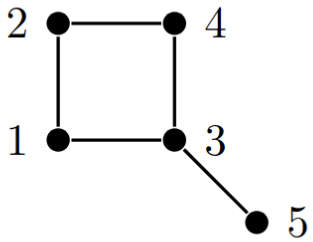
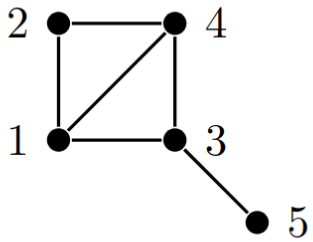
Problem 1: Appreciate Cholesky Graphically

Problem Statement



(a) Graph G_1



(b) Graph G_2

a

Find a matrix $\mathbf{A} \in \mathbb{R}^{5\times5}$ s.t $\mathcal{G}(\mathbf{A}) = G_1$ in the figure above, and s.t. $\mathbf{A} \succ 0$. *Hint: Associate positive numbers or weights to each node and each edge of G_1. Define $A_{ij} = A_{ji}$ as the negative of the weight on the edge (i, j) of G_1. Define A_{ii} to be the sum of the weight of node i and the weights of all edges connected to node i. Such a matrix is called a Laplacian matrix for G_1. Compute the eigenvalues to verify it is indeed PD.*

b

Compute the lower triangular Cholesky factor $\mathbf{L} \in \mathbb{R}^{5\times5}$ with positive diagonals such that $\mathbf{LL}^T$ equals $\mathbf{A}$ in part (a). Draw a graph on 5 nodes that describes the sparsity pattern of $\mathbf{L}$; i.e. draw $G(\mathbf{L})$. Verify that $G(\mathbf{L})$ is a chordal graph.

c

Notice that G_2 in figure 1b is a chordal graph. Find a permutation matrix $\mathbf{P}$ s.t. that when you compute the Cholesky factor $\mathbf{L}'$ of $\mathbf{PAP}^T$ with your $\mathbf{A}$ from part (a), then $G(\mathbf{L}') = G_2$. *Hint: Eliminate the nodes of G_1 in the sequence $(5, 2, 3, 1, 4)$. Encode that elimination order in a permutation matrix.*

d(**)

Let $\mathbf{X} \in \mathbb{R}^{n\times n}$ be a PD matrix and $\mathbf{\Gamma} \in \mathbb{R}^{n\times n}$ be a lower triangular matrix with positive diagonals, s.t. $\mathbf{X} = \mathbf{\Gamma\Gamma}^T$. Prove that $G(\mathbf{X})$ is a subgraph of $G(\mathbf{\Gamma})$ and $G(\mathbf{\Gamma})$ is a chordal graph. *Hint: Use an induction argument on the size of the matrix. Characterize how the steps in Cholesky factorization shapes the sparsity pattern of $\mathbf{\Gamma}$. Finally, utilize the fact that if a node and a set of edges from that node is added to a chordal graph G, then the obtained graph is chordal IFF every two neighbors of the new node in G already shared an edge in G.*

Solution

For script, see relevant section at the end of the document.

a

Relevant script output:

```
1a Laplacian:
[[ 3. -1. -1.  0.  0.]
[-1.  3.  0. -1.  0.]
[-1.  0.  4. -1. -1.]
[ 0. -1. -1.  3.  0.]
[ 0.  0. -1.  0.  2.]]
1a eigenvalues:
[5.481 1.    1.83  3.    3.689]
1a PD: True
```

