Optimization Project Report

[APM_51657_EP] - Optimization

Master-I Applied Mathematics and Statistics

Submitted by:

Eva Mukherjee

Nizalia Siddiqui

December 31, 2024

1 Introduction

Optimization plays a fundamental role in mathematics, engineering, machine learning, and many other disciplines that deal with performing iterative procedures that are applied to minimize or maximize a given function based on gradually improving solutions.

One of the techniques of optimization is steepest descent; an iterative optimization process by which differentiable scalar functions are minimized. In this method, the variable x is iteratively updated as it goes in opposite direction to the gradient of the objective function $f(x)$ that represents steepest ascent at that point. The value of x is updated at each step using the form:

$$x_{k+1} = x_k - \alpha \nabla f(x_k),$$

where α is the step size, and $\nabla f(x_k)$ is the gradient of $f(x)$ at x_k.

The step sizes used in this method may either be constant or non-constant (calculated during each iteration). The process then halts when the $||\nabla f(x)|| = 0$. This indicates that the gradient is nearly equal to a local minimum.

In this report; we have the application of the steepest descent method on two optimization functions: the Rosenbrock function and the Himmelblau function. We implemented the steepest descent method with constant as well as non-constant Armijo's step size using two different starting points $x_0 = (0, 0)$ and $x_\pi = (\pi + 1, \pi - 1)$ and then used the results to analyze the performance of the algorithm in terms of convergence behavior, number of iterations, and the CPU running time while highlighting its sensitivity with respect to step size as well as initial values.

2 Function Definitions

We will be applying the steepest descent algorithm to the Rosenbrock's and Himmelblau's functions, and for the application of the algorithm we will be needing the gradients of these functions. The functions and their gradients are defined as follows:

Rosenbrock's Function:

$$R(x_1, x_2) = (1 - x_1)^2 + 100 * ((x_2 - (x1^2))^2)$$

Rosenbrock's Function's Gradient:

$$\frac{\partial R}{\partial x_1}(x_1, x_2) = -2 * (1 - x_1) - 400 * x_1 * (x_2 - x_1^2)$$

$$\frac{\partial R}{\partial x_2}(x_1, x_2) = 200 * (x_2 - x_1^2)$$

Himmelblau's Function:

$$H(x_1, x_2) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2$$

Himmelblau's Function Gradient:

$$\frac{\partial H}{\partial x_1}(x_1, x_2) = 4x_1(x_1^2 + x_2 - 11) + 2(x_1 + x_2^2 - 7)$$

$$\frac{\partial H}{\partial x_2}(x_1, x_2) = 2(x_1^2 + x_2 - 11) + 4x_2(x_1 + x_2^2 - 7)$$

We have also included the graphs of these functions in our code.

3 Implementation

We will start off our code by importing the libraries and defining the Rosenbrock and Himmelblau functions and their gradients.

3.1 Steepest Descent Algorithm with Constant Step-Size

The code implements the steepest descent algorithm with a constant step size to minimize a function. The algorithm follows a systematic process to iteratively update the input variable x until it converges to a point where the gradient of the function is close to zero, indicating a local minimum.
The execution begins by recording the CPU start time using *process_time*() which is used to measure the runtime of the algorithm. The initial point x_0, provided as input, is assigned to the variable x. An iteration counter is initialized to zero to track the number of steps performed during the optimization process.
The loop continues as long as the norm of the gradient of the function at the current point x is greater than the specified tolerance. This means that the algorithm continues until it is close enough to a local minimum. The current point x is moved in the opposite direction of the gradient by a distance determined by the step size. The iteration counter is incremented to reflect the progress of the algorithm.
Once the loop terminates, the total number of iterations is printed. The CPU end time is recorded using *process_time*() to compute the total runtime.
The algorithm returns the final value of x, which is expected to be close to the local minimum of the function.

3.2 Application of the Steepest Descent Algorithm to the Rosenbrock's and Himmelblau's Function

First, values are assigned to the step size and tolerance. The steepest descent constant function is then called with the initial point x_0, the gradient function specific to the Rosenbrock function, the defined step size and the tolerance. The result of the function, representing the optimized point x, is stored in the result variable. The optimized point obtained from the steepest descent algorithm is printed, which represents the minimum of the Rosenbrock function.
Similarly, the optimized point obtained from the steepest descent algorithm, representing the minimum of the Himmelblau function, is also printed.

