Note on Matrix functions

Nathan Rousselot

September 1, 2023

Contents

1 Introduction

In this document we introduce the notion of matrix functions. Say we have a function $f : \mathbb{C} \to \mathbb{C}$, then we can define the function f on a matrix $\mathbf{A}$ as follows: $f : \mathbb{C}^{n\times n} \to \mathbb{C}^{n\times n}$. In the following, we will propose a formal definition for matrix functions. Then, we will present an elegant approach in evaluating any sufficiently differentiable function f on a matrix $\mathbf{A}$. Then, we will put in persepective the computational issues that arise when computing matrix functions, and we will note that the matrix-vector product $f(\mathbf{A})\mathbf{b}$ can be approached in a more efficient way when working on a lower dimension Krylov subspace. Finally, some numerical experiments will be presented to illustrate the results.

2 Theoretical background

In this section, we will see how we can define a matrix function.

2.1 Natural Definition

Polynomial functions

Let $p : \mathbb{C} \to \mathbb{C}$ be a polynomial function of degree d:

$$p(t) = \sum_{k=0}^{d} c_k t^k \tag{2.1}$$

Then, considering a matrix $\mathbf{A} \in \mathbb{C}^{n\times n}$, and posing $\mathbf{A}^0 = I_n$, we can define the polynomial function $p : \mathbb{C}^{n\times n} \to \mathbb{C}^{n\times n}$ on a matrix $\mathbf{A}$ as follows:

$$p(\mathbf{A}) = \sum_{k=0}^{d} c_k \mathbf{A}^k \tag{2.2}$$

Rational functions

Let $f : \mathbb{C} \to \mathbb{C}$ be a rational function of the form:

$$f(t) := \frac{p(t)}{q(t)} \tag{2.3}$$

It is not immediate how one would approach this function with a matrix. We want to define

$$f(\mathbf{A}) := q(\mathbf{A})^{-1} p(\mathbf{A}) \tag{2.4}$$

However, this is not well defined if $q(\mathbf{A})$ is singular. In other words, we need to make sure that $q(\mathbf{A})$ is invertible. This is the case if and only if $q(\lambda) \neq 0$ for all eigenvalues λ of $\mathbf{A}$. This is a very strong condition, and it is not always possible to find a rational function f such that $q(\lambda) \neq 0$ for all eigenvalues λ of $\mathbf{A}$. We note that a choice in notation has been made in equation 2.4, the other notation is still valid.

Lemma 1. *Let $\mathbf{A} \in \mathbb{C}^{n\times n}$, and let $p : \mathbb{C}^{n\times n} \to \mathbb{C}^{n\times n}$ and $q : \mathbb{C}^{n\times n} \to \mathbb{C}^{n\times n}$, two matrix polynomials. Then,*

$$q(\mathbf{A})p(\mathbf{A}) = p(\mathbf{A})q(\mathbf{A}) \tag{2.5}$$

Proof. Let $f(\mathbf{A}) := q(\mathbf{A})p(\mathbf{A})$, and $g(\mathbf{A}) = p(\mathbf{A})q(\mathbf{A})$. Then

$$
\begin{aligned}
f(\mathbf{A}) = \left(\sum_{k=0}^{d} c_k \mathbf{A}^k\right)\left(\sum_{j=0}^{m} b_j \mathbf{A}^j\right) \\
= \sum_{k=0}^{d}\sum_{j=0}^{m} c_k b_j \mathbf{A}^{j+k} \\
= \sum_{j=0}^{m}\sum_{k=0}^{d} b_j c_k \mathbf{A}^{k+j} \\
= \left(\sum_{j=0}^{m} b_j \mathbf{A}^j\right)\left(\sum_{k=0}^{d} c_k \mathbf{A}^k\right) = g(\mathbf{A})
\end{aligned}
$$

□

Theorem 1. *Let $\mathbf{A} \in \mathbb{C}^{n\times n}$, and let $p : \mathbb{C}^{n\times n} \to \mathbb{C}^{n\times n}$ and $q : \mathbb{C}^{n\times n} \to \mathbb{C}^{n\times n}$, two matrix polynomials such that $q(\mathbf{A})$ is non-singular. Then,*

$$q(\mathbf{A})^{-1}p(\mathbf{A}) = p(\mathbf{A})q(\mathbf{A})^{-1} \tag{2.6}$$

Proof. Let $\mathbf{A} \in \mathbb{C}^{n\times n}$ be a matrix such that $q(\mathbf{A})$ is non-singular

$$q(\mathbf{A})^{-1}p(\mathbf{A}) = q(\mathbf{A})^{-1}p(\mathbf{A})q(\mathbf{A})q(\mathbf{A})^{-1}$$

Using Lemma 1

$$
\begin{aligned}
q(\mathbf{A})^{-1}p(\mathbf{A})q(\mathbf{A})q(\mathbf{A})^{-1} &= q(\mathbf{A})^{-1}q(\mathbf{A})p(\mathbf{A})q(\mathbf{A})^{-1} \\
&= p(\mathbf{A})q(\mathbf{A})^{-1}
\end{aligned}
$$

□

Power Series

Let $f : \mathbb{C} \to \mathbb{C}$ be a function that can be expressed as a power series:

$$f(t) = \sum_{k=0}^{\infty} c_k t^k \tag{2.7}$$

Then, considering a matrix $\mathbf{A} \in \mathbb{C}^{n\times n}$, and posing $\mathbf{A}^0 = I_n$, we can define the power series function $f : \mathbb{C}^{n\times n} \to \mathbb{C}^{n\times n}$ on a matrix $\mathbf{A}$ as follows:

$$f(\mathbf{A}) = \sum_{k=0}^{\infty} c_k \mathbf{A}^k \tag{2.8}$$

In the scalar case, we know that the power series converges if $|t| < r$, where r is the radius of convergence. Obviously this translates in the matrix power series

Theorem 2. *Let $f : \mathbb{C}^{n\times n} \to \mathbb{C}^{n\times n}$ be a matrix power series. Then, the series converges if and only if $\rho(\mathbf{A}) < r$, where $\rho(\mathbf{A})$ is the spectral radius of $\mathbf{A}$, and r is the radius of convergence of the scalar power series.*

Proof is provided in Frommer and Simoncini 2008. In the case of a finite-order Laurent Series, *i.e*:

$$f(t) = \sum_{k=-d}^{d} c_k t^k \tag{2.9}$$

For the matrix case, we need to ensure convergence (similarly to power series), but also ensure existance of the inverse, as Laurent series do have negative powers. If both of those conditions are satisfied, we can write the Laurent series as a matrix function:

$$f(\mathbf{A}) = \sum_{k=-d}^{d} c_k \mathbf{A}^k \tag{2.10}$$

2.2 Spectrum-Based Definition

Diagonalizable Matrices

Let $\mathbf{A} \in \mathbb{C}^{n\times n}$ be a diagonalizable matrix. That means that there exists a matrix $\mathbf{V} \in \mathbb{C}^{n\times n}$ such that $\mathbf{V}$ is invertible, and $\mathbf{A} = \mathbf{V\Lambda V}^{-1}$, where $\mathbf{\Lambda}$ is a diagonal matrix containing the eigenvalues of $\mathbf{A}$:

$$\mathbf{\Lambda} = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix} \tag{2.11}$$

Theorem 3. *Let $\mathbf{A} \in \mathbb{C}^{n\times n}$ be a diagonalizable matrix. Then we can define the function $f(\mathbf{A})$ as:*

$$f(\mathbf{A}) := \mathbf{V} f(\mathbf{\Lambda})\mathbf{V}^{-1} \tag{2.12}$$

with

$$f(\mathbf{\Lambda}) = \begin{bmatrix} f(\lambda_1) & 0 & \cdots & 0 \\ 0 & f(\lambda_2) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & f(\lambda_n) \end{bmatrix} \tag{2.13}$$

This property is very handy, as it allows us to compute matrix functions by simply applying the function to the eigenvalues of the matrix. Computationally, this avoids inverting matrices, and lots of matrix products. However, this property is only valid for diagonalizable matrices. This property puts constraints on $f(\mathbf{A})$, as its eigenvectors must form a basis $\mathbf{F}^n$. Another more practical constraint, that is sufficient but not necessary, is if $\mathbf{A}$ is a full rank matrix, then it is diagonalizable.

Defective Matrices

In some cases, the matrix $\mathbf{A}$ is not diagonalizable, that means the sum of the dimensions of the eigenspaces is less than n, we call that a *Defective Matrix.* In that case, we can generalize the principle of diagonalization using the Jordan canonical form of $\mathbf{A}$:

$$\mathbf{A} = \mathbf{VJV}^{-1} \tag{2.14}$$

where $\mathbf{J}$ is a Jordan matrix, and $\mathbf{V}$ is a matrix containing the generalized eigenvectors of $\mathbf{A}$. The Jordan matrix is a block diagonal matrix, where each block is a Jordan block. A Jordan block is a matrix of the form:

$$\mathbf{J}_k(\lambda) = \begin{bmatrix} \lambda & 1 & 0 & \cdots & 0 \\ 0 & \lambda & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda & 1 \\ 0 & 0 & \cdots & 0 & \lambda \end{bmatrix} \tag{2.15}$$

Definition 1. *Let $\mathbf{A} \in \mathbb{C}^{n \times n}$ be a defective matrix. Then we can define the function $f(\mathbf{A})$ as:*

$$f(\mathbf{A}) := \mathbf{V} f(\mathbf{J}) \mathbf{V}^{-1} \tag{2.16}$$

with

$$f(\mathbf{J}) = \begin{bmatrix} f(\mathbf{J}_1(\lambda_1)) & 0 & \cdots & 0 \\ 0 & f(\mathbf{J}_2(\lambda_2)) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & f(\mathbf{J}_k(\lambda_k)) \end{bmatrix} \tag{2.17}$$

and

$$f(\mathbf{J}_i(\lambda_i)) = \begin{bmatrix} f(\lambda_i) & f'(\lambda_i) & \frac{f''(\lambda_i)}{2!} & \cdots & \frac{f^{(k-1)}(\lambda_i)}{(k-1)!} \\ 0 & f(\lambda_i) & f'(\lambda_i) & \cdots & \frac{f^{(k-2)}(\lambda_i)}{(k-2)!} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & f(\lambda_i) & f'(\lambda_i) \\ 0 & 0 & \cdots & 0 & f(\lambda_i) \end{bmatrix} \tag{2.18}$$

Obviously, both definitions of matrix functions, based on diagonalization and Jordan canonical form, presuppose that the spectral radius of $\mathbf{A}$, $\rho(\mathbf{A})$, is less than r, the radius of convergence.