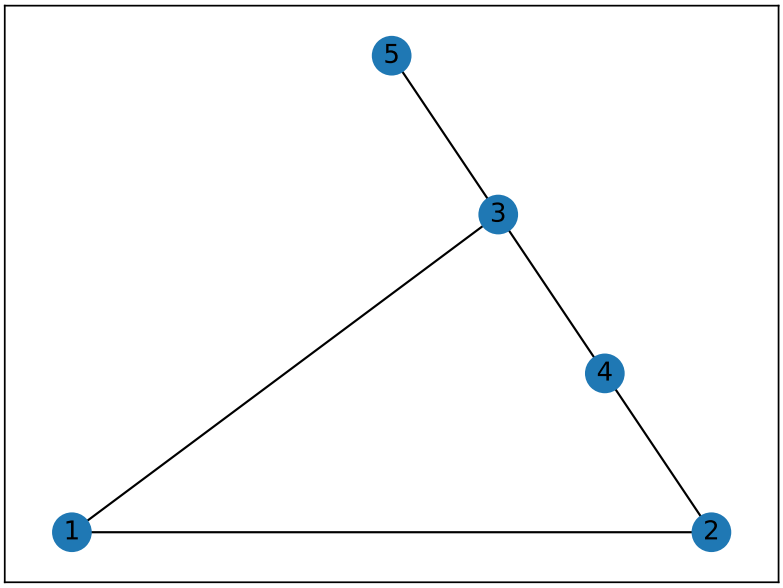


Figure 1: Graph G_1 generated in networkx, for verification

b

```
1b L:
[[ 1.732  0.     0.     0.     0.   ]
 [-0.577  1.633  0.     0.     0.   ]
 [-0.577 -0.204  1.904  0.     0.   ]
 [ 0.    -0.612 -0.591  1.509  0.   ]
 [ 0.     0.    -0.525 -0.206  1.297]]
```

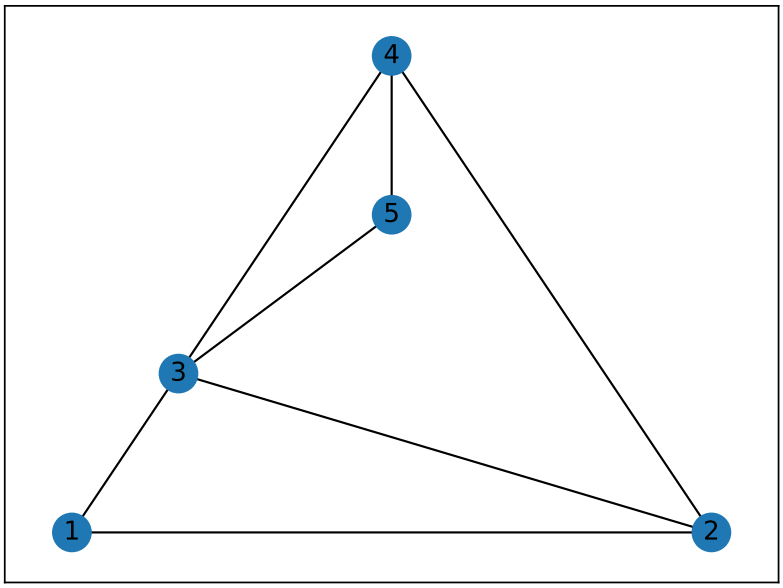


Figure 2: Graph $\mathcal{G}(L)$ generated in networkx. Note it is indeed chordal.

c

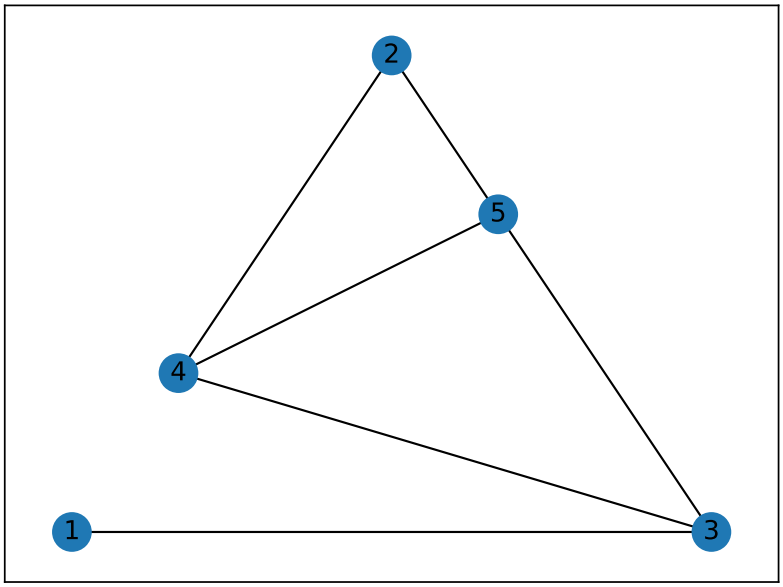


Figure 3: Graph $\mathcal{G}(L')$ generated in networkx. Note it indeed matches the structure of G_2 with the nodes swapped as described.

d

...meh, it's 4PM on the Friday before break.

