Project Report

Spectral Computations

Table of Contents

Prepared by:
Ralph Khairallah

Date: June 4, 2025

1 Introduction

1.1 Context

Numerical methods for computing eigenvalues play a fundamental role in many fields, including engineering, computational physics, and data analysis. These methods make it possible to extract essential features from large matrices without performing a full spectral decomposition, which is often time-consuming and memory-intensive.

As part of this project, we focus on partial spectrum computation methods, particularly in the context of image compression. The objective is to implement several variants of iterative algorithms and compare their numerical efficiency.

1.2 Reminders and Introduction

In many data analysis problems, we are only interested in a few dominant eigenpairs of a matrix, without computing the full spectral decomposition. For example, the PCA method studied in data analysis, and in this project we will see the case of image compression. The basic power method computes a dominant eigenpair but takes time to converge, as it starts from scratch to compute the remaining (non-dominant) eigenvalues.

In this project, we will study several versions of methods to deal with this problem. We will examine two versions of the power method: `v11` and `v12`, and four variants of the subspace iteration method: `v0`, `v1`, `v2`, and `v3`. All will be compared to MATLAB's classical `eig` method.

In Part 1, we implement all these algorithms and compare their performance on matrices of different sizes and types, to draw conclusions about the most efficient algorithms and the contexts in which they perform best.

In Part 2, we will use the algorithms implemented in Part 1 in an image compression pipeline, and similarly try to compare their performance on various images.

1.3 Objectives

1. **Algorithms**
   - Implement `power_v12` (one matrix-vector product per iteration), which is an improved version of `power_v11`, and the four methods `subspace_iter_v0`, v1, v2, and v3.
   - Complete the algorithm files as well as the test files to compare the different methods.
2. **Numerical Study**
   - Compare the algorithms with each other and present them in tables.
   - Measure the execution time of each algorithm for different types and sizes of matrices, and compute the number of iterations needed to reach convergence.
3. **Comparative Analysis**
   - Identify the types of matrices that accelerate or slow down convergence.

- Identify the algorithms that offer the best trade-off in execution time, number of iterations, and accuracy — for example, the impact of the parameter p on convergence.

2 Subspace Iteration Method

2.1 Limitations of the Power Iteration Method

Question 1: Comparison of execution times between MATLAB's `eig` (full spectral decomposition) and `power_v11` (power iteration with deflation).

Type 1 — Eigenvalues $D(i) = i$

Size	Method	Time (s)	# EigPairs	Eigpair Quality	Eigval Quality
100	eig	0.190	-	[5.94e-16, 3.85e-14]	[0, 7.36e-15]
	power_v11	0.160	6	[9.94e-09, 1.42e-08]	[0, 9.95e-15]
500	eig	0.280	-	[6.18e-16, 2.25e-13]	[0, 1.50e-13]
	power_v11	146.5	26	[1.14e-16, 1.64e-08]	[–, 5.03e-14]
1000	eig	1.150	-	[5.74e-16, 5.54e-13]	[0, 1.33e-13]
	power_v11	fail	—	[0, 0]	[0, 0]

Table 1: Results for Type 1 matrices

Comments: In this case, we clearly see that `power_v11` converges quickly for small matrices ($n = 100$). But for $n = 500$, the method becomes much slower, and completely fails for $n = 1000$. On the other hand, `eig` remains very fast and accurate.

Type 2 — Eigenvalues $D(i) = random(1/cond, 1)$

Size	Method	Time (s)	# EigPairs	Eigpair Quality	Eigval Quality
100	eig	0.030	-	[2.63e-16, 5.04e-07]	[0, 1.70e-07]
	power_v11	0.010	1	[7.14e-09, 7.14e-09]	[4.38e-16]
500	eig	0.230	-	[5.12e-16, 3.45e-06]	[0, 2.70e-07]
	power_v11	0.610	2	[9.96e-09, 1.53e-08]	[6.18e-16, 1.25e-15]
1000	eig	1.030	-	[5.34e-16, 1.31e-06]	[0, 2.12e-07]
	power_v11	5.89	4	[9.98e-09, 1.49e-08]	[1.24e-15, 6.97e-15]

Table 2: Results for Type 2 matrices

Comments: Here, `power_v11` manages to extract dominant eigenvalues accurately. As matrix size increases, the number of iterations increases too, raising execution time. `eig` is still more accurate.

Type 3 — Eigenvalues $D(i) = cond^{(-(i-1)/(n-1))}$

Size	Method	Time (s)	# EigPairs	Eigpair Quality	Eigval Quality
100	eig	0.040	-	[4.00e-16, 1.94e-11]	[0, 5.76e-12]
	power_v11	0.010	1	[9.38e-09, 9.38e-09]	[8.88e-16]
500	eig	0.240	-	[5.89e-16, 3.15e-11]	[0, 1.06e-11]
	power_v11	2.46	5	[9.83e-09, 1.44e-08]	[0, 4.33e-15]
1000	eig	1.020	-	[5.40e-16, 5.14e-11]	[0, 1.14e-11]
	power_v11	25.99	10	[9.94e-09, 1.43e-08]	[0, 8.33e-15]

Table 3: Results for Type 3 matrices

Comments: Here, we see that the closer the eigenvalues are, the more iterations are required. `power_v11` is faster for small matrices, but overall `eig` is faster and more stable.

