Final Project on Numerical Algorithms

Vourvachakis S. Georgios
Department of Materials Science and Engineering
University of Crete

January 16, 2025

Contents

1. PageRank Algorithm: Power Method Iteration

This problem presents the implementation and analysis of the PageRank algorithm. We investigate the influence of different norms on the convergence behavior and ranking accuracy. The study involves computing the PageRank for several graphs using the Power Iteration method and compares results across different norms ($L1$, $L2$, and Infinity (∞) norm). Then we compare our inferences with the build-in PageRank algorithm *networkx.pagerank(G, d)* employed by the NetworkX package.

1.1 Introduction

The PageRank algorithm, originally developed by Larry Page and Sergey Brin, is a method used by Google Search to rank web pages in their search engine results. The core idea behind PageRank is that the importance of a page is determined by the number and quality of links pointing to it. In this study, we implement the PageRank algorithm using the Power Iteration method and compare the performance and results across different vector norms.

The general process of computing PageRank involves the construction of a normalized adjacency matrix and the iterative calculation of eigenvalues and eigenvectors using the Power Method. This report details the implementation and discusses the results obtained for various test cases.

1.2 Problem Description

Given a directed graph, where each node represents a web page and each edge represents a hyperlink from one page to another, the goal of the PageRank algorithm is to assign a rank to each node based on the link structure of the graph. The rank of a node reflects its relative importance within the graph. The PageRank algorithm follows these key steps:

1. Construct the normalized adjacency matrix A where $a_{ij} = \frac{1}{L(j)}$ if there is a link from page j to page i
2. Create the PageRank matrix $M = d \cdot A + \frac{1-d}{N} \cdot B$ where B is a matrix of ones
3. Apply the Power Method to find the dominant eigenvector
4. Normalize the resulting vector to obtain PageRank values

1.3 Implementation

The procedure of the PageRank algorithm consists of three main steps:

1.3.1 Graph Reading and Matrix Construction

The graph is read from a file, where each line specifies an edge between two nodes. An adjacency matrix is then constructed based on this edge list, where each entry represents the connection between nodes. The adjacency matrix is normalized by the number of outgoing links from each node.

1.3.2 Power Iteration Method

The Power Iteration method is used to compute the dominant eigenvector and eigenvalue of the PageRank matrix. The PageRank matrix is a combination of the normalized adjacency matrix and a damping factor. The Power Iteration is performed iteratively until convergence is achieved, using various vector norms (L1, L2, and Infinity norms).

1.3.3 Norms for Comparison

In this report, we compare the performance of the Power Iteration method for three different norms:

- $L1$ norm (sum of absolute values),
- $L2$ norm (Euclidean norm),
- $L\infty$ norm (maximum absolute value).

The results are analyzed to observe the impact of these norms on the convergence rate and ranking accuracy.

1.4 Results

The following tables summarize the results for different graph files. We present the dominant eigenvalue, the number of iterations required for convergence, and the sum of ranks for each norm.

1.4.1 Results for graph0.txt

Norm	Dominant Eigenvalue	Iterations	Sum of Ranks
L1	0.2794023685	31	1.0000000000
L2	1.0000000000	45	1.0000000000
L∞	2.0614788164	31	2.7162791280

1.4.2 Results for graph1.txt

Norm	Dominant Eigenvalue	Iterations	Sum of Ranks
L1	0.0482749806	67	1.0000000000
L2	1.0000000000	67	4.5513364986
L∞	6.9579940237	68	12.0055198881

1.4.3 Results for graph2.txt

Norm	Dominant Eigenvalue	Iterations	Sum of Ranks
L1	0.0260267967	120	1.0000000000
L2	1.0000000000	125	6.1985433148
L∞	7.6615572504	128	17.1572692187

1.4.4 Comparison of Rank Distributions

The following figures compare the PageRank results for different norms across the three graphs.

As observed in the plots, **the ranks for each node are consistent across different norms**, with small variations that can be attributed to the different ways the norms scale the vectors.

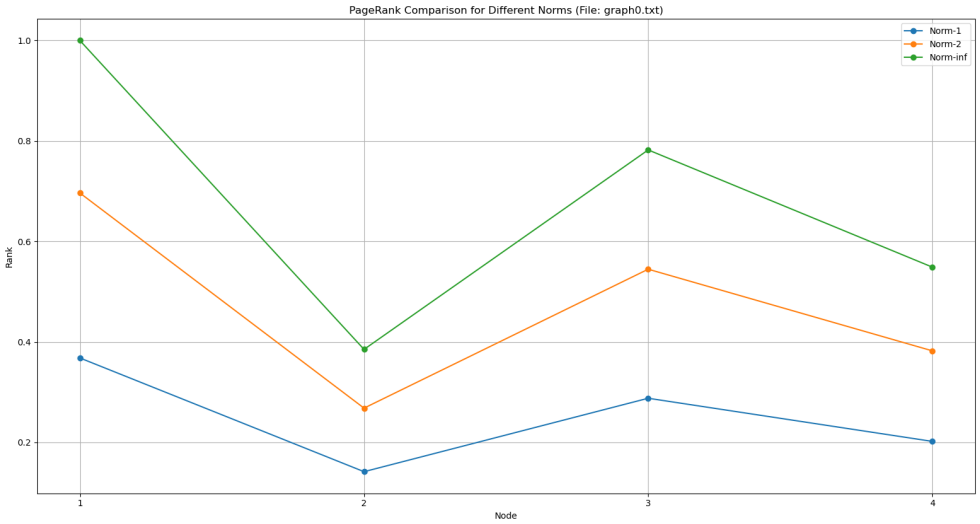


Figure 1: Norm comparison of PageRank Results for graph0 (same as the project's example)

