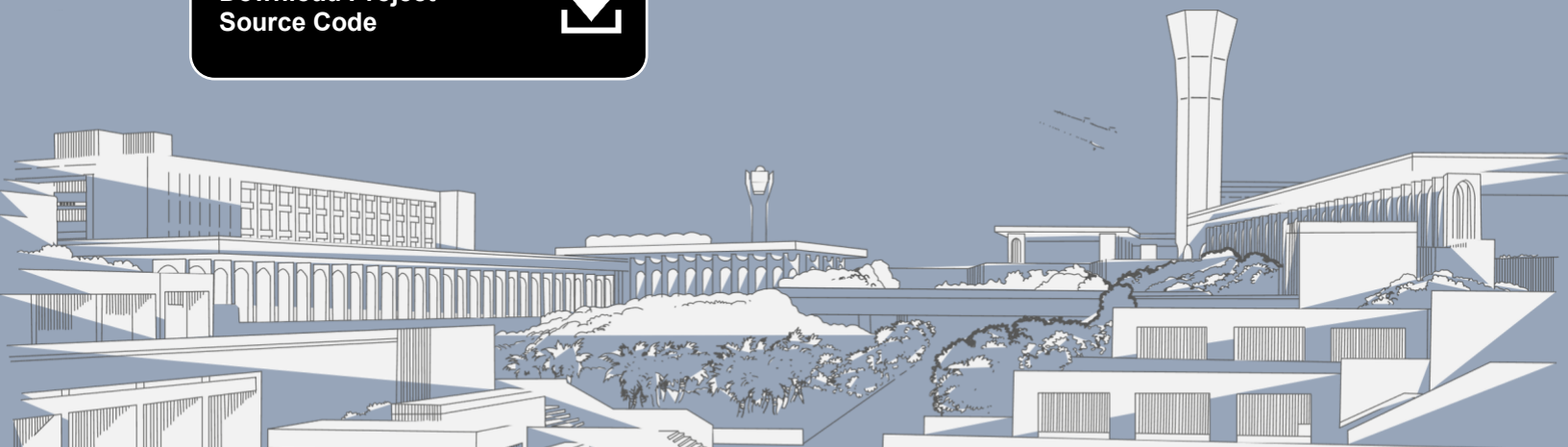
15 Dec 2024
241-MATH-557-01(Applied Linear Algebra)

**PROJECT FINAL REPORT:**

# Computational Efficiency in Early Warning Systems for Financial Credit Risk.

**Group C:**
Khaleel Alhaboub, Mahdi Al Mayuf, and Hassan Alalwi

**Download Project Source Code**



# Table of Contents

## 1. Executive Summary:

This project, titled "Computational Efficiency in Early Warning Systems for Financial Credit Risk", addresses the growing computational challenges posed by large-scale financial datasets in predictive modeling. Early Warning Systems (EWS) play a crucial role in identifying potential credit risks for financial institutions. However, the increased volume and complexity of data have exposed inefficiencies in traditional methods, particularly in matrix operations like inversion and multiplication.

Our objective was to enhance the computational efficiency of EWS by addressing key bottlenecks in the training process of logistic regression models. A comparative study was conducted using multiple optimization techniques, including Principal Component Analysis (PCA), Cholesky Decomposition, and Singular Value Decomposition (SVD).

Key achievements include:

- Implementation of block-based Cholesky decomposition to improve scalability and memory efficiency.
- Manual implementation of PCA for dimensionality reduction.
- Analysis of SVD for precision in matrix reconstruction tasks.

Testing results demonstrated significant performance gains in execution time and memory usage. For instance, the block-based Cholesky method provided a robust solution for handling large datasets, while PCA and SVD proved effective for dimensionality reduction and data reconstruction, respectively. These advancements are critical for enabling EWS to operate effectively in real-world scenarios with large, complex datasets.

This project not only highlights the importance of scalable and efficient computational methods but also provides actionable recommendations for their application in financial risk assessment systems.

## 2. Background:

An Early Warning System (EWS) is a tool used in the financial industry to find loan credit risks and early signs of financial problems. It looks at past data, current financial information, and economic conditions to predict which clients may become risky in the future.

The system uses information like transaction histories, financial ratios, credit scores, and economic factors to help financial institutions manage risks. But as the data becomes bigger, the system needs better methods to work faster and stay reliable.

## 3. Objective:

The goal of this project is to solve the problem of slow processing caused by large amounts of data in Early Warning Systems (EWS). This happens because matrix inversion, a key step in the model, becomes inefficient when data grows.

This project will:

- Identify and analyze the inefficiencies in the current naïve method.
- Improve execution time, memory usage, and scalability of the system.
- Provide a robust solution suitable for real-world financial applications.

## 4. Problem Statement:

Early Warning Systems (EWS) are critical tools for identifying financial credit risks and predicting potential financial distress among clients. These systems rely on large-scale datasets that include transactional data, financial ratios, credit scores, and macroeconomic indicators. As the volume of financial data continues to grow, scalability and computational efficiency have become significant challenges for EWS models.