Type 4 — Eigenvalues $D(i) = 1 - ((i-1)/(n-1)) * (1 - 1/cond)$

Size	Method	Time (s)	# EigPairs	Eigpair Quality	Eigval Quality
100	eig	0.040	-	[6.49e-16, 5.24e-14]	[0, 1.27e-14]
	power_v11	0.090	6	[9.94e-09, 1.42e-08]	[1.16e-16, 9.77e-15]
500	eig	0.250	-	[5.52e-16, 4.52e-14]	[0, 1.87e-14]
	power_v11	173.1	26	[9.99e-09, 1.64e-08]	[0, 5.04e-14]
1000	eig	1.100	-	[4.95e-16, 6.49e-14]	[0, 1.51e-14]
	power_v11	fail	—	[0, 0]	[0, 0]

Table 4: Results for Type 4 matrices

Comments: The method `power_v11` becomes unstable for $n = 1000$ and very slow starting from $n = 500$. `eig` remains fast and stable for any matrix.

General Conclusion

We clearly see that the **`eig` Method** is generally fast, stable, and accurate for any matrix size and type.

This is not the case for the **`power_v11` Method**, which is acceptable only for small matrices with a well-separated spectrum. It is often very slow or fails to converge, which makes the `eig` method more advantageous.

Finally, the type of matrix used is very important and must be considered, as it strongly influences the performance of the methods. For example, for ill-conditioned spectra (Types 2, 3, 4), the `power_v11` method performs poorly.

Question 2: Comparison of execution times between `power_v11` and improved power iteration `power_v12`

In this part, we aim to improve Algorithm 1 to avoid computing the product Av twice inside the loop. This greatly reduces execution time when matrix size increases. We obtain the following improved algorithm:

Algorithm 1 Improved Power Iteration Method

Input: Matrix $A \in R^{n \times n}$
Output: (λ_1, v_1) eigenpair associated to the largest (in modulus) eigenvalue
$v \in R^n$ given
$z \leftarrow A \cdot v$
$\beta \leftarrow v^T \cdot z$
$v \leftarrow z/\|z\|$
$\beta_{\text{old}} \leftarrow \beta$
$z \leftarrow A \cdot v$
$\beta \leftarrow v^T \cdot z \; |\beta - \beta_{\text{old}}|/|\beta_{\text{old}}| < \varepsilon$
$\lambda_1 \leftarrow \beta$ and $v_1 \leftarrow v$

Thus, here is the modification we make in the Matlab code:

Listing 1 Improved Power Iteration Method (power_v12.m)

```
while (~convg && k < m)
    k = k + 1;

    % Improved power iteration method
    v = randn(n,1);
    z = A*v;
    beta = v'*z;

    % conv = || beta * v - A*v||/|beta| < eps
    % see section 2.1.2 of the instructions
    norme = norm(beta*v - z, 2)/norm(beta,2);
    nb_it = 1;

    while (norme > eps && nb_it < maxit)
        y = z;
        v = z / norm(z,2);
        beta = v'*z;
        norme = norm(beta*v - z, 2)/norm(beta,2);
        nb_it = nb_it + 1;
    end
```

With this algorithm, here are the results obtained for different matrix types and sizes:

Size	Type	Time `power_v11` (s)	Time `power_v12` (s)	Speed-up
100	1	5.763e-02	9.850e-03	5.85x
500	1	1.116e+01	2.856e-02	390.66x
1000	1	1.604e+00	5.972e-02	26.86x
100	2	1.184e-02	6.782e-03	1.75x
500	2	6.200e-02	2.146e-02	2.89x
1000	2	4.817e-01	6.080e-02	7.92x
100	3	1.174e-02	8.169e-03	1.44x
500	3	1.603e-01	2.174e-02	7.38x
1000	3	2.169e+00	6.294e-02	34.47x
100	4	4.959e-02	1.003e+00	4.94x
500	4	1.200e+01	2.744e-02	437.37x
1000	4	1.646e+00	6.431e-02	25.60x

Table 5: Execution time comparison between `power_v11` and `power_v12` depending on matrix size and type

Comment: The improved power iteration method accelerates convergence by avoiding redundant matrix multiplications at each iteration. This becomes particularly beneficial for large matrices. As we can see, `power_v12` is between 1.4× and up to 400× faster than `power_v11` in some cases.

Question 3: Main drawback of the power iteration method with deflation in terms of computation time.

The main drawback of the power iteration method with deflation (implemented in `power_v11`) is that it restarts a full power loop for each eigenpair, while subsequent eigenvalues are typically harder to isolate due to spectral closeness. As a result:

- The method does not scale well when extracting a large number of eigenvalues.
- It becomes very slow for large matrices, since each iteration requires a matrix-vector multiplication, and convergence is sometimes not reached.
- Numerical errors accumulate progressively, leading to a loss of precision and orthogonality.

2.2 Extension of the Power Iteration Method

Question 4: Convergence matrix V of the power iteration method when applied to multiple vectors m.

