Singular Value Decomposition (SVD) Benchmarks

1. Introduction

According to lecture notes on The Singular Value Decomposition (2016), SVD is one of the key concepts in linear algebra. It decomposes a matrix $A \in \mathbb{R}^{m \times n}$, its SVD is: $A = U\Sigma V^T$ where:

- $U \in \mathbb{R}^{m \times m}$ is orthogonal (left singular vectors),
- $\Sigma \in \mathbb{R}^{m \times n}$ is diagonal (singular values),
- $V^T \in \mathbb{R}^{n \times n}$ is orthogonal (right singular vectors).

2. Goal of the Experiment

The objective of this experiment is to evaluate the performance of different SVD implementations in Python:

1. A raw Python implementation uses basic Python operations and nested loops.
2. NumPy implementation (numpy.linalg.svd) uses LAPACK routine _gesdd (The NumPy Developers, 2025).
3. PyTorch implementation (torch.linalg.svd) by default (driver= None), 'gesvdj' is used and, if it fails, it uses 'gesvd' (The PyTorch Developers, 2025).

The experiment aims to answer the following:

- How does each implementation perform in terms of execution time and scalability?
- How accurate is the reconstruction of the original matrix from its SVD components?
- How does each implementation perform with multi threads?

3. Experiment Setup

3.1 Hardware and Software Environment

- CPU: Apple M2 Pro
- Total Number of Cores: 12
- GPU: None
- RAM: 16 GB
- OS: macOS
- Python 3.13.x
- NumPy 2.2.6

- PyTorch 2.7.x

3.2 Matrix Generation Method

To ensure reproducibility and control over experiment parameters, input matrices are generated using:

```
import numpy as np

def generate_random_matrix(n, m, seed=42):
    np.random.seed(seed)
    return np.random.rand(n, m)
```

- Square matrices (n=m) such as 500x500, 1000x1000, 2000x2000, 4000x4000 will be tested.
- The same matrix will be used across all implementations for comparison.

4. Methodology

4.1 Implementation

- Raw Python: A custom implementation using the power iteration method to compute the dominant (top-1) singular value and corresponding singular vectors.
- NumPy: Uses np.linalg.svd.
- PyTorch: Uses torch.linalg.svd.

Each implementation returns the top k (k=1) singular values and vectors for benchmarking.

4.2 Metrics Definition

- Size: The dimension (n) of the square matrix (n x n) used for the test.
- Threads: The number of CPU threads were allocated for the computation.
- Execution Time: Time required to compute the SVD of a single matrix (using time.perf_counter).
- Frobenius Error: A measure of how far the approximated matrix (from SVD) is from the original matrix. A lower Frobenius error means the approximation is more accurate.
- Relative Error: The Frobenius error scaled by the total "size" (norm) of the original matrix. It tells what percentage of the original matrix's information is lost. For example, relative error of 0.1 means 90% of the matrix is preserved.

All results will be shown with graphs and tables to highlight the performance, accuracy, and scalability.

5. Results Analysis

In [150…
```
import pandas as pd
import re
import matplotlib.pyplot as plt
import glob
```

In [151…
```
csv_files = glob.glob("data/benchmark_*.csv")
dfs = []

for file in csv_files:
    match = re.search(r"benchmark_s(\d+)_i(\d+)_t(\d+)\.csv", file)
    if match:
        size, iterations, threads = match.groups()
        df = pd.read_csv(file)
        df['size'] = int(size)
        df['iterations'] = int(iterations)
        df['threads'] = int(threads)
        dfs.append(df)

merged_df = pd.concat(dfs, ignore_index=True)
sizes = merged_df['size'].unique()
```

