PSEUDOSPECTRA AND UTMOST RIGHT NORM

1 Pseudospectra and Dynamical System Stability

1.1 Introduction

We consider continuous-time linear dynamical systems of the form:

$$\dot{x}(t) = Ax(t), \quad x(0) = x_0, \quad A \in \mathbb{C}^{n\times n}$$

where $x(t) \in \mathbb{C}^n$ is the state vector at time t, and A is a constant system matrix. The central question is whether the system is asymptotically stable, i.e., whether $x(t) \to 0$ as $t \to \infty$ for all initial conditions x_0.

The solution to this system is given by the matrix exponential:

$$x(t) = e^{At}x_0$$

Theorem 1.1. *The system $\dot{x} = Ax$ is asymptotically stable if and only if $\mathrm{Re}(\lambda_j) < 0$ for all eigenvalues λ_j of A.*

Proof. If A is diagonalizable, then:

$$e^{At} = Ve^{\Lambda t}V^{-1}, \quad \text{with } \Lambda = \mathrm{diag}(\lambda_1, \dots, \lambda_n)$$

so that:

$$x(t) = \sum_{j=1}^{n} e^{\lambda_j t} v_j (w_j^* x_0)$$

where v_j and w_j are right and left eigenvectors, respectively.

If A is not diagonalizable, we use Jordan normal form:

$$A = VJV^{-1}, \quad \text{where} \quad J = \begin{bmatrix} J_1 & & \\ & \ddots & \\ & & J_m \end{bmatrix}$$

and each J_k is a Jordan block:

$$J_k = \begin{bmatrix} \lambda_k & 1 & 0 & \cdots & 0 \\ 0 & \lambda_k & 1 & \cdots & 0 \\ \vdots & & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_k & 1 \\ 0 & 0 & \cdots & 0 & \lambda_k \end{bmatrix}$$

The matrix exponential of a Jordan block is:

$$e^{J_k t} = e^{\lambda_k t} \begin{bmatrix} 1 & t & \frac{t^2}{2!} & \cdots & \frac{t^{m_k-1}}{(m_k-1)!} \\ 0 & 1 & t & \cdots & \frac{t^{m_k-2}}{(m_k-2)!} \\ \vdots & & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & t \\ 0 & 0 & \cdots & 0 & 1 \end{bmatrix}$$

Since

$$e^{At} = Ve^{Jt}V^{-1} = V \begin{bmatrix} e^{J_1 t} & & \\ & \ddots & \\ & & e^{J_m t} \end{bmatrix} V^{-1},$$

the solution behavior depends on terms $e^{\lambda_k t}$ multiplied by polynomials in t. If $\mathrm{Re}(\lambda_k) < 0$ for all λ_k, then

$$\lim_{t\to\infty} e^{\lambda_k t} t^p = 0 \quad \text{for any } p \geq 0,$$

implying asymptotic stability.

Conversely, if $\mathrm{Re}(\lambda_j) \geq 0$ for some λ_j, the corresponding term does not vanish as $t \to \infty$. □

1.2 Problem Statement

The issue is that for non-normal matrices, small perturbations can cause large changes in the eigenvalues. Even when $\|E\| \ll 1$, the spectrum of $A + E$ can vary a lot from the one of A.

To measure this sensitivity and better understand stability under perturbations, the notion of pseudospectra of A is used. The pseudospectrum describes the regions in the complex plane where perturbed matrices may have eigenvalues.

From there, algorithms can be implemented to calculate the rightmost point of the set $\{\lambda(A + \varepsilon E) \in \mathbb{C} : E \in \mathbb{C}^{n,n}, \|E\|_F = 1\}$ focusing on maximizing the function $f(\lambda) = -Re(\lambda)$.

2 Equivalent Definitions of Pseudospectra

The following definitions are equivalent for the ϵ-pseudospectrum of a matrix $A \in \mathbb{C}^{n\times n}$ and $\epsilon > 0$ [1].

Definition 2.1 (Resolvent Norm)**.** *For $A \in \mathbb{C}^{n\times n}$ and $\epsilon > 0$:*

$$\sigma_\epsilon(A) = \{z \in \mathbb{C} : \|(zI - A)^{-1}\| > \epsilon^{-1}\}$$

Definition 2.2 (Perturbed Spectrum)**.**

$$\sigma_\epsilon(A) = \bigcup_{\|E\|<\epsilon} \sigma(A + E)$$

Definition 2.3 (Approximate Eigenvector)**.**

$$\sigma_\epsilon(A) = \{z \in \mathbb{C} : \exists v \in \mathbb{C}^n, \|v\| = 1 \text{ with } \|(zI - A)v\| < \epsilon\}$$

3 Equivalence Proofs

3.1 Resolvent and Perturbed Spectrum

Theorem 3.1. *Definitions 1 and 2 are equivalent.*

Proof. ($\Rightarrow$) If $\|(zI - A)^{-1}\| > \epsilon^{-1}$, construct rank-1 E with $\|E\| < \epsilon$ making z an eigenvalue of $A + E$.

($\Leftarrow$) If $z \in \sigma(A + E)$ for $\|E\| < \epsilon$, then $\|(zI - A)^{-1}\| > \epsilon^{-1}$. □

3.2 Perturbed Spectrum and Approximate Eigenvector

Theorem 3.2. *Definitions 2 and 3 are equivalent.*

Proof. ($\Rightarrow$) From $(A + E - zI)v = 0$, construct E showing z is an approximate eigenvalue.
($\Leftarrow$) Given approximate eigenvector v, build perturbation E with $\|E\| < \epsilon$. □

4 Pseudospectrum via Minimum Singular Value

4.1 Rank-1 Perturbations

Theorem 4.1. *For any matrix $A \in \mathbb{C}^{n\times n}$ and $\epsilon > 0$, the minimal norm perturbation E that moves $z \in \mathbb{C}$ into $\sigma(A + E)$ is a rank-1 matrix of the form:*

$$E = -\sigma uv^*$$

where $\sigma = s_{\min}(A - zI)$, and u, v are the corresponding left/right singular vectors of A.

Proof. Let $A - zI = U\Sigma V^*$ be the SVD, where $\Sigma = \mathrm{diag}(\sigma_1, \ldots, \sigma_n)$ with $\sigma_1 \geq \cdots \geq \sigma_n \geq 0$. We take the minimal singular value $\sigma_n = s_{\min}(A - zI)$ with corresponding singular vectors u_n, v_n. Defining:

$$E_{\text{opt}} = -\sigma_n u_n v_n^*$$

This is a rank-1 matrix with $\|E_{\text{opt}}\|_2 = \sigma_n$. Given that:

$$(A + E_{\text{opt}} - zI)v_n = (A - zI)v_n + E_{\text{opt}}v_n = \sigma_n u_n - \sigma_n u_n = 0$$

Thus $z \in \sigma(A + E_{\text{opt}})$. For any other perturbation E' with $\|E'\|_2 < \sigma_n$, we have:

$$s_{\min}(A + E' - zI) \geq s_{\min}(A - zI) - \|E'\|_2 > 0$$

Hence $z \notin \sigma(A + E')$. Therefore, E_{opt} is minimal. □

4.2 Singular Value Characterization

Theorem 4.2 (Pseudospectrum via singular value decomposition, SVD)**.** *For any $A \in \mathbb{C}^{n\times n}$ and $\epsilon > 0$:*

$$z \in \sigma_\epsilon(A) \iff s_{\min}(A - zI) \leq \epsilon$$

, where s_{min} is the smallest singular value.

Proof. ($\Rightarrow$) Suppose $z \in \sigma_\epsilon(A)$. By definition, there exists E with $\|E\|_2 \leq \epsilon$ such that:

$$\det(A + E - zI) = 0 \implies s_{\min}(A + E - zI) = 0$$

From the singular value perturbation theorem:

$$|s_{\min}(A - zI) - s_{\min}(A + E - zI)| \leq \|E\|_2 \leq \epsilon$$

Thus:

$$s_{\min}(A - zI) \leq \epsilon + 0 = \epsilon$$

($\Leftarrow$) Assume $s_{\min}(A - zI) \leq \epsilon$. Let $A - zI = U\Sigma V^*$ be the SVD. Construct the rank-1 perturbation:

$$E = -\sigma_n u_n v_n^*, \quad \text{where } \sigma_n = s_{\min}(A - zI)$$

Therefore $z \in \sigma(A + E) \subset \sigma_\epsilon(A)$. □

Theorem 4.3. *Let $A \in \mathbb{C}^{n\times n}$, $\lambda \in \mathbb{C}$, and $\epsilon > 0$. Then:*

$$s_{\min}(A - \lambda I) \leq \epsilon \quad \iff \quad \lambda \in \sigma(A + \epsilon E) \text{ for some } E \in \mathbb{C}^{n\times n} \text{ with } \|E\| \leq 1.$$

Proof. ($\Rightarrow$) We suppose $s_{\min}(A-\lambda I) \leq \epsilon$. By the smallest singular value properties, there exist unit vectors $u, v \in \mathbb{C}^n$ such that:

$$(A - \lambda I)v = \sigma u$$

where $\sigma = s_{\min}(A - \lambda I) \leq \epsilon$. Let $w = \frac{\sigma}{\epsilon}u$, noting that $\|w\| \leq 1$.

We define a matrix E such that :

$$Ex = -wv^*x \quad \forall x \in \mathbb{C}^n$$

This gives $Ev = -w$ and $\|E\| \leq 1$ since:

$$\|Ex\| = \|wv^*x\| \leq \|w\|\|v\|\|x\| \leq \|x\|$$

Given that we have

$$(A + \epsilon E - \lambda I)v = (A - \lambda I)v + \epsilon Ev = \sigma u - \epsilon w = 0$$

Thus $\lambda \in \sigma(A + \epsilon E)$ with $\|E\| \leq 1$.