If we apply the power iteration method simultaneously to a matrix V of m vectors without enforcing orthonormalization, the columns of V will not converge to the m eigenvectors of A. Instead, they will all tend to align with the dominant eigenvector (associated with the eigenvalue of largest modulus).

In this case, the matrix V will be almost rank 1, and we won't be able to extract multiple distinct eigenpairs.

Therefore, the subspace iteration method enforces orthonormalization at each iteration, which keeps the columns of V independent and allows convergence to the dominant eigenspace of dimension m.

Question 5:

In the provided subspace iteration algorithm, we compute the full spectral decomposition of $H = V^T AV$. This is not a problem because:

- H is of size $m \times m$, where m is the number of vectors in the subspace and $m \ll n$.
- Diagonalizing H, which has complexity $\mathcal{O}(m^3)$, is negligible compared to $A \cdot V$, which has complexity $\mathcal{O}(n^2 m)$;

So computing the full spectral decomposition of H is not problematic since H is a small matrix compared to A. This allows efficient and fast computation, while extracting the dominant eigenvalues and eigenvectors of A.

Question 6: Comparison of execution times and iteration numbers for `subspace_iter_v0` and `subspace_iter_v1`

Size	Type	Method	Time (s)	# Iterations
100	1	subspace_iter_v0	3.680	1629
100	1	subspace_iter_v1	1.140	87
300	1	subspace_iter_v0	37.47	4869
300	1	subspace_iter_v1	8.77	678
100	2	subspace_iter_v0	0.150	39
100	2	subspace_iter_v1	0.060	4
500	2	subspace_iter_v0	2.840	219
500	2	subspace_iter_v1	1.330	17

Table 6: Comparison of `subspace_iter_v0` and `subspace_iter_v1` for Type 1 and 2 matrices

Comment: We clearly see that version `subspace_iter_v1` is more efficient than the first one, with significantly fewer iterations and faster execution times.

Question 7: Identification of steps in Algorithm 4

Below is the Subspace Iteration Method v1 with Rayleigh-Ritz projection, annotated with blue comments indicating the corresponding lines in the `subspace_iter_v1` code.

Algorithm 2 Subspace iteration method v1 with Rayleigh-Ritz projection

Input: Symmetric matrix $A \in R^{n \times n}$, tolerance ε, $MaxIter$ (maximum number of iterations) and $PercentTrace$, the target percentage of the trace of A % lines 21–22
Output: n_{ev} dominant eigenvectors V_{out} and the corresponding eigenvalues Λ_{out} % lines 120–125
Generate an initial set of m orthonormal vectors $V \in R^{n \times m}$; $k = 0$; $PercentReached = 0$ % lines 37–48
repeat
  $k = k + 1$ % line 54
  Compute $Y = A \cdot V$ % line 56
  $V \leftarrow$ orthonormalize columns of Y % line 58
  Apply *Rayleigh-Ritz projection* to matrix A and orthonormal vectors V % line 61
  Convergence analysis step: save converged eigenpairs and update $PercentReached$ % lines 63–115
until ($PercentReached > PercentTrace$ or $n_{ev} = m$ or $k > MaxIter$) % line 52

2.3 `subspace_iter_v2` and `subspace_iter_v3`: toward an efficient solver

Question 8: Computational cost of A^p followed by $A^p \cdot V$

Computing A^p followed by $A^p \cdot V$ is costly in terms of flops. If A is an $n \times n$ matrix and V is $n \times m$, then calculating A^p requires $(p-1)$ matrix-matrix multiplications, each with complexity $\mathcal{O}(n^3)$ flops.

Then, multiplying A^p by V takes another $\mathcal{O}(n^2 m)$ flops. So the total cost is dominated by $\mathcal{O}(pn^2 m)$, especially when $m \ll n$.

An alternative approach avoids explicitly computing A^p. Instead, we compute $A^p \cdot V$ iteratively by chaining p matrix-vector multiplications as follows:

$$V \leftarrow A \cdot (A \cdot (\cdots (A \cdot V) \cdots)).$$

In this case, we perform p matrix multiplications, each costing $\mathcal{O}(n^2 m)$ flops, giving a total of $\mathcal{O}(pn^2 m)$ flops. This avoids computing and storing A^p, which could be burdensome.

Question 10: Impact of parameter p

Method	Matrix Size	p (for v2)	Time (s)	Iterations
v0	100	-	2.46	1629
v1	100	-	0.58	87
v2	100	5	0.64	18
v2	100	10	0.63	9
v2	100	20	0.60	5
v2	100	40	0.70	3
v0	200	-	14.04	3192
v1	200	-	2.24	263
v2	200	5	2.65	53
v2	200	10	1.86	27
v2	200	20	1.80	14
v2	200	40	2.99	7

Table 7: Comparison of 3 subspace iteration methods for $n = 100$ and $n = 200$, matrix of type 1

Comment: From the table, we see that `subspace_iter_v2` requires far fewer iterations than `subspace_iter_v0` and `subspace_iter_v1`, thanks to the for-loop introduced to compute $A^p \cdot V$.
Also, for $n = 100$ and $n = 200$, as p increases, the number of iterations decreases, reaching convergence more quickly.
However, if p becomes too large, computation becomes heavy and costly per iteration, possibly increasing total time or preventing convergence. Thus, it is advisable to choose p in the range 5–50, where both runtime and convergence are optimal.

