assignment_1

April 10, 2024

1 Assignment 1

Assignment 1: Optimization

Goal: Get familiar with gradient-based and derivative-free optimization by implementing these methods and applying them to a given function.

In this assignment we are going to learn about **gradient-based** (GD) optimization methods and **derivative-free optimization** (DFO) methods. The goal is to implement these methods (one from each group) and analyze their behavior. Importantly, we aim at noticing differences between these two groups of methods.

Here, we are interested in minimizing the following function:

$$f(\mathbf{x}) = x_1^2 + 2x_2^2 - 0.3\cos(3\pi x_1) - 0.4\cos(4\pi x_2) + 0.7$$

in the domain $\mathbf{x} = (x_1, x_2) \in [-100, 100]^2$ (i.e., $x_1 \in [-100, 100]$, $x_2 \in [-100, 100]$).

In this assignemnt, you are asked to implement: 1. The gradient-descent algorithm. 2. A chosen derivative-free algorithm. *You are free to choose a method.*

After implementing both methods, please run experiments and compare both methods. Please find a more detailed description below.

1.1 1. Understanding the objective

Please run the code below and visualize the objective function. Please try to understand the objective function, what is the optimum (you can do it by inspecting the plot).

If any code line is unclear to you, please read on that in numpy or matplotlib docs.

[84]:
```python
import matplotlib.pyplot as plt
import numpy as np
```

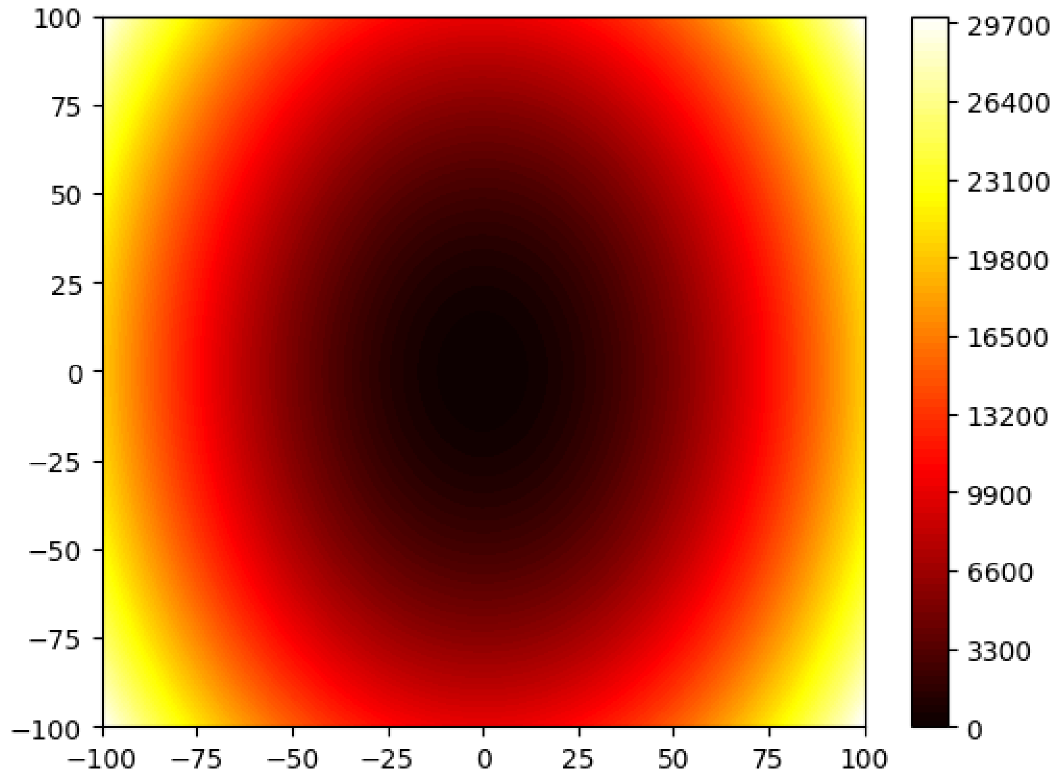
[85]:
```python
# PLEASE DO NOT REMOVE!
# The objective function.
def f(x):

    return (
        x[:, 0] ** 2
        + 2 * x[:, 1] ** 2
        - 0.3 * np.cos(3.0 * np.pi * x[:, 0])
        - 0.4 * np.cos(4.0 * np.pi * x[:, 1])
        + 0.7
    )
```

