Laboratory 1

Variant 5

Group 5

By Jan Szachno and Aleksandra Głogowska

---

1. Introduction

1.1. Task description

The task was to write a Python program that minimizes a given function using the straight gradient descent algorithm. The program's output should be the found result x and the number of iterations needed to find the minimum. The task included using a code template, that we had to fill with our solution. We were also to provide the visualization of our function in ranges [-5,5] for the x and y parameters. The task requires writing the code, testing it, and discussing the impact of different initial vectors and learning rates. The function that we have been given to minimize is $f(x, y) = 2sin(x) + 3cos(y)$.

1.2. Algorithm description

The straight gradient descent is an optimization algorithm used to iteratively minimize a function. It takes a function that is dependent on one or multiple variables. In our case, the function is dependent on two variables - x and y. This allows us to visualize the output of this function as a '3d surface plot'. For the minimization process, other parameters are needed, such as the learning rate and the initial point. The learning rate is a coefficient dictating the step size by which the current point is being moved with each iteration. As the naming suggests, the initial point is the (x,y) coordinate that the algorithm has to take as the starting point and then move from it according to further instructions.

Given that the function, the initial point, and the learning rate are known, the function's gradient of the initial point must be calculated first. The gradient vector points in the direction where the slope is the steepest, so by taking the negative of it, we get to know where the current (x,y) point has to be moved to reach the descent of the slope. The current point is being moved by some distance, depending on the learning rate. The cycle repeats, with every new point as a parameter until it gets to the minimum. In our case the distance that the point is being moved by is calculated by multiplying the gradient of the current point and the learning rate. The loop ends if the distance gets below the tolerance, convergence criteria equal to 10^{-6}.

The straight gradient descent algorithm is used in machine learning models and neural networks training, as well as some numerical optimizations. The examples of them are visible, for example, in finance, where optimization algorithms are used to construct optimal investment portfolios that maximize returns while minimizing risk, or in statistics, where these techniques are used to estimate parameters of statistical models, such as maximum likelihood estimation (MLE) or maximum a posteriori (MAP) estimation.

The main advantage is that the algorithm is simple in computation and implementation, as well as it is easy to understand even for people who are inexperienced in the field of optimization and machine learning. The minimum can be efficiently found with the correctly adjusted initial parameters and a non-extensive dataset. On the other side, wrongly chosen parameters can lead to a long and not optimal realization of the algorithm. The initial point is definitely not as important as the learning rate choice. If the step size is too large, it may never converge, or it will take a long time, and if the learning rate is too small, it will also be unnecessarily prolonged. Another condition is that the function has to have a derivative in every (x,y) coordinate in the domain, so the function has to be differentiable. In our case, the

function is differentiable, but it is periodic, which means that there are multiple minima in the domain, and the algorithm will only find the one that is the closest to the initial point.

2. Implementation

2.1. Implementation of the algorithm

The algorithm is implemented with the help of four external libraries. The *Numpy* library is used for doing math operations like sine and cosine, handling arrays for efficient numerical computations, and performing linear algebra tasks such as finding vector norms. Additionally, the library is used for creating coordinate grids to visualize functions in 3D space. The *argparse* library is used for parsing command-line arguments, allowing users to specify parameters such as starting positions and learning rate from the command line. The *matplotlib* library is used for creating 3D visualizations of functions, enabling users to inspect the behavior of the function being optimized visually.

The function called *function*, has been defined first and has been shown in Figure 2.1.1. It takes the coordinates (x,y) as floats, and passes them to the $f(x, y) = 2sin(x) + 3cos(y)$. It returns the result of it as a float.

```
def function(x: float, y: float) -> float:
    return 2 * np.sin(x) + 3 * np.cos(y)
```

Figure 2.1.1. - Function called “function”.