5.1 Single Thread

5.1.1 Data Consistency Analysis

Expectation

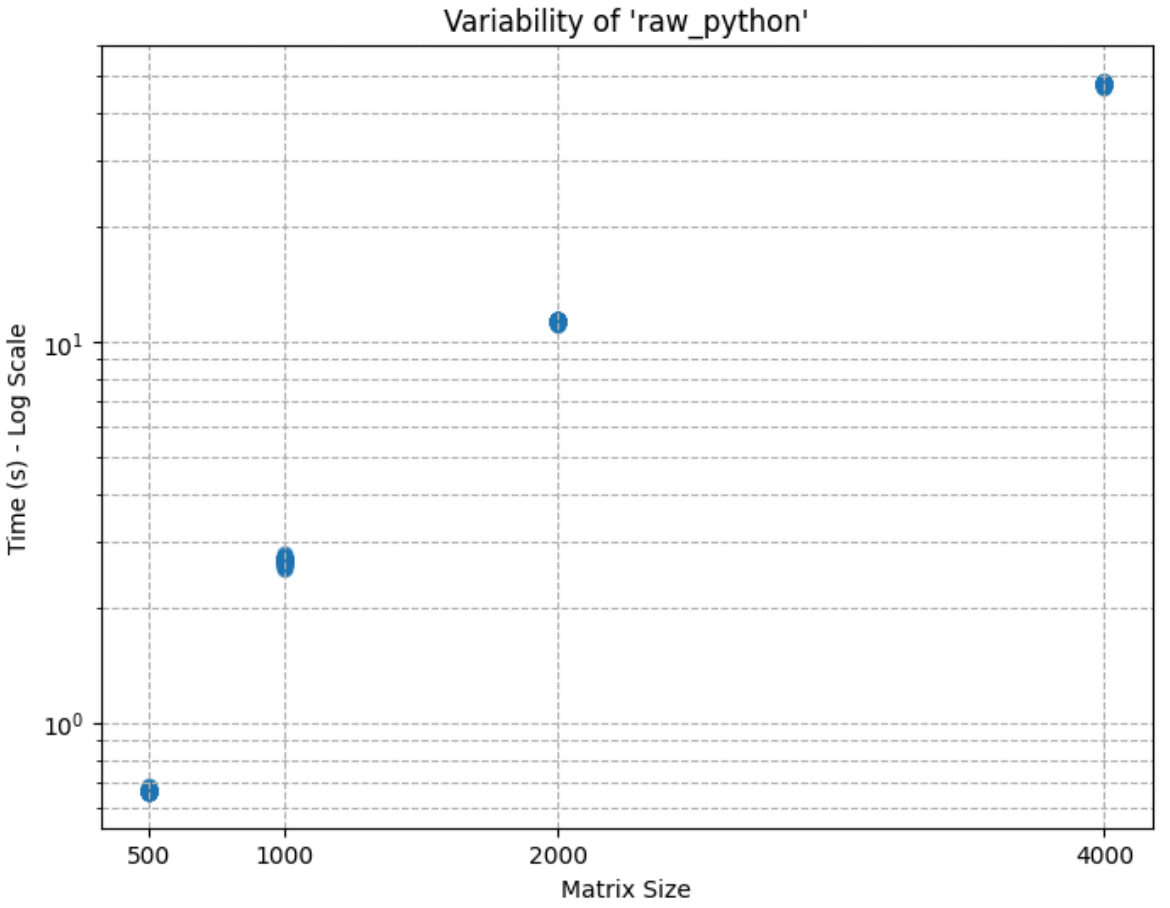
- The raw_python implementation is expected to be very consistent due to its simple implementation and execution path is predictable.
- The numpy and pytorch implementations are expected to be very consistent as well but slightly more variable than raw_python. They use OpenBLAS or MKL implementations of the LAPACK standard, which are multithreaded and processor dependent (The NumPy Developers, 2025 & The PyTorch Developers, 2021). Some warm-up are expected.

