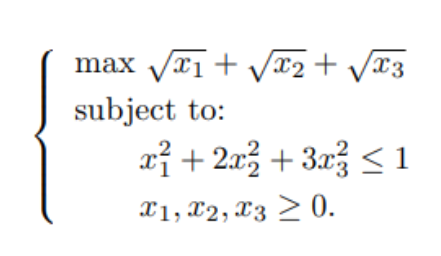
**Part A**

**Q-A.1**

**Introduction**Genetic Algorithm (GA) is a search-based optimization technique based on the principles of Genetics and Natural Selection. It is frequently used to find optimal or near-optimal solutions to difficult problems which otherwise would take a lifetime to solve. It is frequently used to solve optimization problems, in research, and in machine learning.

**Now let’s look at how can we solve one problem using genetic algorithm,**

Let us estimate the optimal values of a and b using GA which satisfies for any optimization problem starts with an objective function, so our question this is our objective function,



It is understood that the value of the function is 0. This function is our objective function and the aim is to estimate values of a and b such that the value of the objective function gets minimized to zero. The entire optimization process is explained below in four major steps and coded in R for one iteration (or generation).

**Initializing Population,**

**T**his step starts with guessing of initial sets of x1, x2 and x3values which may or may not include the optimal values. These sets of values are called as ‘**chromosomes** and the step are called **‘initialize population’**. Here population means sets of x1, x2 and x3, [x1, x2, x3,]. Random uniform function is used to generate initial values of x1, x2 and x3

**Selecting Chromosomes,**

In this step, the value of the objective function for each chromosome is computed. The value of the objective function is also called fitness value. This step is very important and is called ‘selection’ because fittest chromosomes are selected from the population for subsequent operations.

Based on the fitness values, more suitable chromosomes who have possibilities of producing low values of fitness function (because the value of our objective function needs to be 0) are selected and allowed to survive in succeeding generations. Some chromosomes are discarded to be unsuitable to produce low fitness value.

**Now that we know how to initialize the population and select chromosomes, Now let’s look at the algorithm with respect to this problem,**

**Algorithm**

1. Start
2. IMPORT numpy
3. IMPORT sys
4. SET equation\_INPUTs TO [1,1,1]
5. SET num\_weights TO 3
6. SET chromosomes TO 30
7. SET num\_generations TO 100
8. SET num\_parents\_mating TO 4
9. SET pop\_size TO (chromosomes,num\_weights)
10. SET new\_population TO numpy.random.uniform(low=0, size=pop\_size)
11. SET smallestFitness TO -sys.maxsize - 1
12. new\_population[0:5]
13. **DEFINE FUNCTION cal\_pop\_fitness(equation\_INPUTs, pop):**
    1. SET fitness TO []
    2. FOR i,chromosome IN enumerate(pop):

* (x1,x2,x3)=chromosome
* IF (x1\*x1+2\*x2\*x2+3\*x3\*x3) >1 or x1<0 or x2<0 or x3<0: #x1^2+2\*x2^2+3\*x3^2<=1 # x1,x2,x3>=0
* fitness.append(smallestFitness )
* continue
* chromosome=numpy.sqrt(chromosome)
* fitness.append(sum(chromosome))
* RETURN numpy.array(fitness)

1. **DEFINE FUNCTION select\_mating\_pool(pop, fitness, num\_parents):**
   1. SET parents TO numpy.empty((num\_parents, pop.shape[1]))
   2. FOR parent\_num IN range(num\_parents):

* SET max\_fitness\_idx TO numpy.where(fitness EQUALS numpy.max(fitness))
* SET max\_fitness\_idx TO max\_fitness\_idx[0][0]
* SET parents[parent\_num, :] TO pop[max\_fitness\_idx, :]
* SET fitness[max\_fitness\_idx] TO smallestFitness
* RETURN parents

1. **DEFINE FUNCTION crossover(parents, offspring\_size):**
   1. SET offspring TO numpy.empty(offspring\_size)
   2. SET crossover\_point TO numpy.uint8(offspring\_size[1]/2)
   3. FOR k IN range(offspring\_size[0]):

* SET parent1\_idx TO k%parents.shape[0]
* SET parent2\_idx TO (k+1)%parents.shape[0]
* SET offspring[k, 0:crossover\_point] TO parents[parent1\_idx, 0:crossover\_point]
* SET offspring[k, crossover\_point:] TO parents[parent2\_idx, crossover\_point:]
* RETURN offspring