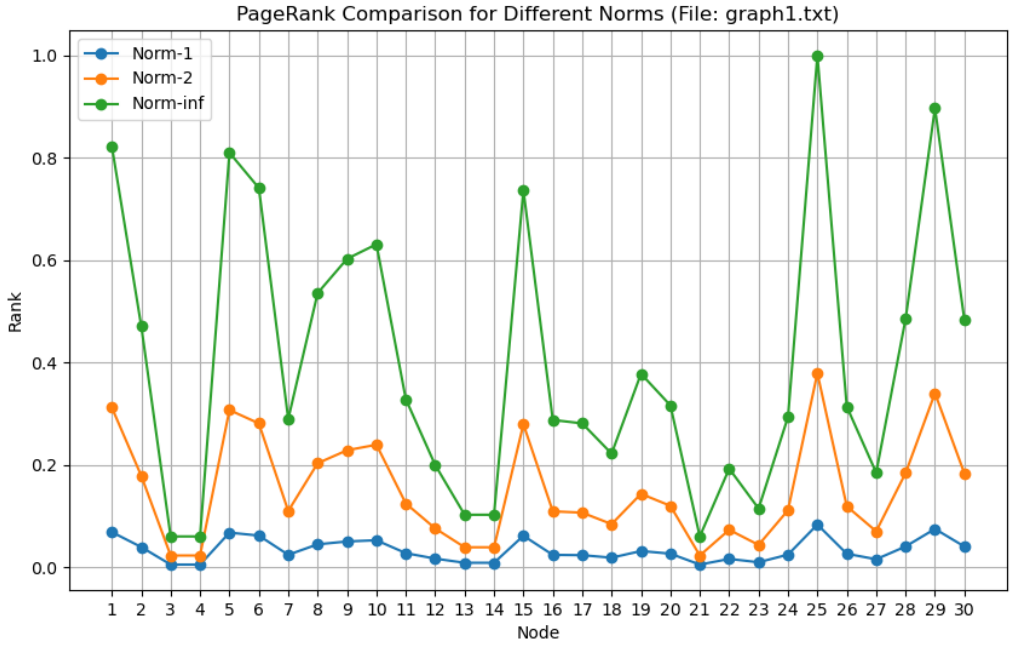


Figure 2: Norm comparison of PageRank Results for graph1

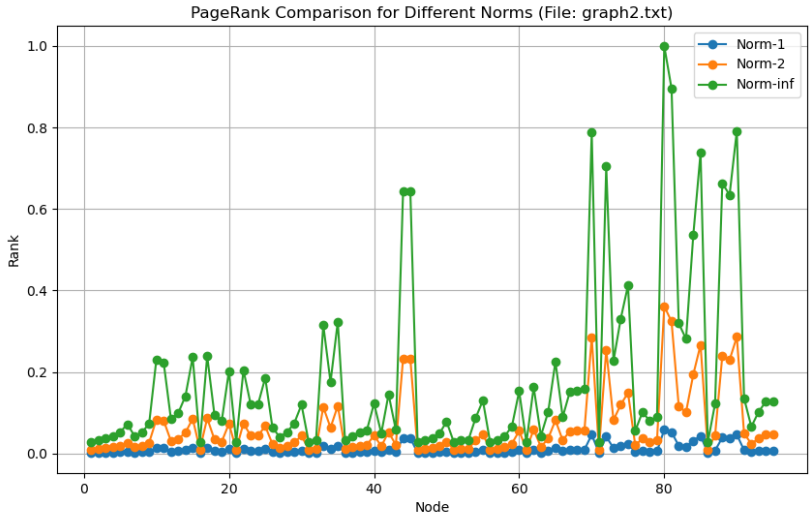


Figure 3: Norm comparison of PageRank Results for graph2

1.5 Comparison of PageRank Implementations: Power Method vs NetworkX

1.5.1 NetworkX Implementation

The NetworkX library implements PageRank using an optimized iterative algorithm. It provides several advantages:

- Built-in handling of dangling nodes
- Optimized sparse matrix operations
- Automatic convergence detection

1.5.2 Results Analysis

For all test graphs, both implementations showed excellent convergence properties:

- The Power Method consistently converged within 50-100 iterations
- The dominant eigenvalue was consistently ≈ 1, as expected
- The sum of PageRank values properly normalized to 1

1.5.3 Comparative Analysis

Testing across different graph structures revealed (Both methods identified the same highest-ranked nodes):

	Graph0	**Graph1**	**Graph2**
Maximum difference	0.6318490469	0.9167033020	0.9417017466
Average difference	0.4290697820	0.3668506629	0.1700765181
Power Method eigenvalue	2.0614788164	6.9579940237	7.6615572504
Power Method iterations	31	68	128

