Functions from Week 4

Below are the functions that I used in Week 4 assignment.

Function 1: 1*(x-2) ^4 + 6*(y-9) ^2

class Function1 :

```
def get_quadratic_equation_value(self, x, y) :
    value = ((x - 2) ** 4) + (6 * ((y - 9) ** 2))
    return value

def get_quadratic_equation_derivative_value(self, x, y) :
    derivative_x_value = 4 * ((x - 2) ** 3)
    derivative_y_value = (12 * y) - 108
    return derivative_x_value, derivative_y_value
```

Function 2: Max(x-2,0) + 6*|y-9|

class Function2 :

```
def get_quadratic_equation_value(self, x, y) :
    value = (max((x - 2), 0)) + (6 * np.abs(y - 9))
    return value

def get_quadratic_equation_derivative_value(self, x, y) :
    derivative_x_value = np.heaviside((x - 2), 1)
    derivative_y_value = 6 * np.sign(y - 9)
    return derivative_x_value, derivative_y_value
```

Question (a)(i): Implement the Global Random Search algorithm

Global Random Search is an optimisation algorithm which is based on sampling the parameters randomly from a given search space and then using these parameter to obtain the optimal solution for the given function. The algorithm randomly selects parameter for the given function to generate a set of candidate solutions, and then evaluates each of these solutions to find the optimal solution. This is done in iterations and in each iteration, algorithm selects the optimal solution in hope of finding the global minima.

I have used below code to implement Global Random Search technique.

```
def execute_global_random_search(self, function_object, num_parameters = 2, lower_bound = [0,0], upper_bound = [5,5],
                                 num_samples = 100) :

    cost_iteration = []
    time_iteration = []
    optimised_cost = float('inf')
    start_time = time.time() * 1000

    for i in range(num_samples) :

        parameter_x, parameter_y = [random.uniform(lower_bound[j], upper_bound[j]) for j in range(num_parameters)]
        current_cost = function_object.get_quadratic_equation_value(parameter_x, parameter_y)

        if current_cost < optimised_cost :
            optimised_cost = current_cost

        cost_iteration.append(optimised_cost)
        time_iteration.append((time.time() * 1000) - start_time)

    return cost_iteration, time_iteration
```

Figure 1: Implementation of Global Random Search Algorithm

Explanation of code:

As mentioned above, the Global Random Search algorithm randomly select candidate parameters from search space in order to find optimal solution. In order to implement it, I have provided **the function that needs to be optimised as** input in parameter **function_object**. I have also provided **the number of parameters that this function takes** as input as **num_parameters**. The **lower bound and upper bound of the search space** are provided as input in **lower_bound** and **upper_bound** variables. Also, the **number of iterations** that we will run this optimisation for is provided as input in **num_samples** variable.

I have first initialised the **cost and time arrays to record the optimised cost and time taken in each iteration**. Then, I have calculated the **start time** of iterations using **time function of the time library** in Python.

In each iteration, I **first randomly choose parameters** from the search space **using the lower and upper bounds** provided to the function. These **parameters are then passed to function** to get the **current cost** for these parameters. Now, I evaluate that **if the current cost** calculated using the randomly chosen parameters is **less then optimal cost** in this iteration. **If the current cost is less than the optimised cost**, then I **update optimised cost to current cost.**

Finally, cost is stored in the cost array and time of the iteration is calculated by subtracting current time from start time and is stored in time array. At the end of iterations, I return the cost and time array for generating plots.

Question (a)(ii): Execute the Global Random Search and Gradient Descent algorithm for Functions provided in Week 4

I have executed the Global Random Search and Gradient Descent function for both the functions provided to me.

For **Global Random Search**, the initial conditions for both the functions are set same. **Number of parameters is set to 2**, since the cost function only takes 2 parameters. **Lower bound is set to 0,0 and Upper bound is set to 10,10** and **number of iterations is set to 200**. For **Gradient Descent**, the initial conditions are set same for both the functions. The **starting point is set as 5,5** and **learning rate α is set to 0.01**.

Following are the graphs generated:

For Function 1: 1*(x-2) ^4 + 6*(y-9) ^2

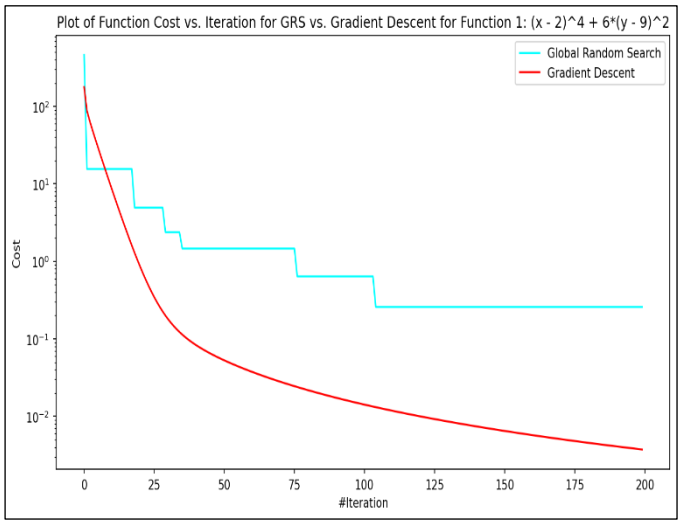

Figure 2a: Plot of Cost vs. Iterations for Function 1

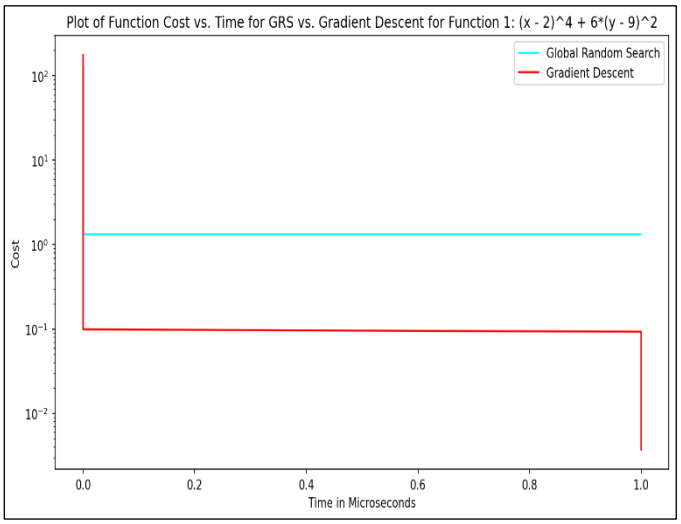

Figure 2b: Plot of Cost vs. Time for Function 1

From **Figure 2a**, I can clearly analyse that the **Gradient Descent algorithm performs better than the Global Random Search Algorithm**. The **Gradient Descent Algorithm** was able to **converge better** and **reach closer to global minima** than Global Random Search Algorithm. Furthermore, **Gradient Descent converged in lesser iterations than Global Random Search**. The primary reason for this behaviour is that **Function1 has a smooth convex surface**. Therefore, Gradient Descent performs better as it **constantly converges in the direction of the minima because of steepness in the convex surface,** while the Global Random Search **selects random variables from search space and only converges if it finds optimal results** than previous iterations. Another reason could be that the **search space for Global Random Search is not appropriate**, therefore it failed to converge fully.