Problem 2: Split 'em matrices for Linear Systems

Problem Statement

Consider a linear system of equations $Ax = b$ where $A \in \mathbb{R}^{n\times n}$ and $\exists A^{-1}$. The matrix splitting method splits A as $A = M - n$ and successively solves the equation:

$$x^{k+1} = M^{-1}\left(Nx^k + b\right) \tag{1}$$

for $k \geq 0$ to perform a fixed-point iteration in order to solve the linear system $Ax = b$. Identify L, D and U as the strictly lower triangular, diagonal, and strictly upper triangular part of A. Then, $A = L + D + U$. Whem $M = L + D$, then the fixed point iteration becomes:

$$x_i^{k+1} \coloneqq \frac{1}{A_{ii}}\left[b_i - \sum_{j<i} A_{ij}x_j^{k+1} - \sum_{j>i} A_{ij}x_j^k\right] \tag{2}$$

for $i = 1, \ldots, n$ and $k \geq 0$.

a

Write the update equation for each element of x as in equation 2 when $M = D+U$ using backward induction to compute the iterates in equation 1 successively.

b

Call the method you devised in part (a) as the *backward* Gauss-Seidel method. Which among this method and Jacobi methods would you expect to converge faster to the solution of $Ax = b$? Which one will you choose if distributed computation is involved, and why?

c

Implement both backward Gauss-Seidel and Jacobi methods to solve the linear system of equations $Ax = b$ with

$$A \coloneqq \begin{pmatrix} 10 & 5 & 3 & 4 \\ 4 & 10 & 2 & 1 \\ 1 & 3 & 8 & 2 \\ 1 & 6 & 3 & 9 \end{pmatrix}, \; b \coloneqq \begin{pmatrix} 4 \\ -5 \\ 4 \\ -11 \end{pmatrix}$$

starting with $x^0 \coloneqq (1, 1, 1, 1)^T$. Plot the residues and errors in the succesive iterates on a semilog plot. Iterate until the residue falls below 10^-5. The error and residue, respectively, at iteration k are given by:

$$\epsilon^k \coloneqq \|x - x^*\|, \; r^k \coloneqq \|b - Ax^k\|$$

where x^* is the solution to the linear system.

d

Recall that the method of succesive over-relaxation (SOR) uses $M = \frac{1}{\omega}D + L$. Any matrix splitting method converges to the solution of a linear system *IF* the spectral radius of the matrix $I - M^{-1}A$ is less than 1. Plot the spectral radius of $I - M^{-1}A$ for SOR as a function of ω in the range $[0.01, 2.20]$ for 20 randomly generated PD A matrices on the same graph. Based on your plot, can you guess for what values of ω SOR converges for PD matrices?

Solution

a

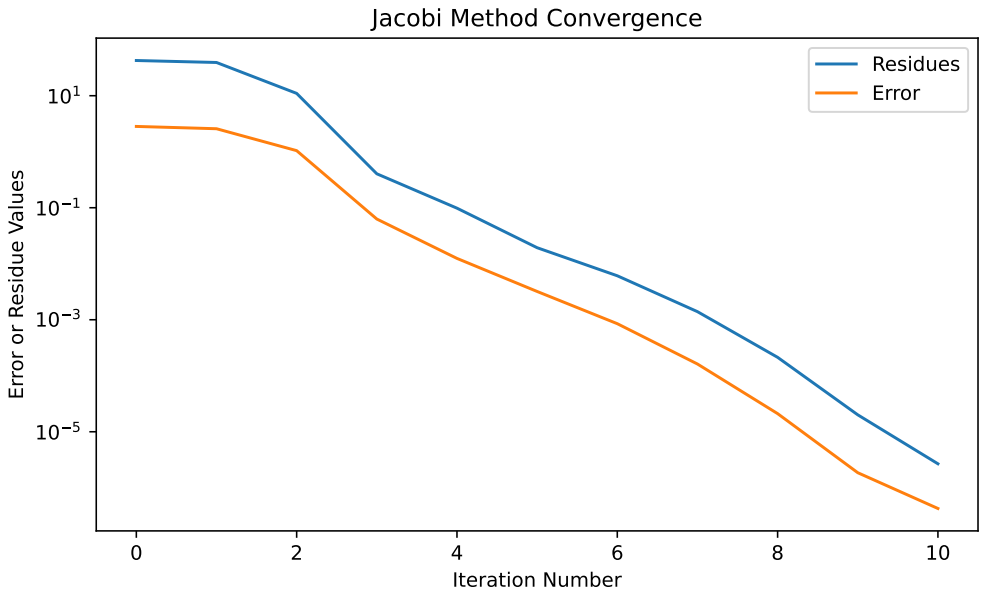
$$x_i^{k+1} := \frac{1}{A_{ii}} \left[b_i - \sum_{j>i} A_{ij} x_j^{k+1} - \sum_{j<i} A_{ij} x_j^k \right]$$