[86]:
```
# PLEASE DO NOT REMOVE!
# Calculating the objective for visualization.
def calculate_f(x1, x2):
    f_x = []
    for i in range(len(x1)):
        for j in range(len(x2)):
            f_x.append(f(np.asarray([[x1[i], x2[j]]])))

    return np.asarray(f_x).reshape(len(x1), len(x2))
```

[87]:
```
# PLEASE DO NOT REMOVE!
# Define coordinates
x1 = np.linspace(-100.0, 100.0, 400)
x2 = np.linspace(-100.0, 100.0, 400)

# Calculate the objective
f_x = calculate_f(x1, x2).reshape(len(x1), len(x2))
```

[88]:
```
# PLEASE DO NOT REMOVE!
# Plot the objective
plt.contourf(x1, x2, f_x, 100, cmap="hot")
plt.colorbar()
```

[88]:
```
<matplotlib.colorbar.Colorbar at 0x10f82ac90>
```



1.2 2. The gradient-descent algorithm

First, you are asked to implement the gradient descent (GD) algorithm. Please take a look at the class below and fill in the missing parts.

NOTE: Please pay attention to the inputs and outputs of each function.

NOTE: To implement the GD algorithm, we need a gradient with respect to **x** of the given function. Please calculate it on a paper and provide the solution below. Then, implement it in an appropriate function that will be further passed to the GD class.

Question 1 (0-1pt): What is the gradient of the function $f(\mathbf{x})$? Please fill below both the mathematical expression and within the code.

Answer:

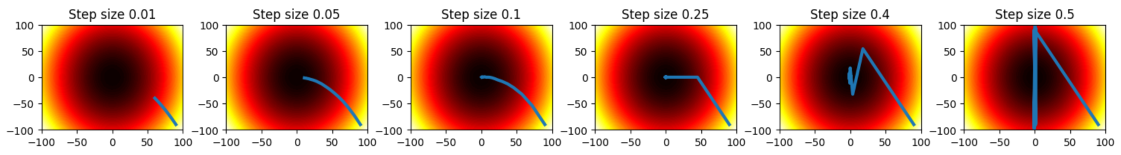
$$\nabla_{\mathbf{x}_1} f(\mathbf{x}) = 2x_1 + 0.9\pi \sin(3\pi x_1) \ \textbf{(0.15 pt)}$$
$$\nabla_{\mathbf{x}_2} f(\mathbf{x}) = 4x_2 + 1.6\pi \sin(4\pi x_2) \ \textbf{(0.15 pt)}$$

[89]:
```
# =========
# GRADING:
# 0
# 0.5pt - if properly implemented and commented well
# =========
# Implement the gradient for the considered f(x).

def grad(x):

    grad_x1 = 2 * x[:, 0] + 0.9 * np.pi * np.sin(3 * np.pi * x[:, 0]) #partial␣
 ↪derivative of the obj func respect to x1
    grad_x2 = 4 * x[:, 1] + 1.6 * np.pi * np.sin(4 * np.pi * x[:, 1]) # partial␣
 ↪derivative of  the obj func respect to  x2

    grad = np.column_stack((grad_x1, grad_x2)) #putting the partial derivatives␣
 ↪into a 2D gradient array. first column respect to x1, second is for x2.

    return grad
```

[90]:
```
# =========
# GRADING:
# 0
# 0.5pt if properly implemented and commented well
# =========
# Implement the gradient descent (GD) optimization algorithm.
# It is equivalent to implementing the step function.
class GradientDescent(object):
    def __init__(self, grad, step_size=0.1):
        self.grad = grad
        self.step_size = step_size

    def step(self, x_old):

        gradient = self.grad(x_old) #calculating the gradient with at the x_old␣
 ↪point.

        #here gradient provides the direction of the optimizer,
        #step size is to determine how far it should move from x_old point and
        #since we are looking for the steepest descent we eliminate self.
 ↪step_size * gradient from x_old

        x_new = x_old - self.step_size * gradient
        return x_new
```