From **Figure 2b**, I can analyse that **Gradient Descent converges to a lesser cost with time as compared to Global Random Search**. The primary reason for this behaviour is, as mentioned earlier, due to the presence of the convex surface of function 1.

For Function 2: Max(x-2,0) + 6*|y-9|

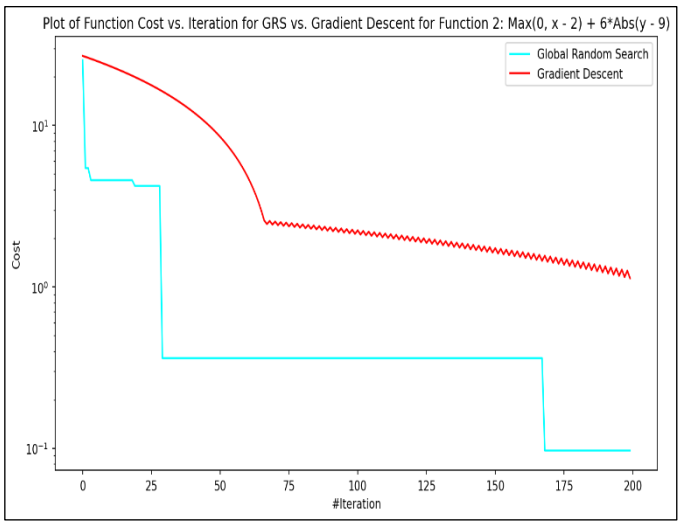

Figure 3b: Plot of Cost vs. Iterations for Function 2

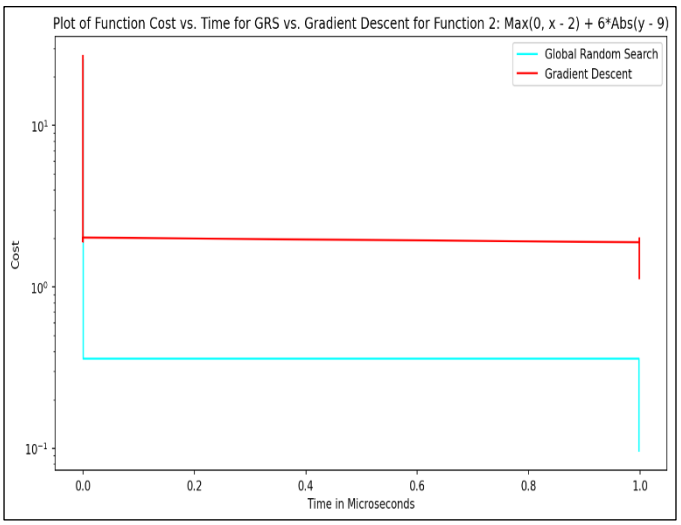

Figure 3b: Plot of Cost vs. Time for Function 2

From **Figure 3a**, I can clearly analyse that the **Global Random Search algorithm performs better than the Gradient Descent Algorithm**. The **Global Random Search Algorithm** was able to **converge better** and **reach closer to global minima** than Gradient Descent Algorithm. Furthermore, **Global Random Search converged in lesser iterations than Gradient Descent**. The primary reason for this behaviour is that **Function 2 has a non-convex surface and there is kink in the surface**. Therefore, **Gradient Descent** could not perform well on this, and **once it reached the minima It started to oscillate over the minima**. On the other hand, the **Global Random Search selects random variables from search space and only converges if it finds optimal results** compared to previous iterations. Therefore, **its performance is not affected by the non-smoothness of the cost surface and the presence of kink in the surface**.

From **Figure 3b,** I can analyse that **Gradient Descent is much slower to converge with respect to time than Global Random Search** for Function 2. The primary reason for this behaviour is, as mentioned earlier, the non-convex surface of Function 2

Question (b)(i): Implement the Global Population Search algorithm

Global Population Search is an optimisation algorithm which is based on sampling the parameters randomly from a given search space and then selecting best parameters to obtain the optimal solution for the given function. The algorithm randomly selects parameter for the given function to generate a set of candidate solutions, and then evaluates each of these solutions to find the optimal solution. Then, it chooses the a subset of the population that has the best parameters on the basis of the cost. Then, it uses these parameters to evaluate the solutions and find the optimal solution and again selects a population of best parameters. This is done in iterations and in each iteration, algorithm selects the optimal solution in hope of finding the global minima.

I have used below code to implement Global Population Search technique.

```
def execute_global_population_search(self, function_object, num_parameters = 2, lower_bound = [0,0], upper_bound = [5,5],
                                     num_samples = 100, M_best_params = 10, num_iterations = 200) :

    cost_iteration = []
    time_iteration = []
    optimised_cost = float('inf')
    start_time = time.time() * 1000
    optimised_params = None
    params_iteration = []

    n_sample_cost = []

    for i in range(num_samples) :

        parameter_x, parameter_y = [random.uniform(lower_bound[j], upper_bound[j]) for j in range(num_parameters)]
        current_cost = function_object.get_quadratic_equation_value(parameter_x, parameter_y)

        if current_cost < optimised_cost :
            optimised_cost = current_cost
            optimised_params = [parameter_x, parameter_y]

        n_sample_cost.append(optimised_cost)
        cost_iteration.append(optimised_cost)
        params_iteration.append([parameter_x, parameter_y])
        time_iteration.append((time.time() * 1000) - start_time)

    for _ in range(num_iterations) :
        print(f'num_iterations: {_}')
        best_cost = sorted(n_sample_cost)[:M_best_params]
        best_params = [params_iteration[n_sample_cost.index(cost)] for cost in n_sample_cost]

        best_cost_iteration = []
        best_param_iteration = []
        best time iteration = []

        for best_params in best_params :
            for _ in range(num_samples) :
                sample_parameter_x, sample_parameter_y = [random.uniform(best_params[j] - (upper_bound[j] - lower_bound[j])
                            * 0.1, best_params[j] + (upper_bound[j] - lower_bound[j]) * 0.1) for j in range(num_parameters)]
                sample_cost = function_object.get_quadratic_equation_value(sample_parameter_x, sample_parameter_y)

                if sample_cost < optimised_cost :
                    optimised_cost = sample_cost
                    optimised_params = [sample_parameter_x, sample_parameter_y]

                best_cost_iteration.append(optimised_cost)
                cost_iteration.append(optimised_cost)
                best_param_iteration.append(optimised_params)
                best_time_iteration.append(time.time() - start_time)

        params_iteration.extend(best_param_iteration)
        n_sample_cost = best_cost_iteration
        time_iteration.extend(best_time_iteration)

    return cost_iteration, time_iteration, params_iteration
```