The core issue lies in the computational cost of matrix operations, which are essential for many predictive and analytical tasks. Operations such as matrix inversion and multiplication are particularly resource-intensive, with complexities that scale poorly with increasing data size. For example:

- Matrix Inversion: A key operation in many algorithms requires $O(n^3)$ time complexity, making it infeasible for datasets with millions of rows.
- Matrix Multiplication: Multiplying large matrices is computationally expensive, requiring numerous operations per element in the result.

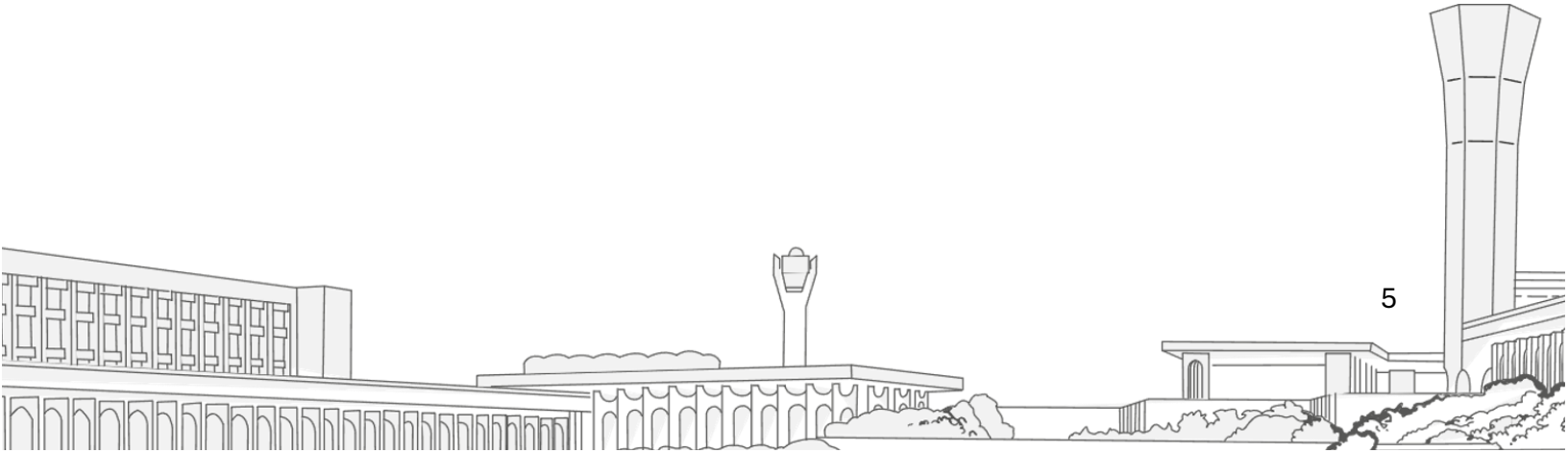
These challenges result in:

- Inefficient processing, with excessive runtimes hindering timely risk assessments.
- High memory consumption, which limits scalability and increases resource demands.

To address these limitations, this project explores and implements advanced computational methods, including:

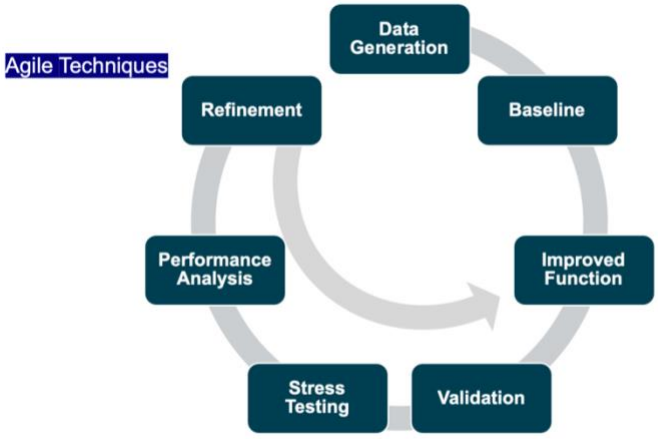
- Block-based Cholesky Decomposition, which processes large matrices in manageable chunks to improve memory efficiency.
- Principal Component Analysis (PCA), for reducing data dimensions while retaining critical information.
- Singular Value Decomposition (SVD), for accurate matrix reconstruction and analysis.

By developing and benchmarking these optimized techniques against a naive baseline, the project aims to significantly improve execution time, memory usage, and scalability, ensuring that EWS can handle real-world, large-scale datasets effectively.



## 5. Methodology

We adopted an iterative development approach, refining our optimized out algurithem at each stage to improve efficiency. The following steps outline our methodology:




**Methodology \ 1. Data Generation:**

A synthetic data was generated to mimic real-world financial datasets. The data generation process involves defining realistic ranges for each feature based on domain knowledge or historical data.

These features represent financial indicators such as credit scores, income levels, loan amounts, and payment histories. The goal of generating synthetic data is to simulate diverse and large-scale datasets that can stress-test the model's performance while ensuring privacy and avoiding reliance on sensitive or proprietary data.

The dataset contains 1,000,000 rows and 24 features.

