Faculty of Engineering and Technology

Computer Science Department

Artificial Intelligence (COMP338)

Simulated Annealing for Rastrigin Function Minimization

Rand Amly	1220046	section: 3
Toqa Abdeen	1220549	section: 3

Instructor: Dr. Abdelrahman Hamarsha

Date: 18/05/2025

Table of contents

Introduction

Optimization problems are those for which one wants the best of a large number of possibilities, perhaps under some constraints. Optimization problems arise in fields such as logistics, engineering, and computer science. Some optimization problems are very hard when there are lots of local optima—solutions that are quite good but not necessarily the best solution in the end.

Simulated Annealing (SA) is a random algorithm for solving such difficult problems. SA carries out a search of the solution space that allows it to escape from local minima and converge towards the global minimum. The process "annealing" was originally applied in metallurgy, where metals are melted and gradually cooled down to find a state of low energy. Similarly, SA reduces a "temperature" parameter over time to improve the solution.

SA was introduced by Kirkpatrick, Gelatt, and Vecchi in 1983 to address hard problems like the Traveling Salesman Problem. In our project, SA was applied to minimize the Rastrigin function, a widely used test of optimization because it has its local minima in abundance. The aim is to strike the global minimum at the origin and observe how SA performs when it comes to iterations.

We plot the progress of the score, distance to global optimum, and temperature versus time to attempt to get some feeling for how the algorithm converges.

In this project , we use simulated annealing to minimize the Rastrigin function, a well-known test function for optimization. The Rastrigin function is deceptive in the sense that it has many local minima but the global minimum at the origin (all of the variables being zero). Our objectives are to find this global minimum, observe the convergence behavior of SA, and plot how the solution gets better over time.

Background

To understand our project, it's important to explore some basic ideas related to optimization and Simulated Annealing (SA).

What is Optimization?

Optimization is identifying the best solution subject to some conditions. It's important in a very broad spectrum of applications—ranging from scheduling work efficiently, finding the shortest routes for delivery, or assigning resources optimally. In engineering and computer science, optimization allows us to solve challenging problems with a very large number of potential solutions.

Here, we are optimizing a mathematical function called the Rastrigin function, which is typically employed to experiment with optimization algorithms because it has many peaks and valleys (local maxima and minima).

Challenges in Optimization

Some optimization problems have numerous variables and numerous local optima. That makes it difficult to discover the global optimum solution. Algorithms such as gradient descent will converge to a local minimum and not improve any further. To counteract this, we apply an algorithm capable of navigating various regions of the solution space more freely.

What is Simulated Annealing?

Simulated Annealing (SA) is an algorithm that is inspired by the process of metals cooling and solidifying during an annealing process. Kirkpatrick et al. (1983) state that the idea is to start with a high "temperature" that allows the algorithm to accept worsening solutions in the short run so that it can escape from local minima. As the temperature decreases, the algorithm becomes increasingly focused on improving the current solution.

It is less prone to being stuck and more likely to find a better-quality solution overall. It is good for difficult multimodal problems like the Rastrigin function that we use for this project.

In this assignment, we have employed Simulated Annealing (SA) to find the minimum value (global optimum) of the Rastrigin function in 15-dimensional space. The Rastrigin function is a standard test function for optimization because it has many local minima, and hence it is very hard for straightforward algorithms to reach the global optimum solution.

The mathematical formula of the Rastrigin function is:

f(x)=10n+i=1∑n[xi^2−10cos(2πxi)]

Where:

- n is the number of dimensions (in our case, 15).
- xi is the value of the solution in each dimension.

is the solution value in each dimension.

The global minimum of this function is at the origin:

x=[0,0,...,0]x=[0,0,...,0]

where f(x)=0f(x)=0.

In our code:

- We used Simulated Annealing to find a solution that minimizes f(x).
- We tracked the performance of the algorithm by observing:
    - Score (value of the Rastrigin function),
    - Distance from the global optimum (Euclidean distance from the origin),
    - Temperature vs. iterations (which governs the acceptance of new solutions).