Figure 4: Implementation of Global Population Search Algorithm

Explanation of code:
I have provided **the function that needs to be optimised as** input in parameter **function_object**. I have also provided **the number of parameters that this function takes** as input as **num_parameters**. The **lower bound and upper bound of the search space** are provided as input in **lower_bound** and **upper_bound** variables. I have also provided the **number of best parameters** to be selected in the parameter **M_best_params**. Also, the **number of iterations** that we will run this optimisation for is provided as input in **num_iterations** variable.

In first iteration, I **first randomly choose parameters** from the search space **using the lower and upper bounds** provided to the function. These **parameters are then passed to function** to get the **current cost** for these parameters. Now, I evaluate that **if the current cost** calculated using the randomly chosen parameters is **less then optimal cost** in this iteration. **If the current cost is less than the optimised cost**, then I **update optimised cost to current cost.**

Now, I **find the best 'n' parameters on the basis of cost** and again follow the same procedure to **find optimal cost** in each iteration. Then, I **update the current parameters to the best parameters** found in that iteration and **repeat the process for provided number of iterations**.

Finally, cost is stored in the cost array and time of the iteration is calculated by subtracting current time from start time and is stored in time array. At the end of iterations, I return the cost and time array for generating plots

Question (b)(ii): Execute the Global Random Search, Global Population Search and Gradient Descent algorithm for Functions provided in Week 4

I have reused the code for Global Random Search and Gradient Descent from previous questions and executed all three algorithms for both the functions provided to me.

For **Global Random Search**, the initial conditions for both the functions are set same. **Number of parameters is set to 2**, since the cost function only takes 2 parameters. **Lower bound is set to 0,0 and Upper bound is set to 10,10** and **number of iterations is set to 250**. For Global Population Search, all parameter remains same and **M_best_params is set to 10**. For **Gradient Descent**, the initial conditions are set same for both the functions. The **starting point is set as 5,5** and **learning rate α is set to 0.01**.
Following are the graphs generated:

For Function 1: 1*(x-2) ^4 + 6*(y-9) ^2

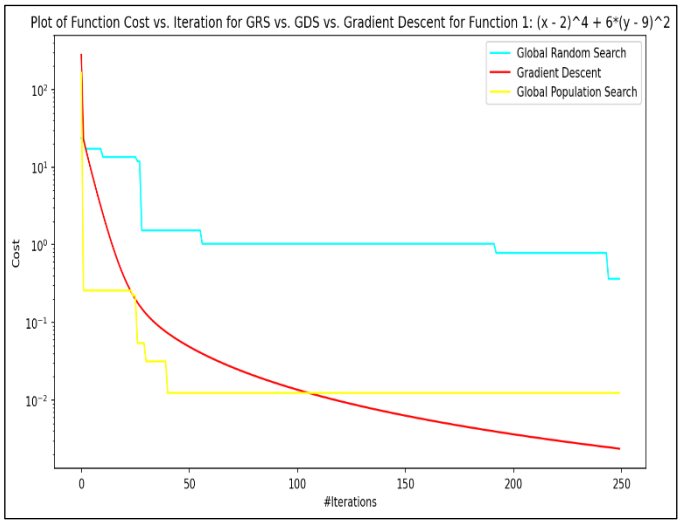

Figure 5a: Plot of Cost vs. Iterations for Function 1

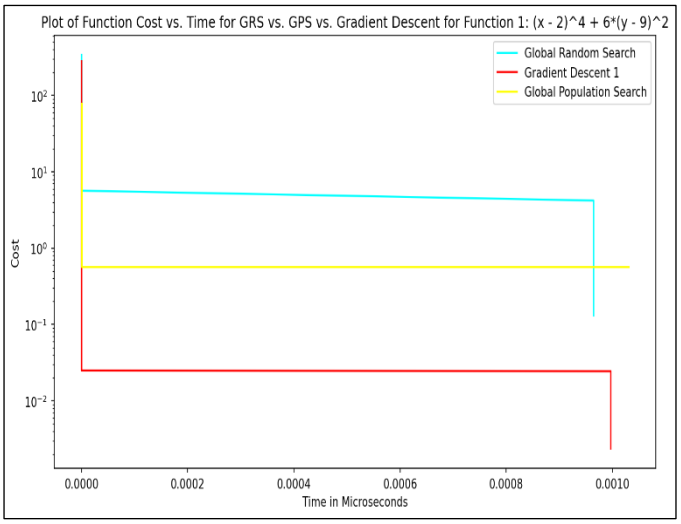

Figure 5b: Plot of Cost vs. Time for Function 1

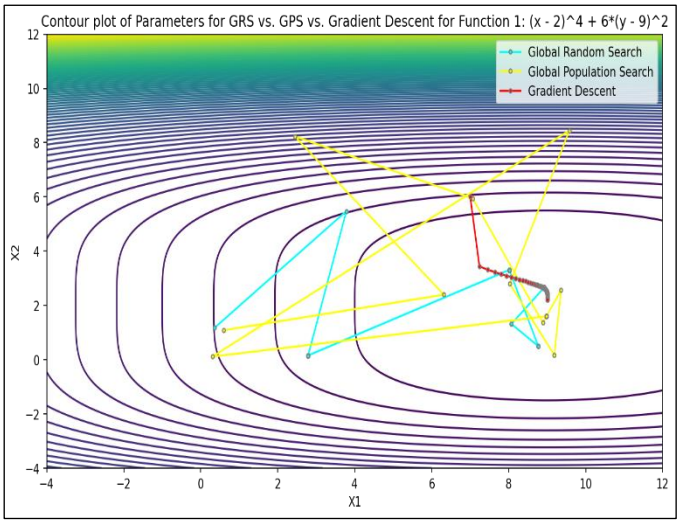

Figure 5c: Contour Plot for 3 algorithms for Function 1

From **Figure 5a**, I can clearly analyse that the **Gradient Descent algorithm performs better than the Global Random Search and Global Population Search Algorithm**. The **Gradient Descent Algorithm** was able to **converge better** and **reach closer to global minima** than the other 2 algorithms. The primary reason for this behaviour is that **Function1 has a smooth convex surface**. Therefore, Gradient Descent performs better as it **constantly converges in the direction of the minima because of steepness in the convex surface,** while the the other 2 algorithms depends on finding the optimal solution from random parameters to converge. Another reason could be that the **search space for Global Random Search is not appropriate**, therefore it failed to converge fully.