[91]:
```
# PLEASE DO NOT REMOVE!
# An auxiliary function for plotting.

def plot_optimization_process(ax, optimizer, title):
    # Plot the objective function
    ax.contourf(x1, x2, f_x, 100, cmap="hot")

    # Init the solution
    x = np.asarray([[90.0, -90.0]])
    x_opt = x
    # Run the optimization algorithm
    for i in range(num_epochs):
        x = optimizer.step(x)
        x_opt = np.concatenate((x_opt, x), 0)

    ax.plot(x_opt[:, 0], x_opt[:, 1], linewidth=3.0)
    ax.set_title(title)
```

[92]:
```
# PLEASE DO NOT REMOVE!
# This piece of code serves for the analysis.
# Running the GD algorithm with different step sizes
num_epochs = 20  # the number of epochs
step_sizes = [0.01, 0.05, 0.1, 0.25, 0.4, 0.5]  # the step sizes

# plotting the convergence of the GD
fig_gd, axs = plt.subplots(1, len(step_sizes), figsize=(15, 2))
fig_gd.tight_layout()

for i in range(len(step_sizes)):
    # take the step size
    step_size = step_sizes[i]
    # init the GD
    gd = GradientDescent(grad, step_size=step_size)
    # plot the convergence
    plot_optimization_process(
        axs[i], optimizer=gd, title="Step size " + str(gd.step_size)
    )
```



Question 2 (0-0.5pt): Please analyze the plots above and comment on the behavior of the gradient-descent for different values of the step size. What happens in the small and large step sizes and what is the optimum step size?

Answer: For a small step size, more iterations are required to reach the target point, or the minimum, compared to other step sizes. As can be seen from the visualizations, because it proceeds with small steps, the optimizer has terminated without reaching the target point. However, with a large step, due to the step size, there may be deviations from the target point or an inability to precisely land on it. As can be observed from the graphs, the most optimal step size is between 0.1 and 0.25. Because within the given *iterations*(20 in this case), this step size has led the optimizer to the minimum point.

Question 3 (0-0.5pt): How can we improve the convergence when the step size equals 0.01? What about when the step size equals 0.5?

Answer: - For step size 0.01, increasing iterations could be a solution. But there is a possiblity that it can miss the point because of the iterations. So in order to improve the convergence a learning rate schedule can be used. Also there can be a momentum implemantation that helps the optimizer to navigate the landscape more smoothly.

- For the step size 0.5, decreasing iterations could be the solution. But again since we decreasing the iterations and the stepsize is too big, there is also a possibility of missing the minimum point. To prevent overshooting the minimum point, the step size can be proportionally reduced based on the rate of increase in the loss function value.

1.3 3. The derivative-free optimization

In the second part of this assignment, you are asked to implement a derivative-free optimziation (DFO) algorithm. Please notice that you are free to choose any DFO method you wish. Moreover, you are encouraged to be as imaginative as possible! Do you have an idea for a new method or combine multiple methods? Great!

Question 4 (0-0.5-1-1.5-2-2.5-3pt): Please provide a description (a pseudocode) of your DFO method here.

NOTE (grading): Please keep in mind: start simple, make sure your approach works. You are encouraged to use your creativity and develop more complex approaches that will influence the grading. TAs will also check whether the pseudocode is correct.

Answer:

```
Note: The questions is not specifying the type of pseudocode, so i did in my way :)
```

DFO Algorithm Pseudocode:

Input: - obj_fun: A function that computes the value for minimization/maximization - step_size: The size of exploration in each dimension - mode: Either 'min' for minimization or 'max' for maximization, according to the user preferences - bounds: this will determine the bounds for the points so the points for the optimizer will betwen the bounds

class DFO Initialization: 1. Initialize the algorithm with obj_fun, step_size, and mode 2. Create an empty list best_solution_history to keep track of the progress and store the best solutions 3. Create an empty list stuck_points to store the points when the optimizer gets stuck

min_or_max(self, x_new, x_best) method: - If the chosen mode is 'min', return true if obj_fun(x_new) < obj_fun(x_best). - Else return true if obj_fun(x_new) > obj_fun(x_best), which is for 'max'.