The next function is the one that handles the computation of the straight gradient descent. It is called *gradient_descent* and the snippets of it are pictured in Figures 2.2.2 - 2.2.3. For the first parameter, it takes an initial guess, which is the starting point in the computation. It is passed as a tuple of two floats, x and y. The second parameter is the learning rate, the step size, which is also a float. Additional parameters are provided as defaults, the tolerance is set to 10^{-6}, since it is so close to 0 that the results of the algorithm are still accurate, and the maximal iterations parameter is set to 1000. This is the limit of the iterations that could be done in case of finding a minimum of the objective function. The gradient_descent returns the coordinates of the locally found minimum (as a tuple of floats - x and y) and the number of iterations that have been executed. Both information are returned in a tuple.

```python
def gradient_descent(
    initial_guess: Tuple[float, float],
    learning_rate: float,
    tol=1e-6,
    max_iter=1000,
) -> Tuple[Tuple[float, float], float]:
    """Find local minimum of function

    Args:
        initial_guess (Tuple[float, float]): point at which we start searching for the local minimum
        learning_rate (float): coefficient dictating size of step made during the search
        tol (_type_, optional): accuracy at which we consider calculations "good enough". Defaults to 1e-6.
        max_iter (int, optional): maximum allowed amount of iterations. Defaults to 1000.

    Returns:
        Tuple[Tuple[float, float], float]: tuple containing coordinates of the local minima, amount of iterations needed
        to reach the point
    """
    x, y = initial_guess
```

Figure 2.2.2. - First snippet of the gradient_descent function.

The main part of this function is the loop that will repeat itself either until it reaches the limit of the iterations passed as a parameter, or (more likely) if the distance vector length between the current and previous position is less than the tolerance. In each iteration, there is a calculation of the gradient, so there are the partial derivatives of the $f(x, y) = 2sin(x) + 3cos(y)$ of both x and y components, which are respectively $2cos(x)$ and $-3sin(y)$. Having the gradient of the current coordinate, the new position has to be calculated. The gradient (array) and the learning rate are multiplied by each other creating the actual step for the next point. This step is being subtracted from both current coordinates x and y to define a new position. The subtraction signifies the movement towards the local minimum of the function. As the positive gradient indicates the direction of the steepest ascent, taking the negative gradient guides the algorithm towards the steepest descent, representing the quickest path towards reaching the local minimum. Next, the displacement vector is being calculated, and then the length of it, using the numpy *linalg.norm* function. At the end of every loop, the convergence criteria is being checked to see if the local minimum has been already found. If not, the current values of x and y change to the new previously calculated ones and the next iteration begins.

```python
    for i in range(1, max_iter + 1):
        # calculate partial derivatives
        df_dx = 2 * np.cos(x)
        df_dy = - 3 * np.sin(y)

        # calculate gradient
        grad = (df_dx, df_dy)

        # find next point to move to
        new_x, new_y = np.array([x, y]) - learning_rate * np.array(grad)

        # calculate displacement vector between current and next position
        displacement_vector = np.array([new_x, new_y]) - np.array([x, y])
        # find length between current and next position
        displacement = np.linalg.norm(displacement_vector)
        # if the length is sufficiently small, return the calculated result
        if displacement < tol:
            return (new_x, new_y), i

        # move to the next position
        x, y = new_x, new_y

    # return current position after reaching the iteration limit
    return (x, y), max_iter
```

Figure 2.2.3 - Second snippet of the gradient_descent function.

The next part of the code is the *visualization* function. It creates a 3d surface plot of the $f(x, y) = 2sin(x) + 3cos(y)$ in the range [-5,5] for both x and y coordinates. The four minima (x,y) visible within this range are (-1.5708,-3.14159), (-1.5708,3.14159), (4.71239,3.14159), and (4.71239,-3.14159). The visualization is visible in Figure 2.2.4. The code snippet of the visualization function is shown in Figure 2.2.5. The comments are explaining step by step what the lines of the code do.