Question 11: Precision of `subspace_iter_v1` and `subspace_iter_v2`

In both `subspace_iter_v1` and `subspace_iter_v2`, we observe that the precision of the eigenpairs—measured via $\|Av - \beta v\|/|\beta|$—varies across the eigenvectors.

Indeed, in `subspace_iter_v1`, convergence does not occur at the same rate for all eigenvectors. The method first captures eigenvectors associated with the largest eigenvalues in modulus, so those dominant vectors converge quickly.

As a result, the remaining eigenvectors in V (associated with smaller eigenvalues) will converge more slowly and with lower precision.

The same applies to `subspace_iter_v2`. Although the A^p acceleration speeds up general convergence, it still does not distinguish already converged dominant eigenvectors from remaining ones. This may cause divergence or poor accuracy for some eigenpairs in certain cases.

Question 12: Anticipation of eigenpairs in `subspace_iter_v3`

In `subspace_iter_v3`, the idea is to isolate already converged eigenpairs so they are not recalculated in the next iterations. This avoids re-orthonormalizing and performing

Rayleigh-Ritz projections on already treated vectors.
This should increase both precision and stability, since only unconverged eigenvectors will be modified in the next iterations. It also reduces computational cost, especially when computing a large number of eigenpairs.

2.4 Numerical Experiments

Question 14: Comparison of 4 matrix types

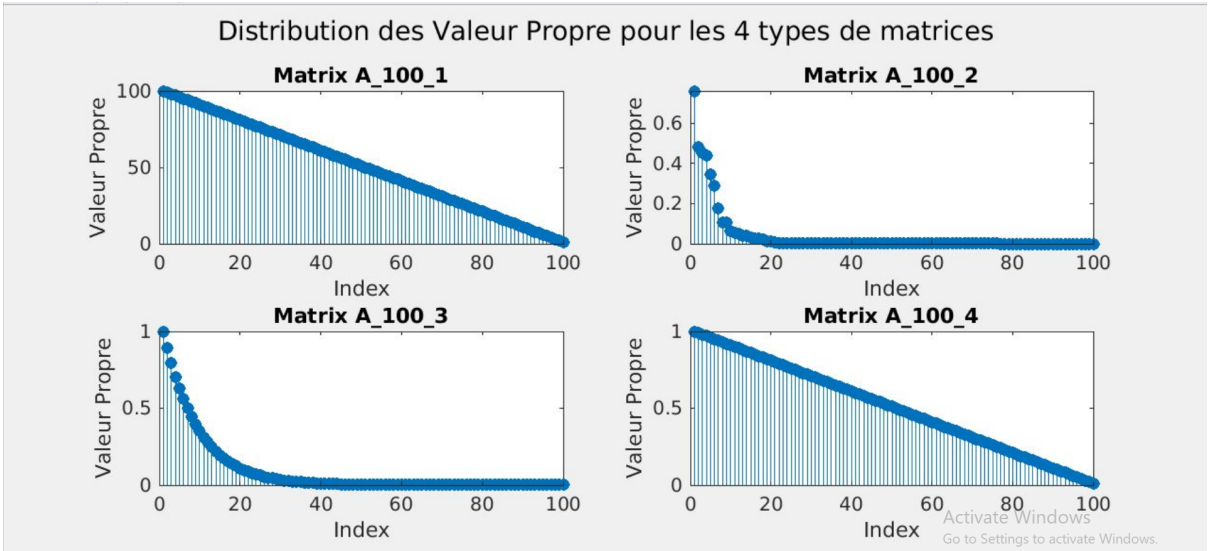

Figure 1: Eigenvalue distributions for the 4 matrix types

This figure shows the eigenvalue distributions for the 4 types of 100×100 matrices:

- **imat = 1**: Eigenvalues decrease linearly from 100 to 1. Convergence is slower than other types.
- **imat = 2**: Large eigenvalues decrease sharply at first, meaning dominant eigenpairs are easily isolated. Fast convergence of dominant eigenvectors.
- **imat = 3**: Similar to imat = 2 but with a more gradual decay.
- **imat = 4**: Similar to imat = 1 but eigenvalues are distributed from 0 to 1.

This highlights how eigenvalue distribution affects convergence speed, which is faster for types 2 and 3.

Question 15: Algorithm performance comparison

The table below presents the performance results for various methods on different matrix types and sizes.

Table 8: Comparison of algorithm performance on different matrix types and sizes

From the table we observe:
Execution time:

- For both matrix sizes ($n = 100$ and $n = 300$), `eig` is much faster than other methods.
- Among subspace methods, `v1` and `v2` are generally the fastest and most consistent.

Convergence:

- `v0` is the slowest to converge and performs a large number of iterations, which becomes problematic as matrix size increases.
- `v1` converges faster than `v0` with significantly fewer iterations.
- `v2` and `v3` reduce the number of iterations even more, with `v2` generally being faster. `v3` shows poorer precision—likely due to the deflation step—as seen in the "Couple Quality" column.

Impact of matrix type:

- For matrix types `imat = 2` and `imat = 3`, subspace methods `v1` and `v2` perform well in terms of speed and precision—possibly because dominant eigenvalues are well separated.
- For types `imat = 1` and `imat = 4`, convergence is more difficult, with longer runtime and lower accuracy.
- `eig` remains fast and effective across all four matrix types.