Result

In [152…
```
single_thread_df = merged_df[merged_df["threads"] == 1]
```

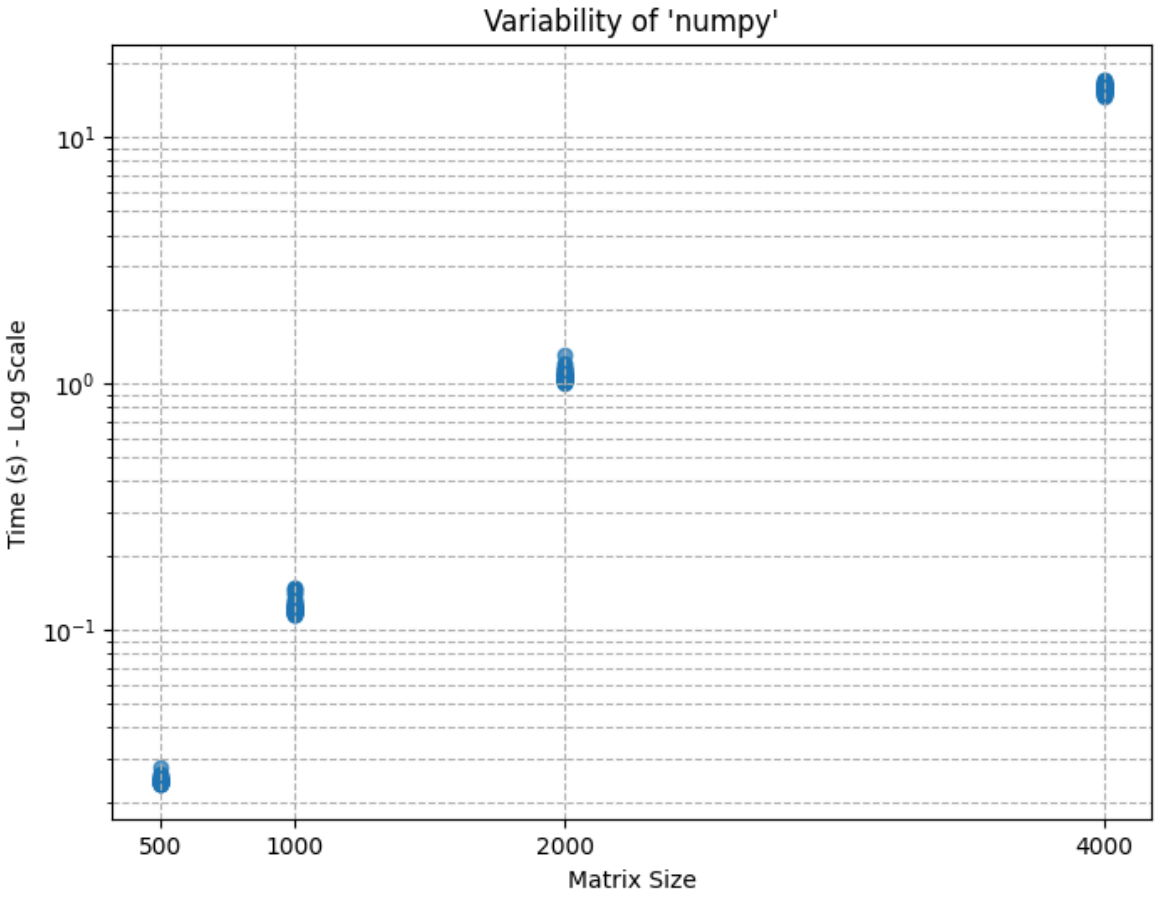
In [153…
```
impl_name = "raw_python"
subset = single_thread_df[single_thread_df["implementation"] == impl_name

plt.figure(figsize=(8, 6))
plt.scatter(subset["size"], subset["time"], alpha=0.7)
plt.title(f"Variability of '{impl_name}'")
plt.xlabel("Matrix Size")
plt.ylabel("Time (s) - Log Scale")
plt.yscale('log')
plt.grid(True, which="both", linestyle='--')
```

```
plt.xticks(sizes)
plt.show()
```