2.3 Interpolation-based definition

Interestingly, in this section we will show that for any $\mathbf{A} \in \mathbb{C}^{n \times n}$ and any sufficiently differentiable f, we can find a polynomial p such that $f(\mathbf{A}) = p(\mathbf{A})$. First, let us observe from previous sections that only the eigenvalues of $\mathbf{A}$ are actually important for matrix polynomials. Also recall that every matrix $\mathbf{A} \in \mathbb{C}^{n \times n}$ with spectrum $\{\lambda_1, \ldots, \lambda_n\}$ has a minimal polynomial $\phi_{\mathbf{A}}$ given by

$$\phi_{\mathbf{A}}(t) := \prod_{i=1}^{k} (t - \lambda_k)^{n_i} \tag{2.19}$$

which is the unique monic minimal degree ($\text{degr}(\phi_{\mathbf{A}}) = n_1 + \cdots + n_k \leq n$) polynomial such that $\phi_{\mathbf{A}}(\mathbf{A}) = 0$.

Theorem 4. *Let $\mathbf{A} \in \mathbb{C}^{n \times n}$ be a matrix with eigenvalues $\{\lambda_1, \ldots, \lambda_k\}$, and minimal polynomial given by equation 2.19. Then for any two polynomials p_1, p_2 we have that $p_1(\mathbf{A}) = p_2(\mathbf{A})$ if and only if*

$$\forall j \in \{1, \ldots, k\} : \forall i \in \{0, \ldots, n_k - 1\} : p_1^{(i)}(\lambda_j) = p_2^{(i)}(\lambda_j).$$

Proof. Let $\mathbf{A} \in \mathbb{C}^{n\times n}$, with spectrum $\{\lambda_1, \dots, \lambda_k\}$, and two polynomials p_1 and p_2 such that $p_1(\mathbf{A}) = p_2(\mathbf{A})$. Let us define q such that

$$q := p_1 - p_2$$

Then, $q(\mathbf{A}) = 0$, and is thus divisible by $\phi_{\mathbf{A}}$, meaning that

$$\forall j \in \{1, \dots, k\} : \forall i \in \{0, \dots, n_k - 1\}, q(\lambda_j) = 0 \Rightarrow p_1^{(i)}(\lambda_j) = p_1^{(i)}(\lambda_i)$$

Similarly, consider two polynomials p_1 and p_2 such that

$$\forall j \in \{1, \dots, k\} : \forall i \in \{0, \dots, n_k - 1\}, p_1^{(i)}(\lambda_j) = p_2^{(i)}(\lambda_i)$$

For $i = 0$, $q := p_1 - p_2 = 0$ on the spectrum of $\mathbf{A}$, and is then divisible by $\phi_{\mathbf{A}}$. Then

$$q = K\phi_{\mathbf{A}}$$

with K a polynomial. Then, $q(\mathbf{A}) = K(\mathbf{A})\phi_{\mathbf{A}}(\mathbf{A}) = 0$ since by definition, $\phi_{\mathbf{A}}(\mathbf{A}) = 0$. And thus, $p_1(\mathbf{A}) = p_2(\mathbf{A})$.
From this reasoning, we conclude that

$$\begin{aligned} & p_1(\mathbf{A}) = p_2(\mathbf{A}) \\ \Leftrightarrow \forall j \in \{1, \dots, k\} : \forall i \in \{0, \dots, n_k - 1\} : \ & p_1^{(i)}(\lambda_j) = p_2^{(i)}(\lambda_j) \end{aligned}$$

□

In theorem 4, the conditions involve the evaluation of the polynomials p_1 and p_2 and their derivatives up to order $n_k - 1$ at the eigenvalues of A.
However, when the spectrum of A is simple, each eigenvalue λ_j is of multiplicity $n_j = 1$. This means that there are no higher order terms corresponding to these eigenvalues in the minimal polynomial, or in other words, there are no repeated roots. Consequently, there is no need to consider the derivatives of the polynomials p_1 and p_2 because there are no repeated roots for the polynomials to "match up" with. Therefore, in this simpler case, we only need to check that the polynomials p_1 and p_2 agree at the eigenvalues of A. In formal terms, the condition becomes:

Corollary 4.1. *Let $\mathbf{A} \in \mathbb{C}^{n\times n}$ be a matrix with eigenvalues $\{\lambda_1, \dots, \lambda_k\}$, and minimal polynomial given by equation 2.19. If $\mathbf{A}$ has simple spectrum, then for any two polynomials p_1, p_2 we have that $p_1(\mathbf{A}) = p_2(\mathbf{A})$ if and only if*

$$\forall j \in \{1, \dots, k\} : p_1(\lambda_j) = p_2(\lambda_j).$$

From this corollary, and from theorem 4, we can confirm our earlier statement : only the spectrum of $\mathbf{A}$ is important for matrix polynomials. More importantly, we observe that $p(\mathbf{A})$ is uniquely defined by its values on the spectrum of $\mathbf{A}$. It seems then natural to extend this definition to any function f.

Definition 2. *Let $\mathbf{A} \in \mathbb{C}^{n\times n}$ be a matrix with minimal polynomial given as in equation 2.19, and let f be a function that is at least $\max_k\{n_k - 1\}$ times differentiable. Say p is its $(n_1, \dots, n_k)$*-Hermite interpolant *i.e. the polynomial satisfying*

$$\forall j \in \{1, \dots, k\} : \forall i \in \{0, \dots, n_k - 1\} : p^{(i)}(\lambda_j) = f^{(i)}(\lambda_j)$$

of minimal degree. Then we define $f(\mathbf{A}) = p(\mathbf{A})$.

3 The matrix-vector product $f(\mathbf{A})\mathbf{b}$

3.1 Introduction

To motivate the need for a matrix-vector product, we will consider a matrix $\mathbf{A} \in \mathbb{C}^{n\times n}$. Let us consider the matrix exponential, such as :

$$e^{\mathbf{A}} = \sum_{k=0}^{\infty} \frac{\mathbf{A}^k}{k!} \tag{3.1}$$

Not only does this computation is very heavy (see section 5.1 for computation strategies), but it also affects the structure of the matrix. Say for example, we have the following laplacian matrix:

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & 0 & \dots & 0 \\ 0 & -1 & 2 & -1 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & -1 & 2 & -1 \\ 0 & 0 & \dots & 0 & 0 & -1 & 2 \end{bmatrix}$$

Obviously, $\mathbf{A}$ is a sparse matrix, and also a rank-structured one : it is a tridiagonal matrix. However, by computing $e^{\mathbf{A}}$, one will get a dense matrix. Storing a dense matrix often becomes challenging as the dimension of the problem grows. Besides storing, computing such a dense matrix is also an issue. However, the matrix-vector product $f(\mathbf{A})\mathbf{b}$ can be stored and computed efficiently.

3.2 The Method

As motivated by 3.1, when encountering specific structure in $\mathbf{A}$, such as sparsity, there is a lot to gain if we can store $f(\mathbf{A})\mathbf{b}$. In this section we will work towards a way to evaluate $f(\mathbf{A})\mathbf{b}$ in an intuitive way, thanks to Arnoldi method.

3.2.1 Formal Definitions

Firstly, we will define the notion of $\phi_{\mathbf{A},\mathbf{b}}$, i.e. the minimal polynomial of $\mathbf{A}$ with respect to the vector $\mathbf{b}$. This is simply the polynomial

$$\phi_{\mathbf{A},\mathbf{b}}(t) := \prod_{i=1}^{k} (t-\lambda_i)^{m_i} \tag{3.2}$$

of minimal degree such that $\phi_{\mathbf{A},\mathbf{b}}(\mathbf{A})\mathbf{b} = \mathbf{0}$. Here $\lambda_1, \dots, \lambda_k$ are again the eigenvalues of $\mathbf{A}$.

