



**Jagannath University**

Department of Mathematics

# **Randomized Low-Rank Approximation for PDE-Derived Kernel Matrices**

by

**Jibon Krishna Joy**

Supervised by

**Professor Dr. Rabindra Nath Mondal**

Department of Mathematics, Jagannath University

# Abstract

This thesis compares three randomized low-rank approximation methods, Gaussian rSVD, SRHT-rSVD, and randQB\_EI, on matrices that come from partial differential equations. First we show that the phrase “PDE matrix” is hiding an important difference. Two real finite-element matrices from the SuiteSparse collection resist compression: one still needs 86% of full rank for a relative Frobenius error of 10%, and the other still carries 57% error at 37% of full rank, while a boundary-integral kernel matrix, built from a smooth Green’s function between two separated point clouds, reaches a relative Frobenius error of  $3.7 \times 10^{-5}$  at rank 200. We then compare the three methods on that boundary-integral matrix at  $n = 13,514$  and  $n = 20,000$ . Gaussian rSVD and SRHT-rSVD reach the same accuracy at matched rank, and randQB\_EI trails both by about  $1.4\text{--}1.6\times$ , which is the price paid for choosing its rank automatically. SRHT’s better time complexity ( $O(n \log n)$  instead of  $O(n\ell)$ ) does not turn into a real speed win: its sketch step is  $2.1\times$  slower than Gaussian’s even though it does  $12\times$  fewer floating-point operations, because a BLAS matrix multiply sustains about 85 GFLOP/s on our machine while our compiled Hadamard transform only sustains 3.3. A plain, uncompiled implementation of the same transform is slower still, so the gap that remains after compiling and optimizing it is a hardware and library efficiency effect, and not a shortcoming of the implementation. Accuracy at matched rank does not change between the two problem sizes, which fits with the fact that this kernel’s decay is set by geometry rather than by the number of points.

# Table of Contents

|            |                                                                          |    |
|------------|--------------------------------------------------------------------------|----|
| Chapter 1  | Introduction .....                                                       | 1  |
| Chapter 2  | Background .....                                                         | 3  |
|            | 2.1 Notation and two classical results .....                             | 3  |
|            | 2.2 Symbols used in this thesis .....                                    | 3  |
|            | 2.3 Terminology .....                                                    | 4  |
|            | 2.4 Why a random sketch finds the range .....                            | 5  |
|            | 2.5 Gaussian randomized SVD .....                                        | 6  |
|            | 2.6 SRHT randomized SVD .....                                            | 7  |
|            | 2.7 randQB_EI: fixed precision instead of fixed rank .....               | 8  |
|            | 2.8 Cost summary .....                                                   | 8  |
|            | 2.9 Why boundary-integral kernel matrices compress well .....            | 9  |
| Chapter 3  | Methods .....                                                            | 10 |
|            | 3.1 Matrices .....                                                       | 10 |
|            | 3.2 Metrics .....                                                        | 11 |
|            | 3.3 Comparing a fixed-rank method against a fixed-precision method ..... | 12 |
|            | 3.4 Ground truth for large matrices .....                                | 12 |
|            | 3.5 The SRHT power-iteration extension .....                             | 12 |
|            | 3.6 Tools, software and reproducibility .....                            | 13 |
| Chapter 4  | Results .....                                                            | 15 |
|            | 4.1 FEM stiffness matrices resist low-rank approximation .....           | 15 |
|            | 4.2 Accuracy comparison on boundary-integral kernel matrices .....       | 17 |
|            | 4.3 Runtime, memory, and the SRHT speed question .....                   | 20 |
|            | 4.4 How much faster than a full SVD? .....                               | 24 |
|            | 4.5 How much do the tuning parameters matter? .....                      | 25 |
|            | 4.6 Scaling from $n = 13,514$ to $n = 20,000$ .....                      | 27 |
| Chapter 5  | Discussion .....                                                         | 29 |
|            | 5.1 Practical recommendations .....                                      | 29 |
|            | 5.2 Limitations .....                                                    | 30 |
| Chapter 6  | Future work .....                                                        | 32 |
| Chapter 7  | Conclusion .....                                                         | 34 |
| Appendix A | Key implementations .....                                                | 35 |
|            | A.1 The Fast Walsh–Hadamard Transform .....                              | 35 |
|            | A.2 The SRHT sketch .....                                                | 35 |
|            | A.3 randQB_EI’s error indicator .....                                    | 36 |
|            | A.4 Error metrics without forming the residual .....                     | 36 |
| References | .....                                                                    | 38 |

