The Two-Pass Lanczos Method

Luca Lombardo

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Contents

1 Introduction

The computation of the action of a matrix function on a vector, an operation denoted as $\mathbf{x} = f(\mathbf{A})\mathbf{b}$, represents a fundamental task in numerical linear algebra and scientific computing [1, 2]. For large, sparse Hermitian matrices, methods based on Krylov subspaces are standard. A foundational algorithm in this class is the Lanczos process, first introduced as a *method of minimized iterations* for eigenvalue problems [3]. Its standard implementation, however, encounters a severe practical limitation: the necessity of storing an ever-growing basis of Lanczos vectors. This memory requirement often becomes prohibitive for the large-scale problems encountered in practice [1].

To address this memory bottleneck, we examine a simple and effective variant known as the two-pass Lanczos method [2]. This approach fundamentally alters the standard algorithm by separating the computation into two distinct phases. It first generates the projection of the matrix without storing the basis, and then regenerates the basis in a second pass to construct the final solution. This report provides an analysis of this method, focusing on the explicit trade-off it presents: a significant reduction in memory storage at the cost of an increased computational workload.

1.1 The Problem of Computing Matrix Functions

We consider the problem of computing the vector $\mathbf{x} \in \mathbb{C}^n$ defined by the action of a matrix function on a vector $\mathbf{b} \in \mathbb{C}^n$,

$$\mathbf{x} = f(\mathbf{A})\mathbf{b}, \tag{1}$$

where $\mathbf{A} \in \mathbb{C}^{n\times n}$ is a large, sparse Hermitian matrix and $f : \Omega \subseteq \mathbb{C} \to \mathbb{C}$ is a function defined on the spectrum of $\mathbf{A}$, $\sigma(\mathbf{A}) \subset \Omega$. The explicit computation of the matrix $f(\mathbf{A})$ is generally infeasible due to its high computational cost and the fact that $f(\mathbf{A})$ is typically a dense matrix even when $\mathbf{A}$ is sparse [2]. Our focus, therefore, is on methods that compute $\mathbf{x}$ directly.

This problem is fundamental in many areas of scientific computing [1]. A primary example is the solution of a large system of linear equations $\mathbf{Ax} = \mathbf{b}$, which corresponds to the case where $f(z) = z^{-1}$ and $\mathbf{A}$ is nonsingular [4]. Another critical application arises in the numerical solution of systems of linear ordinary differential equations. The solution to the initial value problem $\mathbf{y}'(t) = \mathbf{Ay}(t)$, with $\mathbf{y}(0) = \mathbf{y}_0$, is given by $\mathbf{y}(t) = \exp(t\mathbf{A})\mathbf{y}_0$. This computation is an instance of our problem where the function is the matrix exponential, $f(z) = \exp(tz)$ [2].

1.2 Krylov Subspace Methods: The Standard Approach

The standard framework for computing an approximation to $\mathbf{x} = f(\mathbf{A})\mathbf{b}$ for large matrices is based on projection onto a Krylov subspace [2, 4]. We begin by formally defining this subspace.

Definition 1.1 (Krylov Subspace)**.** Given a matrix $\mathbf{A} \in \mathbb{C}^{n\times n}$ and a vector $\mathbf{b} \in \mathbb{C}^n$, the k-th Krylov subspace generated by $\mathbf{A}$ and $\mathbf{b}$ is the vector space

$$\mathcal{K}_k(\mathbf{A}, \mathbf{b}) = \operatorname{span}\{\mathbf{b}, \mathbf{Ab}, \ldots, \mathbf{A}^{k-1}\mathbf{b}\}. \tag{2}$$

The fundamental principle of a Krylov subspace method is to seek an approximate solution $\mathbf{x}_k$ within the low-dimensional subspace $\mathcal{K}_k(\mathbf{A}, \mathbf{b})$, where $k \ll n$. This is achieved through a projection process that can be summarized in three distinct stages. First, we construct an orthonormal basis, typically via the Arnoldi or Lanczos iteration, for the subspace $\mathcal{K}_k(\mathbf{A}, \mathbf{b})$. Let this basis be the columns of the matrix $\mathbf{V}_k \in \mathbb{C}^{n \times k}$.

Second, we project the high-dimensional operator $\mathbf{A}$ onto $\mathcal{K}_k(\mathbf{A}, \mathbf{b})$, which yields a small $k \times k$ matrix representation of the operator, typically a Hessenberg or tridiagonal matrix $\mathbf{T}_k = \mathbf{V}_k^H \mathbf{A} \mathbf{V}_k$. The original problem is thereby reduced to the evaluation of $f(\mathbf{T}_k)$, a task which is computationally feasible for small k.

Finally, the solution to the projected problem is lifted back to the original n-dimensional space to form the final approximation. Assuming the starting vector for the basis construction is $\mathbf{v}_1 = \mathbf{b}/\|\mathbf{b}\|_2$, the approximation $\mathbf{x}_k$ to $\mathbf{x}$ is given by

$$\mathbf{x}_k = \mathbf{V}_k f(\mathbf{T}_k) \mathbf{e}_1 \|\mathbf{b}\|_2, \tag{3}$$

where $\mathbf{e}_1$ is the first canonical basis vector in $\mathbb{C}^k$ [2].

1.3 The Memory Bottleneck of the Standard Lanczos Process

The practical utility of approximating the solution via $\mathbf{x}_k = \mathbf{V}_k f(\mathbf{T}_k) \mathbf{e}_1 \|\mathbf{b}\|_2$ is conditioned on our ability to construct and store the basis matrix $\mathbf{V}_k$. In a standard one-pass implementation, the Lanczos vectors $\{\mathbf{v}_j\}_{j=1}^k$ are generated sequentially and must all be retained in memory. This is because the final step involves forming a linear combination of these vectors, weighted by the components of the vector $\mathbf{y}_k = f(\mathbf{T}_k) \mathbf{e}_1 \|\mathbf{b}\|_2$.

This requirement imposes a memory cost that scales as $O(nk)$. For many problems of practical interest, particularly those arising from the discretization of partial differential equations, the dimension n can be on the order of millions or larger. Concurrently, achieving a desired accuracy may require a number of iterations k that is substantial, leading to a storage demand that exceeds the capacity of modern computing systems [1]. This memory constraint is the primary bottleneck of the standard Lanczos process.

