CSE 598 - Bio Inspired AI & optimisation

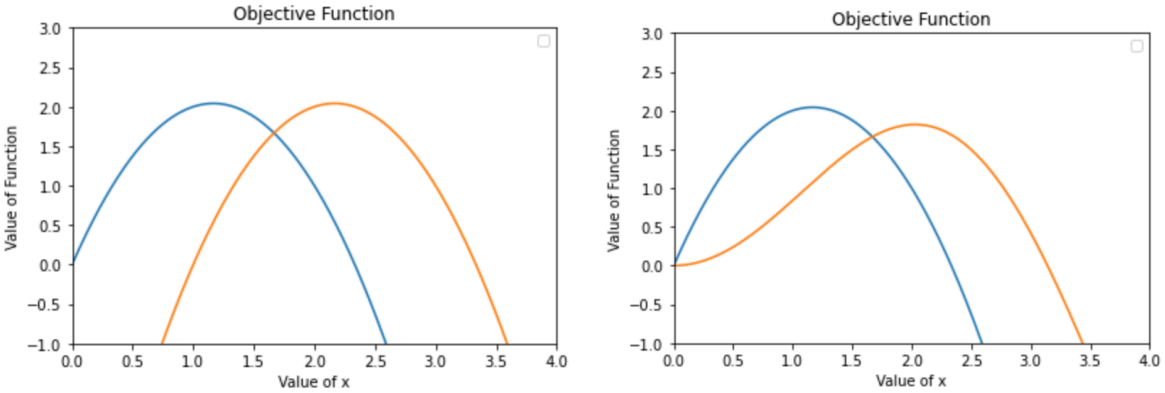
Ajay Kannan - 1219387832

Use this mini-project to demonstrate your understanding of some of the Unit 3 Learning Outcomes that relate to evolutionary algorithms for multi-objective optimisation. The scope of this mini-project should be similar to that of Mini-Project 1; in other words, try to formulate a small toy problem (or a set of small toy problems) that you can implement quickly in order to highlight some basic principle from this unit.

Problem - Developing a simple multi-objective evolutionary algorithm (e.g., using Pareto ranking, RWGA-like, or a VEGA-like approach) that is able to estimate the Pareto frontier for a number of non-linear optimisation objectives.

Solution -
For a multi-objective optimisation problem, there is often no single optimal solution, but rather a set of optimal solutions, called as Pareto-optimal solutions. Multi-objective optimisation is an area of multiple criteria decision making that is concerned with mathematical optimization problems involving more than one objective function to be optimised simultaneously. Multi-objective optimisation has been applied in many fields of science, including engineering, economics and logistics where optimal decisions need to be taken in the presence of trade-offs between two or more conflicting objectives.

To show the simple multi-objectives, two graphs are used with the following functions.
A) 3.5*x - (3 * x^2)/2 and 3.5*(x-1) - (3 * (x-1)^2)/2
B) 3.5*x - (3 * x^2)/2 and x * sin(x)



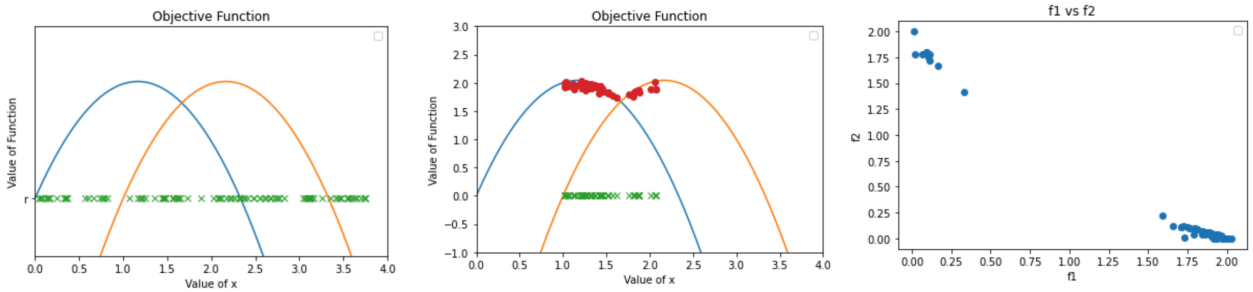
I developed a simple multi-objective evolutionary algorithm using Random weight genetic algorithm. Functions used while developing the mini project were as follows -