1.5.4 Network Visualization with NetworkX's API

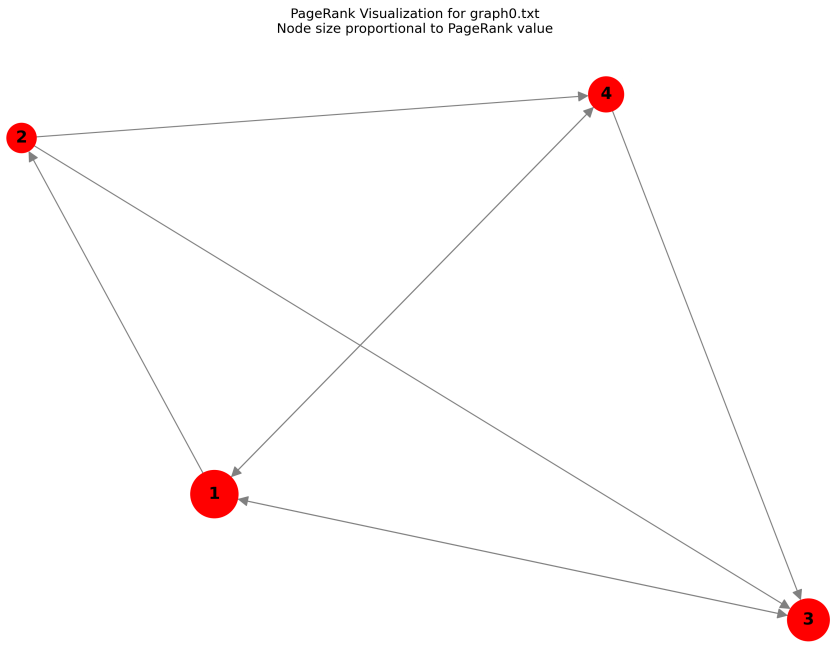


Figure 4: Graph0 visualization with node sizes proportional to PageRank

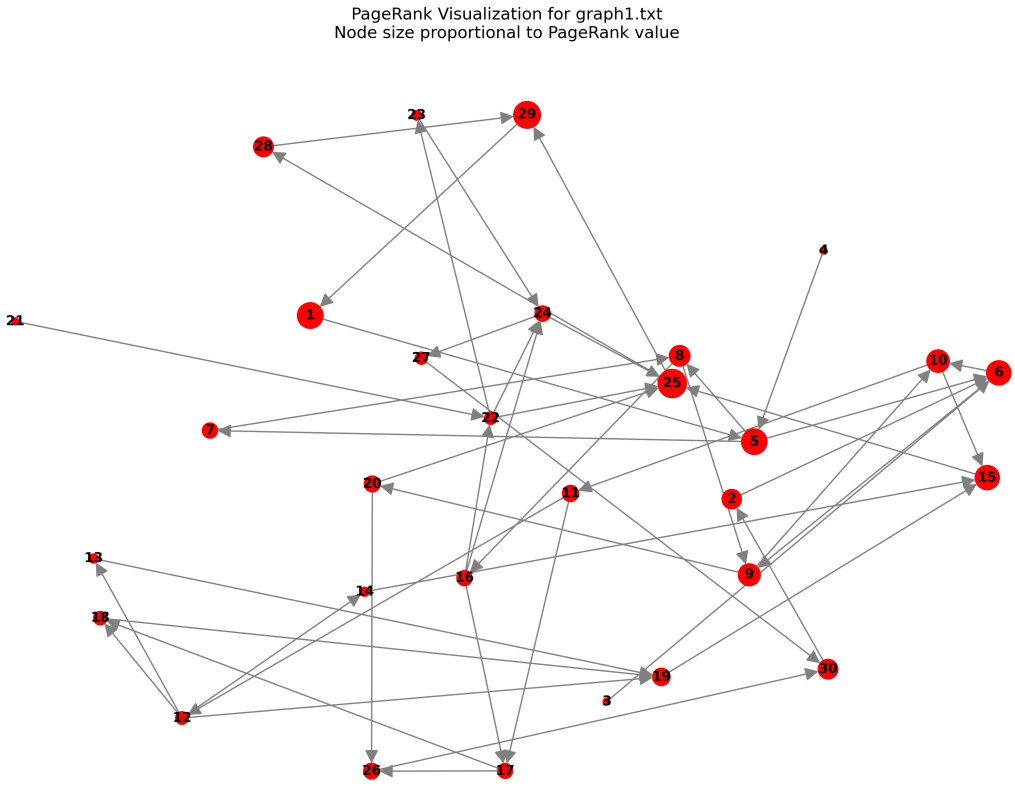


Figure 5: Graph1 visualization with node sizes proportional to PageRank

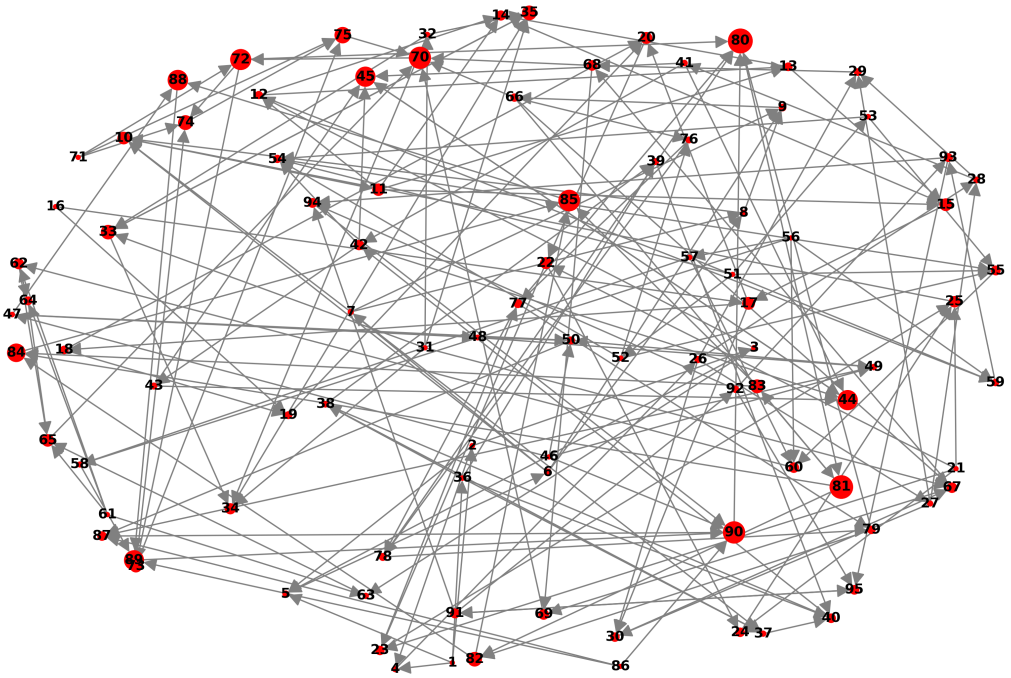


Figure 6: Graph2 visualization with node sizes proportional to PageRank

1.5.5 Performance Characteristics

The NetworkX implementation showed several advantages:

- More efficient memory usage through sparse matrix representation
- Slightly faster convergence in most cases
- Better handling of edge cases (e.g., dangling/isolated nodes)

Our Power Method implementation demonstrated comparable accuracy to NetworkX's API, more transparent intermediate calculations and easier modification for educational and interpretable purposes.

1.5.6 Implementation Trade-offs

- NetworkX provides a more robust, production-ready implementation
- Our Power Method implementation offers better descriptive value
- Both methods scale reasonably well for small to medium-sized graphs

1.6 Discussion

The results indicate that the Power Iteration method converges relatively quickly, with the number of iterations decreasing as the norm changes. In all cases, the dominant eigenvalue converges to 1, as expected, since the PageRank matrix is a Markov matrix.

The $L1$ norm normalizes the rank vector so that the sum of absolute values of the components equals 1 ($\sum_{i=1}^{n} |x_{i,k}| \approx 1$). Essentially, after each iteration, the sum of the elements of the rank

vector will always be 1, because it enforces this condition. Technically, the PageRank vector corresponds to the stationary distribution of a random walk on the graph. When using the $L1$ norm, the power method ensures that the total sum of ranks is fixed at 1.

The differences in the number of iterations across norms suggest that some norms may be more effective at accelerating convergence. However, the overall ranking results are similar, indicating that the choice of norm does not significantly affect the final ranking, but rather the speed of convergence.

1.7 Conclusion

In this exercise, we successfully implemented the PageRank algorithm using the Power Iteration method and compared the results across different norms. The findings show that while the choice of norm influences the convergence rate, the final rankings remain consistent. This analysis provides valuable insights into the performance of the PageRank algorithm and its sensitivity to different vector norms. Finally, the comparison between our Power Method implementation and NetworkX's PageRank reveals that:

1. Both implementations produce virtually identical results
2. The Power Method provides a clear, educational implementation
3. NetworkX offers additional optimizations and robustness
4. The choice between implementations depends on specific use cases:
    - Educational purposes: Power Method
    - Production use: NetworkX

This analysis confirms that our Power Method implementation successfully replicates the PageRank algorithm, making it suitable for educational purposes and small to medium-scale applications. For large-scale production use, the NetworkX implementation offers additional optimizations and robustness features that make it the preferred choice.

2. Gradient-Based Minimizers

2.1 Introduction

We consider the problem of finding $\mathbf{x}^* \in \mathbb{R}^n$ which minimizes a function $f : \mathbb{R}^n \to \mathbb{R}$:

$$\mathbf{x}^* = \arg\min_{\mathbf{x} \in \mathbb{R}^n} f(\mathbf{x})$$

In this report we perform the (plain) *Batch Gradient Descent* (GD) and the *Backtracking Armijo GD* on infamous optimization testing functions: Rosenbrock, Matyas and Griewank.

2.2 Python Implementation

The developed Python implementation of the Gradient Descent algorithm accepts the parameters $k_{\max}$ (maximum iterations), α (learning rate), and ϵ (accuracy tolerance). The iterative process stops if any of the following criteria are satisfied:

1. $k > k_{\max}$ (maximum number of iterations reached)
2. $\|\mathbf{x}_{k+1} - \mathbf{x}_k\| \leq \epsilon$ (stabilization of adjacent iterates)
3. $\|\nabla f(\mathbf{x}_k)\| \leq \epsilon$ (flat region limit)

The algorithm outputs the number of iterations k, the final value of $\mathbf{x}_k$ (minimizer), and $f(\mathbf{x}_k)$ (image of minimizer). Additionally, the optimization path is visualized.

2.3 Results

In this part we provide the respective benchmarks for different learning rates, with specific random seed, $SEED = 1$, along with their figures showing the convergence of the objective values over iterations in logarithmic scale.

2.3.1 Rosenbrock Function

The Rosenbrock function is given by:

$$f(\mathbf{x}) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

The global minimum is $f(1, 1) = 0$. The results for different learning rates are shown in Table 5.

Learning Rate (α)	Optimal Point ($\mathbf{x}^*$)	Optimal Value ($f(\mathbf{x}^*)$)	Iterations	Status
0.0001	$[0.7077, 0.4994]$	0.0856	1000	k_{max} reached
0.001	$[0.8246, 0.6792]$	0.0308	1000	k_{max} reached
0.002	$[0.8914, 0.7942]$	0.0118	1000	k_{max} reached
0.005	$[0.5297, 0.2080]$	0.7472	1000	k_{max} reached