# List of Tables

|          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |    |
|----------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| Table 1  | Symbols used throughout the thesis. ....                                                                                                                                                                                                                                                                                                                                                                                                                                         | 4  |
| Table 2  | Leading operation counts, with $\ell = k + p$ . Power iteration multiplies the pass count by about $2q + 1$ for the first two methods. Only the sketch step differs between Gaussian rSVD and SRHT-rSVD; that single column is what Section 4.3 puts to the test. ....                                                                                                                                                                                                           | 8  |
| Table 3  | Every matrix used in this thesis. The two pde900/poisson3Da rows are downloaded real-world data and carry the RQ0 negative result; the boundary-integral rows are the main test case; the synthetic row is the control layer. Densities are nonzeros as a fraction of $n^2$ (4,380 and 352,762 nonzeros); dense sizes are float64 storage. The synthetic matrices are three separate spectra (exponential, polynomial, near-flat) sharing one size. ....                         | 11 |
| Table 4  | Software used. Versions are pinned in requirements.txt so the environment can be rebuilt exactly. ....                                                                                                                                                                                                                                                                                                                                                                           | 13 |
| Table 5  | Which script produces which result. Scripts live in src/experiments/ unless a path is given. The last one draws figures from CSVs the others write; it computes nothing of its own except the geometry panels, so a figure can always be redrawn without repeating an experiment. ....                                                                                                                                                                                           | 14 |
| Table 6  | Rank needed for a modest accuracy target, on two real FEM stiffness matrices. Neither one compresses well at any practical rank. poisson3Da’s exact answer comes from SuiteSparse’s own precomputed SVD; pde900’s comes from a direct dense SVD (possible at $n = 900$ ). ....                                                                                                                                                                                                   | 15 |
| Table 7  | Gaussian rSVD and SRHT-rSVD run directly on the sparse pde900 matrix, 12-seed means. Even at $k = 800$ out of 900, almost no compression at all, the error stays above 0.17, confirming Table 6’s rank-needed number with the actual randomized algorithms rather than only the exact SVD. ....                                                                                                                                                                                  | 16 |
| Table 8  | Accuracy at chosen ranks, $n = 13, 514$ boundary-integral kernel matrix, 12-seed means (randQB_EI at its achieved mean rank). “vs. opt.” is the ratio to the Eckart–Young optimal error at the same rank. Two things are worth noticing here: the gap to optimal grows with $k$ for every method, since the fixed oversampling $p = 10$ gives proportionally less slack at higher rank, and that gap is consistently worse in the spectral norm than in the Frobenius norm. .... | 18 |
| Table 9  | Synthetic exponential-decay spectrum ( $n = 2000$ ), 12-seed means. Same pattern as Table 8: Gaussian and SRHT agree closely; randQB_EI trails by a wider margin here than on the boundary-integral matrix. ....                                                                                                                                                                                                                                                                 | 18 |
| Table 10 | randQB_EI on the $n = 13, 514$ boundary-integral kernel matrix, 12-seed means, $b = 10$ , $k_{\max} = 400$ . All 72 trials converged. “Ratio” is the delivered error as a fraction of the requested tolerance: below 1 everywhere, so the tolerance is never violated, but well below 1 at the loose end. Peak memory was 1393 MB at every tolerance, unchanged by the rank reached. ....                                                                                        | 19 |
| Table 11 | randQB_EI’s convergence rate (out of 12 seeds) at $n = 2000$ , $b = 10$ , $k_{\max} = 300$ . A failure here is not a bad answer, it is no answer at all: the call raises instead of returning a factorization. ....                                                                                                                                                                                                                                                              | 19 |
| Table 12 | Runtime and memory at $n = 13, 514$ , 12-seed means. SRHT’s cost stays almost flat across $k$ , because it comes mostly from a fixed-size sketch step that does not                                                                                                                                                                                                                                                                                                              |    |