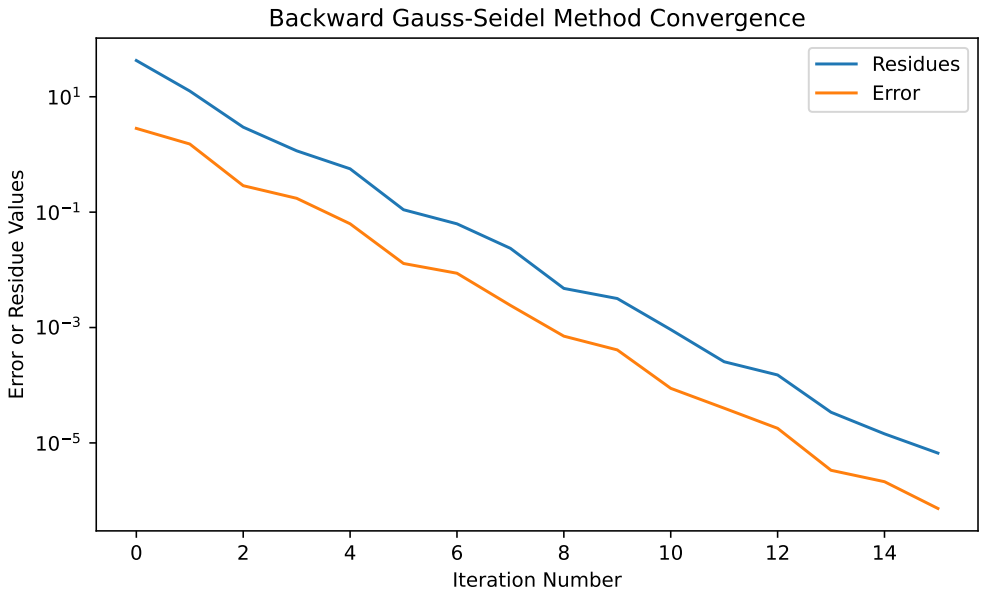
b

It will depend. Gauss-Seidel methods use more "up to date" information, and so intuitively we would expect faster convergence. However, this is not a hard and fast rule, and there are counterexamples wherein one method converges and the other doesn't, OR that one will converge faster in certain cases, or slower in others. See [1].

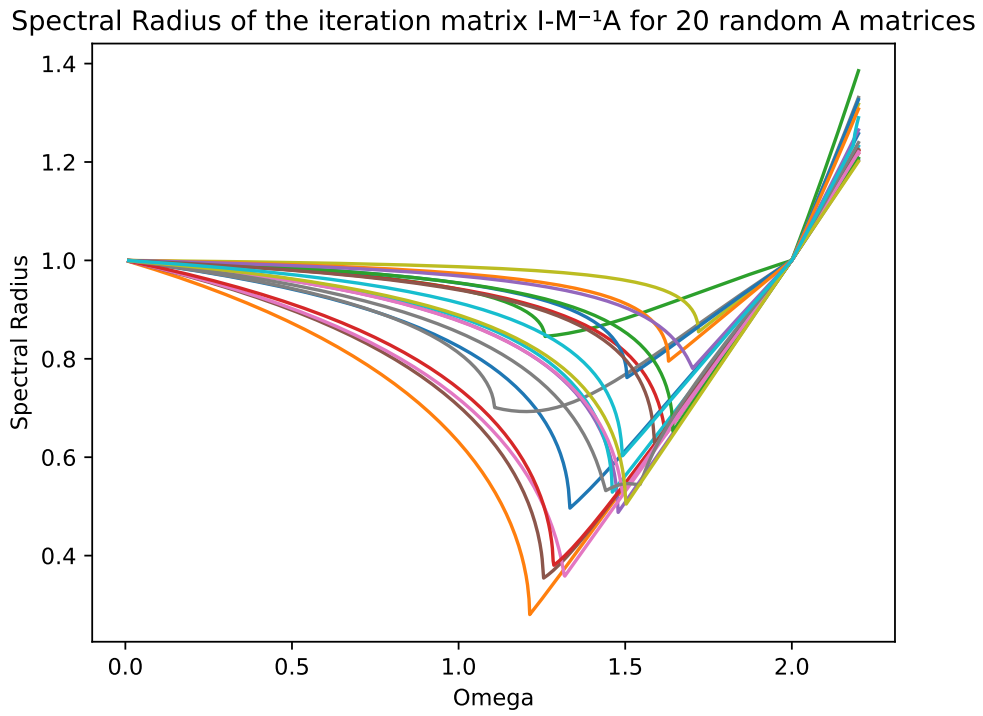
I think that using a typical scatter-gather scheme for distributed computation, Jacobi iteration would be preferred for distributed computing. Each element of x^{k+1} can be calculated in an embarrasingly parallel fashion (i.e. each element can be calculated in a fully independent fashion), whereas Gauss-Seidel will require iterations and sharing current estimates of x^{k+1} around, which will limit the ability to parallelize.