step(x_old) method: 1. Set x_new as a copy of x_old to have independent copy of the coordinates 2. Set best_value as the result of obj_fun at x_old

3. For i from 0 to dimensionality of the problem (in this case 2D): For each direction in set {-1, 1}:

    - Create a step vector with zero values for 2D, [[0.0, 0.0]]
    - Set step_size * direction (-1 for neg, 1 for pos) to the i-th dimension of the step vector. [0, i], 0 stands for row. i is for column. each column is for a dimension.
    - Add the step vector to x_old to get a candidate point
    - Check whether the new point is in between the bounds.
    - Calculate candidate_value with the obj_fun at the candidate coordinate
    - If candidate_value is better than best_value (based on mode):
        - Update best_value with candidate_value
        - Update x_new with the candidate point

4. After exploring all directions and dimensions between the bounds: If there is no significant difference between last 5 steps and current point and the difference is below threshold:

    - Print a message to inform about the stuck point
    - Record the current step number in stuck_points
    - Generate random noise to escape the area within the range [-step_size, step_size] for each dimension
    - Add the noise to x_new
    - Check whether the x_new is in between bounds

5. Append x_new to best_solution_history

6. Repeat the step method until the termination criteria are met (like a set number of iterations or convergence threshold)

Output: - Information text that informs the user about the stuck point. - Mode type, step size, last point's coordinate and its obj function value. - Visualization of the optimization process with epochs, x(for each step) and o(stuck point).

Numpy functions that i used:

- np.copy: To create independent copies of arrays.
- np.zeros: For initializing zero-filled arrays.
- np.max: To find the maximum value in an array, used in the escape mechanism.
- np.abs: To calculate absolute values.
- np.clip: To ensure values remain within specified bounds.
- np.random.uniform: For generating random noise within a specified range, used in the escape mechanism.
- np.vstack: convert 3D to 2D.

Visualization:

To have a clear visualization of the optimization process: i changed the bounds(bounds[0][0], bounds[0][1]) so y and x-axes become 0.6. I also wanted to show the process of optimizer to find min or max so i added for steps x, and for stuck points o. For the other not important implementations(implemantations that help the iteration and the effciency of the algorithm Visualization) i made comments.