|          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |    |
|----------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
|          | depend on the target rank. So its disadvantage gets smaller as $k$ grows and Gaussian's cost catches up. ....                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        | 20 |
| Table 13 | The sketch step under the FWHT implementation used throughout this thesis and under the one it replaced, at $l = 210$ , mean of 3 calls. The pre-fix code is taken verbatim from this project's own version history, not rewritten from a description of it. For scale, Gaussian rSVD's sketch on the same matrix takes 0.90s and 22.7 MB (Table 15), so the pre-fix version was $16.7 \times$ slower and $390 \times$ heavier where the current one is $2.1 \times$ and $80 \times$ . ....                                                                                                                                          | 21 |
| Table 14 | SRHT-rSVD's runtime and peak memory, as a multiple of Gaussian rSVD's, at every size tested in this thesis. SRHT is always slower and always uses more memory, though the gap narrows at higher rank since SRHT's sketch cost is roughly fixed while Gaussian's grows with $k$ . One caveat on the memory column: the $n = 20,000$ runs use the batched sketch described in Section 4.6, while the two smaller sizes do not, so the memory ratios in the last three rows are much smaller than they would be without batching and should not be read as SRHT's overhead falling away with size. The time ratios are unaffected. .... | 21 |
| Table 15 | Sketch step only, $n = 13,514$ , $\ell = k + p = 210$ , mean of 3 timed calls after a warm-up call (the Numba kernel compiles on first use). Operation counts are analytic: $2mn\ell$ for the dense product, and $2mn_{\text{pad}} \log_2 n_{\text{pad}}$ for one padded Walsh-Hadamard transform, with $n_{\text{pad}} = 16,384$ . Note the SRHT count does not contain $\ell$ at all, it transforms everything and only keeps $\ell$ rows. ....                                                                                                                                                                                    | 22 |
| Table 16 | Power iteration at fixed rank $k = 50$ , $n = 13,514$ , 12-seed means. The two methods agree to three significant figures at every $q$ , which is the check the SRHT power-iteration extension had to pass. Peak memory (not shown) is 24.8 MB for Gaussian rSVD at $q = 0$ and 43.4 MB at every $q \geq 1$ ; SRHT-rSVD stays at 1702 MB throughout, since its cost is dominated by the padded sketch rather than by power iteration. ....                                                                                                                                                                                           | 23 |
| Table 17 | Against a full dense SVD of the same $n = 13,514$ boundary-integral kernel matrix, 3-seed means. The full SVD row is a single measurement, at 637 seconds it is not something to repeat unnecessarily. The randomized runtimes and memory figures here reproduce Table 12's to within a few percent, which is a useful check: those came from a separate benchmark run on a different day. ....                                                                                                                                                                                                                                      | 24 |
| Table 18 | Oversampling sweep at fixed rank $k = 50$ , synthetic polynomial-decay spectrum, $n = 2000$ . Accuracy improves slowly and steadily with $p$ ; doubling $p$ from 10 to 20 buys about 6% error reduction for about 13% more time in Gaussian rSVD. The two methods track each other at every $p$ , as they do at every $k$ . ....                                                                                                                                                                                                                                                                                                     | 25 |
| Table 19 | Block-size sweep for randQB_EI at fixed tolerance, synthetic polynomial-decay spectrum, $n = 2000$ . Larger blocks overshoot the tolerance, they stop at a higher rank than needed, but run faster, because the method performs fewer reorthogonalization passes. ....                                                                                                                                                                                                                                                                                                                                                               | 26 |
| Table 20 | Relative Frobenius error at matched rank, Gaussian rSVD, $n = 13,514$ vs. $n = 20,000$ . The ratios stay close to 1.0 with no clear trend, that is, no size effect we can detect over this range. ....                                                                                                                                                                                                                                                                                                                                                                                                                               | 28 |
| Table 21 | The recommendations above as a lookup table. Every entry is a finding from this thesis's own measurements at $n = 2000$ to $20,000$ , not general advice, see Section 5.2 for where it may stop holding. ....                                                                                                                                                                                                                                                                                                                                                                                                                        | 29 |