1. Population Size - This is the collection of candidate solutions, each represented by an array where each element represents a digit of the solution, where 100 is the population size.
2. Evaluation - Evaluation is done with the population which is in the range of [0,4]. Evaluation function is => e = f1 * alpha + f2 * (1 - alpha), where alpha is random value

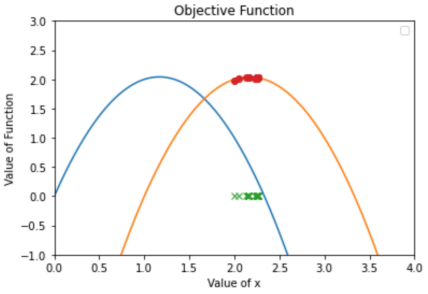
between [0,1] and is initialised every iteration. If e value goes out of bounds ([0,4]), it is not counted.

3. Selection - Selection is the process where number of population with high ‘e’ is taken. The function is implemented by taking high the population and doing a pairwise comparison with the other half. It generally induces low selective pressure is the parents set.
4. Elites Selection - Generally, a set of elites is stored. After selecting the population, if any of the population is having a higher fitness ‘e’ than any of the population in the elite set, then it is replaced. The final output will be the elite set.
5. Double point crossover - From parents, double-point crossover is performed to produce offsprings for next generation. Both children and parents are taken to the next level.
6. Mutation - Mutation performed only to the children set. Randomly genotype is modified to provide a bit of variance from the original set. Randomness is gauged upon a probability that is set to 0.001.
7. Then the new population is parents + children. It then is iterated over to the next generation. The number of generations is a predetermined value (300 say!).

Results - A)

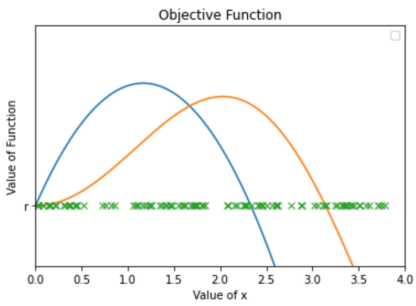


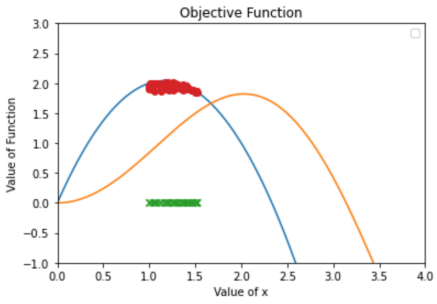
1. The initialisation of 100 members is shown in the left graph where the population is randomly spread between [0,4].
2. The middle graph shows the Pareto efficient set after 300 iterations (green points) and their respective objective points (red). The red points symbolise ‘e’ with the maximum value comparing f1 and f2.
3. The final graphs the Pareto frontier. Since RWGA is used, the of points is not apparent (RWGA is a low algorithm).

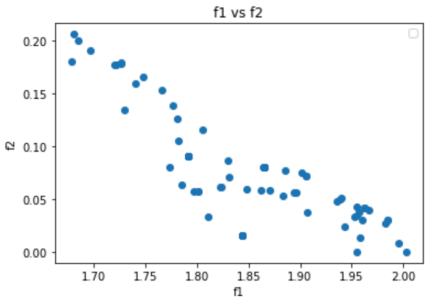


Most of the times when the algorithm is run, it less towards the left or centre. In some rare instances, it jump ship as shown here. Since the algorithm works on low selective pressure, this output is technically right.

