Comparative Study of Optimization Techniques: Gradient Descent Minimization For Rastrigin Function and Simulated Annealing

Edgar Rodriguez[a,1], **Saishnu Ramesh Kumar**[b,1], **Ryan Vu Hang**[c,1]

[a]College of Science and Mathematics; [b]Department of Computer Science

Professor Athanasios Aris (Thanos) Panagopoulos

1. Motivation and Problem Statement

In the field of optimization and Minimization, scholars often seek help in the realm of Computational Mathematics and Computer Science to achieve their goals. In this report, two optimization algorithms from the sphere of computational mathematics—Gradient Descent and Simulated Annealing—are among these algorithms that aid in the resolution of complex problems with their distinctive approaches to finding optima.

This report delves into the comparative analysis of these two algorithms by applying them to a well-known, complex function—the Rastrigin function. The purpose of this study is not only to understand the operational mechanisms of each algorithm but also to determine their efficiency and adaptability under the constraints of a highly oscillatory landscape that the Rastrigin function presents.

2. Methodology

2.1. Overview

The testing used was designed to provide a comprehensive evaluation of prominent optimization algorithms: Gradient Descent and Simulated Annealing. These algorithms were tested under the challenging conditions presented by the Rastrigin function, a widely recognized benchmark in optimization research. The approach incorporates the implementation of these algorithms in Visual Studio Python, utilizing NumPy and SymPy to ensure focus on the algorithms' mechanisms. Preliminary testing involved simpler functions to validate their implementations, followed by rigorous testing using the two-dimensional Rastrigin Function to evaluate performance under more demanding conditions.

2.2. Gradient Descent

Gradient Descent is a first-order optimization algorithm that iteratively moves toward the minimum of a function by taking steps proportional to the negative of the gradient. This is due to the gradient being the direction of greatest ascent. The idea is simple yet powerful: move in the direction opposite of the gradient as the function will decrease the fastest there.

In addition, the Gradient Descent with momentum was chosen to potentially improve the convergence speed and stability of the algorithm. The algorithm's parameters include a cosine-adjusted learning rate to modulate the step size dynamically across iterations, a momentum coefficient to maintain directionality in successive steps, and epsilon for convergence threshold. The pseudocode provided in the class formed the basis for the Python implementation, with modifications to accommodate the momentum feature and the cosine learning rate formula:

$$\alpha(t) = \alpha_{\text{init}} \cdot \frac{1 + \cos\left(\frac{\pi \cdot t}{T}\right)}{2}$$

where:

- $\alpha(t)$ is the learning rate at iteration t,
- α_{init} is the initial learning rate,
- t is the current iteration, and
- T is the total number of iterations.

Momentum helps Gradient Descent by accumulating a velocity vector in directions of persistent reduction in the function's value. In turn, this will dampen oscillations and accelerate convergence in shallow gradients. The momentum term increases for dimensions whose gradient points in the same direction and reduces updates for dimensions whose gradient changes directions.

2.3. Simulated Annealing

Simulated Annealing is an algorithm inspired by the annealing process in welding, a technique involving heating and controlled cooling of material to reduce defects. In the context of optimization, Simulated Annealing, or SA for short, seeks to find the minimum of a function by randomly exploring the search space, accepting not only improvements but also, a certain probability. This probabilistic acceptance helps the algorithm escape local minima.

The key to SA's effectiveness lies in its temperature-controlled exploration. Initially, when the temperature is high, the algorithm is more likely to accept worse solutions, allowing it to explore the search space freely and escape local optima. As the temperature decreases, the algorithm becomes increasingly conservative, refining its search around the best solutions found.

The SA implementation was adapted to include an adaptive mechanism allowing the algorithm to modify its exploration based on the Metropolis criterion from the optimization landscape.

$$P(\Delta E, T) = e^{(-\frac{\Delta E}{T})}$$

where:

- $P(\Delta E, T)$ is the probability of accepting a new solution with a change in energy ΔE at temperature T.
- $\Delta E = E_{\text{new}} - E_{\text{current}}$ represents the change in energy between the new solution and the current solution. If the new solution has lower energy, ΔE is negative, leading to a higher acceptance probability.
- T is the current temperature,

The SA cooling schedule also employed a logarithmic decay to decrease the "temperature" parameter gradually, influencing the probability of accepting worse solutions and thus enabling escape from local minima. This cooling schedule in turn allows sufficient exploration of the solution space at early stages while gradually focusing on exploitation by accepting fewer

worse solutions as the process advances. This approach was encapsulated in a Python program, guided by the pseudocode, and adjusted to integrate the logarithmic cooling schedule as follows:

$$T(t) = \frac{T_0}{1 + k \cdot \log(1 + t)}$$

where:

- $T(t)$ represents the temperature at time step t.
- T_0 is the initial temperature at the start of the algorithm.
- k is a positive constant that controls the rate of cooling.
- t is the current iteration or time step, starting from 0.