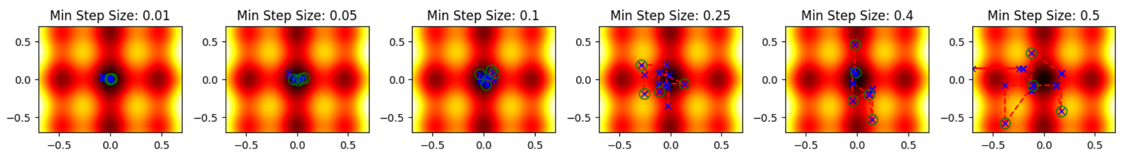
```
[110]: # =========
# GRADING: 0-0.5-1-1.5-2pt
# 0
# 0.5pt the code works but it is very messy and unclear
# 1.0pt the code works but it is messy and badly commented
# 1.5pt the code works but it is hard to follow in some places
# 2.0pt the code works and it is fully understandable
# =========
# Implement a derivative-free optimization (DFO) algorithm.
# REMARK: during the init, you are supposed to pass the obj_fun and other␣
↪objects that are necessary in your method.

class DFO(object):
    def __init__(self, obj_fun, step_size, bounds, mode='min'):
        self.obj_fun = obj_fun
        self.step_size = step_size
        self.bounds = bounds
        self.best_solution_history = []  # Storing the best solution of each␣
↪steps
        self.mode = mode  # Depends on user preferences its either 'min'␣
↪minimization or 'max' maximization
        self.stuck_points = [] # Stores the stucked points

    def min_or_max(self, x_new, x_best):
        if self.mode == 'min': # minimization problem
            return x_new < x_best
        else:  # if not 'min' then 'max', maximization
            return x_new > x_best

    def step(self, x_old):

        #Initialized with default values
        x_new = np.copy(x_old)
        best_value = self.obj_fun(x_old)

        #coordinate descent approach
        for i in range(2): # iterates over the two dimention
            for direction in [-1, 1]:  # Explore both neg and pos directions
                step = np.zeros((1, 2)) # gives us [[0. 0.]] since its for 2D
                step[0, i] = direction * self.step_size #exploring both␣
↪directions negative and positive
                candidate = x_old + step

                candidate = np.clip(candidate, [self.bounds[0][0], self.
↪bounds[1][0]], [self.bounds[0][1], self.bounds[1][1]]) #keeping the␣
↪candidate between the bounds
```

```
                candidate_value = self.obj_fun(candidate)
                if self.min_or_max(candidate_value, best_value):
                    best_value = candidate_value
                    x_new = candidate

        # If optimizer stuck at some point
        if len(self.best_solution_history) > 5:
            previous_point = self.best_solution_history[-5] # taking last 5␣
↪points
            changes = np.max(np.abs(x_new - previous_point)) # Calculating the␣
↪absolute value of the difference

            if changes < 1e-6: # 1e-6 is the threshold
                #If the last 5 solution is not considerably changed, means its␣
↪stuck
                print("The optimizer seems to be stuck. Applying noise to␣
↪escape local extremum...")
                self.stuck_points.append(len(self.best_solution_history))␣
↪#best_solution_history is later used to visualize stuck points on the␣
↪landscape

                #Applying noise method
                noise = np.random.uniform(-self.step_size, self.step_size,␣
↪size=x_old.shape) #randomly assign a coordinate within the range␣
↪-step_size-step_size
                x_new += noise

                x_new = np.clip(x_new, [self.bounds[0][0], self.bounds[1][0]],␣
↪[self.bounds[0][1], self.bounds[1][1]]) #keeping the new point between the␣
↪bounds.

        # Updating the history
        self.best_solution_history.append(x_new) #storing x_new and now inside␣
↪best_solution_history they are 3D.
        #print(self.best_solution_history)

        return x_new
```