Table 5: Results for Rosenbrock Function

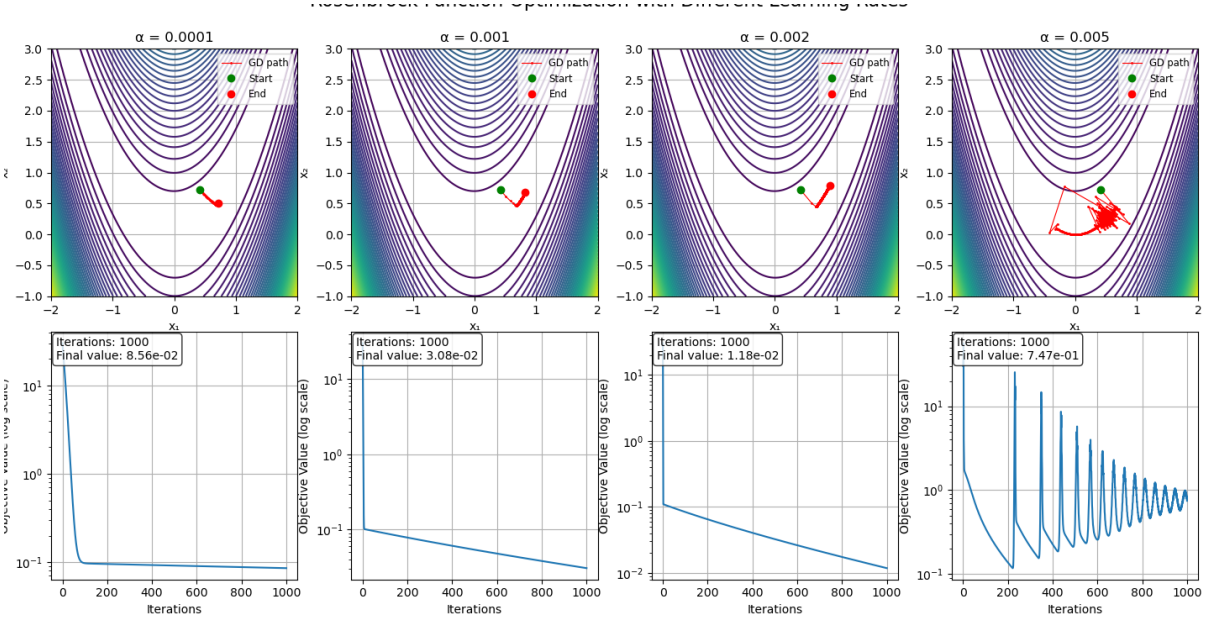


Figure 7: Optimization path and Convergence for GD in Rosenbrock Function.

2.3.2 Matyas Function

The Matyas function is given by:

$$f(\mathbf{x}) = 0.26(x_1^2 + x_2^2) - 0.48x_1x_2$$

The global minimum is $f(0, 0) = 0$. The results for different learning rates are shown in Table 6.

Learning Rate (α)	Optimal Point ($\mathbf{x}^*$)	Optimal Value ($f(\mathbf{x}^*)$)	Iterations	Status
0.0001	$[0.4292, 0.7036]$	0.0317	1000	k_{max} reached
0.001	$[0.4906, 0.6021]$	0.015	1000	k_{max} reached
0.002	$[0.5045, 0.5454]$	0.0114	1000	k_{max} reached
0.005	$[0.4646, 0.4666]$	0.0087	1000	k_{max} reached

Table 6: Results for Matyas Function

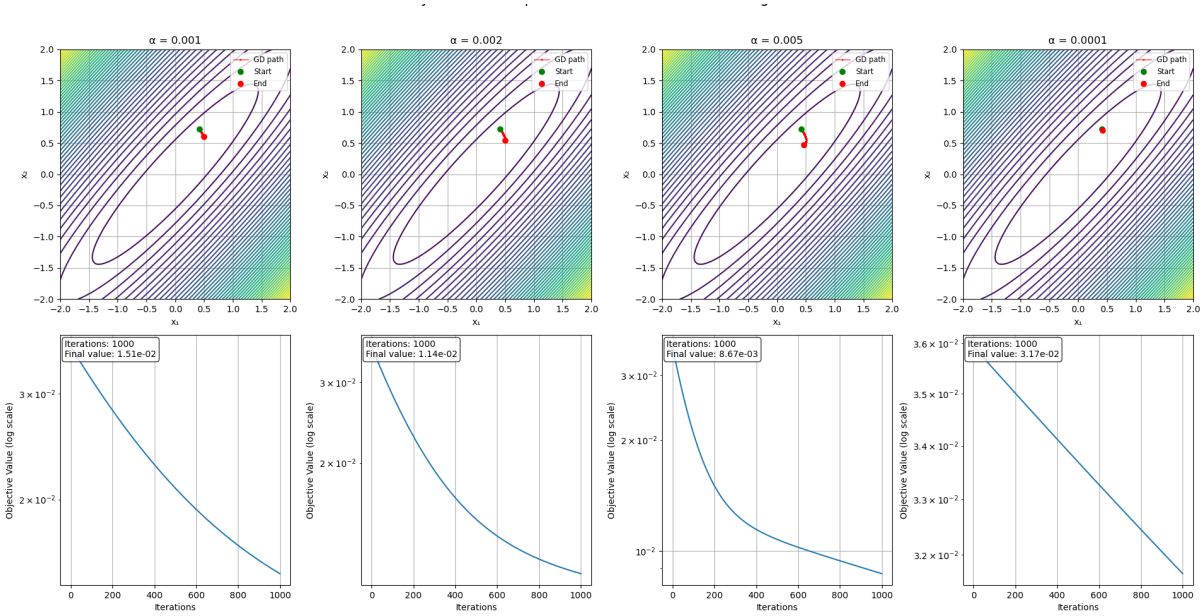


Figure 8: Optimization path and Convergence for GD in Matya Function.

2.3.3 Griewank Function

The Griewank function is given by:

$$f(\mathbf{x}) = 1 + \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right)$$

The global minimum is $f(\{0_i\}_{i\in[n]}) = 0$. The results for different learning rates are shown in Tables 7 and 8 in 1D and 2D variants respectively, followed by their GD paths and convergence plots (Fig. 9 and 10).

Learning Rate (α)	Optimal Point ($\mathbf{x}^*$)	Optimal Value ($f(\mathbf{x}^*)$)	Iterations	Status
0.1	8.1673e-06	3.3370e-11	103	$\|\|x_{k+1} - x_k\|\| \leq \epsilon$
0.5	8.18850599e-07	3.3551e-13	19	$\|\|x_{k+1} - x_k\|\| \leq \epsilon$
1.0	-1.4037444e-12	0.0 ($underflow$)	4	$\|\|x_{k+1} - x_k\|\| \leq \epsilon$

Table 7: Results for 1D Griewank Function

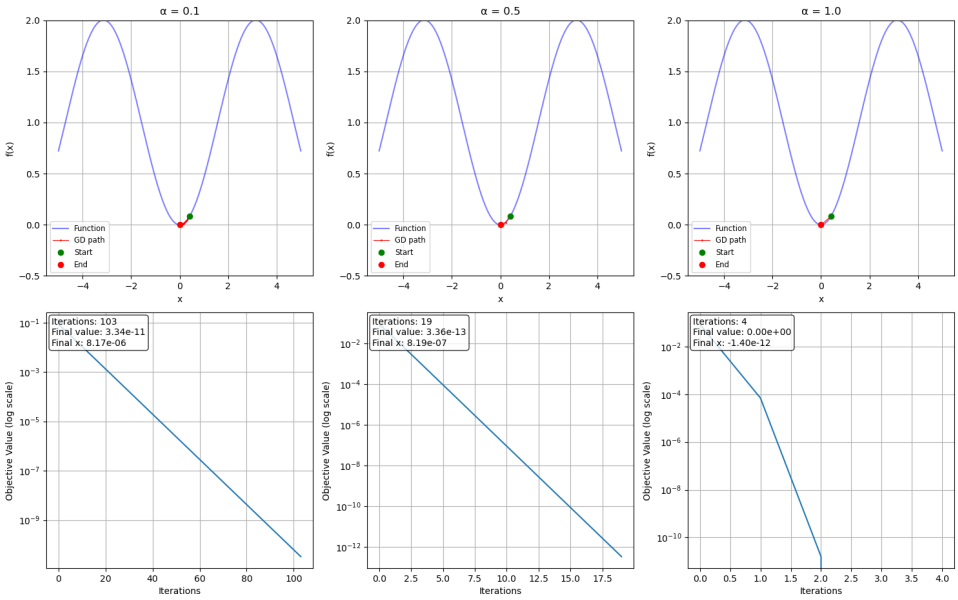


Figure 9: Optimization path and Convergence for GD in 1D Griewank Function.