1. **DEFINE FUNCTION mutation(offspring\_crossover, num\_mutations=1):**
   1. SET mutations\_counter TO numpy.uint8(offspring\_crossover.shape[1] / num\_mutations)
   2. FOR idx IN range(offspring\_crossover.shape[0]):

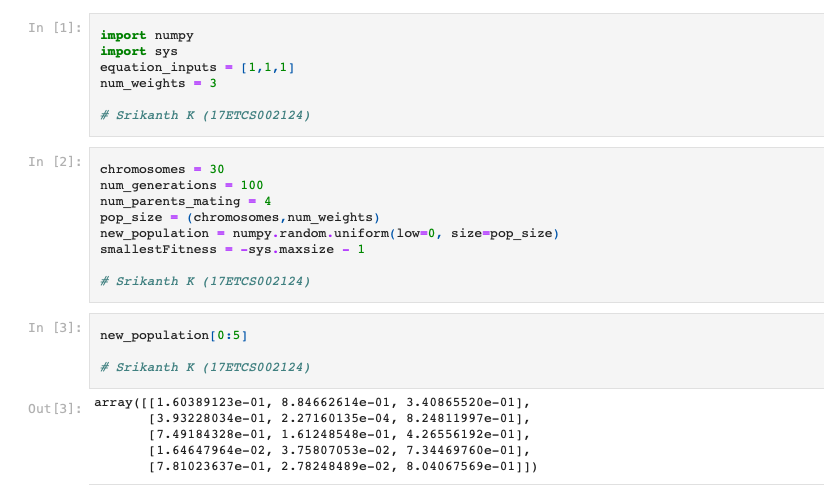
* SET gene\_idx TO mutations\_counter - 1
* FOR mutation\_num IN range(num\_mutations):
* SET random\_value TO numpy.random.uniform(-1.0, 1.0, 1)
* SET offspring\_crossover[idx, gene\_idx] TO offspring\_crossover[idx, gene\_idx] + random\_value
* SET gene\_idx TO gene\_idx + mutations\_counter
* RETURN offspring\_crossover

1. **FOR generation IN range(num\_generations):**

* OUTPUT(f"\nGeneration : {generation}")
* SET fitness TO cal\_pop\_fitness(equation\_INPUTs, new\_population)
* SET parents TO select\_mating\_pool(new\_population, fitness,num\_parents\_mating)
* OUTPUT("Parents selected FOR crossover")
* OUTPUT(parents)
* SET offspring\_crossover TO crossover(parents,offspring\_size=(pop\_size[0]-parents.shape[0], num\_weights))
* OUTPUT("Crossover results")
* OUTPUT(offspring\_crossover)
* SET offspring\_mutation TO mutation(offspring\_crossover)
* OUTPUT("Mutation results")
* OUTPUT(offspring\_mutation)
* SET new\_population[0:parents.shape[0], :] TO parents
* SET new\_population[parents.shape[0]:, :] TO offspring\_mutation
* OUTPUT(f"\nBest result FOR iteration {generation} : {numpy.max(numpy.sum(new\_population\*equation\_INPUTs, axis=1))} ")
* SET fitness TO cal\_pop\_fitness(equation\_INPUTs, new\_population)
* SET best\_match\_idx TO numpy.where(fitness EQUALS numpy.max(fitness))
* OUTPUT("Best solution : ", new\_population[best\_match\_idx, :])
* OUTPUT("Best solution fitness : ", fitness[best\_match\_idx])