[112]:
```
# PLEASE DO NOT REMOVE!
def plot_optimization_process(ax, optimizer, bounds, title):

    x = np.linspace(bounds[0][0], bounds[0][1], 400)
    y = np.linspace(bounds[1][0], bounds[1][1], 400)
    Z = calculate_f(x, y).reshape(len(x), len(y))
    ax.contourf(x, y, Z, 100, cmap='hot')
    ax.set_title(title)
    # Plot the path of the optimizer
    if len(optimizer.best_solution_history) > 0:
        #since x_new in best_solution_history is 3D, making them array and 2D␣
↪for next operations
        best_solutions = np.vstack(optimizer.best_solution_history)  #3D to 2D

        ax.plot(best_solutions[:, 0], best_solutions[:, 1], 'r--', marker='x',␣
↪markeredgecolor='b')

        # Stuck points with green color w shape o on the graph
        if optimizer.stuck_points:
            stuck_solutions = best_solutions[optimizer.stuck_points]
            ax.plot(stuck_solutions[:, 0], stuck_solutions[:, 1], 'o',␣
↪markerfacecolor='none', markeredgecolor='g', markersize=10)

# Running the DFO algorithm with different step sizes
num_epochs = 30  # the number of epochs (you may change it!)
step_sizes = [0.01, 0.05, 0.1, 0.25, 0.4, 0.5]  # the step sizes

## PLEASE FILL IN
## Here all hyperparameters go.
## Please analyze at least one hyperparameter in a similar manner to the
## step size in the GD algorithm.
bounds = [(-0.7, 0.7), (-0.7, 0.7)]  # Bounds for each dimension

# asking user to choose the mode of the optimizer
mode = input("Type 'min' to minimize or 'max' to maximize the objective␣
↪function: ").strip().lower()
while mode not in ['min', 'max']:
    print("Invalid mode. Please enter 'min' for minimization or 'max' for␣
↪maximization.")
    mode = input("Type 'min' to minimize or 'max' to maximize the objective␣
↪function: ").strip().lower()



## plotting the convergence of the DFO
## Please uncomment the two lines below, but please provide the number of axes␣
↪(replace HERE appriopriately)
fig_dfo, axs = plt.subplots(1, len(step_sizes), figsize=(15, 2))
fig_dfo.tight_layout()

# the for-loop should go over (at least one) parameter(s) (replace HERE␣
↪appriopriately)
# and uncomment the line below
for i, step_size in enumerate(step_sizes):
```

```
##    PLEASE FILL IN
     dfo = DFO(f, step_size, bounds, mode=mode)

#     plot the convergence
     x0 = np.random.uniform(bounds[0][0], bounds[0][1], (1, 2)) # randomly␣
 ↪initialed point in 2D
     for _ in range(num_epochs): # runs for num_epochs iteration
          x0 = dfo.step(x0)  # iterate through the running optimizer
     final_value = f(x0)

#     please change the title accordingly!

     print(f"Mode: {mode.capitalize()}, Final coordinates for step size␣
 ↪{step_size}: {x0}, Objective function value: {final_value}")
     plot_optimization_process(axs[i], dfo, bounds, title=f"{mode.capitalize()}␣
 ↪Step Size: {step_size}")

plt.show()
```

```
The optimizer seems to be stuck. Applying noise to escape local extremum…
The optimizer seems to be stuck. Applying noise to escape local extremum…
The optimizer seems to be stuck. Applying noise to escape local extremum…
Mode: Min, Final coordinates for step size 0.01: [[0.0003471  0.00462922]],
Objective function value: [0.0007212]
The optimizer seems to be stuck. Applying noise to escape local extremum…
The optimizer seems to be stuck. Applying noise to escape local extremum…
The optimizer seems to be stuck. Applying noise to escape local extremum…
The optimizer seems to be stuck. Applying noise to escape local extremum…
Mode: Min, Final coordinates for step size 0.05: [[ 0.01899922 -0.00084254]],
Objective function value: [0.00518153]
The optimizer seems to be stuck. Applying noise to escape local extremum…
The optimizer seems to be stuck. Applying noise to escape local extremum…
The optimizer seems to be stuck. Applying noise to escape local extremum…
The optimizer seems to be stuck. Applying noise to escape local extremum…
Mode: Min, Final coordinates for step size 0.1: [[-0.04005786 -0.03245418]],
Objective function value: [0.05764524]
The optimizer seems to be stuck. Applying noise to escape local extremum…
The optimizer seems to be stuck. Applying noise to escape local extremum…
The optimizer seems to be stuck. Applying noise to escape local extremum…
The optimizer seems to be stuck. Applying noise to escape local extremum…
Mode: Min, Final coordinates for step size 0.25: [[-0.00868766  0.0567124 ]],
Objective function value: [0.10486536]
The optimizer seems to be stuck. Applying noise to escape local extremum…
The optimizer seems to be stuck. Applying noise to escape local extremum…
The optimizer seems to be stuck. Applying noise to escape local extremum…
The optimizer seems to be stuck. Applying noise to escape local extremum…
The optimizer seems to be stuck. Applying noise to escape local extremum…
Mode: Min, Final coordinates for step size 0.4: [[-0.02480055  0.0595446 ]],
Objective function value: [0.12271441]
The optimizer seems to be stuck. Applying noise to escape local extremum…
The optimizer seems to be stuck. Applying noise to escape local extremum…
The optimizer seems to be stuck. Applying noise to escape local extremum…
The optimizer seems to be stuck. Applying noise to escape local extremum…
Mode: Min, Final coordinates for step size 0.5: [[-0.12035743 -0.15492883]],
Objective function value: [0.78259029]
```



Question 5 (0-0.5-1pt) Please comment on the behavior of your DFO algorithm. What are the strong points? What are the (potential) weak points? During working on the algorithm, what kind of problems did you encounter?

Answer:

How does it work?