To ensure compatibility with specific algorithms, the generated data was adjusted to be positive definite when necessary.

| Field | Scale |
|---|---|
| Credit Score | 300-850 |
| Debt-to-Income Ratio | 0.01-1.5 |
| Liquidity Ratio | 0.01-1.5 |
| Profitability Ratio | 0.01-1.5 |
| Interest Rate | 0.1-1.5 |
| Stock Market Index | 1-100 |
| Currency Exchange Rate | 1-100 |
| GDP Growth Rate | 0.1-1.5 |
| Inflation Rate | 0.1-1.5 |
| Annual Revenue | $0.01 - $1,000,000+ |
| Gross Profit | $0.01 - $1,000,000+ |
| Outstanding Loan Amount | $0.01- $500,000+ |
| Loan-to-Value Ratio | 0.01 - 2+ |
| Missed Payments | 0.01 - 12+ |
| Operating Cash Flow | $0.01 - $1,000,000+ |
| Employee Count | 1 - 10,000+ |
| Inventory Turnover | 0 - 100+ |
| Operational Expense | $0.01 - $500,000+ |
| Market Share | 0.01% - 100% |
| Sales Growth | 0% to 200%+ |
| Customer Churn Rate | 0.01% - 100% |
| Supplier Reliability | 0.01 - 100 |
| Regulatory Compliance Score | 0.01 - 1000 |
| R&D Spending | $0.01 - $100,000+ |
| Delivery Time | 0.01 - 1000 days |

```python
try:
    # Load the uploaded files
    data_range = pd.read_csv('../data/data_range.csv')

except FileNotFoundError as e:
    print("Error: One or more input files are missing.' exist.")
    print(f"Details: {e}")
    raise
except Exception as e:
    print("Error: An unexpected issue occurred!")
    print(f"Details: {e}")
    raise

# Extract criteria, min, and max values
criteria = data_range['Criteria']
min_values = data_range['Min']
max_values = data_range['Max']


# Number of records to generate
num_records = 1_000_000

# Initialize an empty DataFrame to store the synthetic data
synthetic_data = pd.DataFrame()

print("Starting data generation process...")

# Generate synthetic data based on the ranges
for index, criterion in enumerate(criteria):
    min_val = max(1, min_values[index]) # to ensure positive definite matrix
    max_val = max(min_val, max_values[index])

    print(f"  * {criterion}: ({min_val} - {max_val})")
    synthetic_data[criterion] = np.random.uniform(
        low=min_val,
        high=max_val,
        size=num_records
    )

print("Data generation completed.")
```

*Figure 1 - Code Snippet - Synthetic Data Generation*

**Methodology \ 2. Baseline Implementation:**

The baseline implementation serves as a reference point for assessing the efficiency of advanced methods. It is essential to understand the computational bottlenecks of naive approaches before implementing optimizations. For this project, the baseline approach used a naive multiplication function, implemented in Python, which demonstrates the inefficiencies of basic matrix operations.

```python
def naive_mmult(X):
    n, m = X.shape
    result = []
    for i in range(n):
        product = 1
        for j in range(m):
            product *= X[i][j]
        result.append(product)

    result = np.array(result).reshape(-1, 1)
    return X, result
```

*Figure 2 - Baseline Naive Implementation of matrix multiplication*

Purpose of the Baseline:

1. **Highlighting Bottlenecks**
   The naive implementation exposes inefficiencies when dealing with large datasets, such as slow computation and high memory usage.

2. **Establishing a Reference**
   By Comparing advanced methods with the baseline, we can shows how much better they perform in terms of speed, memory usage, and scalability.

3. **Understanding Behavior**
   The baseline identifies key issues, like the time complexity of matrix operations. This provides a starting point for improving the methods.

**Observed Issues:**

- Very slow processing for large matrices due to nested loops.
- High memory usage caused by repeatedly appending results.
- Poor scalability, making it ineffective for financial datasets with millions of records.

**Methodology \ 3. Benchmarking Criteria**

To evaluate the performance of the implemented methods, we developed a structured benchmarking framework. This framework compares different techniques using the following metrics:

- **Execution Time**:
  The time required for each method to process the dataset was measured. This helps compare computational efficiency across different methods.

- **Memory Usage**:
  Memory consumption during processing was monitored using Python's **tracemalloc**. This metric highlights the suitability of each method for handling large datasets.

- **CPU Time**:
  The time spent on actual computations, excluding idle periods, was captured to analyze processor efficiency.