($\Leftarrow$) Suppose $\lambda \in \sigma(A + \epsilon E)$ for some E with $\|E\| \leq 1$. Then there exists $v \neq 0$ such that:

$$(A + \epsilon E - \lambda I)v = 0 \implies (A - \lambda I)v = -\epsilon Ev \Rightarrow \|(A - \lambda I)v\| = \epsilon\|Ev\| \leq \epsilon\|E\|\|v\| \leq \epsilon\|v\|$$

Therefore:

$$s_{\min}(A - \lambda I) \leq \left\|(A - \lambda I)\frac{v}{\|v\|}\right\| \leq \epsilon$$

□

4.3 Remarks

Remark 4.4. *The proof shows that rank-1 perturbations are sufficient to achieve any point in the ϵ-pseudospectrum.*

Remark 4.5. *The minimal perturbation corresponds to the smallest singular value and its singular vectors.*

Remark 4.6. *For normal matrices, $s_{\min}(A - zI) = \text{dist}(z, \sigma(A))$.*

4.4 Behavior for Normal Matrices

Proposition 4.7. *Let A be a normal matrix (i.e., $A^*A = AA^*$). Then, for any $\epsilon > 0$, the ϵ-pseudospectrum $\sigma_\epsilon(A)$ is the ϵ-neighborhood of the spectrum $\sigma(A)$:*

$$\sigma_\epsilon(A) = \{z \in \mathbb{C} \mid \text{dist}(z, \sigma(A)) \leq \epsilon\}.$$

Moreover, the worst-case rate at which eigenvalues can move under perturbations E with $\|E\| \leq \epsilon$ is 1.

Proof. By the spectral theorem, A admits a diagonalization of the form

$$A = U\Lambda U^*,$$

where U is unitary and $\Lambda = \text{diag}(\lambda_1, \ldots, \lambda_n)$. The resolvent $(zI - A)^{-1}$ is then given by

$$(zI - A)^{-1} = U \cdot \text{diag}\left(\frac{1}{z - \lambda_1}, \ldots, \frac{1}{z - \lambda_n}\right) \cdot U^*.$$

Taking the operator norm, we obtain

$$\left\|(zI - A)^{-1}\right\|_2 = \max_i \left|\frac{1}{z - \lambda_i}\right| = \frac{1}{\min_i |z - \lambda_i|} = \frac{1}{\text{dist}(z, \sigma(A))}.$$

By definition, $z \in \sigma_\epsilon(A)$ if and only if $\left\|(zI - A)^{-1}\right\|_2 > \epsilon^{-1}$, which is equivalent to $\text{dist}(z, \sigma(A)) < \epsilon$. If we take the closure, it gives the stated equality. □

4.5 Utmost Right Pseudospectral Abscissa

The rightmost boundary,

$$\alpha_\epsilon(A) = \sup\{\Re(z) : z \in \sigma_\epsilon(A)\},$$

measures the worst-case stability under perturbations $\|E\| \leq \epsilon$. It satisfies:

$$\alpha_\epsilon(A) = \inf\{\alpha(A+E) : \|E\| \leq \epsilon\}$$

4.6 Pseudospectral Radius

The ϵ-pseudospectral radius $\rho_\epsilon(A)$ is defined as:

$$\rho_\epsilon(A) := \max\{|\lambda| : \lambda \in \sigma_\epsilon(A)\},$$

It is the worst case spectral radius under perturbations.

4.7 Transient Growth

4.7.1 Continuous-Time Systems ($\dot{x} = Ax$)

For a linear system:

$$\frac{dx}{dt} = Ax, \quad A \in \mathbb{C}^{n\times n},$$

the maximal transient growth is:

$$G_{\max}^{\text{cont}} := \sup_{t\geq 0} \|e^{At}\|.$$

4.7.2 Discrete-Time Systems ($x_{k+1} = Ax_k$)

For a linear iteration:

$$x_{k+1} = Ax_k, \quad A \in \mathbb{C}^{n\times n},$$

the maximal transient growth is:

$$G_{\max}^{\text{disc}} := \sup_{k\geq 0} \|A^k\|.$$

4.8 The use of pseudosabscissa and pseudoradius

The pseudospectral abscissa $\alpha_\epsilon(A)$ and pseudospectral radius $\rho_\epsilon(A)$ are important for both asymptotic stability and transient behavior of linear systems:

4.8.1 Asymptotic Stability

Continuous-time systems: $\alpha_0(A) < 0$ guarantees stability ($\lim_{t\to\infty} \|e^{At}\| = 0$)
Discrete-time systems: $\rho_0(A) < 1$ guarantees stability ($\lim_{k\to\infty} \|A^k\| = 0$)

4.8.2 Transient Growth Connection

For non-normal systems ($AA^* \neq A^*A$) it is possible to have large transient growth $G_{\max}$ even when asymptotically stable and sensitivity to perturbations is revealed by pseudospectra
Continuous-time systems:

$$\sup_{t\geq 0} \|e^{At}\| \geq \frac{e^{\alpha_\epsilon(A)t} - 1}{\epsilon} \quad \text{for some } t > 0$$

Discrete-time systems:

$$\sup_{k\geq 0} \|A^k\| \geq \frac{\rho_\epsilon(A)^k - 1}{\epsilon} \quad \text{for some } k \geq 0$$

5 Plotting Pseudospectra for Normal and Non-Normal Matrices

Normal matrix: the pseudospectra is just at distance ε from the eigenvalue of A (Fig1 is an example of a diagonal matrix).
Toeplitz matrix: This type of matrix can be non-normal so its pseudospectra is more sensitive to perturbations. A special type of Toeplitz matrix is the Grcar matrix. An $n \times n$ Grcar matrix has entries:

$$a_{ij} = \begin{cases} 1 & \text{if } i \leq j \leq i+3 \\ -1 & \text{if } j = i-1 \\ 0 & \text{otherwise} \end{cases}$$

5.1 MATLAB Code Examples

```
% Matrix size
n = 100;

% Define diagonal entries (eigenvalues)
lambda = linspace(-10, 10, n);

% Create diagonal matrix
A = diag(lambda);

% Display a message
fprintf('Diagonal matrix of size %d created.\n', n);

% Run eigtool
eigtool(A);
```

Listing 1: MATLAB code for plotting 100×100 diagonal matrix

```
% Set Grcar matrix size and bandwidth
n = 50;            % matrix size
k = 3;             % number of superdiagonals

% Create the Grcar matrix
A = gallery('grcar', n, k);

% Display matrix information
disp('Grcar matrix:');
disp(A);

% Launch eigtool
eigtool(A);
```

Listing 2: MATLAB code for plotting Grcar matrix

6 Computing $\lambda(t)$: Trajectories Approaching the Boundary of the Pseudospectrum Using the Guglielmi–Lubich Algorithm

The following algorithm computes trajectories of the rightmost eigenvalue $\lambda(t)$ of a perturbed matrix $A+\varepsilon E(t)$, where $E(t)$ is a rank-one matrix evolving under a specific differential equation flow. This method, due to Guglielmi and Lubich method presented in the following paper

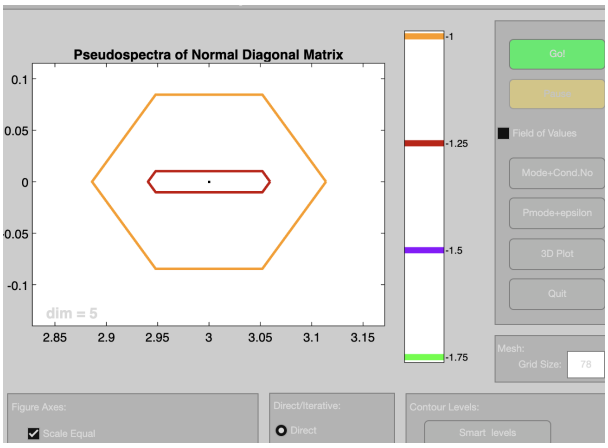

Figure 1: Pseudospectra of normal diagonal matrix plotted in MATLAB

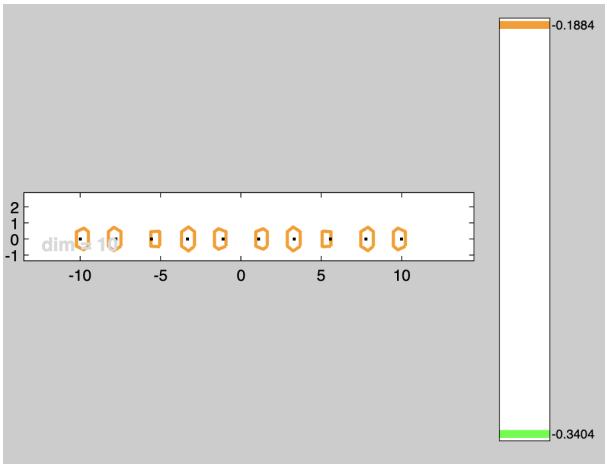

Figure 2: Pseudospectra of 10×10 diagonal matrix with evenly spaced values between -10 and 10

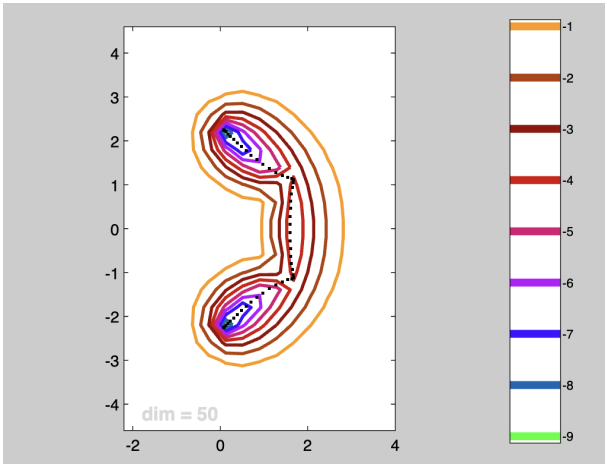

Figure 3: Pseudospectra of 50×50 Grcar matrix with 3 superdiagonals

[2], aims to find points on the ε-pseudospectrum boundary by evolving perturbations in a constrained, optimal manner.