To overcome this limitation, memory-efficient strategies are necessary. The two-pass Lanczos method is a simple but effective approach designed explicitly to address this problem of memory consumption [2]. The method is structured to avoid storing the entire basis $\mathbf{V}_k$ by decoupling the generation of the projected matrix $\mathbf{T}_k$ from the final synthesis of the solution vector $\mathbf{x}_k$.

2 Symmetric Lanczos Process

To analyze the two-pass variant, we first take a step back and review the standard symmetric Lanczos process. The derivation begins from the method of minimized iterations, as originally formulated by Lanczos [3]. This approach demonstrates that the process of generating orthogonal

vectors through successive applications of a symmetric operator yields a three-term recurrence relation. We then formalize this recurrence using modern matrix notation [1].

2.1 Derivation via Successive Orthogonalization

We derive the Lanczos recurrence following the method of minimized iterations for a Hermitian operator $\mathbf{A} \in \mathbb{C}^{n\times n}$ [3]. The procedure constructs a sequence of mutually orthogonal vectors $\{\mathbf{b}_k\}_{k=0}^{m-1}$ from an arbitrary starting vector $\mathbf{b}_0 \in \mathbb{C}^n$, where $\mathbf{b}_0 \neq \mathbf{0}$.

The sequence is generated iteratively. At each step $k \geq 1$, a new vector $\mathbf{b}_k$ is formed by minimizing the Euclidean norm of a vector obtained by applying $\mathbf{A}$ to the previous vector $\mathbf{b}_{k-1}$ and removing its components along all previously generated vectors. For the first step, we define $\mathbf{b}_1$ as

$$\mathbf{b}_1 = \mathbf{A}\mathbf{b}_0 - \alpha_0 \mathbf{b}_0,$$

where the scalar α_0 is chosen to minimize $\|\mathbf{b}_1\|_2$. This minimization is equivalent to enforcing the orthogonality condition $\mathbf{b}_1 \perp \mathbf{b}_0$. The inner product $(\mathbf{A}\mathbf{b}_0 - \alpha_0\mathbf{b}_0, \mathbf{b}_0) = 0$ yields the coefficient

$$\alpha_0 = \frac{(\mathbf{A}\mathbf{b}_0, \mathbf{b}_0)}{(\mathbf{b}_0, \mathbf{b}_0)}.$$

The general step for $k \geq 2$ is to construct $\mathbf{b}_k$ from $\mathbf{A}\mathbf{b}_{k-1}$ by ensuring it is orthogonal to the entire set $\{\mathbf{b}_0, \mathbf{b}_1, \ldots, \mathbf{b}_{k-1}\}$. This is achieved by defining $\mathbf{b}_k$ as

$$\mathbf{b}_k = \mathbf{A}\mathbf{b}_{k-1} - \sum_{i=0}^{k-1} c_i \mathbf{b}_i.$$

The coefficients c_i are determined by the orthogonality conditions $(\mathbf{b}_k, \mathbf{b}_i) = 0$ for $i = 0, \ldots, k-1$. Due to the mutual orthogonality of the basis vectors $\{\mathbf{b}_j\}$, this simplifies to

$$c_i = \frac{(\mathbf{A}\mathbf{b}_{k-1}, \mathbf{b}_i)}{(\mathbf{b}_i, \mathbf{b}_i)}.$$

A key property emerges when the operator $\mathbf{A}$ is Hermitian. For any i, we can write

$$(\mathbf{A}\mathbf{b}_{k-1}, \mathbf{b}_i) = (\mathbf{b}_{k-1}, \mathbf{A}\mathbf{b}_i).$$

From the construction of the sequence, the vector $\mathbf{b}_{i+1}$ is a linear combination of $\mathbf{A}\mathbf{b}_i$ and the preceding vectors $\mathbf{b}_i, \ldots, \mathbf{b}_0$. It follows that $\mathbf{A}\mathbf{b}_i$ lies in the span of $\{\mathbf{b}_0, \ldots, \mathbf{b}_{i+1}\}$. By the orthogonality of the sequence, the inner product $(\mathbf{b}_{k-1}, \mathbf{A}\mathbf{b}_i)$ must be zero if $k-1 > i+1$, or equivalently, if $i < k-2$. Consequently, the coefficients c_i are zero for all $i < k-2$.

This establishes that the summation in the general step collapses, leaving a three-term recurrence relation. Following the notation in [3], we define $\alpha_{k-1} = c_{k-1}$ and $\beta_{k-2} = c_{k-2}$. The recurrence simplifies to

$$\mathbf{b}_k = \mathbf{A}\mathbf{b}_{k-1} - \alpha_{k-1}\mathbf{b}_{k-1} - \beta_{k-2}\mathbf{b}_{k-2},$$

which is the foundational recurrence of the symmetric Lanczos process.

2.2 Matrix Representation of the Lanczos Recurrence

The three-term recurrence relation derived from the principle of minimized iterations can be expressed in a compact matrix form. Let us define a sequence of orthonormal vectors $\{\mathbf{v}_j\}_{j=1}^k$ from the orthogonal vectors $\{\mathbf{b}_j\}_{j=1}^k$. We initialize the process with a unit-norm vector $\mathbf{v}_1 = \mathbf{b}_0/\|\mathbf{b}_0\|_2$, where we adopt the notation from [4]. The recurrence relation can be written as

$$\beta_j \mathbf{v}_{j+1} = \mathbf{A}\mathbf{v}_j - \alpha_j \mathbf{v}_j - \beta_{j-1}\mathbf{v}_{j-1},$$

where, following the convention in [4], we set $\mathbf{v}_0 = \mathbf{0}$. The orthonormality of the vectors $\{\mathbf{v}_j\}$ dictates the choice of the coefficients. Specifically, taking the inner product of the above relation with $\mathbf{v}_j$ yields

$$\alpha_j = \mathbf{v}_j^H \mathbf{A}\mathbf{v}_j.$$

The coefficient β_j is determined by the normalization condition $\|\mathbf{v}_{j+1}\|_2 = 1$, which gives $\beta_j = \|\mathbf{A}\mathbf{v}_j - \alpha_j\mathbf{v}_j - \beta_{j-1}\mathbf{v}_{j-1}\|_2$.