Furthermore, **Global Population Search performed better than Global Random Search and converged in fewer iterations toward minima**. The primary reason for this behaviour is that **Global Population search selects the best parameter** at the end of each iteration to find the optimal solution, **unlike Global Random Search**, which **uses random parameters** in each iteration.

From **Figure 5b**, I can analyse that **Gradient Descent converges to a lesser cost with time as compared to other 2 algorithms**. The primary reason for this behaviour is, as mentioned earlier, due to the presence of the convex surface of function 1. Furthermore, Global Population Search again performs better than Global Random Search.

From **Figure 5c,** I can analyse that **all three algorithms are converging towards the global minima**. Furthermore, **Gradient Descent** is **consistently moving towards the minima and is closest to minima** at the end of all iterations. Also, **Global Population Search is closer to minima than Global Random Search**. The explanation for this behaviour is already mentioned above.

For Function 2: Max(x-2,0) + 6*|y-9|

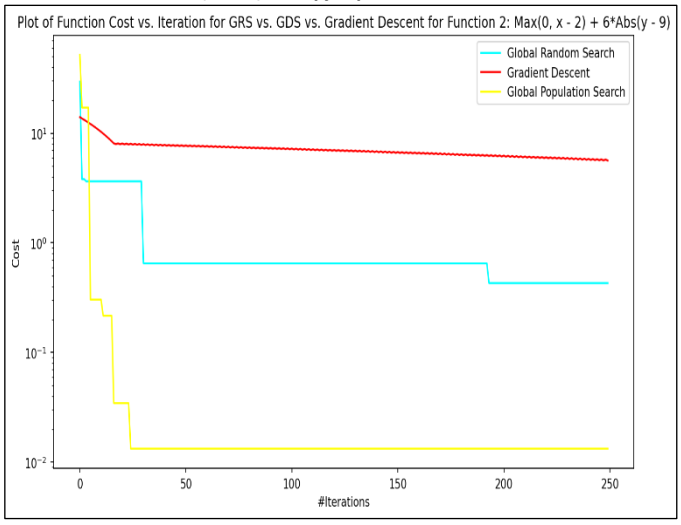

Figure 6a: Plot of Cost vs. Iterations for Function 2

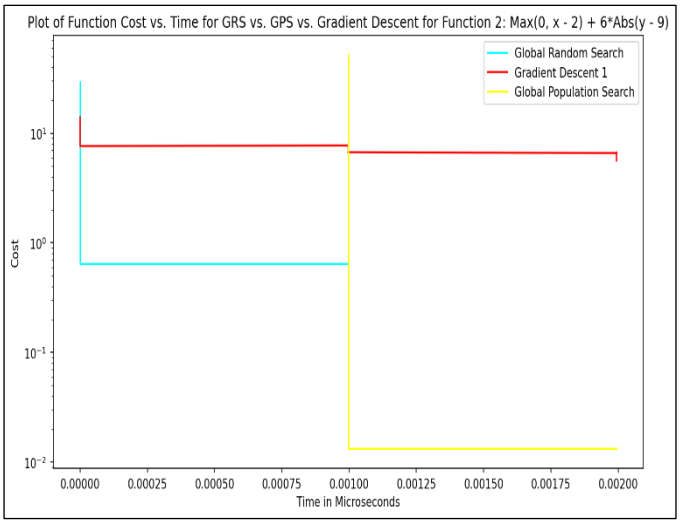

Figure 6b: Plot of Cost vs. Time for Function 2

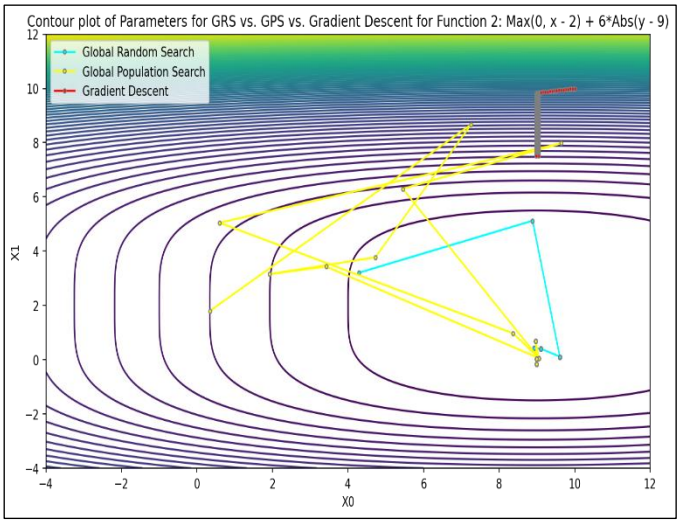

Figure 6c: Contour Plot for 3 algorithms for Function 2

From **Figure 6a**, I can clearly analyse that the **Global Population Search algorithm performs better than the Global Random Search and Gradient Descent Algorithm**. The **Global Random Population Algorithm** was able to **converge better** and **reach closer to global minima** than both of the algorithms. Furthermore, **Global Population Search converged in lesser itertations than Gradient Descent**. The primary reason for this behaviour is that **Function 2 has a non-convex surface and there is kink in the surface**. Therefore, the **Global Population Search selects best paramters from search space and only converges if it finds optimal results** compared to previous iterations. Therefore, **its performance is not affected by the non-smoothness of the cost surface and the presence of kink in the surface**.

Furthermore, Global Random Search performs better than Gradient Descent algorithm as seen previously in question 1. **Gradient Descent** could not perform well on this, and **once it gets closer to the minima It started to oscillate over the minima**.

From **Figure 6b**, I can analyse that **Global Population Search converges to a lesser cost with time as compared to other 2 algorithms**. The primary reason for this behaviour is, as mentioned earlier, the non-convex surface of Function 2.