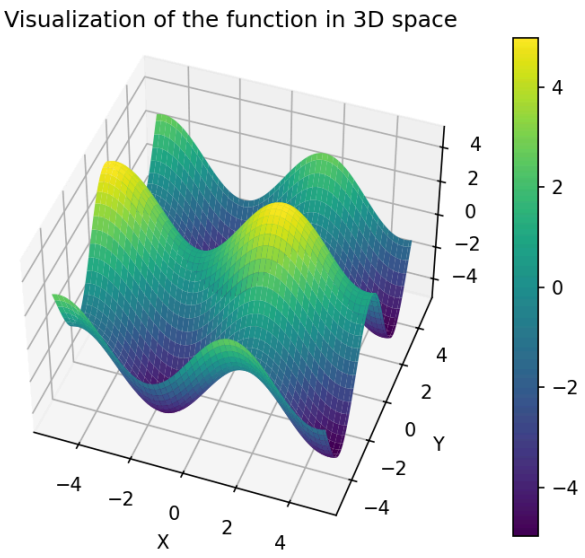

Figure 2.2.4. - Visualization of $f(x, y) = 2sin(x) + 3cos(y)$.

```python
def visualize() -> None:
    """Create a 3d plot of the function"""
    # Generate the x and y coordinate arrays
    x = np.linspace(-5, 5, 100)
    y = np.linspace(-5, 5, 100)
    x, y = np.meshgrid(x, y)

    # Define the function to be plotted
    f = np.vectorize(function)
    z = f(x, y)

    # Create the figure and a 3D subplot
    fig = plt.figure()
    ax = fig.add_subplot(111, projection="3d")

    # Plot the surface with a colormap
    surf = ax.plot_surface(x, y, z, cmap="viridis")

    # Add a colorbar to show the Z-coordinate color mapping
    fig.colorbar(surf)

    plt.xlabel("X")  # Label for the x-axis
    plt.ylabel("Y")  # Label for the y-axis
    plt.title("Visualization of the function in 3D space")  # Title of the plot

    # Show the plot
    plt.show()
```

Figure 2.2.6. - Visualization function code snippet.

The code is being used by either executing the program with command line parameters such as initial point --x, --y, and --rate which is the learning rate. If the user decides to not provide any parameters, the program will take default ones (respectively 0.0, 0.0 and 0.1). The parser takes the arguments and the program passes them to the gradient descent function, prints out the local minimum coordinates along the number of the iterations needed to find it. Later, the visualization function is being called. The code snippet that has been described is shown in Figure 2.2.7.

```python
def main() -> None:
    # Create the parser
    parser = argparse.ArgumentParser()

    # Add arguments
    parser.add_argument("--x", type=float, default=0.0, help="starting x position")
    parser.add_argument("--y", type=float, default=0.0, help="starting y position")
    parser.add_argument("--rate", type=float, default=0.1, help="learning rate")

    # Parse the command line arguments
    args = parser.parse_args()
    initial_guess = (args.x, args.y)
    learning_rate = args.rate

    # Compute the result
    minimum, num_iterations = gradient_descent(initial_guess, learning_rate)

    # Print the result
    print(
        f"Minimum approximation with initial guess {initial_guess}: {minimum}, Iterations: {num_iterations}"
    )

    # Visualize the function
    visualize()
```

Figure 2.2.7. - Code snippet of the main function.

3. Discussion and results

3.1. Testing Correctness of the Implementation

As said before, the minima (x,y) within the range [-5,5] for both x and y are (-1.5708,-3.14159), (-1.5708,3.14159), (4.71239,3.14159), and (4.71239,-3.14159). These values have been calculated outside of this implementation to check the correctness of the code. Let's check different starting points, and see if the results will be the same, as well as how fast they will converge. The outputs of the program are shown in Figures 3.1.1.-3.1.4. These four tests have their initial guesses chosen, so each one is the closest to the one of the four local minima. They are designed to check if the algorithm is correct. The learning rate is set to 0.1 for all of them. The number of iterations for all of them is dependent on how close the initial point were to the local maximum. All four tests prove that the algorithm is implemented correctly, since the results of the minima coordinates are confirmed to be right.

```
Minimum approximation with initial guess (-1.0, -1.0): (-1.5707928596158274, -3.1415926323766477), Iterations: 54
```

Figure 3.1.1. - Output of the program for initial point (x,y) = (-1,-1) and learing rate = 0.1.

```
Minimum approximation with initial guess (2.0, -2.0): (4.712385596063074, -3.1415926535492664), Iterations: 68
```

Figure 3.1.2. - Output of the program for initial point (x,y) = (2,-2) and learing rate = 0.1.

```
Minimum approximation with initial guess (2.0, 2.0): (4.712385596063074, 3.1415926535492664), Iterations: 68
```

Figure 3.1.3. - Output of the program for initial point (x,y) = (2,2) and learing rate = 0.1.

```
Minimum approximation with initial guess (-2.0, 2.0): (-1.5707995313974878, 3.141592645053503), Iterations: 53
```

Figure 3.1.4. - Output of the program for initial point (x,y) = (-2,2) and learing rate = 0.1.