After k steps of this process, for $j = 1,\ldots,k$, we can write the set of recurrence relations in matrix form. Let $\mathbf{V}_k = [\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k] \in \mathbb{C}^{n\times k}$ be the matrix whose columns are the Lanczos vectors. The recurrences can be collected into a single matrix equation

$$\mathbf{A}\mathbf{V}_k = \mathbf{V}_k\mathbf{T}_k + \beta_k \mathbf{v}_{k+1}\mathbf{e}_k^T,$$

where $\mathbf{e}_k$ is the k-th canonical basis vector in $\mathbb{C}^k$, and $\mathbf{T}_k$ is the $k \times k$ real symmetric tridiagonal matrix

$$\mathbf{T}_k = \begin{pmatrix} \alpha_1 & \beta_1 & & \\ \beta_1 & \alpha_2 & \ddots & \\ & \ddots & \ddots & \beta_{k-1} \\ & & \beta_{k-1} & \alpha_k \end{pmatrix}$$

From the orthonormality of the columns of $\mathbf{V}_k$, i.e., $\mathbf{V}_k^H\mathbf{V}_k = \mathbf{I}_k$, it follows that the tridiagonal matrix $\mathbf{T}_k$ is the orthogonal projection of $\mathbf{A}$ onto the Krylov subspace $\mathcal{K}_k(\mathbf{A}, \mathbf{v}_1)$, since

$$\mathbf{V}_k^H\mathbf{A}\mathbf{V}_k = \mathbf{V}_k^H(\mathbf{V}_k\mathbf{T}_k + \beta_k\mathbf{v}_{k+1}\mathbf{e}_k^T) = \mathbf{T}_k.$$

The tridiagonal and symmetric structure of $\mathbf{T}_k$ is a direct consequence of the three-term recurrence relation inherent to the orthogonalization process with a Hermitian operator.

3 The Two-Pass Lanczos Algorithm

Having established the theoretical framework for the standard Lanczos process and identified its principal limitation[1] we now present a memory-efficient variant known as the two-pass Lanczos algorithm [2, 5]. The algorithm resolves the memory constraint by decoupling the computation of the projected tridiagonal matrix $\mathbf{T}_k$ from the synthesis of the final solution vector $\mathbf{x}_k$.

[1] the prohibitive memory cost of storing the basis vectors

In this section, we provide a description of the two distinct phases of the algorithm: an initial pass to generate the coefficients of $\mathbf{T}_k$ without storing the basis, and a second pass to reconstruct the solution vector. We then present an analysis of the method's core trade-off, quantifying its memory savings against its increased computational cost.

3.1 Algorithmic Formulation

The two-pass Lanczos algorithm computes the standard Lanczos approximation $\mathbf{x}_k$ while avoiding the $O(nk)$ memory cost associated with storing the basis matrix $\mathbf{V}_k$. It achieves this by executing two distinct computational passes. The first pass computes the tridiagonal projection $\mathbf{T}_k$, and the second pass synthesizes the solution vector $\mathbf{x}_k$.

3.1.1 First Pass: Projection and Coefficient Generation

The objective of the first pass is to compute the scalar entries of the $k \times k$ symmetric tridiagonal matrix $\mathbf{T}_k$. This is accomplished by executing k steps of the Lanczos iteration. The process generates a sequence of orthonormal vectors $\{\mathbf{v}_j\}$ via a three-term recurrence, which we derive here following the formulation in [1].

The process is initialized with a starting vector of unit norm, $\mathbf{v}_1 = \mathbf{b}/\|\mathbf{b}\|_2$. The core of the algorithm is the following recurrence relation, where we define $\beta_0 = 0$ and $\mathbf{v}_0 = \mathbf{0}$:

$$\beta_j \mathbf{v}_{j+1} = \mathbf{A}\mathbf{v}_j - \alpha_j \mathbf{v}_j - \beta_{j-1}\mathbf{v}_{j-1}. \tag{4}$$

The scalars α_j and β_j are determined at each step $j = 1, \ldots, k$ by enforcing the orthonormality of the sequence $\{\mathbf{v}_j\}$. Taking the inner product of equation (4) with $\mathbf{v}_j$ and leveraging the orthogonality of the previously constructed vectors, we obtain the expression for the diagonal elements of $\mathbf{T}_k$:

$$\alpha_j = \mathbf{v}_j^H \mathbf{A}\mathbf{v}_j.$$

The off-diagonal elements, β_j, are determined by the normalization condition $\|\mathbf{v}_{j+1}\|_2 = 1$. Let $\mathbf{r}_j$ be the residual vector on the right-hand side of (4) before normalization:

$$\mathbf{r}_j = \mathbf{A}\mathbf{v}_j - \alpha_j \mathbf{v}_j - \beta_{j-1}\mathbf{v}_{j-1}.$$

Then, the coefficient β_j is its Euclidean norm,

$$\beta_j = \|\mathbf{r}_j\|_2.$$

If $\beta_j = 0$, the process terminates. Otherwise, the next Lanczos vector is computed as $\mathbf{v}_{j+1} = \mathbf{r}_j/\beta_j$.

The structure of the three-term recurrence in (4) is a critical property for memory-efficient implementations. In exact arithmetic, it ensures that the newly generated vector $\mathbf{v}_{j+1}$ is orthogonal to all preceding vectors $\mathbf{v}_1, \ldots, \mathbf{v}_j$, despite its construction using only $\mathbf{v}_j$ and $\mathbf{v}_{j-1}$ [1]. This allows for the sequential generation of the basis while only requiring storage for a constant number of

n-dimensional vectors at any given time, thus addressing the memory bottleneck of the standard Lanczos process [2].

After k iterations, the stored sequences of scalars $\{\alpha_j\}_{j=1}^{k}$ and $\{\beta_j\}_{j=1}^{k-1}$ are used to construct the real, symmetric tridiagonal matrix $\mathbf{T}_k$. The final operation of the first pass is the computation of the coefficient vector $\mathbf{y}_k \in \mathbb{C}^k$:

$$\mathbf{y}_k = f(\mathbf{T}_k)\mathbf{e}_1 \|\mathbf{b}\|_2. \tag{5}$$

This vector contains the coordinates of the approximate solution $\mathbf{x}_k$ with respect to the orthonormal basis $\mathbf{V}_k$. The computation of $f(\mathbf{T}_k)$ is a small-scale dense matrix problem whose cost is independent of n, and is thus considered negligible for $k \ll n$.