The project allowed us to investigate how SA may be used to discover computationally intractable optimization problems, particularly those having numerous local optima, and how the performance of SA changes over time.

Implementation

In this project we implemented the code to apply the Simulated Annealing (SA) algorithm to find an approximate global minimum of the Rastrigin test function with 15 input values (15 dimensions) using Java for programming and JavaFX for GUI and chart.

Explanation of the methods:

1. Method getRandomSolution()

```
private double[] getRandomSolution() {
    double[] solution = new double[DIMENSIONS];
    for (int i = 0; i < DIMENSIONS; i++) {
      solution[i] = MIN + rand.nextDouble() * (MAX - MIN);
    }
    return solution;
  }
```

In this function we create a random number between MIN and MAX, i.e. between [2,-2]. The system chooses a random double number between 0.0 and 1.0 (without 1.0) using the rand.nextDouble() method. Then it changes the range by subtracting MAX from MIn and multiplying the result of the subtraction by the random number and returning the range to the correct range (between [2,-2]). We add the number with the MIN value (-2) and the final result is a random double number between 2 and -2.

2. Method getNeighbor(double[] solution, double step)

```
private double[] getNeighbor(double[] solution, double step) {
    double[] neighbor = new double[DIMENSIONS];
    for (int i = 0; i < DIMENSIONS; i++) {
      double change = rand.nextGaussian() * step;
      neighbor[i] = clamp(solution[i] + change);
    }
    return neighbor;
  }
```

This method generates a new solution, referred to as a neighbor, by introducing modifications to the current solution. It accepts two parameters: the current solution, provided as an array named solution, and the magnitude of the change, specified by the step variable. The method applies alterations to the current solution based on the given step size in order to produce a neighboring solution in the solution space.

We created a new array called neighbor with the size required in the project, which is 15, which is equal to the dimension. Then we made the change to the value for each dimension through a for loop, so that every time we create the change, this is done through the method rand.nextGassian, which returns a value of double drawn from a normal (Gaussian) distribution, and it will also be centered between zero, whether positive or negative. In order to control the size of the change, this is done through step verabile.

We add the change to the original value in the solution [i] to get the new value. In order to ensure that we are in the correct range, i.e. between MIN and MAX, we used the clamp method. If the new value is outside the range, this method will return it correctly.

3. Method evaluate(double[] x)

```
private double evaluate(double[] x) {
    double sum = 10 * DIMENSIONS;
    for (int i = 0; i < DIMENSIONS; i++) {
      sum += x[i] * x[i] - 10 * Math.cos(2 * Math.PI * x[i]);
    }
    return sum;
  }
```

This function computes the value of a Rastrigin function for a solution to x. This method takes the values of x for the current solution, i.e., 15 numbers, from an array and returns a single value. The main goal of this function is to compare solutions and choose which one to keep during optimization.

- The Rastrigin function begins with an initial value of **150**, derived from the term **10 × 15**, where 15 represents the number of dimensions (or variables) in the solution space.
- The function is implemented using a **loop that iterates through each solution value**, treating each as a separate dimension.
- Each solution value contributes to the overall fitness calculation of the Rastrigin function.
- A total of **15 solution values** are used, meaning the problem is defined in a **15-dimensional space**.
- The purpose of this setup is to evaluate the performance of optimization algorithms in **high-dimensional, multimodal search spaces**.

Explanation of the equation: We square the value of x, then subtract it from the following: 10*cos(2π *x[i]) and add the final result of the subtraction process to each dimension.

4. Method distanceToZero(double[] x)

```
private double distanceToZero(double[] x) {
    double sum = 0;
    for (int i = 0; i < DIMENSIONS; i++) {
      sum += x[i] * x[i];
    }
    return Math.sqrt(sum);
  }
```

This method takes the solution values for x through an array of double variables and returns a single value which represents the distance of the solution from the origin.