2.4. Benchmark Function: The Rastrigin Function

The Rastrigin function is a popular test case in the field of optimization as a large number of local minima characterizes it. It is a benchmark for evaluating the robustness and efficiency of optimization algorithms. Hence, it is particularly useful for testing an algorithm's ability to avoid local minima and converge towards a global minimum.

The universal form of the Rastrigin function for an n dimensional vector $\chi = (x_1, x_2 ... x_n)$ is given by:

$$f(\mathbf{x}) = An + \sum_{i=1}^{n} \left[x_i^2 - A\cos(2\pi x_i)\right] \quad (1)$$

where:

- A is a constant that typically equals 10,
- n is the dimensionality of the function, indicating the number of variables,
- $\mathbf{x} = (x_1, x_2, ..., x_n)$ represents the variables of the function, with each x_i being a component of the vector $\mathbf{x}$.

For this experiment, the two-dimensional version of the Rastrigin, which simplifies the universal form, would suffice. This form still maintains the key characteristics of a high frequency of local minima and the global minimum at the origin, making it an ideal test scenario for this study. This version is described by the equation:

$$f(\mathbf{x}, \mathbf{y}) = 20 + x^2 + y^2 - 10 \cdot (\cos(2 \cdot \pi \cdot x) + \cos(2 \cdot \pi \cdot y)) \quad (2)$$

2.5. Programming Environment and Approach

The algorithms were crafted in Python, utilizing Visual Studio as the IDE benefits from useful development features. The libraries used were only NumPy and SymPy. SymPy is used for the sake of convenience and symbolic mathematics for understanding Gradient Descent computations. The code itself is structured in a modular fashion with separate components for each algorithm's logic, the Rastrigin function evaluation, and outputting results. Emphasis was placed on clarity and modularity to support easy understanding, adaptation, and testing.

Lastly, there are two files for each algorithm and their application. The first file contains the Gradient Descent algorithms with the Python function called gradient_descent(). This function asks for the mathematical expression, initial parameters in the NumPy array, initial learning rate, the gamma coefficient for momentum, max iterations, and epsilon for convergence threshold. The general_gradient_compute() asks for the SymPy expression and its variables to compute the gradient of any given function. The last function cosine_learning_rate() is a helper function that simply computes the cosine learning rate of a given time. The second file has the function called adaptive_simulated_annealing() which is the simulated annealing algorithm that asks for the mathematical expression, initial guess, the max iterations, initial temperature, and initial kappa. Kappa in this case is a controlled constant for the cooling schedule. It also has a function called logarithmic_cooling() where it calculates the temperature at the current time t using the logarithmic cooling schedule.

3. Experiment

When delving into the specifics of the experimental setup, it is important to note that parameters must be chosen precisely to accurately evaluate the performance of the Simulated Annealing (SA) and Gradient Descent (GD) algorithms. Due to this reason, the parameters for each algorithm are very similar.

3.1. Simulated Annealing

The Simulated Annealing algorithm was tested to validate its functionality and to observe its performance under different initial conditions. The primary parameters set for this experiment were:

- Maximum iterations (max_iter): 10,000
- Initial temperature (initial_temp): 100.0
- Cooling constant (Cooling_constant): 10.0
- Convergence threshold: 1×10^{-6}

The initial conditions were randomized to test the algorithm's robustness across various starting points. It is important to note that these random variables must include some close to the origin, one at the origin, others around the various optimal, and some far from the solution. The initial solution candidates tested included:

$$(2, -2), (1.5, -1), (8, 3), (0, 0), (0.1, 0.1), (0.5, 0.7)$$

3.2. Gradient Descent

The Gradient Descent algorithm was similarly evaluated, with the following parameters:

- Maximum iterations (max_iter): 10, 000
- Gamma coefficient (gamma_coef): 0.8 (noted that this could be set between 0.8 and 0.9 to observe effects on convergence)
- Initial cosine learning rate: 0.01
- Convergence threshold: 1×10^{-6}

The initial conditions that were chosen provided a diverse set of starting points across the function's landscape, from near the global minimum at (0,0) to significantly far positions. Like SA, the random initial positions used for testing the algorithm's efficacy and adaptability were:

$$(2, -2), (1.5, -1), (8, 3), (0, 0), (0.1, 0.1), (0.5, 0.7)$$

These positions were selected to evaluate the algorithm's capability to navigate the complex topology of the Rastrigin function and converge towards the global minimum.

3.3. Testing Environment

The experiments were conducted using Python, leveraging the NumPy and SymPy libraries to handle numerical operations and symbolic mathematics.

As for setup, each run was initialized with an array with a unique starting position, and the results included the best position, and energy were logged. Also, the convergence of both algorithms was monitored and recorded, highlighting the iteration count upon final position.

The algorithm was first tested with a simple 2D function to evaluate whether or not the algorithms were ready for testing. The function utilized is the following:

$$f(x) = x^4 - (2x^3) + y^4 - 2y^3$$

For the 2D Rastrigin function, the implementation and input varied between the two algorithms. For Gradient Descent, the use of SymPy facilitated the computation of gradients for not just the Rastrigin function but also for other potential functions. On the contrary, SA did not directly employ SymPy for ongoing calculations; instead, the function was converted into a standard callable function using "sp. lambdify". This conversion was crucial as it allowed the function to be effectively used as the energy function in the annealing process.

4. Results and Discussion

4.1. Gradient Descent

During the experiment Gradient Descent showed variable convergence depending on the initial solutions with coordinates landing on the following:

$$[(1.9899, -1.9899), (1.9899, -0.9949), (7.9592, 2.9848), (0, 0), (0.000000339, 0.000000339), (0.9495, 0.00000046)]$$

An increase in maximum iterations was necessary to enhance results, highlighting its dependence on momentum and its gradient computation for better convergence. The runtime for Gradient Descent was also notably lengthy as each iteration required a recalculation of the gradient of the current position in the function.

When it came to the algorithm's behavior, Gradient Descent delivered consistent results across different runtimes. This was something that was predicted considering this algorithm is considered deterministic.

The initial conditions significantly influenced the algorithm's results as when the algorithm was initiated close to the global minimum, the algorithm achieved high precision in locating the minimum. In contrast, when the initial solution was positioned far from the absolute optimum, the results deviated substantially from the solution, which emphasizes the importance of a good initial guess.

4.2. Simulated Annealing

Simulated Annealing presented a robust performance, landing near the absolute minimum in various trial runtimes with results such as:

$$[(-0.0061305, -0.097170), (-0.004949, -0.02316837), (-0.027099, -0.03812), (0, 0), (0.01614, -0.00835), (0.014155, -0.008622)]$$

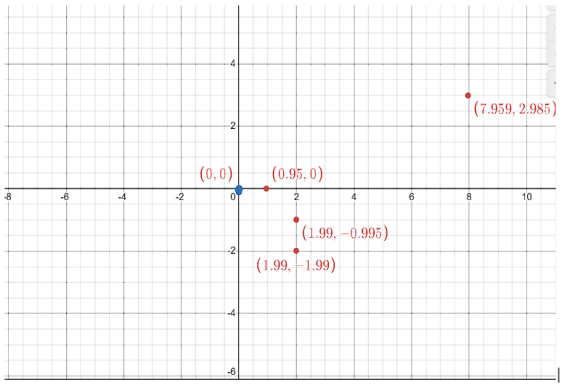

Figure 1. *Convergence paths of the Gradient Descent algorithm under varying initial conditions.*

It is important to note that the algorithm's probabilistic nature led to diverse outcomes, even with the same initial conditions. This still demonstrated SA's adaptability with different initial conditions.

Unlike Gradient Descent, Simulated Annealing completed runs within a shorter time when reaching high iterations. This efficiency makes it suitable for quick searches across potentially complex landscapes.