3.1.2 Second Pass: Solution Reconstruction

In the second pass, we construct the final solution vector $\mathbf{x}_k$. The vector $\mathbf{x}_k$ is the linear combination of the Lanczos basis vectors, where the coefficients are the components of the vector $\mathbf{y}_k$ computed in the first pass. We define the approximation as

$$\mathbf{x}_k = \mathbf{V}_k\mathbf{y}_k = \sum_{j=1}^{k} (\mathbf{y}_k)_j \mathbf{v}_j. \tag{6}$$

To compute this sum without storing the matrix $\mathbf{V}_k$, we re-generate the Lanczos vectors sequentially [2].

We re-initialize the process with the starting vector $\mathbf{v}_1 = \mathbf{b}/\|\mathbf{b}\|_2$. We then execute the Lanczos iteration a second time, using the scalars $\{\alpha_j\}_{j=1}^{k}$ and $\{\beta_j\}_{j=1}^{k-1}$ that we stored during the first pass. For each $j = 1, \ldots, k-1$, we reconstruct the subsequent Lanczos vectors via the same three-term recurrence from equation (4):

$$\mathbf{v}_{j+1} = \frac{1}{\beta_j}\left(\mathbf{A}\mathbf{v}_j - \alpha_j\mathbf{v}_j - \beta_{j-1}\mathbf{v}_{j-1}\right). \tag{7}$$

As we re-generate each vector $\mathbf{v}_j$, we immediately use it to form the sum in equation (6). We initialize the solution vector $\mathbf{x}_k$ to zero. After computing $\mathbf{v}_1$, we form the first term of the sum. Then, for each $j = 1, \ldots, k-1$, after computing $\mathbf{v}_{j+1}$ via (7), we update the solution as follows:

$$\mathbf{x}_k \leftarrow \mathbf{x}_k + (\mathbf{y}_k)_{j+1}\mathbf{v}_{j+1}. \tag{8}$$

The memory management is identical to that of the first pass. We use each vector $\mathbf{v}_j$ to compute $\mathbf{v}_{j+1}$ and to update the sum for $\mathbf{x}_k$. Afterwards, we retain it only as long as required for the next step of the recurrence. After k steps, the accumulation is complete. The procedure is formally described in Algorithm 1.

Algorithm 1 The Two-Pass Lanczos Method for $\mathbf{x} = f(\mathbf{A})\mathbf{b}$

Input: Hermitian matrix $\mathbf{A} \in \mathbb{C}^{n\times n}$, vector $\mathbf{b} \in \mathbb{C}^n$, function f, number of iterations k.
Output: Approximate solution $\mathbf{x}_k \in \mathbb{C}^n$.

▷ — *First Pass: Coefficient Generation* —

Store $\mathbf{b}$ for the second pass. Let $\|\mathbf{b}\|_2$ be its norm.
$\beta_0 \leftarrow 0$, $\mathbf{v}_{\text{prev}} \leftarrow \mathbf{0}$, $\mathbf{v} \leftarrow \mathbf{b}/\|\mathbf{b}\|_2$.
for $j = 1, \ldots, k$ **do**
  $\mathbf{w} \leftarrow \mathbf{A}\mathbf{v} - \beta_{j-1}\mathbf{v}_{\text{prev}}$
  $\alpha_j \leftarrow \mathbf{v}^H\mathbf{w}$
  $\mathbf{w} \leftarrow \mathbf{w} - \alpha_j\mathbf{v}$
  $\beta_j \leftarrow \|\mathbf{w}\|_2$
  Store α_j and β_j.
  if $\beta_j = 0$ **then** break **end if**
  $\mathbf{v}_{\text{prev}} \leftarrow \mathbf{v}$
  $\mathbf{v} \leftarrow \mathbf{w}/\beta_j$
end for
Let k be the final iteration count.
Construct $\mathbf{T}_k \in \mathbb{R}^{k\times k}$ from stored $\{\alpha_j\}_{j=1}^{k}$ and $\{\beta_j\}_{j=1}^{k-1}$.
Compute coefficient vector $\mathbf{y}_k \leftarrow f(\mathbf{T}_k)\mathbf{e}_1\|\mathbf{b}\|_2$.

▷ — *Second Pass: Solution Reconstruction* —

$\beta_0 \leftarrow 0$, $\mathbf{v}_{\text{prev}} \leftarrow \mathbf{0}$, $\mathbf{v} \leftarrow \mathbf{b}/\|\mathbf{b}\|_2$.
$\mathbf{x}_k \leftarrow (\mathbf{y}_k)_1\mathbf{v}$.
for $j = 1, \ldots, k-1$ **do**
  $\mathbf{w} \leftarrow \mathbf{A}\mathbf{v} - \alpha_j\mathbf{v} - \beta_{j-1}\mathbf{v}_{\text{prev}}$ ▷ Use stored α_j, β_{j-1}
  $\mathbf{v}_{\text{next}} \leftarrow \mathbf{w}/\beta_j$ ▷ Use stored β_j
  $\mathbf{x}_k \leftarrow \mathbf{x}_k + (\mathbf{y}_k)_{j+1}\mathbf{v}_{\text{next}}$
  $\mathbf{v}_{\text{prev}} \leftarrow \mathbf{v}$
  $\mathbf{v} \leftarrow \mathbf{v}_{\text{next}}$
end for
return $\mathbf{x}_k$.

3.2 Computational and Memory Complexity

We now provide an analysis of the memory requirements and computational cost of the two-pass Lanczos algorithm 1. Our analysis demonstrates that the method achieves a significant reduction in memory usage at the expense of an increased number of floating-point operations.

Proposition 3.1 (Memory Complexity)**.** *Let n be the dimension of the matrix $\mathbf{A}$ and k be the number of iterations. The standard one-pass Lanczos algorithm requires $O(nk)$ memory for the basis vectors. The two-pass Lanczos algorithm reduces this requirement to $O(n)$.*

Proof. The standard one-pass Lanczos method must store the entire basis matrix $\mathbf{V}_k = [\mathbf{v}_1, \ldots, \mathbf{v}_k] \in \mathbb{C}^{n\times k}$ to synthesize the final solution $\mathbf{x}_k = \mathbf{V}_k\mathbf{y}_k$. The storage for this matrix is of size nk, leading to a memory complexity of $O(nk)$ [1].

