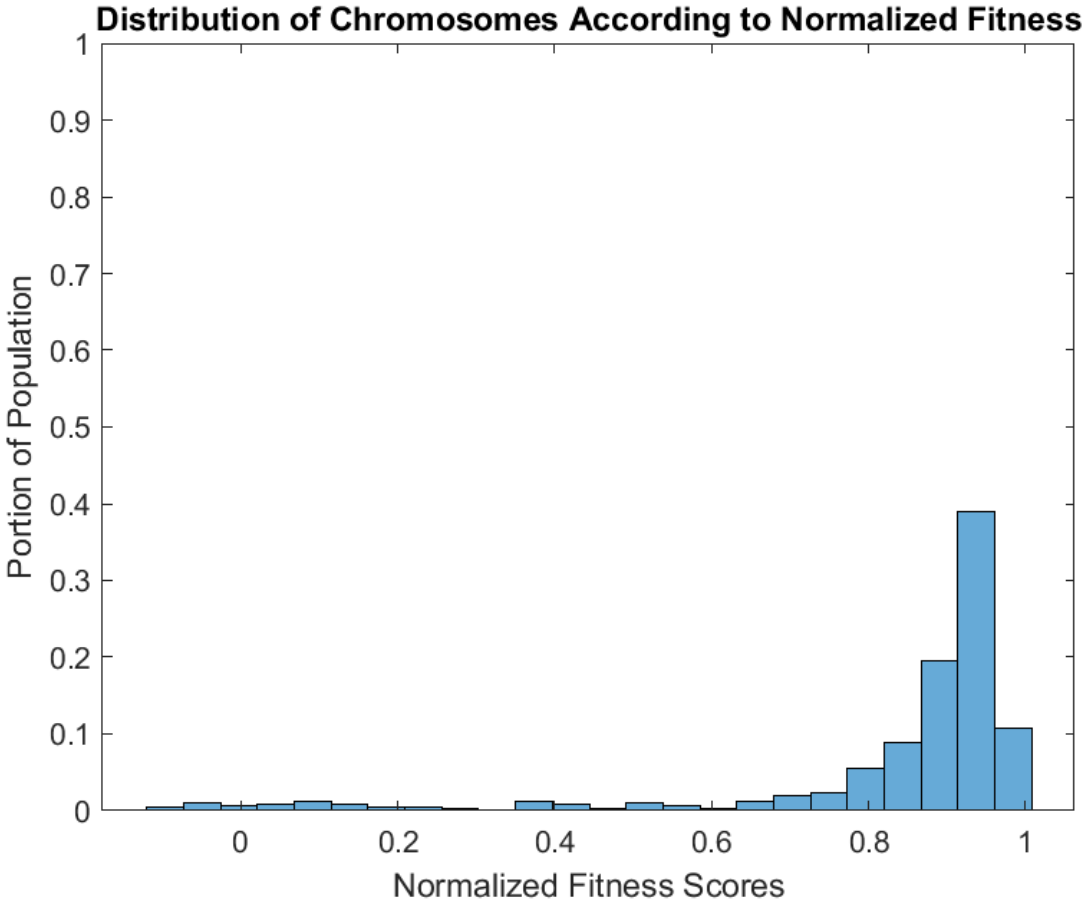

Figure 4: A histogram depicting the distribution of fitness among the entire final population. It is evident that the algorithm is at or near convergence based on the high distribution in the three bins closest to 1 (perfect fit).

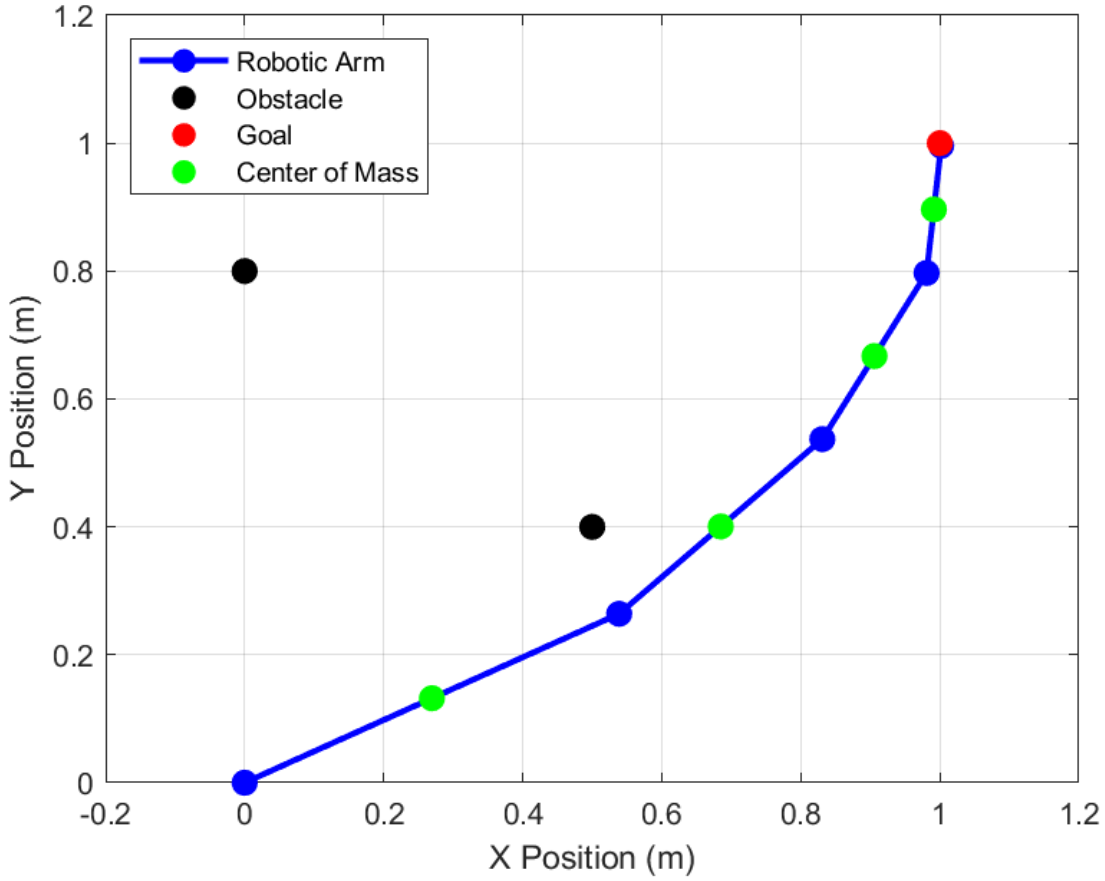

Figure 5: This plot is a graphical representation of the simulated robotic arm. The endpoint of the arm and the goal are extremely close to each other, being nearly indistinguishable at first glance of the plot. Note that in this example (and all others produced in testing), the optimal angles choose to go below both obstacles instead of between them.

5 Discussion

This project is an excellent comprehensive test of everything that this course has taught. The genetic algorithm provides an incredible glimpse into machine learning and how simulation of realistic events within MATLAB can be used to solve real-world problems that are far beyond the scope of human calculation. While this genetic algorithm is designed to solve the specific problem regarding the robotic arm, it can be easily generalized to solve other high dimensional non-convex optimization problems. More specifically, changing the decodeChromosome.m function and therefore the data encoded in each chromosome allows for quick adaptation to new problems. Completing this project has greatly improved my view of the usefulness of computer science in terms of machine learning and similar fields.