Lemma 2. *Let $\mathbf{A} \in \mathbb{C}^{n\times n}$ be a matrix with eigenvalues $\{\lambda_1, \dots, \lambda_k\}$, and minimal polynomial given by equation 2.19. Then for any two polynomials p_1, p_2 we have that $p_1(\mathbf{A})\mathbf{b} = p_2(\mathbf{A})\mathbf{b}$ if and only if*

$$\forall j \in \{1, \dots, k\} : \forall i \in \{0, \dots, n_k - 1\} : p_1^{(i)}(\lambda_j) = p_2^{(i)}(\lambda_j).$$

Proof. From theorem 4, we know that

$$\begin{aligned} & p_1(\mathbf{A}) = p_2(\mathbf{A}) \\ \Leftrightarrow\ & \forall j \in \{1, \dots, k\} : \forall i \in \{0, \dots, n_k - 1\} : p_1^{(i)}(\lambda_j) = p_2^{(i)}(\lambda_j) \end{aligned}$$

We also know that

$$p_1(\mathbf{A}) = p_2(\mathbf{A}) \Leftrightarrow p_1(\mathbf{A})\mathbf{b} = p_2(\mathbf{A})\mathbf{b}$$

Thus, we conclude that

$$\begin{aligned} & p_1(\mathbf{A})\mathbf{b} = p_2(\mathbf{A})\mathbf{b} \\ \Leftrightarrow \forall j \in \{1, \ldots, k\} : \forall i \in \{0, \ldots, n_k - 1\} : & p_1^{(i)}(\lambda_j) = p_2^{(i)}(\lambda_j) \end{aligned}$$

□

Theorem 5. *Let f be a sufficiently differentiable function that has no singularities on the spectrum of a given matrix $\mathbf{A} \in \mathbb{C}^{n\times n}$. Then, with p the unique Hermite interpolating polynomial of $\mathbf{A}$ w.r.t. $\mathbf{b}$ i.e.*

$$\forall j \in \{1, \ldots, k\} : \forall i \in \{0, \ldots, m_k - 1\} : p^{(i)}(\lambda_j) = f^{(i)}(\lambda_j)$$

we have that $f(\mathbf{A})\mathbf{b} = p(\mathbf{A})\mathbf{b}$.

Proof. Let f be a function that is at least $\max_k\{m_k - 1\}$ times differentiable, and let p_1 be its $(m_1, \ldots, m_k)$-Hermite interpolant. Then, by definition 2, we have that

$$f(\mathbf{A}) = p_1(\mathbf{A})$$

Then, by lemma 2, let us define p_2 such that

$$p_2(\mathbf{A})\mathbf{b} = p_1(\mathbf{A})\mathbf{b}$$

Then, we have that

$$p_1(\mathbf{A})\mathbf{b} = p_2(\mathbf{A})\mathbf{b} = f(\mathbf{A})\mathbf{b}$$

□

3.2.2 The Arnoldi Method

The Arnoldi method is a reduction method that allows low-rank approximation of a given matrix $\mathbf{A}$. It is based on the Hessenberg reduction of a matrix $\mathbf{A}$. More formally

Definition 3. *Let $\mathbf{A} \in \mathbb{C}^{n\times n}$, the Arnoldi method approximates its Hessenberg reduction given by*

$$\mathbf{V}^*\mathbf{A}\mathbf{V} = \mathbf{H} \tag{3.3}$$

where $\mathbf{V} \in \mathbb{C}^{n\times n}$ is unitary, and $\mathbf{H} \in \mathbb{C}^{n\times n}$ is upper Hessenberg. As it is a (low-rank) approximation, Arnoldi will compute the following decomposition

$$\mathbf{V}_k^*\mathbf{A}\mathbf{V}_k = \mathbf{H}_k \tag{3.4}$$

with $\mathbf{V}_k \in \mathbb{C}^{n\times k}$ unitary, and $\mathbf{H}_k \in \mathbb{C}^{k\times k}$ upper Hessenberg. That means that $\mathbf{H}_k$ is the orthogonal projection of $\mathbf{A}$ onto the kth Krylov subspace $\mathcal{K}_k(\mathbf{A}, \mathbf{v_1})$, with $\mathbf{v_1}$ the first column of $\mathbf{V}$.

The Arnoldi method is an iterative method, following this algorithm:
Here the orthogonalization process (lines 3-6) is a modified Gram-Schmidt process. Also, when implementing, we will consider an ℓ_2 norm. Numerical conditions will often imply loss of orthogonality. To avoid this, it will be necessary to reorthogonalize, *i.e.* to reapply the Gram-Schmidt process.

Algorithm 1: Arnoldi Iteration

Data: $A \in \mathbb{C}^{n \times n}$, $v_1 \in \mathbb{C}^n$ a unit vector in the chosen norm (line 7)
Result: $H_n \in \mathbb{C}^{n \times n}$

1 **for** $k = 1$**to**n **do**
2     $\mathbf{w} \leftarrow \mathbf{A}\mathbf{v}_k$;
3   **for** $i = 1$**to**k **do**
4         $h_{ik} \leftarrow \mathbf{v}_i^* \mathbf{w}$;
5         $\mathbf{w} \leftarrow \mathbf{w} - h_{ik}\mathbf{v}_i$;
6   **end**
7     $h_{k+1,k} \leftarrow \|\mathbf{w}\|$;
8   **if** $h_{k+1,k} = 0$ **then**
9   **break**;
10   **end**
11    $\mathbf{v_{k+1}} \leftarrow \mathbf{w}/h_{k+1,k}$
12 **end**

3.2.3 The matrix-vector product $f(\mathbf{A})\mathbf{b}$

Recall the problem is to approximate the matrix-vector product $f(\mathbf{A})\mathbf{b}$. Now we have all the tools to achieve that efficiently.

Lemma 3. *Let $\mathbf{A} \in \mathbb{C}^{n \times n}$ and $\mathbf{b} \in \mathbb{C}^n$. Then, the matrix vector product $f(\mathbf{A})\mathbf{b}$ can be approximated by*

$$f(\mathbf{A})\mathbf{b} \approx \|\mathbf{b}\|_2 \mathbf{V}_k f(\mathbf{H}_k)\mathbf{e}_1 \tag{3.5}$$

Where $\mathbf{V}_k$ and $\mathbf{H}_k$ are the matrices computed by the Arnoldi method after k iterations. $\mathbf{H}_k$ is upper Hessenberg and $\operatorname{span}(\mathbf{V}_k) = \mathcal{K}_k(\mathbf{A}, \mathbf{b}) = \operatorname{span}\big(\mathbf{b}, \mathbf{A}\mathbf{b}, \mathbf{A}^2\mathbf{b}, \ldots, \mathbf{A}^{k-1}\mathbf{b}\big)$*. $\mathbf{e}_1$ is the first column of the identity matrix.*

Proof. Consider we want to approximate de matrix-vactor product $f(\mathbf{A})\mathbf{b}$. We will use the Arnoldi method (algorithm 12). For initial unit vector we choose $\mathbf{v}_1 = \mathbf{b}/\|\mathbf{b}\|_2$ which is indeed an ℓ_2 unit vector. According to the Hessenberg reduction of $\mathbf{A}$, we have that

$$f(\mathbf{A})\mathbf{b} \approx f(\mathbf{V_k}\mathbf{H}_k\mathbf{V}_k^*)\mathbf{b}$$

Since $\mathbf{V}_k$ is unitary, we have

$$f(\mathbf{A})\mathbf{b} \approx \mathbf{V}_k f(\mathbf{H}_k)\mathbf{V}_k^*\mathbf{b}$$

Furthermore, as we chose $\mathbf{v}_1 = \mathbf{b}/\|\mathbf{b}\|_2$, we have that

$$\mathbf{b} = \|\mathbf{b}\|_2\mathbf{v}_1 = \|\mathbf{b}\|_2\mathbf{V}_k\mathbf{e}_1$$

Thus, we have that

$$f(\mathbf{A})\mathbf{b} \approx \|\mathbf{b}\|_2\mathbf{V}_k f(\mathbf{H}_k)\mathbf{V}_k^*\mathbf{V}_k\mathbf{e}_1 = \|\mathbf{b}\|_2\mathbf{V}_k f(\mathbf{H}_k)\mathbf{e}_1$$

$\square$

Lemma 4. *Let $\mathbf{A} \in \mathbb{C}^{n \times n}$ and let $\mathbf{V}_k$, $\mathbf{H}_k$ be the matrices computed by the Arnoldi method after k iterations. Then, for any polynomial p_j of degree $j \leq k-1$, we have that*

$$p_j(\mathbf{A})\mathbf{b} = \mathbf{V}_k p_j(\mathbf{H}_k)\mathbf{e}_1 \tag{3.6}$$

Lemma 5. *Let $\mathbf{A} \in \mathbb{C}^{n\times n}$ and let $\mathbf{V}_k$, $\mathbf{H}_k$ be the matrices computed by the Arnoldi method after k iterations. Then, given p_{k-1} the unique Hermite interpolant of $\mathbf{A}$ w.r.t. $\mathbf{b}$ of degree $k-1$, we have that*

$$\|\mathbf{b}\|_2 \mathbf{V}_k f(\mathbf{H}_k)\mathbf{e}_1 = p_{k-1}(\mathbf{A})\mathbf{b} \tag{3.7}$$

Proof. Since p_{k-1} is the unique Hermite interpolant of $\mathbf{A}$ w.r.t. $\mathbf{b}$ of degree $k-1$, we have that

$$\|\mathbf{b}\|_2 \mathbf{V}_k f(\mathbf{H}_k)\mathbf{e}_1 = \|\mathbf{b}\|_2 \mathbf{V}_k p_{k-1}(\mathbf{H}_k)\mathbf{e}_1$$

And since

$$\|\mathbf{b}\|_2 \mathbf{V}_k p_{k-1}(\mathbf{H}_k)\mathbf{e}_1 = \|\mathbf{b}\|_2 p_{k-1}(\mathbf{A})\mathbf{q}_1 = p_{k-1}(\mathbf{A})\mathbf{b}$$

We indeed have

$$\|\mathbf{b}\|_2 \mathbf{V}_k f(\mathbf{H}_k)\mathbf{e}_1 = p_{k-1}(\mathbf{A})\mathbf{b}$$

□

Theorem 6. *Let $\mathbf{A} \in \mathbb{C}^{n\times n}$ and let $m = \deg(\phi_{\mathbf{A},\mathbf{b}})$ then*

$$f(\mathbf{A})\mathbf{b} = \|\mathbf{b}\|_2 \mathbf{V}_m f(\mathbf{H}_m)\mathbf{e}_1 \tag{3.8}$$

Proof. Following theorem 5, we have that

$$f(\mathbf{A})\mathbf{b} = p_{m-1}(\mathbf{A})\mathbf{b}$$

With p_{m-1} the unique Hermite interpolant of $\mathbf{A}$ w.r.t. $\mathbf{b}$ of degree $m-1$ with $\deg(\phi_{\mathbf{A},\mathbf{b}}) = m$. Then, by lemma 3, 4 and 5, we have that

$$f(\mathbf{A})\mathbf{b} = \|\mathbf{b}\|_2 \mathbf{V}_m f(\mathbf{H}_m)\mathbf{e}_1$$

□

4 Algorithms

In this section, we will implement basic algorithms. From the previous section, we will distinguish two cases :

- the dense case, where $f(\mathbf{A})$ is to be computed
- and the case where only the matrix-vector product $f(\mathbf{A})\mathbf{b}$ is to be computed

4.1 Dense case

First, let us consider the computation of $f(\mathbf{A})$. Meaning we put aside the matrix-vector product (for now). More specifically, our approach will be to use definition 2 to compute $f(\mathbf{A})$. All implementations are done in `Matlab 2023a`.