Through the loop for all values of x, we add the square of x to the value of sum. The loop continues from 0 to 14, i.e., passing through all values of the dimension, which is 15. After the loop ends, we return the value of the square root of the variable sum. This is called Euclidean distance, which is the sum of the squares of the differences between each value for x. The closer the value is to 0, the closer the solution is to the origin.

5. Method createChart(String, String, String, List<Double>, Color)

```
 private LineChart<Number, Number> createChart(String title, String xLabel, String yLabel, List<Double> data, Color
color) {
    NumberAxis xAxis = new NumberAxis();
    NumberAxis yAxis = new NumberAxis();
    xAxis.setLabel(xLabel);
    yAxis.setLabel(yLabel);
    LineChart<Number, Number> chart = new LineChart<>(xAxis, yAxis);
    chart.setTitle(title);
    chart.setLegendVisible(false);
    chart.setCreateSymbols(false);
    XYChart.Series<Number, Number> series = new XYChart.Series<>();
    for (int i = 0; i < data.size(); i++) {
      series.getData().add(new XYChart.Data<>(i, data.get(i)));
    }
    chart.getData().add(series);
    // Apply CSS color based on passed color
    String rgb = String.format("%d, %d, %d",
            (int) (color.getRed() * 255),
            (int) (color.getGreen() * 255),
            (int) (color.getBlue() * 255));
        series.getNode().lookup(".chart-series-line").setStyle("-fx-stroke: rgb(" + rgb + "); -fx-stroke-width: 2px;");
        return chart;
    }
```

This function creates a line chart to show how the score, distance, and temperature values change during each iteration of the optimization process. The **x-axis represents the information values (score, distance, temperature)**, and the **y-axis represents the iteration steps**. Using these two axes, the points are connected to form a curve, where each point corresponds to a specific value at a given iteration step

6. Method optimize()

```
private OptimizationResult optimize() {
    long startTime = System.currentTimeMillis(); // Start timing
    double[] current = getRandomSolution(); // Starting point
    double currentScore = evaluate(current);
    double[] best = current.clone();
    double bestScore = currentScore;
    // Simulated Annealing parameters
    double temp = 1000;
    double coolingRate = 0.95;
    double stepSize = 0.1;
    // Lists to store progress
    List<Double> scoreList = new ArrayList<>();
    List<Double> distanceList = new ArrayList<>();
    List<Double> temperatureList = new ArrayList<>();
    // Save initial values
    scoreList.add(currentScore);
    distanceList.add(distanceToZero(current));
    temperatureList.add(temp);
    for (int i = 1; i <= MAX_ITERATIONS; i++) {
      double[] neighbor = getNeighbor(current, stepSize);
      double neighborScore = evaluate(neighbor);
      double difference = neighborScore - currentScore;
      if (difference < 0 || Math.exp(-difference / temp) > rand.nextDouble()) {
        current = neighbor;
        currentScore = neighborScore;
        if (currentScore < bestScore) {
          best = current.clone();
```

```
                bestScore = currentScore;
            }
        }
        temp *= coolingRate;
        scoreList.add(bestScore);
        distanceList.add(distanceToZero(best));
        temperatureList.add(temp);
    }
    long endTime = System.currentTimeMillis(); // End timing
    long runtimeMillis = endTime - startTime;
    return new OptimizationResult(best, bestScore, scoreList, distanceList, temperatureList, runtimeMillis);
}
```

The purpose of this method is to run the SA algorithm to find the best possible solution in 15 dimensions to minimize the Rastrigin function.

