Contents

Analysis of Iterative Methods: Jacobi and Gauss-Seidel for Linear Systems

Arthur Rabello Oliveira

March 28, 2025

Abstract

This document presents the implementation of the iterative methods of Jacobi and Gauss-Seidel for solving linear systems, the conduction of tests on 2x2 and 3x3 matrices (including cases of convergence, divergence, and extreme conditions) and a comparative analysis of the results. Code snippets and graphs that illustrate the evolution of the error over the iterations are presented, along with a detailed theoretical analysis (Item E). For full access to the source code, please consult the repository at https://github.com/arthurabello/nla-assignment-1.

1 Introduction

The numerical solution of linear systems is a fundamental problem in various areas of applied mathematics and data science. Iterative methods, such as **Jacobi** and **Gauss-Seidel**, are widely used due to their simplicity and ease of implementation, especially for large-scale systems. However, the behavior of these methods strongly depends on the properties of the system matrix, particularly diagonal dominance.

This report describes the Python implementation of the two methods, presents a comprehensive set of tests (including diagonally dominant, non-dominant, and ill-conditioned matrices), and analyzes the obtained results.

2 Jacobi and Gauss-Seidel Methods (items a-b)

2.1 Jacobi Method

The functions `jacobi_method` and `gauss_seidel_method` were implemented in the `methods` package. Each function receives:

- The matrix A and the vector b from the system $Ax = b$;
- An initial approximation x_0;

- A stopping criterion defined by the tolerance and the maximum number of iterations;
- The exact solution (computed via `np.linalg.solve`) to compute the error.

The error at each iteration is calculated as the norm of the difference between the current approximation and the exact solution.

The Jacobi iteration updates each component $x_i^{(k+1)}$ using only the values from $x^{(k)}$, according to:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} \, x_j^{(k)} \right).$$

Below is the function code:

```
import numpy as np

def jacobi_method(A, b, x0, tolerance=1e-6,
    maximum_number_of_iterations=100, x_exact=None):

    """
    Jacobi's method to solve a linear system Ax=b.

    Args:
        1.A (the matrix of the system)
        2.b (The vector b of Ax=b)
        3.x0 (The initial guess of the algorithm)
        4.tolerance (The tolerance (distance from the actual
            solution) for which we accept it to be a solution)
        5.maximum_number_of_iterations (self-explanatory)

    Returns:
        1.The approximate solution to Ax=b after some
            iterations
        2.The error aftr each iteration
    """

    n = len(b) #dimension of the system
    x = x0.copy()
    errors = [] #we will fill this with the errors later

    if x_exact is None:
        x_exact = np.linalg.solve(A, b) #exact solution to the
            system using

    for k in range(maximum_number_of_iterations):
        x_new = np.zeros_like(x)
        for i in range(n):
            s = 0
            for j in range(n):
                if j != i:
```

```
                    s += A[i, j] * x[j]
            x_new[i] = (b[i] - s) / A[i, i]

        current_error = np.linalg.norm(x_new - x_exact) # distance between what we just got and the actual solution
        errors.append(current_error)

        if current_error< tolerance:
            break #its close enough

        x = x_new.copy()

    return x_new, errors
```

Listing 1: Jacobi Method

2.2 Gauss-Seidel Method

The Gauss-Seidel method uses the already updated values immediately in the iteration:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij}\, x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}\, x_j^{(k)} \right).$$

The corresponding code is:

```
import numpy as np

def gauss_seidel_method(A, b, x0, tolerance=1e-6, maximum_number_of_iterations=100, x_exact=None):

    """
    Gauss-Seidel's method to solve a linear system Ax=b.

    Args:
        1.A (the matrix of the system)
        2.b (The vector b of Ax=b)
        3.x0 (The initial guess of the algorithm)
        4.tolerance (The tolerance (distance from the actual solution) for which we accept it to be a solution)
        5.maximum_number_of_iterations (self-explanatory)

    Returns:
        1.The approximate solution to Ax=b after some iterations
        2.The error after each iteration
    """

    n = len(b) #dimension
    x = x0.copy()
```

```
    list_of_errors = [] #we will fill this with the errors
        later

    if x_exact is None:
        x_exact = np.linalg.solve(A, b) #actual solution of the
            system

    for k in range(maximum_number_of_iterations):
        x_old = x.copy()
        for i in range(n):
            s = 0
            for j in range(i):
                s += A[i, j] * x[j]
            for j in range(i+1, n):
                s += A[i, j] * x_old[j]

            x[i] = (b[i] - s) / A[i, i]

        current_error = np.linalg.norm(x - x_exact) #distance
            between what we just got and the actual solution
        list_of_errors.append(current_error)

        if current_error < tolerance:
            break #it's close enough

    return x, list_of_errors
```