3.3 Step-Size Calculation Using Armijo's Condition

The function *armijo_stepsize* is defined to calculate the step size using the Armijo Condition. A scaling factor is initialized which provides an initial estimate for the step size. The initial value of t is computed using a gradient descent formula involving the dot product of the gradient and the descent direction d, scaled by the norm of d. The function iteratively adjusts t in a loop that runs up to *max_iter* times or until the Armijo condition is satisfied. The next point is calculated based on the current step size. If the Armijo Condition is satisfied, then the current step size, t is returned. If the condition is not met, the step size is reduced by a factor of β. To prevent endless iterations, the loop exits if t becomes too small. Then, the function returns the final step size.

3.4 Steepest Descent Implementation

The function *steepest_descent_armijo* employs the steepest descent algorithm combined with the Armijo step size calculation to minimize an objective function f. The starting point x is set to the initial input x_0. The function also specifies a tolerance level for convergence and a maximum number of iterations. At each iteration, the gradient and its norm are computed.
If the gradient norm is smaller than the tolerance, the algorithm terminates and declares convergence. The descent direction d is calculated as the normalized negative gradient.

$$d = -\frac{\nabla f}{||\nabla f||}$$

The step size t is obtained by calling the *armijo_stepsize* function. The current point x is updated. If the algorithm converges, it returns the optimized point. Otherwise, it terminates with a message indicating either divergence or reaching the maximum number of iterations.

3.5 Finding Multiple Minima For Himmelblau's Function

The function $find\_multiple\_minima$ identifies and returns unique minima of a given objective function f by performing optimization from multiple random starting points within specified bounds.
A function named $find\_multiple\_minima$ that takes the objective function, its gradient function, the search bounds, and the number of starting points as input is defined. An empty list, minima, is initialized to store the unique minima found. The function $np.random.uniform$ generates the number of random starting points within the specified bounds. Each starting point is a 2-dimensional vector since Himmelblau's function is defined in two dimensions. A loop is initiated that iterates through each generated starting point x_0. Inside the loop, the *steepest_descent_armijo* function is called with x_0 to find the corresponding local minimum. The result is stored in result variable. If result is *None*, the optimization did not converge, and the point is skipped. If a valid minimum is found, it will round the result to 4 decimal places to prevent duplicates due

to slight variations in floating-point precision. Then, the rounded result is compared to existing minima in the list using *np.allclose*. If the rounded result is unique, it is added to the minima list. After processing all starting points, the function returns the list of unique minima.
The bounds for the starting points are defined. The *find_multiple_minima* function is called to find the minima of Himmelblau's function using the specified bounds and the corresponding function and gradient definitions.

4 Results

4.1 Constant Step Size

4.1.1 Rosenbrock's Function

- $x_0 = (0, 0)$:
  When the step size, $t = 0.0005$ the solution converges to $x = (1., 1.)$ in 120271 iterations with a CPU running time of 1.6404995749999998 seconds and the value of the function at x is $1.2516258105461727e - 22 \approx 0$
  When we decrease the step size further (for instance 0.0005) the solution still converges to the same value but takes more iterations and much longer, as we would expect, and when we increase the step size (for instance 0.005) the algorithm fails to converge and we do not get a result.

- $x_\pi = (\pi + 1, \pi - 1)$:
  When the step size, $t = 0.0005$ the solution fails to converge (in 8 iterations). But when we take a smaller step size, $t = 0.0005$, it converges to the expected value, $x = (1., 1.)$ but it takes 1296467 iterations with a CPU running time of 16.78345112400001

4.1.2 Himmelblau's Function

- $x_0 = (0, 0)$:
  When the step size, $t = 0.0005$ the solution converges to $x = (3., 2.)$ in 2238 iterations with a CPU running time of 0.05862229800000307 seconds and the value of the function at x is $1.9379977413000806e - 24 \approx 0$
  When we decrease the step size further (for instance 0.0005) the solution still converges to the same value but takes more iterations (22554) and much longer, as we would expect, and when we increase the step size (for instance 0.005) the algorithm gives the expected result in much fewer iterations and a lesser CPU running time

- $x_\pi = (\pi + 1, \pi - 1)$:
  When the step size, $t = 0.0005$ the solution converges to $x = (3., 2.)$ in 2093 iterations with a CPU running time of 0.04579011299999536 seconds and the value of the function at x is $1.9106968430921918e - 24 \approx 0$.

