A Memory-Efficient Two-Pass Lanczos Algorithm

Luca Lombardo

Abstract

The two-pass Lanczos algorithm presents a memory-efficient alternative to the standard one-pass method for computing the action of a matrix function, $f(\mathbf{A})\mathbf{b}$. We analyze its theoretical complexities, confirming a memory reduction from $O(nk)$ to $O(n)$ in exchange for doubling the number of matrix-vector products. Numerical experiments reveal two key insights. First, despite the increased computational work, the two-pass method's wall-clock performance is often superior in practice due to its more favorable memory access patterns in cache-bound scenarios. Second, the algorithm maintains numerical accuracy equivalent to the one-pass approach, as the basis regeneration process does not introduce additional error, even for challenging ill-conditioned systems. These results establish the two-pass algorithm as a robust and often preferable practical alternative for large-scale computations.

Contents

1 Introduction **2**
1.1 The Problem of Computing Matrix Functions 2
1.2 Krylov Subspace Methods: The Standard Approach 3
1.3 The Memory Bottleneck of the Standard Lanczos Process 5

2 Symmetric Lanczos Process **5**

3 The Two-Pass Lanczos Algorithm **6**
3.1 Algorithmic Formulation 7
3.1.1 First Pass: Projection and Coefficient Generation 7
3.1.2 Second Pass: Solution Reconstruction 7
3.2 Computational and Memory Complexity 8

4 Numerical Experiments **10**
4.1 Experimental Setup and Performance Metrics 10
4.2 Test Problem Generation 10
4.3 Memory and Computation Trade-off 11
4.3.1 Results 12
4.3.2 Dense Matrix 13
4.4 Scalability with Problem Dimension 15
4.4.1 Results 15
4.5 Numerical Accuracy and Stability Analysis 17
4.5.1 Exponential on a Well-Conditioned Spectrum 17

4.5.2 Inverse on a Well-Conditioned Spectrum 18
4.5.3 Exponential on an Ill-Conditioned Spectrum 19
4.5.4 Inverse on an Ill-Conditioned Spectrum 19

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

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

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 a set Ω containing the spectrum of $\mathbf{A}$, $\sigma(\mathbf{A})$. The most direct definition of a matrix function arises when the function f is a polynomial. For a scalar polynomial $p(z) = \sum_{j=0}^{m} c_j z^j$, the corresponding matrix function $p(\mathbf{A})$ is defined by substituting the matrix $\mathbf{A}$ for the scalar variable z:

$$p(\mathbf{A}) = c_0\mathbf{I} + c_1\mathbf{A} + \cdots + c_m\mathbf{A}^m.$$

This elementary case provides the foundation for defining $f(\mathbf{A})$ for more general functions. A key result from the theory of matrix functions is that for any function f defined on the spectrum of $\mathbf{A}$, the matrix $f(\mathbf{A})$ is equivalent to a polynomial in $\mathbf{A}$.

Proposition 1.1. *Let f be a function defined on the spectrum of $\mathbf{A} \in \mathbb{C}^{n\times n}$. Then $f(\mathbf{A}) = p(\mathbf{A})$, where p is the unique polynomial of minimal degree that interpolates f and its derivatives on the spectrum of $\mathbf{A}$ in the Hermite sense.*

$$\frac{d^j p}{dz^j}(\lambda_i) = \frac{d^j f}{dz^j}(\lambda_i), \quad j = 0, \ldots, m_i - 1,$$

for every distinct eigenvalue $\lambda_i \in \sigma(\mathbf{A})$, where m_i is the size of the largest Jordan block associated with λ_i. The matrix function is then defined as $f(\mathbf{A}) := p(\mathbf{A})$.

The polynomial equivalence provides the foundation for the methods we consider. The computation of the matrix $f(\mathbf{A})$ itself, however, presents significant challenges. A direct approach might involve the spectral decomposition of the matrix, $\mathbf{A} = \mathbf{V}\Lambda\mathbf{V}^{-1}$, from which $f(\mathbf{A}) = \mathbf{V}f(\Lambda)\mathbf{V}^{-1}$. This method is only numerically viable if the matrix of eigenvectors $\mathbf{V}$ is well-conditioned. For general non-Hermitian matrices, or even for Hermitian matrices with clustered eigenvalues, this approach can suffer from a severe loss of accuracy.

A more stable method for computing $f(\mathbf{A})$ for small to medium-sized matrices is the Schur-Parlett algorithm. The algorithm begins by computing the Schur form of the matrix, $\mathbf{A} = \mathbf{QTQ}^*$, where $\mathbf{Q}$ is unitary and $\mathbf{T}$ is upper triangular. The problem is then reduced to computing $f(\mathbf{T})$, after which the full matrix is recovered as $f(\mathbf{A}) = \mathbf{Q}f(\mathbf{T})\mathbf{Q}^*$. The asymptotic cost of this procedure is $O(n^3 + c_f n)$, where c_f is the cost of evaluating the scalar function f. The cubic term arises from the initial Schur decomposition, rendering this approach impractical for the large-scale matrices.

A further impediment is that even when $\mathbf{A}$ is sparse, the resulting matrix $f(\mathbf{A})$ is generally dense. Storing this matrix would require $O(n^2)$ memory, which is often prohibitive. For these reasons, our focus is on methods that compute the action $\mathbf{x} = f(\mathbf{A})\mathbf{b}$ directly, without forming $f(\mathbf{A})$. This problem is an evergreen 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 polynomial representation $f(\mathbf{A}) = p(\mathbf{A})$ seen in 1.1 suggests that the solution vector $\mathbf{x} = f(\mathbf{A})\mathbf{b}$ can be constructed from a sequence of vectors generated by successive applications of the matrix $\mathbf{A}$ to the starting vector $\mathbf{b}$. This observation provides a direct path to the use of Krylov subspace methods.

Definition 1.2 (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

$$\mathscr{K}_k(\mathbf{A}, \mathbf{b}) = \text{span}\{\mathbf{b}, \mathbf{Ab}, \ldots, \mathbf{A}^{k-1}\mathbf{b}\}.$$

This space is equivalently characterized as the set of all vectors of the form $q(\mathbf{A})\mathbf{b}$, where q is a polynomial of degree less than k [5].