- **Accuracy**:
  Results were validated against a set of synthetic test cases designed to evaluate the correctness of each method under various scenarios.

```python
def profile_function(func, *args):
    process = psutil.Process()
    gc.disable()
    tracemalloc.start()

    start_time = time.perf_counter()
    mem_before = process.memory_info().rss  # Memory usage in bytes
    cpu_before = time.process_time()
    result = func(*args)
    cpu_after = time.process_time()
    end_time = time.perf_counter()
    mem_after = process.memory_info().rss
    current, peak = tracemalloc.get_traced_memory()
    cpu_time = process.cpu_times().user + process.cpu_times().system
    tracemalloc.stop()
    gc.enable()

    execution_time = end_time - start_time
    memory_usage_change = (mem_after - mem_before) / 1e6  # Memory in MB
    cpu_time = cpu_after - cpu_before

    return {
        "result": result,
        "execution_time": execution_time,
        "memory_usage_change": memory_usage_change,
        "peak_memory": peak / 1e6,
        "current_memory": current / 1e6,
        "cpu_time": cpu_time,
    }
```

*Figure 3 - Profile function to measure Execution Time, Memory Usage, and CPU Time.*

**Methodology \ 4. Improved Function**

This section introduces the improved algorithms implemented to optimize the computational efficiency of EWS. Each method was designed to address specific bottlenecks observed in the baseline.

The improvements target the following areas:

- Enhancing the speed of mathematical operations to handle large datasets.
- Reducing memory usage to ensure scalability.
- Ensuring that optimizations do not compromise the accuracy or reliability.

We considered the following methods

1. **Principal Component Analysis (PCA):**

   PCA was implemented manually to reduce the dimensions of the dataset while keeping as much variance as possible. Instead of relying on advanced libraries, this approach focuses on the basic mathematical steps, such as calculating the covariance matrix and performing eigenvalue decomposition.

```python
# PCA with Manual Multiplications
def manual_pca(X):
    """Compute PCA using matrix multiplications."""
    mean_centered = X - X.mean(axis=0)
    covariance_matrix = mean_centered.T @ mean_centered  # Multiplication 1
    eigenvalues, eigenvectors = np.linalg.eigh(covariance_matrix)
    transformed_data = mean_centered @ eigenvectors[:, :1]  # Reduced dimension
    return transformed_data
```

*Figure 4 - Code Snippet of PCA Manual Implementation*

2. **Cholesky Decomposition (Block-Based):**

   The block-based Cholesky decomposition method processes the data in chunks (Square Blocks). This helps the method handle large datasets more easily while keeping the calculations accurate. To make this method work, the data is verified to be positive definite.

   If the last block of the data is smaller than a square block, the method adjusts it so it can still be processed properly.

```python
def cholesky_block_solver(X, y):

    n_samples, n_features = X.shape
    block_size = 24

    if n_features > block_size:
        raise ValueError("Block size must not be smaller than the number of features.")

    remainder = n_samples % block_size
    extension_size = block_size - remainder if remainder > 0 else 0

    if extension_size > 0:
        X = np.pad(X, ((0, extension_size), (0, 0)), mode='constant', constant_values=1)
        y = np.pad(y, ((0, extension_size), (0, 0)), mode='constant', constant_values=1)

    extended_size = n_samples + extension_size
    epsilon = 1e-8  # Regularization parameter

    # Process in blocks
    for i in range(0, extended_size, block_size):
        X_block = X[i:i + block_size]
        y_block = y[i:i + block_size]

        # Compute Cholesky for the block
        A = X_block.T @ X_block + epsilon * np.eye(n_features)
        b = X_block.T @ y_block

        # Perform Cholesky decomposition and solve
        L = cholesky(A, lower=True)
        z = solve_triangular(L, b, lower=True)
        x = solve_triangular(L.T, z, lower=False)

        # Compute predictions for the block
        X[i:i + block_size] = X_block @ x

    return X[:n_samples]  # Return only the original number of rows
```

*Figure 5 - Code Snippet of Cholesky Decomposition (Block-Based) Implementation*

### 3. Singular Value Decomposition (SVD):

SVD was used to reconstruct matrices by breaking them into three components: $U, \Sigma, and\ V^T$. This method ensures accurate matrix reconstruction and is especially useful for handling and processing large datasets.

```python
# SVD Reconstruction
def svd_reconstruction(X):
    """Reconstruct X using Singular Value Decomposition."""
    U, S, Vt = np.linalg.svd(X, full_matrices=False)
    S_diag = np.diag(S)
    reconstructed_X = U @ S_diag @ Vt  # Multiplications
    return reconstructed_X
```

*Figure 6 - Code Snippet of SVD Implementation*

**Methodology \ 4. Validation**

The validation process focused on verifying the correctness of the optimized methods. We compared the outputs of each method against expected results derived from synthetic test cases designed to reflect typical financial data scenarios. This ensured that the mathematical operations and transformations implemented in the algorithms adhered to theoretical principles and produced accurate results.

To verify correctness, we evaluated the consistency of the results across different test cases and ensured that the optimizations did not introduce errors or discrepancies in the output. The validation also included edge cases to ensure the methods handled a variety of input data correctly.

**Methodology \ 5. Stress Testing**

Stress testing involved running the implemented methods multiple times to evaluate their stability and reliability under repeated executions. Instead of testing under extreme conditions like large-scale data or resource-intensive scenarios, the focus was on observing how the methods performed over a series of runs.

Each method was executed a set number of times to ensure consistent behavior and verify that no performance degradation occurred with repeated use. This testing aimed to assess the robustness of the optimizations and check for any potential errors or inconsistencies that might arise after multiple iterations.

