In [1]:
```
import numpy
```

In [2]:
```
def cal_pop_fitness(equation_inputs, pop):
    # Calculating the fitness value of each solution in the current population.
    # The fitness function caulcuates the sum of products between each input and its corresponding weight.
    fitness = numpy.sum(pop*equation_inputs, axis=1)
    return fitness
```

In [3]:
```
def select_mating_pool(pop, fitness, num_parents):
    # Selecting the best individuals in the current generation as parents for producing the offspring of the next generation.
    parents = numpy.empty((num_parents, pop.shape[1]))
    for parent_num in range(num_parents):
        max_fitness_idx = numpy.where(fitness == numpy.max(fitness))
        max_fitness_idx = max_fitness_idx[0][0]
        parents[parent_num, :] = pop[max_fitness_idx, :]
        fitness[max_fitness_idx] = -99999999999
    return parents
```

In [4]:
```
def crossover(parents, offspring_size):
    offspring = numpy.empty(offspring_size)
    # The point at which crossover takes place between two parents. Usually it is at the center.
    crossover_point = numpy.uint8(offspring_size[1]/2)

    for k in range(offspring_size[0]):
        # Index of the first parent to mate.
        parent1_idx = k%parents.shape[0]
        # Index of the second parent to mate.
        parent2_idx = (k+1)%parents.shape[0]
        # The new offspring will have its first half of its genes taken from the first parent.
        offspring[k, 0:crossover_point] = parents[parent1_idx, 0:crossover_point]
        # The new offspring will have its second half of its genes taken from the second parent.
        offspring[k, crossover_point:] = parents[parent2_idx, crossover_point:]
    return offspring
```

In [5]:
```python
def mutation(offspring_crossover):
    # Mutation changes a single gene in each offspring randomly.
    for idx in range(offspring_crossover.shape[0]):
        # The random value to be added to the gene.
        random_value = numpy.random.uniform(-1.0, 1.0, 1)
        offspring_crossover[idx, 4] = offspring_crossover[idx, 4] + random_value
    return offspring_crossover
```

In [11]:
```python
# Inputs of the equation.
equation_inputs = [4,-2,3.5,5,-11,-4.7]

# Number of the weights we are looking to optimize.
num_weights = 6

"""
Genetic algorithm parameters:
    Mating pool size
    Population size
"""
sol_per_pop = 8
num_parents_mating = 4
```

In [12]:
```python
# Defining the population size.
# The population will have sol_per_pop chromosome where each chromosome has num_weights genes.
pop_size = (sol_per_pop,num_weights)
```

In [13]:
```python
#Creating the initial population.
new_population = numpy.random.uniform(low=-4.0, high=4.0, size=pop_size)
print(new_population)
```

```
[[-1.0468133   0.40023236 -2.40371356  1.74377671  0.88016318 -3.6241827 ]
 [ 2.11198553  2.47938845  0.68806781  1.17797051  0.4074285  -1.3461995 ]
 [-2.18064226 -3.98830864 -0.40041838 -0.76319955  2.15288246 -2.29012487]
 [ 3.8568017  -1.48299349  2.33560053  0.27147509 -0.83590856  0.4334214 ]
 [-1.35737105  0.90403876  2.1925174   1.8344854  -2.81130544 -2.38289254]
 [-3.58192008  2.91725726  0.54977895  3.03201971  1.03198915 -1.59130146]
 [ 0.6801786   2.93103623 -0.54010573  3.32311348 -2.54004858 -2.45065836]
 [ 1.10662555 -1.53517317 -2.02204031 -3.96555534  3.97112888  2.1092782 ]]
```

In [14]:
```
num_generations = 5
for generation in range(num_generations):
    print("Generation : ", generation)
    # Measing the fitness of each chromosome in the population.
    fitness = cal_pop_fitness(equation_inputs, new_population)

    # Selecting the best parents in the population for mating.
    parents = select_mating_pool(new_population, fitness,
                                  num_parents_mating)

    # Generating next generation using crossover.
    offspring_crossover = crossover(parents,
                                    offspring_size=(pop_size[0]-parents.shape[0], num_weights))

    # Adding some variations to the offsrping using mutation.
    offspring_mutation = mutation(offspring_crossover)

    # Creating the new population based on the parents and offspring.
    new_population[0:parents.shape[0], :] = parents
    new_population[parents.shape[0]:, :] = offspring_mutation

    # The best result in the current iteration.
    print("Best result : ", numpy.max(numpy.sum(new_population*equation_inputs, axis=1)))
```

```
Generation :  0
Best result :  56.50163256488645
Generation :  1
Best result :  61.86697069483192
Generation :  2
Best result :  68.5022132798091
Generation :  3
Best result :  68.5022132798091
Generation :  4
Best result :  69.90126297984114
```

In [15]:
```python
# Getting the best solution after iterating finishing all generations.
#At first, the fitness is calculated for each solution in the final generation.
fitness = cal_pop_fitness(equation_inputs, new_population)

# Then return the index of that solution corresponding to the best fitness.
best_match_idx = numpy.where(fitness == numpy.max(fitness))

print("Best solution : ", new_population[best_match_idx, :])
print("Best solution fitness : ", fitness[best_match_idx])
```

```
Best solution :  [[[ 2.11198553  2.47938845  0.68806781  1.8344854  -3.96653077
   -2.38289254]]]
Best solution fitness :  [69.90126298]
```

In [ ]:
```python

```