The standard method for computing an approximation to $\mathbf{x}$ from within $\mathscr{K}_k(\mathbf{A}, \mathbf{b})$ is the Arnoldi method, an iterative process that constructs an orthonormal basis for the subspace. After k steps, this process generates a set of orthonormal vectors $\{\mathbf{v}_1, \ldots, \mathbf{v}_k\}$ that form the columns of a matrix $\mathbf{V}_k \in \mathbb{C}^{n\times k}$. The underlying recurrence relation of the method can be summarized in the Arnoldi decomposition

$$\mathbf{AV}_k = \mathbf{V}_k\mathbf{H}_k + h_{k+1,k}\mathbf{v}_{k+1}\mathbf{e}_k^T, \tag{1}$$

where $\mathbf{H}_k = \mathbf{V}_k^H \mathbf{A} \mathbf{V}_k$ is the $k \times k$ orthogonal projection of $\mathbf{A}$ onto the subspace $\mathcal{K}_k(\mathbf{A}, \mathbf{b})$. The structure of the Arnoldi process, which orthogonalizes each new vector only against the preceding ones, ensures that $\mathbf{H}_k$ is an upper Hessenberg matrix [5].

The approximation for $f(\mathbf{A})\mathbf{b}$ is constructed as an extension of the Full Orthogonalization Method (FOM) for linear systems. The original problem is replaced by its projection onto the subspace, and the solution to this smaller problem is then lifted back to the original space. Assuming the starting vector is normalized as $\mathbf{v}_1 = \mathbf{b}/\|\mathbf{b}\|_2$, this yields the approximation

$$\mathbf{x}_k = \|\mathbf{b}\|_2 \mathbf{V}_k f(\mathbf{H}_k) \mathbf{e}_1. \tag{2}$$

This approach reduces the high-dimensional problem to the computation of the function of the small $k \times k$ matrix $\mathbf{H}_k$. This subproblem is computationally tractable, typically solved using methods like the Schur-Parlett algorithm at a cost of $O(k^3)$, which is negligible for $k \ll n$.

For the case of a Hermitian matrix $\mathbf{A}$, the Arnoldi process simplifies considerably. The projected matrix $\mathbf{H}_k = \mathbf{V}_k^H \mathbf{A} \mathbf{V}_k$ must also be Hermitian:

$$(\mathbf{H}_k)^H = (\mathbf{V}_k^H \mathbf{A} \mathbf{V}_k)^H = \mathbf{V}_k^H \mathbf{A}^H \mathbf{V}_k = \mathbf{V}_k^H \mathbf{A} \mathbf{V}_k = \mathbf{H}_k.$$

A matrix that is simultaneously upper Hessenberg and Hermitian must necessarily be a real, symmetric tridiagonal matrix [5]. This structural simplification allows the long recurrence of the Arnoldi method to be replaced by a much more efficient three-term recurrence. This specialized algorithm is the symmetric Lanczos process.

The projected matrix is thus a real, symmetric tridiagonal matrix, commonly denoted $\mathbf{T}_k$, and the final approximation takes the form

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

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

The quality of this approximation is rigorously related to the error of the best uniform polynomial approximant of f on the spectrum of $\mathbf{A}$. This relationship is formalized in the following theorem.

Theorem 1.3. *Let $\mathbf{A} \in \mathbb{C}^{n \times n}$ be a Hermitian matrix with spectrum $\sigma(\mathbf{A})$, and let $\mathbf{x}_k$ be the approximation to $\mathbf{x} = f(\mathbf{A})\mathbf{b}$ obtained after k steps of the Lanczos algorithm. Then the error is bounded by*

$$\|\mathbf{x} - \mathbf{x}_k\|_2 \leq 2\|\mathbf{b}\|_2 \min_{p \in \mathscr{P}_{k-1}} \max_{z \in \sigma(\mathbf{A})} |f(z) - p(z)|,$$

where $\mathscr{P}_{k-1}$ is the set of polynomials of degree at most $k-1$ [5].

This bound demonstrates that the convergence of the Lanczos approximation depends directly on how well the function f can be approximated by a low-degree polynomial over the spectral interval of $\mathbf{A}$.

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[1] 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 we need memory-efficient strategies. In section 3 we will discuss the two-pass Lanczos method, 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 must first formalize the standard symmetric Lanczos process. As seen in section 1.2, the structure of this process is a direct consequence of applying the Arnoldi method to a Hermitian matrix $\mathbf{A}$. We begin from the Arnoldi decomposition, given in equation (1). We have seen that for a Hermitian operator, this relation simplifies to the compact matrix equation

$$\mathbf{A}\mathbf{V}_k = \mathbf{V}_k\mathbf{T}_k + \beta_k \mathbf{v}_{k+1}\mathbf{e}_k^T, \tag{3}$$

where $\mathbf{V}_k = [\mathbf{v}_1, \ldots, \mathbf{v}_k] \in \mathbb{C}^{n\times k}$ is the matrix of orthonormal Lanczos vectors, $\mathbf{e}_k$ is the k-th canonical basis vector, and $\mathbf{T}_k$ is the $k \times k$ real symmetric tridiagonal matrix

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

This matrix equation contains the set of vector relations that define the algorithm. By equating the j-th column of both sides of (3) for $j = 1, \ldots, k$, we can extract the underlying vector recurrence. The j-th column of the left-hand side is $\mathbf{A}\mathbf{v}_j$. The j-th column of the term $\mathbf{V}_k\mathbf{T}_k$ is given by the linear combination of the columns of $\mathbf{V}_k$ weighted by the entries of the j-th column of $\mathbf{T}_k$:

$$(\mathbf{V}_k\mathbf{T}_k)_j = \beta_{j-1}\mathbf{v}_{j-1} + \alpha_j\mathbf{v}_j + \beta_j\mathbf{v}_{j+1},$$

[1] particularly those arising from the discretization of partial differential equations

where we have defined the conventions $\beta_0 = 0$ and $\mathbf{v}_0 = \mathbf{0}$. For $j < k$, the term $\beta_k \mathbf{v}_{k+1} \mathbf{e}_k^T$ does not contribute to the j-th column. Equating the columns thus yields:

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

Rearranging this expression gives us the three-term recurrence relation of the Lanczos process:

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

The real scalar coefficients are determined at each step by enforcing the orthonormality of the sequence $\{\mathbf{v}_j\}$. From the recurrence (4), taking the inner product of both sides with $\mathbf{v}_j$ and using the orthogonality conditions $\mathbf{v}_j^H \mathbf{v}_{j+1} = 0$ and $\mathbf{v}_j^H \mathbf{v}_{j-1} = 0$ isolates α_j:

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

The off-diagonal elements, β_j, are determined from the normalization condition $\|\mathbf{v}_{j+1}\|_2 = 1$, which requires that β_j be the norm of the unnormalized residual vector:

$$\beta_j = \|\mathbf{A}\mathbf{v}_j - \alpha_j \mathbf{v}_j - \beta_{j-1}\mathbf{v}_{j-1}\|_2.$$

If at some step $j < n$, $\beta_j = 0$, the algorithm terminates. This occurs if the subspace $\mathcal{K}_j(\mathbf{A}, \mathbf{b})$ is invariant under $\mathbf{A}$. Otherwise, for $j < n$, $\beta_j > 0$.