Listing 2: Gauss-Seidel Method

3 Numerical Experiments (item c-d)

Here are the results of the experiments performed on 2x2 matrices:

3.1 Test Code

In the file linear_tests.py, several test cases were defined to evaluate the methods:

- **2x2 Matrices:**
  1. Diagonally dominant matrix.
  2. Non-dominant matrix.
  3. Ill-conditioned (nearly singular) matrix.
- **3x3 Matrices:**
  1. Diagonally dominant matrix.
  2. Non-dominant but invertible matrix.

3. Special case: the Jacobi method converges while the Gauss-Seidel method maintains a constant error.

The function `run_test` runs both methods and returns the exact solution and the error trajectories. Then, the function `plot_subplots` organizes the plots (subplots) to display the evolution of the errors at each iteration.

Below is the complete test code:

```
import numpy as np
import matplotlib.pyplot as plt
from methods.jacobi import jacobi_method
from methods.gauss_seidel import gauss_seidel_method

def run_test(A, b, x0, tolerance, maximum_number_of_iterations,
    test_label):

    """
    Executes the Gauss Seidel and JAcobi methods.

    Args:
        1.A (the matrix of the system)
        2.b (The vector b of Ax=b)
        3.x0 (The initial guess of the algorithm)
        4.tolerance (The tolerance (distance from the actual
            solution) for which we accept it to be a solution)
        5.maximum_number_of_iterations (self-explanatory)
        6.test_label (self-explanatory)

    Returns:
        1.The exact solution to Ax=b using numpy's solver
        2.The list of errors from the Jacobi method
        3.The list of errors from the Gauss-Seidel method
    """

    try:
        x_exact = np.linalg.solve(A, b)
    except np.linalg.LinAlgError:
        print(f"{test_label}: Sigular Matrix/Weird Matrix")
        return None, None, None

    x_j, errors_j = jacobi_method(A, b, x0, tolerance=tolerance
        , maximum_number_of_iterations=max_iter, x_exact=
        x_exact)
    x_g, errors_g = gauss_seidel_method(A, b, x0, tolerance=
        tolerance, maximum_number_of_iterations=max_iter,
        x_exact=x_exact)

    return x_exact, errors_j, errors_g

def plot_subplots(tests, nrows, ncols, overall_title):

    """
     Plots the graphs associated to each test.
```

```
    Args:
        1.tests (A list of tuples containing (label, errors_j,
            errors_g) for each test)
        2.nrows (The number of rows in the subplot grid)
        3.ncols (The number of columns in the subplot grid)
        4.overall_title (The title for the entire figure)
    Returns:
        1.None (Displays the plot using plt.show())
    """

    fig, axes = plt.subplots(nrows, ncols, figsize=(5*ncols, 4*
        nrows))
    axes = axes.flatten() if isinstance(axes, np.ndarray) else
        [axes]

    for i, (label, errors_j, errors_g) in enumerate(tests):
        ax = axes[i]
        ax.plot(errors_j, 'o-', label='Jacobi')
        ax.plot(errors_g, 's-', label='Gauss-Seidel')
        ax.set_title(label)
        ax.set_xlabel('Iteration')
        ax.set_ylabel('Error␣(norm)')
        ax.set_yscale('log')   #log scale for better
            visualization
        ax.legend()
        ax.grid(True)

    for j in range(i+1, len(axes)): #deactivates empty subplots
        axes[j].axis('off')

    plt.suptitle(overall_title, fontsize=16)
    plt.tight_layout(rect=[0, 0, 1, 0.95])
    plt.show()


max_iter = 50
tolerance = 1e-8 #general config

# ================================
# 2X2 MATRICES TESTING
# ================================

tests_2x2 = []

A_2_1 = np.array([[4.0, 1.0], #diagonally dominant matrix (it
    should converge faster hopefully)
                  [2.0, 3.0]])
b_2_1 = np.array([1.0, 2.0])
x0_2_1 = np.zeros_like(b_2_1)
label_2_1 = "2x2␣Test␣1:␣Dominant␣Matrix"
_, errors_j, errors_g = run_test(A_2_1, b_2_1, x0_2_1,
    tolerance, max_iter, label_2_1)
tests_2x2.append((label_2_1, errors_j, errors_g))