3.2. Learning Rate and Initial Point Dependence

Our program has another two files of code with visualizations, that are created to check how the learning rate and the initial point guess affect the convergence. Both files have the same gradient_descent functions as the main file, but they have different visualizations.

The learning rate testing file, plots a graph of the learning rate and the number of iterations needed to converge, with the initial point 0. The graph is in Figure 3.2.1.

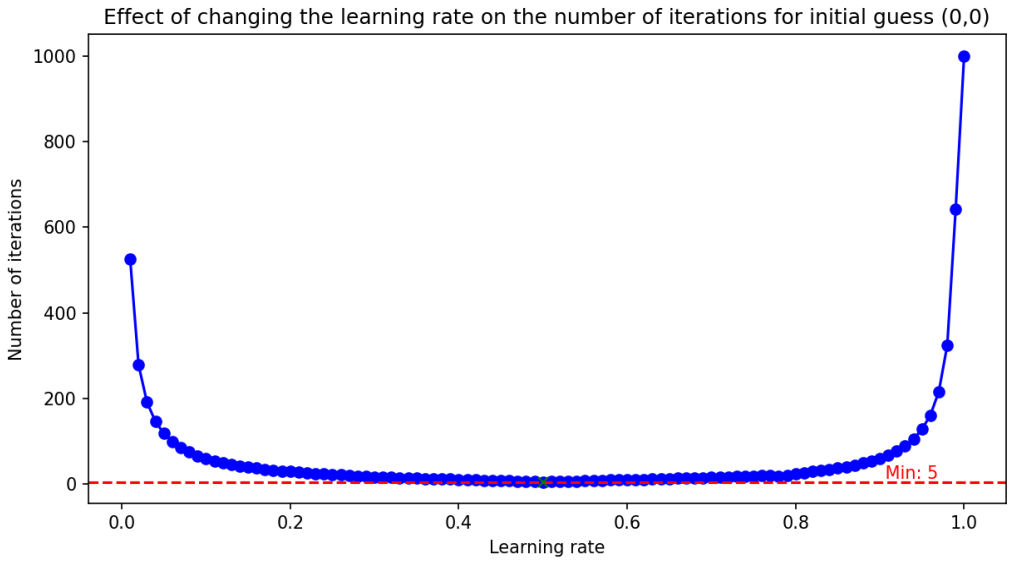

Figure 3.2.1. - Dependence of the learning rate on the number of iterations, with the initial point (x,y) = (0,0).

From the graph we can clearly see that the best learning rate for this function, and the initial point (0,0), is about 0.5 units. The clear conclusion is that if the step size is too small, the algorithm becomes ineffective and the number of iterations increases, but if it does not cross the maximal number of iterations, it will finally converge. However, if the learning rate is too high, the number of experiments will increase until it's too high and the algorithm will not converge. Changing different starting points such as (x,y)= (3.0,3.0) shows almost the same behavior but the shorter optimal learning rate and on the side of the graph where the step sizes are higher, the number of iterations reaches the maximal iterations limit, which is 1000. This graph is depicted in Figure 3.2.2. The minimal number of iterations is also higher, since this initial guess is further from the closest local minimum than in case of the initial guess (x,y) equal to (0,0).