From **Figure 6c,** I can analyse that **Global Population Search and Global Random Search are converging towards the global minima while Gradient Descent is very far from minima**. Furthermore, **Global Population Search** is **consistently moving towards the minima and is closest to minima** at the end of all iterations. The explanation for this behaviour is already mentioned above.

Question (c): Execute the Global Random Search, Global Population Search algorithm for CNN

I have implemented the Global Random Search Algorithm and Global Population Search algorithm for the CNN cost function provided to us. I have passed mini batch size in range [16, 64], Adam parameters α in range [0.0001, 0.01], β1 in range [0.25, 0.9] and β2 in range [0.9, 0.999], and number of epochs in range [10,20].

Following is the graph generated:

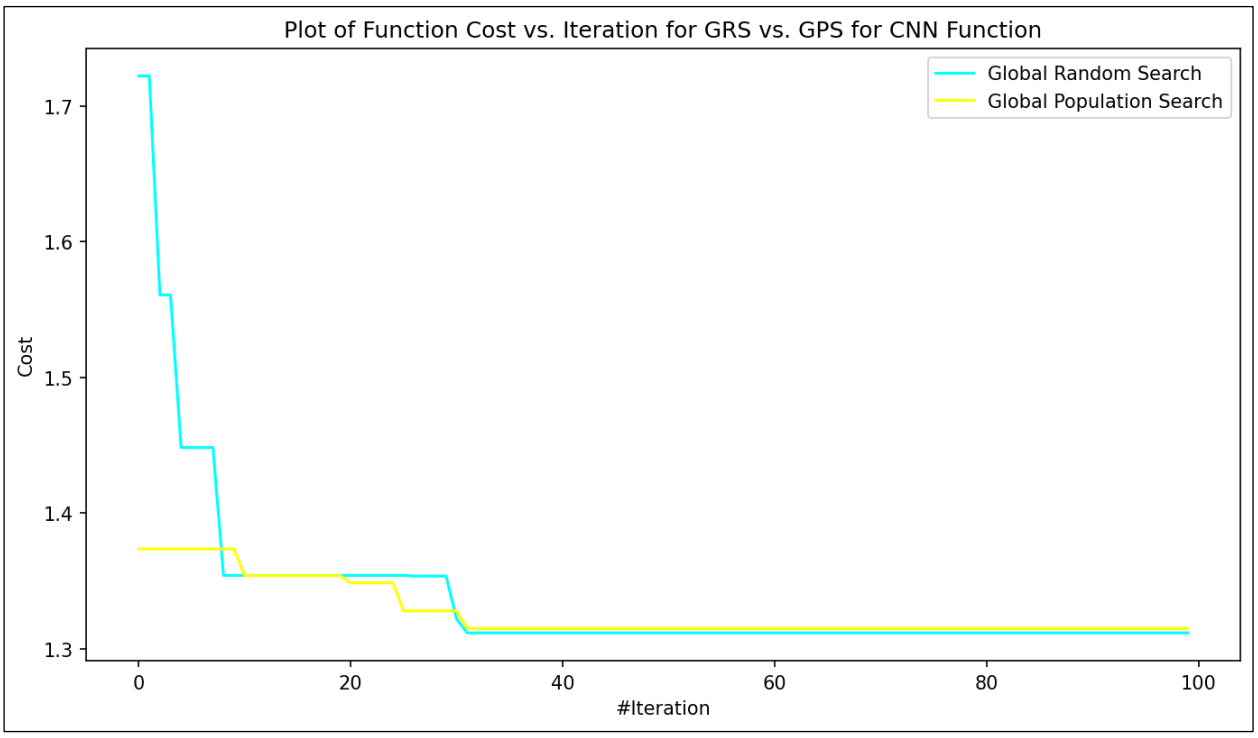

Figure 7: Plot of Cost vs. Iterations for CNN

From **Figure 7,** I can clearly analyse that the Global Population search performs slightly better than the Global Random Search. It is relatively more stable than Global Random Search as it has very a smaller number of fluctuations. Furthermore, it starts at a lower cost compared to Global Random Search and converges faster. The primary reason for this behaviour is that Global Population Search used the best parameters in each iteration to find optimal costs.