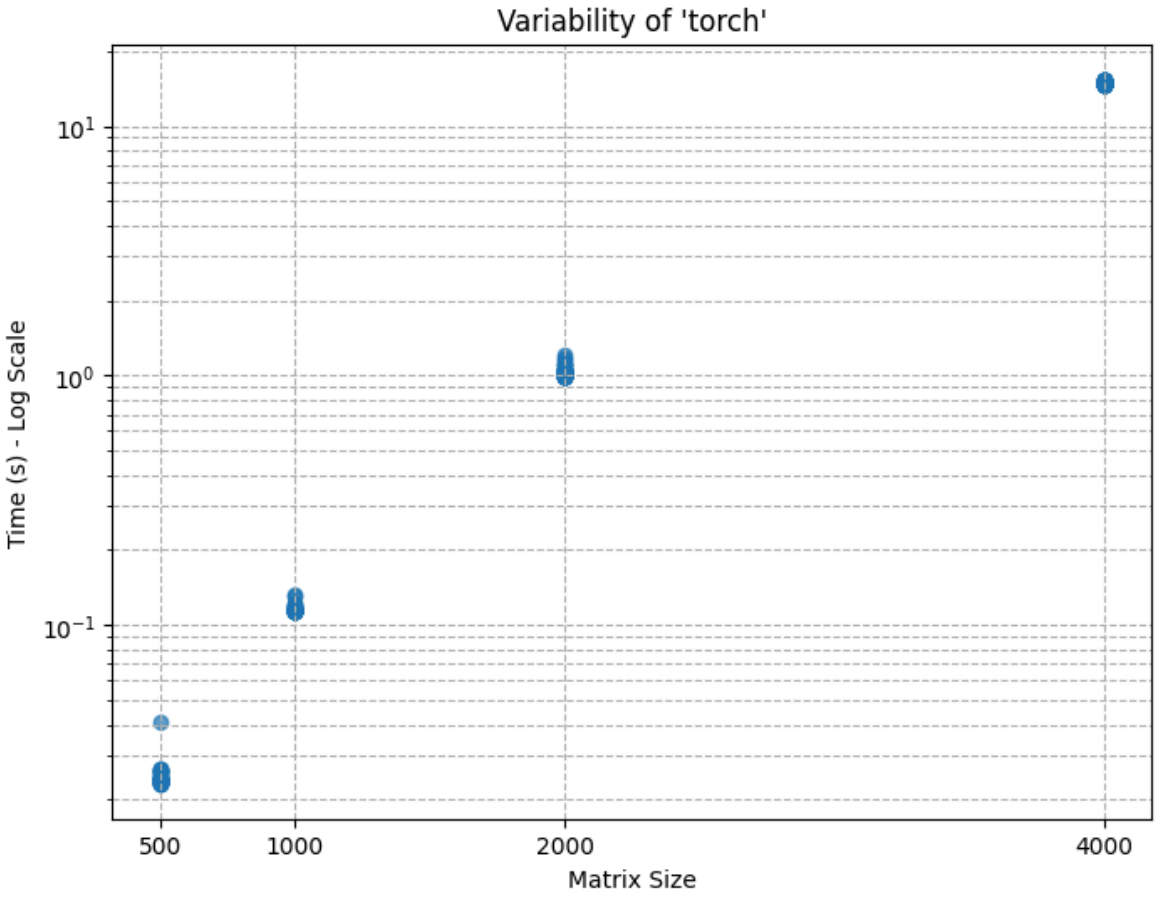
In [154…

```
impl_name = "numpy"
subset = single_thread_df[single_thread_df["implementation"] == impl_name

plt.figure(figsize=(8, 6))
plt.scatter(subset["size"], subset["time"], alpha=0.7)
plt.title(f"Variability of '{impl_name}'")
plt.xlabel("Matrix Size")
plt.ylabel("Time (s) - Log Scale")
plt.yscale('log')
plt.grid(True, which="both", linestyle='--')
plt.xticks(sizes)
plt.show()
```



In [155…

```
impl_name = "torch"
subset = single_thread_df[single_thread_df["implementation"] == impl_name

plt.figure(figsize=(8, 6))
plt.scatter(subset["size"], subset["time"], alpha=0.7)
plt.title(f"Variability of '{impl_name}'")
plt.xlabel("Matrix Size")
plt.ylabel("Time (s) - Log Scale")
plt.yscale('log')
plt.grid(True, which="both", linestyle='--')
plt.xticks(sizes)
plt.show()
```



Conclusion

- For raw_python implementation, the data points for each matrix size are tightly clustered. This show a strong consistency that the result remains almost identical for all test runs of the same matrix sizes.
- For numpy implementation, the plot shows a minor variability for all matrix sizes. The execution time is slightly inconsistent across all test runs, especially for the 2000 matrix.
- For pytorch implementation, the plot shows variability for the 500 and 2000 matrix sizes, while the 1000 and 4000 matrix sizes are tightly clustered. In 500 matrix size, one test run takes longer than the rest is a warm up run.
- This variability is due to the high-performance code on modern hardware that even without changing any code parameters, factors outside the code's direct control such as CPU state or OS scheduling introduce a small amount of non-determinism in the execution time.
- raw_python is the most consistent among all the three implementations.

5.1.2 Performance Analysis

Expectation

- Since numpy and torch use OpenBLAS or MKL implementations of the LAPACK standard (The NumPy Developers, 2025 & The PyTorch Developers, 2021), they would significantly outperform the raw_python implementation in terms of execution time, especially as the size of the input matrix increases.