Finally, the matrix form (3) also confirms that $\mathbf{T}_k$ is the orthogonal projection of $\mathbf{A}$ onto the Krylov subspace. By left-multiplying by $\mathbf{V}_k^H$ and using the orthonormality conditions $\mathbf{V}_k^H \mathbf{V}_k = \mathbf{I}_k$ and $\mathbf{V}_k^H \mathbf{v}_{k+1} = \mathbf{0}$, we recover

$$\mathbf{V}_k^H \mathbf{A}\mathbf{V}_k = \mathbf{V}_k^H (\mathbf{V}_k \mathbf{T}_k + \beta_k \mathbf{v}_{k+1} \mathbf{e}_k^T) = \mathbf{I}_k \mathbf{T}_k + \mathbf{0} = \mathbf{T}_k.$$

This set of relations forms the computational basis for both the standard and the two-pass Lanczos algorithms.

3 The Two-Pass Lanczos Algorithm

Having established the theoretical framework for the standard Lanczos process (section 2) and identified its principal limitation[2] we can now introduce a memory-efficient variant known as the two-pass Lanczos algorithm [2]. 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 $\mathbf{x}_k$.

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.

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

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 symmetric Lanczos process, as detailed in Section 2

The algorithm iteratively generates the Lanczos coefficients $\{\alpha_j\}_{j=1}^{k}$ and $\{\beta_j\}_{j=1}^{k-1}$ using the three-term recurrence relation previously derived in equation (4). The crucial distinction in this pass is the memory management strategy: while the recurrence requires the two most recent Lanczos vectors ($\mathbf{v}_j$ and $\mathbf{v}_{j-1}$) to compute the next, the full basis $\{\mathbf{v}_j\}_{j=1}^{k}$ is not stored. Only a constant number of vectors are kept in memory at any given time.

After k iterations, the sequences of scalars are used to construct the matrix $\mathbf{T}_k$. The final operation of this 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 with respect to the (discarded) 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), rearranged to isolate $\mathbf{v}_{j+1}$:

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

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:

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

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 a formal analysis of the memory requirements and computational cost of the two-pass Lanczos algorithm. 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:

$$M(n,k) = M_{\text{base}}(n) + M_{\text{alg}}(n,k). \tag{8}$$

The base component, $M_{\text{base}}(n)$, is common to both algorithms. It includes the memory for storing the sparse matrix $\mathbf{A}$ and a constant number of work vectors. The term $M_{\text{alg}}(n,k)$ represents the additional memory specific to each algorithm's execution strategy, which is the source of the substantial difference between the two methods.

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 has a memory complexity of $O(nk)$, while the two-pass variant reduces this requirement to $O(n)$.*

Proof. The base memory component, $M_{\text{base}}(n)$, includes storage for the sparse 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 the class of sparse problems considered, $\text{nnz}(\mathbf{A}) \propto n$. The base memory cost is therefore strictly linear in n, so $M_{\text{base}}(n) = O(n)$.

The difference between the methods lies in the algorithm-specific term, $M_{\text{alg}}(n,k)$. The standard one-pass algorithm must store the entire basis matrix $\mathbf{V}_k \in \mathbb{R}^{n\times k}$ to synthesize the final solution. This imposes a significant memory overhead:

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

The total memory is therefore $M_{\text{std}}(n,k) = M_{\text{base}}(n) + M_{\text{alg, std}}(n,k) = O(n) + O(nk) = O(nk)$.

The two-pass algorithm is designed to minimize this term. It stores only the scalar coefficients of $\mathbf{T}_k$, resulting in a negligible cost of $M_{\text{alg, TP}}(k) = O(k)$. The total memory requirement is thus:

$$M_{\text{TP}}(n,k) = M_{\text{base}}(n) + M_{\text{alg, TP}}(k) = O(n) + O(k)$$

Under the common assumption that $n \gg k$, this simplifies to $O(n)$. □

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, whereas the two-pass algorithm requires $2k$.*

Proof. For large, sparse matrices, the matrix-vector product $\mathbf{A}\mathbf{v}_j$ is the most computationally expensive operation per iteration [4]. The standard one-pass method performs one such product at each of its k iterations, for a total of k products.

The two-pass method executes two distinct 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 the re-computation of the same sequence of k matrix-vector products. The total count is therefore $2k$. □

4 Numerical Experiments

In this section, we present a series of numerical experiments designed to validate the theoretical analysis presented in 3.2 and to assess the practical performance of the two-pass Lanczos algorithm. Our investigation is structured into three distinct experiments. The first validates the memory-versus-computation trade-off as a function of iteration count. The second assesses the scalability of both algorithms with respect to the problem dimension. The third provides an analysis of numerical stability and accuracy across several challenging problem scenarios.

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[3]. The algorithms were implemented in Rust and compiled with `rustc` version 1.90.0 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 memory 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 $\mathbf{x}_k$ by its relative error with respect to a known ground-truth solution $\mathbf{x}_{\text{true}}$. We assess numerical stability through two primary metrics. First, we quantify the loss of orthogonality of the Lanczos basis $\mathbf{V}_k$ via the Frobenius norm $\|I - \mathbf{V}_k^H \mathbf{V}_k\|_F$, a known phenomenon in finite-precision implementations of the process [1]. Second, to confirm that the two-pass regeneration is stable, we measure the numerical equivalence of the final solutions by computing the L2-norm of their deviation, $\|\mathbf{x}_k - \mathbf{x}'_k\|_2$.

4.2 Test Problem Generation

For the performance and scalability experiments, 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

[3] however, for this experiments, all parallelism has been disabled

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 provides two key advantages for creating robust test cases. First, by setting $d_i \geq 1$, we ensure that D is a symmetric positive definite matrix. This choice guarantees that the resulting KKT matrix A is indefinite, which can be verified by examining the quadratic form $x^T Ax$. Operating on indefinite matrices provides a robust test for the Lanczos algorithm, as the process does not require positive definiteness, a property that distinguishes it from methods such as the Conjugate Gradient algorithm [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 Krylov subspace methods [1]. By varying C_D, we can therefore generate both well-conditioned and ill-conditioned problem instances.

For these performance tests, we adopt a known-solution methodology. We first define a ground-truth solution vector $\mathbf{x}_{\text{true}}$ and then construct the right-hand side vector as $\mathbf{b} := \mathbf{A}\mathbf{x}_{\text{true}}$.