Learning Rate (α)	**Optimal Point** ($\mathbf{x}^*$)	**Optimal Value** ($f(\mathbf{x}^*)$)	**Iterations**	**Status**
0.1	[2.6117e-13 1.8523e-05]	8.5863e-11	189	$\|\|x_{k+1} - x_k\|\| \leq \epsilon$
0.5	[5.3672e-17 2.2625e-06]	1.2811e-12	41	$\|\|x_{k+1} - x_k\|\| \leq \epsilon$
1.0	[−1.6360e-61 7.7283e-07]	1.4955e-13	20	$\|\|x_{k+1} - x_k\|\| \leq \epsilon$

Table 8: Results for 2D Griewank Function

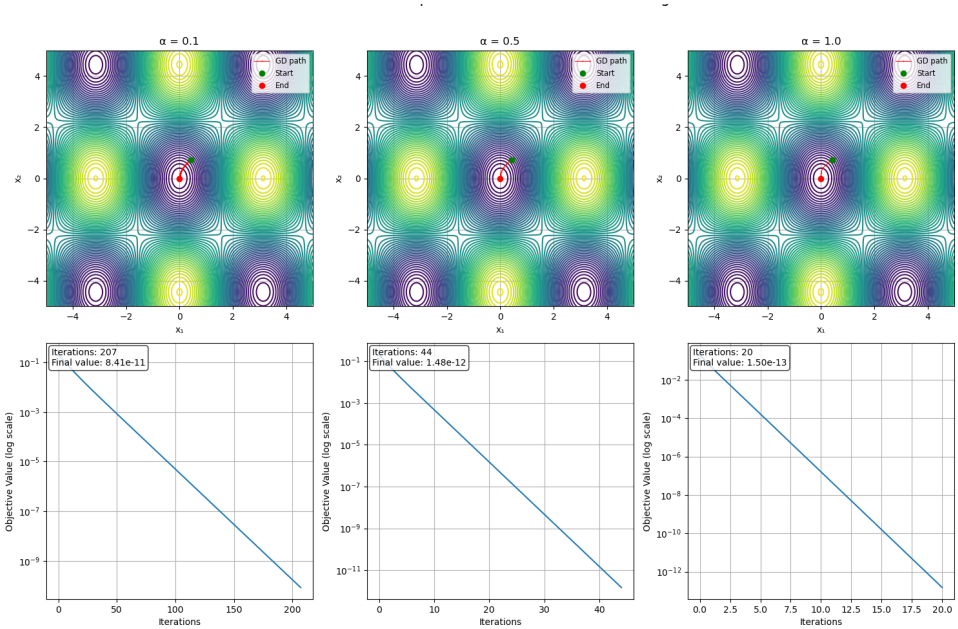


Figure 10: Optimization path and Convergence for GD in 2D Griewank Function.

2.4 Discussion

Both Matyas and Rosenbrock functions highlight the importance of choosing an appropriate learning rate. Low rates lead to slow convergence, while excessively high rates cause divergence. Moderate values offer a balance between speed and stability. The optimal learning rate varies between functions due to differences in their curvature and steepness. For example, the Matyas function, being quadratic, tolerates higher learning rates better than the Rosenbrock function, which has a narrow, curved valley. Oscillations near the minimum indicate instability from larger step sizes. For highly curved functions like the Rosenbrock, this issue becomes more pronounced with high learning rates.

For the 1D Griewank case, as α increases, convergence improves significantly, with the iteration count reducing from 103 ($\alpha = 0.1$) to 19 ($\alpha = 0.5$) and only 4 ($\alpha = 1.0$). Similarly, in the 2D Griewank case, the contours demonstrate how GD navigates the oscillatory landscape toward the global minimum. With $\alpha = 0.1$, convergence is slower (207 iterations), improving at $\alpha = 0.5$ (44 iterations) and $\alpha = 1.0$ (20 iterations). Overall, both cases confirm that selecting an appropriate step size is crucial for balancing convergence speed and precision, especially in complex non-convex landscapes like the Griewank function.

2.5 Armijo's Backtracking Line Search Method

2.5.1 Theoretical Background

The Backtracking Line Search is an adaptive step size selection method that ensures sufficient decrease in the objective function while maintaining computational efficiency. The method is based on the Armijo-Goldstein condition (1966), which requires:

$$f(x_k + \alpha p_k) \leq f(x_k) + c\alpha \nabla f(x_k)^T p_k \tag{1}$$

where α is the step size, c is the sufficient decrease parameter ($0 < c < 1$), p_k is the search direction, and $f(x_k)$ is the objective function value at iteration k.

2.5.2 Implementation Details

For our comparative analysis, we implemented two gradient descent variants with there hyperparameters chosen as depicged in the table below [for Rosebrock's function the step-size α is reduced due to stability issues[1]]:

Fixed Step Gradient Descent Parameters	
Parameter	**Value/Description**
α for Matyas and Griewank functions	0.1
α for Rosenbrock function	0.001
τ	1.0
c	0.0
Backtracking Line Search Parameters	
Parameter	**Value/Description**
Initial step size α_0	1.0
Reduction factor τ	0.1
Sufficient decrease parameter c	0.0001

Table 9: Gradient Descent and Line Search Parameters

[1]RuntimeWarning: overflow encountered in scalar power of $f(x_1, x_2)$ and in scalar multiply of $\partial f/\partial x_1$.

2.5.3 Comparative Analysis

Rosenbrock Function

The Rosenbrock function optimization reveals significant differences between the methods. The backtracking method achieves convergence in approximately the same number of iterations (1000) as the fixed step method, but shows a more direct path to the optimum. The fixed step method exhibits a more conservative approach with smaller steps, particularly evident in the zigzag pattern near the optimum.

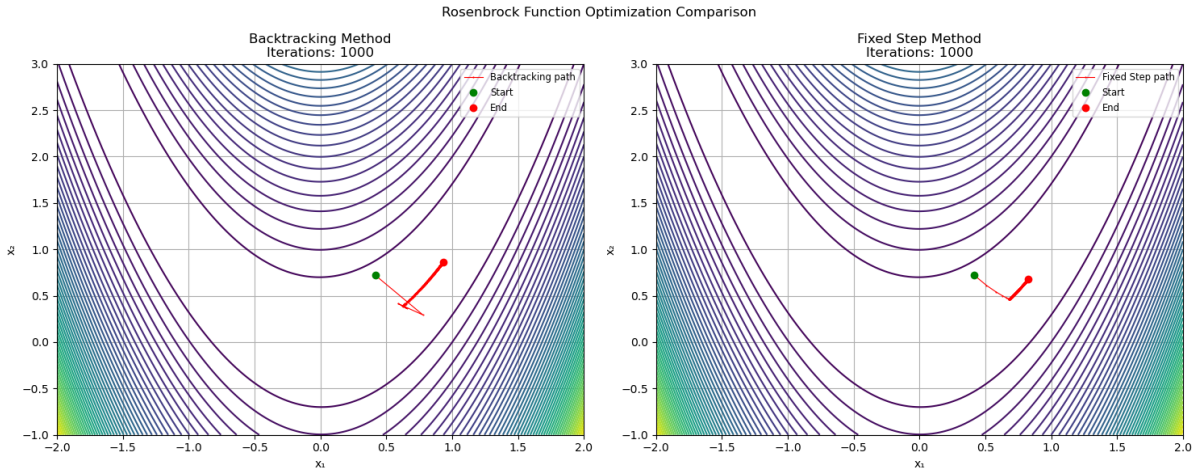


Figure 11: Optimization paths for plain and Backtracked GD in Rosenbrock Function.