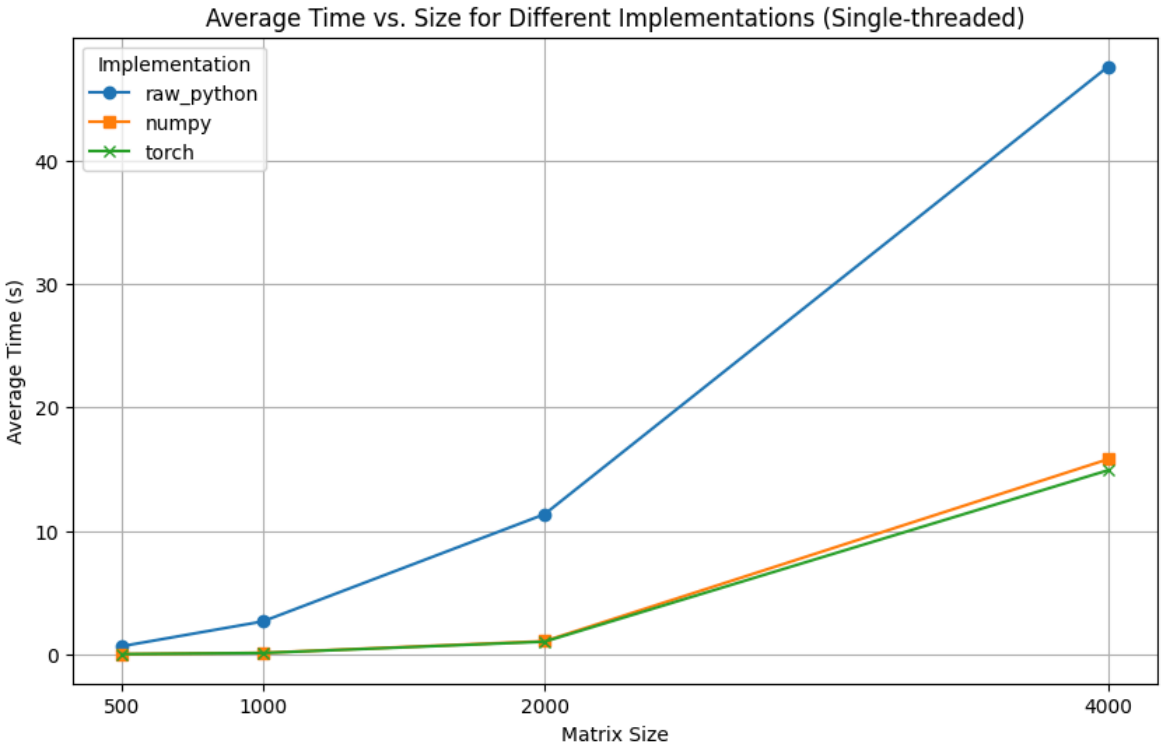
- It is expected that NumPy and PyTorch would have similar performance on CPU as both rely on the same backend libraries.

Result

In [156…
```
summary = single_thread_df.groupby(["implementation", "size"]).agg({
    "time": "mean",
    "frobenius_error": "mean",
    "relative_error": "mean"
}).reset_index()

summary = summary.sort_values(by=["size", "implementation"])
implementation_order = ["raw_python", "numpy", "torch"]

summary["implementation"] = pd.Categorical(summary["implementation"], cat
summary = summary.sort_values(by=["size", "implementation"])
print(summary)
```

```
   implementation  size       time  frobenius_error  relative_error
4      raw_python   500   0.665647       144.055514        0.498730
0           numpy   500   0.024520       144.055514        0.498730
8           torch   500   0.024610       144.055514        0.498730
5      raw_python  1000   2.671522       288.295932        0.499129
1           numpy  1000   0.124485       288.295932        0.499129
9           torch  1000   0.115859       288.295932        0.499129
6      raw_python  2000  11.351349       577.027958        0.499721
2           numpy  2000   1.072373       577.027958        0.499721
10          torch  2000   1.033268       577.027958        0.499721
7      raw_python  4000  47.591475      1154.324588        0.499823
3           numpy  4000  15.806322      1154.324588        0.499823
11          torch  4000  14.910589      1154.324588        0.499823
```

In [157…
```
markers = {"raw_python": "o", "numpy": "s", "torch": "x"}
plt.figure(figsize=(10, 6))

for implementation_name in summary["implementation"].unique():
    subset = summary[summary["implementation"] == implementation_name]
    plt.plot(subset["size"], subset["time"], marker=markers.get(implement

plt.xlabel("Matrix Size")
plt.ylabel("Average Time (s)")
plt.title("Average Time vs. Size for Different Implementations (Single-th
plt.legend(title="Implementation")
plt.grid(True)
plt.xticks(sizes)
plt.show()
```