6.1 Main Ideas Behind the Algorithm

The method evolves perturbations of the form $E(t) = u(t)v(t)^*$, constrained to lie on the manifold of unit-norm rank-one matrices:

$$\mathcal{M} = \{uv^* \,|\, u, v \in \mathbb{C}^n,\ \|u\| = \|v\| = 1\}\,.$$

At each time step, we define $B(t) = A + \varepsilon E(t)$, and extract the rightmost eigenvalue $\lambda(t)$ of $B(t)$, along with its corresponding normalized left and right eigenvectors $x(t)$ and $y(t)$. $E(t)$ is given by the differential equation:

$$\dot{E} = P_E(xy^*), \tag{1}$$

where P_E is a projection operator onto the tangent space of $\mathcal{M}$ at the point $E = uv^*$. It is defined as:

$$P_E(Z) = i\,\mathrm{Im}(u^*Zv)\,uv^* + (I - uu^*)Zvv^* + uu^*Z(I - vv^*). \tag{2}$$

Equivalently, the differential equations for $u(t)$ and $v(t)$ are:

$$\frac{du}{dt} = \frac{i}{2}\,\mathrm{Im}(\alpha\,\overline{\beta})\,u + (I - uu^*)\,x\,(y^*v), \tag{3}$$

$$\frac{dv}{dt} = \frac{i}{2}\,\mathrm{Im}(\beta\,\overline{\alpha})\,v + (I - vv^*)\,y\,(x^*u), \tag{4}$$

where $\alpha = u^*x$ and $\beta = v^*y$.

6.2 Mathematical Properties of the Flow

This dynamic system satisfies the following:
Monotonicity: The real part of the tracked eigenvalue increases monotonically, as proved in [2]: $\frac{d}{dt}\mathrm{Re}(\lambda(t)) \geq 0$.
Stationary Points: Points where the flow stagnates correspond to eigenvalues lying on the boundary of the pseudospectrum.
Norm Preservation: The dynamics preserve the unit norm of $u(t)$ and $v(t)$, as they are projected onto the tangent space of the constraint manifold $\mathcal{M}$.
Adjustment: The phase of the eigenvectors x, y is adjusted to ensure that x^*y is real and positive, which aligns the projection with the gradient flow. Otherwise, as explained in the paper [4], we need to rotate the matrix A to e^iA, and apply the algorithm to the rotated matrix. Otherwise, u and v could be rotated and still be eigenvectors of $A + u(t)v(t)$. This would imply $(A + e^i(t)xy^{)y=(t)y}$, so that (t) lies on a circle which isn't always the case.

6.3 Boundary Tracking

The goal of this algorithm is to identify a point on the boundary of the ε-pseudospectrum of A, defined as:

$$\sigma_\varepsilon(A) = \left\{ z \in \mathbb{C} \,\middle|\, \|(zI - A)^{-1}\| \geq \frac{1}{\varepsilon} \right\}.$$

The method simulates how the eigenvalue $\lambda(t)$ of the perturbed matrix $B(t) = A + \varepsilon uv^*$ evolves under perturbations that maximize $\mathrm{Re}(\lambda(t))$. When ε is a simple singular value of $(\lambda I - A)$, then the evolution: $\dot{E} = P_E(xy^*)$ tracks trajectories that remain exactly on the boundary $\partial\sigma_\varepsilon(A)$.

6.4 Algorithm Steps

1. Generate random initial unit vectors:

$$u_0 = \frac{\text{random complex vector}}{\|\text{random complex vector}\|}$$
$$v_0 = \frac{\text{random complex vector}}{\|\text{random complex vector}\|}$$

2. Initialize trajectory storage for $\lambda(t)$

We define a loop for each time step:

1. Form perturbation matrix:

$$E = uv^*$$

2. Construct perturbed matrix:

$$B = A + \epsilon E$$

3. Compute eigenvalues/vectors:

$$\text{Solve } Bx = \lambda x \text{ and } y^*B = \lambda y^*$$

4. Select rightmost eigenvalue:

$$\lambda_0 = _{\lambda \in \sigma(B)} \operatorname{Re}(\lambda)$$

5. Normalize eigenvectors:

$$x = \frac{x}{\|x\|}$$
$$y = \frac{y}{\|y\|}$$

6. Adjust phase of x so that x^*y is real and positive

7. Compute projection coefficients:

$$\alpha = u^*x$$
$$\beta = v^*y$$

8. Compute derivatives:

$$\frac{du}{dt} = \frac{i}{2}\operatorname{Im}(\alpha\overline{\beta})u + (I - uu^*)x(y^*v)$$
$$\frac{dv}{dt} = \frac{i}{2}\operatorname{Im}(\beta\overline{\alpha})v + (I - vv^*)y(x^*u)$$

9. Update vectors (Euler step):

$$u_{\text{new}} = u + h\frac{du}{dt}$$
$$v_{\text{new}} = v + h\frac{dv}{dt}$$
$$u = \frac{u_{\text{new}}}{\|u_{\text{new}}\|}$$
$$v = \frac{v_{\text{new}}}{\|v_{\text{new}}\|}$$

10. Store eigenvalue λ_0 in trajectory

6.5 Matlab implementation

Here is a MATLAB implementation for computing trajectories that approach the boundary of pseudospectra $\sigma_\epsilon(B)$ for a given matrix B. The algorithm follows a gradient flow approach to find points on the pseudospectral boundary for different values of ϵ: $10^{-0.5}, 10^{-0.3}, 1$.

```
function fig41
% Matrix definition
A = [0.1019, -0.8350, 0.2966, -0.0756, -2.2079, 1.2682;
     1.1813, -1.4224, -0.8664, 0.8003, -1.3413, 1.3547;
    -1.2457, -0.1737, -1.1910, -0.3194, -0.2909, 0.8230;
    -0.7830, -0.5115, -0.0109, 0.8860, 0.4878, 0.3246;
    -0.5740,  0.0268,  1.1950, -0.1729,  0.9966, -0.8003;
    -0.3815, -0.4476, -0.9740,  1.4030,  1.0361,  0.7399];

A = complex(A);
n = size(A, 1);
epsilons = [10^(-0.5), 10^(-0.3), 1];
colors = [0.2 0.8 0.2;   % Light green for 10^-0.5
          0.6 0.2 0.8;   % Purple for 10^-0.3
          1   0.5 0];    % Orange for 1
num_steps = 500;
h = 0.01;

% Plot pseudospectra
opts = struct();
opts.levels = log10(epsilons);
opts.npts = 100;
figure;
eigtool(A, opts);
hold on;

% Plot eigenvalues of A
scatter(real(eig(A)), imag(eig(A)), 40, 'k');

% Main loop for each epsilon
for k = 1:length(epsilons)
    epsilon = epsilons(k);
    color = colors(k, :);

    % Initial random unit vectors u, v
    u = randn(n, 1) + 1i * randn(n, 1);
    v = randn(n, 1) + 1i * randn(n, 1);
    u = u / norm(u);
    v = v / norm(v);

    lambda_traj = zeros(num_steps, 1);

    for step = 1:num_steps
        % Form perturbed matrix
        E = u * v';
        B = A + epsilon * E;

        % Compute eigenvalues and right eigenvectors of B
        [Vr, Dr] = eig(B, 'vector');
        [~, idx] = max(real(Dr));
        lambda = Dr(idx);
        y = Vr(:, idx);   % Right eigenvector

```

```
        % Compute left eigenvectors of B via eig(B')
        [Vl, Dl] = eig(B', 'vector');
        [~, idx_left] = min(abs(Dl - conj(lambda)));
        x = Vl(:, idx_left);   % Left eigenvector

        % Normalize and adjust phase
        y = y / norm(y);
        x = x / norm(x);
        c = x' * y;
        if abs(c) > 1e-14
            x = x * exp(1i * angle(c));
        end

        lambda_traj(step) = lambda;

        % Compute alpha = u'*x, beta = v'*y
        alpha = u' * x;
        beta  = v' * y;

        % Gradient flow
        du = (0.5i) * imag(alpha * conj(beta)) * u + (eye(n) - u * u')
            * (x * (y' * v));
        dv = (0.5i) * imag(beta * conj(alpha)) * v + (eye(n) - v * v')
            * (y * (x' * u));

        % Euler step
        u = u + h * du;
        v = v + h * dv;
        u = u / norm(u);
        v = v / norm(v);
    end

    % Plot trajectory
    plot(real(lambda_traj), imag(lambda_traj), 'Color', color, '
        LineWidth', 2);
end

% Customize the plot
xlabel('Re(\lambda)');
ylabel('Im(\lambda)');
title('Trajectories to Pseudospectral Boundary');
grid on;
end
```

Listing 3: MATLAB Code for computing trajectories that approach boundary of pseudospectra for given matrix A

7 Low rank matrix approximation methods

7.1 Introduction

The subspace method, as presented in **Section 6**, provides an accurate way, as demonstrated in **Figure 4**, for tracking the rightmost point and finding the pseudospectra border. The problem is that its direct implementation can become expensive for large-scale problems. In particular, each Euler step involves forming and manipulating dense perturbation matrices, and since $\dot{E}(t)$ has rank $\text{rank}(E) + 1$, starting with a rank-1 perturbation E_0 will quickly lead to full-rank

