RandomizedLinearAlgebra

Nicola Noventa, Emanuele Severino

August 2025

Contents

1 Introduction

Randomized algorithms for matrix approximations have emerged as a powerful alternative to traditional deterministic methods. The key idea is to exploit randomness to construct low-dimensional subspaces that capture the dominant action of a matrix. Compared to classical algorithms, randomized approaches offer several advantages: they are often simpler to implement, naturally parallelizable, and capable of reducing both memory traffic and computational cost, especially when dealing with very large or structured matrices. These properties make them particularly appealing in modern high-performance computing scenarios, where efficiency and scalability are essential.

The aim of this project is to design and implement a C++ library that collects randomized methods for low-rank matrix approximation described in [1] and [2] .

2 Header-only Design Choice

A central architectural decision was to implement the library as *header-only*. All type definitions, templates, and algorithm implementations are contained within `.hpp` files, without requiring a separate compilation into binary objects. This design simplifies distribution and integration: users only need to include the headers.

The header-only structure is tightly coupled with a template-based design. All public components are templated on the scalar type (e.g., `float`, `double`) and on the matrix representation (dense or sparse). This allows the library to integrate seamlessly with Eigen's templates and type system. By taking advantage of Eigen's optimized kernels for the specific matrix type at compile time, algorithms can transparently operate on both dense and sparse inputs.

In practice, this design provides both flexibility and efficiency: users can experiment with different numeric precisions or storage schemes without rewriting the algorithms, while the compiler and Eigen handle the specialization and performance tuning.

3 Library Structure

The project is organized with all source code located under `include/randla`, together with test, benchmark, and build infrastructure. The structure is the following:

- **Top-level files.**
  - `CMakeLists.txt`, `build.sh`: build configuration and automation.
  - `README.md`: documentation.
  - `benchmark/`: .cpp files for performance evaluation.
  - `tests/`: correctness tests (GoogleTest).
- **Core library (`include/randla/`).**
  - `types.hpp`: common type aliases (matrices, vectors, sparse matrices, complex types, result containers).
  - `aliases.hpp`: convenient shortcuts for commonly used template instantiations (e.g., double precision).
  - `randla.hpp`: umbrella header to include the whole library at once.
- **Algorithms (`algorithms/`).**
  - `rand_range_finder.hpp`: randomized range finding methods with fixed-rank.
  - `adaptive_rand_range_finder.hpp`: adaptive range finding algorithms with fixed-precision (given tolerance as stopping criteria).
  - `matrix_factorizer.hpp`: factorizations (approximate SVD, ID, EVD).
- **Utilities.**

- `metrics/error_estimators.hpp`: posterior error estimates and residual norms.
- `random/random_generator.hpp`: Gaussian and complex random vector/matrix generation.
- `utils/matrix_generators.hpp`: synthetic test matrices (low-rank, exponential decay, PSD, sparse).
- `threading/threading.hpp`: wrapper for controlling Eigen's number of threads.

3.1 Core Library Files

`types.hpp.` Declares the `Types<FloatType>` template, which centralizes the numeric types used across the library. It collects aliases for dense and sparse Eigen objects (`Matrix`, `Vector`, `SparseMatrix`, complex variants) and defines small result containers for Stage B decompositions (`SVDResult`, `IDResult`, `EigenvalueDecomposition`). This design exposes scalar precision and storage policy through a single template parameter .

`aliases.hpp.` Provides ready-to-use aliases for the most common instantiations (float, double, long double) of the core classes: `RandRangeFinder`, `AdaptiveRandRangeFinder`, `MatrixFactorizer`, and `MatrixGenerators`. This allows users to work directly in single, double, or extended precision without repeatedly specifying template parameters .

`randla.hpp.` Acts as the umbrella header that exposes the public API of the library. It includes the core types and aliases, thread control, StageA and StageB algorithms, error estimators, random generators, and synthetic matrix generators. Users can simply include `<randla/randla.hpp>` to access the full functionality.

3.2 Utility Files

The library provides several auxiliary modules that support randomized linear algebra algorithms. In particular, three utility headers play a central role:

`error_estimators.hpp` This header introduces the `ErrorEstimators` class, which provides utilities to quantify approximation accuracy. It includes methods for posterior error estimation using randomized probes, exact error computation, and spectral norm approximation through power iteration. These tools are essential for evaluating the quality of computed subspaces.

`random_generator.hpp` This file implements the `RandomGenerator` class, a utility for producing Gaussian random matrices and vectors, both real and complex. It offers reproducibility through user-defined seeds as well as automatic seeding based on the system clock.

`threading.hpp` This header provides a lightweight abstraction for controlling multithreading in Eigen. Depending on the compilation backend, it allows the user to set and query the number of threads via OpenMP or OpenBLAS, ensuring flexible performance tuning.

`matrix_generators.hpp` This module defines the `MatrixGenerators` class, which contains static routines for constructing synthetic test matrices with controlled spectral properties. Examples include sparse random matrices, matrices with exponential singular value decay, low-rank structures with added noise, and positive semidefinite or Hermitian matrices. These generators are particularly useful for benchmarking and validating randomized algorithms.

3.3 Threading and Kernel Utilization

The library provides a lightweight threading interface through `threading.hpp`, which enables users to directly control the number of threads exploited by Eigen kernels. This is achieved via two functions:

- `setThreads(int num_threads)`: sets the number of threads used internally by Eigen. Depending on the build configuration, this call is forwarded either to `Eigen::setNbThreads` (OpenMP backend) or to `openblas_set_num_threads` (OpenBLAS backend).
- `getThreads()`: retrieves the current number of threads in use, returning the value managed by Eigen with OpenMP or OpenBLAS.

At runtime, the user can therefore adapt kernel utilization to the available resources deciding how many threads to use. Moreover, the same algorithm can transparently run with different kernel backends—OpenMP or OpenBLAS—depending on the chosen configuration.

(The build-time configuration options controlling which backend is used are explained in a dedicated section later on.)