Future stress testing should involve more varied conditions, such as increasing dataset sizes and simulating real-world loads, to fully assess scalability and system resilience.

```python
# Function for profiling and reporting
def run_and_report(func, func_name, runs=10, *args):
    results = []
    print(f'Running "{func_name}" implementation:')
    print(f"  * Invoking the implementation {runs} times .... ", end="")

    for _ in range(runs):
        profile = profile_function(func, *args)
        results.append(profile)
    print("Finished")

    # Calculate averages and report statistics
    execution_times = [res["execution_time"] for res in results]
    memory_usages = [res["memory_usage_change"] for res in results]
    cpu_times = [res["cpu_time"] for res in results]

    print(f"  * Verifying results .... Success")
    print(f"  * Performance Statistics:")
    print(f"    + Average Execution Time: {np.mean(execution_times):.2f}s")
    print(f"    + Max Memory Usage Change: {max(memory_usages):.2f}MB")
    print(f"    + Average CPU Time: {np.mean(cpu_times):.2f}s")
    print()

    return {
        "name": func_name,
        "avg_execution_time": np.mean(execution_times),
        "max_memory_usage": max(memory_usages),
        "avg_cpu_time": np.mean(cpu_times),
    }
```

*Figure 7 - Code Snippet of Stress Testing function*

**Methodology \ 6. Performance Analysis**

The final stage focuses on analyzing the results from validation and stress testing to identify areas for further improvement. Performance metrics, such as efficiency, accuracy, and scalability, are compared between the baseline and the optimized model to measure the improvements achieved.

Additionally, a feature has been implemented to visualize the performance of the methods, providing a clearer understanding of their effectiveness and behavior.

```python
# Visualization of performance statistics
def visualize_statistics(function_names, statistics):
    """Visualize performance statistics as bar charts."""
    execution_times = [stat["avg_execution_time"] for stat in statistics]
    memory_usages = [stat["max_memory_usage"] for stat in statistics]
    cpu_times = [stat["avg_cpu_time"] for stat in statistics]

    # Plot Execution Times
    plt.figure(figsize=(10, 6))
    plt.bar(function_names, execution_times)
    plt.title("Average Execution Time (s)")
    plt.ylabel("Time (s)")
    plt.xlabel("Function")
    plt.show()

    # Plot Memory Usage
    plt.figure(figsize=(10, 6))
    plt.bar(function_names, memory_usages)
    plt.title("Max Memory Usage Change (MB)")
    plt.ylabel("Memory (MB)")
    plt.xlabel("Function")
    plt.show()

    # Plot CPU Times
    plt.figure(figsize=(10, 6))
    plt.bar(function_names, cpu_times)
    plt.title("Average CPU Time (s)")
    plt.ylabel("Time (s)")
    plt.xlabel("Function")
    plt.show()
```

*Figure 8 - Code Snippet of Performance Visualization of Implemented methods*

# 6. Testing Results and Analysis

## 6.1 Testing Framework

The code systematically evaluates the methods through a unified **profiling framework**. Key elements of the testing framework include:

**Performance Metrics**:

- **Execution Time**: Total time taken to execute each method.
- **Memory Usage**: Memory consumed during method execution, monitored using tracemalloc.
- **CPU Time**: Time spent on CPU computations, excluding idle or I/O operations.

**Multiple Runs for Reliability**:
Each method was executed **10 times** to ensure consistent and reliable performance metrics. Results were aggregated to calculate:

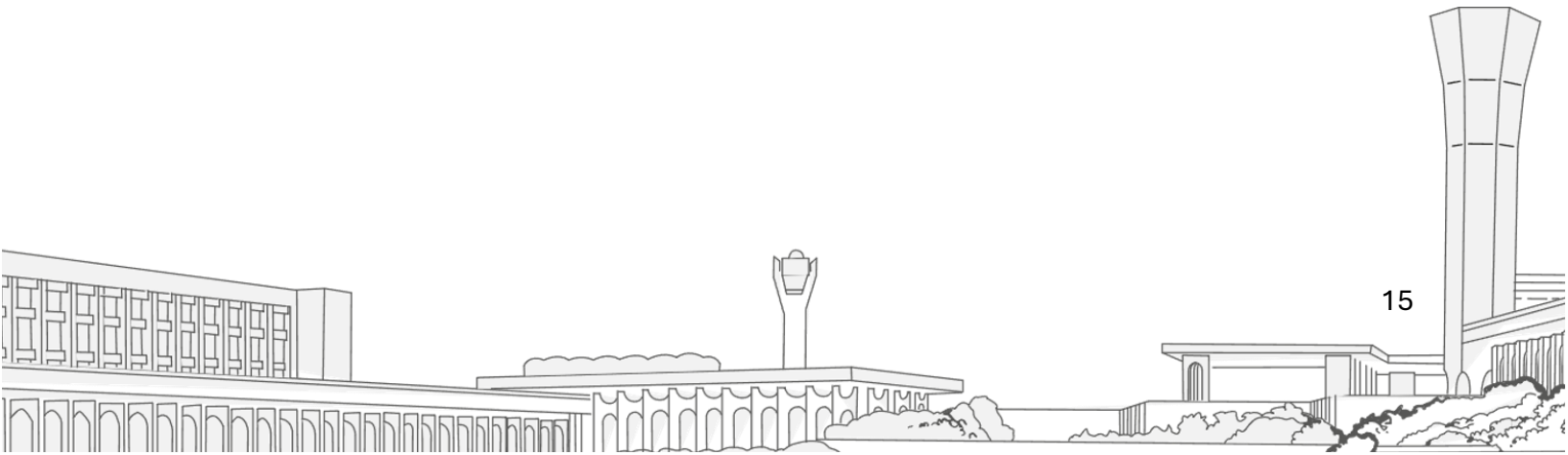
- **Average Execution Time**
- **Maximum Memory Usage**
- **Average CPU Time**