In [158…

```
baseline_times = summary[summary['implementation'] == 'raw_python'].set_i

summary['baseline_time'] = summary['size'].map(baseline_times)
summary['speedup_vs_python'] = summary['baseline_time'] / summary['time']

columns_to_drop = ['frobenius_error', 'relative_error', 'baseline_time']
summary_cleaned = summary.drop(columns=columns_to_drop)

print(summary_cleaned.to_string(formatters={'speedup_vs_python': '{:,.1f}
```

```
   implementation  size       time speedup_vs_python
4      raw_python   500   0.665647              1.0x
0           numpy   500   0.024520             27.1x
8           torch   500   0.024610             27.0x
5      raw_python  1000   2.671522              1.0x
1           numpy  1000   0.124485             21.5x
9           torch  1000   0.115859             23.1x
6      raw_python  2000  11.351349              1.0x
2           numpy  2000   1.072373             10.6x
10          torch  2000   1.033268             11.0x
7      raw_python  4000  47.591475              1.0x
3           numpy  4000  15.806322              3.0x
11          torch  4000  14.910589              3.2x
```

Conclusion

- Both numpy and pytorch implementations are significantly faster than raw_python
  - Approximately 27 times faster for the 500 matrix size
  - Approximately 23 times faster for the 1000 matrix size
  - Approximately 11 times faster for the 2000 matrix size
  - Approximately 3 times faster for the 4000 matrix size
- As the matrix size increases, the execution time for raw_python grows much faster than numpy and pytorch, indicating its inefficiency for large matrix or

problems.

- numpy and pytorch perform almost identically in terms of execution time. However, pytorch is very slightly better as the matrix size increases.
- Overall, these results demonstrate the necessity of using optimized libraries like NumPy or PyTorch for computationally intensive tasks such as SVD, particularly with larger matrix or datasets.

5.1.3 Accuracy Analysis

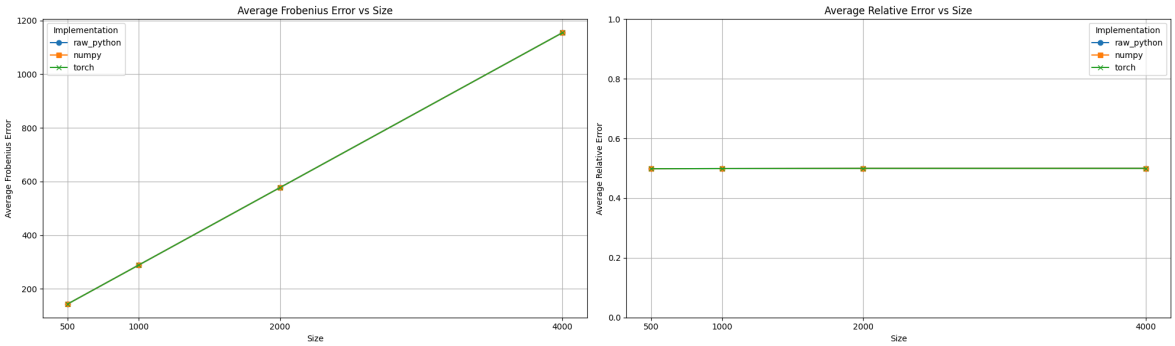
Expectation

- The numpy and pytorch implementations are expected to have lower frobenius error and relative error compared to the raw Python implementation because they use standard linear algebra libraries for the computation, which should provide more accurate results.
- It is expected that the accuracy of NumPy and PyTorch would be the same as both rely on the same backend libraries.

Result

In [159…

```
fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, figsize=(20, 6))

for implementation_name in summary["implementation"].unique():
    subset = summary[summary["implementation"] == implementation_name]
    ax1.plot(subset["size"], subset["frobenius_error"], marker=markers.ge

ax1.set_title('Average Frobenius Error vs Size')
ax1.set_xlabel('Size')
ax1.set_ylabel('Average Frobenius Error')
ax1.set_xticks(sizes)
ax1.grid(True)
ax1.legend(title='Implementation')

for implementation_name in summary["implementation"].unique():
    subset = summary[summary["implementation"] == implementation_name]
    ax2.plot(subset["size"], subset["relative_error"], marker=markers.get

ax2.set_title('Average Relative Error vs Size')
ax2.set_xlabel('Size')
ax2.set_ylabel('Average Relative Error')
ax2.set_xticks(sizes)
ax2.grid(True)

ax2.set_ylim(0, 1)
ax2.legend(title='Implementation')

plt.tight_layout()
```



Conclusion

- The two graphs contradicts the expectation that the raw_python implementation would be less accurate. All three implementations have the same frobenius error and relative error for all matrix sizes.
- As the matrix size increases, the relative error remains almost the same and still below 0.50 for all implementations.