Although this algorithm is not as precise as Gradient Descent when starting near the solution, Simulated Annealing consistently approached the vicinity of the solution, illustrating its capacity to escape local minima effectively

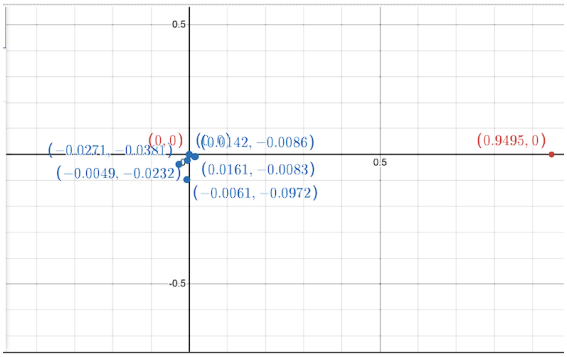

Figure 2. *Performance of Simulated Annealing across different trials.*

4.3. Comparative Analysis

The deterministic nature of Gradient Descent makes it suitable for applications where precision is critical and computational power is not a constraint. On the contrary, Simulated Annealing's probabilistic approach and quicker runtime enable this algorithm to be better suitable for problems with complex landscapes and where the solution is less predictable.

In essence, the choice between the two algorithms should consider the complexity of the function in question, the quality of available initial guesses, computational limitations, and the required precision of the solution.

5. Conclusion

In conclusion, the experiments revealed distinct characteristics and performance metrics for each algorithm. Gradient

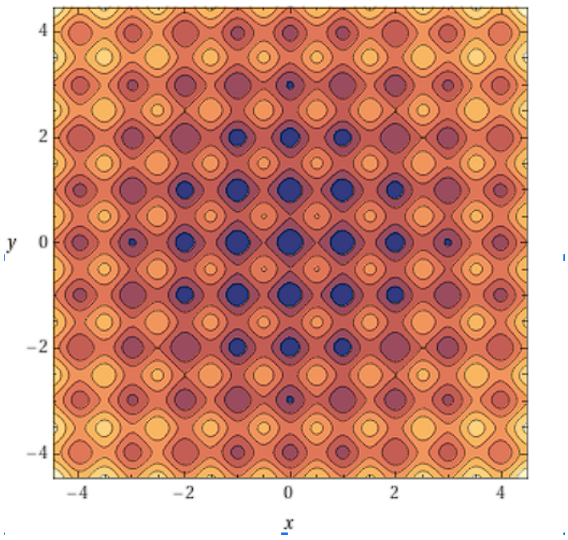

Figure 3. *Heatmap for Rastrigin Function. Use it as a reference to see where Figure 1 and Figure 2 values are located on the function*

Descent, enhanced by cosine-adjusted learning rate and momentum, demonstrated strong convergence when initiated near the global minimum. However, its performance significantly varied with initial conditions and required recalculations of gradients every iteration, making the algorithm computationally intensive. On the other hand, Simulated Annealing demonstrated robustness across various initial solutions due to its probabilistic acceptance and adaptive temperature schedule. This feature allowed SA to constantly and more effectively escape local minima than Gradient Descent. Although this statement is true, the algorithm did require fine-tuning of parameters such as cooling rate to optimize performance.

Finally, this comparative analysis is significant as it not only underscores the strengths and weaknesses of each algorithm but also guides others in selecting the appropriate optimization technique based on the specific problem. By understanding the operational dynamics and effectiveness of these algorithms that navigate complex landscapes, users can better tailor their approaches to achieve more accurate and efficient outcomes.

6. Reflection

Throughout this report, several insights have been gained regarding the comparative performance of Gradient Descent and Simulated Annealing. More specifically, I identify key areas where future modifications could be made to enhance and deepen my understanding of this analysis.

Firstly, the versions of the algorithms employed were selected to highlight their distinct operational mechanics under various conditions. While this choice provided detailed insights into more advanced functionalities, it consequently limited the scope of comparison against their simpler variants. In future studies, incorporating both advanced and basic versions of Gradient Descent and Simulated Annealing could offer a more layered understanding of how incremental enhancements impact algorithmic performance.

Lastly, the interaction with the audience during presentations was primarily informative rather than interactive. I should have had more engaging discussions that could have been fostered by posing direct questions about the audience's experiences and views regarding the optimization algorithms. For example, I could have asked a question regarding the deterministic nature of Gradient Descent compared to the probabilistic tendencies of Simulated Annealing. Such interactions could provide qualitative insights that enrich the data gathered, offering a more in-depth view of the algorithms' applicability in real-world scenarios.