Precision:

- `v0`, `v1`, and `v2` generally deliver high accuracy (close to machine epsilon).
- `v3` is less accurate, with eigenpair errors often around 10^{-2} or worse.

Conclusion:

- `eig` remains the best in terms of convergence and speed.
- Among subspace methods, `v1` and `v2` offer the best trade-off between convergence speed and precision.

3 Application to Image Compression

3.1 Overview and Introduction

As previously discussed, digital images are represented by large matrices where each element encodes the color or grayscale intensity of a pixel. While this format is useful for image processing, it can also be heavy and require extra processing. In image compression, the goal is to retain the essential information in the matrix to visualize the image while reducing size for storage efficiency.

Several modern methods allow for this, especially *linear and algebraic* techniques. Most images can be accurately represented by a small number of dominant modes (which echoes our earlier work with dominant eigenpairs). To compute these, we extract singular values and dominant eigenvectors from the image matrix.

This section of the report will explore:

1. The concept of low-rank approximation using singular value decomposition and the best approximation theorem.
2. How compression is applied to grayscale comic images by extracting dominant eigenpairs using the algorithms developed in Part 1 and reconstructing the compressed images.

3.2 Image Compression

Question 1: Dimensions of (Σ_k, U_k, V_k)

The dimensions of each element in the triplet are:

- $U_k \in R^{q\times k}$
- $\Sigma_k \in R^{k\times k}$
- $V_k \in R^{p\times k}$

In the case where $q < p$:
Here, the SVD is performed via $M = I^T I \in R^{p\times p}$, which yields:

- $V_k \in R^{p\times k}$
- $U_k \in R^{q\times k}$, computed by:
$$U_k = \frac{1}{\sigma_i} I V_k$$
- $\Sigma_k \in R^{k\times k}$

Thus, the dimensions remain: $U_k \in R^{q\times k}$, $\Sigma_k \in R^{k\times k}$, and $V_k \in R^{p\times k}$.

Question 2: Image Reconstruction

In this section, we aim to reconstruct a compressed image using various eigenvalue computation methods, including `eig`, power iteration methods (`power_v11`, `power_v12`), and the four variants of the subspace iteration method (`v0` to `v3`).

The image used is shown below:

The parameters used for all methods are:

- Tolerance: `eps = 1e-8`
- Maximum number of iterations: `maxit = 10000`
- Search space size: `search_space = 400`
- Target trace percentage: `percentage = 0.99`
- Number of restarts for methods `v2` and `v3`: `puiss = 1`

Visual results using `power_v11`

Below are the reconstructed images using the `power_v11` method for various rank values k:

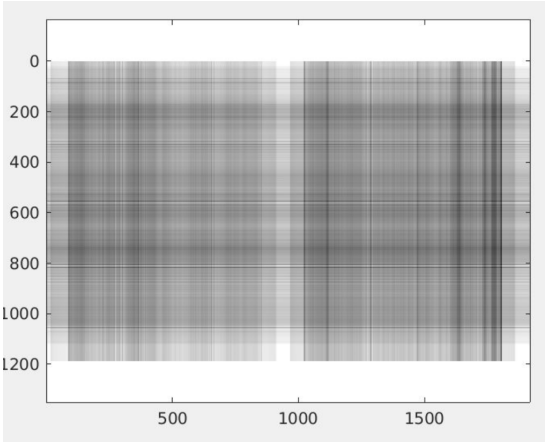

Figure 2: Reconstructed image with $k = 1$

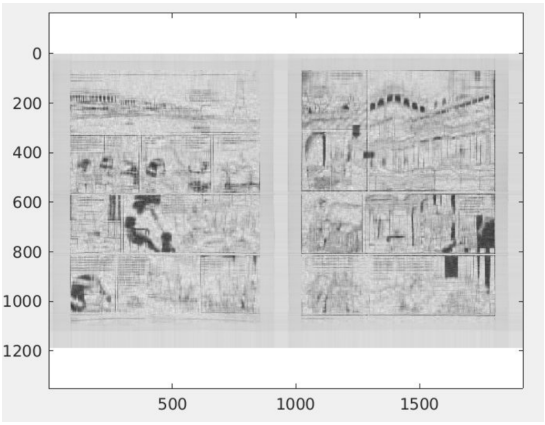

Figure 3: Reconstructed image with $k = 41$

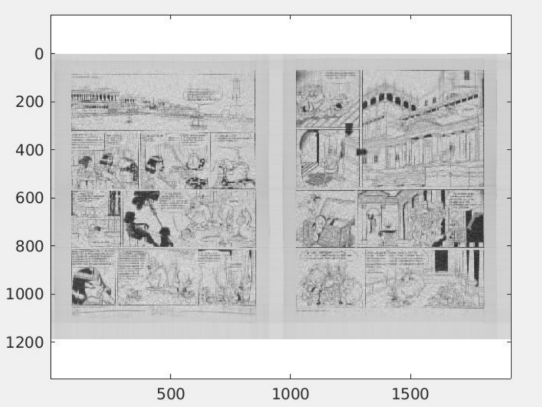

Figure 4: Reconstructed image with $k = 81$

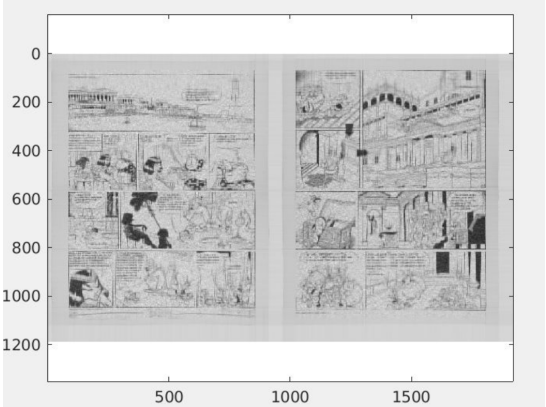

Figure 5: Reconstructed image with $k = 121$

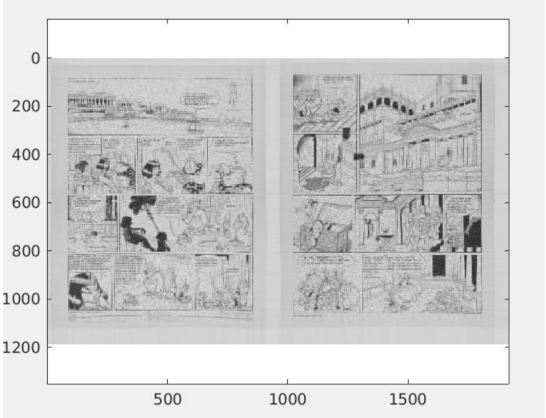

Figure 6: Reconstructed image with $k = 201$

Comparison of image reconstruction across different methods

This section compares the results of image reconstruction using various eigenvalue computation methods. The reconstructed images are shown for each method using specific values of k.

Method `eig`

The `eig` method is a classical approach for computing all eigenvalues and eigenvectors of a matrix. It provides a full spectral decomposition, enabling very high-quality image reconstruction that captures nearly all details from the original image. However, this accuracy comes at a cost: the method requires a high value of k, making it computationally expensive. Therefore, while ideal for high-fidelity reconstructions, it may not be suitable for applications needing fast computations or limited resources.

Figure 7: Reconstruction using `eig`

Method `power_v11`

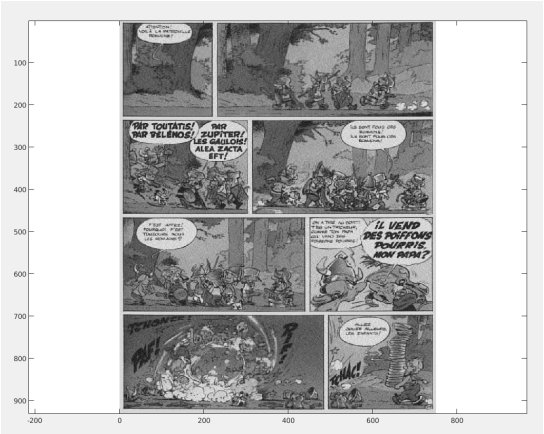

Figure 8: Reconstruction using `power_v11`

Method `power_v12`

The `power_v12` method is an improved version of `power_v11`, optimized to reduce the number of matrix multiplications per iteration.

Figure 9: Reconstruction using `power_v12`

Methods `subspace_iter_v0` to `subspace_iter_v3`

Subspace iteration methods, from `subspace_iter_v0` to `subspace_iter_v3`, represent a more advanced approach for eigenvalue computation. These methods iterate on a subspace of vectors and use techniques such as Rayleigh-Ritz projection to extract dominant eigenvalues. The figures below show the reconstruction results for each version. A progressive improvement in image quality can be seen as the algorithm versions become more advanced. These methods are particularly effective for large matrices and offer a good trade-off between computational cost and reconstruction quality.