**Q-A-2**

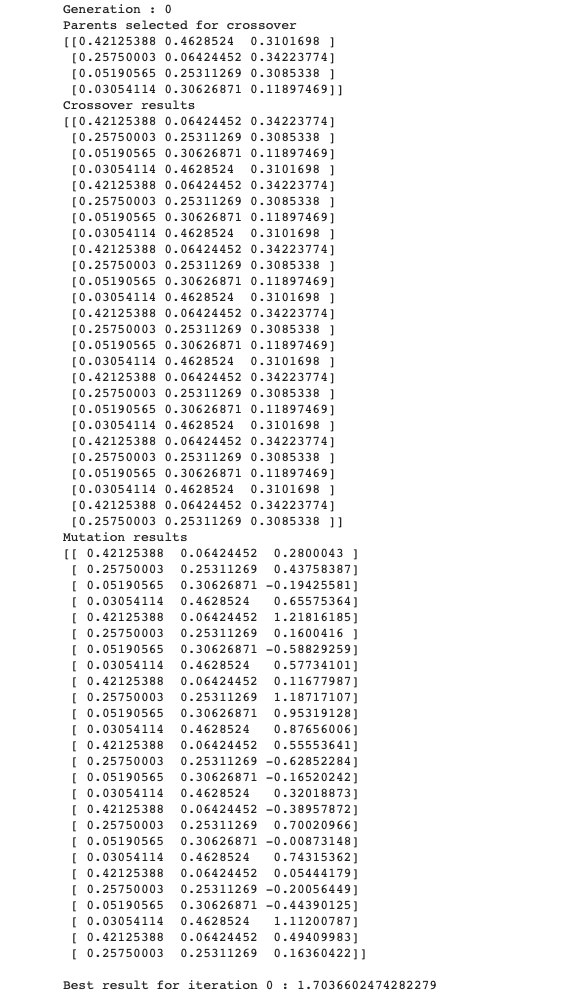
**Python Code**

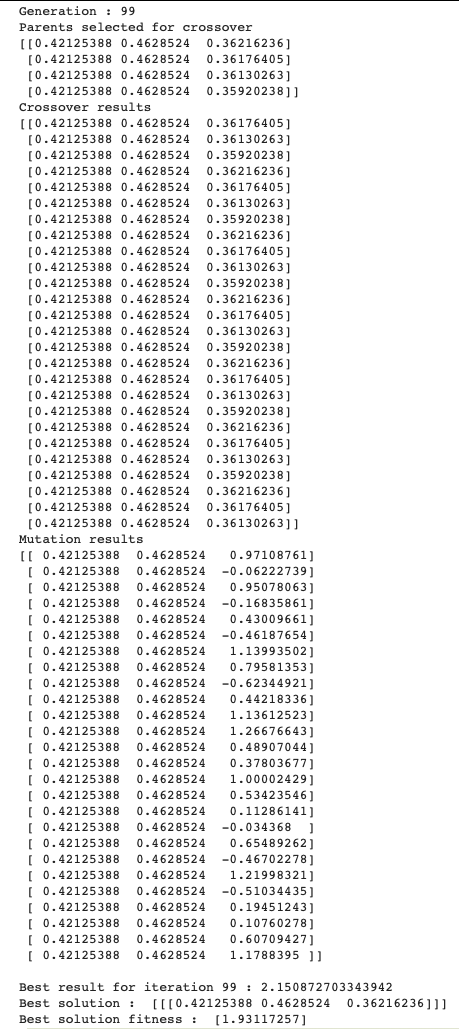
****

****

****

**Output of Python Code**

****

****

**Conclusion**

As we can see that after 100 iterations (Couldn’t include with the space limitation) the best solution that was generated with the chromosomes are

* **0.42125388**
* **0.4628524**
* **0.36216236**

With the best fitness solution being **1.93117257.**

**Part B**

**Q-B.1**

The algorithm I’m going to choose to solve this problem is: **Particle Swarm Optimization.**

**Introduction**

PSO is old and is the most used swarm intelligence algorithm. The general idea of PSO is inspired by a flying swarm of birds searching for food.). The task of object tracking is considered as a numerical optimization problem, where a PSO is used to track the local mode of the similarity measure and to seek a good local minimum, and then the conjugate gradient is utilized to find the local minimum accurately. But the ordinary PSO is not well suited for multiple object tracking. The algorithm introduces two new components to PSO: a self-adapting component, which is robust against drastic brightness changes of the image sequence, and a self-splitting component, which decides to track the scene as one connected object, or as more stand-alone objects.

**Justification**

PSO is best used to find the maximum or minimum of a function defined on a multidimensional vector space. Assume we have a function f(X) that produces a real value from a vector parameters radius and height and Radius can take on virtually any value in the space then we can apply PSO. The PSO algorithm will return the parameter X it found that produces the minimum f(X).

**Let’s start with the following function,**

****

It is not a **convex function** and therefore it is hard to find its minimum because a **local minimum** found is not necessarily the **global minimum**. So how can we find the minimum point in this function? For sure, we can resort to exhaustive search: If we check the value of function for every point on the plane, we can find the minimum point. Or we can just randomly find some sample points on the plane and see which one gives the lowest value on function if we think it is too expensive to search every point. However, we also note from the shape of function that if we have found a point with a smaller value of function it is easier to find an even smaller value around its proximity.

The main advantages of the PSO algorithm are summarized as: simple concept, easy implementation, robustness to control parameters, and computational efficiency when compared with mathematical algorithm and other heuristic optimization techniques. maximum iteration number, Iteration current iteration number.