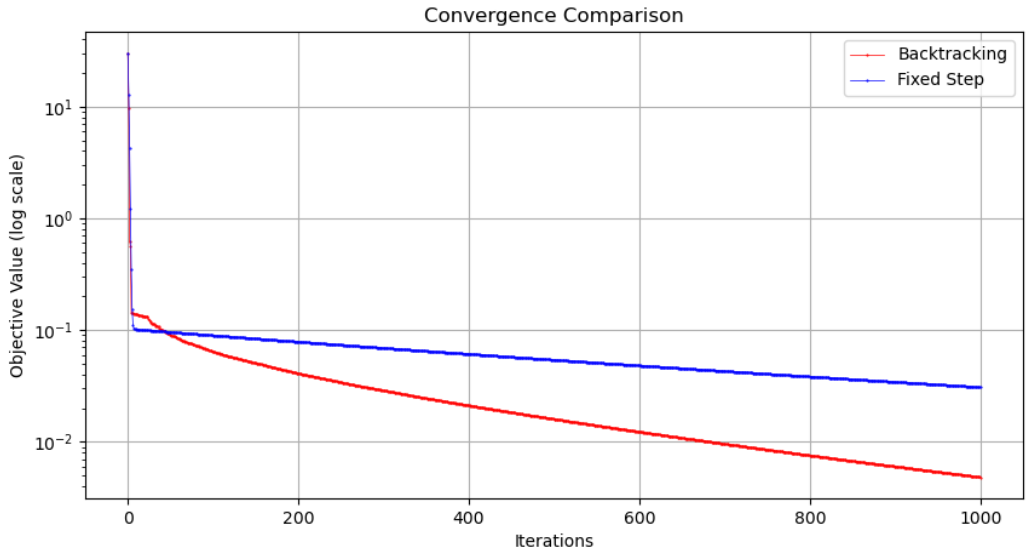


Figure 12: Convergence plot for plain and Backtracked GD in Rosenbrock Function.

Matyas Function

For the Matyas function, the backtracking method demonstrates superior efficiency, requiring only 255 iterations compared to 1000 for the fixed step method. Both methods follow similar paths, but the backtracking method's adaptive step size allows for larger steps when possible, leading to faster convergence.

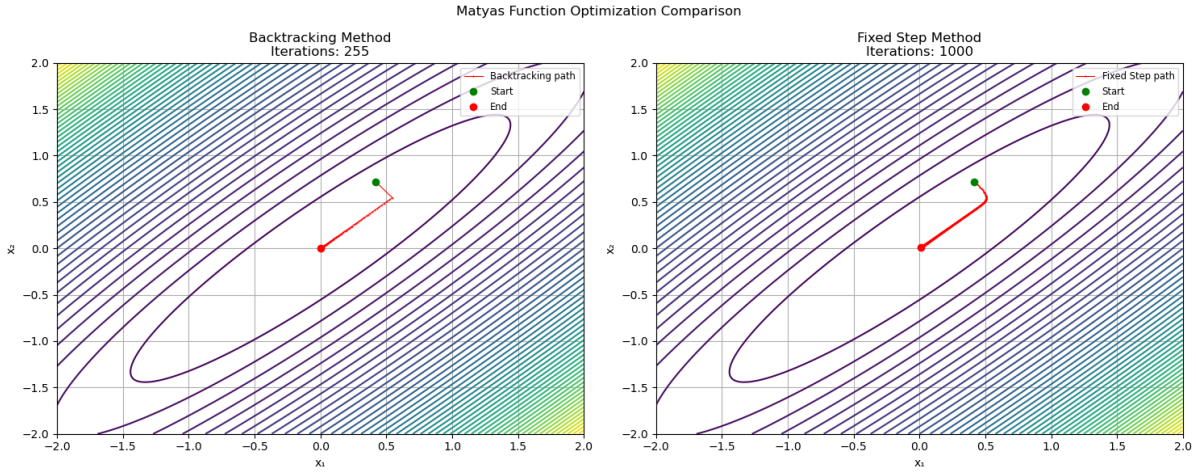


Figure 13: Optimization paths for plain and Backtracked GD in Matyas Function.

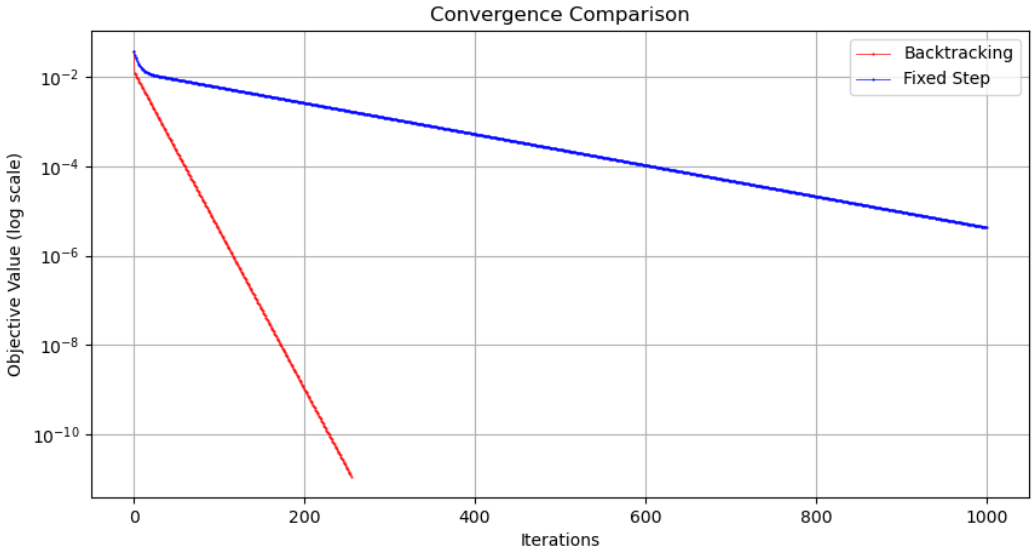


Figure 14: Convergence plot for plain and Backtracked GD in Matyas Function.

1D Griewank Function

The most dramatic difference is observed in the 1D Griewank function optimization:

- Backtracking method: 3 iterations
- Fixed step method: 103 iterations

This remarkable efficiency difference highlights the advantage of adaptive step sizing in simple univariate objectives.

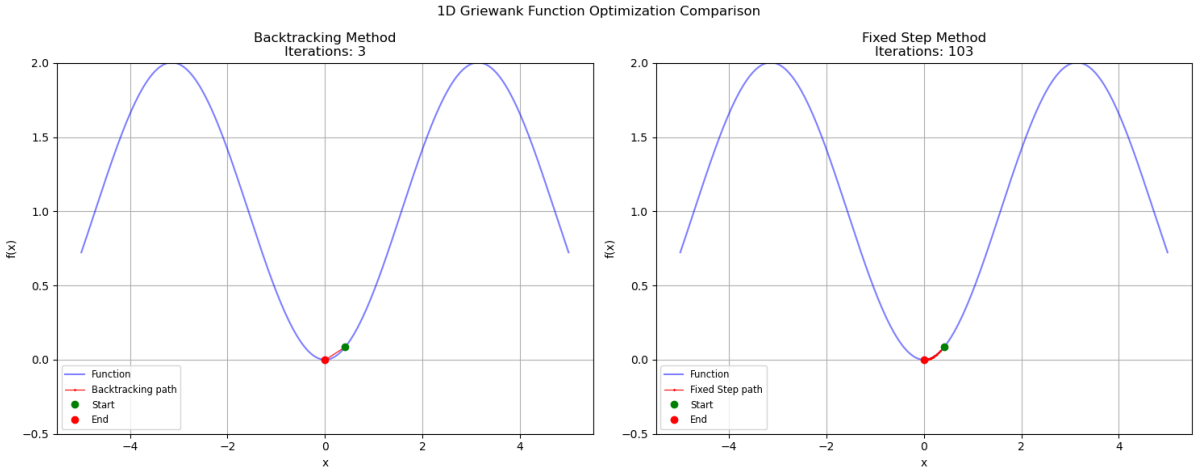


Figure 15: Optimization paths for plain and Backtracked GD in 1D Griewank Function.

