CSE 598 - Bio Inspired AI & optimisation

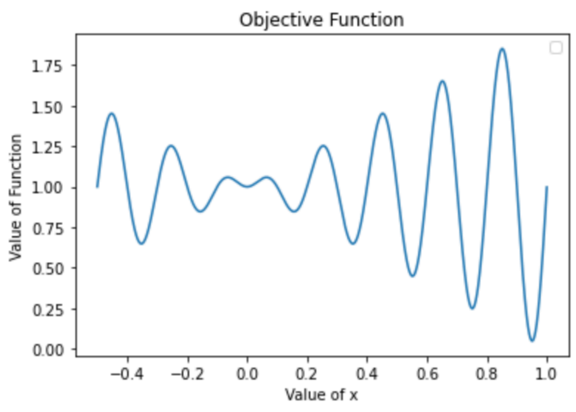
Ajay Kannan - 1219387832

Use this mini-project to demonstrate your understanding of some of the Unit 4 Learning Outcomes that relate to distributed genetic algorithms and/or multi-modal 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 that you can implement quickly in order to highlight some basic principle from this unit.

Problem - Implement a simple multi-modal optimisation algorithm with and without a particular niche-preserving technique to demonstrate the effect of that particular technique.

Solution -

To show the niche preserving technique, I implemented the RPS algorithm. Restricted tournament selection (RTS) is **a modification of the classical tournament selection for multimodal optimisation that exhibits niching capabilities**. RTS selects two elements from the population uniformly at random (u. a. r.) to undergo recombination and mutation to produce two new offspring. My implementation considers 50 such new pairs of population to undergo recombination and mutation.



The plot of the function as shown beside.

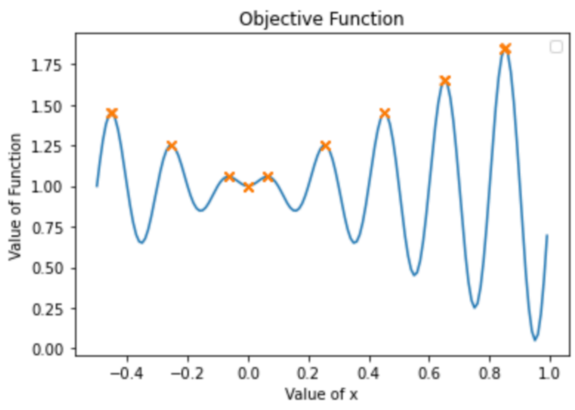
$f(x) = x \sin(10 \pi x) + 1$

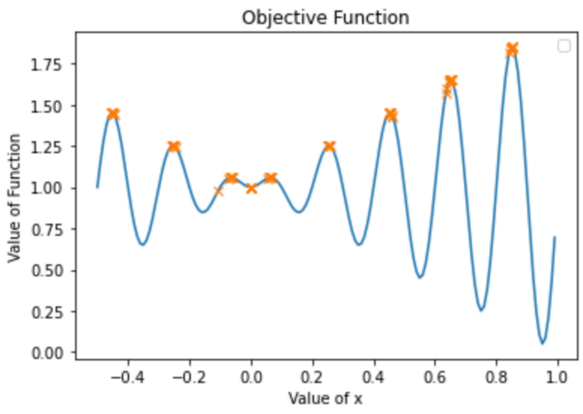
The range is $x \in [-0.5, 1]$

To show the multi-model evolutionary algorithm the following functions are used -

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. Evaluating the fitness of each candidate solution in the population by restricting to a range of -0.5 to 1. After the first iteration, if any number of population goes out of bounds, the fitness is set to -1000, so that we can effectively discard the points.

3. 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.
4. 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.
5. Selection - We have seen tournament selection before. Rank Selection is similar to roulette wheel selection except that **selection probability is proportional to relative fitness rather than absolute fitness**. It doesn't make any difference whether the fittest candidate is ten times fitter than the next fittest or 0.001% fitter. So, Restrictive tournament selection is done. There are two sets of population, old and new children. One at a time, take a child and the closest solution from the old generation, compare the two and which ever is higher, take it into the iteration. Like that we create a new population where it is unbounded by the specificity and it is completely random.
6. The new population is then iterated over to the next generation. The number of generations is a predetermined value (100 say!).





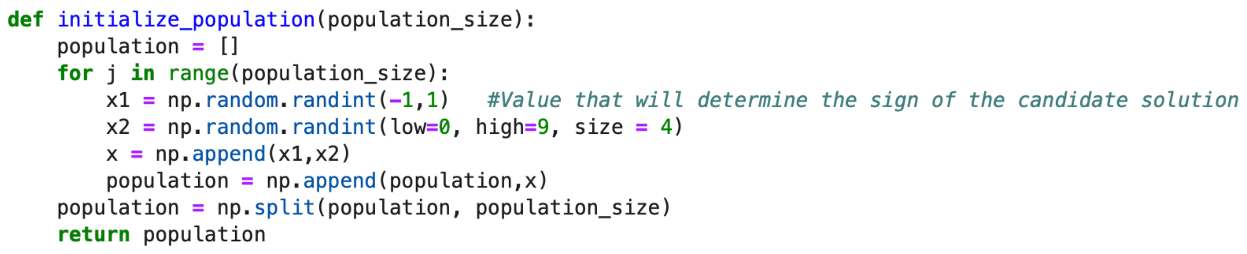
Results -

1) The points were placed in the peaks of function and the population his plate across the peaks evenly leaning towards the maxima. The algorithm is run for 1000 iterations and it is converging in the peaks.
2) For the second graph, it is run for 100 iterations. That is itself converging and a few points are missing out.