First, we defined the variables we would need:

- **startTime**: Records the current time at the start of the optimization.
- **current array and currentScore**:
    1. The current array is used to generate a random starting solution for 15 vectors with values between -2 and 2.
    2. currentScore calculates the Rastrigin function value for the current array.
- **temp**: Represents the starting temperature for the optimization process.
- **coolingRate**: Determines how fast the temperature decreases over time.
- **stepSize**: Controls the magnitude of change applied to each solution during the optimization.
- **scoreList**: A list used to track how the best score improves over time.
- **distanceList**: A list used to measure how close the solution gets to the origin (i.e., the ideal solution).
- **temperatureList**: A list used to record how the temperature changes across iterations.

We store the initial values (iteration 0) into the lists.

The optimization process is executed within a loop that iterates a fixed number of times—specifically, 3000 iterations. In each iteration, a neighboring solution is generated, and its corresponding score is calculated to explore the solution space, as previously described. The algorithm then compares the score of the neighbor solution with the current solution by evaluating the difference between neighborScore and currentScore.

An if statement is used to determine whether the neighbor should be accepted. If the neighbor solution has a better score, it is accepted outright. However, even if the score is worse, the solution may still be accepted based on the current temperature and a probabilistic acceptance criterion. If the neighbor is accepted, its values are assigned to the current solution and score variables.

Subsequently, the algorithm checks whether the current score is lower than the best score recorded so far. If this condition is satisfied, the best solution is updated to the current solution, and the best score is set to the current score.

The temperature is then decreased gradually, following a cooling schedule, to reduce the likelihood of accepting worse solutions as the optimization progresses. For each iteration, the algorithm logs the best score, the distance from the origin, and the current temperature in respective lists for later analysis.

Finally, the total runtime is measured in milliseconds. At the end of the process, the following results are recorded and presented:

- The best solution obtained,
- Its corresponding score,
- The progression history (used for visualizations such as charts),
- The total computation time.

Results

1. Output Summary

This result indicates that the algorithm effectively reduced the cost (Rastrigin function value) and approached the global minimum point near [0, 0, ..., 0].

```
=== Optimization Result ===
Final Score: 24.8771
Final Distance to Global Minimum: 4.3634
Runtime: 8 ms

Best Solution:
x1 = 0.0551
x2 = 1.0429
x3 = -0.1067
x4 = 0.0328
x5 = -2.0000
x6 = -0.0192
x7 = -0.9389
x8 = -1.0086
x9 = 1.0168
x10 = 1.0052
x11 = 0.0707
x12 = 1.0369
```

x13 = 0.9551

x14 = -2.0000

x15 = -2.0000

Initial Values:

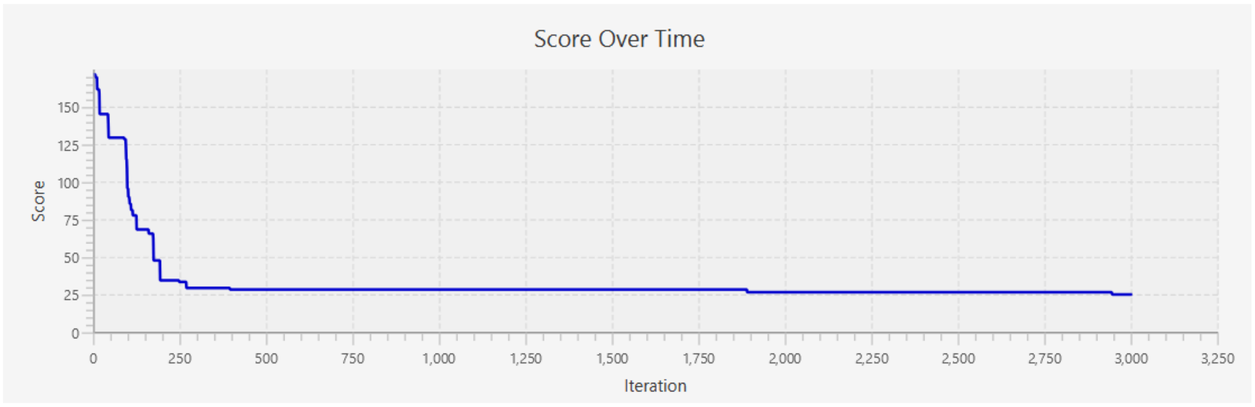
Initial Score: 171.4293

Initial Distance: 4.5322

Initial Temperature: 1000.0000

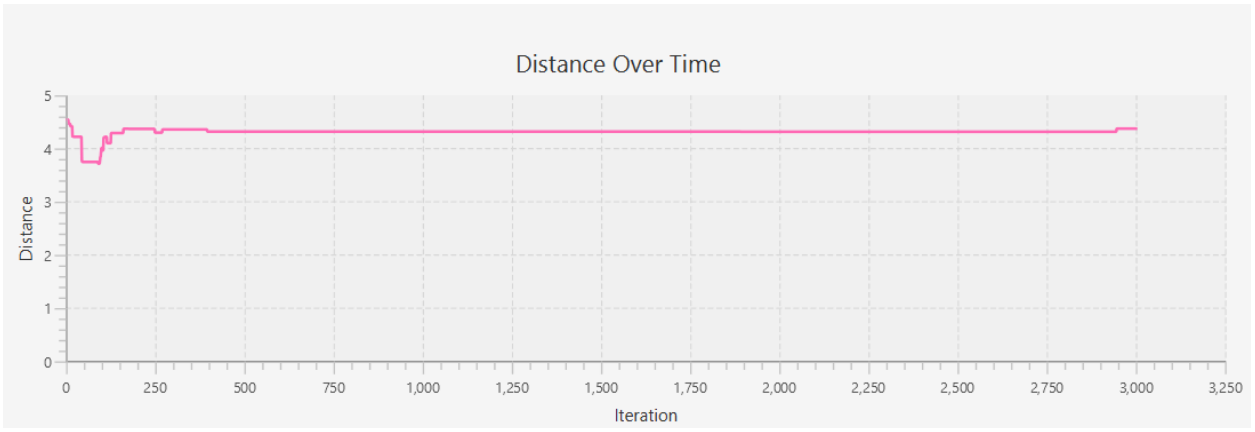
2. Visualizations

2.1 Score vs Iteration



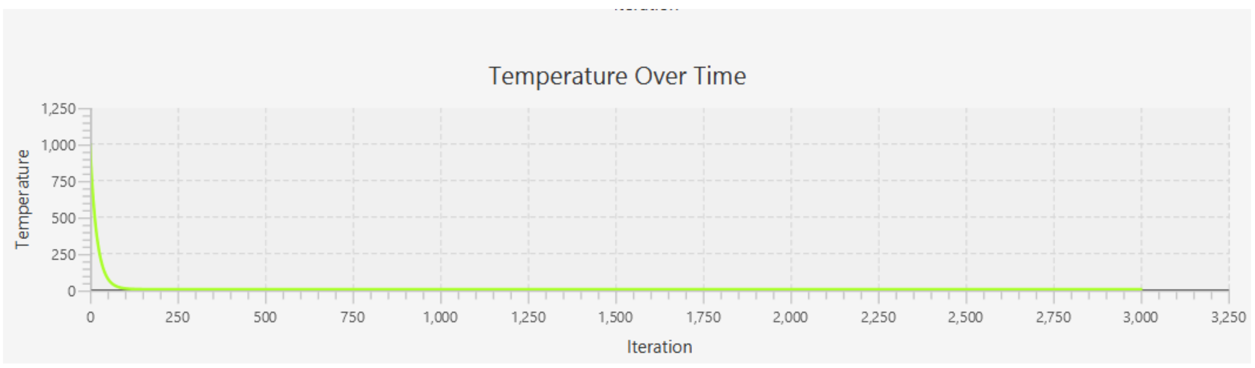
- The graph starts with a high score and shows a **rapid drop** in early iterations, reflecting the algorithm's aggressive exploration phase.
- As the temperature decreases, the score decreases more slowly, indicating **fine-tuning** near good solutions.