```

```
A_2_2 = np.array([[1.0, 2.0], #non diagonally dominant matrix (it can diverge or oscillate hopefully)
                  [2.0, 1.0]])
b_2_2 = np.array([3.0, 3.0])
x0_2_2 = np.zeros_like(b_2_2)
label_2_2 = "2x2 Test 2: Non-Dominant Matrix"
_, errors_j, errors_g = run_test(A_2_2, b_2_2, x0_2_2, tolerance, max_iter, label_2_2)
tests_2x2.append((label_2_2, errors_j, errors_g))

A_2_3 = np.array([[1e-4, 1.0], #nearly singular matrix
                  [1.0, 1.0]])
b_2_3 = np.array([1.0, 2.0])
x0_2_3 = np.zeros_like(b_2_3)
label_2_3 = "2x2 Test 3: Nearly Singular Matrix"
_, errors_j, errors_g = run_test(A_2_3, b_2_3, x0_2_3, tolerance, max_iter, label_2_3)
tests_2x2.append((label_2_3, errors_j, errors_g))

plot_subplots(tests_2x2, nrows=1, ncols=3, overall_title="2x2 Matrices")


# ================================
# 3X3 MATRICES TESTING
# ================================

tests_3x3 = []

A_3_1 = np.array([[5.0, 1.0, 1.0],
                  [2.0, 6.0, 1.0], #diagonally dominant matrix (it'll converge faster (hopefully))
                  [1.0, 1.0, 7.0]])
b_3_1 = np.array([7.0, 8.0, 9.0])
x0_3_1 = np.zeros_like(b_3_1)
label_3_1 = "3x3 Test 1: Dominant Matrix"
_, errors_j, errors_g = run_test(A_3_1, b_3_1, x0_3_1, tolerance, max_iter, label_3_1)
tests_3x3.append((label_3_1, errors_j, errors_g))

A_3_2 = np.array([
    [1.0, 3.0, 1.0],
    [2.0, 1.0, 2.0],   #non-dominant matrix but invertible
    [1.0, 2.0, 2.0]
])
b_3_2 = np.array([5.0, 6.0, 7.0])
x0_3_2 = np.zeros_like(b_3_2)
label_3_2 = "3x3 Test 2: Non-Dominant Matrix"
_, errors_j, errors_g = run_test(A_3_2, b_3_2, x0_3_2, tolerance, max_iter, label_3_2)
tests_3x3.append((label_3_2, errors_j, errors_g))

A_3_3 = np.array([[1.0, 0.0, 1.0],
                  [-1.0, 1.0, 0.0],
```

```
                    [1.0, 2.0, -3.0]])    #hopefully jacobi will
                        converge and Gauss-Seidel will have
                        constant error
b_3_3 = np.array([1.0, 2.0, 3.0])
x0_3_3 = np.zeros_like(b_3_3)
label_3_3 = "3x3 Test 3: Special Event"
_, errors_j, errors_g = run_test(A_3_3, b_3_3, x0_3_3,
    tolerance, max_iter, label_3_3)
tests_3x3.append((label_3_3, errors_j, errors_g))

plot_subplots(tests_3x3, nrows=1, ncols=3, overall_title="3x3
    Matrices")
```

Listing 3: Test Code and Plotting

3.2 2x2 Matrices used for the plots

- Dominant matrix:

$$A = \begin{pmatrix} 4 & 1 \\ 2 & 3 \end{pmatrix}, \quad b = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

- Non-dominant matrix:

$$A = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}, \quad b = \begin{pmatrix} 3 \\ 3 \end{pmatrix}$$

- Nearly singular matrix:

$$A = \begin{pmatrix} 10^{-4} & 1 \\ 1 & 1 \end{pmatrix}, \quad b = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

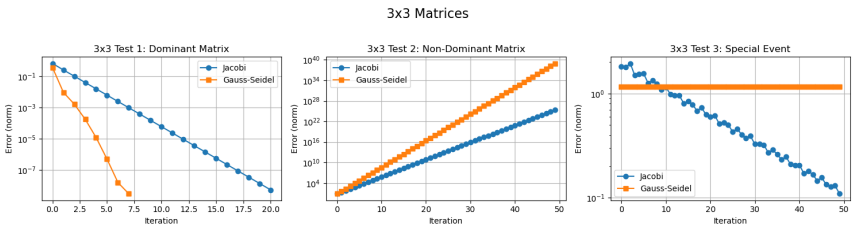

Figure 1: Convergence behavior for 3x3 matrices

3.3 3x3 Matrices used for the plots:

- Dominant matrix:

$$A = \begin{pmatrix} 5 & 1 & 1 \\ 2 & 6 & 1 \\ 1 & 1 & 7 \end{pmatrix}, \quad b = \begin{pmatrix} 7 \\ 8 \\ 9 \end{pmatrix}$$

- Non-dominant matrix:

$$A = \begin{pmatrix} 1 & 3 & 1 \\ 2 & 1 & 2 \\ 1 & 2 & 2 \end{pmatrix}, \quad b = \begin{pmatrix} 5 \\ 6 \\ 7 \end{pmatrix}$$

- Special case:

$$A = \begin{pmatrix} 1 & 0 & 1 \\ -1 & 1 & 0 \\ 1 & 2 & -3 \end{pmatrix}, \quad b = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

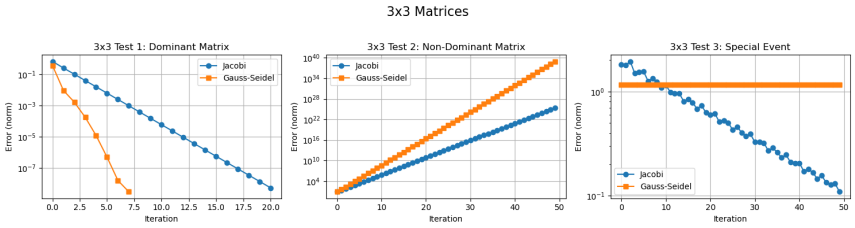

Figure 2: Convergence behavior for 3x3 matrices

4 Random Analysis (Items d-f)

Here are the random experiments done with the matrices, we have performed 10 experiments of each kind (diagonal, non-diagonal and complete perturbation.

4.1 Perturbation Effects On a Special Matrix (Item d)

We have analyzed different kinds of random perturbations on the matrix and vector:

$$A = \begin{pmatrix} 1 & 2 & -2 \\ 1 & 1 & 1 \\ 2 & 2 & 1 \end{pmatrix}, \quad b = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

10 random experiments were conducted per type of perturbation.