We contrast this with the requirements of the two-pass algorithm. During the first pass, the computation of $\mathbf{v}_{j+1}$ at any step j of the recurrence relation (4) depends only on the vectors $\mathbf{v}_j$ and $\mathbf{v}_{j-1}$. Therefore, the number of n-dimensional vectors that we must hold simultaneously in memory is a small constant, independent of the iteration count j. The memory for these vectors is $O(n)$. The second pass has analogous requirements, as it employs the same recurrence. The total memory complexity is therefore dominated by the storage for a few n-dimensional vectors, which amounts to $O(n)$. We must also store the scalar coefficients, but this only adds an $O(k)$ term, which is negligible for $k \ll n$. □

Proposition 3.2 (Computational Complexity)**.** *Let the dominant computational cost of the Lanczos process be the matrix-vector products. For k iterations, the standard one-pass Lanczos algorithm requires k matrix-vector products. The two-pass algorithm requires $2k$ matrix-vector products.*

Proof. The primary computational cost per iteration for a large, sparse matrix $\mathbf{A}$ is the matrix-vector product $\mathbf{A}\mathbf{v}_j$. The standard one-pass method performs one such product per iteration, resulting in a total of k matrix-vector products.

The two-pass method, by its design, executes two separate passes. The first pass requires k matrix-vector products to generate the coefficients of $\mathbf{T}_k$. To synthesize the solution, the second pass must re-generate the Lanczos basis vectors, which forces us to re-compute the same sequence of k matrix-vector products. This brings the total number of matrix-vector products to $2k$. □

4 Numerical Experiments

In this section, we present a series of numerical experiments designed to validate the theoretical analysis and to assess the practical performance of the two-pass Lanczos algorithm. Our investigation is structured around three distinct objectives: to provide empirical evidence for the memory-versus-computation trade-off; to evaluate the scalability of both algorithms with respect to the problem dimension; and to conduct an analysis of the numerical stability and the potential for error propagation in the second pass.

4.1 Experimental Setup and Performance Metrics

All experiments were executed on a dual-socket machine running Ubuntu 22.04.2 LTS (GNU/Linux kernel 5.15.0-119-generic). The system is equipped with two Intel(R) Xeon(R) Gold 5318Y CPUs, each operating at a base frequency of 2.10GHz, for a total of 96 logical cores. The algorithms were implemented in Rust and compiled with `rustc` version 1.89.0 using release-level optimizations. All computations were performed using double-precision floating-point arithmetic.

We defined a set of quantitative metrics. The primary measure of computational work is the total number of matrix-vector products, which provides a hardware-independent basis for comparison [4], as it is the dominant operation in Krylov subspace methods for large, sparse matrices. We also report the total wall-clock time as a practical measure of performance. The mem-

ory footprint is quantified by the Peak Resident Set Size, corresponding to the `VmPeak` field in the `/proc/self/status` virtual file on our Linux system.

We measure the accuracy of a computed solution x_k by its relative error with respect to a known ground-truth solution x_{true}. We assess the stability of the basis generation by quantifying the loss of orthogonality of the Lanczos basis V_k via the Frobenius norm $\|I - V_k^H V_k\|_F$, a known phenomenon in finite-precision implementations of the process [1]. For the stability experiment, we also measure the numerical deviation, or *drift*, between the standard basis V_k and the re-generated basis V_k', as well as the drift in the scalar recurrence coefficients. We evaluate the ultimate impact by the relative difference between the final solution vectors obtained from the one-pass and two-pass methods.

4.2 Test Problem Generation

We derive our test problems from Karush-Kuhn-Tucker (KKT) systems associated with convex quadratic separable Min-Cost Flow problems. Such systems are sparse, symmetric, and exhibit the block saddle-point structure

$$A = \begin{pmatrix} D & E^T \\ E & 0 \end{pmatrix} \in \mathbb{R}^{n \times n}. \tag{9}$$

We construct the node-arc incidence matrix, E, from network topologies generated by the `netgen` utility, which allows for precise control over problem dimension and sparsity. The diagonal block D is defined as $D = \text{diag}(d_1, \ldots, d_m)$, with its entries drawn from a uniform random distribution, $d_i \sim U[1, C_D]$.

This construction allows us to control key properties of A. First, by setting $d_i \geq 1$, we ensure that D is a symmetric positive definite matrix, as all its eigenvalues are positive. This choice is critical because it guarantees that the resulting KKT matrix A is indefinite. The indefiniteness can be directly verified by examining the quadratic form $x^T Ax$: vectors of the form $[x_1^T, 0]^T$ with $x_1 \neq 0$ can yield positive values since $x_1^T D x_1 > 0$, while other choices of x can lead to non-positive values. Operating on indefinite matrices provides a robust test for the Lanczos algorithm, which is formulated for general symmetric matrices and does not require positive definiteness [1].

Second, the parameter C_D gives us control over the spectral properties of the problem. For a symmetric positive definite matrix, the 2-norm condition number is the ratio of its largest to its smallest eigenvalue, $\kappa_2(D) = \lambda_{\max}(D)/\lambda_{\min}(D)$ [4]. Our construction thus yields $\kappa_2(D) \approx C_D$. The spectral properties of the operator are known to fundamentally govern the convergence rate of the Lanczos process [1]. By varying C_D, we can therefore generate both well-conditioned and ill-conditioned problem instances. Using a uniform random distribution for the entries of D ensures that our test instances are generic and not biased toward an artificially simple spectral structure.

To permit the exact computation of solution error, we adopt a known-solution methodology. We first define a ground-truth solution vector x_{true} and then construct the right-hand side vector as $b := Ax_{\text{true}}$.

To assess the robustness of the approach, we conduct experiments using two functions, f, chosen for their distinct analytical properties. The first is the matrix exponential, $f(z) = \exp(z)$. As

an entire function, it is a canonical and well-behaved test case for algorithms computing matrix functions [2]. The Lanczos process itself is independent of f. Using a smooth function like the exponential therefore allows a clear analysis of the algorithm's intrinsic properties, such as the stability of the basis re-generation, without confounding factors from singularities in f.

The second function is the inverse, $f(z) = z^{-1}$, which recasts our problem as the solution of the linear system $Ax = b$. We select this function to evaluate the algorithm in the presence of a singularity at the origin. The convergence of Krylov methods for this problem is highly sensitive to the eigenvalues of A near zero, making it an essential and challenging test case for numerical robustness, particularly for the indefinite KKT systems we employ [4].