4.1.1 Dependencies

We are provided with a function `hess_and_phi()` that computed the Hessenberg reduction of a given matrix and its associated minimal polynomial. It takes as input a matrix $\mathbf{A} \in \mathbb{C}^{n \times n}$. It then follows these simple steps:

1. Compute the Hessenberg reduction of $\mathbf{A}$, i.e. $\mathbf{V}^* \mathbf{A} \mathbf{V} = \mathbf{H}$
2. It computes its Jordan Canonical Form from $\mathbf{H}$
3. It computes the minimal polynomial of $\mathbf{A}$, $\phi_{\mathbf{A}}$ through its Jordan Canonical Form
4. It returns $\mathbf{V}$, $\mathbf{H}$, $\mathbf{J}$, λ_j (the eigenvalues of $\mathbf{A}$) and n_i (the multiplicity of λ_i in $\phi_{\mathbf{A}}$)

Note that step 2 makes perfect sense.

Theorem 7. *Let $\mathbf{A} \in \mathbb{C}^{n \times n}$ with a Hessenberg reduction defined by $\mathbf{V}^* \mathbf{A} \mathbf{V} = \mathbf{H}$. Then $\mathbf{A}$ and $\mathbf{H}$ share the same Jordan Canonical Form.*

Proof. $\mathbf{V}$ is unitary, hence $\mathbf{A}$ and $\mathbf{H}$ are similar matrices, meaning they have the same eigenvalues. Thus, they share the same Jordan Canonical Form. $\square$

To recover the minimal polynomial of $\mathbf{A}$, one simply retrives the λ_j and the n_i from the previously described steps, and then compute the polynomial this way:

$$\phi_{\mathbf{A}}(t) = \prod_{i=1}^{k} (t - \lambda_k)^{n_i}$$

In the supplementary materials, you will find an implementation of this, called `construct_minimal_polynomial(`
One quick way to assess if the minimal polynomial is constructed correctly, is to evaluate it at $\mathbf{A}$. Taking the ℓ_2 norm of the result should be close to zero. Here, with `test1.mat` (a 5 by 5 matrix), we obtain $\phi_{\mathbf{A}}(\mathbf{A}) = 3.3523e - 12$. This is indeed numerically close to zero.
For the Hermite interpolation, several options are possible. First, a routine `hermite_interp()` is provided in the supplementary materials, it computed the Hermite interpolation by divided differences. However, I had more precise results with an alternative method proposed by Or Werner, BGU, Israel, based on the Hermite method. In the following, I used the latter approach, the routine is provided in the supplementary material aswell.

4.1.2 Implementation

From the previous elements, we can construct a routine called `matrix_function()` that takes as input a matrix $\mathbf{A} \in \mathbb{C}^{n \times n}$, and a function f. It has extremely simple steps:

1. Retrieve λ_j and n_i from `hess_and_phi()`
2. Construct the matrix `FdF` which is f and its derivatives evaluated at the eigenvalues of $\mathbf{A}$
3. Construct the Hermite interpolation of $\mathbf{A}$ with `hermite_interp()`
4. Evaluate the Hermite interpolation at $\mathbf{A}$, and return the result

The routine is provided in the supplementary materials.

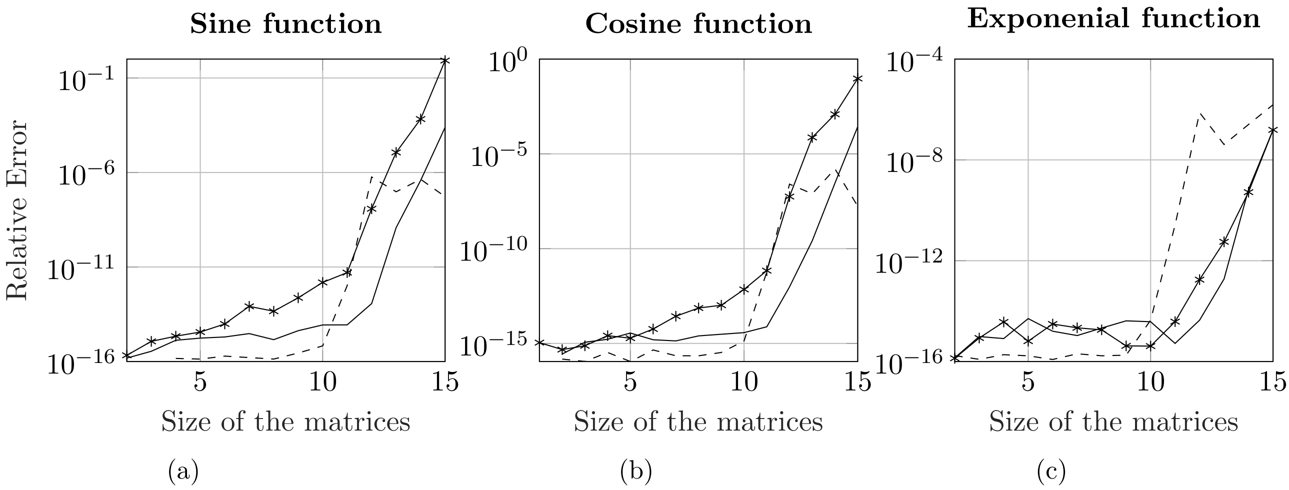


Figure 1: Measure of performance of the algorithm to compute $f(\mathbf{A})$ according to definition 2. We assume that Davies and Higham 2003 (Schur Parlett) algorithm is correct, and we measure the relative error. We compare the performance of our algorithm on several types of matrices : diagonal (- -), symmetric (-*) and dense (–). Matrix coefficients are randomly filled such that for all non-zero entry: $a_{ij} \sim \mathcal{U}(0, 1)$, this ensures $\mathbf{A}$ is full rank. We test it on three different functions : Sine (a), Cosine (b) and Exponenial (c). We note that starting from a certain matrix size, the algorithm lose considerably in performance.

4.1.3 Results

In this part, we will compare the performance of our routine `matrix_function()` with the built-in `funm()` function from `Matlab`. The built-in Matlab function uses a Schur-Parlett algorithm from Davies and Higham 2003. To compare both methods, we generate three types of matrices of different sizes. More precisely, we will investigate both algorithm's performances on symmetric, diagonal and dense matrices, on the following functions: cosine, sine and exponential.
Figure 1 illustrates the result of the measurement of performance of the algorithm. The relative error between Davies and Higham 2003 and our results is computed:

$$\text{relative error} = \frac{\|f(\mathbf{A}) - \texttt{funm}(\mathbf{A}, f)\|_2}{\|\texttt{funm}(\mathbf{A}, f\|_2}$$

We note that for small matrices ($n \leq 10$), the hermite interpolation approach is precise and has residual error inferior to 10^{-11}, which is somewhat close to numerical precision (around 10^{-16} as we work with double precision). However, starting from a certain matrix size, the algorithm loses considerably in performance: both in precision and in computation time. Interestingly, this behavior is independant to the matrix structure : diagonal, symmetric or dense. It also seem to be independant to the function we are evaluating (though we has slightly lower relative error for the exponential function as depicted in figure 1c).
We conclude that this algorithm, while being elegant, is quickly inefficient, and lead to both numerical and computational issues, even at small matrix sizes.

4.2 Matrix-vector product

4.2.1 Implementation

In this section, we will implement the matrix-vector product $f(\mathbf{A})\mathbf{b}$, as described in section 3.1. We will use the Arnoldi method to achieve this. The implementation is done in `Matlab 2023a`. As the theory has been well described in section 3.1, the implementation is straight-forward :

Algorithm 2: Matrix-Vector Product

Data: $\mathbf{A} \in \mathbb{C}^{n\times n}$, $\mathbf{b} \in \mathbb{C}^{n}$, f a function and $k \in \mathbb{N}$ the dimension of the Krylov subspace
Result: $f(\mathbf{A})\mathbf{b}$

$\mathbf{v}_1 \leftarrow \mathbf{b}/\|\mathbf{b}\|_2$;
$[\mathbf{V}, \mathbf{H}] \leftarrow \texttt{arnoldi}(\mathbf{A}, \mathbf{v}_1, k)$;
$e_1 \leftarrow$ first column of the identity matrix;
$f \leftarrow \|b\|_2 \mathbf{V} f(\mathbf{H})\mathbf{e}_1$;

Where `arnoldi()` is the Arnoldi iteration described in algorithm 12. This routine is provided in the supplementary materials.