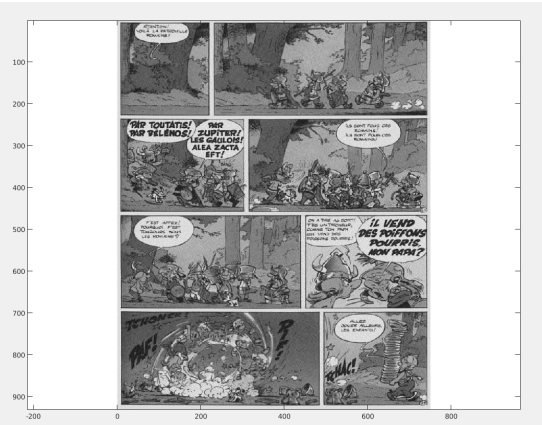

Figure 10: Reconstruction using `subspace_iter_v0`

Figure 11: Reconstruction using `subspace_iter_v1`

Figure 12: Reconstruction using `subspace_iter_v2`

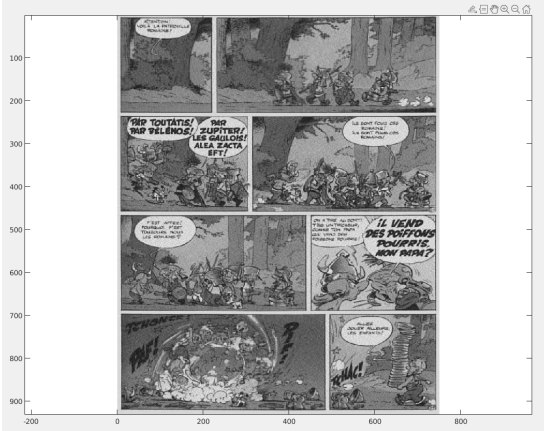

Figure 13: Reconstruction using `subspace_iter_v3`

Comparative analysis

A comparative analysis of the methods shows that `eig` provides the best reconstruction quality but at the cost of a high k. Power methods (`power_v11` and `power_v12`) are limited by lower trace coverage, which slightly affects reconstruction but still yield valid results. In contrast, subspace iteration methods (`subspace_iter_v0` to `subspace_iter_v3`) show steady improvement in reconstruction quality, with the more advanced versions producing better results. These methods are more efficient computationally and offer a good balance between performance and reconstruction quality, making them suitable for practical applications.

RMSE Visualization

We plotted the various **RMSE** values for each method across different values of k. Below are the results:

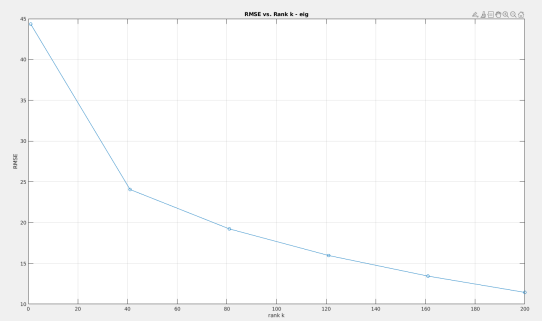

Figure 14: Difference between I and I_k for method `eig`

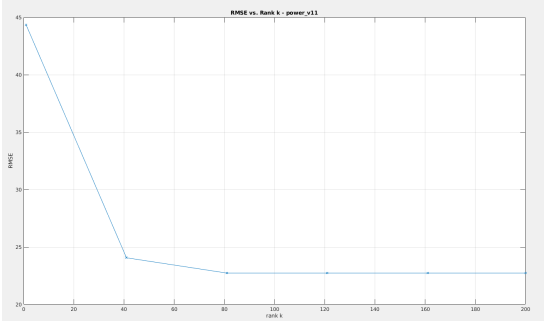

Figure 15: Difference between I and I_k for method `power_v11`

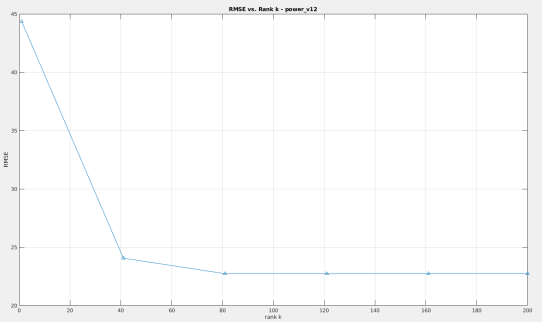

Figure 16: Difference between I and I_k for method `power_v12`

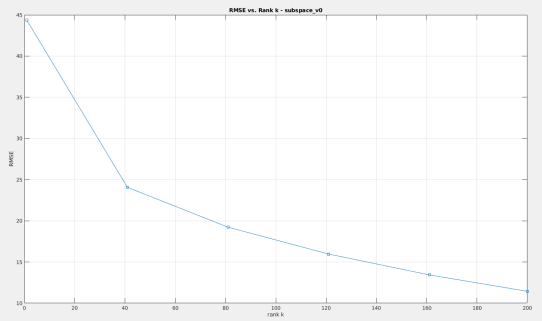

Figure 17: Difference between I and I_k for method `subspace_iter_v0`

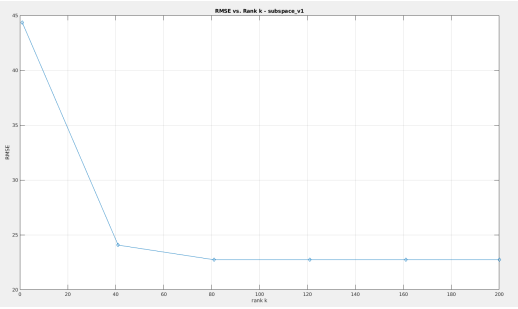

Figure 18: Difference between I and I_k for method `subspace_iter_v1`

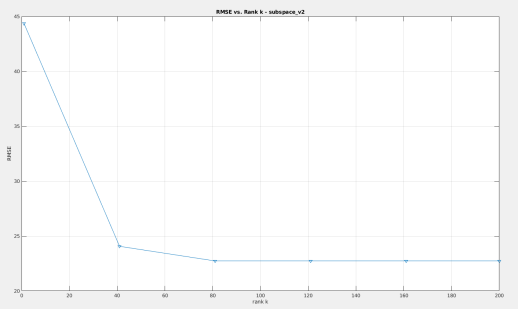

Figure 19: Difference between I and I_k for method `subspace_iter_v2`

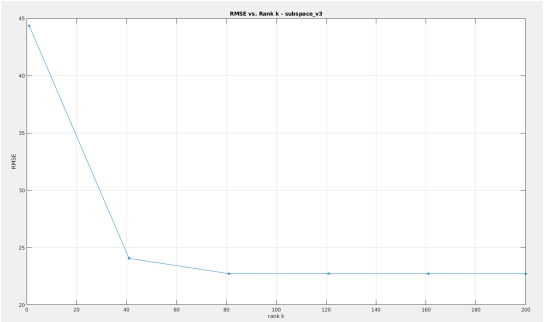

Figure 20: Difference between I and I_k for method `subspace_iter_v3`

Performance Comparison (RMSE). The RMSE curves obtained for the various methods show a similar global trend: a rapid decrease in error at low ranks followed by a plateau. The `eig` method remains the most accurate in terms of reconstruction quality, with a steady RMSE decay even for high values of k. The `power_v11` and `power_v12` methods converge quickly to a plateau but do not reach the same approximation level as `eig`. The `subspace_v1`, `v2`, and `v3` versions offer similar performances, converging to a minimal error as early as $k = 80$, which reflects their efficiency in extracting dominant

modes. In contrast, `subspace_v0` stands out with a slower convergence, similar to `eig`, but without reaching its final precision.

Bonus Question: Color Image Reconstruction

Previously, we attempted grayscale image reconstruction; we now attempt the same task but with the color image shown below:

Figure 21: Colored image before compression

To achieve this, we use the same algorithm as for grayscale image reconstruction, but apply it separately to the three RGB channels, and then merge the three reconstructed images to obtain the final version.

While the implemented methods can reconstruct a color image, their high execution time may hinder practical use cases requiring fast and accurate color image reconstruction.

Moreover, we observed that reconstructing a color image takes considerably more time than a grayscale one, due to the larger data volume and the need to process three matrices instead of one.

Below are reconstruction results using `subspace_iter_v2`:

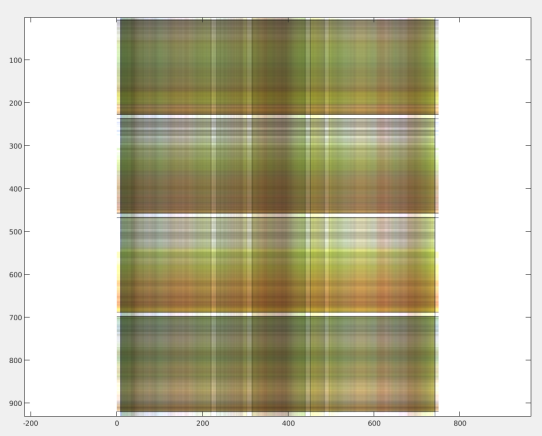

Figure 22: Reconstructed image with $k = 1$

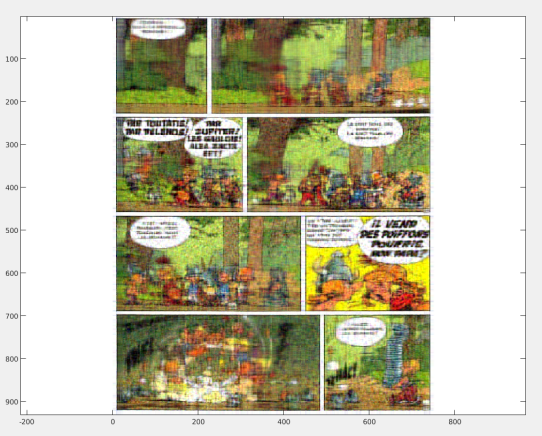

Figure 23: Reconstructed image with $k = 41$

Figure 24: Reconstructed image with $k = 81$

Figure 25: Reconstructed image with $k = 121$

Figure 26: Reconstructed image with $k = 161$

Figure 27: Reconstructed image with $k = 201$

4 Conclusion

In this report, we explored various methods for computing eigenvalues and their application to image compression. We began by examining the limitations of the basic power iteration method and introduced improvements through the enhanced version (`power_v12`). These improvements demonstrated significant gains in execution time, especially for large matrices.

We then studied subspace iteration methods, starting from the basic version (`subspace_iter_v0`) and progressing to more advanced versions (`subspace_iter_v3`). These methods showed increasing efficiency, both in terms of the number of iterations needed and the precision of the computed eigenvalues. Numerical experiments revealed that the subspace iteration methods, particularly the more advanced ones, offer a good trade-off between computational cost and accuracy.

In the image compression application section, we used these methods to reconstruct images from their compressed representations. The results showed that subspace iteration methods allow satisfactory image reconstruction with a reasonable computational cost. The `eig` method, although providing the best reconstruction quality, is significantly more expensive in terms of the required values of k.

For color image reconstruction, we successfully applied most methods, although the power methods `v11` and `v12` struggled. These methods are efficient at extracting dominant singular values but do not ensure optimal image approximation compared to SVD or the three subspace methods, which performed well.

In conclusion, this report highlighted the importance of choosing the appropriate method based on computational constraints and desired quality. Subspace iteration methods, especially the advanced versions, have proven to be effective tools for image compression, offering a solid balance between performance and cost.