c





d



Empirically it appears that the spectral radius will be less than 1 for values between 0 and 2. This is in fact a proven property of SOR methods, according to the Wikipedia article on them.

Code

Problem 1

```python
import matplotlib.pyplot as plt
import numpy as np
import networkx as nx

label_conversion_dict = {0: 1, 1: 2, 2: 3, 3: 4, 4: 5}

np.set_printoptions(precision=3)


def get_G1() -> nx.Graph:
    g = nx.Graph()
    g.add_nodes_from((1, 2, 3, 4, 5))
    g.add_edges_from(((1, 2), (2, 4), (4, 3), (3, 5), (3, 1)))

    return g


def get_G2() -> nx.Graph:
    g = get_G1()
    g.add_edge(1, 4)

    return g


def plot_graph(G, label_conversion=label_conversion_dict):
    pos = nx.planar_layout(G)
    nx.draw_networkx_nodes(G, pos)
    nx.draw_networkx_edges(G, pos)
    nx.draw_networkx_labels(G, pos, label_conversion)


def one_a():
    # The definition of the Laplacian given by the HW assigns a weight to the
    # node as well, so we must add some positive diagonal matrix (use I for
    # convenience)
    laplacian = nx.laplacian_matrix(get_G1()).toarray() + np.eye(5)
    print(f"1a Laplacian:\n{laplacian}")
    eigvals = np.linalg.eigvals(laplacian)
    print(f"1a eigenvalues:\n{eigvals}")
    print(f"1a PD: {not np.any(np.isclose(eigvals, 0.0)) and np.all(eigvals > 0)}")
    plot_graph(get_G1(), label_conversion=None)
    plt.savefig("one_a.pdf")


def one_b():
    laplacian = nx.laplacian_matrix(get_G1()).toarray() + np.eye(5)
    L = np.linalg.cholesky(laplacian)
    print("1b L:")
    print(L)
    G_of_L = nx.from_numpy_array(L, create_using=nx.Graph)
    G_of_L.remove_edges_from(nx.selfloop_edges(G_of_L))
    plt.figure()
    plot_graph(G_of_L)
    plt.savefig("one_b.pdf")


def one_c():
    P = np.array(
        (
            (0, 0, 0, 0, 1),
            (0, 1, 0, 0, 0),
            (0, 0, 1, 0, 0),
            (1, 0, 0, 0, 0),
            (0, 0, 0, 1, 0),
        )
    )
    laplacian = nx.laplacian_matrix(get_G1()).toarray() + np.eye(5)
    reordered = P @ laplacian @ P.T
    L = np.linalg.cholesky(reordered)
    G_of_L = nx.from_numpy_array(L, create_using=nx.Graph)
    G_of_L.remove_edges_from(nx.selfloop_edges(G_of_L))
    plt.figure()
    plot_graph(G_of_L, label_conversion=label_conversion_dict)
    plt.savefig("one_c.pdf")


if __name__ == "__main__":
    one_a()
    one_b()
    one_c()
```