Conclusion - The multi-model genetic algorithm is clearly explained using the RTS (restrictive tournament selection) method where the peaks are getting converged upon the local maxima. This proves that the niche is getting captured and hence preserved.

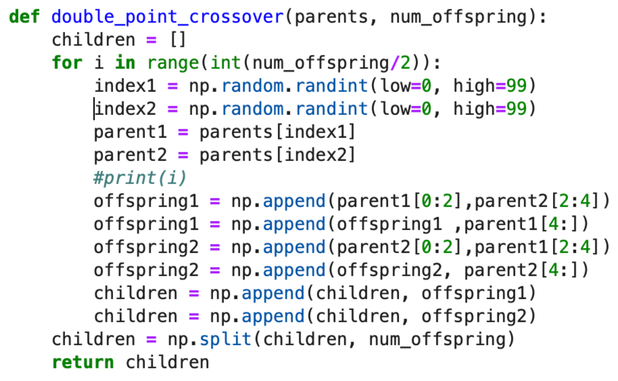
Code Snippets -

Initialisation -



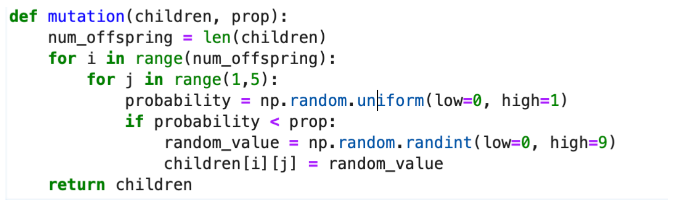
```python
def initialize_population(population_size):
    population = []
    for j in range(population_size):
        x1 = np.random.randint(-1,1)   #Value that will determine the sign of the candidate solution
        x2 = np.random.randint(low=0, high=9, size = 4)
        x = np.append(x1,x2)
        population = np.append(population,x)
    population = np.split(population, population_size)
    return population
```

Crossover Function -



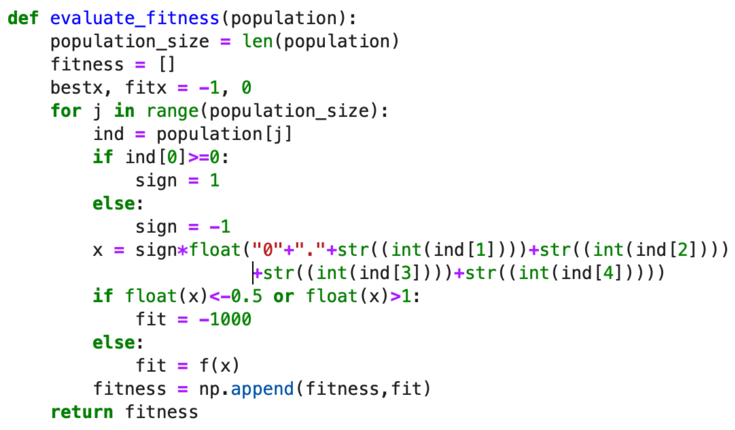
```python
def double_point_crossover(parents, num_offspring):
    children = []
    for i in range(int(num_offspring/2)):
        index1 = np.random.randint(low=0, high=99)
        index2 = np.random.randint(low=0, high=99)
        parent1 = parents[index1]
        parent2 = parents[index2]
        #print(i)
        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
```

Mutation -



```python
def mutation(children, prop):
    num_offspring = len(children)
    for i in range(num_offspring):
        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
```

Evaluation -



```python
def evaluate_fitness(population):
    population_size = len(population)
    fitness = []
    bestx, fitx = -1, 0
    for j in range(population_size):
        ind = population[j]
        if ind[0]>=0:
            sign = 1
        else:
            sign = -1
        x = sign*float("0"+"."+str((int(ind[1])))+str((int(ind[2])))
                       +str((int(ind[3])))+str((int(ind[4]))))
        if float(x)<-0.5 or float(x)>1:
            fit = -1000
        else:
            fit = f(x)
        fitness = np.append(fitness,fit)
    return fitness
```

RTS -

```python
def rtselection(population,children):
    populationf = []
    for ind in population:
        if ind[0]>=0:
            sign = 1
        else:
            sign = -1
        populationf.append(sign*float("0."+str((int(ind[1])))+str((int(ind[2])))
                                      +str((int(ind[3])))+str((int(ind[4])))))

    for c in range(len(children)):
        if children[c][0]>=0:
            sign = 1
        else:
            sign = -1
        child = sign*float("0."+str((int(children[c][1])))+str((int(children[c][2])))+
                           str((int(children[c][3])))+str((int(children[c][4]))))

        pc = []
        for i in populationf:
            pc.append(abs(i - child))

        pcmin, k = 10000, 0
        for i in range(len(pc)):
            if pcmin > pc[i]:
                k = i
                pcmin = pc[i]
        if f(populationf[k]) < f(child):
            populationf[k] = child
            population[k] = children[c]

    return population
```