This DFO algorithm designed to be user friendly and shows the process of optimization clearly. The algorithm focused on 2D. I combined coordinate descent with other numpy functions to make it more efficient. Normally, coordinate descent is for minimization problems but since question asked to be creative, in this algorithm i focused on both minimizaiton and maximization problem. The model asks whether user want to maximize or minimize. Optimizer then explore both negative and positive directions through iterating over each dimention to find the optimum coordinate for objective function at a time, while keeping other coordinates fixed (this part is: for i in range(2)…candidate=step+x_old). But there is a posibility in coordinate descent that optimizer could get stuck at some point(e.g. local minima) if it does not find any optimum coordinate around it. To solve this problem and make the algorithm working system more clear, i implemented a part that adds the stuck point a noise that helps optimizer to escape stuck point. Noise is ramdomly added with help of numpy between the values of (-)step size. And it is also shown on the pictures of the landscape with green circles. The potential optimal values that i found during trial-and-error of algorithm for noise implementation threshold and iteration is 1e-6 and 30 respectively. Also there are blue 'x' on the visual landscape for step size in each iteration, so to see how many distance the optimizer took in each iteration with the different steps.

The strong points: - Its derivative-free so it can be used on the non-differentiable functions. - The DFO method coordinate descent, simplifies the optimization process by breaking it down into one-dimensional problems (iterating over the columns, which are the dimensions). - This algorithm can be used both for minimization and maximization problems. - This has an escape mechanism from local minima/maxima if it stuck, which helps to solve optimization problem efficiently.

The weak points: - With the complicated functions and high-dimensional problems i dont think the algorithm would be efficient as e.g. gradient descent. Because with the high-dimensional problems, the computation cost increases. - The escape mechanism could not work efficiently on some complex formulas which might have quite complex landspace. - There could be a efficiency problem if the step size chosen wrongly which could lead optimizer to skip optimum point. - Could be slower if the number of dimensions increases.

Problem that I encounter: Understanding how to navigate the coordinates systems and how to manage multiple dimensions was quite challenging. In order to implement the system i had to understand coorditane descent and how to combine it with the system i wanted. Also combining algorithm with the visual parts of the code was challenging. And for the escape mechanishm, the logic of this part (changes = np.max(np.abs(x_new - previous_point))) took quite some time to implement.

1.4 4. Final remarks: GD vs. DFO

Eventually, please answer the following last question that will allow you to conclude the assignment draw conclusions.

Question 6 (0-0.5pt): What are differences between the two approaches?

Answer: - Gradient descent uses gradient information to determine the direction of the steepest descent. For it to find the minimum, the objective function absolutely must be differentiable. On the other hand, DFO does not require the objective function to be differentiable and does not use derivatives to find the optimal solution. This can be advantageous at times when deriving some objective functions is costly and nearly impossible.

- Gradient descent can be more efficient in finding a local minimum if the objective function is smooth and the gradient can be calculated more accurately. Conversely, DFO can be more costly and require more computation to find the optimum, as it uses a trial-and-error method.
- Gradient descent can be disadvantageous when its used with noisy, discontinuous, or non-differentiable functions, but it is advantageous for smooth and continuous problems. Conversely, the applicability of DFO is multifaceted.
- Because of the trial-and-error nature of DFO, it may get stuck at suboptimal points, and an escape method should be applied at this point. Gradient descent might not work effectively in complex functions. Likewise, while gradient descent can quickly find local minima, it can get stuck at some minima. DFO may be better in discovering the paths to the minimum as it is more apt in exploring the landscape.
- Adjusting the step size in gradient descent can be more challenging compared to DFO.
- Gradient descent usually used for the continuous functions and the DFO used for discontinuous Functions.

Question 7 (0-0.5): Which of the is easier to apply? Why? In what situations? Which of them is easier to implement in general?

Answer:

Gradient descent can be more easily applied in differentiable and smooth functions because the gradient descent method has the ability to determine direction and can clearly show the direction towards the function's minimum. On the other hand, DFO can be applied when it is difficult or impossible to take the derivative of the function. DFO relies directly on function evaluations and does not require derivative information, such as in black box optimization problems. In terms of ease of general application, gradient descent is simpler. Including mathematical operations like taking derivatives and calculating gradients facilitates implementation and is effective. However, for this, the function must be smoothly differentiable. As a result, which method can be more easily applied depends on the type of function and the requirements of the problem.