4 Algorithms

4.1 Introduction

Randomized matrix approximation methods are typically organized into two conceptual phases, often referred to as *Stage A* and *Stage B*.

- **Stage A (Range Approximation).** The goal of Stage A is to construct a low-dimensional subspace that captures the dominant action of a large input matrix A. Random sampling techniques are used to generate a set of test vectors and project them through A, producing a basis Q with orthonormal columns such that

  $$A \approx QQ^T A.$$

  This step reduces the effective dimension of the problem and is the cornerstone of randomized methods.
- **Stage B (Postprocessing / Factorization).** Once a basis Q has been obtained, Stage B uses it to build an approximate standard factorization of A (e.g. SVD, eigenvalue decomposition, or interpolative decomposition). The matrix is projected onto the small subspace spanned by Q, computations are performed at this reduced scale, and the results are lifted back to the original space. This strategy ensures both efficiency and accuracy, while avoiding the full cost of deterministic algorithms.

In summary, Stage A provides a compact surrogate for the range of A, and Stage B leverages this surrogate to compute approximate factorizations. Together, they form a two-stage framework that underlies most modern randomized linear algebra algorithms.

4.2 Stage A: Randomized Range Finding

The library provides several algorithms, implemented in `rand_range_finder.hpp` and `adaptive_rand_range_finder.h`. These methods can be employed in two modes: *fixed-rank*, where the target dimension k of the subspace is prescribed in advance, and *fixed-precision*, where columns are added adaptively until the approximation of the error $\|(I - QQ^T)A\|$ falls below a user-defined tolerance [1].

Template structure. All algorithms in Stage A are implemented as `static` template methods inside the class `RandRangeFinder<FloatType>` (and its adaptive counterpart). The scalar type is controlled by the template parameter `FloatType`, while the input matrix is expressed as a generic `MatLike`, so that both dense and sparse Eigen objects can be used with the same API.

For instance, the classical randomized range finder is defined as:

```
template<typename FloatType = double>
class RandRangeFinder {
public:
```

```
    template<class MatLike>
    static Matrix randomizedRangeFinder(
        const MatLike& A, int l, int seed) {
        ...
    }
};
```

In addition, some methods (such as the SRFT-based fast randomized range finder) restrict the template parameter to dense Eigen matrices, using `MatrixBase<Derived>` and `static_assert` to enforce constraints at compile time:

```
template<typename Derived>
static CMatrix fastRandRangeFinder(
    const Eigen::MatrixBase<Derived>& A,
    int l, int seed) {
    ...
}
```

4.2.1 Randomized Range Finder

The simplest realization of Stage A is the *Randomized Range Finder*, corresponding to Algorithm 4.1 in [1]. Given a target dimension l, the procedure samples a Gaussian test matrix $\Omega \in \mathbb{R}^{n \times l}$ and forms the sketch $Y = A\Omega$. An orthonormal basis Q for the columns of Y is then computed via a thin QR factorization, yielding the approximation $A \approx QQ^T A$.

Algorithm 1 randomizedRangeFinder

Require: $A \in \mathbb{R}^{m \times n}$, target l

1: Draw $\Omega \in \mathbb{R}^{n \times l}$ with i.i.d. $\mathcal{N}(0, 1)$ entries
2: $Y \leftarrow A\,\Omega$ ▷ Dense or sparse mat–mat product, multi-threaded
3: Compute thin QR: $Y = QR$
4: **return** $Q \in \mathbb{R}^{m \times l}$ with orthonormal columns

In our implementation (see `rand_range_finder.hpp`), the classical randomized range finder is exposed as the static method `randomizedRangeFinder`. Random test matrices are drawn using our templated `RandomGenerator`, ensuring reproducibility via seed control. The computationally dominant step, $Y = A\Omega$, directly benefits from Eigen's overload of the matrix–matrix product operator, which internally dispatches to optimized multi-threaded kernels. The subsequent basis extraction is then carried out using Eigen's `HouseholderQR`.

Moreover, since Eigen provides a unified interface for dense and sparse types, the same code transparently handles sparse inputs: the product $A\Omega$ is automatically dispatched to sparse kernels, resulting in a speed-up when A has low density. This design makes the algorithm suitable not only for dense numerical linear algebra, but also for large-scale problems where A is sparse or structured.

4.2.2 Randomized Power Iteration

The method `randomizedPowerIteration` extends the basic range finder by repeatedly applying the power map $(AA^T)^q A\Omega$. This suppresses the contribution of small singular values and amplifies the dominant ones, leading to a more accurate subspace approximation when the singular value spectrum decays slowly. The trade-off is a moderate increase in computational cost due to additional matrix multiplications.

4.2.3 Randomized Subspace Iteration

This variant improves the stability of the power method by alternating multiplications with A and A^T, combined with explicit orthonormalization at each step. At every iteration, the intermediate subspace is projected onto both the row and column space of A, reducing numerical

Algorithm 2 Randomized Power Iteration

Require: Matrix $A \in \mathbb{R}^{m \times n}$, target dimension l, power parameter q
Draw a Gaussian random matrix $\Omega \in \mathbb{R}^{n \times l}$
$Y \leftarrow A\Omega$
for $i = 1, \ldots, q$ **do**
    $Y \leftarrow A^T Y$
    $Y \leftarrow AY$
end for
Orthonormalize Y with thin QR to obtain Q
Ensure: Orthonormal basis $Q \in \mathbb{R}^{m \times l}$

drift and improving robustness, especially when the singular spectrum decays slowly or when A is ill-conditioned.

4.2.4 Fast Randomized Range Finder (SRFT)

Use of FFTW. For the implementation of the `fastRandRangeFinder`, we relied on the external `FFTW3` library. The reason is that this algorithm requires applying the Discrete Fourier Transform to each row of the input matrix as part of the SRFT sketching process. While Eigen provides efficient BLAS-based kernels for linear algebra, it does not include optimized FFT routines.