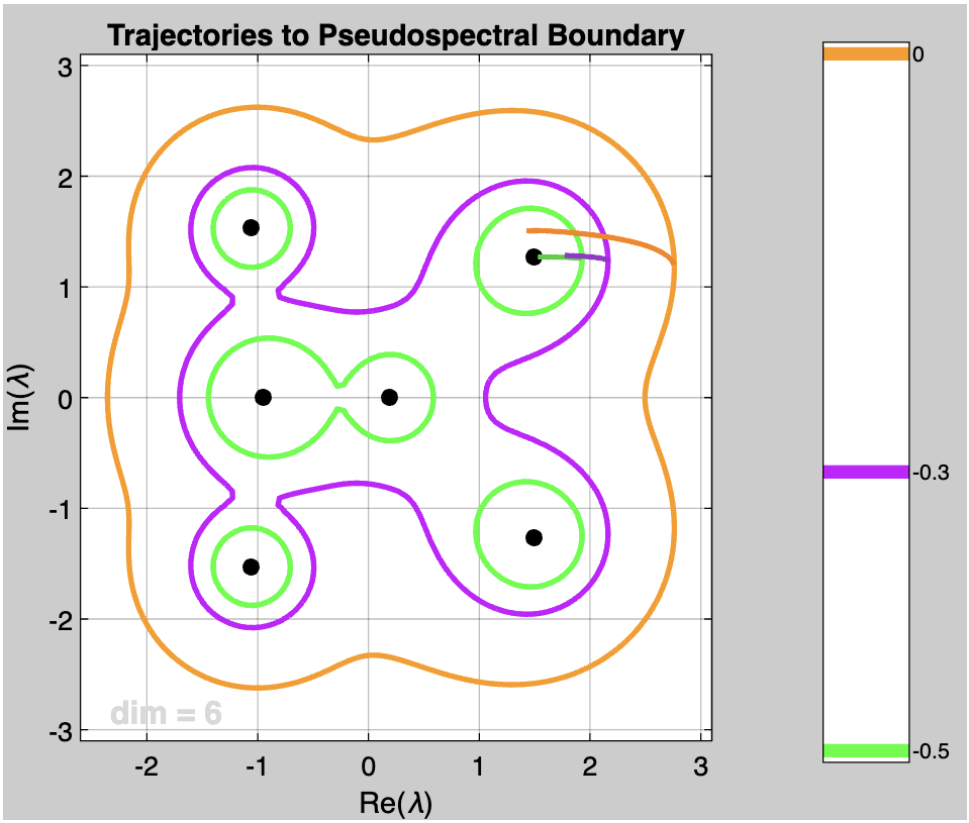

Figure 4: Example trajectory approaching pseudospectral boundary (as in Figure 4.1 [2])

matrices as the integration proceeds.

The solution to this issue is to use low-rank matrix approximation methods that projects the dynamics back onto the manifold of fixed-rank matrices at every step. This reduces cost while preserving the essential structure of the evolution.

Two randomized projection techniques are considered:
Randomized SVD: We want to find a matrix A_k of rank k such that A is approximated the better possible with this matrix of lower rank. It approximates the dominant singular subspace of a matrix via random sampling and gives an explicit low-rank factorization
Generalized Nyström method: It constructs a rank-r approximation $N(Z) = Z\,\Omega\,(\Psi^\top Z\Omega)^\dagger\,\Psi^\top Z$ from random test matrices Ψ and Ω, requiring only $O(r)$ matrix–vector products with Z or $F(Z)$.

7.2 Randomized SVD: The Algorithm

7.2.1 Idea behind randomized SVD

We start by taking a random Gaussian matrix Ω of dimensions $n * (k + p)$, where n is the number of columns in A, and where n is the number of columns in A, k is the wanted rank of the approximation, and we add p, called an oversampling parameter. It is used because it can give additional information about A, and help approximate better than just the k singular vectors of the approximations.
In the random Gaussian matrix, each entry is drawn independently from a standard normal distribution. The random matrix has then a high probability of capturing the structure of A.
Then, we project the columns of A into a lower dimensional subspace given by the columns of Ω.

7.2.2 Why it works

If $\Omega \in \mathbb{R}^{n\times r}$ is Gaussian, then for fixed matrices A, B,

$$\mathbb{E}\left[\|A\Omega B\|_F^2\right] = \|A\|_F^2 \cdot \|B\|_F^2$$

It justifies using Gaussian matrices to randomly sample the column space of A. The optimal rank-k approximation is given by the truncated SVD:

$$A \approx U_k \Sigma_k V_k^T, \quad \text{where } \Sigma_k = \text{diag}(\sigma_1, \ldots, \sigma_k).$$

With

$$\|A - U_k\Sigma_k V_k^T\|_2 = \sigma_{k+1}, \quad \|A - U_k\Sigma_k V_k^T\|_F = \sqrt{\sum_{i=k+1}^{n} \sigma_i^2}.$$

7.2.3 The algorithm

Let $A \in \mathbb{R}^{m\times n}$, and suppose we want a low-rank approximation of rank k. The algorithm of randomized SVD is the following:

1. We start by generating a Gaussian random matrix $\Omega \in \mathbb{R}^{n\times(r+p)}$
2. Then, we the matrix $Y = A\Omega \in \mathbb{R}^{m\times(r+p)}$.
3. We compute the QR decomposition: $Y = QR$, where $Q \in \mathbb{R}^{m\times(r+p)}$ has orthonormal columns.
4. We form $B = Q^T A \in \mathbb{R}^{(r+p)\times n}$.
5. The approximate matrix is $\hat{A} = QB \in \mathbb{R}^{m\times n}$.

This algorithm gives a rank-$r + p$ approximation of A.

7.3 Randomized SVD Error Comparison

7.3.1 Introduction

Here, we compare three different ways to implement the randomized SVD algorithm, the first method using no oversampling, but then using the rank r approximation of B when calculating the approximate matrix A. In the second subgraph (second method), we use oversampling to get a more precise result. Then, in the last subgraph, we use (as explained in the slides) Tr(B) to get the approximate A. We then see that the error bound is low in the three cases, even if it is much less precise (but demands much less calculation, since there is no explicit SVD to find the rank r approximation of B) than the first two graphs. In each graph, we compare the error for different types of matrices: symmetric matrices, Grcar (a particular form of toeplitz matrix, so non normal matrix), and a diagonal matrix.

7.3.2 Developping the different methods

Randomized SVD without oversampling: This first method projects A onto a lower-dimensional subspace using a random Gaussian matrix without oversampling. We use a rank-r approximation by computing the SVD of the projected matrix.

Randomized SVD with oversampling: This second method extends the first by adding a small oversampling parameter p, choosen as 5 in the graphs which improves the the approximation, and we see a different in the error especially for matrices with slowly decaying singular values.

7.3.3 Python implementation

```
1
2  import numpy as np
3  import matplotlib.pyplot as plt
4  from scipy.linalg import qr, svd, toeplitz
5
6
7  def randomized_svd_oversampled(A, rank, p=5):
8      m, n = A.shape
9      rank = min(rank, min(m, n))
10     Omega = np.random.randn(n, rank + p)
11     Y = A @ Omega
12     Q, _ = qr(Y, mode='economic')
13     B = Q.T @ A
14     U, s, V = svd(B, full_matrices=False)
15     return Q @ U[:, :rank], s[:rank], V[:rank, :]
16
17 def simple_randomized_svd(A, rank):
18     m, n = A.shape
19     rank = min(rank, min(m, n))
20     Omega = np.random.randn(n, rank)
21     Y = A @ Omega
22     Q, _ = qr(Y, mode='economic')
23     B = Q.T @ A
24     U, s, V = svd(B, full_matrices=False)
25     return Q @ U[:, :rank], s[:rank], V[:rank, :]
26
27 def trace_approximation(A, rank):
28     m, n = A.shape
29     rank = min(rank, min(m, n))
30     Omega = np.random.randn(n, rank)
31     Y = A @ Omega
32     Q, _ = qr(Y, mode='economic')
33     B = Q.T @ A
34     trace_B = np.trace(B)
35     # Scale projection matrix Q Q^T by trace(B)
36     A_approx = (trace_B / rank) * (Q @ Q.T)
37     return A_approx
38
39 # Test matrices
40 n = 50
41 A_sym = np.random.randn(n, n)
42 A_sym = (A_sym + A_sym.T) / 2
43
44 def grcar_matrix(n):
45     c = np.zeros(n)
46     r = np.zeros(n)
47     r[0:4] = 1
48     c[0] = 1
49     return toeplitz(c, r)
50
51 A_grcar = grcar_matrix(n)
52 A_diag = np.diag(np.linspace(1, 0.1, n))
53
54 test_matrices = {
55     "Symmetric matrix": A_sym,
56     "Grcar matrix": A_grcar,
```

```
    "Diagonal": A_diag
}

# Use ranks safely within dimensions
ranks = np.arange(10, min(n, 51), 20)  # max rank capped at 50 (matrix size)

for rank in ranks:
    errors_simple = {}
    errors_oversampled = {}

    for name, A in test_matrices.items():
        U, s, V = simple_randomized_svd(A, rank)
        A_approx = U @ np.diag(s) @ V
        errors_simple[name] = np.linalg.norm(A - A_approx, 'fro') / np.linalg.norm(A, 'fro')

        U_o, s_o, V_o = randomized_svd_oversampled(A, rank, p=5)
        A_approx_o = U_o @ np.diag(s_o) @ V_o
        errors_oversampled[name] = np.linalg.norm(A - A_approx_o, 'fro') / np.linalg.norm(A, 'fro')

    fig, axes = plt.subplots(1, 3, figsize=(18, 5))

    axes[0].bar(errors_simple.keys(), errors_simple.values(), color='skyblue', width=width)
    axes[0].set_title("Randomized SVD (No Oversampling)")
    axes[0].set_ylim(0, 1)
    axes[0].set_ylabel("Relative Frobenius Norm Error")

    axes[1].bar(errors_oversampled.keys(), errors_oversampled.values(), color='purple', width=width)
    axes[1].set_title("Randomized SVD (With Oversampling p=5)")
    axes[1].set_ylim(0, 1)

    plt.suptitle(f"Randomized SVD Error Comparison (rank = {rank})", fontsize=16)
    plt.show()
```

Listing 4: SVD error comparaison Python code

7.4 Oversampling

The matrix $\hat{A}$ have rank $k+p$, but we want exactly rank k. Thus, we need to compute a rank-k truncation: Let $T_k(B)$ be the best rank-k approximation to B. Then we define: $\tilde{A} = QT_k(B)$, which has rank k.