- Random perturbations: The full perturbation showed divergence on most plots

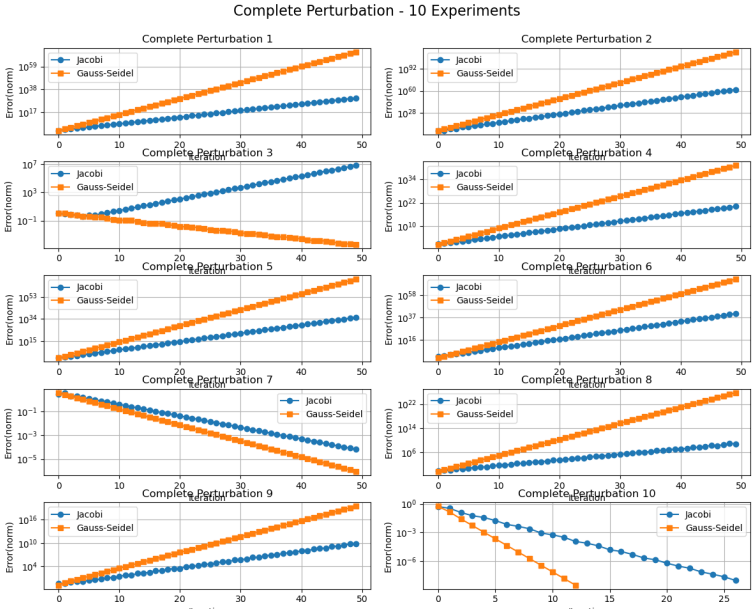

Figure 3: Random full-matrix perturbation

- The diagonal-only perturbation has shown convergence on most plots

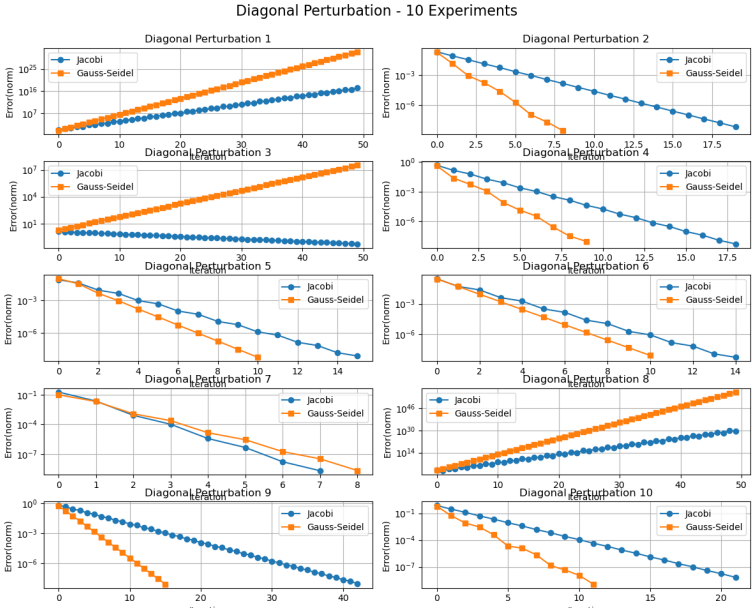

Figure 4: Diagonal-Only Perturbation

- The non-diagonal plots show diverge on most cases

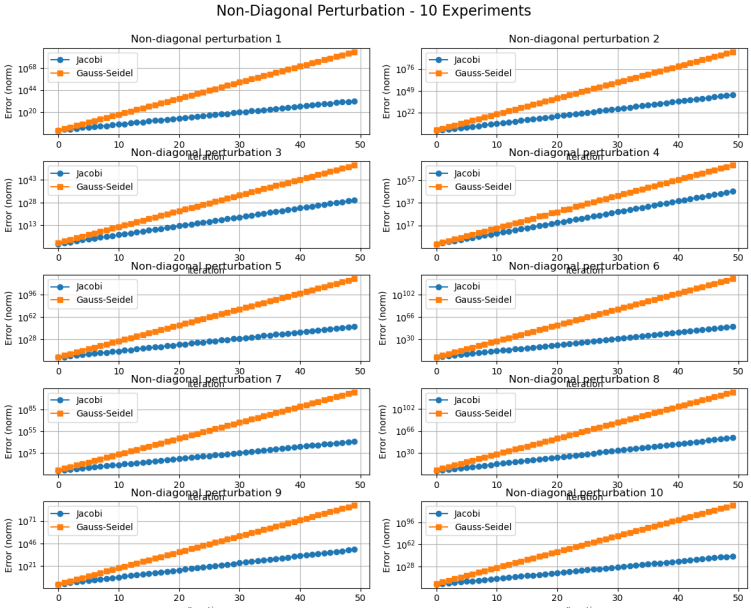

Figure 5: Only Non-Diagonal perturbation

4.2 Analysis of the plots

The conducted experiments with random perturbations: complete, diagonal, and off-diagonal-reveal shows us that:

- **Complete Perturbation:** Most matrices lost diagonal dominance, resulting in divergence for both methods. Some experiments showed exponential growth in the error norm, especially for Gauss-Seidel.
- **Diagonal Perturbation:** Several matrices improved convergence behavior, especially when the perturbation increased the diagonal dominance. In some cases, Gauss-Seidel outperformed Jacobi.
- **Non-Diagonal Perturbation:** This consistently weakened convergence properties, leading to divergence in most cases. Increasing the weight of off-diagonal entries typically increased the spectral radius.

These results align with the theoretical prediction that convergence depends on the spectral radius $\rho(T)$(see proof here). Gauss-Seidel, although often more efficient, is more sensitive to unfavorable perturbations than Jacobi. This validates the importance of matrix conditioning and dominance in iterative method selection.

4.3 Convergence Anomalies (Item e)

The given matrix is:

$$A = \begin{pmatrix} 1 & 0 & 1 \\ -1 & 1 & 0 \\ 1 & 2 & -3 \end{pmatrix}$$

The behavior described can be explained through the spectral analysis of our iterative methods. (See proof here)

For an iterative method of the form:

$$x^{(k+1)} = Tx^{(k)} + c,$$

convergence occurs if and only if the spectral radius $\rho(T) < 1$.