5.2 Multi Threads

5.2.1 Performance Analysis

Expectation

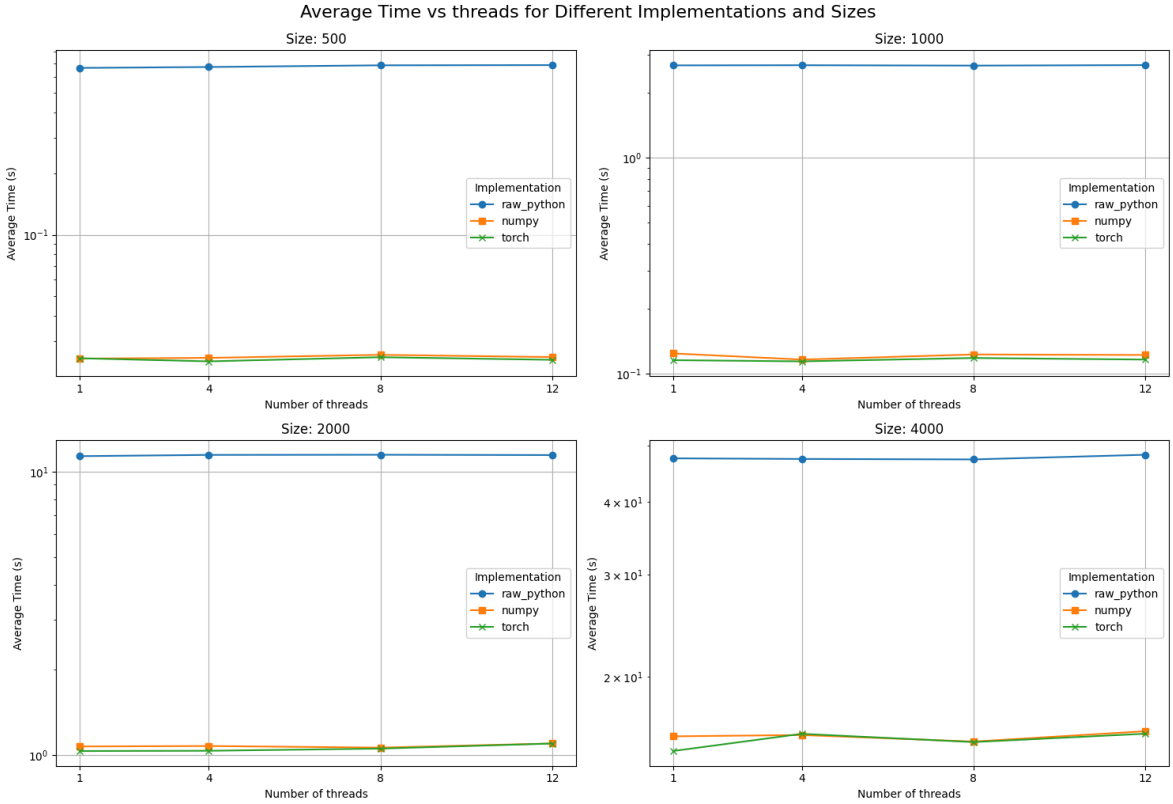
- The raw_python implementation is expected to not have any performance improvement with more threads due to Python's Global Interpreter Lock (GIL), which allows only one thread can execute Python code at once (The Python Developers, 2025). So, this cannot achieve true parallelism across multiple threads.
- The numpy and pytorch implementation are expected to see performance improvement as the number of threads increases because their backend libraries are designed for parallel execution and support multi-threading computation.

