Part 1:

I have slightly changed the code provided in class for the basic mga. It is now more efficient and can correctly compare two different genotypes. The difference between the discrete and real is that the discrete is just a binary 0 and 1 difference with the fitness just depending on which has more 1's. The same is for mutation and it's just going to be mutated from 0 to 1 or 1 to 0. The same goes for other things like recombination probability, etc. Mga real are real numbers that propagate between 0 to 1 including decimals. It represents a much fuller range and is probably more close to real problems that involve nature. However, both aren't very suitable to directly represent an evolutionary algorithm that is inspired by nature.
The github link can be found here: https://github.com/rhit-hsinr/EvoRoboPP

Part 2:

The above solves the problem of a simple 0 or 1 and calculates the initial fitness of the genotypes. After winning tournaments, mutations, and recombination, it can be seen from multiple runs that it eventually reaches close to 1 but never 1 for MGA real.

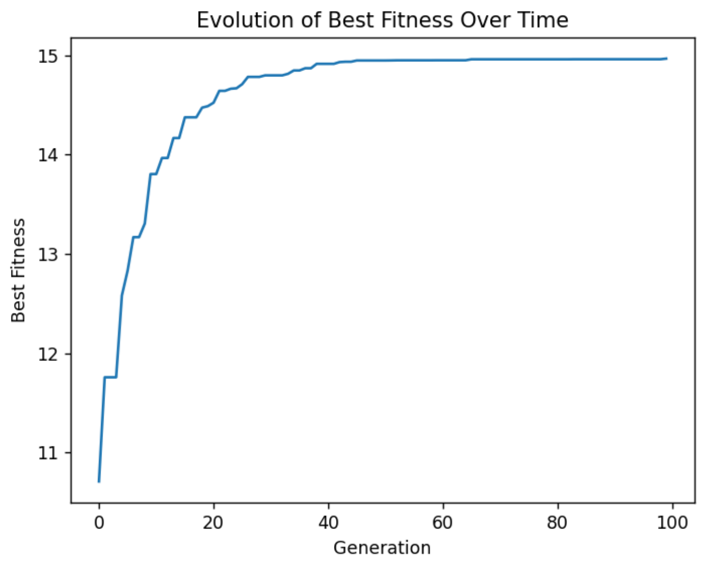
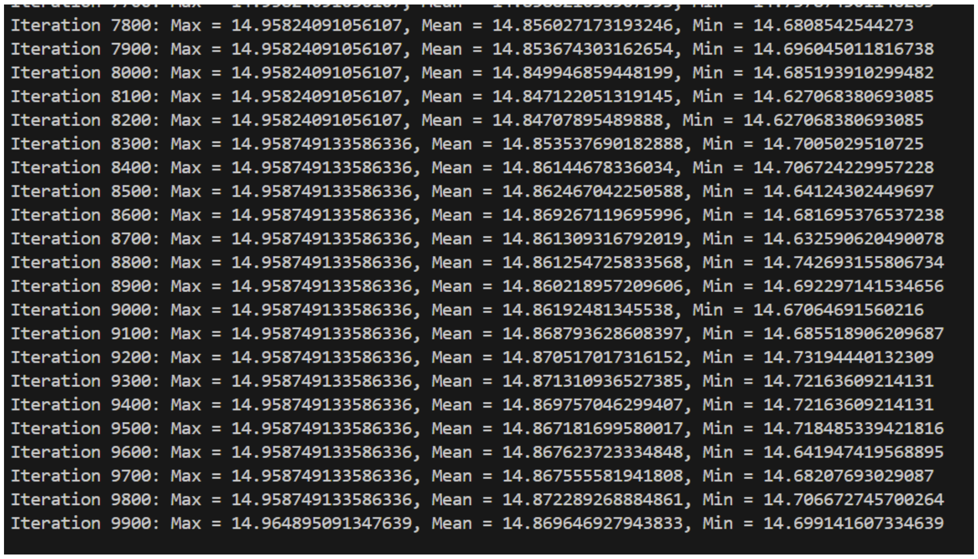