The iteration matrices are: - For Jacobi: $T_J = D^{-1}(L+U)$ - For Gauss-Seidel: $T_{GS} = (D-L)^{-1}U$

In the case of the given matrix, it was found that:

$$\rho(T_J) < 1 \quad \text{and} \quad \rho(T_{GS}) = 1$$

Therefore, the Jacobi method converges because the error

$$E^{(k)} = T_J^k E^{(0)}$$

tends to zero. On the other hand, for the Gauss-Seidel method, since $\rho(T_{GS}) = 1$, we have:

$$E^{(k)} = T_{GS}^k E^{(0)} = E^{(0)} \quad \Rightarrow \quad \|E^{(k)}\| = \|E^{(0)}\|,$$

meaning the error remains constant and the method does not converge.

This situation illustrates that although Gauss-Seidel is generally more efficient, it may fail in specific cases where the Jacobi method still converges—particularly when the spectral radius of T_{GS} is exactly 1.

4.4 Constructed Divergence (Item f)

To construct a matrix where Jacobi converges ($\rho(T_J) < 1$) and Gauss-Seidel diverges ($\rho(T_{GS}) > 1$), consider the following steps:

1. Design A such that $T_J = D^{-1}(L + U)$ has eigenvalues inside the unit circle.
2. Ensure $T_{GS} = (D - L)^{-1}U$ has at least one eigenvalue outside the unit circle.

An example is:

$$A = \begin{pmatrix} 1 & 2 & -2 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix}, \quad b = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}.$$

For this matrix:

- Jacobi converges because $\rho(T_J) = 0$ (nilpotent matrix).
- Gauss-Seidel diverges because $\rho(T_{GS}) = 2$ (easily verifiable via the eigenvalues of T_{GS}).

5 Foundations

Here we recap Let $T \in \mathbb{R}^{n \times n}$ be a square matrix. The *spectral radius* of T is defined as

$$\rho(T) = \max\{|\lambda| : \lambda \text{ is an eigenvalue of } T\}.$$

Theorem

Consider the iterative method

$$x^{(k+1)} = Tx^{(k)} + c,$$

and let x^* be its fixed point, i.e., the solution to

$$x^* = Tx^* + c.$$

Then, the iterative method converges (i.e., $x^{(k)} \to x^*$ for any initial guess $x^{(0)}$) if and only if

$$\rho(T) < 1.$$

Proof

1. Error Propagation:
Define the error at iteration k as

$$E^{(k)} = x^{(k)} - x^*.$$

Subtracting the fixed point equation from the iterative equation gives

$$\begin{aligned} E^{(k+1)} &= x^{(k+1)} - x^* \\ &= Tx^{(k)} + c - \big(Tx^* + c\big) \\ &= T\big(x^{(k)} - x^*\big) \\ &= TE^{(k)}. \end{aligned}$$

By induction, we have

$$E^{(k)} = T^k E^{(0)}.$$

2. Sufficient Condition ($\rho(T) < 1$):
Assume that $\rho(T) < 1$. By the properties of matrix norms and the Gelfand formula (see proof here) the Gelfand formula, there exists an induced norm $\|\cdot\|$ and a constant $C > 0$ such that

$$\|T^k\| \leq C\rho(T)^k.$$

Then, the error satisfies

$$\|E^{(k)}\| = \|T^k E^{(0)}\| \leq \|T^k\| \, \|E^{(0)}\| \leq C\rho(T)^k \|E^{(0)}\|.$$

Since $\rho(T)^k \to 0$ as $k \to \infty$ (because $\rho(T) < 1$), it follows that

$$\lim_{k\to\infty} \|E^{(k)}\| = 0,$$

and hence, $x^{(k)} \to x^*$.

3. Necessary Condition ($\rho(T) \geq 1$ implies non-convergence):
Conversely, assume that $\rho(T) \geq 1$. Then, there exists at least one eigenvalue λ of T with $|\lambda| \geq 1$. Let v be an eigenvector corresponding to λ, so that

$$Tv = \lambda v.$$

Choose the initial error $E^{(0)} = v$. Then,

$$E^{(k)} = T^k v = \lambda^k v.$$

- If $|\lambda| > 1$, then $|\lambda|^k \to \infty$ as $k \to \infty$, causing $E^{(k)}$ to diverge.
- If $|\lambda| = 1$, then $\|E^{(k)}\| = \|v\|$ remains constant, and the error does not decay.

In either case, the iterative method does not converge to x^*.

□

6 Source Code and Repository

All the source code used in this report is available on the GitHub repository: https://github.com/arthurabello/nla-assignment-1

References

[1] Llyod N. Trefethen, David Bau III: *Numerical Linear Algebra*, SIAM - Society for Industrial and Applied Mathematics, Philadelphia.