While the KKT systems provide realistic, large-scale test cases for performance, a different approach is necessary for the numerical stability and accuracy analysis, which requires a precisely known analytical ground-truth solution for $f(\mathbf{A})\mathbf{b}$. For that experiment, we employ synthetic diagonal matrices, where the action of $f(\mathbf{A})$ can be computed to machine precision. This methodology is detailed further in Section 4.5.

4.3 Memory and Computation Trade-off

The first experiment was designed to validate the theoretical complexity analysis presented in Section 3.2. The goal was to demonstrate the trade-off between memory consumption and computational workload by comparing the one-pass and two-pass Lanczos algorithms.

Based on our model, we formulate two distinct hypotheses. First, concerning memory usage, we expect the peak resident set size of the standard algorithm to exhibit linear growth with respect to the iteration count k, consistent with its $O(nk)$ complexity derived from storing the basis $\mathbf{V}_k$. On the other hand, we expect the two-pass method to maintain a nearly constant memory footprint, consistent with its $O(n)$ complexity. Second, regarding computational time, the model based on floating-point operations predicts that the wall-clock time for the two-pass method should be approximately double that of the one-pass method, due to the doubled number of matrix-vector products. However, this model neglects the cost of memory access. A more realistic hypothesis is that the actual performance will be governed by the relation between the additional arithmetic operations and the algorithms' memory access patterns. For problem sizes where the basis matrix $\mathbf{V}_k$ is too large to fit in processor caches, the cost of memory bandwidth may dominate, reducing the observable time penalty for the two-pass method.

We conducted this experiment using several KKT system instances of fixed dimension n and vary-

ing sparsity. For each instance, we executed both algorithms while varying the number of iterations k across a predefined range, recording the peak RSS and the total wall-clock time for each run.

4.3.1 Results

The memory consumption results, presented in Figures 1, 3, and 5, provide unambiguous empirical support for the theoretical complexity analysis. The standard one-pass algorithm exhibits a clear 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 it only requires simultaneous storage for a small, constant number of n-dimensional vectors. The medium-scale case, shown in Figure 3, illustrates this dynamic most clearly: both methods start with an identical base memory cost, after which the standard method's requirement grows linearly while the two-pass method's remains flat.

The analysis of wall-clock time reveals a more interesting interaction between computational cost and memory architecture. We begin by examining the large-scale instance in Figure 2. The data shows that the two-pass method is consistently slower than the standard one-pass algorithm. However, the performance penalty is significantly less than the factor of two predicted by a simple model based on matrix-vector products. This discrepancy suggests that the cost of matrix-vector products is not the sole determinant of performance.

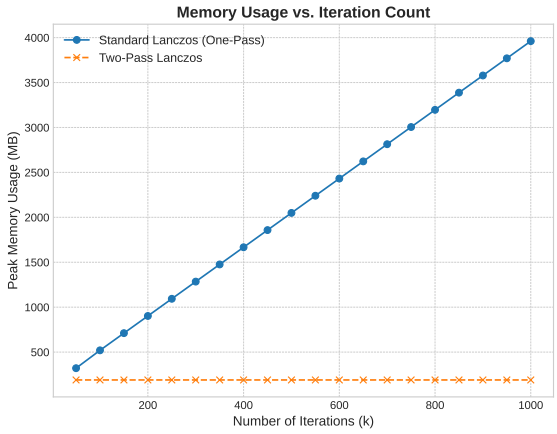

Figure 1: Peak memory usage for a large-scale problem (500k arcs).

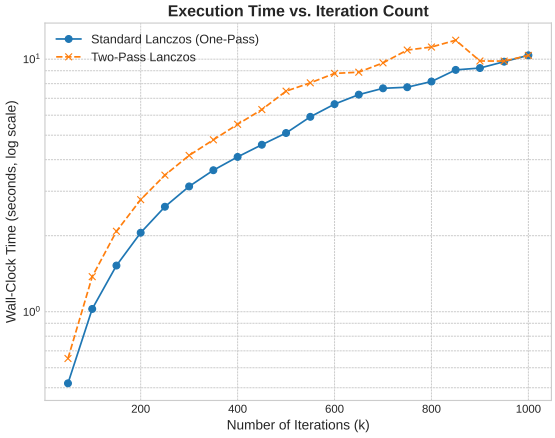

Figure 2: Wall-clock time for a large-scale problem (500k arcs).

We hypothesize that this behavior is governed by the memory access patterns during the solution reconstruction phase. The total execution time can be modeled as the sum of costs for matrix-vector products and vector operations. For the standard method, the final solution is computed via a single dense matrix-vector product, $\mathbf{x}_k = \mathbf{V}_k \mathbf{y}_k$. When n and k are large, the basis matrix $\mathbf{V}_k$ becomes too large to fit in any level of the CPU cache. Consequently, this operation becomes

memory-bandwidth-bound: its performance is limited by the speed at which data can be streamed from main memory, leading to high latency. In contrast, the two-pass method reconstructs the solution incrementally, operating on a small working set of vectors at each step ($\mathbf{v}_{\text{prev}}$, $\mathbf{v}_{\text{curr}}$, and $\mathbf{x}_k$). This approach exhibits superior data locality, allowing the processor's cache hierarchy to manage the data effectively and maintain a high cache-hit rate. The superior cache efficiency of the two-pass method's reconstruction phase thus mitigates a substantial portion of the cost of its doubled matrix-vector products. The standard method's apparent computational advantage is eroded by its poor memory access patterns..

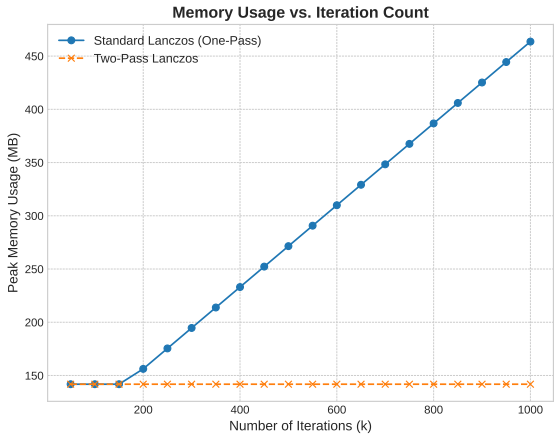

Figure 3: Peak memory usage for a medium-scale problem (50k arcs).