4.2.2 Results

To evaluate the performance of this Matrix-vector product routine, we will test it versus the naive way of doing it in `Matlab`, meaning computing $f(\mathbf{A})$ explicitely, then multiplying it by $\mathbf{b}$. In our first experiment (figure 2), where $\mathbf{A}$ is a large sparse matrix. More specifically it is a graded L-shape pattern from George and Liu 1978. The vector $\mathbf{b}$ is filled with uniformly distributed random coefficients such that $b_i \sim \mathcal{U}(0, 1)$. The figure 2b shows that for such a setup, the proposed algorithm reaches a relative error equals to numerical precision few iterations afters 20. This means

$$\frac{\|\|\mathbf{b}\|_2 \mathbf{V}_k f(\mathbf{H}_k)\mathbf{e}_1 - f(\mathbf{A})\mathbf{b}\|_2}{\|f(\mathbf{A})\mathbf{b}\|_2} = \epsilon$$

for $k > 20$, and where ϵ is the machine precision. Note that this result is very specific to the problem depicted in figure 2, and the threshold for k will differ in function of the considered problem. Another interpretation is that computing $f(\mathbf{A})\mathbf{b}$ on the smaller Krylov subspace $\mathcal{K}_k(\mathbf{A}, \mathbf{b})$ gives extremely good results, even when $k << n$.
Obviously, this very interesting result lets us think that there is a lot of potential computational gain in evaluating $f(\mathbf{A})\mathbf{b}$ on this smaller Krylov subspace. The computational gain for the problem solved in figure 2 is described in Table 1. We note that for this problem, our method considerably outperforms the naive approach of evaluating $f(\mathbf{A})$ seperately, regardless of the function (though the biggest gain seem to come from the matrix exponential).
However, this does not give us any insight about how those two algorithms sacle up or down. To investigate this, we will use the BCSPWR matrix collection. This collection contains matrices of different sizes, and different sparsity patterns. We will test our method against the naive approach on this collection. The results are depicted in Table 2. We notice that for very low dimension ($n < 300$) the naive approach slightly outperforms ours for trigonometric functions. However, for larger matrices, and for the exponential function, working in the Kryloc subspace is the way to go. For the largest matrix, where $n = 5300$, the naive approach takes half a minute, while our method takes less than a tenth of a second. This is a considerable gain in performance, and could be crucial in some applications. As this margin increases with the matrix size, we can expect our method to

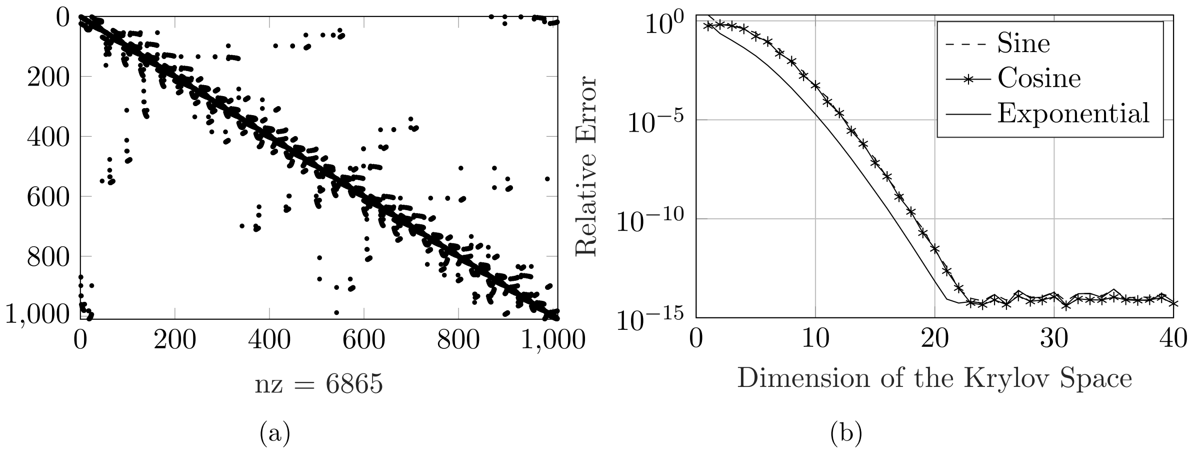


Figure 2: Evaluation of the performance of the matrix-vector product routine. We compare the performance of our method against the naive way of doing it in `Matlab`. We test it on three different functions : Sine (- -), Cosine (-*) and Exponenial (–) on figure (b). Those functions are applied to a large matrix $\mathbf{A}$ with sparisity pattern given in (a). Our method allows for precise approximation of $f(\mathbf{A})b$ even when working on a very low rank Hessenberg reduction (b). The vector $\mathbf{b}$ is a random vector whose coefficients are uniformly distributed such that $b_i \sim \mathcal{U}(0,1)$.

Table 1: Comparison of computational performance of both methods for evaluating $f(\mathbf{Ab})$ with $\mathbf{A}$ the matrix in 2a. Performance was measured in ms. Optimal k was determined when the relative error went below $1e-14$, as we worked on double precision. Tests were run on an Intel Core i7-1185G7 @ 3.00GHz with 16GB RAM. We note that for this problem, our method is outperforming by a significant margin the naive approach.

fun	**Naive way (ms)**	**Krylov method (ms)**	**Optimal k**
`exp()`	151.2	6.3	21
`cos()`	121.8	11.2	23
`sin()`	112.6	8.1	23

be even more efficient for larger matrices. One final interesting note, is that the optimal Krylov space dimension does not change much with the matrix size. This is a very interesting result, as it means that we can expect our method to be efficient for a wide range of matrix sizes, and explains why it does not suffer from scaling up ($f(\mathbf{H_k})$, which is the costly operation, is somewhat the same size whatever the tested matrix).

5 Applications

In this section, we will try to apply the previously mentionned algorithm to more practical problems. We will see how to take advantage of problem's nature thanks to those tools in order to solve them with more efficiency.

Table 2: Comparison of the naive approach and the Krylov approach on BCSPWR matrix collection. Performance was measured in ms. Optimal k was determined when the relative error went below $1e-14$.

Matrix	n	`exp()`		`cos()`		`sin()`	
		Naive	Krylov	Naive	Krylov	Naive	Krylov
BCSPWR01	39	**1.0**	1.3	**0.55**	1.5	**0.38**	5.3
BCSPWR02	49	3.2	**1.9**	**0.66**	3.7	**0.48**	1.6
BCSPWR03	118	16.6	**2.3**	**1.3**	3.0	**0.87**	3.1
BCSPWR04	274	6.5	**1.6**	**5.2**	8.7	**5.9**	6.4
BCSPWR05	443	26.4	**2.7**	21.8	**4.6**	21.5	**3.6**
BCSPWR06	1454	436.6	**11.9**	299.6	**10.3**	399.6	**12.1**
BCSPWR07	1612	484.0	**11.9**	392.0	**14.7**	393.2	**14.7**
BCSPWR08	1624	504.4	**13.4**	414.5	**12.4**	433.8	**14.6**
BCSPWR09	1723	628.1	**11.7**	593.7	**10.3**	531.8	**10.1**
BCSPWR10	5300	32868	**98.4**	27447	**96.7**	23948	**89.7**

5.1 Matrix Exponential

5.1.1 Context

Consider the simple system of ODEs

$$\frac{d\mathbf{x}}{dt} = \mathbf{A}\mathbf{x} \tag{5.1}$$

with initial condition $\mathbf{x}(0) = \mathbf{x}_0 \in \mathbb{R}^n$. Then we know the solution to be given by $\mathbf{x}(t) = e^{\mathbf{A}t}\mathbf{x}_0$. However, for all but the stablest systems, this is not a good method, due to issues such as stability and stiffness. Here we consider for instance the 2D convection-diffusion equation for the flow $\mathbf{u}(x, y)$:

$$\frac{d\mathbf{u}}{dt} = \epsilon\Delta\mathbf{u} + \alpha \cdot \nabla\mathbf{u} \tag{5.2}$$

with Dirichlet boundary conditions and $\epsilon \in \mathbb{R}_0^+$ and $\alpha \in \mathbb{R}^2$. Simple time-stepping methods are known to be unstable at large time-steps, and our exponential scheme suffers from similar problems, i.e. t cannot be taken too large. However, it remains an interesting subject to evaluate the impact of using appropriate routines to compute $\mathbf{x}(t)$.
The convection-diffusion equation (equation 5.2) is split in two parts. First, a diffusion term $\epsilon\Delta\mathbf{u}$, and a convection term $\alpha\cdot\nabla\mathbf{u}$. The variable ϵ is the diffusivity and α the velocity. The ODE is formed by discretizing the 2D convection-diffusion equation using a finite difference scheme. The discrete Laplacian and gradient operators (L and D in the routine) represent diffusion and convection, respectively. The domain is discretized using a uniform grid with finite difference methods. For the Laplacian, a central difference is used, and for the gradient, a forward difference is used. The matrix $\mathbf{A}$ is then formed by combining the two discretized operators, and the solution is formed :

$$\mathbf{u}(t) = e^{\mathbf{A}t}\mathbf{u}_0 \tag{5.3}$$

Note that this equation (5.2) is a simple case of convection-diffusion where it is assumed that ϵ is constant, and that there are no sources or sinks (else it would make the equation slightly more complex).
We notice that the solution (equation 5.3) is a simple matrix-vector product $f(\mathbf{A})\mathbf{b}$ where $f() :=$ exp() and $\mathbf{b} = \mathbf{u}_0$. We will use the matrix-vector product routine described in section 3.1 to