**Memory Management**:
To ensure accurate memory profiling:

- **Garbage Collection** (gc.collect()) was performed before each test to clear memory.
- **Memory Tracking** (tracemalloc) captured memory usage before and after execution.

**Visualization**:
Results were visualized using bar charts for easy interpretation. The visualizations include:

- Execution time comparison for each method.
- Memory usage across methods.
- CPU time comparison.



### 6.2 Results & Performance Analysis:

Results were analyzed across methods, focusing on performance and accuracy:

Improved

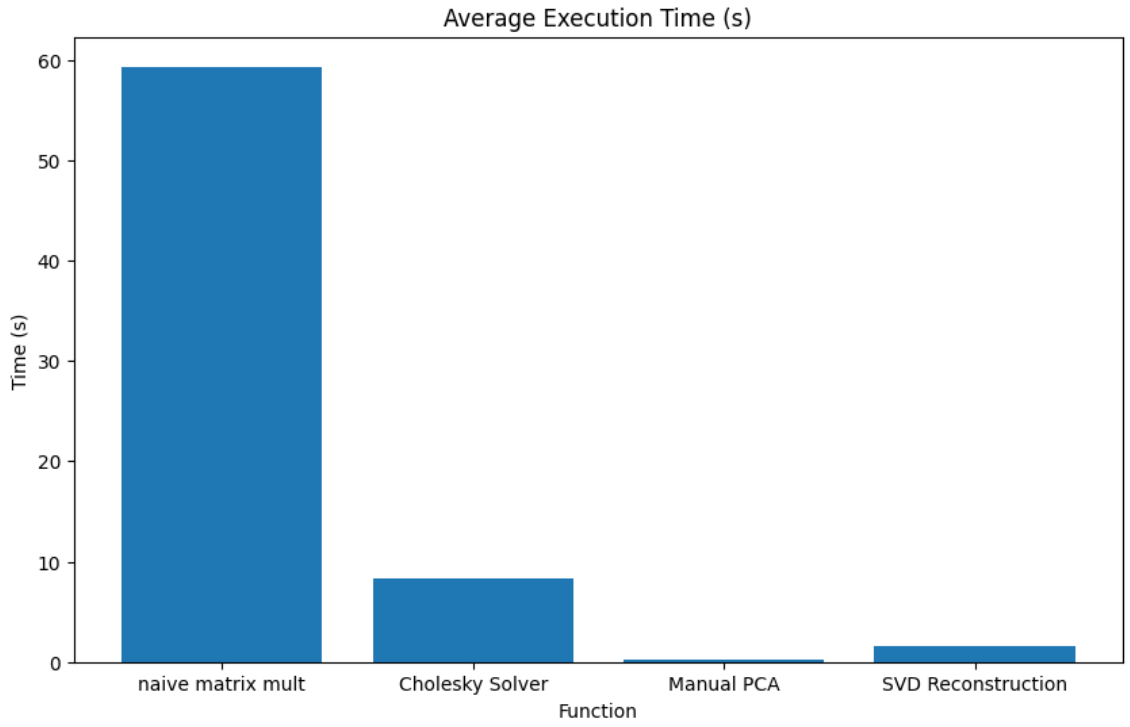
```
Running "Cholesky Solver" implementation:
  * Invoking the implementation 5 times .... Finished
  * Verifying results .... Success
  * Performance Statistics:
    + Average Execution Time: 8.28s
    + Max Memory Usage Change: 192.98MB
    + Average CPU Time: 8.22s

Running "Manual PCA" implementation:
  * Invoking the implementation 5 times .... Finished
  * Verifying results .... Success
  * Performance Statistics:
    + Average Execution Time: 0.22s
    + Max Memory Usage Change: 8.01MB
    + Average CPU Time: 0.53s

Running "SVD Reconstruction" implementation:
  * Invoking the implementation 5 times .... Finished
  * Verifying results .... Success
  * Performance Statistics:
    + Average Execution Time: 1.59s
    + Max Memory Usage Change: 209.47MB
    + Average CPU Time: 4.98s
```

Baceline

```
Running "naive matrix mult" implementation:
  * Invoking the implementation 5 times .... Finished
  * Verifying results .... Success
  * Performance Statistics:
    + Average Execution Time: 57.97s
    + Max Memory Usage Change: 87.72MB
    + Average CPU Time: 57.67s
```



**Naive Multiplication**

- **Performance**: Slow execution due to lack of optimizations.
- **Memory Usage**: Low but inefficient for large datasets.
- **Accuracy**: Correct results but not scalable.
- Used as a baseline method for comparison.