Figure 1 : Evolution of MGA real



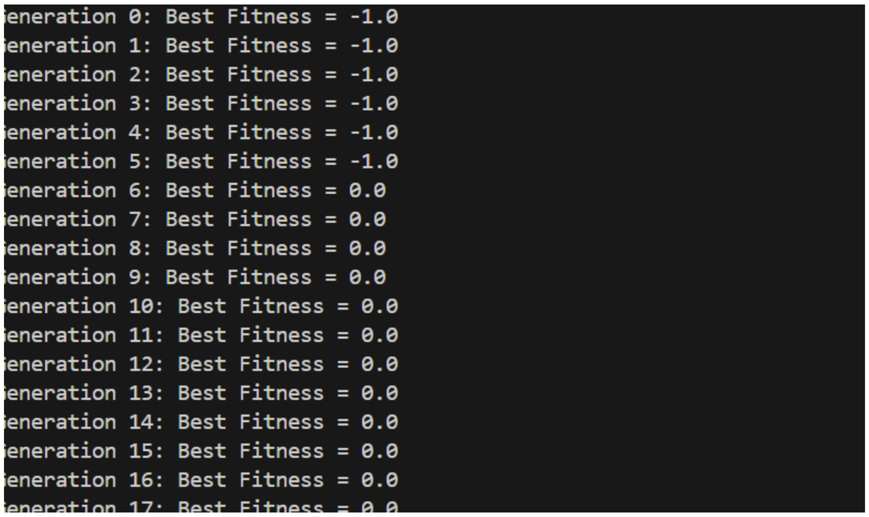
```
Iteration 7800: Max = 14.95824091056107, Mean = 14.856027173193246, Min = 14.6808542544273
Iteration 7900: Max = 14.95824091056107, Mean = 14.853674303162654, Min = 14.696045011816738
Iteration 8000: Max = 14.95824091056107, Mean = 14.849946859448199, Min = 14.685193910299482
Iteration 8100: Max = 14.95824091056107, Mean = 14.847122051319145, Min = 14.627068380693085
Iteration 8200: Max = 14.95824091056107, Mean = 14.84707895489888, Min = 14.627068380693085
Iteration 8300: Max = 14.958749133586336, Mean = 14.853537690182888, Min = 14.7005029510725
Iteration 8400: Max = 14.958749133586336, Mean = 14.86144678336034, Min = 14.706724229957228
Iteration 8500: Max = 14.958749133586336, Mean = 14.862467042250588, Min = 14.64124302449697
Iteration 8600: Max = 14.958749133586336, Mean = 14.869267119695996, Min = 14.681695376537238
Iteration 8700: Max = 14.958749133586336, Mean = 14.861309316792019, Min = 14.632590620490078
Iteration 8800: Max = 14.958749133586336, Mean = 14.861254725833568, Min = 14.742693155806734
Iteration 8900: Max = 14.958749133586336, Mean = 14.860218957209606, Min = 14.692297141534656
Iteration 9000: Max = 14.958749133586336, Mean = 14.86192481345538, Min = 14.67064691560216
Iteration 9100: Max = 14.958749133586336, Mean = 14.868793628608397, Min = 14.685518906209687
Iteration 9200: Max = 14.958749133586336, Mean = 14.870517017316152, Min = 14.73194440132309
Iteration 9300: Max = 14.958749133586336, Mean = 14.871310936527385, Min = 14.72163609214131
Iteration 9400: Max = 14.958749133586336, Mean = 14.869757046299407, Min = 14.72163609214131
Iteration 9500: Max = 14.958749133586336, Mean = 14.867181699580017, Min = 14.718485339421816
Iteration 9600: Max = 14.958749133586336, Mean = 14.867623723334848, Min = 14.641947419568895
Iteration 9700: Max = 14.958749133586336, Mean = 14.867555581941808, Min = 14.68207693029087
Iteration 9800: Max = 14.958749133586336, Mean = 14.872289268884861, Min = 14.706672745700264
Iteration 9900: Max = 14.964895091347639, Mean = 14.869646927943833, Min = 14.699141607334639
```

Figure 2 : Snapshot of the end of the runs

So to solve the Card problem, there needs to be some sort of indication. I will represent the number of 0's as the number of cards that goes into the addition pile and the number of 1's that goes into the multiplication pile.

To tackle this problem, a different fitness function would be needed. It is also needed to track each distribution to represent which index is which number out of the 10 numbers in the bracket. The fitness function would first separate the numbers based on their values, 0 or 1, into an addition pile and a multiplication pile respectively. The addition pile would be using their index + 1 as their value and it will be isolated into a pile that goes into summing. The same would go for multiplication. After the sum and product have been calculated based on the current genotype, it will calculate the difference between 36 and 360 and add the difference together, which represents the current fitness. That means that the best fitness would be 0.