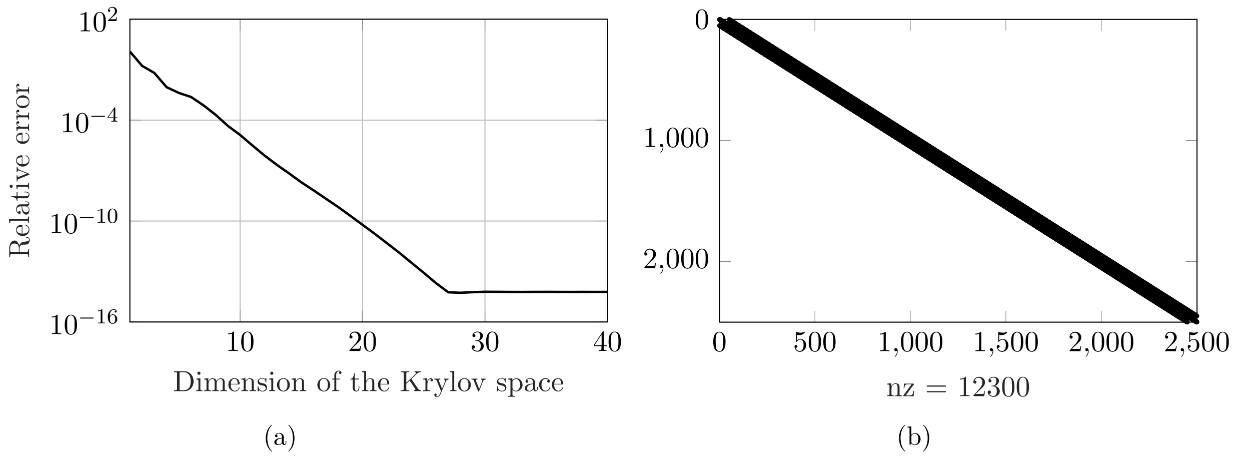


Figure 3: Convergence of our algorithm for computing $f(\mathbf{A})\mathbf{b}$ for solving the ODE (equation 5.3). We note that on our setup, $\mathbf{A} \in \mathbb{C}^{2500\times 2500}$. Machine precision is quickly reached, at only dimension 27 (a). The matrix $\mathbf{A}$ is strongly structured (b).

compute this solution. We will compare it to the naive approach of computing $f(\mathbf{A})$ explicitely, then multiplying it by $\mathbf{u}_0$.

5.1.2 Results

We observe that once again, working on a much small Krylov subspace give machine-precision approximation (figure 3). The optimal k found was 27, which should allow for a big speed-up in solving this ODE. Indeed, the naive approach takes 13.54 seconds to compute the solution, whereas when working on this smaller Krylov subspace, for k optimally chosen, it only took 225 milliseconds, this is almost a 60 times speed-up.
Interestingly, we note that the choice of parameters in the ODE is having a big impact on the performance of the algorithm. If we vary the diffusivity ϵ, we observe that the convergence is considerably slowed down (figure 4). The algorithm remains quicker than the naive approach, though it seems important to bear in mind that minor perturbation to some of the problem's component (figures 4a and 4b) can lead to a considerable change in the convergence behavior.

5.2 The sign function

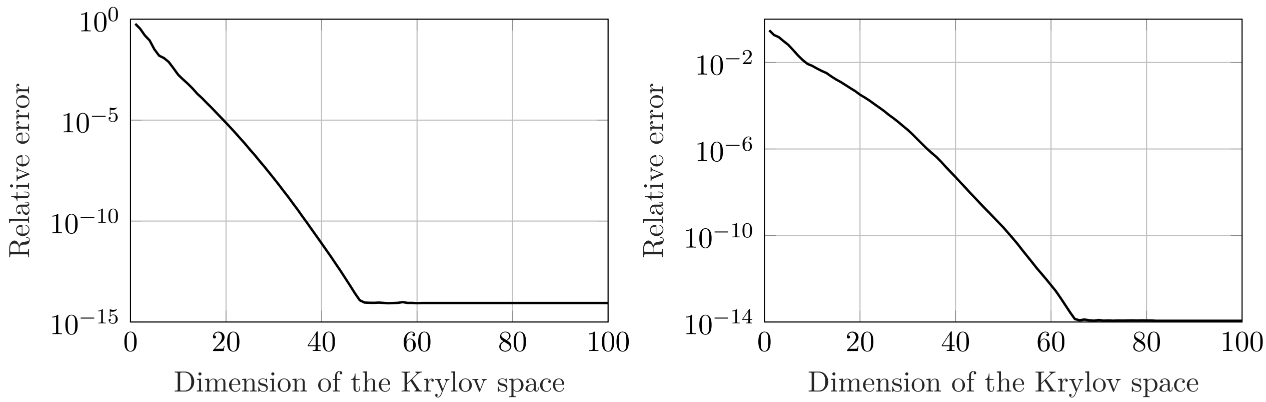
5.2.1 Background and theory

In control theory we are often interested in the eigenvalues λ of system matrices with $Re(\lambda) > 0$, since they correspond to unstable poles. In the design of controllers it is therefore interesting to have an efficient way to count the number of eigenvalues of a matrix in the right half-plane $Re(z) > 0$. Here we will build such a method.

Theorem 8. *Let $\mathbf{A} \in \mathbb{C}^{n\times n}$ be a matrix with k_- eigenvalues in the left plane, k_+ eigenvalues in the right plane and none on the imaginary axis, counting multiplicity. Let $sgn : \mathbb{C} \mapsto \{1, -1\}$ be defined by*

$$sgn(z) = \begin{cases} 1 & Re(z) \geq 0 \\ -1 & Re(z) < 0. \end{cases}$$

Then $trace(sgn(\mathbf{A})) = k_+ - k_-$.



(a) $\epsilon = 0.05$

(b) $\epsilon = 0.1$

Figure 4: Convergence of our algorithm for different diffusivity value ϵ. We note that as the dissufivity increases, the need to work in higher dimension increases too, thus slowing down the convergence of our algorithm.

Proof. Let $\mathbf{A} \in \mathbb{C}^{n \times n}$. To stay in a very general scenario, and consider cases where $\mathbf{A}$, let us consider its Jordan Canonical Form :

$$\mathbf{A} = \mathbf{V}\mathbf{J}\mathbf{V}^{-1}$$

Recall that by theorem 1, consider f a function, then

$$f(\mathbf{A}) = \mathbf{V}f(\mathbf{J})\mathbf{V}^{-1}$$

Thus let us consider the decomposition where

$$\mathbf{J} = \begin{pmatrix} \mathbf{J}_{k_+} & 0 \\ 0 & \mathbf{J}_{k_-} \end{pmatrix}$$

such that $\mathbf{J}_{k_+}$ has k_+ eigenvalues in the right plane, and $\mathbf{J}_{k_-}$ has k_- eigenvalues in the left plane. Then, we have that

$$\text{sgn}(\mathbf{J}) = \begin{pmatrix} \mathbf{I}_{k_+} & 0 \\ 0 & -\mathbf{I}_{k_-} \end{pmatrix}$$

where $\mathbf{I}_n$ is the identity matrix of size n. Then, we have that

$$\text{trace}(\text{sgn}(\mathbf{J})) = k_+ - k_-$$

And thus, we have that

$$\text{trace}(\text{sgn}(\mathbf{A})) = \text{trace}(\text{sgn}(\mathbf{V}\mathbf{J}\mathbf{V}^{-1})) = \mathbf{V}\text{trace}(\text{sgn}(\mathbf{J}))\mathbf{V}^{-1} = (k_+ - k_-)\mathbf{V}\mathbf{V}^{-1} = k_+ - k_-$$

□

5.2.2 Stability of a system

In this section, we will see how theorem 8 is a powerful tool for system stability assessment. Let us consider a system modeled by the matrix $\mathbf{A} \in \mathbb{C}^{n \times n}$, large and sparse. Say it has p distinct eigenvalues λ_i, $i = 1, \dots, p$. We know that if $\forall i, Re(\lambda_i) < 0$, then the system is stable. Using the sign function, it immediately rewrites, if $\forall i, sign(Re(\lambda_i)) = -1$ then the system is stable. From theorem 8, we can design an algorithm that counts positive eigenvalues of a system. Obviously, when n is extremely large, computing eigenvalues directly on $\mathbf{A}$ is not an option. Algorithm 3 takes advantage of the properties of Arnoldi method and estimates via Monte-Carlo sampling the number of positive eigenvalues of $\mathbf{A}$.

Algorithm 3: Computing $k_+(A)$

Data: $A \in \mathbb{C}^{n\times n}$, $k \in \mathbb{N}$, $N \in \mathbb{N}$
Result: $k_+ \in \mathbb{R}$
1 $\mathbf{q} \leftarrow \mathbf{0} \in \mathbb{R}^N$;
2 **for** $i = 1$ **to** N **do**
3 $\quad \mathbf{u} \leftarrow \mathrm{randn}(n, 1)$;
4 $\quad \hat{\mathbf{u}} \leftarrow \frac{\mathbf{u}}{\|\mathbf{u}\|}$;
5 $\quad [H, Q] \leftarrow \mathrm{arnoldi}(A, \hat{\mathbf{u}}, k)$;
6 $\quad q(i) \leftarrow \mathrm{trace}(\mathrm{sign}(H)))$;
7 **end**
8 $q_+ := \mathrm{mode}(\mathbf{q})$;
9 $k_+ \leftarrow (q + k)/2$

5.2.3 Limitation of simple Arnoldi Method

In the previous algorithm to compute positive eigenvalues, we assume the function `arnoldi()` is the Matlab's translation of Algorithm 1, *i.e* Modified Gram-Schmidt Arnoldi method. In this section we will see that the convergence of Arnoldi Method is actually not guaranteed in a reasonable time for all systems. The efficiency of the Arnoldi algorithm is highly correlated to the condition number of the matrix $\mathbf{A}$. We know from experience, that Ritz Value struggle to converge properly where eigenvalues of the matrix are very close to each other.
Provided with this note, a matrix that has those properties. Using Matlab's `eigs` function, we observe that the eigenvalues of $\mathbf{A}$ are clustered and very close to each other (figure 5a). This leads ultimately to very slow convergence of the eigenvalues using Arnolid method (figure 5c).
From this observation, we can easily say that with our previous implementation of Arnoldi, it seems unsafe to use Algorithm 3 as a tool to estimate the number of positive eigenvalues. We need to find a way to replace line 5 of Algorithm 3 by a more robust method.