4.3 Experiment 1: Memory and Computation Trade-off

The first experiment was designed to empirically validate the theoretical complexity analysis of the two-pass method. The objective was to demonstrate that it maintains a constant memory footprint with respect to the iteration count, unlike the linear growth of the one-pass method, at the cost of a doubled computational workload. The experiment was conducted by selecting a large, fixed-size matrix $\mathbf{A}$ and executing both algorithms on the same problem instance, varying the number of iterations k across a predefined range. For each value of k, we recorded the peak RSS and the total wall-clock time. We expected that a plot of peak RSS versus k would exhibit a linear profile for the one-pass algorithm and a constant profile for the two-pass algorithm, consistent with their respective $O(nk)$ and $O(n)$ memory complexities. Furthermore, we anticipated that the wall-clock time for the two-pass method would be approximately double that of the one-pass method. However, we will see that the actual timing far deviates from this expectation due to memory access patterns and cache effects.

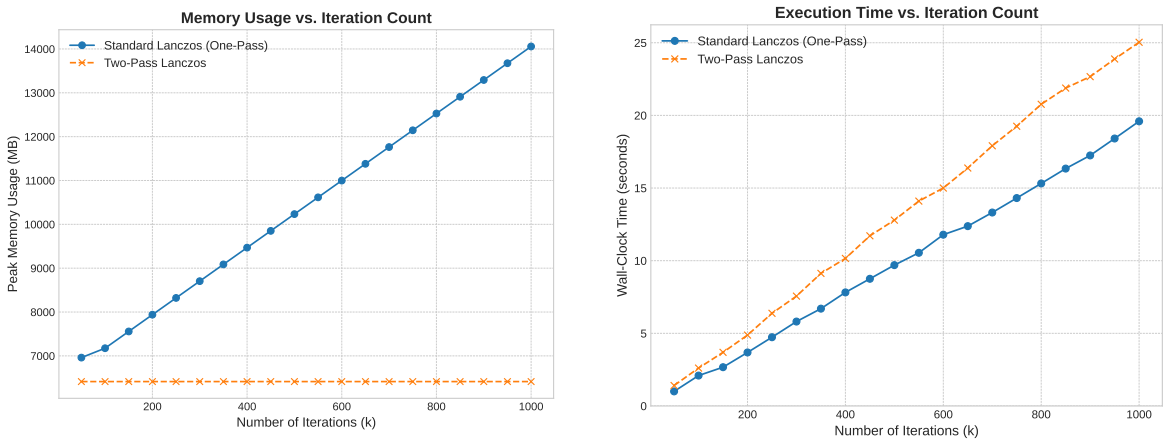
4.3.1 Results

We executed both algorithms on test instances of small, medium, and large dimensions. For each instance, we varied the number of iterations, k, and recorded the Peak Resident Set Size and the total wall-clock time.

The memory consumption results, presented in the left panels of Figures 1, 2, and 3, confirm the complexity analysis across all problem scales. The standard one-pass algorithm exhibits a linear growth in memory usage with respect to the iteration count k. This behavior stems directly from the need to store the entire basis matrix $\mathbf{V}_k \in \mathbb{C}^{n\times k}$, a requirement with a cost of $O(nk)$ [1]. In contrast, the two-pass algorithm maintains a constant memory footprint, independent of k. This aligns with its theoretical $O(n)$ memory complexity, as the algorithm only requires simultaneous storage for a small, constant number of n-dimensional vectors.

The analysis of wall-clock time, however, reveals a more complex behavior than the simplified theoretical model suggests. For the large-scale instance with 500k arcs, shown in Figure 1 (right panel), the wall-clock time of the two-pass method is only marginally greater than that of the one-pass method. Their execution time ratio remains close to unity, approximately 1.1. This observa-

tion appears to contradict the theoretical expectation that the two-pass method should perform nearly twice the work [2]. The discrepancy arises because for very large n, the dominant computational cost shifts from the sparse matrix-vector products to the dense vector operations for solution reconstruction. Both the final matrix-vector product $\mathbf{x}_k = \mathbf{V}_k \mathbf{y}_k$ in the one-pass method and the incremental accumulation in the two-pass method have a complexity of $O(nk)$. When the matrix $\mathbf{V}_k$ is too large to fit in any CPU cache, these operations become fundamentally limited by the system's main memory bandwidth. This large, common cost term dominates the total execution time for both methods and masks the effect of the doubled sparse matrix-vector products.

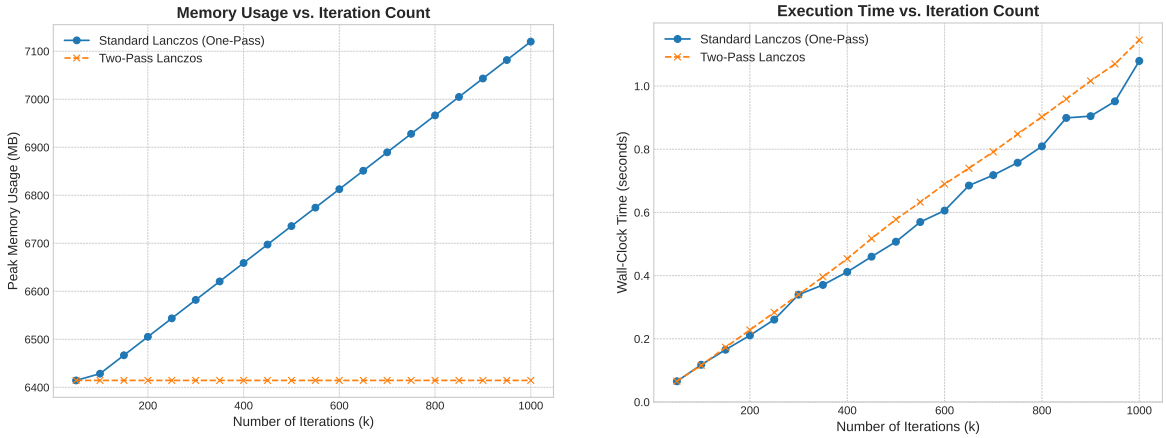


(a) Peak Memory Usage

(b) Wall-Clock Time

Figure 1: Performance results for a large-scale problem instance (500k arcs). The left panel shows peak memory usage, while the right panel shows wall-clock time.