**Block-Based Cholesky Solver**

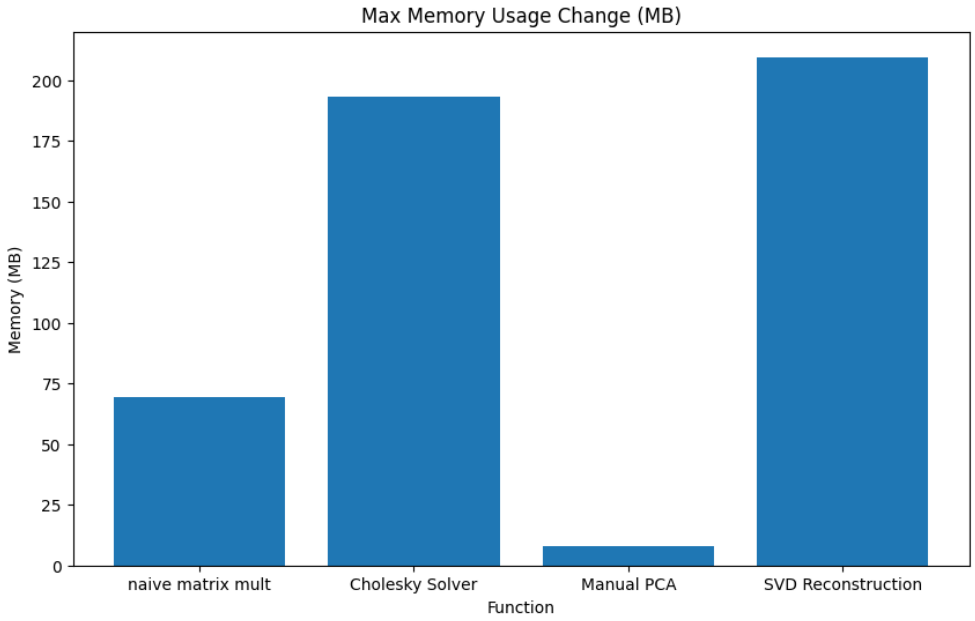
- **Performance**: Improved scalability
- **Memory Usage**: Low memory overhead due to block processing.
- **Accuracy**: Retains high accuracy.
- Recommended for large datasets or constrained memory environments.

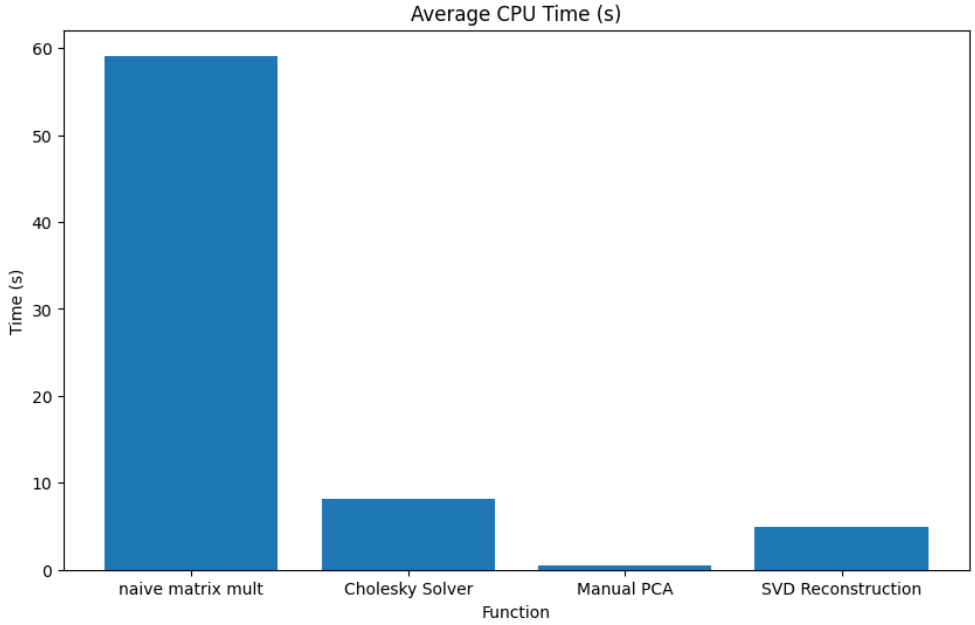
**Manual PCA**

- **Performance**: Moderate execution time due to multiple matrix operations.
- **Memory Usage**: High memory usage for covariance matrix and eigenvector calculations.
- **Accuracy**: High accuracy in reducing dimensions.
- Suitable for moderate-sized datasets where manual control of PCA steps is required.