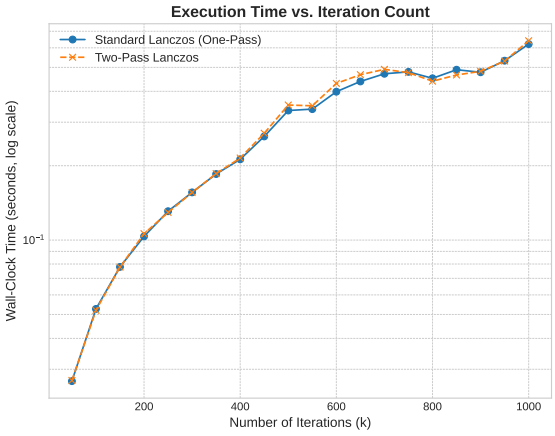

Figure 4: Wall-clock time for a medium-scale problem (50k arcs).

This hypothesis is further supported by the results from the medium and small scale experiments. For the medium-scale instance, shown in Figure 4, the execution times of both algorithms are nearly identical. This represents an equilibrium point. At this scale, the performance penalty incurred by the standard method due to increasing cache misses becomes significant enough to almost perfectly offset the additional computational cost of the doubled matrix-vector products in the two-pass method.

Finally, for the small-scale problem shown in Figure 6, the performance of both algorithms is again nearly indistinguishable. In this regime, the problem is so small that the entire working set for both algorithms, including the growing basis $\mathbf{V}_k$ of the standard method, fits comfortably within the CPU's cache. Memory access is no longer a bottleneck for either approach. Consequently, performance is governed by raw computational cost. Since the matrix-vector products for such a small 'n' are extremely cheap, the doubled cost for the two-pass method is negligible compared to other fixed overheads, leading to nearly identical wall-clock times.

4.3.2 Dense Matrix

To confirm our hypothesis that the surprising performance of the two-pass method on sparse matrices is due to its superior memory access patterns, we conducted a final experiment using a dense matrix. For a dense matrix, the computational cost of the matrix-vector product, which

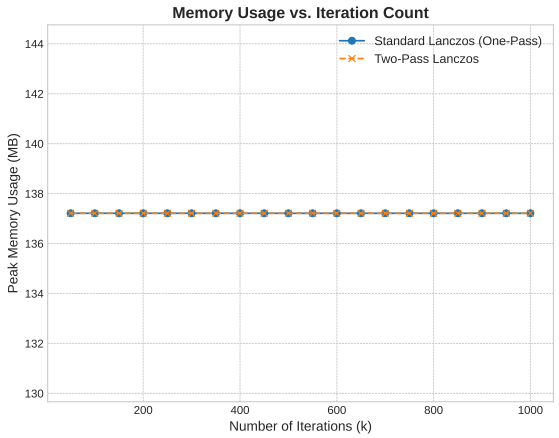

Figure 5: Peak memory usage for a small-scale problem (5k arcs).

Figure 6: Wall-clock time for a small-scale problem (5k arcs).

scales as $O(n^2)$, is expected to be the dominant factor in the total execution time. In this compute-bound regime, the benefits of cache efficiency from vector operations should become negligible. We therefore hypothesize that the performance of the algorithms will revert to the simple model based on operation count, with the wall-clock time of the two-pass method being approximately double that of the standard one-pass method.

For this test, we generated a dense, symmetric random matrix of dimension $n = 10000$ and measured the wall-clock time for both algorithms as a function of the iteration count, k.

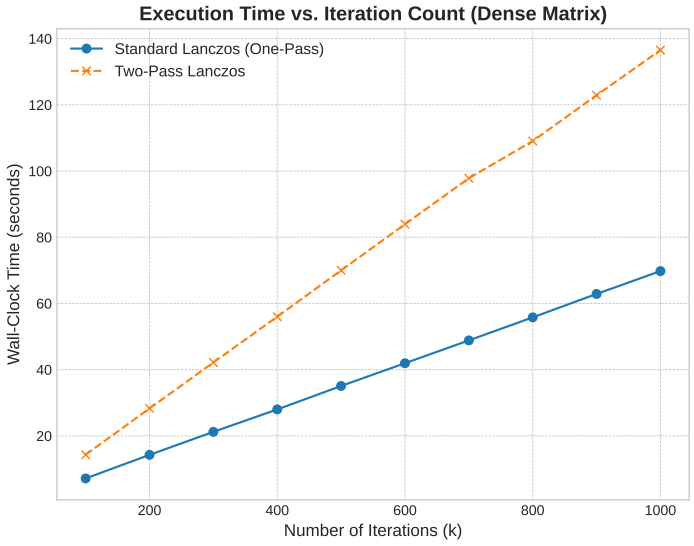

Figure 7: Wall-clock time vs. iteration count for a dense matrix ($n = 10000$). This time we don't need to use the log scale on the y-axis, as the times are much larger.

The results, presented in Figure 7, align perfectly with this hypothesis. The execution time for both

methods scales linearly with the number of iterations, but the slope for the two-pass algorithm is almost exactly double that of the standard method. For example, at $k = 1000$ iterations, the standard method takes approximately 70 seconds, while the two-pass method takes approximately 137 seconds.

This outcome provides conclusive evidence for our analysis. When the matrix-vector product is computationally dominant, the cost of the doubled products in the two-pass method dictates its performance, resulting in a nearly 2x slowdown. This confirms that the competitive, and sometimes superior, performance of the two-pass method on sparse matrices is indeed an artifact of its more favorable memory access patterns and superior cache efficiency, which mitigate the cost of its additional arithmetic operations.

4.4 Scalability with Problem Dimension

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. The network topologies are constructed to maintain a comparable sparsity across all instances. We hypothesize that the memory advantage of the two-pass algorithm will become increasingly significant as n grows. This should manifest as a substantially steeper slope in the memory usage plot for the one-pass method, reflecting its linear dependence on the product nk, compared to a gentler slope for the two-pass method, which depends only linearly on n.

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. Figures 8 and 9 plot the peak memory usage and wall-clock time against n.

The memory consumption profiles in Figure 8 show a stark contrast between the two methods, confirming our hypothesis. The memory usage of both methods grows linearly with n, but the slope of this growth differs significantly. The two-pass method exhibits a gentle slope, corresponding to the linear growth of the base memory requirement, $M_{\text{base}}(n)$. The standard method, in contrast, shows a much steeper slope, reflecting the combined cost of the base memory and the algorithm-specific term for storing the basis, $M_{\text{std}}(n,k) = M_{\text{base}}(n) + M_{\text{alg, std}}(n,k)$.

In Table 1, we provide the raw data underlying the memory plot. The memory usage of the two-pass method serves as a stable empirical baseline for $M_{\text{base}}(n)$. The final column, representing the difference in memory consumption, effectively isolates the cost of storing the basis matrix $\mathbf{V}_k$. A linear regression performed on this column's data yields a growth rate of approximately 3.81 MB per 1000 units of increase in n, which corresponds to 3998 bytes per unit n. This observed rate aligns remarkably well with the theoretical rate predicted by our model, which is $k \cdot s_d = 500 \cdot 8 = 4000$ bytes per unit n.