As we reduce the problem dimension, this dynamic changes. For the medium-scale instance with 50k arcs, the results in Figure 2 (right panel) show that the two-pass method is consistently faster than the standard one-pass method. This performance reversal comes from the memory access patterns of the algorithms. The standard algorithm's reconstruction step accesses a large contiguous block of memory for $\mathbf{V}_k$, generating an inefficient access pattern with poor data locality that leads to a high rate of cache misses. The two-pass algorithm, by contrast, operates on a small working set of vectors at each step. The processor's cache hierarchy manages this small working set effectively, resulting in a high cache hit rate and superior performance.

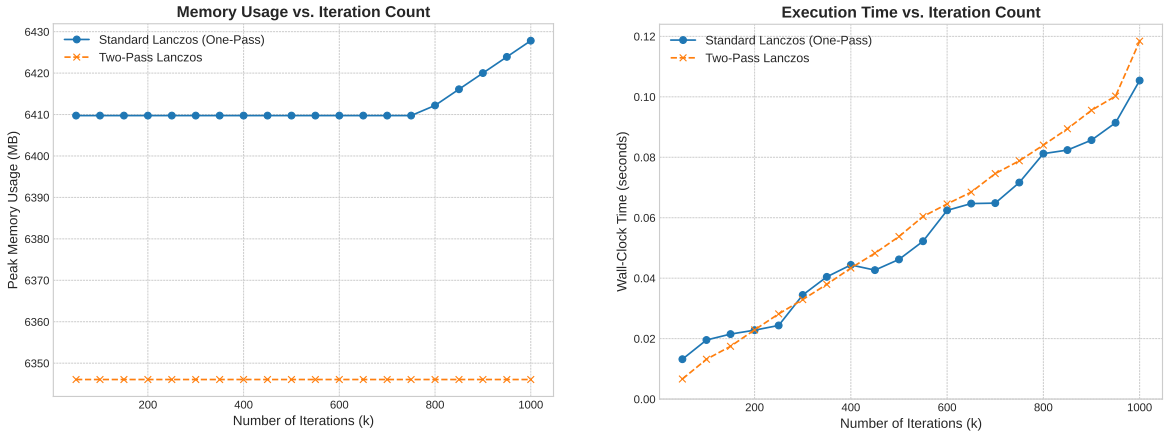


(a) Peak Memory Usage

(b) Wall-Clock Time

Figure 2: Performance results for a medium-scale problem instance (50k arcs). The left panel shows peak memory usage, while the right panel shows wall-clock time.

This effect becomes even more pronounced for the small-scale instance with 5k arcs, shown in Figure 3 (right panel). Here, the working set of the two-pass method can reside almost entirely within the fastest L1 and L2 caches, maximizing its performance advantage. In this cache-friendly regime, the performance gain from eliminating cache misses far outweighs the cost of the additional matrix-vector products. These results demonstrate that the algorithmic structure of the two-pass method, designed for memory efficiency, can yield an unexpected but significant advantage in computational speed by aligning more effectively with the memory architecture of modern processors.



(a) Peak Memory Usage

(b) Wall-Clock Time

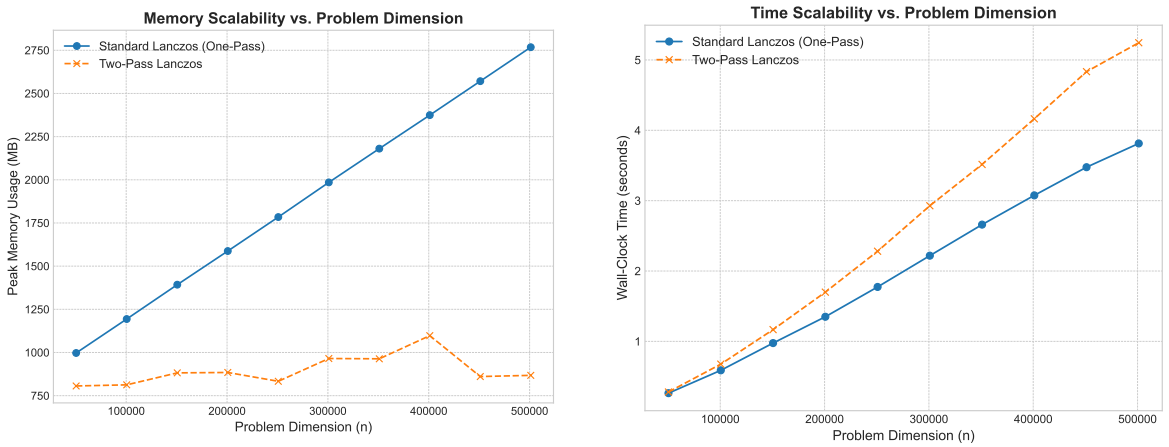
Figure 3: Performance results for a small-scale problem instance (5k arcs). The left panel shows peak memory usage, while the right panel shows wall-clock time.

4.4 Experiment 2: Scalability

The second experiment assesses how the resource requirements of both algorithms scale with the problem dimension n. For this test, we fix the number of iterations k to a constant value, and generate a family of test matrices of increasing dimension n. The network topologies are constructed to maintain a constant ratio of arcs to nodes, ensuring comparable sparsity across all instances. For each problem size, we execute both algorithms and record their peak RSS and total wall-clock time. We expect the results to show that the memory advantage of the two-pass algorithm becomes increasingly significant as n grows. Specifically, a plot of peak RSS versus n should show a substantially steeper slope for the one-pass method than for the two-pass method.

4.4.1 Results

We fixed the number of iterations at $k = 500$ and analyzed the performance of both algorithms as a function of the problem dimension, n. Figure 4 plots the peak memory usage and wall-clock time against n.



(a) Peak Memory Usage

(b) Wall-Clock Time

Figure 4: Performance and memory scalability with respect to problem dimension n for a fixed number of iterations ($k = 500$).

The memory consumption profiles in Figure 4 (a) show a stark contrast between the two methods. To formalize this analysis, we model the total peak memory usage, $M(n,k)$, as a function of the problem dimension n and the number of iterations k. This total memory can be decomposed into a base cost required for problem representation and an additional cost specific to the algorithm's execution:

$$M(n,k) = M_{\text{base}}(n) + M_{\text{alg}}(n,k). \tag{10}$$

The base component, $M_{\text{base}}(n)$, is common to both algorithms. It includes the memory for storing the sparse KKT matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ and a constant number of work vectors, c_v, each of dimension n. Let $\text{nnz}(\mathbf{A})$ be the number of non-zero entries in $\mathbf{A}$, and let s_d be the size of a double-precision

float. The storage for $\mathbf{A}$ in a Compressed Sparse Column (CSC) format requires approximately $s_d(\text{nnz}(\mathbf{A}) + n) + s_i(\text{nnz}(\mathbf{A}))$, where s_i is the size of an integer index. For our problem instances, $\text{nnz}(\mathbf{A}) \propto n$. Therefore, the base memory cost is strictly linear in n:

$$M_{\text{base}}(n) = (c_A + c_v s_d)n + O(1), \tag{11}$$

where c_A is a constant related to the storage of the sparse matrix.