Implementation. We accelerate the sampling $Y = A\Omega$ by using a subsampled randomized Fourier transform (SRFT). In our code (`fastRandRangeFinder` in `rand_range_finder.hpp`), we build Ω implicitly as $\Omega \approx \frac{1}{\sqrt{l}} DFR$, where: (i) R selects l column indices uniformly without replacement, (ii) D is a diagonal complex phase matrix with entries $e^{i\theta_j}$, and (iii) F is the unitary discrete Fourier transform (DFT). We then compute $Y = A\Omega$ row-by-row: apply the phases to each row of A, run a 1D FFT (via FFTW) on that row, and subsample the l selected Fourier coordinates with the $1/\sqrt{l}$ scaling; finally, a thin complex QR on Y returns $Q \in \mathbb{C}^{m \times l}$. The routine enforces *dense-only* inputs with a `static_assert` (SRFT path) and uses a single FFTW plan re-executed for each row to amortize planning cost.

Algorithm 3 Fast Randomized Range Finder (SRFT)

Require: Dense $A \in \mathbb{R}^{m \times n}$, sketch size l, seed
Ensure: Orthonormal $Q \in \mathbb{C}^{m \times l}$
Draw l distinct indices $I = \{i_1, \ldots, i_l\} \subset \{0, \ldots, n-1\}$ (uniform)
Draw phases $D = \mathrm{diag}(e^{i\theta_0}, \ldots, e^{i\theta_{n-1}})$ with $\theta_j \sim \mathrm{Unif}[0, 2\pi)$
Create a 1D FFT plan of length n (FFTW), reusable for each row
$Y \leftarrow 0 \in \mathbb{C}^{m \times l}$; $s \leftarrow 1/\sqrt{l}$
for $r = 1, \ldots, m$ **do**
    Load row $a_r \in \mathbb{R}^{1 \times n}$; form $x \leftarrow a_r \odot \mathrm{diag}(D)$ (apply complex phases)
    $z \leftarrow \mathrm{FFT}(x)$
    for $t = 1, \ldots, l$ **do**
        $Y(r, t) \leftarrow s \cdot z[i_t]$ ▷ subsample Fourier coefficients
    end for
end for
Compute thin complex QR: $Y = QR$ (e.g., Householder QR);
return Q

4.2.5 Adaptive Randomized Range Finder (ARF)

A more sophisticated realization of Stage A is the *Adaptive Randomized Range Finder*, corresponding to Algorithm 4.2 in [1]. Instead of fixing the subspace dimension in advance, the algorithm grows the basis Q incrementally until the approximation error $\|(I - QQ^T)A\|$ is deemed sufficiently small. Since the exact error is expensive to compute, the algorithm uses a probabilistic

upper bound based on multiple independent Gaussian probes:

$$\|(I - QQ^T)A\| \;\le\; 10\sqrt{2/\pi}\,\max_{i=1,\dots,r} \|(I - QQ^T)A\omega^{(i)}\|,$$

which holds with probability at least $1 - 10^{-r}$. The procedure therefore terminates once all probe vectors fall below the corresponding threshold, guaranteeing with high probability that the actual error is below the user-defined tolerance. In our implementation (see `adaptive_rand_range_finder.hpp`), the method is exposed as `adaptiveRangeFinder`, using our generic `RandomGenerator` for reproducible randomization and Eigen's kernels for efficient matrix–vector updates.

4.2.6 Adaptive Power Iteration (API)

The *Adaptive Power Iteration* enhances the sketching step by applying q alternating multiplications with A and A^T to each random probe before forming the basis. This iterative refinement produces vectors more aligned with the dominant singular subspace, yielding higher accuracy especially when singular values decay slowly. The termination criterion remains probabilistic as in ARF, ensuring the approximation meets the user-specified tolerance. For the full theoretical analysis and error bounds we refer to [1].

4.2.7 Adaptive Fast Randomized Range Finder (AFRRF)

This routine wraps the fixed–rank `fastRandRangeFinder` in a simple outer loop. Starting from an initial target size l_0, it repeatedly: (i) computes a basis Q with rank l, (ii) evaluates the absolute residual $\|(I - QQ^T)A\|$, and (iii) increases l until the residual falls below the tolerance ε or l reaches $\min(m, n)$. The update of l follows a user–controlled growth rule: multiplicative ($l \leftarrow \lceil l \cdot$ `growth_factor`$\rceil$ for `growth_factor` > 1) or additive in steps of $\lceil$`growth_factor` $\cdot\, l_0\rceil$ when $0 <$ `growth_factor` ≤ 1, with a guaranteed minimum increment. This delivers a pragmatic fixed-precision strategy built on top of the fast fixed–rank kernel.

4.3 Stage B: Matrix Factorization from a Randomized Range

Stage B provides routines for constructing approximate standard factorizations of a given matrix $A \in \mathbb{R}^{m\times n}$, once an orthonormal basis $Q \in \mathbb{R}^{m\times k}$ for its range has been computed in Stage A. The guarantee underlying this stage is that

$$\|A - QQ^*A\| \le \varepsilon,$$

for a prescribed tolerance ε. Under this condition, any low-rank factorization $A \approx CB$ can be converted into a standard matrix decomposition by exploiting the relation $C = Q$, $B = Q^*A$.

The simplest approach forms the reduced matrix $B = Q^*A$ explicitly and applies to it a classical factorization, such as SVD or QR. Since $B \in \mathbb{R}^{k\times n}$ is much smaller than A, this step provides significant computational savings without degrading the accuracy of the approximation. More elaborate approaches, such as those based on row extraction or interpolative decompositions, avoid the explicit computation of matrix–matrix products and can be more efficient depending on the problem structure.

Template Structure The implementation mirrors that of Stage A. All Stage B algorithms are implemented as *static* template methods inside the class `MatrixFactorizer<FloatType>`. The scalar type is determined by the template parameter `FloatType`, while the input matrix is expressed through a generic `MatLike` interface, allowing both dense and sparse Eigen objects to be used seamlessly within the same API.