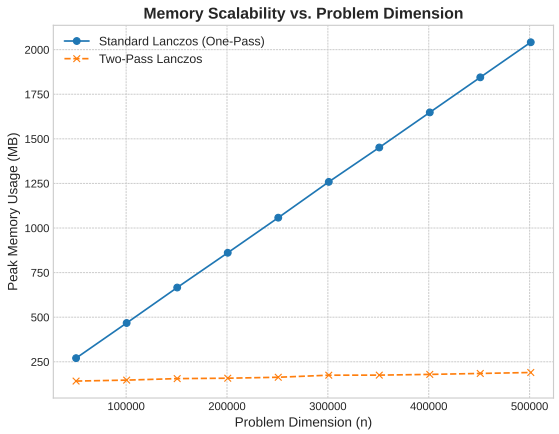

Figure 8: Peak memory usage vs. problem dimension n for a fixed number of iterations ($k = 500$), illustrating the significant $O(nk)$ cost of the standard method.

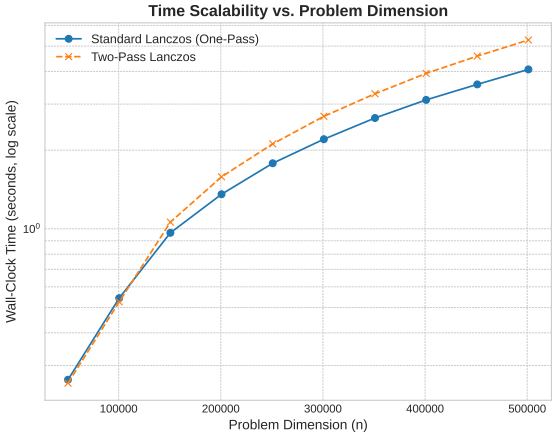

Figure 9: Wall-clock time vs. problem dimension n for a fixed number of iterations ($k = 500$), showing linear scaling for both algorithms.

Dimension (n)	**Standard (MB)**	**Two-Pass (MB)**	**Difference (MB)**
50,365	270.4	141.8	128.6
100,516	467.4	147.2	320.2
150,632	666.3	155.6	510.7
200,730	861.2	157.9	703.3
250,816	1057.9	163.2	894.7
300,894	1258.6	174.8	1083.8
350,966	1451.4	175.2	1276.2
401,033	1648.2	179.2	1469.0
451,095	1844.8	184.6	1660.2
501,155	2041.5	190.0	1851.5

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 9, scales linearly with the problem dimension n. This is consistent with the theoretical cost, which is dominated by k iterations of operations (such as sparse matrix-vector products) whose complexity is linear in n. With k held constant, the total time complexity is $O(nk)$, resulting in the observed linear scaling. For smaller problem dimensions ($n < 150,000$), the two algorithms exhibit nearly identical performance. This aligns with the *equilibrium* regime identified in Section 4.3, where the cache efficiency benefits of the two-pass method compensate for its additional computational work. As n grows larger, the cost of matrix-vector products becomes more pronounced, and the performance of the two-pass method begins to diverge, becoming progressively slower than the standard method. This confirms that for a sufficiently large number of iterations, the computational cost eventually becomes the dominant factor over memory access patterns.

4.5 Numerical Accuracy and Stability Analysis

In our final experiment, we investigate the numerical accuracy and stability of the two-pass Lanczos method. Our primary objective is to verify that the solution computed via basis regeneration is numerically equivalent to the solution from the standard one-pass algorithm across a range of distinct scenarios. This analysis addresses two fundamental questions: what is the absolute accuracy of the Lanczos approximation for a given problem, and does the two-pass method maintain this accuracy, or does the process of regenerating the basis vectors introduce a significant numerical degradation?

To isolate the factors governing algorithmic performance, we designed four distinct experimental scenarios built upon synthetic diagonal test matrices. This approach allows us to exercise precise control over the problem's characteristics and to compute the ground-truth solution, $\mathbf{x}_{\text{true}} = f(\mathbf{A})\mathbf{b}$, to machine precision. We measure the relative error of the computed solutions $\mathbf{x}_k$ (one-pass) and $\mathbf{x}'_k$ (two-pass) against this reference. We also track the relative deviation $\|\mathbf{x}_k - \mathbf{x}'_k\|_2 / \|\mathbf{x}_k\|_2$ to quantify their numerical equivalence.

Finally, we assess the stability of the basis generation by quantifying the loss of orthogonality of the Lanczos basis V_k via the Frobenius norm $\|I - V_k^H V_k\|_F$ [1]. It is important to note that we expect the loss of orthogonality to be computationally identical for both the standard and regenerated bases. This is a direct consequence of the deterministic nature of floating-point arithmetic: the second pass re-executes the exact same sequence of vector operations using the exact same stored scalar coefficients as the first pass. This results in a bit-for-bit identical regenerated basis and, therefore, an identical orthogonality loss. In contrast, a small but non-zero relative deviation between the final solutions $\mathbf{x}_k$ and $\mathbf{x}'_k$ is also expected. This arises because the two algorithms use different computational paths for solution reconstruction: the one-pass method uses a single matrix-vector product ($\mathbf{x}_k = \mathbf{V}_k\mathbf{y}_k$), while the two-pass method performs an incremental accumulation ($\mathbf{x}_k = \sum_j (\mathbf{y}_k)_j \mathbf{v}_j$). These different orderings of floating-point operations lead to minor, predictable differences at the level of machine precision.

4.5.1 Exponential on a Well-Conditioned Spectrum

We first test $f(z) = \exp(z)$ on a matrix $\mathbf{A}$ whose spectrum is a compact interval on the negative real axis. As an entire function, the exponential is analytic everywhere. The theory of polynomial approximation predicts that for such functions on a compact set, the error of the Lanczos approximation converges super-linearly to zero [2]. This scenario serves as a baseline control experiment. In the absence of any problem-induced instabilities, we hypothesize that both Lanczos variants will converge rapidly to the ground truth and that their solutions will be numerically indistinguishable.