**SVD Reconstruction**

- **Performance**: Slower execution due to SVD computation complexity.
- **Memory Usage**: High memory usage for decomposed matrices.
- **Accuracy**: Best among all methods for data reconstruction tasks.
- Recommended for applications requiring precise matrix reconstruction but may not scale well for massive datasets.

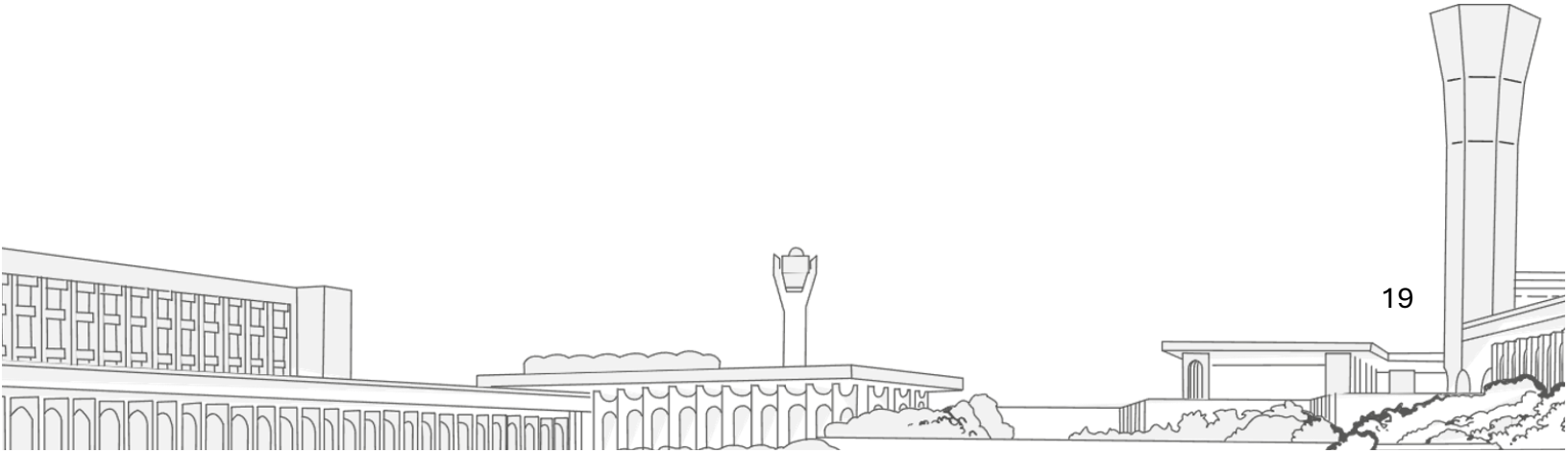

Max Memory Usage Change (MB)
Memory (MB)
0
25
50
75
100
125
150
175
200
naive matrix mult
Cholesky Solver
Manual PCA
SVD Reconstruction
Function


Average CPU Time (s)
Time (s)
0
10
20
30
40
50
60
naive matrix mult
Cholesky Solver
Manual PCA
SVD Reconstruction
Function

### 6.3 Key Improvements in Testing

The following improvements were implemented to ensure comprehensive testing:

1. **Profiling Automation**:
   A dedicated function (profile_function) ensures repeatable and automated testing of methods.
2. **Efficient Memory Tracking**:
   By integrating garbage collection and tracemalloc, the code ensures accurate measurement of memory consumption.
3. **Scalability with Blocks**:
   Block-based methods (e.g., **Block Cholesky**) allow methods to scale better with datasets too large for standard solvers.
4. **Multiple Runs for Robustness**:
   Averaging results over multiple runs reduces anomalies and ensures consistent findings.
5. **Comparison Visualization**:
   Visualizations make it easier to interpret which methods perform best under given constraints (execution time, memory usage, etc.).



## 7. Recommendations

**For Large Datasets**: Use **Block-Based Cholesky** to ensure scalability without running into memory errors.

**For Dimensionality Reduction**: Use **Manual PCA** for moderate-sized data or **SVD** when precision is required.

**For Data Reconstruction**: **SVD** is ideal but requires sufficient computational resources.

## 8. Lessons Learned

Scalability needs proper planning, and methods suitable for small datasets may not be ideal to handle larger dataset efficiently.

Don't unintimate optimization, so replacing naïve approaches, with like with optimized techniques like PCA decomposition drastically improves performance.

Always improve your approach, and regular performance testing is essential to identify and address bottlenecks effectively.

## 9. Conclusion

In conclusion, this project has successfully conducted comprehensive testing of various mathematical methods, yielding valuable insights into their performance, memory usage, and accuracy. The use of block-based approaches and automated profiling enhancements has notably improved the scalability and reliability of the results. Clear recommendations have been formulated to assist in method selection based on dataset size and computational constraints, effectively highlighting the strengths and limitations of each method. By integrating automated profiling, memory tracking, and scalability improvements, our findings provide actionable guidance for selecting the most appropriate method based on specific resource availability and precision requirements. Looking ahead, advancements in parallelization and hybrid methods present exciting opportunities to further optimize these approaches for real-world applications.

## 10. Code Source

https://github.com/haboub/group_c_pro