7.5 More details on the projection

7.5.1 Orthogonal Projectors

Let $V \subseteq \mathbb{R}^n$ be a subspace. The orthogonal projector Π onto V is defined by:

$$\Pi = QQ^T, \quad \text{where } Q \text{ is an orthonormal basis for } V$$

Orthogonal projectors satisfy $\Pi^2 = \Pi$, $\Pi^T = \Pi$, and for any x, $\|x - \Pi x\|^2 = \min_{v \in V} \|x - v\|^2$

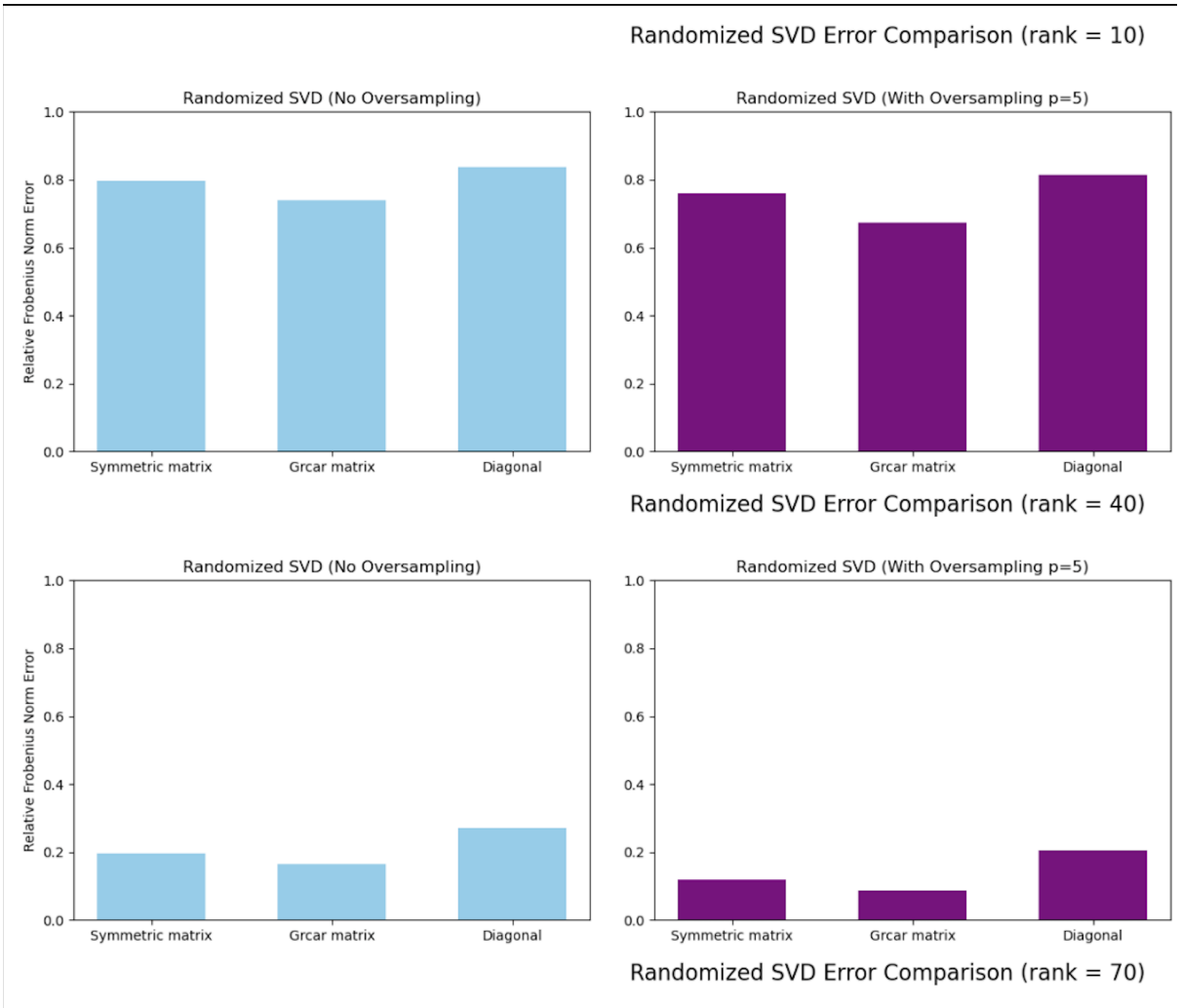

Figure 5: Randomized SVD error comparaison plot on Python

For any x, we have the Pythagorean identity:

$$\|x\|^2 = \|\Pi x\|^2 + \|x - \Pi x\|^2$$

7.5.2 Oblique Projectors

An oblique projector projects onto a subspace V, but not orthogonally. It takes the form:

$$\tilde{\Pi} = V(W^T V)^{-1} W^T$$

where:

$V, W \in \mathbb{R}^{n\times r}$ are full-rank, V spans the target space, W defines the directions along which projection is made.

Oblique projectors satisfies: $\tilde{\Pi}^2 = \tilde{\Pi}$

7.5.3 Optimality of $QQ^T A$

The projection minimizes:

$$\|A - QQ^T A\|_F = \min_{\text{rank}(B)\leq k} \|A - B\|_F.$$

Indeed, $A = QQ^T A + (I - QQ^T)A$. The residual $(I - QQ^T)A$ is orthogonal to Q. We need the oblique projection form to bound the error, because it is easier to deal with given that it projects into the span of a vector space V.

7.6 Generalized Nyström method

7.6.1 Idea behind the algorithm

The generalized Nyström method is presented in details in the paper [3]. It constructs a rank-r approximation $\widetilde{Z}$ of a given matrix $Z \in \mathbb{C}^{m\times n}$ from matrix-vector products. We approximate both the column and row spaces of Z using a small number of randomly sampled basis vectors. To do this, we select two independent random test matrices

$$\Omega \in \mathbb{C}^{n\times r}, \quad \Psi \in \mathbb{C}^{m\times r},$$

which are the directions for the columns and rows of Z. This construction ensures that $\widetilde{Z}$ has rank at most r.

The plain generalized Nyström method relies on the direct inversion of the small $r \times r$ matrix R obtained from the QR factorization of Y^*AX. This approach isn't numerically stable when R is ill-conditioned. This is why we have the stabilized gneralized Nyström method which replaces R^{-1} with a regularized pseudoinverse $R_\varepsilon^\dagger$, where only singular values above a defined threshold ε are inverted.

7.6.2 The algorithm

1. We start by fixing the target rank r and oversampling $\ell > 0$ choosen as $\ell = \lceil 0.5\, r \rceil$. : Then, we create random test matrices

$$X \in \mathbb{C}^{n\times r}, \qquad Y \in \mathbb{C}^{m\times(r+\ell)}.$$

2. We compute

$$C := AX \in \mathbb{C}^{m\times r}, \qquad W := Y^*A \in \mathbb{C}^{(r+\ell)\times n},$$

and the reduced QR factorization of the $(r + \ell) \times r$ matrix,

$$S := Y^*AX = QR, \quad Q \in \mathbb{C}^{(r+\ell)\times r} \text{ with } Q^*Q = I_r, \quad R \in \mathbb{C}^{r\times r} \text{ upper triangular.}$$

3. For the plain generalized Nystrom, we set

$$\widehat{A}_r \;:=\; C\,R^{-1}\,(Q^*W).$$

4. For the stabilized generalized Nyström, we replace R^{-1} by an ε pseudoinverse $R_\varepsilon^\dagger$ which invert only singular values $> \varepsilon$, and set

$$\widehat{A}_r \;:=\; C\,R_\varepsilon^\dagger\,(Q^*W).$$

7.7 Singular values and low rank approximation

Plotting the singular values of a matrix A before running any low-rank approximation algorithm tells us what the best possible error is for any rank-r approximation, for any randomization method that we implement afterwards.

Suppose the singular values of A are

$$\sigma_1 \geq \sigma_2 \geq \sigma_3 \geq \cdots \geq 0.$$

The optimal rank-r approximation to A, is given by truncating the singular value decomposition (SVD) of A to its top r singular values and vectors.

The Frobenius norm error of this optimal rank-r approximation is

$$\|A - A_r^{\text{opt}}\|_F = \sqrt{\sigma_{r+1}^2 + \sigma_{r+2}^2 + \cdots}.$$