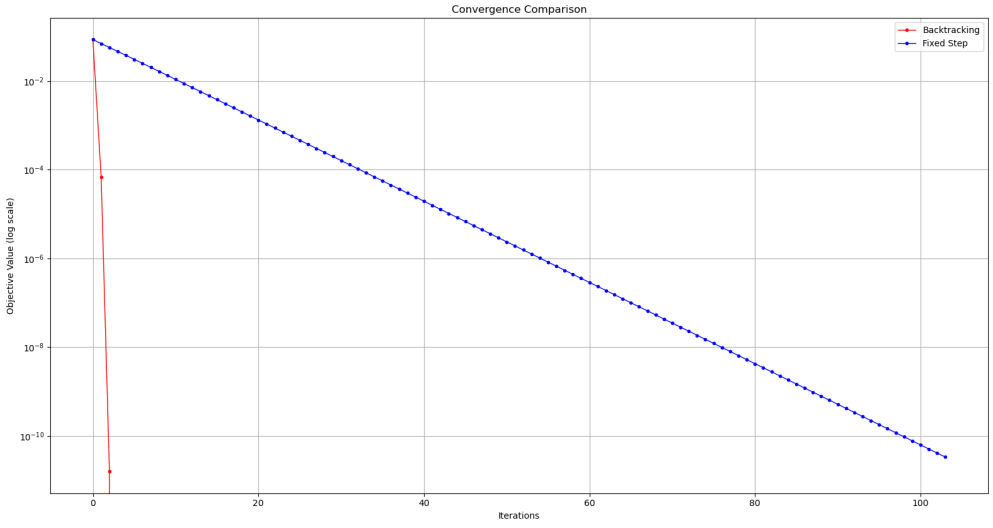


Figure 16: Convergence plot for plain and Backtracked GD in 1D Griewank Function.

2D Griewank Function

The 2D Griewank function results further reinforce the backtracking method's efficiency:

- Backtracking method: 19 iterations
- Fixed step method: 207 iterations

The contour plots show both methods following similar paths, but the backtracking method takes more aggressive steps when the multimodal landscape permits.

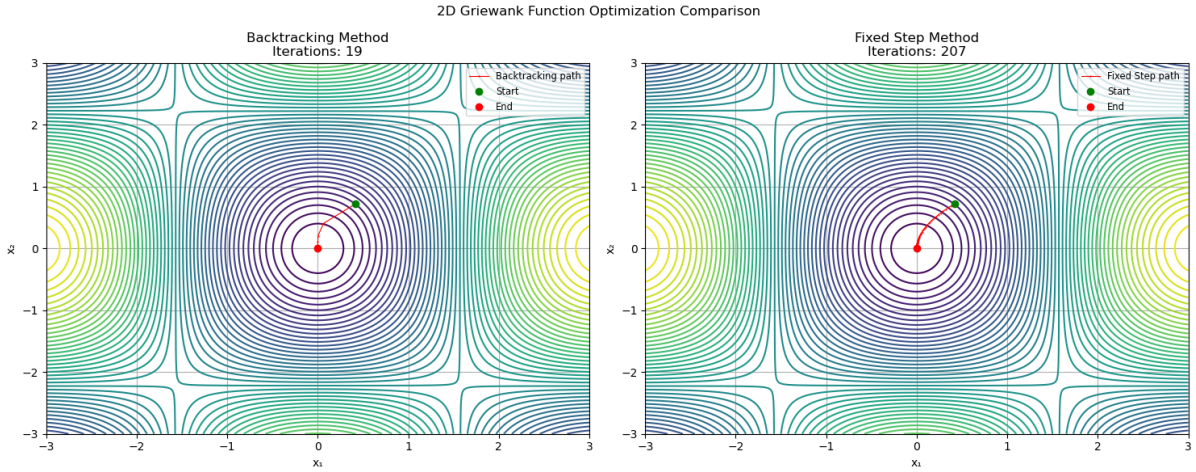


Figure 17: Optimization paths for plain and Backtracked GD in 2D Griewank Function.

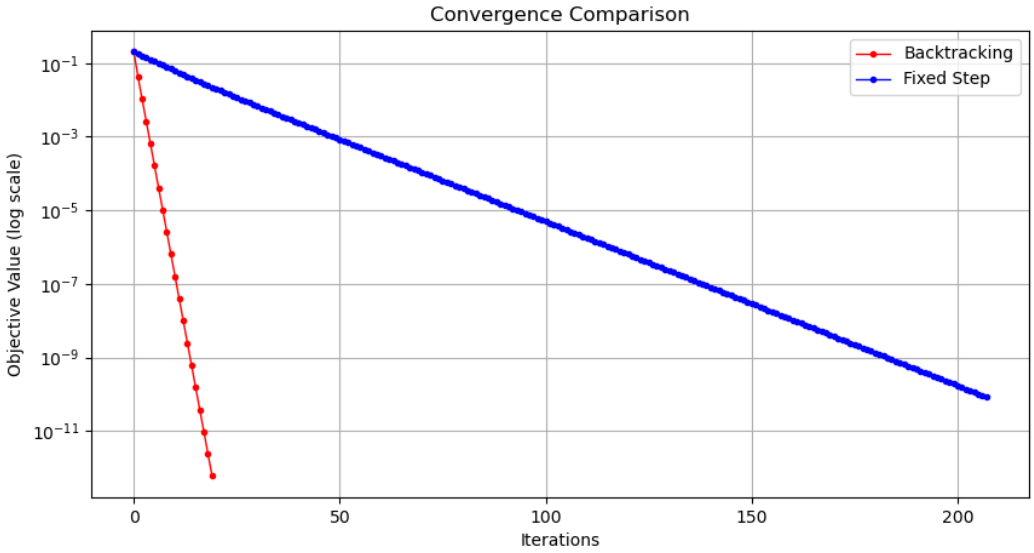


Figure 18: Convergence plot for plain and Backtracked GD in 2D Griewank Function.

2.5.4 Key Observations

Efficiency: The backtracking method consistently requires fewer iterations across all test functions, with the most dramatic improvements in the Griewank functions.

Stability: The need for a smaller step size ($\alpha = 0.001$) in the fixed step method for the Rosenbrock function highlights the robustness of the backtracking method, which automatically adjusts its step size to maintain stability.

Path Characteristics: The backtracking method generally produces more direct paths to the optimum, while the fixed step method tends to take more conservative steps, often resulting in zigzag patterns.

Computational Considerations: While the backtracking method requires fewer iterations, each iteration involves multiple function evaluations to determine the appropriate step size. However, the reduction in total iterations often outweighs this additional per-iteration cost.

2.5.5 Implementation Note

All experiments were conducted with RANDOM_SEED=1 to ensure reproducibility, along with the (pseudo)randomly generated initial position vectors, sampled uniformly in the domain $\mathcal{D} = [0, 1]$, thus we were already close on the global minima ($f(x^*) = 0$ for all trial functions). The Rosenbrock function implementation required special consideration due to potential overflow issues in the function evaluation and gradient computation.

3. Spectral Face Recognition System

In this problem, we implemented and analyzed a face recognition system using the eigenface approach based on Principal Component Analysis (PCA). The system was tested on a dataset of facial images to classify them as known faces, unknown faces, or non-faces. We introduced normalization techniques to improve the robustness of the classification and empirically determined optimal threshold values.