Result

In [163…
```
thread_counts = merged_df["threads"].unique()
multi_thread_df = merged_df[merged_df["threads"].isin(thread_counts)]
multi_thread_summary = multi_thread_df.groupby(["implementation", "size",
    "time": "mean",
}).reset_index()

multi_thread_summary["implementation"] = pd.Categorical(multi_thread_summ
multi_thread_summary = multi_thread_summary.sort_values(by=["size", "thre
```

In [164…
```
plt.figure(figsize=(15, 10))

num_rows = (len(sizes) + 1) // 2
sizes.sort()
for i, s in enumerate(sizes):
    plt.subplot(num_rows, 2, i + 1)
    size_subset = multi_thread_summary[multi_thread_summary["size"] == s]
```

```
    for impl in implementation_order:
        impl_subset = size_subset[size_subset["implementation"] == impl]
        plt.plot(impl_subset["threads"], impl_subset["time"], marker=mark

    plt.xlabel("Number of threads")
    plt.ylabel("Average Time (s)")
    plt.title(f"Size: {s}")
    plt.xticks(thread_counts)
    plt.legend(title="Implementation")
    plt.grid(True)
    plt.yscale('log')

plt.tight_layout()
plt.suptitle("Average Time vs threads for Different Implementations and S
plt.show()
```



In [ ]:
```
baseline_time = multi_thread_summary[multi_thread_summary['threads'] == 1
baseline_time = baseline_time.rename(columns={'time': 'baseline_time'})

summary_with_baseline = pd.merge(multi_thread_summary, baseline_time, on=
summary_with_baseline['speedup'] = summary_with_baseline['baseline_time']

speedup_table = summary_with_baseline.pivot_table(
    index=['size', 'implementation'],
    columns='threads',
    values='speedup',
    observed=False
)

speedup_table.columns.name = 'Threads'
speedup_table = speedup_table.round(2)

print("--- Speedup Summary ---")
print(speedup_table)
```

```
--- Speedup Summary ---
Threads                1     4     8     12
size implementation
500  raw_python      1.0  0.99  0.97  0.97
     numpy           1.0  0.99  0.96  0.98
     torch           1.0  1.04  0.99  1.02
1000 raw_python      1.0  1.00  1.00  1.00
     numpy           1.0  1.07  1.01  1.02
     torch           1.0  1.01  0.98  0.99
2000 raw_python      1.0  0.99  0.99  0.99
     numpy           1.0  1.00  1.01  0.98
     torch           1.0  1.00  0.98  0.94
4000 raw_python      1.0  1.00  1.00  0.99
     numpy           1.0  0.99  1.02  0.98
     torch           1.0  0.93  0.97  0.93
```

Conclusion

- The results contradict the expectations for the optimized libraries. The raw_python implementation shows no speedup from multi-threading as expected, while numpy and pytorch's performance even get worse in some cases as the number of thread increases.
- There are two reasons for this performance degradation:
  - **Automatic Parallelization by the Accelerate Framework**: According to the Apple's Developer Documentation (2025), it states that "Accelerate's BLAS and LAPACK implementations abstract the processing capability of the CPU so code written for them will execute the appropriate instructions for the processor available at runtime. This means that both BLAS and LAPACK are optimized for high performance and low-energy consumption.". This means the framework already be optimizing execution by default, even in a "single-thread" benchmark.
  - **Thread Management Overhead**: Since the computation was already being handled efficiently at a low level, adding more threads at the application level introduced unnecessary overhead.

6. Limitations

- **Hardware Constraints**: All experiments were conducted on a single hardware configuration (Apple M2 Pro, CPU only). Results may differ on systems with different CPUs, more cores, or on GPU, especially for PyTorch.
- **Fixed Data Size Range**: All tested matrices are square matrices with fixed sizes (500, 1000, 2000, 4000). Non-square matrices such as Tall & Skinny (m > n) or Short & Fat (m < n) matrices may result in different trends.

References

- The Apple Developers. (2025). Accelerate. Apple. Retrieved from https://developer.apple.com/accelerate/
- The NumPy Developers. (2025). numpy.linalg. NumPy. Retrieved from https://numpy.org/doc/stable/reference/routines.linalg.html
- The NumPy Developers. (2025). numpy.linalg.svd. NumPy. Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.linalg.svd.html
- The PyTorch Developers. (2021). torch.linalg.autograd. PyTorch. Retrieved from https://pytorch.org/blog/torch-linalg-autograd/
- The PyTorch Developers. (2025). torch.linalg.svd. PyTorch. Retrieved from https://docs.pytorch.org/docs/stable/generated/torch.linalg.svd.html
- The Python Developers. (2025). Python. Python Software Foundation. Retrieved from https://docs.python.org/3/library/threading.html
- The Singular Value Decomposition (SVD). (2016). [Lecture Notes]. MIT Course 18.095, Mathematics Lecture Series. Retrieved from https://math.mit.edu/classes/18.095/2016IAP/lec2/SVD_Notes.pdf

Note: AI was used to assist with citation formatting, graph visualization in Section 5, and parts of the benchmark.py script.