4.3.1 Direct SVD

The *Direct SVD*, corresponding to Algorithm 5.1 in [1], is the simplest method in Stage B. Given $Q \in \mathbb{R}^{m\times k}$ such that $\|A - QQ^TA\| \le \varepsilon$, the matrix is projected as $B = Q^TA \in \mathbb{R}^{k\times n}$. An exact SVD $B = \widetilde{U}\Sigma V^T$ is then computed, and the left singular vectors are lifted back to the original

space: $U = Q\widetilde{U}$. This yields the approximation $A \approx U\Sigma V^T$, preserving the residual bound from Stage A.

Algorithm 4 directSVD

Require: $A \in \mathbb{R}^{m\times n}$, $Q \in \mathbb{R}^{m\times k}$ with orthonormal columns
1: $B \leftarrow Q^T A$
2: Compute SVD: $B = \widetilde{U}\Sigma V^T$
3: $U \leftarrow Q\widetilde{U}$
4: **return** U, Σ, V

In our implementation (`matrix_factorizer.hpp`), the routine `directSVD` first checks that Q satisfies the tolerance condition, then forms B, computes its SVD with Eigen's `JacobiSVD`, and returns the factors (U, Σ, V) in an `SVDResult` structure. The main cost is the projection $Q^T A$, which makes the method efficient whenever $k \ll m$.

4.3.2 Direct Eigenvalue Decomposition

When $A \in \mathbb{R}^{n\times n}$ is Hermitian, Stage B can exploit the symmetry to compute an approximate eigendecomposition instead of an SVD. Given an orthonormal basis $Q \in \mathbb{R}^{n\times k}$ with $\|A - QQ^T A\| \leq \varepsilon$, we form the small projected matrix

$$T = Q^T AQ \ \in \mathbb{R}^{k\times k}.$$

Since $A \approx QTQ^T$, computing the eigendecomposition

$$T = \widetilde{U}\Lambda\widetilde{U}^T$$

yields the approximate factors

$$A \ \approx \ U\Lambda U^T, \qquad U = Q\widetilde{U}.$$

The residual satisfies $\|A - U\Lambda U^T\| \leq 2\varepsilon$ as shown in [1].

Algorithm 5 directEigenvalueDecomposition

Require: Hermitian $A \in \mathbb{R}^{n\times n}$, $Q \in \mathbb{R}^{n\times k}$
1: $T \leftarrow Q^T AQ$
2: Compute eigendecomposition: $T = \widetilde{U}\Lambda\widetilde{U}^T$
3: $U \leftarrow Q\widetilde{U}$
4: **return** U, Λ

In `matrix_factorizer.hpp`, the method `directEigenDecomposition` first verifies the approximation quality of Q and ensures that A is Hermitian. It then builds the small matrix T, diagonalizes it with Eigen's `SelfAdjointEigenSolver`, and expands the eigenvectors as $U = Q\widetilde{U}$. This approach is efficient, requiring $O(kn^2)$ flops, and preserves accuracy while exploiting the Hermitian structure of A.

4.3.3 Eigenvalue Decomposition via Nyström Method

When $A \in \mathbb{R}^{n\times n}$ is positive semidefinite, the *Nyström method* provides a more accurate factorization than direct projection, while preserving symmetry at essentially no extra cost [1]. Given an orthonormal basis $Q \in \mathbb{R}^{n\times k}$ with $\|A - QQ^T AQQ^T\| \leq \varepsilon$, we form

$$B_1 = AQ, \qquad B_2 = Q^T AQ.$$

A Cholesky factorization $B_2 = CC^*$ is computed, and the factor

$$F = B_1 C^{-1}$$

is obtained by triangular solve. Since $A \approx FF^*$, an SVD of F,

$$F = U\Sigma V^T,$$

yields the approximate eigendecomposition

$$A \;\approx\; U\Lambda U^T, \qquad \Lambda = \Sigma^2.$$

Algorithm 6 eigenvalueDecompositionViaNystromMethod

Require: PSD matrix $A \in \mathbb{R}^{n\times n}$, $Q \in \mathbb{R}^{n\times k}$
1: $B_1 \leftarrow AQ, \quad B_2 \leftarrow Q^T AQ$
2: Compute Cholesky: $B_2 = CC^*$
3: $F \leftarrow B_1 C^{-1}$
4: Compute SVD: $F = U\Sigma V^T, \quad$ set $\Lambda = \Sigma^2$
5: **return** U, Λ

In `matrix_factorizer.hpp`, the method `eigenvalueDecompositionViaNystromMethod` first checks that A is positive semidefinite, then computes B_1 and B_2, performs a Cholesky decomposition of B_2, and constructs $F = B_1 C^{-1}$. The final eigenpairs are obtained from the SVD of F, with $\Lambda = \Sigma^2$. Compared to direct Hermitian postprocessing, the Nyström method provides higher accuracy for PSD matrices while requiring only a single pass over A.

4.3.4 Single-Pass Eigenvalue Decomposition

For Hermitian $A \in \mathbb{R}^{n\times n}$, a single pass over A suffices to build both a basis and an approximate eigendecomposition [1]. Let $\Omega \in \mathbb{R}^{n\times \ell}$ be a test matrix and set $Y = A\Omega$. From Y we compute an orthonormal basis $Q \in \mathbb{R}^{n\times k}$ verifying $\|A - QQ^T AQQ^T\| \le \varepsilon$. Defining the reduced (unknown) Hermitian $B \approx Q^T AQ$, the identity

$$BQ^T\Omega \;\approx\; Q^T Y$$

allows us to recover B by least squares; then $B = V\Lambda V^T$ and the eigenvectors of A are $U = QV$. This yields $A \approx U\Lambda U^T$ in $O(k^2 n)$ flops.