The results for this scenario, presented in Figure 10, confirm this hypothesis. The relative error for both methods converges super-linearly, reaching the level of machine precision in fewer than 30 iterations. The lower panel shows that the relative deviation between the two solutions remains stable and of the order of machine epsilon (10^{-16}), consistent with the expected minor differences from their distinct reconstruction paths. The stability of the basis itself is shown in Figure 11. The

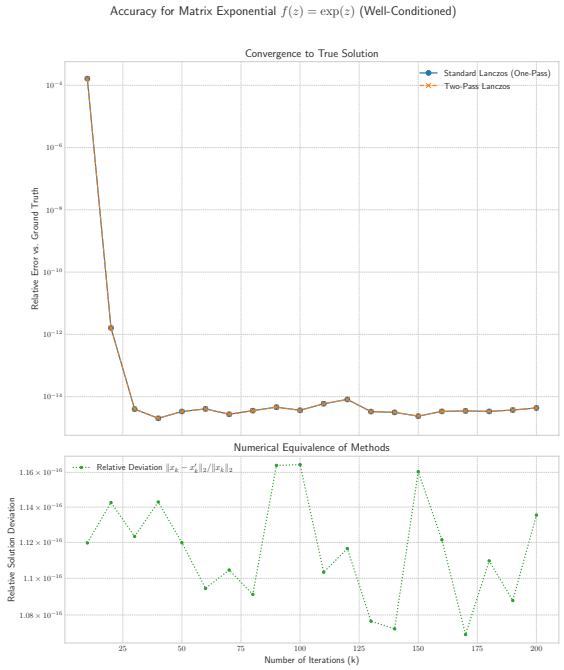

Figure 10: Accuracy for $f(z) = \exp(z)$ on a well-conditioned matrix.

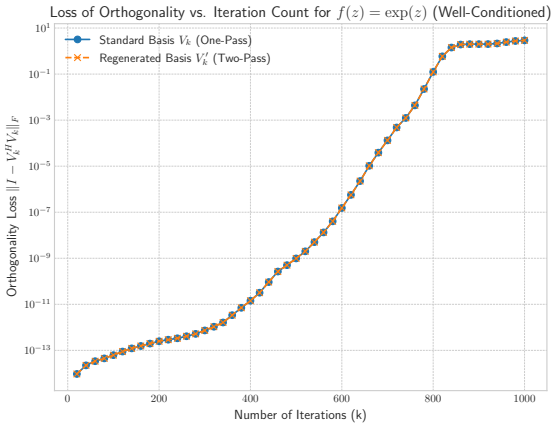

Figure 11: Orthogonality loss for $f(z) = \exp(z)$ on a well-conditioned matrix.

loss of orthogonality increases predictably with the number of iterations and, as expected, the profiles for the standard and regenerated bases are computationally identical.

4.5.2 Inverse on a Well-Conditioned Spectrum

Next, we test $f(z) = z^{-1}$ on a symmetric positive definite matrix $\mathbf{A}$ whose spectrum is strictly bounded away from zero. This recasts the problem as the solution of a well-conditioned linear system. The classical theory of the conjugate gradient method, which is algebraically equivalent to Lanczos for this problem, predicts a monotonic error decrease, with a convergence rate governed by the condition number $\kappa(\mathbf{A})$ [4]. This experiment validates the standard convergence behaviour and verifies the numerical equivalence of the two methods for a function with a singularity, provided the spectrum is kept in a safe region.

Figure 12 presents the results for this case. The convergence is linear and the error decreases monotonically until it stagnates at the level of machine precision. Again, the two methods produce nearly identical error curves, and their relative deviation is negligible. The underlying basis stability, shown in Figure 13, mirrors this result. The loss of orthogonality grows steadily but remains numerically identical for both the stored and regenerated bases.

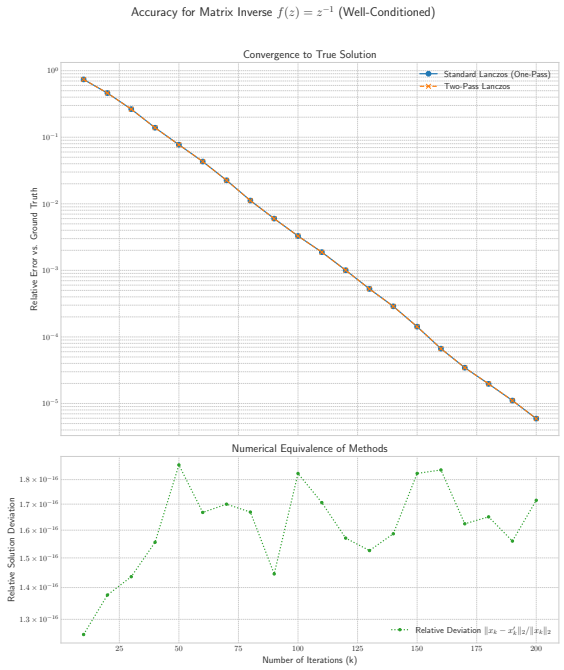

Figure 12: Accuracy for $f(z) = z^{-1}$ on a well-conditioned, positive definite matrix.

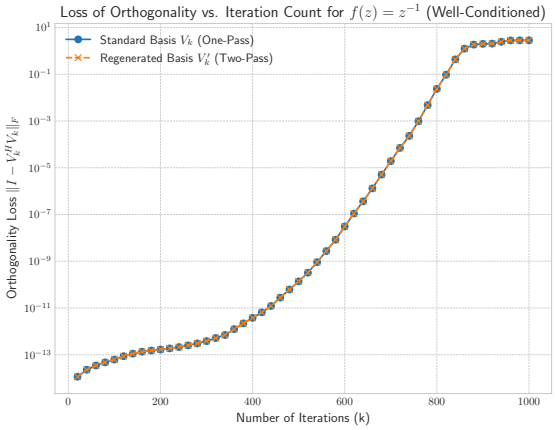

Figure 13: Orthogonality loss for $f(z) = z^{-1}$ on a well-conditioned matrix.

4.5.3 Exponential on an Ill-Conditioned Spectrum

We then test $f(z) = \exp(z)$ on a matrix with a very wide spectrum. While the problem remains well-posed, approximating $\exp(z)$ with a single polynomial over a large interval is known to be difficult, requiring a high polynomial degree k. This scenario tests the robustness of the methods under conditions of slow convergence. We hypothesize that both algorithms will exhibit identical, albeit slower, convergence, demonstrating that the two-pass approach remains stable even over a large number of iterations.

The results in Figure 14 show a significantly slower convergence rate compared to the well-conditioned case, an expected consequence of the approximation challenge. Nonetheless, the convergence remains monotonic, and the two methods continue to produce numerically close results, with their relative deviation stable at the level of machine precision. The loss of orthogonality in Figure 15 is more pronounced than in the well-conditioned case, but the profiles for the standard and regenerated bases remain indistinguishable.