**Q-B.2**

**How does this algorithm work? in \*mathematical terms,**

Assume we have P particles and we denote the position of particle ‘I’ at iteration t as Xi(t), which in the example of above, we have it as a coordinate,

**Xi(t)=(xi(t),yi(t))**

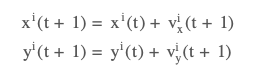
Besides the position, we also have a velocity for each particle, denoted as

**Vi(t)=(vxi(t),vyi(t)).**

At the next iteration, the position of each particle would be updated as

****

or, equivalently,

****

and at the same time, the velocities are also updated by the rule

where r1 and r2 are random numbers **between 0 and 1, constants w, c1, and c2** are parameters to the PSO algorithm, and **pbest** is the position that gives the best f(X) value ever explored by particle**i** and **gbest** is that explored by all the particles in the swarm.

Note that **pbest** and Xi(t) are two position vectors and the difference **pbest–Xi(t)** is a vector subtraction. Adding this subtraction to the original velocity Vi(t) is to bring the particle back to the position **pbest.** Similar are for the difference **gbest–Xi(t).**

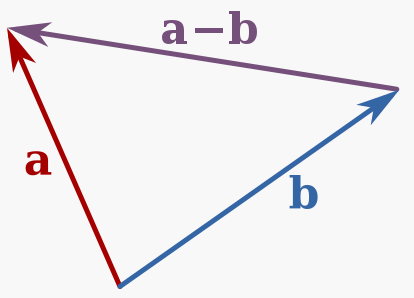


Figure 1 Vector Subtraction of PSO

We call the parameter w the inertia weight constant. It is between 0 and 1 and determines how much should the particle keep on with its previous velocity (i.e., speed and direction of the search). The parameters c1 and c2 are called the cognitive and the social coefficients respectively. They controls how much weight should be given between refining the search result of the particle itself and recognizing the search result of the swarm. We can consider these parameters controls the trade-off between **exploration** and **exploitation**.

The positions**pbest**and**gbest** are updated in each iteration to reflect the best position ever found thus far.

One interesting property of this algorithm that distinguish it from other optimization algorithms is that it does not depend on the gradient of the objective function. In gradient descent, for example, we look for the minimum of a function f(X) by moving X to the direction of −∇f(X) as it is where the function going down the fastest. For any particle at the position X at the moment, how it moves does not depend on which direction is the “downhill” but only on where are **pbest** and **gbest.** This makes PSO particularly suitable if differentiating f(X) is difficult.

Another property of PSO is that it can be parallelized easily. As we are manipulating multiple particles to find the optimal solution, each particles can be updated in parallel and we only need to collect the updated value of **gbest** once per iteration. This makes map-reduce architecture a perfect candidate to implement PSO.

This is how a particle swarm optimization does. Similar to the flock of birds looking for food, we start with a number of random points on the plane (call them **particles**) and let them look for the minimum point in random directions. At each step, every particle should search around the minimum point it ever found as well as around the minimum point found by the entire swarm of particles. After certain iterations, we consider the minimum point of the function as the minimum point ever explored by this swarm of particles.

**Q-B.3**

**Python Code for PSO**

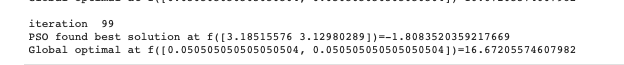
****

**Output for PSO**

At 1st iteration

****

At 100th iteration

****

**Q-B.4**

In the present work, we have used PSO Algorithm to find out best fit for our problem and optimized the solution with global best being **16.67205574607982** and PSO best being -**1.8083520359217669.** As discussed earlier, it is impracticable to say that the result obtained by an optimization method such as PSO is the global maximum or minimum, so some authors call the results as the most likely optimal global. Thus, some strategies can be employed in order to verify the validity of the optimal results obtained. One of the strategies is to compare with the results obtained by other optimization algorithms, as used in the present work. In the absence of optimal data available, due to either computational limitations or even lack of results of the subject, it is possible to use as strategy the comparison of information from real physical models, that is, that were not obtained through optimization algorithms, but instead good engineering practice and judgment gained through technical experience.

In addition, it was possible to apply the PSO algorithm to different engineering problems. The first involves the spacer grid of the fuel element and the second involves the optimization of the cost function of a cogeneration system. In both problems, satisfactory results were obtained demonstrating the efficiency of the PSO method.