Algorithm 7 eigenvalueDecompositionInOnePass

Require: Hermitian $A \in \mathbb{R}^{n\times n}$, $\Omega \in \mathbb{R}^{n\times \ell}$
1: $Y \leftarrow A\Omega$
2: Compute basis Q for range(Y)
3: Solve (least squares) for Hermitian B_{approx} such that $B_{\text{approx}}(Q^T\Omega) \approx Q^T Y$
4: Eigendecompose $B_{\text{approx}} = V\Lambda V^T$
5: **return** $U \leftarrow QV$, Λ

The scheme touches A only once—avoiding the product $Q^T A$—and, after sketching, requires just $O(k^2 n)$ work, which makes it suitable for streaming, out-of-core, and sparse scenarios. Its accuracy is generally below that of two–pass postprocessing and is sensitive to the conditioning of $Q^T\Omega$; mild oversampling ($\ell = k + p$, $p \in [5, 20]$) improves robustness. For PSD inputs, Nyström postprocessing typically attains tighter spectral errors at a comparable pass budget.

4.3.5 Interpolative Decomposition (ID)

The *Interpolative Decomposition* (ID) approximates a matrix $A \in \mathbb{C}^{m\times n}$ by a product $A \approx BP$, where B consists of k selected columns of A, and $P \in \mathbb{C}^{k\times n}$ is a coefficient matrix such that a subset of its columns forms the $k \times k$ identity matrix and all other entries are bounded in modulus by a small constant (ideally no greater than 2). Moreover, with high probability, the spectral error satisfies

$$\|A - BP\|_2 \lesssim \sigma_{k+1}(A),$$

where $\sigma_{k+1}(A)$ is the $(k+1)$-th singular value of A.

A randomized algorithm for computing the ID is proposed in [2], and is particularly effective when matrix–vector multiplications with A and A^* are fast. The algorithm consists of the following steps:

1. **Sketching.** Draw a Gaussian random matrix $R \in \mathbb{C}^{\ell \times m}$ with $\ell = k + p$, and form the sketch $Y = RA$.

2. **Interpolative decomposition of the sketch.** Use column-pivoted QR on Y to extract k well-chosen columns and form $Z = Y(:, J)$. Then compute a matrix $P \in \mathbb{C}^{k \times n}$ such that

$$\|ZP - Y\|_2 \lesssim \tau_{k+1},$$

   where τ_{k+1} is the $(k+1)$st singular value of Y. The matrix P is constructed so that a subset of its columns forms the identity and $\|P\|_\infty \leq 2$.

3. **Reconstruction.** The same indices J are used to extract $B = A(:, J)$, yielding the final approximation $A \approx BP$.

The overall cost of the algorithm is

$$\mathcal{O}(\ell C_{A^*} + km + k\ell n \log n),$$

where C_{A^*} is the cost of applying A^* to a vector.

4.3.6 ID Factorization

Our implementation. The method `IDFactorization` follows a similar high-level structure, but deviates in the interpolation step. We generate the sketch $Y = RA$ with a Gaussian matrix R, and perform a column-pivoted QR decomposition on $Y^\top$ to identify the k most informative columns of A, which define the matrix B.

Instead of constructing P from the QR factors of Y, we compute it via the least-squares solution

$$P = B^+ A,$$

where B^+ is the Moore–Penrose pseudoinverse of B, computed via singular value decomposition (SVD).

Advantages and limitations. This approach is straightforward and often effective in minimizing Frobenius error. However, it does not enforce the interpolative structure of P: no identity block is guaranteed, and entries of P may be large if B is ill-conditioned. Consequently, the theoretical bound

$$\|A - BP\|_2 \lesssim \sigma_{k+1}(A)$$

does not generally hold in this variant.

4.3.7 Adaptive ID Factorization.

In scenarios where the target rank is not known a priori, we provide an adaptive variant of the interpolative decomposition via the routine `adaptiveIDFactorization`. The algorithm attempts to find the smallest rank k for which the approximation $A \approx BP$ captures a prescribed fraction of the Frobenius norm of A.

Starting from an initial guess k_0, the method performs repeated calls to `IDFactorization`, increasing the rank geometrically (e.g., $k \mapsto \lceil \gamma k \rceil$ with $\gamma > 1$) until the quantity

$$\text{energy} = 1 - \frac{\|A - BP\|_F^2}{\|A\|_F^2}$$

exceeds a user-specified threshold (e.g., 0.9).

At each iteration, the current ID approximation is evaluated in Frobenius norm, and the process terminates once the required energy is preserved. The method does not reuse intermediate sketches across iterations, which may impact performance.

This strategy is effective in practice and removes the burden of specifying a fixed rank. However, the approximation does not inherit the theoretical spectral guarantees of Algorithm I and the structure of P is not explicitly controlled.

4.3.8 Postprocessing via Row Extraction

Several matrix factorizations, including the Singular Value Decomposition (SVD) and the Eigenvalue Decomposition (EVD), can be efficiently approximated using a class of postprocessing techniques based on *row extraction*. These methods operate under the assumption that an orthonormal basis $Q \in \mathbb{C}^{m \times k}$ has already been computed such that

$$A \approx QQ^*A,$$

or, in the Hermitian case, $A \approx QQ^*AQQ^*$. Instead of directly projecting A to obtain Q^*AQ or Q^*A, the approximation is built by selecting a subset of k rows of A, identified via an *Interpolative Decomposition (ID)* of the rows of Q.

This idea leads to highly efficient algorithms that avoid matrix–matrix multiplications, and has been formalized in Algorithm 5.2 and Algorithm 5.4 of [1], respectively for SVD and EVD.

Common framework. Both postprocessing routines share the following structural steps:

1. Compute an ID of the rows of Q, i.e., $Q \approx XQ(J, :)$.
2. Use the indices J to extract corresponding rows or submatrices of A.
3. Form a small core matrix (via QR or direct access to $A(J, J)$).
4. Apply a standard decomposition (SVD or EVD) to the core matrix.
5. Reconstruct the final factors using the interpolation weights.