4.5.4 Inverse on an Ill-Conditioned Spectrum

Finally, we test $f(z) = z^{-1}$ on a symmetric, indefinite matrix $\mathbf{A}$ constructed to be nearly singular, with an eigenvalue λ_j satisfying $|\lambda_j| \ll 1$. This provides a profound stress test for the algorithm. The Lanczos process is known to approximate extremal eigenvalues of $\mathbf{A}$ well [1]. Consequently,

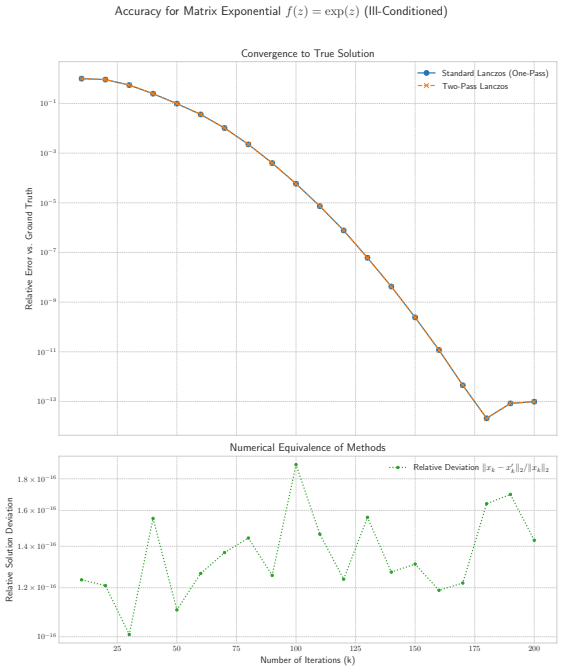

Figure 14: Accuracy for $f(z) = \exp(z)$ on a matrix with a wide spectrum.

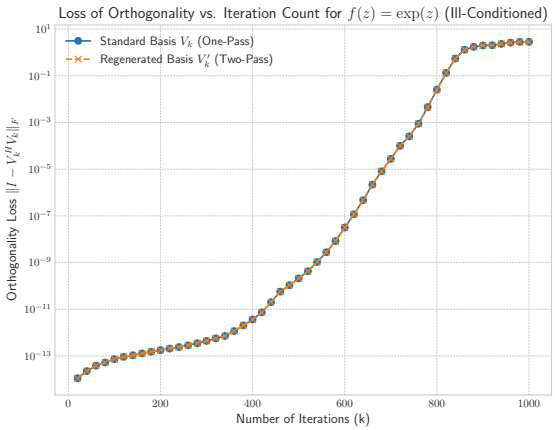

Figure 15: Orthogonality loss for $f(z) = \exp(z)$ on an ill-conditioned matrix.

the projected matrix $\mathbf{T}_k$ is expected to become severely ill-conditioned as one of its eigenvalues converges to λ_j. The computation of $\mathbf{y}_k = \mathbf{T}_k^{-1}\mathbf{e}_1\|\mathbf{b}\|_2$ is therefore expected to be numerically unstable. We hypothesize that both methods will exhibit identical behaviour, demonstrating that any observed instability is an intrinsic property of the projected problem, not a flaw introduced by the two-pass reconstruction.

Figure 16 illustrates this challenging scenario. The error plot shows three distinct phases. Initially, the error stagnates as the Krylov subspace fails to capture the problematic near-null space of $\mathbf{A}$. Subsequently, as the Lanczos process begins to approximate the near-zero eigenvalue, the projected matrix $\mathbf{T}_k$ becomes severely ill-conditioned. This induces a period of erratic, non-monotonic error behaviour, as the inversion of $\mathbf{T}_k$ becomes numerically unstable [1]. Critically, this instability affects both methods identically. Finally, as k becomes sufficiently large, the finite-termination property of the Lanczos method for linear systems comes into effect [4]. The Krylov subspace becomes rich enough to construct a highly accurate polynomial approximant for z^{-1}, and the error converges abruptly. Throughout this entire process, the solutions from the one-pass and two-pass methods remain numerically close, with a relative deviation on the order of 10^{-16}. This is the strongest evidence that the regeneration process is not a source of error; the two-pass algorithm faithfully reproduces the behaviour of the standard method, even under these extreme conditions. The basis analysis in Figure 17 confirms this finding. The severe loss of orthogonality observed coincides with the period of erratic error behavior, affirming that the instability is an intrinsic property of the Lanczos method applied to a nearly singular problem, which the two-pass algorithm correctly reproduces.

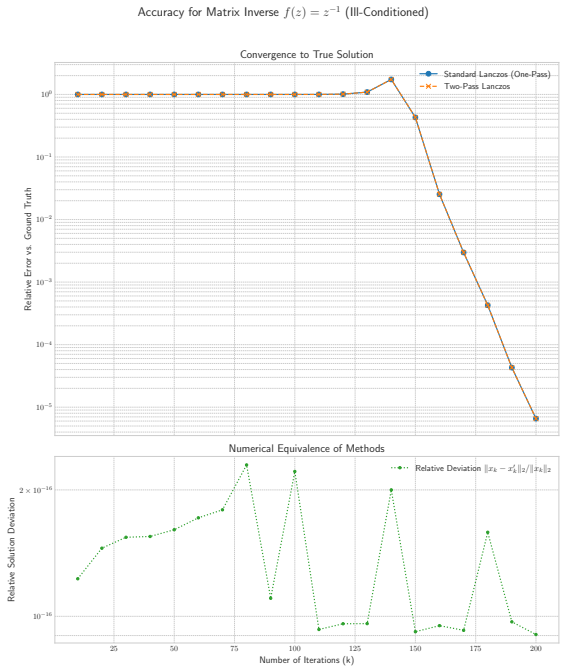

Figure 16: Accuracy for $f(z) = z^{-1}$ on a nearly singular, indefinite matrix.

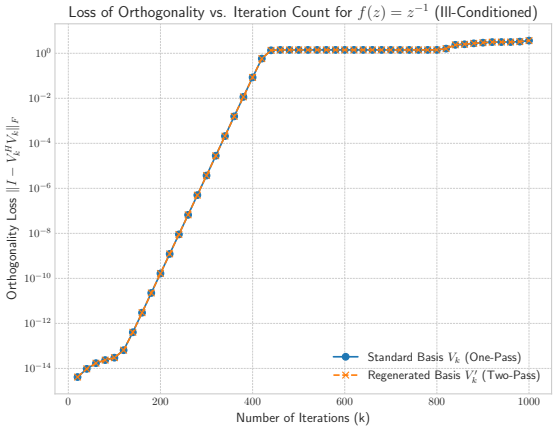

Figure 17: Orthogonality loss for $f(z) = z^{-1}$ on an ill-conditioned matrix.

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] Tyler Chen. The lanczos algorithm for matrix functions: a handbook for scientists. *arXiv preprint arXiv:2410.11090*, 2024.

[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] 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.

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