Appendix: Code for Question a – All Parts

```
import numpy as np
import time
import random
import matplotlib.pyplot as plot
from timeit import timeit

class Function1 :

    def get_quadratic_equation_value(self, x, y) :
        value = ((x - 2) ** 4) + (6 * ((y - 9) ** 2))
        return value

    def get_quadratic_equation_derivative_value(self, x, y) :
        derivative_x_value = 4 * ((x - 2) ** 3)
        derivative_y_value = (12 * y) - 108
        return derivative_x_value, derivative_y_value

class Function2 :

    def get_quadratic_equation_value(self, x, y) :
        value = (max((x - 2), 0)) + (6 * np.abs(y - 9))
        return value

    def get_quadratic_equation_derivative_value(self, x, y) :
        derivative_x_value = np.heaviside((x - 2), 1)
        derivative_y_value = 6 * np.sign(y - 9)
        return derivative_x_value, derivative_y_value

class Global_Random_Search :

    def execute_global_random_search(self, function_object, num_parameters = 2, lower_bound = [0,0], upper_bound = [5,5],
                                     num_samples = 100) :

        cost_iteration = []
        time_iteration = []
        optimised_cost = float('inf')
        start_time = time.time() * 1000

        for i in range(num_samples) :

            parameter_x, parameter_y = [random.uniform(lower_bound[j], upper_bound[j]) for j in range(num_parameters)]
            current_cost = function_object.get_quadratic_equation_value(parameter_x, parameter_y)

            if current_cost < optimised_cost :
                optimised_cost = current_cost

            cost_iteration.append(optimised_cost)
            time_iteration.append((time.time() * 1000) - start_time)

        return cost_iteration, time_iteration

    def execute_gradient_descent(self, function_object, starting_point_x = 1, starting_point_y = 1, alpha = 1, num_iterations = 50) :

        x_point = starting_point_x
        y_point = starting_point_y
        function_values = []
        time_iteration = []
        start_time = time.time() * 1000

        for _ in range(num_iterations) :

            function_values.append(function_object.get_quadratic_equation_value(x_point, y_point))
```

```
        derivative_x_value, derivative_y_value = function_object.get_quadratic_equation_derivative_value(x_point, y_point)
        x_point = x_point - (alpha * derivative_x_value)
        y_point = y_point - (alpha * derivative_y_value)

        time_iteration.append((time.time() * 1000) - start_time)


    return function_values, time_iteration

function1 = Function1()
grs = Global_Random_Search()

num_parameters = 2
lower_bound = [0, 0]
upper_bound = [10, 10]
num_iterations = 200

grs_cost_iteration_func1, grs_time_iteration_func1 = grs.execute_global_random_search(function_object = function1, num_parameters =
num_parameters, lower_bound = lower_bound, upper_bound = upper_bound, num_samples = num_iterations)

function1 = Function1()
grs = Global_Random_Search()

starting_point_x = 5
starting_point_y = 5
alpha = 0.01
num_iterations = 200


gd_cost_iteration_func1, gd_time_iteration_func1 = grs.execute_gradient_descent(function1, starting_point_x, starting_point_y, alpha,
num_iterations)

plot.figure(figsize=(11, 6), dpi=150)
plot.semilogy(range(num_iterations), grs_cost_iteration_func1, color = 'cyan', label = 'Global Random Search')
plot.semilogy(range(num_iterations), gd_cost_iteration_func1, color = 'red', label = 'Gradient Descent')
plot.xlabel('#Iteration')
plot.ylabel('Cost')
plot.title('Plot of Function Cost vs. Iteration for GRS vs. Gradient Descent for Function 1: (x - 2)^4 + 6*(y - 9)^2')
plot.yscale("log")
plot.legend()
plot.show()

plot.figure(figsize=(11, 6), dpi=150)
plot.semilogy(grs_time_iteration_func1, grs_cost_iteration_func1, color = 'cyan', label = 'Global Random Search')
plot.semilogy(gd_time_iteration_func1, gd_cost_iteration_func1, color = 'red', label = 'Gradient Descent')
plot.xlabel('Time in Microseconds')
plot.ylabel('Cost')
plot.title('Plot of Function Cost vs. Time for GRS vs. Gradient Descent for Function 1: (x - 2)^4 + 6*(y - 9)^2')
plot.yscale("log")
plot.legend()
plot.show()

function2 = Function2()
grs = Global_Random_Search()

num_parameters = 2
lower_bound = [0, 0]
upper_bound = [10, 10]
num_iterations = 200

grs_cost_iteration_func2, grs_time_iteration_func2 = grs.execute_global_random_search(function_object = function2, num_parameters =
num_parameters, lower_bound = lower_bound, upper_bound = upper_bound, num_samples = num_iterations)
```

```
function2 = Function2()
grs = Global_Random_Search()

starting_point_x = 5
starting_point_y = 5
alpha = 0.01
num_iterations = 200

gd_cost_iteration_func2, gd_time_iteration_func2 = grs.execute_gradient_descent(function2, starting_point_x, starting_point_y, alpha,
num_iterations)

plot.figure(figsize=(11, 6), dpi=150)
plot.semilogy(range(num_iterations), grs_cost_iteration_func2, color = 'cyan', label = 'Global Random Search')
plot.semilogy(range(num_iterations), gd_cost_iteration_func2, color = 'red', label = 'Gradient Descent')
plot.xlabel('#Iteration')
plot.ylabel('Cost')
plot.title('Plot of Function Cost vs. Iteration for GRS vs. Gradient Descent for Function 2: Max(0, x - 2) + 6*Abs(y - 9)')
plot.yscale("log")
plot.legend()
plot.show()

plot.figure(figsize=(11, 6), dpi=150)
plot.semilogy(grs_time_iteration_func2, grs_cost_iteration_func2, color = 'cyan', label = 'Global Random Search')
plot.semilogy(gd_time_iteration_func2, gd_cost_iteration_func2, color = 'red', label = 'Gradient Descent')
plot.xlabel('Time in Microseconds')
plot.ylabel('Cost')
plot.title('Plot of Function Cost vs. Time for GRS vs. Gradient Descent for Function 2: Max(0, x - 2) + 6*Abs(y - 9)')
plot.yscale("log")
plot.legend()
plot.show()
```