These algorithms provide significant speedups over projection-based methods, and their approximation errors can be rigorously bounded in terms of the residual $\|A - QQ^*A\|$ and the norm of the interpolation matrix X. However, a critical requirement is the quality of the interpolative decomposition used to extract rows. Specifically, the ID must enforce the structural property that a subset of the coefficient matrix forms the identity and all remaining entries are bounded in norm.

Implementation and limitations. In our library, the routines `SVDViaRowExtraction` and `eigenvalueDecompositionViaRowExtraction` implement Algorithm 5.2 and Algorithm 5.4 respectively. Both rely on `adaptiveIDFactorization` applied to $Q^\top$ in order to construct the interpolative model $Q \approx XQ(J, :)$.

While the implementations are structurally faithful to the algorithms described in [1], the current version of `adaptiveIDFactorization` does not guarantee the properties required of a canonical interpolative decomposition. As a result, the ID step introduces significant error, which propagates through the postprocessing and leads to degraded accuracy in the final factorizations $A \approx U\Sigma V^*$ or $A \approx U\Lambda U^*$.

Improving the quality of the ID routine—so that it better respects the theoretical structure—is necessary to make these row-extraction methods reliable in practice.

5 Testing Framework and Setup

To assess the correctness of the implemented algorithms, a suite of unit tests was developed using the `GoogleTest` framework and integrated into the project through `CMake` and `CTest`. The tests cover the main components of the library: (i) fixed-rank range finding algorithms, (ii) adaptive

fixed-precision routines, and (iii) matrix factorization methods (SVD, ID, and eigenvalue decompositions).

All tests were executed on synthetic problems generated by the `MatrixGenerators` utilities, which provide well-controlled families of matrices (low-rank, exponential decay, noisy perturbations, Hermitian, PSD, sparse, and dense). This choice ensures reproducibility and allows validation of the algorithms against matrices with known spectral properties.

The testing setup enforced a single-threaded configuration to guarantee deterministic results and avoid variability due to parallel scheduling. Tolerance thresholds were defined according to the matrix class under consideration (e.g. 10^{-8} for exact low-rank cases, 10^{-3} for exponential decay, looser bounds for noisy instances).

5.1 Fixed-Rank Range Finder Tests

The first group of tests focuses on the fixed-rank setting, where the target subspace dimension $k = l$ is specified in advance. We validate several randomized algorithms implemented in the library, namely the `randomizedRangeFinder`, `randomizedPowerIteration`, `randomizedSubspaceIteration`, and `fastRandRangeFinder`.

Two matrix families are considered:

- **Exponential decay.** A dense matrix with singular values decaying geometrically, generated through `matrixWithExponentialDecay`. This class provides a challenging testbed where the effective numerical rank is smaller than the ambient dimension. The tests ensure that all algorithms capture the dominant singular directions up to tolerance 10^{-3}.
- **Exact low-rank (noise-free).** A synthetic matrix of rank l, built via `lowRankPlusNoise` with zero noise. This case acts as a baseline, verifying that the algorithms recover the column space with high accuracy (tolerance 10^{-8}) and produce an orthonormal basis with exactly l columns.

For each scenario, the tests check both the residual approximation error $\|(I - QQ^T)A\|$ against the chosen tolerance and the consistency of the output dimension ($Q.cols() = l$). This confirms that the fixed-rank routines are numerically stable and respect the prescribed target rank across different problem structures.

5.2 Fixed-Precision Range Finder Tests

The second group of tests addresses the adaptive fixed-precision setting, where the algorithms dynamically expand the basis until the residual error falls below a user-defined tolerance. We validate three methods: `adaptiveRangeFinder`, `adaptivePowerIteration`, and `adaptiveFastRandRangeFinder`.

Four matrix classes are employed:

- **Exact low-rank.** A matrix of rank 20 with unit singular values, generated with `matrixWithSingularValues`. This serves as a reference case, with a stringent tolerance of 10^{-10}.
- **Exponential decay.** A matrix with geometrically decaying singular values, built via `matrixWithExponentialDe` Here the algorithms must adaptively stop once the trailing spectrum is negligible, with tolerance set to 10^{-3}.
- **Low-rank noise-free.** A synthetic low-rank structure without perturbations (`lowRankPlusNoise` with noise level 0), tested at tolerance 10^{-10}.
- **Low-rank with noise.** A low-rank structure perturbed by small Gaussian noise ($\sigma = 10^{-3}$), requiring looser accuracy checks with tolerance $5 \cdot 10^{-3}$.

Each test verifies that the residual approximation error $\|(I - QQ^T)A\|$ is below the prescribed tolerance. This confirms that the routines consistently drive the error below the required accuracy threshold across different spectral profiles and noise levels.

5.3 Matrix Factorization Tests

The third group of tests validates the algorithms in the `MatrixFactorizer` module, which compute low-rank factorizations from an approximate range basis. These include interpolative decompositions (ID), direct SVD/EVD factorizations, and structure-preserving decompositions for Hermitian or PSD matrices.

ID Factorization. For `IDFactorization`, we test the approximation $A \approx BP$ on several matrices where $A = Q$ is an orthonormal basis obtained from Stage A. Since the implementation does not enforce the canonical structure of the ID, we use a relaxed Frobenius error threshold of 0.3 to validate correctness:

$$\|A - BP\|_F / \|A\|_F \leq 0.3.$$

Adaptive ID Factorization. The adaptive routine `adaptiveIDFactorization` is tested indirectly. The method raises an exception if the preserved energy

$$1 - \frac{\|A - BP\|_F^2}{\|A\|_F^2}$$

falls below a user-defined threshold (typically 0.9). Therefore, successful completion of the routine is sufficient to certify its behavior. No additional numerical check is needed.