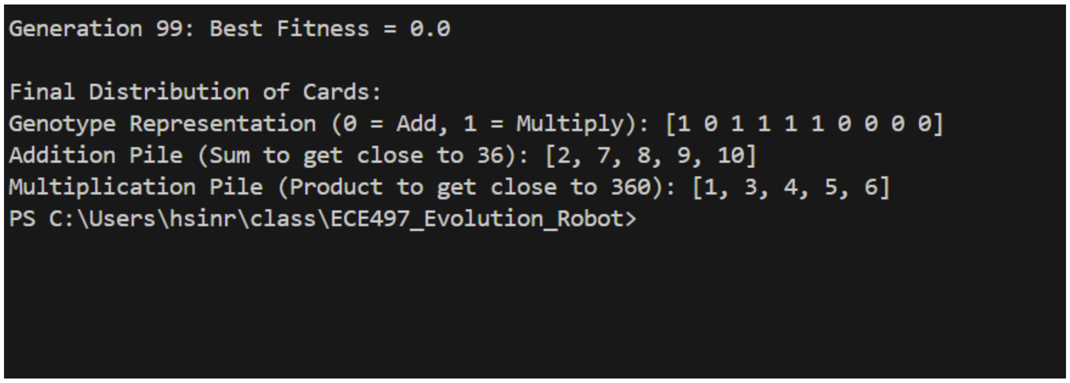
The results can be seen below.



```
ieneration 0: Best Fitness = -1.0
ieneration 1: Best Fitness = -1.0
ieneration 2: Best Fitness = -1.0
ieneration 3: Best Fitness = -1.0
ieneration 4: Best Fitness = -1.0
ieneration 5: Best Fitness = -1.0
ieneration 6: Best Fitness = 0.0
ieneration 7: Best Fitness = 0.0
ieneration 8: Best Fitness = 0.0
ieneration 9: Best Fitness = 0.0
ieneration 10: Best Fitness = 0.0
ieneration 11: Best Fitness = 0.0
ieneration 12: Best Fitness = 0.0
ieneration 13: Best Fitness = 0.0
ieneration 14: Best Fitness = 0.0
ieneration 15: Best Fitness = 0.0
ieneration 16: Best Fitness = 0.0
ieneration 17: Best Fitness = 0.0
```

Figure 3 : snapshot of results.

It can be seen that from generation 6 it has already finished concluding that it can produce the best results and it stayed the same for all the rest of the generations up until generation 100.



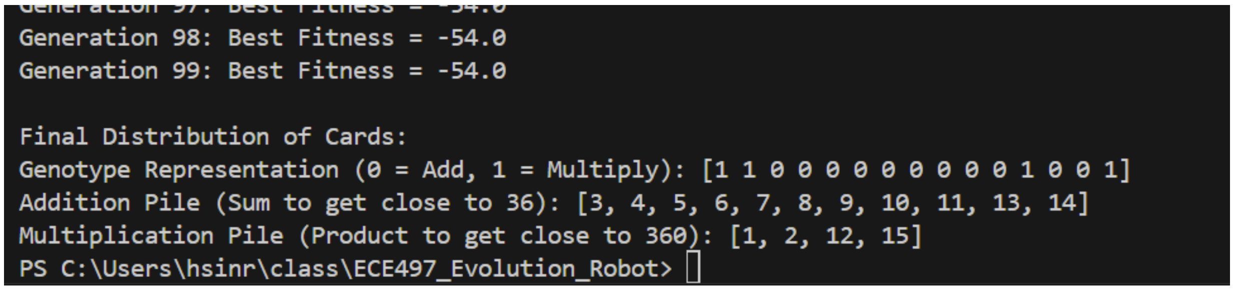
```
Generation 99: Best Fitness = 0.0

Final Distribution of Cards:
Genotype Representation (0 = Add, 1 = Multiply): [1 0 1 1 1 1 0 0 0 0]
Addition Pile (Sum to get close to 36): [2, 7, 8, 9, 10]
Multiplication Pile (Product to get close to 360): [1, 3, 4, 5, 6]
PS C:\Users\hsinr\class\ECE497_Evolution_Robot>
```

Figure 4: Final Distribution of Cards

After representing each, it can be seen that the addition pile sums up to 36 while the multiplication multiplies up to 360. This evolutionary algorithm approach works.

Part 3:

This algorithm doesn't work too well with higher numbers due to nothing to add and nothing to multiply. The best fitness of 15 cards is -54. This card problem isn't really applicable to anything from 10 cards, but this evolutionary algorithm can still be used to find the optimal solution that minimizes the difference.



```
Generation 98: Best Fitness = -54.0
Generation 99: Best Fitness = -54.0

Final Distribution of Cards:
Genotype Representation (0 = Add, 1 = Multiply): [1 1 0 0 0 0 0 0 0 0 0 0 1 0 0 1]
Addition Pile (Sum to get close to 36): [3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14]
Multiplication Pile (Product to get close to 360): [1, 2, 12, 15]
PS C:\Users\hsinr\class\ECE497_Evolution_Robot>
```

Figure 5: Final Fitness of 15 cards

Changing other numbers doesn’t really seem much even if I made the mutation probability high, however, it takes longer for the evolutionary algorithm to realize which is which. Before it was just a full on ramp into a straight line, but this time it took a step due to mutation being set as 1.