5.2.4 Shifted-Invert Arnoldi Method

The Shifted-Invert Arnoldi method is a technique that allows for better convergence on ill-conditioned system, given we know some information on the spectrum of $\mathbf{A}$. The whole idea is to combine two ideas that are common to Power Iterations (Saad 2011), namely Shifted Power and Inverted iteration. The idea is fairly simple, instead of applying Arnoldi iterations on $\mathbf{A}$, you will apply it to $(\mathbf{A} - \sigma\mathbf{I})^{-1}$ where σ is a shift.
The interesting property about the shift is that it alters the eigenvalues but not the eigenvectors (Saad 2011). Once the eigenvalues μ of $(\mathbf{A} - \sigma\mathbf{I})^{-1}$ are computed, we can easily recover the eigenvalues of $\mathbf{A}$ by applying the shift σ and inverting them

$$\lambda_i = \frac{1}{\mu_i} + \sigma$$

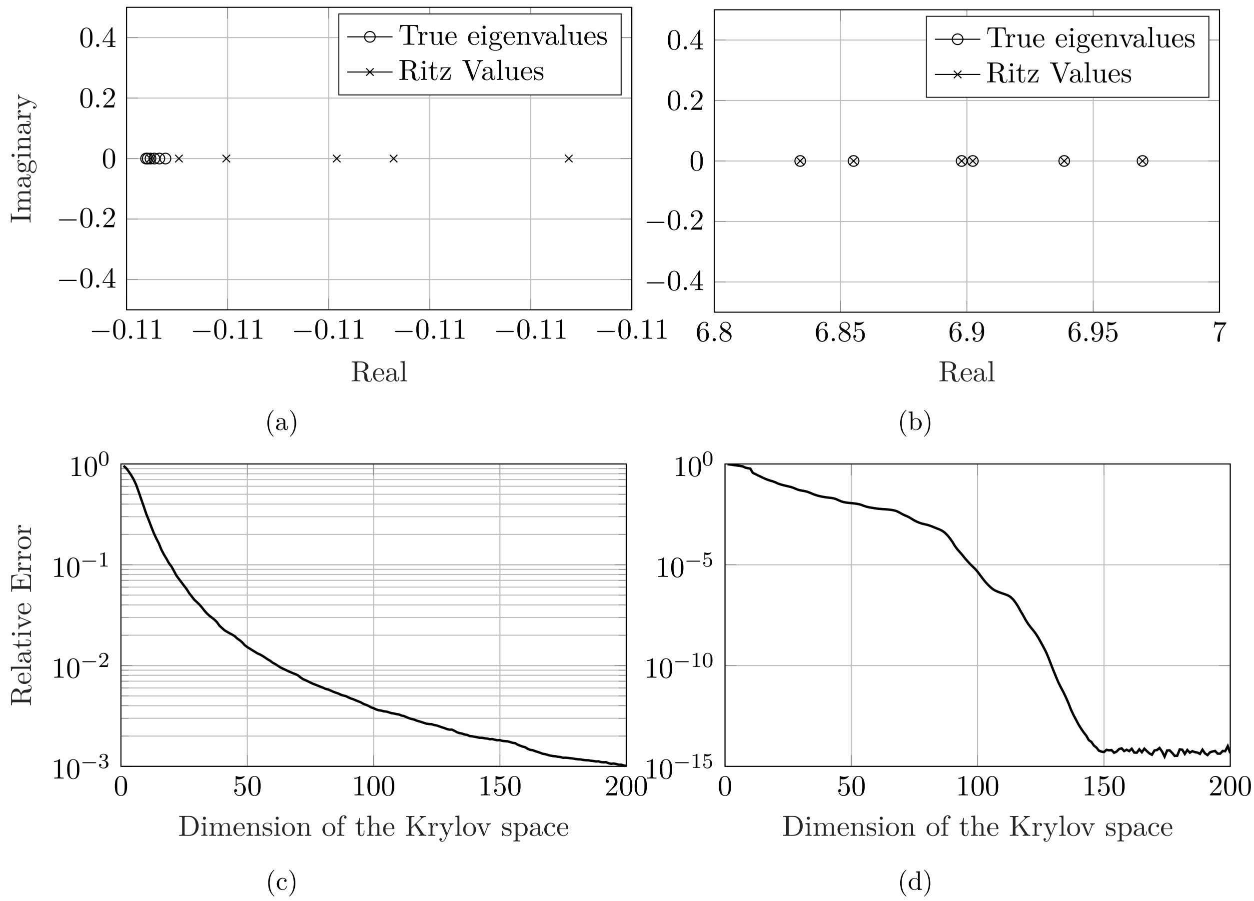


Figure 5: Comparison of the convergence of Arnoldi method to two different matrices. The first matrix (figures 5a and 5c) is from section 5.2. We see from figure 5a that the eigenvalues are clustered and very close to each other, leading to poor convergences of the Ritz values. We see that even after 200 iterations, the method is far from having converged. On the other end of the spectrum, when applied to the matrix we have studied earlier in the matrix-vector product, which is better conditioned (figures 5b and 5d), here, the eigenvalues are well spread, and the Ritz values converge quickly. We see that after 200 iterations, the Ritz values perfectly match the eigenvalues. Figures 5c and 5d show the relative error in the estimation of the eigenvalues. And indeed we see that for the first matrix, the relative error is still very high after 200 iterations, whereas for the second matrix, the relative error is equal to machine precision from 150 iterations.

Theorem 9. *Say you chose an arbitrary σ, then given λ_i and λ_j two eigenvalues of $\mathbf{A}$ such that $\forall k \neq i, j$, $\|\sigma - \lambda_k\| > \|\sigma - \lambda_i, j\|$, then the convergence factor of Shifted-Invert method is given by*

$$\rho_{\mathbf{I}} = \frac{|\sigma - \lambda_i|}{|\sigma - \lambda_j|} \tag{5.4}$$

Consequently, we see that a good initialization of σ is crucial to the convergence of the method. On the other end, a poor choice could have bad effects on the convergence, ultimately not converging at all. Several choices are open to us as for initilization of σ. One could stick with the unity, proceed to some power iterations, using the trace, or even use the Ritz values as computed in figure 5a. Inspired by the convergence rate of Power Iteration that is driven by the ratio of the two largest egeinvalues (in terms of magnitude), I suggest the following σ as a good estimator:

$$\sigma = \mathbb{E}\left[\frac{\lambda_1 + \lambda_2}{2}\right] \tag{5.5}$$

where λ_1 and λ_2 are the two largest eigenvalues of $\mathbf{A}$ in terms of magnitude. Computationally speaking, the Shifted-Invert procedure looks very heavy, especially considering the inverse operator : inverting such a large matrix is very costly. The good news is that we can take the LU factorization of $(\mathbf{A} - \sigma\mathbf{I})$, and the solve two linear system to compute the inverse. This is much more efficient than inverting the matrix directly. The Shifted-Invert Arnoldi method is described in Algorithm 4. Note that in algorithm 4, σ is not recomputed throughout the iterations. One way to potentially

Algorithm 4: Shifted-Invert Arnoldi Iteration

Data: $A \in \mathbb{C}^{n\times n}$, $v_1 \in \mathbb{C}^n$ a unit vector in the chosen norm, $\sigma \in \mathbb{C}$
Result: $H_n \in \mathbb{C}^{n\times n}$

$[L, U] \leftarrow \mathrm{lu}(A - \sigma I)$;
for $k = 1$**to**n **do**
    $\mathbf{w} \leftarrow U^{-1}L^{-1}\mathbf{v}_k$;
    for $i = 1$**to**k **do**
        $h_{ik} \leftarrow \mathbf{v}_i^*\mathbf{w}$;
        $\mathbf{w} \leftarrow \mathbf{w} - h_{ik}\mathbf{v}_i$;
    end
    $h_{k+1,k} \leftarrow \|\mathbf{w}\|$;
    if $h_{k+1,k} = 0$ **then**
        break;
    end
    $\mathbf{v_{k+1}} \leftarrow \mathbf{w}/h_{k+1,k}$
end

fasten the convergence, or to avoid having to define a good σ initially (which can be costly) is to recompute σ at each iteration (or at each i iteration). A drawback obviously is that it requires to recompute the LU factorization, which is arguabely the most costly operation in Algorithm 4. We will not investigate this further in this report, but it is worth noting that this could be a potential improvement to the algorithm.

If we take a look at the convergence of the Ritz Values, similarly as in figure 5, we note that with this method, Ritz values converges no matter the matrix (figure 6), and in much fewer iterations. We note that the final relative error is slightly higher than the machine precision ϵ, and thus is slightly higher than in the regular Arnoldi Method, maybe because there is a numerical bias introduced by the shift and inverse steps. However, the relative error remains very small, around 10^{-14}.