Direct SVD and EVD Methods. We test the routines `directSVD`, `directEigenvalueDecomposition`, `eigenvalueDecompositionViaNystromMethod`, and `eigenvalueDecompositionInOnePass`. In each case, a basis Q is computed for the input matrix A, and the algorithm returns an approximate reconstruction A_{approx}. The Frobenius norm of the residual is evaluated:

$$\|A - A_{\text{approx}}\|_F \leq \texttt{1e-6},$$

Tests are run on different input families, including sparse, dense, Hermitian, and PSD matrices.

Note on Row Extraction Methods. Additional tests for `SVDViaRowExtraction` and `eigenvalueDecompositionV` are included but currently disabled, due to insufficient accuracy caused by limitations in the ID step, as discussed in Section 4.3.6.

6 Benchmark

6.1 Experimental Setup

Hardware. All benchmarks were run on a laptop with an 11th-Gen Intel Core i7-1165G7 (4C/8T, 2.80 GHz base) and 16 GB RAM (4267 MT/s).

Kernels & threading. Eigen kernels were exercised with a thread sweep $\{1, 2, 4, 8\}$ *only* for the fixed-rank benchmarks, setting threads at runtime via `randla::threading::setThreads` and logging the backend tag (`openMP`/`openBLAS`). For the fixed-precision benchmarks we report single-thread results (threads= 1), since these routines are not internally parallelized.

Test problems. *Fixed–rank Stage A:* we consider four dense matrix families with $(m, n) = (5000, 2000)$: (i) low–rank with rank 1000; (ii) low–rank plus noise ($\sigma = 0.05$); (iii) exponential singular-value decay (rate 0.5); and (iv) i.i.d. Gaussian. *Fixed–precision Stage A:* we consider the same four families with $(m, n) = (3000, 1500)$, using rank 500 for the low-rank cases and $\sigma = 0.01$ for the noisy case. All instances are generated with the project's matrix utilities before each run.

Algorithms and parameters. *Fixed–rank Stage A:* Randomized Range Finder (RRF), Randomized Power Iteration (RPI, $q = 2$), Randomized Subspace Iteration (RSI, $q = 2$), and Fast RRF (FRF), all at target dimension $l = 1000$. Results are written to `res_benchmark_fixed_rank_A.csv`. *Fixed–precision Stage A:* Adaptive Randomized Range Finder (ARRF), Adaptive Power Iteration (API, $q = 2$) and Adaptive Fast RRF (AFRRF) with tolerance $\varepsilon = 10^{-2}$ and $r = 10$ Gaussian probes; results are written to `res_benchmark_fixed_precision_A.csv`.

Warmup and timing. Before timing, a light warmup on smaller low-rank matrices is performed. Wall-clock time is measured with `std::chrono::high_resolution_clock`. For fixed-rank, we log (label, $m, n, \|A\|$, method, l, threads, tag, cols, err, time_ms) to CSV. For fixed-precision, we log (label, $m, n, \|A\|$, method, ε, r, q, threads, tag, cols, err, rel_err, time_ms).

Accuracy metric. After computing Q, the absolute residual $\|(I - QQ^\top)A\|$ is evaluated with `Err::realError` and reported as `err`.

6.2 Performance Evaluation

Fixed-rank Tables 1–2 show the *average speedup* of the Stage A (fixed-rank) algorithms relative to the 1-thread baseline. Each row corresponds to a method and each column to a thread count $t \in \{1, 2, 4, 8\}$. Values are arithmetic means over the four dense test families described in the Experimental Setup. Table 1 refers to Eigen using a BLAS backend (OpenBLAS), while Table 2 refers to Eigen's OpenMP kernels.

Table 1: Average speedup per algorithm (openBLAS)

Method	**1t**	**2t**	**4t**	**8t**
RRF	1.00	1.42	1.35	1.11
RPI	1.00	1.26	1.33	1.14
RSI	1.00	1.42	1.39	1.05
FRF	1.00	1.46	1.39	1.05

Table 2: Average speedup per algorithm (openMP)

Method	**1t**	**2t**	**4t**	**8t**
RRF	1.00	1.17	1.25	1.29
RPI	1.00	1.37	1.49	1.64
RSI	1.00	1.09	1.19	1.25
FRF	1.00	0.96	1.01	0.98

The benchmarks show that scalability depends more on the underlying operations than on the high-level algorithm. With the **BLAS** backend, performance improves up to two threads but then quickly saturates. Under **OpenMP**, the *Randomized Power Iteration* benefits the most, reaching about 1.6× speedup at eight threads, while *Randomized Range Finder* and *Randomized Subspace Iteration* show only modest gains, and *Fast Randomized Range Finder* even deteriorates. A deeper timing breakdown highlights that matrix multiplication scales well, whereas the QR factorization remains serial in Eigen, thus limiting overall scalability. According to the Eigen documentation, QR is sequential unless a multithreaded LAPACK backend is used. Future work could explore the use of parallelizable QR algorithms, for instance by linking Eigen to a LAPACK-enabled backend, or by integrating alternative libraries that provide multithreaded QR factorizations.

From the experimental results, we observed that the execution times obtained with BLAS are consistently lower than those obtained with OpenMP. This trend holds across all tested configurations. The reduction in mean execution times with BLAS is significant. Overall, BLAS outperforms OpenMP in every case observed.

Table 3: Execution times (ms) for Gaussian matrix using OpenMP and BLAS.

Method	OpenMP	BLAS
RRF	4267	1704
RPI	8966	2931
RSI	15502	6268
FRF	11960	3298

Fixed-precision All timings were collected with a single thread, the goal here is to compare the *intrinsic* algorithmic cost. Runtimes are mainly driven by the numerical rank induced by the tolerance ($\varepsilon = 10^{-2}$): instances with fast spectral decay (Low-rank, ExpDecay) stop after few blocks, whereas Gaussian/noisy cases require many more and are therefore much slower. Method-wise, ARRF is consistently the fastest; API (q=2) is the slowest due to the extra passes; AFRRF sits in between for the BLAS backend (second fastest) but becomes the slowest with OpenMP.