7.7.1 The algorithm comparing low rank approximation methods in Python

```python
import numpy as np
import scipy.linalg as sc
import matplotlib.pyplot as plt

#Plain generalized Nystrom

def generalized_nystrom(A, r, l=None):
    if l is None:
        l = int(np.ceil(0.5 * r))
    m, n = A.shape
    X = np.random.randn(n, r)
    Y = np.random.randn(m, r + l)
    AX = A @ X
    Z = Y.T @ A
    Q, R = sc.qr(Z @ X, mode='economic')
    return AX @ sc.solve(R, Q.T @ Z)

def randomized_svd_oversampled(A, rank, p=5):
    m, n = A.shape
    Omega = np.random.randn(n, rank + p)
    Y = A @ Omega
    Q, _ = sc.qr(Y, mode='economic')
    B = Q.T @ A
    U, s, Vt = sc.svd(B, full_matrices=False)
    return (Q @ U[:, :rank]) @ np.diag(s[:rank]) @ Vt[:rank, :]

# Randomized SVD without oversampling

def simple_randomized_svd(A, rank):
```

```
    m, n = A.shape
    Omega = np.random.randn(n, rank)
    Y = A @ Omega
    Q, _ = sc.qr(Y, mode='economic')
    B = Q.T @ A
    U, s, V = sc.svd(B, full_matrices=False)
    return (Q @ U[:, :rank]) @ np.diag(s[:rank]) @ V[:rank, :]


# Frobenius error
def frob_err(A, B):
    return np.linalg.norm(A - B, ord='fro')

# Generate exponential spectrum matrix
def make_exp_spectrum_matrix(n, R, q):
    sing_vals = np.concatenate([
        np.ones(R),
        10 ** (-q * np.arange(1, n - R + 1))
    ])
    U, _ = np.linalg.qr(np.random.randn(n, n))
    V, _ = np.linalg.qr(np.random.randn(n, n))
    return U @ np.diag(sing_vals) @ V.T

# Run and plot for one matrix
def run(A, title):
    # SVD and sort singular values in descending order
    _, s_full, _ = np.linalg.svd(A, full_matrices=False)
    s_full = np.sort(s_full)[::-1]

    ranks = list(range(1, min(A.shape) + 1))
    errorsgn = []
    errorsos = []
    errorss = []
    errorsopt = []

    for r in ranks:
        Agn = generalized_nystrom_plain(A, r)
        errorsgn.append(frob_err(A, Agn))

        Aos = randomized_svd_oversampled(A, r, p=2)
        errorsos.append(frob_err(A, Aos))

        As = simple_randomized_svd(A, r)
        errorss.append(frob_err(A, As))

        errorsopt.append(np.sqrt(np.sum(s_full[r:] ** 2)))

    plt.figure(figsize=(7, 5))
    plt.semilogy(ranks, errorsopt, 'k--', label='Optimal SVD')
    plt.semilogy(ranks, errorsgn, 'o-', label='General Nystr m')
    plt.semilogy(ranks, errorsos, 's-', label='Randomized SVD +
       oversampling')
    plt.semilogy(ranks, errorss, 'd-', label='Randomized SVD')
    plt.xlabel('Rank')
    plt.ylabel(r'$\|A - \hat{A}_r\|_F$')
    plt.title(title)
    plt.grid(True, which='both', ls=':')
    plt.legend()
```

```
    plt.show()

# Test matrices


# A1: Rank-1 matrix
A1 = np.array([
    [1., 2., 3., 4., 5.],
    [2., 4., 6., 8., 10.],
    [3., 6., 9., 12., 15.],
    [4., 8., 12., 16., 20.],
    [5., 10., 15., 20., 25.]
])

# A2: Diagonal matrix
A2 = np.diag([5, 4, 3, 2, 1])

# A3: Small exponential spectrum
A3 = make_exp_spectrum_matrix(n=5, R=2, q=0.5)

# A4: Large exponential spectrum
A4 = make_exp_spectrum_matrix(n=1000, R=10, q=0.1)


# Run experiments
run(A1, "Rank-1 Matrix")
run(A2, "Diagonal Matrix")
run(A3, " Exponential Spectrum (n=5, R=2, q=0.5)")
run(A4, "Big Exponential Spectrum (n=1000, R=10, q=0.1)")
```

Listing 5: Low rank approximation method error comparaison Python implementation

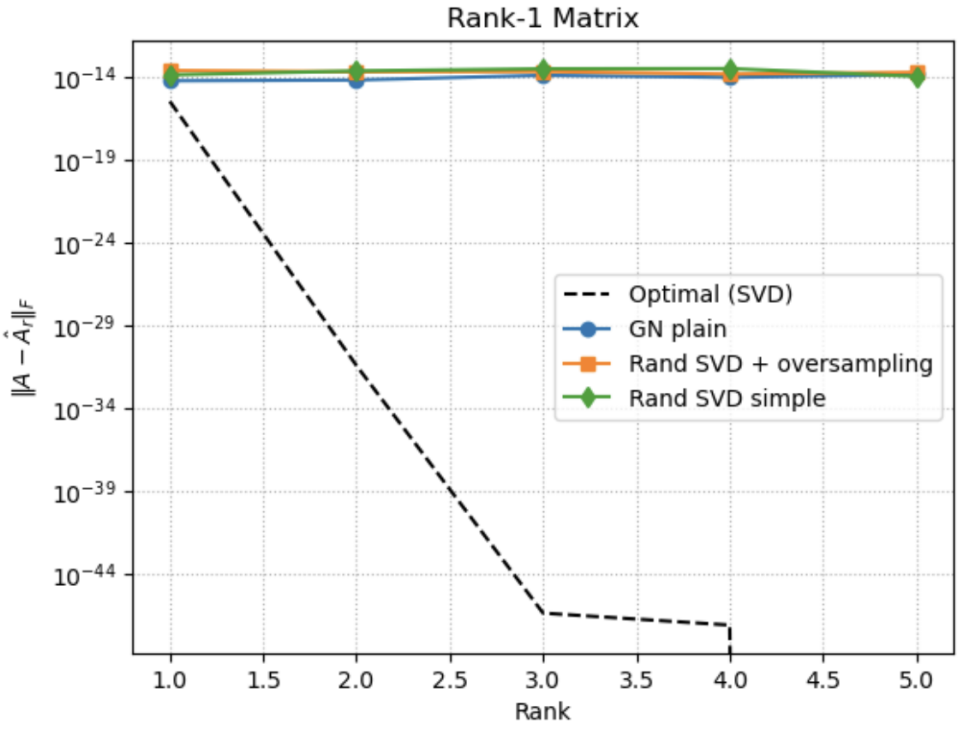

Figure 6: Rank 1 matrix low rank approximation method comparaison

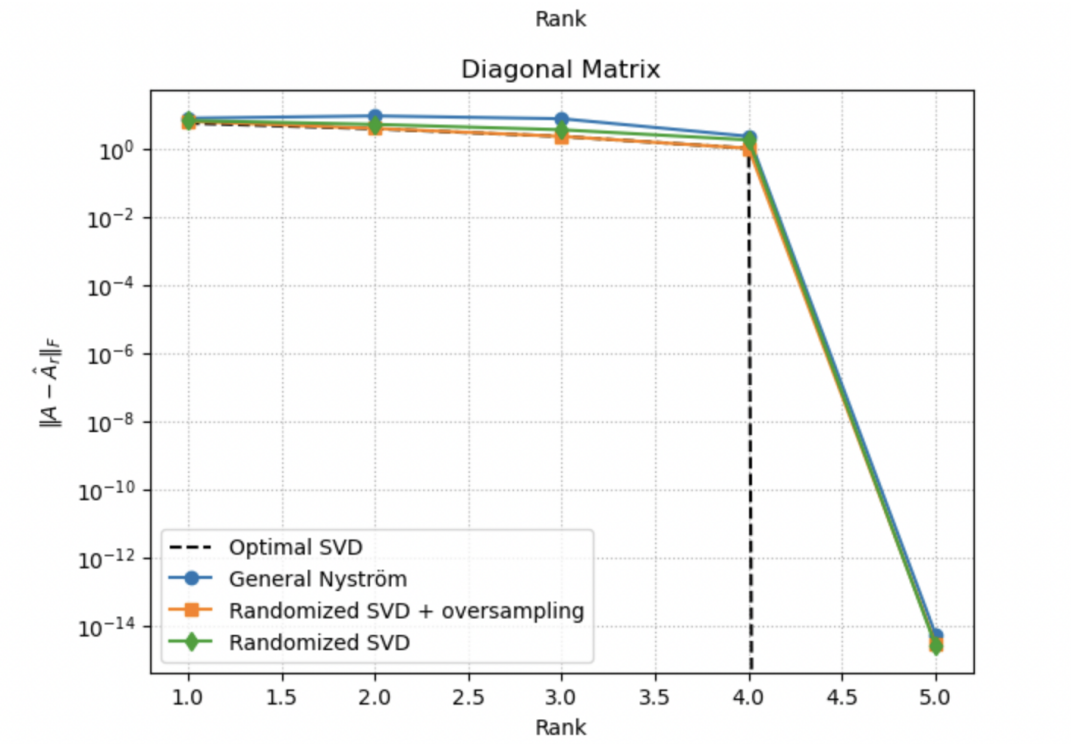

Figure 7: Diagonal matrix low rank approximation method comparaison

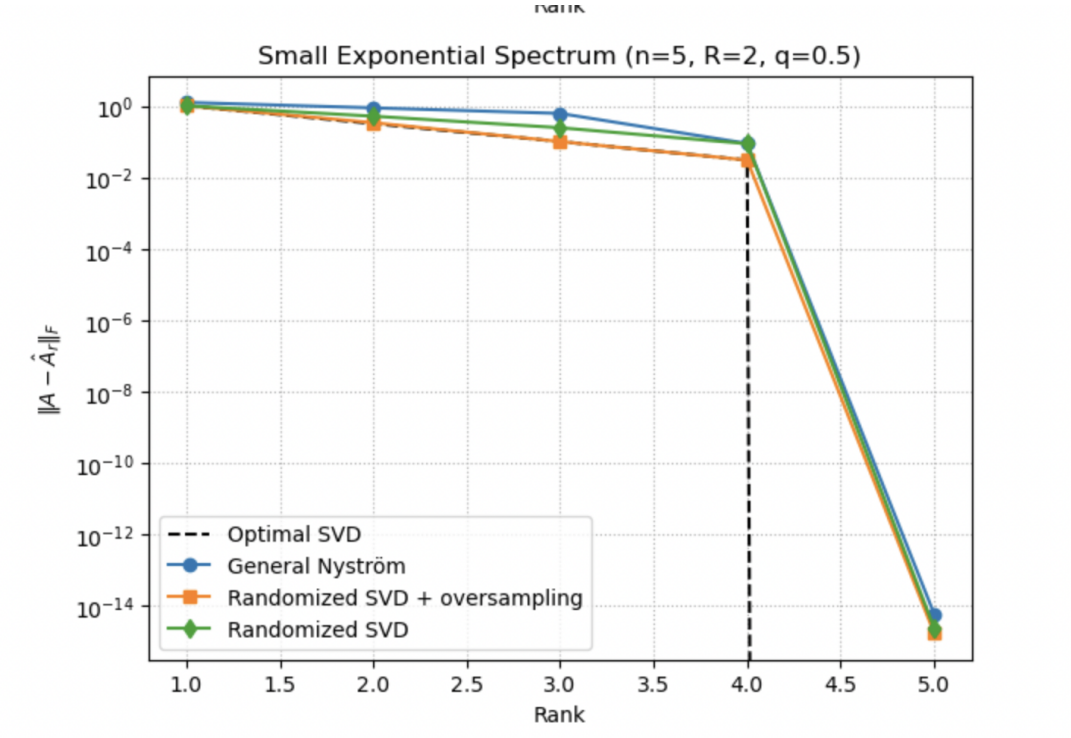

Figure 8: Small exponential spectrum matrix low rank approximation method comparaison