Increasing the step size to $t = 0.005$ results in the algorithm converging to the same result in fewer iterations and lesser time while decreasing the step size to $t = 0.0005$ causes the algorithm to take much longer to converge to the result.

4.2 Armijo's Step Size

For the initial step size, t we take:

$$t = -\gamma \frac{\nabla f(x)^T.d}{||d||}$$

We take different values of γ to test how our algorithm behaves.

4.2.1 Rosenbrock's Function

- $x_0 = (0, 0)$:

  When we take $\gamma = 0.1$ the solution converges to $x = (1., 1.)$ in 26199 iterations with a CPU time of 2.23986786399999 seconds and the value of the function at x is $7.462882692830935e - 23 \approx 0$
  When we decrease the value of γ to 0.001 the solution converges to the same value of x but takes more iterations. When we increase the value of $\gamma = 1$ the algorithm still takes longer to converge than when γ was 0.1.

- $x_\pi = (\pi + 1, \pi - 1)$:

  When the step size, $\gamma = 0.1$ the solution converges to $x = (1., 1.)$ in 26695 iterations with a CPU running time of 3.308606993000012 seconds and the value of the function at x is $7.792105263298982e - 23 \approx 0$.
  Taking a either a lower of higher value of γ, ($\gamma = 0.001$ or $\gamma = 1$) results in the algorithm taking longer to converge.

4.2.2 Himmelblau's Function

- $x_0 = (0, 0)$:

  When the step size, $\gamma = 0.1$ the solution converges to $x = (3., 2.)$ in 37 iterations with a CPU running time of 0.00795972499997788 seconds and the value of the function at x is $6.178492716044852e - 25 \approx 0$.
  When we decrease the value of $\gamma = 0.001$ the solution converges to the same value but the number of iterations (1114) and the CPU running time increases. Increasing the value of γ to $\gamma = 1$ does not affect the number of iterations the algorithm takes to converge.

- $x_\pi = (\pi + 1, \pi - 1)$:

  When the step size, $\gamma = 0.1$ the solution converges to $x = (3., 2.)$ in 33 iterations with a CPU running time of 0.005538691999987577 seconds and the value of the function at x is $1.2936435321410746e - 24 \approx 0$.
  Decreasing the value of γ causes it to take much longer but increasing γ to 1 does not have a major impact on it.

4.3 Finding Multiple Minima for Himmelblau's Function

By theory we know that the Himmelblau's Function has four different global minimas, but our algortihm was only able to provide us with one point, $x = (3., 2.)$. So, to get the remaining values of the minima we added another part to our code Using this code we get the four points: $(3.5844, -1.8481), (-2.8051, 3.1313), (3., 2.)$ and $(-3.7793, -3.2832)$. and we can confirm that all of these points provide us with the global minima by finding the value of the function at each of these points. We can check that the value of the Himmelblau's function at each of these points if equal to 0 (the minima). We can say that the Himmelblau's function exhibits four unique minimas.

5 Comparision

We can see that when we use Armijo's non-constant step size the number of steps have reduced significantly for both Rosenbrock's and Himmelblau's functions. But at the same time the CPU running time per step for the Rosenbrock's function has also increased, this is due to the increased number of computations per step.
The constant step size is much easier to apply and the code for it is much simpler, but it has a slower convergence rate and may not always provide a result for the Rosenbrock's function for the given starting points. The non-constant, Armijo's step size, meanwhile, is adaptive, and provides convergence at a much faster rate, but at the same time it requires a lot of computation.
The different starting points also have an effect on the number of iterations it requires to converge, especially for the Rosenbrock's function. For the Rosenbrock's function it takes a longer time to converge when we use the starting point $x_\pi = (\pi + 1, \pi - 1)$, this is because this value is a greater distance from the solution $x = (1, 1)$, and it does not even converge for the constant step size case. For the Himmelblau's function, it takes longer to converge when we take the starting point $x_0 = (0, 0)$ for the same reason, but the difference in the Himmelblau's function is not as significant as in the case of Rosenbrock's function.