B) Similarly, in this scenario, the two function are off different amplitudes. Therefore, applying RWGA will prefer the highest or fittest value over the other.







But, as ‘e’ rarely goes to zero in a one off the function, some residual value will be in the other function. In the third image, will can the residual values (f1 vs f2 is tending towards f2).

Conclusion - The algorithm is able to find a Pareto frontier for any two graphs or functions for a number of non-linear optimisation objectives through RWGA. Though it is tailor made for dual objective functions, we can transpose to any number of functions.

Pareto movement is the movement in which all objectives are improved simultaneously. By making small Pareto movements for a given gene, it can enter the Pareto efficient set. This is called convergence. By converging, Pareto frontier is formed. It is the Pareto optimal trade-off between f1 and f2 in this case. As you can see, the Pareto effect is clearly shown in the random weight genetic algorithm.

Appendix - From Mini Project 1, three major functions are taken.

Crossover, Mutation and Selection functions are taken without any change.

```
def double_point_crossover(parents, num_offspring):
    children = []
    for i in range(int(num_offspring/2)):
        index1 = np.random.randint(low=0, high=9)
        index2 = np.random.randint(low=0, high=9)
        parent1 = parents[index1]
        parent2 = parents[index2]
        offspring1 = np.append(parent1[0:2],parent2[2:4])
        offspring1 = np.append(offspring1 ,parent1[4:])
        offspring2 = np.append(parent2[0:2],parent1[2:4])
        offspring2 = np.append(offspring2, parent2[4:])
        children = np.append(children, offspring1)
        children = np.append(children, offspring2)
    children = np.split(children, num_offspring)
    return children
```

```
def mutation(children, prop):
    num_offspring = len(children)
    for i in range(num_offspring):
        probability = np.random.uniform(low=0, high=1)
        if probability < prop:
            random_value = np.random.randint(low=0, high=3)
            children[i][0] = random_value
        for j in range(1,5):
            probability = np.random.uniform(low=0, high=1)
            if probability < prop:
                random_value = np.random.randint(low=0, high=9)
                children[i][j] = random_value
    return children
```

```
def selection(population, fitness, num_parents):
   parents =[]
   for i in range(num_parents):
       if fitness[i] > fitness[num_parents*2-i-1]:
           parents.append(population[i])
       else:
           parents.append(population[num_parents*2-i-1])
   return parents
```

Initialise population -

```
def initialize_population(population_size):
    population = []
    for j in range(population_size):
        xs = np.random.randint(low=0, high=4, size =1)
        x = np.random.randint(low=0, high=9, size = 4)
        x = np.append(xs,x)
        population = np.append(population,x)
    population = np.split(population, population_size)
    return population
```

Evaluation function -

```
def evaluate_fitness(population):
    population_size = len(population)
    fitness = []
    alpha = np.random.rand(1)[0]
    for j in range(population_size):
        ind = population[j]
        x = float(str((int(ind[0])))+"."+str((int(ind[1])))+str((int(ind[2])))+str((int(ind[3])))+str((int(ind[4])))
        if float(x)<0 or float(x)>4:
            fit = -1000
        else:
            fit1 = f1(x)
            fit2 = f2(x)
            fit = alpha * fit1 + (1 - alpha) * fit2

        fitness = np.append(fitness,fit)
    return fitness, alpha
```

Forming elites -

```
def form_elites(elites, elites_fx, population, fitness, alpha_x, alpha):
    if elites:
        for i in range(len(fitness)):
            if fitness[i] > elites_fx[i]:
                elites[i] = population[i]
                elites_fx[i] = fitness[i]
                alpha_x[i] = alpha
    else:
        elites = population
        elites_fx = fitness
        alpha_x = np.full(len(population),alpha)
    return elites,elites_fx, alpha_x
```