Problem 2

```python
from typing import Tuple, List

import numpy as np
import matplotlib.pyplot as plt

A = np.array(
    (
        (10, 5, 3, 4),
        (4, 10, 2, 1),
        (1, 3, 8, 2),
        (1, 6, 3, 9),
    )
)

b = np.array((4, -5, 4, -11))
x_exact = np.linalg.solve(A, b)
print(x_exact)
err_func = lambda xk: np.linalg.norm(xk - x_exact)


def residual(A: np.ndarray, x: np.ndarray, b: np.ndarray) -> float:
    return np.linalg.norm(b - A @ x)


def Jacobi(
    A: np.ndarray,
    b: np.ndarray,
    x0: np.ndarray,
    k_max: int = 1000,
    res_max: float = 1e-5,
) -> Tuple[np.ndarray, List[float], List[float]]:
    n = A.shape[0]
    print("n:", n)
    xk = np.copy(x0)
    # "Iteration 0"
    residuals = [residual(A, xk, b)]
    errors = [err_func(xk)]
    xkp1 = np.zeros_like(xk)
    for i in range(n):
        for k in range(k_max):
            for i in range(n):
                sigma = 0
                for j in range(n):
                    if j != i:
                        sigma += A[i, j] * xk[j]
                xkp1[i] = (b[i] - sigma) / A[i, i]
            xk = xkp1
            res = residual(A, xk, b)
            err = err_func(xk)
            residuals.append(res)
            errors.append(err)

            if res < res_max:
                print(f"Converged at iteration {k}")
                return xk, residuals, errors
    else:
        print(f"Failed to converge in {k_max} iterations!")
        return xk, residuals, errors


def backward_Gauss_Seidel(
    A: np.ndarray,
    b: np.ndarray,
    x0: np.ndarray,
    k_max: int = 1000,
    res_max: float = 1e-5,
) -> Tuple[np.ndarray, List[float], List[float]]:
    n = A.shape[0]
    xk = x0
    # "Iteration 0"
    residuals = [residual(A, xk, b)]
    errors = [err_func(xk)]
    for k in range(k_max):
        xkp1 = xk
        for i in reversed(range(n)):
            # j > i
            sum1 = np.sum(A[i, i + 1 :] * xk[i + 1 :])
            # j < i
            sum2 = np.sum(A[i, 0:i] * xk[:i])
            xkp1[i] = (b[i] - sum1 - sum2) / A[i, i]

        xk = xkp1

        res = residual(A, xk, b)
        err = err_func(xk)
        residuals.append(res)
        errors.append(err)

        if res < res_max:
```

```
            print(f"Converged at iteration {k}")
            return xk, residuals, errors
    else:
        print(f"Failed to converged in {k_max} iterations!")
        return xk, residuals, errors


def gen_PD_matrix(n: int) -> np.ndarray:
    tmp = np.random.randn(n, n)
    return tmp @ tmp.T


def main() -> None:
    # Jacobi Plot
    x_jac, jac_res, jac_error = Jacobi(A, b, np.array((1.0, 1.0, 1.0, 1.0)))
    print(x_jac)
    plt.figure(figsize=(8, 4.5))
    plt.semilogy(jac_res, label="Residues")
    plt.semilogy(jac_error, label="Error")
    plt.xlabel("Iteration Number")
    plt.ylabel("Error or Residue Values")
    plt.title("Jacobi Method Convergence")
    plt.legend()
    plt.savefig("jacobi_iter.pdf", bbox_inches="tight")

    # Backwards Gauss-Seidel
    x_bgs, bgs_res, bgs_err = backward_Gauss_Seidel(
        A, b, np.array((1.0, 1.0, 1.0, 1.0))
    )
    print(x_bgs)
    plt.figure(figsize=(8, 4.5))
    plt.semilogy(bgs_res, label="Residues")
    plt.semilogy(bgs_err, label="Error")
    plt.xlabel("Iteration Number")
    plt.ylabel("Error or Residue Values")
    plt.title("Backward Gauss-Seidel Method Convergence")
    plt.legend()
    plt.savefig("bgs_iter.pdf", bbox_inches="tight")

    # SOR Investigation
    omegas = np.linspace(0.01, 2.20, num=1000, endpoint=True)
    plt.figure()
    for _ in range(20):
        A_tmp = gen_PD_matrix(4)
        D = np.diag(np.diag(A_tmp))
        L = np.tril(A_tmp, -1)
        radii = []
        for omega in omegas:
            M = (1.0 / omega) * D + L
            radius = np.max(
                np.abs(np.linalg.eigvals(np.eye(4) - np.linalg.inv(M) @ A_tmp))
            )
            radii.append(radius)
        plt.plot(omegas, radii)

    plt.xlabel("Omega")
    plt.ylabel("Spectral Radius")
    plt.title(
        f"Spectral Radius of the iteration matrix I-M\N{SUPERSCRIPT MINUS}\N{SUPERSCRIPT ONE}A for 20
    random A matrices"
    )
    plt.savefig("sor_omegas.pdf", bbox_inches="tight")


if __name__ == "__main__":
    main()
```

Bibliography

[1] Stewart Venit. "The Convergence of Jacobi and Gauss-Seidel Iteration". In: *Mathematics Magazine* 48.3 (May 1975), pp. 163–167.