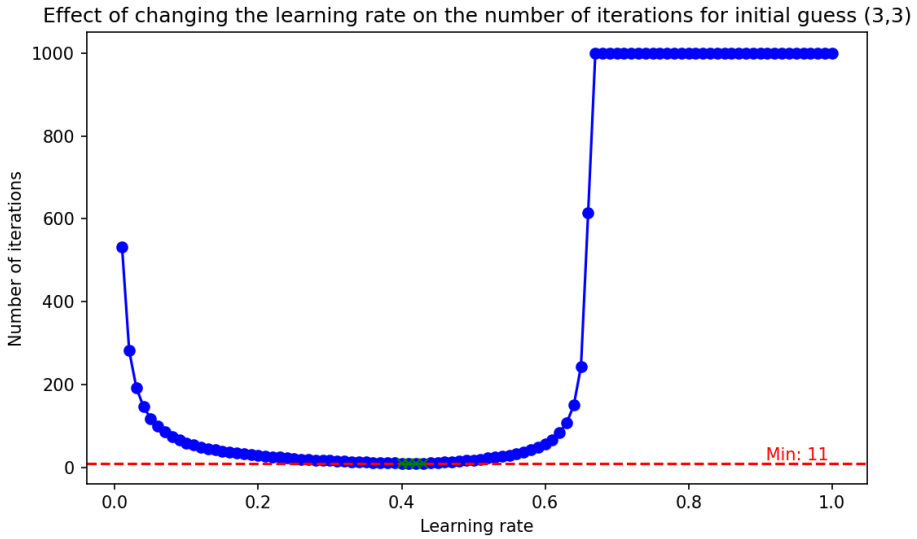

Figure 3.2.2. - Dependence of the learning rate on the number of iterations, with the initial point (x,y) = (3,3).

The initial point file performs an experiment to analyze the effect of changing the starting point on the number of iterations required to reach the local minimum of a given function using gradient descent. The initial guess graph has been created for the unchanging learning rate equal to 0.4 and varying initial points. The plot has been pictured in Figure 3.2.3. In the Figure 3.2.4. there is the graph with the same dependency, but with the step size chosen incorrectly.

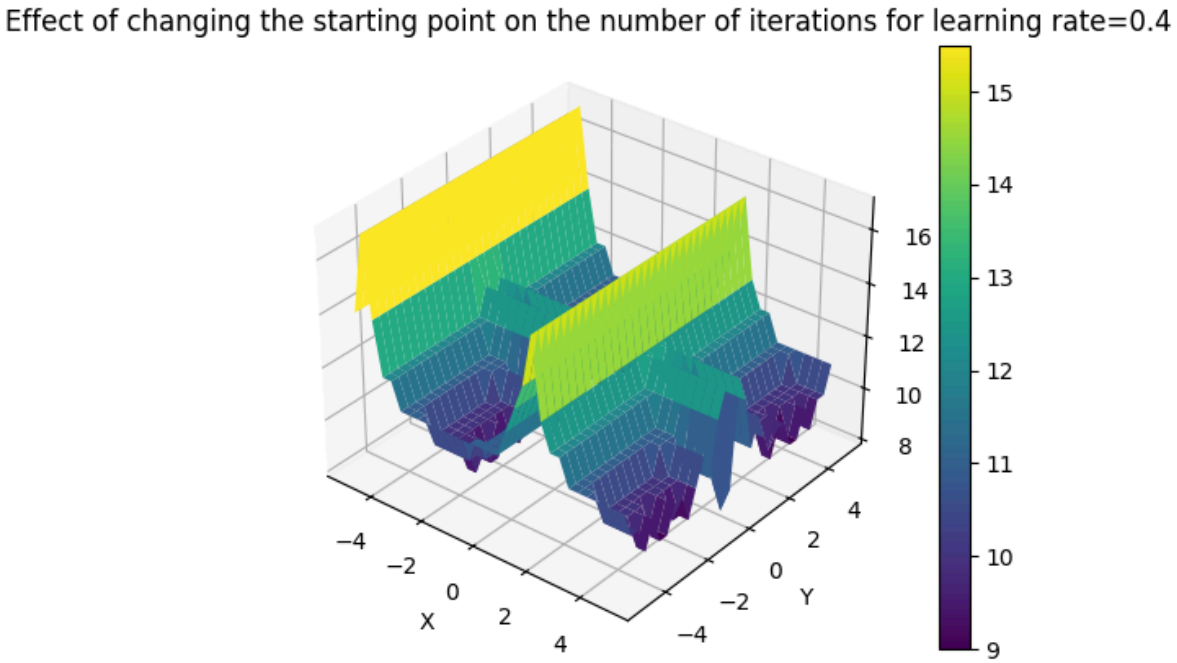

Figure 3.2.3. - Graph showing a dependency between number of experiments and a changing initial point with the learning rate 0.4.