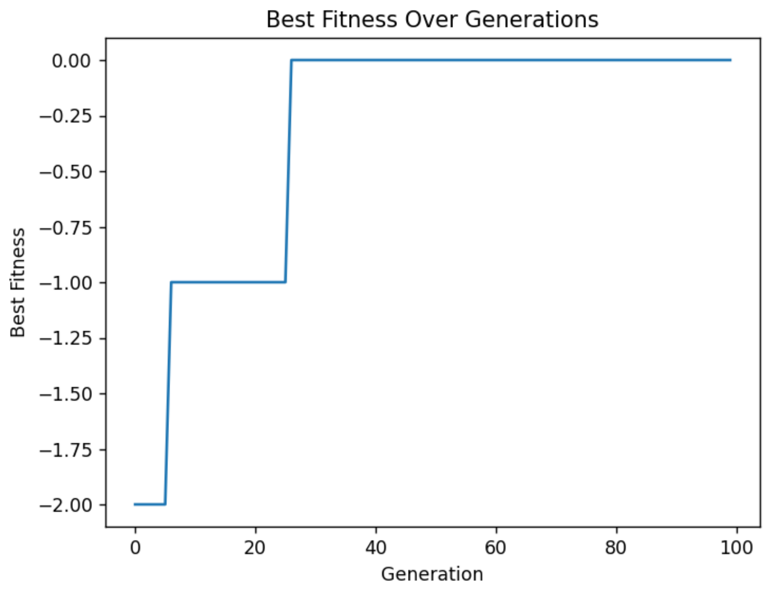
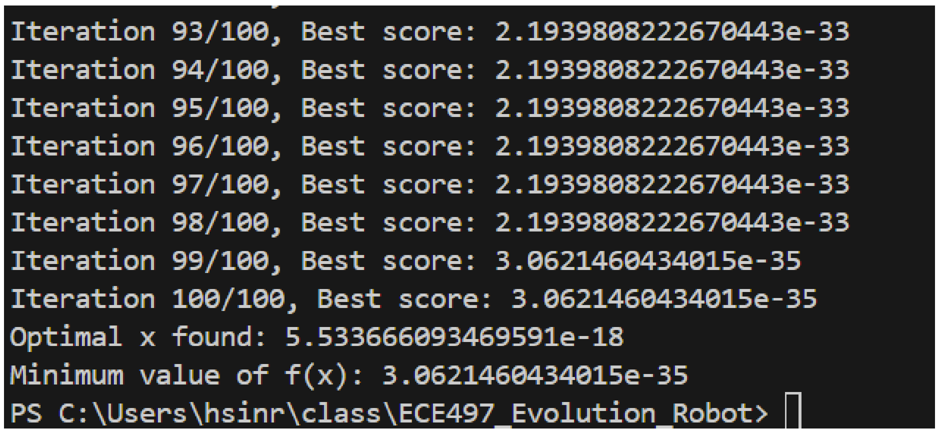


Figure 6: Graph of best fit due to mutation being 1

Part 4:

Now for the application of Nature inspired algorithm. I just researched a few and I found out about the particle swarm optimization, where all the particles will eventually optimize and converge to one optimized solution, like if a whole flock of birds are flying together, one bird that optimizes flying the most will cause all the other ones to fly the same way. I’m using this to just optimize a simple function of x^2 and find the optimization of it.



```
Iteration 93/100, Best score: 2.1939808222670443e-33
Iteration 94/100, Best score: 2.1939808222670443e-33
Iteration 95/100, Best score: 2.1939808222670443e-33
Iteration 96/100, Best score: 2.1939808222670443e-33
Iteration 97/100, Best score: 2.1939808222670443e-33
Iteration 98/100, Best score: 2.1939808222670443e-33
Iteration 99/100, Best score: 3.0621460434015e-35
Iteration 100/100, Best score: 3.0621460434015e-35
Optimal x found: 5.533666093469591e-18
Minimum value of f(x): 3.0621460434015e-35
PS C:\Users\hsinr\class\ECE497_Evolution_Robot>
```

Figure 7 : Optimization result of using PSO for x^2

The fitness function is basically each “particle’s” position within the line. It basically is just trying to get closer and closer to 0 from where they start. The closer they are to the global best position, the better they are and it’s not like a mutating function where it mutates others, but rather it affects others but updating other people’s velocity. Their velocity is how fast the particles change. Lets say 1 particle reached 0, then it is the obvious best solution, so the other particles will change quicker depending on how much of the best particle solution is to the global solution. Position of the particles are then determined based on Position(t + 1) = Position(t) + Velocity(t). It just goes on and on until each particle has reached the optimal solution or the iteration ends.

This seemed very hard at first but after researching into it and just doing it, it turned out to be pretty decent and simple. Utilizing a flock or group of particles that share information and update it is a pretty amazing concept. It is similar to how people dress the same way or study harder because someone else in the class is studying harder.