Future work. A natural extension is to design *blocked* variants of the adaptive schemes—growing Q by blocks of b columns, forming $Y = A\Omega$ and $A^\top Y$ with GEMM, and using blocked QR/Cholesky updates. This would shift the hotspots to matrix–matrix kernels and allow Eigen's optimized (and OpenMP-parallel) GEMM to deliver multi-threaded speedups.

7 Build & Usage

7.1 Prerequisites

- **C++17** compiler and **CMake** ($\geq$ 3.15).
- **Eigen3** (header-only), **FFTW**.
- Optional: **OpenMP** (parallelism), **OpenBLAS** (for BLAS backend in Eigen via `pkg-config`).
- Having `pkg-config` installed is recommended for automatic detection of dependencies.

7.2 Quick start with `build.sh`

The easiest way is to use the provided build script (creates `build/`, configures in **Release**, compiles, and runs tests by default):

```
./build.sh
```

Useful options:

- Select backend/threading: `--threading <openmp|blas|single>`
- Disable tests: `--no-test`
- Run benchmarks: `--benchmark [fr|fp]` (without argument runs both)

Examples:

```
# OpenMP + tests
./build.sh --threading openmp

# Build only (no tests), BLAS backend (OpenBLAS)
./build.sh --no-test --threading blas

# Run fixed-rank benchmarks only (OpenMP)
./build.sh --benchmark fr --threading openmp

# Run fixed-precision benchmarks only (single-thread)
./build.sh --benchmark fp --threading single
```

7.3 Manual build (without `build.sh`)

It is also possible to configure and compile the project directly with `cmake`, without relying on the provided `build.sh` script. The build system exposes the cache variable `THREADING_MODE`, which selects how Eigen kernels are parallelized inside the library (`openmp`, `blas`, or `single`). This choice controls which optional backends are enabled in `CMakeLists.txt`:

- **OpenMP** (`THREADING_MODE=openmp`): enables Eigen's built-in OpenMP parallelism. The target is linked against `OpenMP::OpenMP_CXX`, and the preprocessor flag `EIGEN_USE_OPENMP` is set.
- **BLAS** (`THREADING_MODE=blas`): routes Eigen kernels through a BLAS backend (e.g. OpenBLAS, found via `pkg-config`). In this case, the target links against `OpenBLAS::OpenBLAS`, and the flag `EIGEN_USE_BLAS` is set.
- **Single-thread** (`THREADING_MODE=single`): disables both OpenMP and BLAS backends. All routines run in serial with `OMP_NUM_THREADS=1`.

The general build pattern is:

```
# OpenMP backend
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release -DTHREADING_MODE=openmp
cmake --build build -j

# BLAS backend (OpenBLAS via pkg-config)
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release -DTHREADING_MODE=blas
cmake --build build -j

# Serial (single-threaded)
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release -DTHREADING_MODE=single
cmake --build build -j
```

By default, the build also compiles unit tests, while benchmarks are excluded unless explicitly requested. These behaviors can be adjusted with two cache options:

- `BUILD_TESTS` (default `ON`): if enabled, the GoogleTest-based unit tests are compiled and registered with CTest.
- `BUILD_BENCHMARKS` (default `OFF`): if enabled, benchmark drivers under `benchmark/` are compiled.

Examples:

```
# Disable tests
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release \
      -DTHREADING_MODE=openmp -DBUILD_TESTS=OFF

# Enable benchmarks (in addition to the library)
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release \
      -DTHREADING_MODE=blas -DBUILD_BENCHMARKS=ON
```

The final result is a header-only library target `randomized-linear-algebra`, optionally accompanied by test and/or benchmark executables depending on the chosen options and backend.

7.4 Using the library in your project (CMake)

The library is **header-only** and exposes a CMake target `randomized-linear-algebra`:

```
# In your CMakeLists.txt
add_subdirectory(path/to/randomized-linear-algebra)
add_executable(my_app src/main.cpp)
target_link_libraries(my_app PRIVATE randomized-linear-algebra)
```

The target automatically propagates includes and dependencies (Eigen3, optional FFTW, OpenMP/BLAS if enabled).

7.5 Tests

With `build.sh`, tests are executed automatically (unless `--no-test` is given). Alternatively, run:

```
ctest --test-dir build --output-on-failure
```

Some tests enforce `OMP_NUM_THREADS=1` to ensure reproducibility.

7.6 Benchmarks

If benchmark sources are present, CMake builds: `benchmark_fixed_rank_A` and `benchmark_fixed_precision_A`. You can run them manually from `build/`, or via:

```
./build.sh --benchmark
```

7.7 Minimal example (C++)

```
#include <Eigen/Dense>
#include <randla/randla.hpp>
#include <iostream>

int main() {
    Eigen::MatrixXd A = Eigen::MatrixXd::Random(300, 150);
    int l = 20, seed = 42;

    // Example: range finder (see project headers for details)
    auto Q = randla::RandRangeFinderD::randomizedRangeFinder(A, l, seed);

    return 0;
}
```

8 Conclusions

In this project we developed a modular C++17 header-only library that implements a broad spectrum of randomized algorithms for low-rank matrix approximation. The design emphasizes interoperability with Eigen, flexibility across dense and sparse inputs, and support for multiple parallel backends. Extensive testing confirmed the correctness of both Stage A and Stage B routines, while benchmarks highlighted the trade-offs between OpenMP and BLAS kernels. Possible future work could include blocked implementations and enhanced interpolative decompositions, which might further improve scalability and accuracy on modern multi-core architectures.

References

[1] N. Halko, P.-G. Martinsson, and J. A. Tropp, *Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions*, SIAM Review, 53 (2011), pp. 217–288.

[2] E. Liberty, F. Woolfe, P.-G. Martinsson, V. Rokhlin, and M. Tygert, *Randomized algorithms for the low-rank approximation of matrices*, Proceedings of the National Academy of Sciences, 104 (2007), pp. 20167–20172.