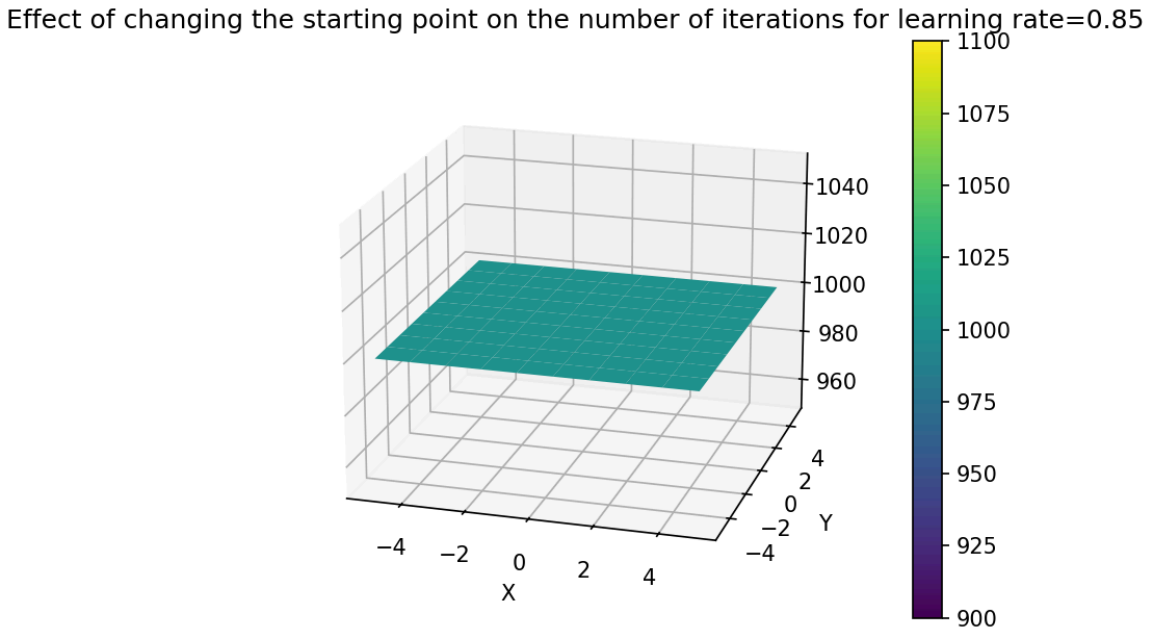

Figure 3.2.4. - Graph showing a dependency between number of experiments and a changing initial point with incorrect learning rate 0.85.

The experiment demonstrates that the number of iterations required to reach the local minimum of the function using gradient descent varies significantly based on the starting point. Points closer to the local minimum generally require fewer iterations, while points farther away require more iterations. Additionally, the experiment highlights the importance of choosing an appropriate learning rate, as it affects the convergence rate and may lead to divergent behavior if improperly selected. The second experiment shows the case where the iterations have reached the limit of 1000, picked for the loop - that is why the graph is flat on the 1000 value. Overall, the visualization provides insights into the sensitivity of the optimization process to the initial conditions and learning rate.

4. Conclusions

The implemented gradient descent algorithm successfully minimized the given function, as evidenced by the results matching the known minima within the specified range. The choice of learning rate significantly affects the convergence behavior of the algorithm. Too small of a learning rate results in slow convergence, while too large of a learning rate may lead to divergence. The optimal learning rate depends on the specific function and initial conditions. The algorithm's convergence rate is highly dependent on the initial point chosen. Points closer to the local minima require fewer iterations to converge, while points farther away require more iterations. This sensitivity emphasizes the importance of selecting appropriate initial points. Visualizing the optimization process provides valuable insights into the behavior of the algorithm under different conditions. It highlights the trade-offs between convergence speed and accuracy and helps identify potential issues such as divergence.

One challenge encountered was determining the optimal learning rate and initial point for convergence. Conducting more comprehensive experiments with a wider range of

initial points and learning rates could provide further insights into the algorithm's behavior. Additionally, this process could be improved by implementing techniques based on heuristics. One such heuristic could be decreasing the learning rate slightly after each iteration such that we progressively take smaller steps as we approach the local minima. This would decrease the risk of overshooting.