Appendix: Code for Question b – All Parts

```
import numpy as np
import time
import random
import matplotlib.pyplot as plot

class Function1 :

   def get_quadratic_equation_value(self, x, y) :
      value = ((x - 2) ** 4) + (6 * ((y - 9) ** 2))
      return value

   def get_quadratic_equation_derivative_value(self, x, y) :
      derivative_x_value = 4 * ((x - 2) ** 3)
      derivative_y_value = (12 * y) - 108
      return derivative_x_value, derivative_y_value

class Function2 :

   def get_quadratic_equation_value(self, x, y) :
      value = (max((x - 2), 0)) + (6 * np.abs(y - 9))
      return value

   def get_quadratic_equation_derivative_value(self, x, y) :
      derivative_x_value = np.heaviside((x - 2), 1)
      derivative_y_value = 6 * np.sign(y - 9)
      return derivative_x_value, derivative_y_value

class Global_Random_Population_Search :

   def execute_global_random_search(self, function_object, num_parameters = 2, lower_bound = [0,0], upper_bound = [5,5], num_samples
= 100) :

      cost_iteration = []
      time_iteration = []
      optimised_cost = float('inf')
      optimised_params = None
      params_iteration = []

      start_time = time.time() * 1000

      for i in range(num_samples) :

         parameter_x, parameter_y = [random.uniform(lower_bound[j], upper_bound[j]) for j in range(num_parameters)]
         current_cost = function_object.get_quadratic_equation_value(parameter_x, parameter_y)

         if current_cost < optimised_cost :
            optimised_cost = current_cost
            optimised_params = [parameter_x, parameter_y]

         params_iteration.append(optimised_params)
         cost_iteration.append(optimised_cost)
         time_iteration.append((time.time() * 1000) - start_time)

      return cost_iteration, time_iteration, params_iteration



   def execute_global_population_search(self, function_object, num_parameters = 2, lower_bound = [0,0], upper_bound = [5,5],
                              num_samples = 100, M_best_params = 10, num_iterations = 200) :

      cost_iteration = []
      time_iteration = []
      optimised_cost = float('inf')
      start_time = time.time() * 1000
      optimised_params = None
      params_iteration = []
```

```
    n_sample_cost = []

    for i in range(num_samples) :

        parameter_x, parameter_y = [random.uniform(lower_bound[j], upper_bound[j]) for j in range(num_parameters)]
        current_cost = function_object.get_quadratic_equation_value(parameter_x, parameter_y)

        if current_cost < optimised_cost :
            optimised_cost = current_cost
            optimised_params = [parameter_x, parameter_y]

        n_sample_cost.append(optimised_cost)
        cost_iteration.append(optimised_cost)
        params_iteration.append([parameter_x, parameter_y])
        time_iteration.append((time.time() * 1000) - start_time)

    for _ in range(num_iterations) :
        print(f'num_iterations: {_}')
        best_cost = sorted(n_sample_cost)[:M_best_params]
        best_params = [params_iteration[n_sample_cost.index(cost)] for cost in n_sample_cost]

        best_cost_iteration = []
        best_param_iteration = []
        best_time_iteration = []

        for best_params in best_params :
            for _ in range(num_samples) :
                sample_parameter_x, sample_parameter_y = [random.uniform(best_params[j] - (upper_bound[j] - lower_bound[j])
                        * 0.1, best_params[j] + (upper_bound[j] - lower_bound[j]) * 0.1) for j in range(num_parameters)]
                sample_cost = function_object.get_quadratic_equation_value(sample_parameter_x, sample_parameter_y)

                if sample_cost < optimised_cost :
                    optimised_cost = sample_cost
                    optimised_params = [sample_parameter_x, sample_parameter_y]

                best_cost_iteration.append(optimised_cost)
                cost_iteration.append(optimised_cost)
                best_param_iteration.append(optimised_params)
                best_time_iteration.append(time.time() - start_time)

        params_iteration.extend(best_param_iteration)
        n_sample_cost = best_cost_iteration
        time_iteration.extend(best_time_iteration)

    return cost_iteration, time_iteration, params_iteration



def execute_gradient_descent(self, function_object, starting_point_x = 1, starting_point_y = 1, alpha = 1, num_iterations = 50) :

    x_point = starting_point_x
    y_point = starting_point_y
    function_values = []
    time_iteration = []
    start_time = time.time() * 1000

    for _ in range(num_iterations) :

        function_values.append(function_object.get_quadratic_equation_value(x_point, y_point))

        derivative_x_value, derivative_y_value = function_object.get_quadratic_equation_derivative_value(x_point, y_point)
        x_point = x_point - (alpha * derivative_x_value)
        y_point = y_point - (alpha * derivative_y_value)

        time_iteration.append((time.time() * 1000) - start_time)


    return function_values, time_iteration
```

```
function1 = Function1()
grs = Global_Random_Population_Search()

num_parameters = 2
lower_bound = [0, 0]
upper_bound = [10, 10]
num_iterations = 250

grs_cost_iteration_func1, grs_time_iteration_func1, grs_param_iteration_func1 = grs.execute_global_random_search(function_object = function1, num_parameters = num_parameters, lower_bound = lower_bound, upper_bound = upper_bound, num_samples = num_iterations)

function1 = Function1()
grs = Global_Random_Population_Search()

num_parameters = 2
lower_bound = [0, 0]
upper_bound = [10, 10]
num_samples = 200
M_best_params = 10
num_iterations = 250

gps_cost_iteration_func1, gps_time_iteration_func1, gps_param_iteration_func1 = grs.execute_global_population_search(function_object = function1, num_parameters = num_parameters, lower_bound = lower_bound, upper_bound = upper_bound, num_samples = num_samples, M_best_params = M_best_params, num_iterations = num_iterations)

function1 = Function1()
grs = Global_Random_Population_Search()plot.figure(figsize=(11, 6), dpi=150)
plot.semilogy(range(num_iterations), grs_cost_iteration_func1, color = 'cyan', label = 'Global Random Search')
plot.semilogy(range(num_iterations), gps_cost_iteration_func1, color = 'yellow', label = 'Global Population Search')
plot.semilogy(range(num_iterations), gd_cost_iteration_func1, color = 'red', label = 'Gradient Descent')
plot.xlabel('#Iteration')
plot.ylabel('Cost')
plot.title('Plot of Function Cost vs. Iteration for GRS vs. Gradient Descent for Function 1: (x - 2)^4 + 6*(y - 9)^2')
plot.yscale("log")
plot.legend()
plot.show()

starting_point_x = 5
starting_point_y = 5
alpha = 0.01
num_iterations = 250


gd_cost_iteration_func1, gd_time_iteration_func1 = grs.execute_gradient_descent(function1, starting_point_x, starting_point_y, alpha, num_iterations)

plot.figure(figsize=(11, 6), dpi=150)
plot.semilogy(range(num_iterations), grs_cost_iteration_func1, color = 'cyan', label = 'Global Random Search')
plot.semilogy(range(num_iterations), gps_cost_iteration_func1, color = 'yellow', label = 'Global Population Search')
plot.semilogy(range(num_iterations), gd_cost_iteration_func1, color = 'red', label = 'Gradient Descent')
plot.xlabel('#Iteration')
plot.ylabel('Cost')
plot.title('Plot of Function Cost vs. Iteration for GRS vs. Gradient Descent for Function 1: (x - 2)^4 + 6*(y - 9)^2')
plot.yscale("log")
plot.legend()
plot.show()

plot.figure(figsize=(11, 6), dpi=150)
plot.semilogy(grs_time_iteration_func1, grs_cost_iteration_func1, color = 'cyan', label = 'Global Random Search')
plot.semilogy(gps_time_iteration_func1, gps_cost_iteration_func1, color = 'yellow', label = 'Global Population Search')
plot.semilogy(gd_time_iteration_func1, gd_cost_iteration_func1, color = 'red', label = 'Gradient Descent')
plot.xlabel('Time in Microseconds')
plot.ylabel('Cost')
plot.title('Plot of Function Cost vs. Time for GRS vs. Gradient Descent for Function 1: (x - 2)^4 + 6*(y - 9)^2')
plot.yscale("log")
plot.legend()
plot.show()
```

```
function2 = Function2()
grs = Global_Random_Population_Search()

num_parameters = 2
lower_bound = [0, 0]
upper_bound = [10, 10]
num_iterations = 250

grs_cost_iteration_func2, grs_time_iteration_func2 = grs.execute_global_random_search(function_object = function2, num_parameters = num_parameters, lower_bound = lower_bound, upper_bound = upper_bound, num_samples = num_iterations)

function2 = Function2()
grs = Global_Random_Population_Search()
num_parameters = 2
lower_bound = [0, 0]
upper_bound = [10, 10]
num_samples = 200
M_best_params = 10
num_iterations = 250
gps_cost_iteration_func2, gps_time_iteration_func2, gps_param_iteration_func2 = grs.execute_global_population_search(function_object = function2, num_parameters = num_parameters, lower_bound = lower_bound, upper_bound = upper_bound, num_samples = num_samples, M_best_params = M_best_params, num_iterations = num_iterations)

function2 = Function2()
grs = Global_Random_Population_Search()

starting_point_x = 5
starting_point_y = 5
alpha = 0.01
num_iterations = 250


gd_cost_iteration_func2, gd_time_iteration_func2 = grs.execute_gradient_descent(function2, starting_point_x, starting_point_y, alpha, num_iterations)

plot.figure(figsize=(11, 6), dpi=150)
plot.semilogy(range(num_iterations), grs_cost_iteration_func2, color = 'cyan', label = 'Global Random Search')
plot.semilogy(range(num_iterations), gd_cost_iteration_func2, color = 'red', label = 'Gradient Descent')
plot.xlabel('#Iteration')
plot.ylabel('Cost')
plot.title('Plot of Function Cost vs. Iteration for GRS vs. Gradient Descent for Function 2: Max(0, x - 2) + 6*Abs(y - 9)')
plot.yscale("log")
plot.legend()
plot.show()

plot.figure(figsize=(11, 6), dpi=150)
plot.semilogy(grs_time_iteration_func2, grs_cost_iteration_func2, color = 'cyan', label = 'Global Random Search')
plot.semilogy(gd_time_iteration_func2, gd_cost_iteration_func2, color = 'red', label = 'Gradient Descent')
plot.xlabel('Time in Microseconds')
plot.ylabel('Cost')
plot.title('Plot of Function Cost vs. Time for GRS vs. Gradient Descent for Function 2: Max(0, x - 2) + 6*Abs(y - 9)')
plot.yscale("log")
plot.legend()
plot.show()
```