The substantial difference between the methods lies in the algorithm-specific term, $M_{\text{alg}}(n,k)$. The two-pass algorithm is designed to minimize this term by storing only the scalar coefficients of $\mathbf{T}_k$, resulting in a negligible cost of $M_{\text{alg, TP}}(k) = O(k)$. In contrast, the standard one-pass algorithm must store the entire basis matrix $\mathbf{V}_k \in \mathbb{R}^{n\times k}$, which imposes a significant memory overhead:

$$M_{\text{alg, std}}(n,k) = n \cdot k \cdot s_d. \tag{12}$$

With k fixed, the asymptotic memory complexities for the two methods are therefore:

$$M_{\text{TP}}(n) = M_{\text{base}}(n) + O(k) \approx c_{\text{base}} \cdot n \tag{13}$$

$$M_{\text{std}}(n,k) = M_{\text{base}}(n) + (k \cdot s_d) \cdot n = (c_{\text{base}} + k \cdot s_d) \cdot n. \tag{14}$$

This model predicts that the memory usage of both methods should grow linearly with n, but the slope of the growth for the standard method should exceed that of the two-pass method by a constant factor of $k \cdot s_d$.

The empirical data presented in Table 1 aligns with our model. The memory usage of the two-pass method provides a stable empirical baseline for $M_{\text{base}}(n)$. The last column, representing the term $M_{\text{alg, std}}(n,k)$, isolates the cost of storing the basis matrix $\mathbf{V}_k$. A linear regression performed on this column yields an empirical growth rate of approximately 3968 bytes per unit increase in n. This is in line with the theoretical rate of $k \cdot s_d = 500 \cdot 8 = 4000$ bytes per unit n.

Dimension (n)	**Standard (MB)**	**Two-Pass (MB)**	**Difference (MB)**
50,365	997.1	806.1	191.0
100,516	1193.8	812.6	381.2
150,632	1392.6	882.0	510.6
200,730	1587.5	884.2	703.3
250,816	1784.3	833.2	951.1
300,894	1985.0	965.1	1019.9
350,966	2180.4	963.4	1217.0
401,033	2374.5	1097.5	1277.0
451,095	2571.2	860.7	1710.5
501,155	2767.9	867.6	1900.3

Table 1: Peak memory usage (RSS) in megabytes for a fixed number of iterations ($k = 500$). The difference isolates the memory cost of the basis matrix $\mathbf{V}_k$ in the standard algorithm.

The wall-clock time for both algorithms, shown in Figure 4 (b), scales linearly with the problem dimension n. This is consistent with the theoretical cost, which is dominated by k iterations of operations (sparse matrix-vector products and vector additions) whose complexity is linear in n. With k held constants, the total time complexity is $O(nk)$, resulting in the observed linear scaling.

4.5 Experiment 3: Numerical Stability

Our final experiment investigates the numerical stability of the two-pass approach. We quantify the numerical error that the basis re-generation introduces by selecting a problem of moderate dimension, for which we can explicitly store both basis matrices. Let $\mathbf{V}_k = [\mathbf{v}_1, \ldots, \mathbf{v}_k]$ be the basis from the standard one-pass algorithm, and let $\mathbf{V}'_k = [\mathbf{v}'_1, \ldots, \mathbf{v}'_k]$ be the basis re-generated during the second pass from the stored coefficients $\{\alpha_j, \beta_j\}$. As a function of the iteration count k, our analysis tracks three principal metrics. First, we compute the loss of orthogonality for both bases via the quantities $\|\mathbf{I}_k - \mathbf{V}_k^H \mathbf{V}_k\|_F$ and $\|\mathbf{I}_k - (\mathbf{V}'_k)^H \mathbf{V}'_k\|_F$. Second, we quantify the numerical drift between the two bases by computing $\|\mathbf{V}_k - \mathbf{V}'_k\|_F$. Third, we evaluate the impact on the final solution by computing the norm of the difference $\|\mathbf{x}_k - \mathbf{x}'_k\|_2$, where $\mathbf{x}_k = \mathbf{V}_k \mathbf{y}_k$ and $\mathbf{x}'_k = \mathbf{V}'_k \mathbf{y}_k$.

We hypothesized that both bases would exhibit a similar rate of orthogonality loss. We expected the basis deviation $\|\mathbf{V}_k - \mathbf{V}'_k\|_F$ to be small but non-zero, growing with k as floating-point errors accumulate. Critically, we expected the final solution deviation $\|\mathbf{x}_k - \mathbf{x}'_k\|_2$ to remain close to machine precision. Such a result would demonstrate that the numerical drift in the re-generated basis does not catastrophically degrade the accuracy of the final solution.

4.5.1 Results

TODO

References

[1] Gene H Golub and Charles F Van Loan. *Matrix computations*. JHU press, 2013.

[2] Andreas Frommer and Valeria Simoncini. Matrix functions. In *Model order reduction: theory, research aspects and applications*, pages 275–303. Springer, 2008.

[3] Cornelius Lanczos. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *Journal of research of the National Bureau of Standards*, 45(4):255–282, 1950.

[4] Yousef Saad. *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics, 2 edition, 2003.

[5] Angelo A Casulli and Igor Simunec. A low-memory lanczos method with rational krylov compression for matrix functions. *SIAM Journal on Scientific Computing*, 47(3):A1358–A1382, 2025.

[6] Artan Boriçi. Fast methods for computing the neuberger operator. In *Numerical Challenges in Lattice Quantum Chromodynamics: Joint Interdisciplinary Workshop of John von Neumann Institute for Computing, Jülich, and Institute of Applied Computer Science, Wuppertal University, August 1999*, pages 40–47. Springer, 2000.

[7] Nicholas J. Higham. *Functions of Matrices: Theory and Computation*. Society for Industrial and Applied Mathematics, 2008.