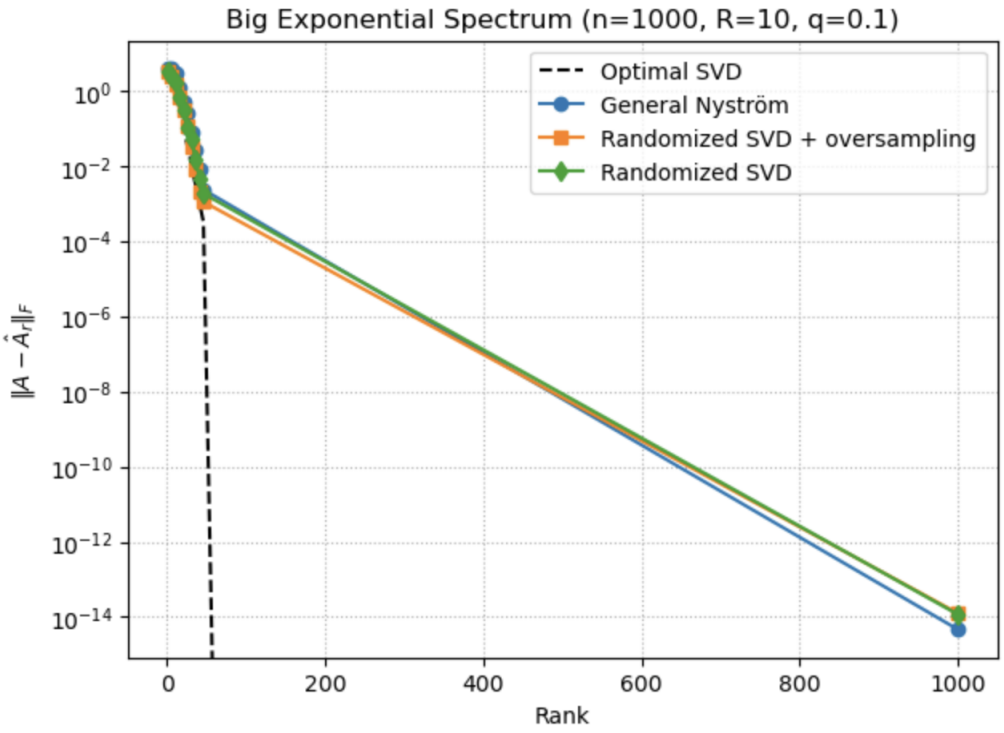

Figure 9: Big exponential spectrum matrix low rank approximation method comparaison

8 Randomized Euler algorithm

8.1 Standard Euler method

The standard Euler method for matrix differential equations updates the solution as $X_{n+1} = X_n + hF(X_n), \quad X_0 = A_0$ It is the method we used in the subspace algorithm in part 6. But as explained before, the matrix produced by each step increases of rank until being full rank which is really costly.

8.2 The truncated SVD Euler method

The truncated SVD Euler method of matrix differential equations updates the solutions as:

$$X_{n+1} = \mathcal{T}_r(X_n + hF(X_n)), \quad X_0 = \mathcal{T}_r(A_0)$$

where $\mathcal{T}_r$ denotes truncation to rank-r via SVD. However, this approach is also quite costly. The intermediate matrix $X_n + hF(X_n)$ have rank that can get bigger and bigger, making the SVD truncation expensive ($O(n^3)$ for $n \times n$ matrices).

8.3 Randomized Euler with Nyström Projection

The randomized Euler method of matrix differential equations updates the solutions as:

$$X_{n+1} = \mathcal{N}_n(X_n + hF(X_n)), \quad X_0 = \mathcal{N}_0(A_0)$$

where $\mathcal{N}_i(Z) = Z\Omega_i(\Psi_i^\top Z\Omega_i)^\dagger \Psi_i^\top Z$ is a randomized rank-r Nyström projection using independent random matrices Ω_i, Ψ_i at each step. The intermediate matrix $X_n + hF(X_n)$ usually has high rank, making the truncated SVD costly. It is the method presented in the paper [8].

8.4 Choosing low rank approximation method to use

When applying Euler's step, we need a low-rank approximation method that maintains a fixed rank throughout the integration of the matrix differential equation. The generalized Nyström approximation is better to use than truncated SVD because it directly produces a rank-r matrix and avoids intermediate higher-rank factors and truncation steps.

8.5 Python implementation

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import norm, svd
import scipy.linalg as sc

# Parameters
alpha = 1.0
T = 1.0
n = 10
t_points = 10
x = np.linspace(-1, 1, t_points)
y = np.linspace(-1, 1, t_points)

# Construct C as in the paper
C = np.zeros((n, n))
for k in range(1, 12):
    for i in range(n):
        for j in range(n):
            C[i, j] += 10**(-(k-1)) * np.exp(-k * (x[i]**2 + y[j]**2))

# Normalize C
C = C / norm(C, 'fro')

# Define L
L = np.diag([1, -2, 1] + [0]*(n-3))  # expand to 128x128
# To make L symmetric and well-defined, we set diag with first 3
    entries then pad
L = np.zeros((n, n))
L[0, 0] = 1
L[1, 1] = -2
L[2, 2] = 1

# Initial A0 (zero matrix)
A0 = np.zeros((n, n))

# RHS of the Lyapunov ODE
def f(A):
    return L @ A + A @ L + alpha * C

# Reference solution using RK4
def rk4(f, A0, T, h):
    steps = int(T / h)
    A = A0.copy()
    for _ in range(steps):
        k1 = f(A)
        k2 = f(A + 0.5*h*k1)
        k3 = f(A + 0.5*h*k2)
        k4 = f(A + h*k3)
```

```
        A = A + (h/6.0)*(k1 + 2*k2 + 2*k3 + k4)
    return A

# Compute reference solution
h_ref = 0.001
X_ref = rk4(f, A0, T, h_ref)

# Generalized Nystr m approximation
def generalized_nystrom_plain(A, r, l=None):
    if l is None:
        l = int(np.ceil(0.5 * r))
    m, n = A.shape
    X = np.random.randn(n, r)
    Y = np.random.randn(m, r + l)
    AX = A @ X
    Z = Y.T @ A
    Q, R = sc.qr(Z @ X, mode='economic')
    return (AX @ sc.inv(R)) @ (Q.T @ Z)

# Randomized  E u l e r Nystrm  scheme
def randomized_euler_nystrom(f, A0, T, h, r):
    steps = int(T / h)
    n = A0.shape[0]
    A = A0.copy()
    for _ in range(steps):
        F = f(A)
        A += h*F
        A =  generalized_nystrom_plain(A, r)
    # truncate to rank-r
    U, s, Vt = svd(A, full_matrices=False)
    U_r = U[:, :r] * np.sqrt(s[:r])[None, :]
    V_r = Vt[:r, :].T * np.sqrt(s[:r])[None, :]
    return U_r @ V_r.T

# Run error analysis
time_steps = np.logspace(-3, -1, 5)
ranks = [1]  # rank-1 approximation
errors_euler_nystrom = {r: [] for r in ranks}

for h in time_steps:
    for r in ranks:
        X_nys = randomized_euler_nystrom(f, A0, T, h, r)
        error = norm(X_nys - X_ref, 'fro') / norm(X_ref, 'fro')
        errors_euler_nystrom[r].append(error)

# Plot results
for r in ranks:
    plt.loglog(time_steps, errors_euler_nystrom[r], marker="o")
plt.xlabel('Time step h', fontsize=12)
plt.ylabel('Relative Frobenius error', fontsize=12)
plt.title('Randomized Euler error analysis with differential Lyapunov
   matrices')
plt.show()
```

Listing 6: Python implementation of randomized Euler for Lyapunov matrix differential equation

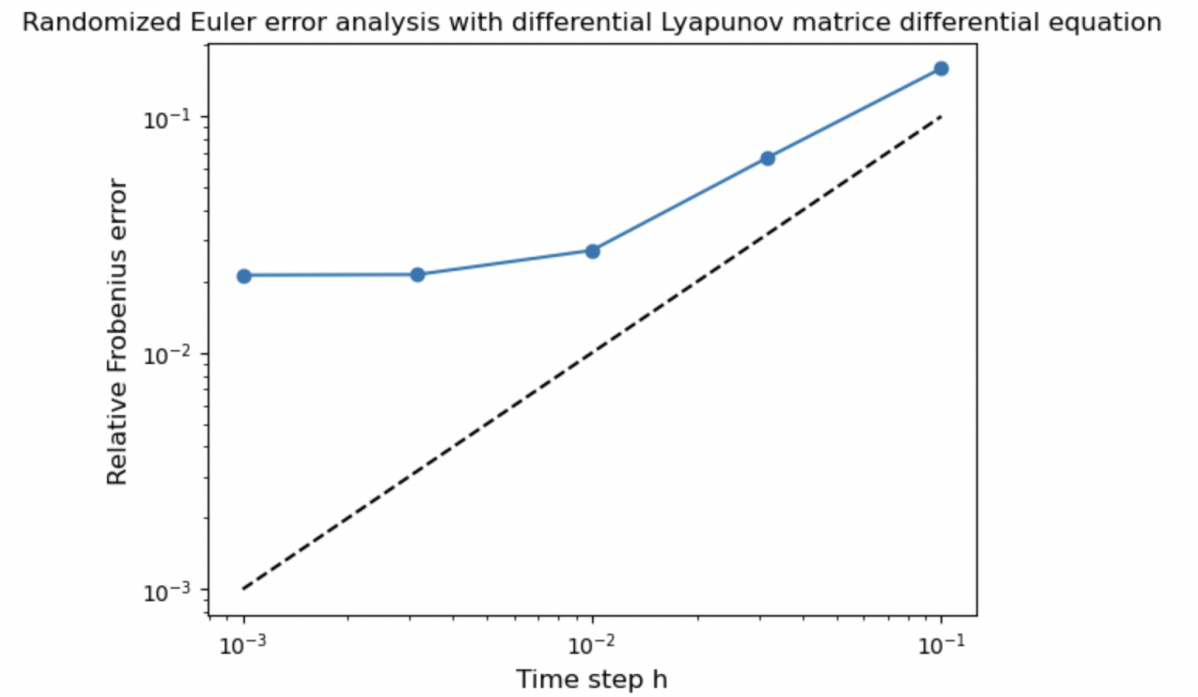

Figure 10: Randomized Euler method applied to Lyapunov matrices differential equation with $\alpha = 1$

8.6 Implementing randomized Euler with Nyström on Guglielmi-Lubich algorithm for plotting pseudospectra

8.7 MATLAB implementation