Appendix: Code for Question c – All Parts

```
import numpy as np
import time
import random
import matplotlib.pyplot as plot
from tensorflow import keras
from tensorflow.keras import layers, regularizers
from keras.layers import Dense, Dropout, Activation, Flatten, BatchNormalization
from keras.layers import Conv2D, MaxPooling2D, LeakyReLU
from keras.losses import CategoricalCrossentropy
from tensorflow.keras.optimizers import Adam
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.utils import shuffle
class Function1 :

   def cnn_function(self, paramters):
      return loss(y_test, y_preds).numpy()

class Global_Random_Population_Search :

   def execute_global_random_search(self, function_object, num_parameters = 2, lower_bound = [0,0], upper_bound = [5,5], num_samples
= 100) :

      cost_iteration = []
      time_iteration = []
      optimised_cost = float('inf')
      optimised_params = None
      params_iteration = []

      start_time = time.time() * 1000

      for i in range(num_samples) :
         print(f'num_samples:{i}')
         parameter = [random.uniform(lower_bound[j], upper_bound[j]) for j in range(num_parameters)]
         current_cost = function_object.cnn_function(parameter)

         if current_cost < optimised_cost :
            optimised_cost = current_cost
            optimised_params = parameter

         params_iteration.append(optimised_params)
         cost_iteration.append(optimised_cost)
         time_iteration.append((time.time() * 1000) - start_time)

      return cost_iteration, time_iteration, params_iteration



   def execute_global_population_search(self, function_object, num_parameters = 2, lower_bound = [0,0], upper_bound = [5,5],
num_samples = 100, M_best_params = 10, num_iterations = 200) :

      cost_iteration = []
      time_iteration = []
      optimised_cost = float('inf')
      start_time = time.time() * 1000
      optimised_params = None
      params_iteration = []

      n_sample_cost = []

      for i in range(num_samples) :

         parameter_x, parameter_y = [random.uniform(lower_bound[j], upper_bound[j]) for j in range(num_parameters)]
```

```
        current_cost = function_object.get_quadratic_equation_value(parameter_x, parameter_y)

        if current_cost < optimised_cost :
            optimised_cost = current_cost
            optimised_params = [parameter_x, parameter_y]

        n_sample_cost.append(optimised_cost)
        cost_iteration.append(optimised_cost)
        params_iteration.append([parameter_x, parameter_y])
        time_iteration.append((time.time() * 1000) - start_time)

    for _ in range(num_iterations) :
        print(f'num_iterations: {_}')
        best_cost = sorted(n_sample_cost)[:M_best_params]
        best_params = [params_iteration[n_sample_cost.index(cost)] for cost in n_sample_cost]

        best_cost_iteration = []
        best_param_iteration = []
        best_time_iteration = []

        for best_params in best_params :
            for _ in range(num_samples) :
                sample_parameter_x, sample_parameter_y = [random.uniform(best_params[j] - (upper_bound[j] - lower_bound[j]) * 0.1,
best_params[j] + (upper_bound[j] - lower_bound[j]) * 0.1) for j in range(num_parameters)]
                sample_cost = function_object.get_quadratic_equation_value(sample_parameter_x, sample_parameter_y)

                if sample_cost < optimised_cost :
                    optimised_cost = sample_cost
                    optimised_params = [sample_parameter_x, sample_parameter_y]

                best_cost_iteration.append(optimised_cost)
                cost_iteration.append(optimised_cost)
                best_param_iteration.append(optimised_params)
                best_time_iteration.append(time.time() - start_time)

        params_iteration.extend(best_param_iteration)
        n_sample_cost = best_cost_iteration
        time_iteration.extend(best_time_iteration)

    return cost_iteration, time_iteration, params_iteration

function1 = Function1()
grs = Global_Random_Population_Search()

num_parameters = 5
lower_bound = [16, 0.0001,0.25,0.9,10]
upper_bound = [64,0.01,0.9,0.999,20]
num_iterations = 100

grs_cost_iteration_func1, grs_time_iteration_func1, grs_param_iteration_func1 = grs.execute_global_random_search(function_object =
function1, num_parameters = num_parameters, lower_bound = lower_bound, upper_bound = upper_bound, num_samples =
num_iterations)

function1 = Function1()
grs = Global_Random_Population_Search()

num_parameters = 2
lower_bound = [0, 0]
upper_bound = [10, 10]
num_samples = 200
M_best_params = 10
num_iterations = 200
```

```
gps_cost_iteration_func1, gps_time_iteration_func1, gps_param_iteration_func1 = grs.execute_global_population_search(function_object = function1, num_parameters = num_parameters, lower_bound = lower_bound, upper_bound = upper_bound,  num_samples = 10, M_best_params = 2, num_iterations = 5)

plot.figure(figsize=(11, 6), dpi=150)
plot.semilogy(range(num_iterations), grs_cost_iteration_func1, color = 'cyan', label = 'Global Random Search')
plot.semilogy(range(num_iterations), gps_cost_iteration_func1, color = 'yellow', label = 'Global Population Search')
plot.xlabel('#Iteration')
plot.ylabel('Cost')
plot.title('Plot of Function Cost vs. Iteration for GRS vs. GPS for CNN Function')
plot.yscale("log")
plot.legend()
plot.show()
```