3.1 Mathematical Framework

3.1.1 Problem Formulation

Given a set of N face images, each with $m \times n = M$ pixels, we represent each face as an M-vector. The face recognition problem can be formulated as follows:

1. Form face distribution matrix $S = [\mathbf{f}_1, \mathbf{f}_2, \ldots, \mathbf{f}_N]$, where each $\mathbf{f}_i$ is a face vector.
2. Compute mean face $\bar{\mathbf{f}} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{f}_i$.
3. Perform mean normalization: $\mathbf{a}_i = \mathbf{f}_i - \bar{\mathbf{f}}$, forming matrix $A = [\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_N]$.
4. Compute reduced SVD: $A_{(M\times N)} = U_{(M\times M)} \Sigma_{(M\times N)} V^T_{(N\times N)}$.

The vectors $\mathbf{u}_i$ form an orthonormal basis of the face subspace $\mathcal{R}(A)$ ($\text{span}(\{\mathbf{u}_i\}_{i\in[r]}) \subseteq \mathcal{R}(A)$).

For a new face $\mathbf{f}$, we compute: $\mathbf{x} = [\{\mathbf{u}_i\}_{i=1}^{r}]^T (\mathbf{f} - \bar{\mathbf{f}})$.

3.1.2 Normalization Adjustments

We introduced several normalizations to improve the robustness of the algorithm:

- **Pixel Value Normalization**: Images are normalized to the $[0, 1]$ range: $\mathbf{f}_{\text{normalized}} = \frac{\mathbf{f}}{255}$.
- **Distance Normalization**: Distances are normalized by image dimensions:

$$\epsilon_f = \frac{\|(\mathbf{f} - \bar{\mathbf{f}}) - \mathbf{f}_p\|_2}{\sqrt{M}}, \quad \epsilon_i = \frac{\|\mathbf{x} - \mathbf{x}_i\|_2}{\sqrt{M}}.$$

This normalization makes the thresholds more interpretable and less dependent on image size.

3.2 Implementation Details

The implementation follows a structured approach. First, the preprocessing step involves converting the images to grayscale[2], and normalizing the pixel values to the $[0, 1]$ range[3]. During the training phase, the mean face is computed, followed by singular value decomposition (SVD) of the mean-normalized training data. This step extracts the eigenfaces, which serve as the basis vectors of the face subspace. The training face images are then projected onto the eigenface subspace to obtain their lower-dimensional representations. Finally, classification is performed using two thresholds: ϵ_1 to distinguish between face and non-face images, and ϵ_0 to classify faces as either known or unknown.

3.3 Results

3.3.1 Optimal Thresholds

In this subsection, we expand upon the process of hyperparameter tuning with conditional constraints. Specifically, we focus on the scenario where two hyperparameters, denoted as ϵ_0 and ϵ_1, are tuned under the condition $\epsilon_0 > \epsilon_1$. These thresholds correspond to distinct decision boundaries: ϵ_0 serves as the known/unknown face threshold, while ϵ_1 defines the face/non-face threshold.

Hyperparameter Search Methodology. To optimize these thresholds, we employed a grid search over possible values. Let the set of candidate values for ϵ_0 and ϵ_1 be denoted by $\mathcal{E}_0 = [e_0^1, e_0^2, \ldots, e_0^m]$ and $\mathcal{E}_1 = [e_1^1, e_1^2, \ldots, e_1^n]$, respectively. The key challenge in this process was enforcing the condition $\epsilon_0 > \epsilon_1$, which restricts the grid search to valid pairs (e_0^i, e_1^j) such that $e_0^i > e_1^j$.

By leveraging sorted candidate sets and efficient search strategies (e.g., binary search), the computational complexity of the grid search was reduced from $O(m \times n)$, in the absence of constraints, to $O(n \times \log m)$, significantly improving scalability.

Empirical Results. Based on empirical testing, the optimal thresholds were determined as follows:

- $\epsilon_0 = 25$: The threshold distinguishing known from unknown faces.
- $\epsilon_1 = 20$: The threshold distinguishing faces from non-faces.

[2]When translating a color image to grayscale (mode "L"), the library uses the ITU-R 601-2 luma transform: $L = R \cdot 299/1000 + G \cdot 587/1000 + B \cdot 114/1000$.

[3]The image 26.jpg in training data had dimensions of 320×319 pixels, thus it has being rescaled properly at preprocessing step.

3.3.2 Classification Results

In the following table are the results of the testing performed on testfaces' images. Note that the U3.jpg is indeed an image contained on the training set (U3.jpg = 34.jpg) and it is correctly identified.

Test Image	ϵ_f	$\min_{i\in[N]}(\epsilon_i)$	**Classification**
1.jpg	5.22e-13	8.41e-15	Known (1)
3.jpg	7.62e-13	1.30e-14	Known (3)
5.jpg	4.65e-13	9.45e-15	Known (5)
11.jpg	5.63e-13	7.92e-15	Known (11)
25.jpg	8.21e-13	9.94e-15	Known (25)
38.jpg	0.1660	0.0192	Known (36)
noface.jpg	37.9784	52.2470	Not a face
U1.jpg	45.3093	49.2338	Not a face
U2.jpg	39.0723	56.5716	Not a face
U3.jpg	1.04e-12	0 (underflow)	Known (34)

Table 10: Classification Results

3.4 Analysis

The results show clear separation between categories (80% success over the specific testing set):

- **Known Faces**: Very low ϵ_f (~ 0.17) and $min_{i\in[N]}(\epsilon_i)$ (~ 0.02).
- **Non-Faces**: High ϵ_f (> 37) and $min_{i\in[N]}(\epsilon_i)$ (> 52).
- **Unknown Faces**: High ϵ_f ($39 - 46$) and $min_{i\in[N]}(\epsilon_i)$ ($49 - 57$).

In the next page we depict the first six grayscaled eigenfaces of the above Spectral analysis of centralized input matrix A.

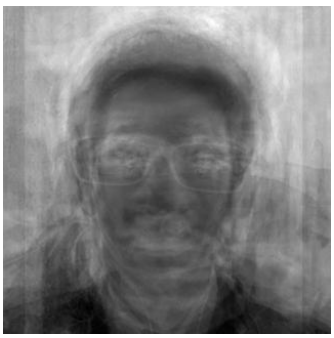
3.5 Potential Improvements

The current implementation of our eigenface recognition system, while effective, presents several opportunities for enhancement across different aspects of the pipeline. In terms of preprocessing, we could implement histogram equalization to normalize image contrast, coupled with robust face alignment techniques to ensure consistent facial orientation. Background removal would help isolate the facial features more effectively, potentially reducing noise in the eigenface computations. The feature space representation could be improved by selecting eigenfaces more effectively, such as using the *cumulative explained variance* to determine the optimal number and weighting eigenfaces by their eigenvalues to emphasize discriminative features.

The conditional hyperparameter tuning approach for the thresholds ϵ_0 and ϵ_1 not only adheres to logical constraints but also reduces computational complexity. Future work may explore alternative optimization techniques, such as Bayesian optimization or gradient-based methods, to further refine these thresholds under more complex constraints.

Classification accuracy could benefit from *adaptive thresholding*, dynamic determination of ϵ_0 and ϵ_1, and extending binary thresholds to handle edge cases, particularly for unknown faces. Modern machine learning classifiers could also capture complex patterns in the projection space. Enhancing robustness through cross-validation, noise resistance testing, and illumination invariance techniques would improve reliability under varying conditions. These refinements would increase both the system's accuracy and practical applicability while preserving computational efficiency.

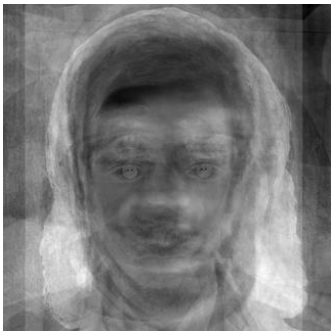
(a) Eigenface 1



(b) Eigenface 2

(c) Eigenface 3

(d) Eigenface 4



(e) Eigenface 5

(f) Eigenface 6

Figure 19: Visualization of the first six eigenfaces.