```
function fig41_nystrom_matrix

    % Parameters
    A = [0.1019 , -0.8350 , 0.2966 , -0.0756 , -2.2079 , 1.2682;
         1.1813 , -1.4224 , -0.8664 , 0.8003 , -1.3413 , 1.3547;
        -1.2457 , -0.1737 , -1.1910 , -0.3194 , -0.2909 , 0.8230;
        -0.7830 , -0.5115 , -0.0109 , 0.8860 , 0.4878 , 0.3246;
        -0.5740 ,  0.0268 ,  1.1950 , -0.1729 , 0.9966 , -0.8003;
        -0.3815 , -0.4476 , -0.9740 ,  1.4030 , 1.0361 , 0.7399];
    A = complex(A);
    n = size(A,1);

    epsilons = [10^(-0.5), 10^(-0.3), 1];
    colors   = [0.2 0.8 0.2;
                0.6 0.2 0.8;
                1.0 0.5 0.0];

    num_steps = 500;
    h = 0.01;           % Euler step size
    r = 1;              % Nystr m rank
    do_norm_fro = true; % keep E bounded

    % Plot pseudospectra
    opts = struct();
    opts.levels = log10(epsilons);
    opts.npts   = 100;
```

```
    figure; eigtool(A, opts); hold on;
    scatter(real(eig(A)), imag(eig(A)), 40, 'k');

    for k = 1:length(epsilons)
        epsilon = epsilons(k);
        col = colors(k,:);

        % Initial rank-1 perturbation
        u = randn(n,1) + 1i*randn(n,1); u = u / norm(u);
        v = randn(n,1) + 1i*randn(n,1); v = v / norm(v);
        E = u*v';               % starting E
        if do_norm_fro, E = E / norm(E,'fro'); end

        lambda_traj = zeros(num_steps,1);


        % Time-stepping loop

        for step = 1:num_steps
            % Perturbed matrix
            B = A + epsilon * E;

            % Dominant eigenpair (max real part)
            [Vr,Dr] = eig(B,'vector');
            [~,idx] = max(real(Dr));
            lambda = Dr(idx);
            y = Vr(:,idx);

            [Vl,Dl] = eig(B','vector');
            [~,idxL] = min(abs(Dl - conj(lambda)));
            x = Vl(:,idxL);

            % Normalize and align phase
            y = y / norm(y);
            x = x / norm(x);
            c = x' * y;
            if abs(c) > 1e-14
                x = x * exp(1i*angle(c));
            end

            lambda_traj(step) = lambda;


            % Ambient-space Nystr m Euler update

            Ey = x*y';
            Ey = E + h*Ey;
            E = generalized_nystrom_plain(Ey, r);   % Nystr m
            E = E / norm(E,'fro');

        end

        % Plot eigenvalue trajectory
        plot(real(lambda_traj), imag(lambda_traj), 'Color', col, '
           LineWidth', 2);
    end

    xlabel('Re(\lambda)'); ylabel('Im(\lambda)');
```

```
    title('Trajectories to pseudospectral boundary (ambient Nystr m
        Euler)');
    grid on;
end


% Generalized Nystr m
function A_approx = generalized_nystrom_plain(A, r, l)
    if nargin < 3, l = ceil(0.5*r); end
    [m,n] = size(A);
    X = randn(n,r);
    Y = randn(m,r+l);
    AX = A*X;
    Z  = Y'*A;
    [Q,R] = qr(Z*X,0);
    A_approx = (AX / R) * (Q'*Z);
end
```

Listing 7: MATLAB code for randomized Euler with Nyström on Gugliemi-Lubich algorithm for plotting pseudospectra with a random matrix A predefined

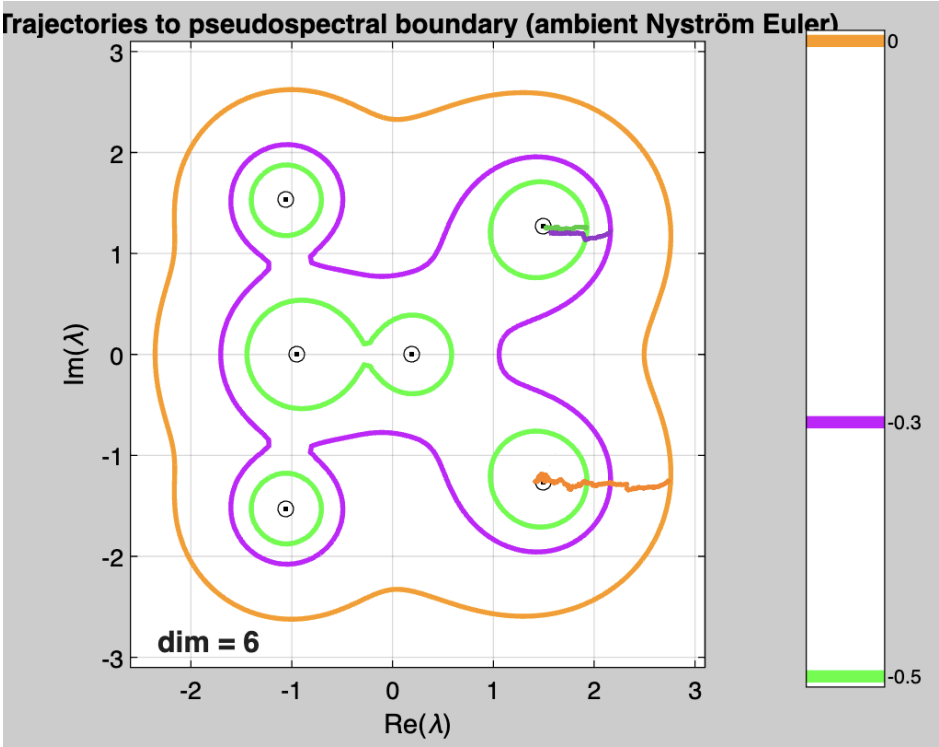

Figure 11: Randomized Euler with Nyström on Guglielmi-Lubich algorithm for plotting pseudospectra plott

9 Conclusion

Here is an overview of what we have done throughout the paper. First, we studied the stability of non-normal matrices under perturbations by focusing on the rightmost boundary of the ϵ-pseudospectrum, since these points determine the worst-case asymptotic growth rate of the system. To address this, we focused on the Guglielmi–Lubich algorithm, which evolves perturbations of the form $A + \epsilon uv^*$ using gradient flows together with projections onto manifolds. We

implemented this method in Python using Euler updates with repeated manifold projections. However, this method has some limitations: the main limitation lies in the repeated projections onto the tangent manifold. These projections are not guaranteed to preserve the ideal rank-one structure uv and may introduce deviations, which accumulate over time and reduce accuracy. This observation motivated the introduction of randomized methods, replacing the projection on manufold and Euler step with a randomized Euler step that evolves directly in the ambient space.

To this end, we first presented different randomization techniques. In particular, we discussed randomized singular value decomposition (randomized SVD), which uses random sketches to approximate dominant subspaces at much lower cost, and the generalized Nyström method, which extends this idea by sampling both rows and columns of a matrix to construct a low-rank approximation.

We used Nyström method to arrive to the randomized Euler method. Instead of forming the full gradient and projecting it, we compress the gradient update to rank one using the generalized Nyström procedure. This guarantees that the update remains rank one by construction, requiring only a simple normalization step, and thus avoids the inaccuracies introduced by repeated tangent projections. The resulting implementation is as a randomized, low-rank variant of the Guglielmi–Lubich algorithm, which returns a similar plot, but that is more stable in preserving rank-one structure especially in the case of large scaled matrices.

References

[1] L. N. Trefethen and M. Embree, *Spectrum and Pseudospectrum: The Behavior of Nonnormal Matrices and Operators*, Princeton University Press, 2005.

[2] N. Guglielmi and C. Lubich, *Differential Equations for Roaming Pseudospectra: Paths to Extremal Points and Boundary Tracking*, SIAM Review, 2011.

[3] Y. Nakatsukasa, *Fast and stable randomized low-rank matrix approximation*, arXiv, 2020.

[4] N. Guglielmi and C. Lubich, *ERRATUM: Differential Equations for Roaming Pseudospectra: Paths to Extremal Points and Boundary Tracking*, SIAM Review, pp. 977–981, 2012.

[5] N. Halko, P. G. Martinsson, J. A. Tropp, *Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions*, SIAM Review, vol. 53, no. 2, pp. 217–288, 2011.

[6] J. A. Tropp, A. Yurtsever, M. Udell, and V. Cevher, *Practical Sketching Algorithms for Low-Rank Matrix Approximation*, SIAM J. Matrix Anal. Appl., vol. 38, no. 4, pp. 1488–1522, 2017.

[7] N. Guglielmi and C. Lubich, *Low-Rank Dynamics for Computing Extremal Points of Real Pseudospectra*, SIAM J. Matrix Anal. Appl., vol. 36, no. 4, pp. 1447–1469, 2013.

[8] H. Y. Lam, G. Ceruti, and D. Kressner, *Randomized Low-Rank Runge–Kutta Methods*, SIAM J. Matrix Anal. Appl., vol. 46, no. 2, pp. 1587–1615, 2025.