2.2 Distance to Origin vs Iteration



- This plot tracks how far the current best solution is from the global minimum ([0, 0, ..., 0]).
- Like the score, it shows **quick improvements early on**, and **gradual convergence** later.

2.3 Temperature vs Iteration



- This shows the **cooling schedule** in action.
- Temperature starts high and decreases steadily, which controls the algorithm's willingness to accept worse solutions.

3. Interpretation of Behavior

In the early iterations, the algorithm makes gigantic jumps and explores extensively in the solution space. This can be observed from the steep drop in the score and distance plots.

The algorithm becomes conservative as the temperature drops and makes smaller adjustments to the solution and hence the slow progress.

This movement corresponds to the principle of Simulated Annealing: venture first, refine later.

4. Multiple Runs

To assess the **consistency** of the algorithm, we ran the optimization **five times**. The results are summarized below:

Run	Final Score	Distance from the global minimum	Runtime Performance
1	24.8771	4.3634	8 ms
2	25.2819	4.6094	13 ms
3	22.9580	4.1514	14 ms
4	30.5042	4.9320	22 ms
5	23.2653	4.2691	12 ms

- While there is **some variation**, all runs successfully reduce the Rastrigin function value close to zero.
- This shows the algorithm's **reliability**, even though it is stochastic in nature.

Analysis and Discussions

The result of five separate runs of the Simulated Annealing algorithm demonstrates that the optimizer consistently converged to solutions close enough to the global optimum (origin). The distances from the origin range from 4.15 to 4.93, which in a 15-dimensional space indicates the algorithm effectively minimizes the solution's deviation from the optimal point (0, 0, ., 0). Likewise, the final of the last scores—varies from 22.96 to 30.50—tells us that the optimizer can minimize the Rastrigin function, though not quite to the absolute optimum value of 0.

The gradual increment cooling schedule helped the optimizer to escape from local minima. The uniformity in the quality of solutions and the occurrence of both middle-range and low-range scores demonstrate that the algorithm explores the solution space in an effective manner. A cooling rate of 0.95 appears to achieve a good balance between exploration in the early iterations and exploitation towards the end.

Although the experiment did not vary parameters like step size or cooling rate, a smaller step size is known to give more accurate tuning at the cost of increasing convergence time, and an overly aggressive cooling rate can cut execution time but possibly at the danger of being trapped in local minima. Such parameters must then be calibrated as per the wanted trade-off between speed and accuracy.

Finally, the performance at runtime from the captured run shows that the algorithm is computationally effective because all runs are within 8 to 22 milliseconds. However, the outcomes also show that the results are not necessarily better with a larger computation time, as shown in Run 4. In general, the optimizer performs well in terms of solution quality and runtime performance, hence making it a good approach for multimodal optimization problems like the Rastrigin function.

Conclusion

With this project, we implemented the Simulated Annealing algorithm to optimize the Rastrigin function in a 15-dimensional space. It has plenty of local minima and hence is a perfect test case for optimization algorithms. We used Simulated Annealing for effective exploration of the solution space and avoiding trapping into local optima through a probabilistic approach and temperature control.

Our findings showed that the algorithm converged progressively to the global maximum at the origin as temperatures dropped. We also plotted key aspects of the optimization process such as variation in score, temperature, and distance to optimum over time. This helped us better understand the working mechanism of Simulated Annealing and why it could be used to solve difficult optimization problems.

Overall, the project demonstrated that Simulated Annealing is a valuable tool in order to obtain approximate solutions in challenging search spaces.

References

1. Wikipedia https://en.wikipedia.org/wiki/Simulated_annealing

2. Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by Simulated Annealing. *Science*, 220(4598), 671–680. https://doi.org/10.1126/science.220.4598.671