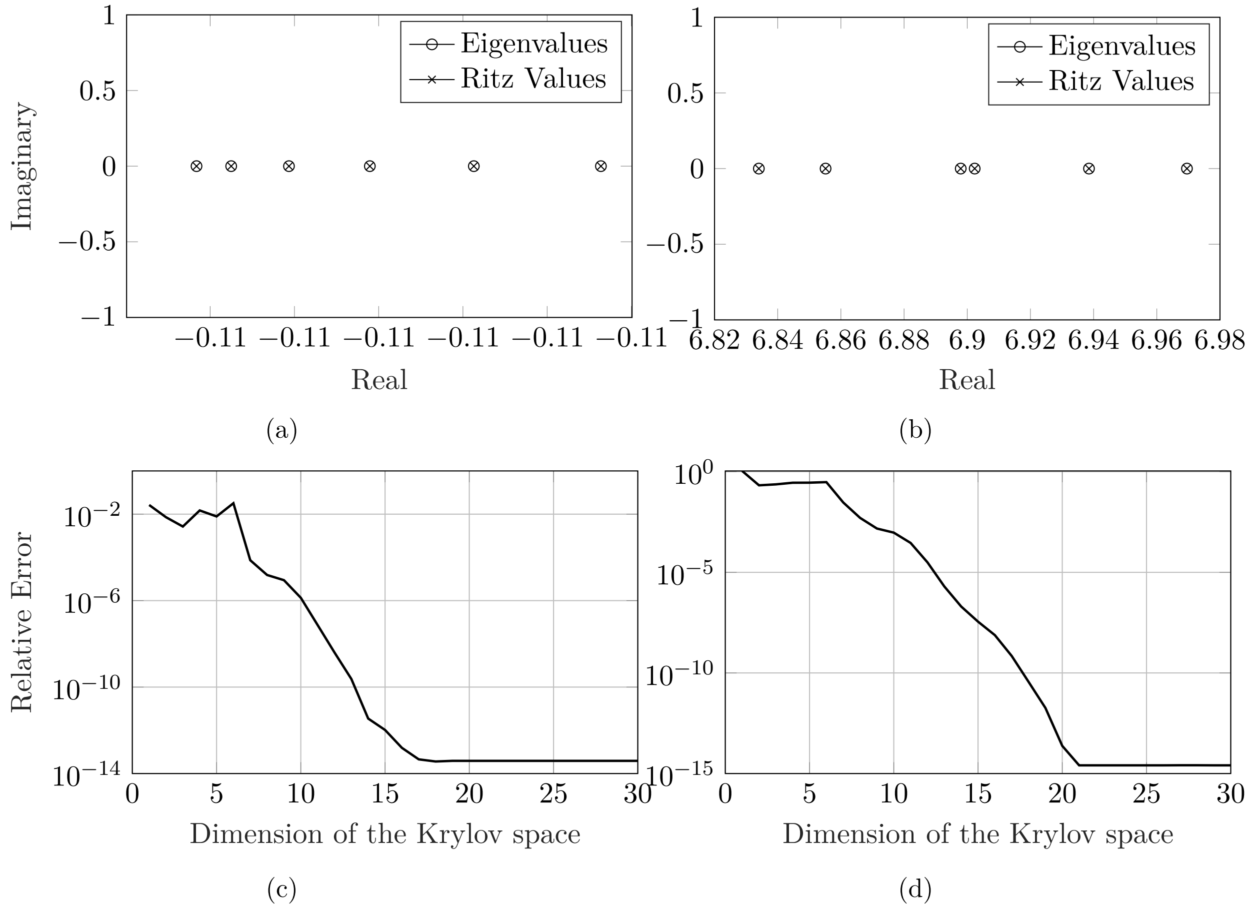


Figure 6: Same experiments as in figure 5 using a static $\sigma = \mathbb{E}[(\lambda_1 + \lambda_2)/2]$. We see that the convergence is much better than in the previous case, and that the Ritz values converge much faster in both matrices. The well-conditioned matrix (figures 6b and 6d) also converges, but un much fewer iterations. Finally, wee see that even the very ill-conditioned matrix (figures 6a and 6c) converges in only 20 iterations.

5.2.5 Counting positive eigenvalues

We now have an efficient arnoldi method for ill-conditioned matrices. Let us refine Algorithm 3 using the shifted invert Arnoldi method. To better assess the spectral component of the matrix, we will not use the same approach to define σ. Instead, σ will be defined by a uniformly distributed variable such that

$$\sigma \sim \mathcal{U}(lb, ub)$$

with

$$\begin{cases} lb = \mathbb{E}\left[\min(\mathrm{real}(\lambda_i))\right] \\ ub = \mathbb{E}\left[\max(\mathrm{real}(\lambda_i))\right] \end{cases}$$

This way, the shifted invert will, throughout the iteration, explore the whole spectrum of $\mathbf{A}$, and thus will be able to estimate the number of positive eigenvalues thanks to the convergence of the Monte-Carlo like approach. Algorithm 5 describes the new method.

Algorithm 5: Corrected Computing $k_+(A)$

Data: $A \in \mathbb{C}^{n\times n}$, $k \in \mathbb{N}$, $N \in \mathbb{N}$, $(lb, ub) \in \mathbb{R}^2$
Result: $k_+ \in \mathbb{R}$
$\mathbf{q} \leftarrow \mathbf{0} \in \mathbb{R}^N$;
for $i = 1$ **to** N **do**
  $\mathbf{u} \leftarrow \mathrm{randn}(n, 1)$;
  $\hat{\mathbf{u}} \leftarrow \frac{\mathbf{u}}{\|\mathbf{u}\|}$;
  $\sigma \leftarrow \mathrm{uniform}(lb, ub)$;
  $[H, Q] \leftarrow \mathrm{shifted\_invert\_arnoldi}(A, \hat{\mathbf{u}}, k, \sigma)$;
  $q(i) \leftarrow \mathrm{trace}(\mathrm{sign}(H))$;
end
$q_+ := \mathrm{mode}(\mathbf{q})$;
$k_+ \leftarrow (q + k)/2$

In figure 7, we show the comparison between Algorithm 3 and 5. For this case, the maximum dimension of the Krylov subspace k was set at 15, as we saw it provided good spectral approximation using the shifed-invert Arnoldi (figure 6). The number of samples is fairly low, we vary this number from 1 to 20.
We conclude that, assuming a good knowledge of the spectral properties of $\mathbf{A}$ (here, the bounds of its spectrum) enables for faster and more precise computations. In this case, it allows for quick assessment of a system's stability, which is a very important property in control theory.

References

Davies, Philip I and Nicholas J Higham (2003). "A Schur-Parlett algorithm for computing matrix functions". In: *SIAM Journal on Matrix Analysis and Applications* 25.2, pp. 464–485.

Frommer, Andreas and Valeria Simoncini (2008). "Matrix functions". In: *Model order reduction: theory, research aspects and applications*. Springer, pp. 275–303.

George, Alan and Joseph WH Liu (1978). "An automatic nested dissection algorithm for irregular finite element problems". In: *SIAM Journal on Numerical Analysis* 15.5, pp. 1053–1069.

Saad, Yousef (2011). *Numerical methods for large eigenvalue problems: revised edition*. SIAM.

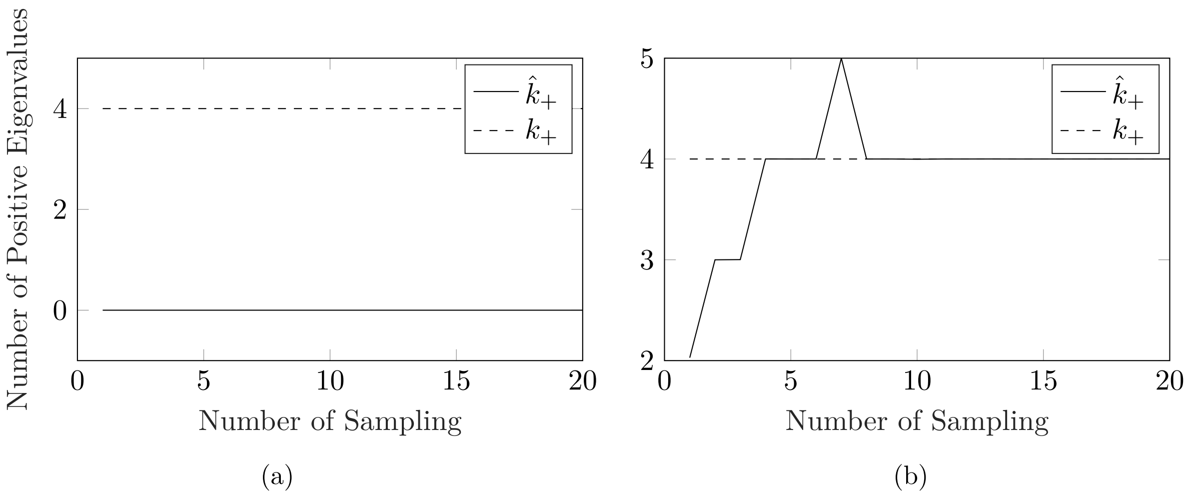


Figure 7: Comparison of the estimation of k_p with Algorithm 3 and 5. We see that Algorithm 3 (figure 7a) does not manage to find any positive eigenvalues. It uses the standard Arnoldi method, and we saw previously that it was not converging. Whereas Algorithm 5 (figure 7b) finds the correct value of k_p very quickly. $\hat{k}_+$ (continuous line) is the estimated value of k_+, and k_+ (dashed line) is the true value of k_+.