## List of Figures

- Figure 1 Left: the construction of Equation 17, sources on one sphere, targets on another of the same radius, centres  $\eta r$  apart (500 points of each shown; the experiments use thousands). Right: best rank- $k$  error at three separation ratios,  $n = 900$ . A wider gap gives faster decay, exactly as the  $O(\eta^{-P})$  rate of Equation 16 predicts. The flat floors at  $\eta = 3$  and  $\eta = 5$  are machine precision: past that rank there is nothing left to approximate. .... 10
- Figure 2 Normalized singular values  $\sigma_j/\sigma_1$  for the two real FEM matrices and the boundary-integral kernel matrix, over the first 250 indices. The FEM spectra are essentially flat; the boundary-integral spectrum decays geometrically, as Equation 16 predicts. `pde900` is from a full dense SVD, `poisson3Da` from SuiteSparse’s precomputed SVD, and the kernel matrix from the top 250 singular values. .... 15
- Figure 3 Relative Frobenius error of the best rank- $k$  approximation, the Eckart–Young optimum, so no algorithm can do better than these curves. Computed as  $\sqrt{(\|A\|_F^2 - \sum_{j \leq k} \sigma_j^2) / \|A\|_F^2}$ , which needs only the top singular values and never forms a residual. Same three matrices and same sources as Figure 2. .... 16
- Figure 4 Relative Frobenius error vs. target rank, on the  $n = 13,514$  boundary-integral kernel matrix. Gaussian rSVD and SRHT-rSVD look the same at every rank. `randQB_EI` stays a small, steady amount behind both. All three follow the Eckart–Young optimal curve closely, which shows the code is correct. .... 17
- Figure 5 Runtime and peak memory vs. target rank,  $n = 13,514$ . SRHT-rSVD is slower and uses far more memory than Gaussian rSVD at every rank we tested, even though it should be cheaper in theory. .... 20
- Figure 6 The sketch step alone,  $n = 13,514$ ,  $l = 210$ . SRHT-rSVD does far less arithmetic than Gaussian rSVD and still takes twice as long, because it retires operations at a small fraction of the rate. The left two panels are what the complexity analysis predicts and what the clock actually says; the right panel is the distance between them. .... 22
- Figure 7 Power-iteration convergence at fixed rank  $k = 50$ ,  $n = 13,514$ . Gaussian rSVD and SRHT-rSVD look the same across  $q \in \{0, 1, 2, 3\}$ . This confirms the SRHT power-iteration extension, added to remove an unfair edge that Gaussian rSVD would otherwise have, gives correct results. .... 23
- Figure 8 Runtime as a multiple of a full SVD’s, at two sizes. Everything on these axes was measured in one process against one baseline per matrix, since a ratio is only meaningful if numerator and denominator were timed under the same conditions. The dotted line is break-even. The left panel averages over the three synthetic spectra, which barely differ: neither a full SVD’s cost nor a fixed-rank sketch’s depends on the spectrum. .... 24
- Figure 9 The same sweep as Table 18, drawn. Left: the two methods’ accuracy curves lie on top of each other at every  $p$ . Right: both get slower as  $p$  grows, but Gaussian rSVD’s time nearly doubles across the sweep (+95%) where SRHT-rSVD’s rises by a third (+34%), since SRHT’s sketch cost does not depend on  $l$ . Error bars are  $\pm 1$  standard deviation over 8 seeds. .... 26
- Figure 10 `randQB_EI`’s block size, drawn from the same runs as Table 19. The middle panel is the trade-off: the method can only stop on a multiple of  $b$ , so  $b = 20$  and  $b = 40$

both return rank 80 where  $b = 5$  returns 67.5 on average. Error and runtime both move in the caller's favour as  $b$  grows, since the cost of a large block is a rank that was not requested, not a worse answer. Error bars are  $\pm 1$  standard deviation over 8 seeds; the zero-width bars at  $b \geq 10$  mean all 8 seeds stopped at the same rank. .... 26

Figure 11 Accuracy vs. rank, Gaussian rSVD,  $n = 13,514$  vs.  $n = 20,000$ . The two curves look the same. .... 27

Figure 12 Power-iteration convergence at  $k = 50$ ,  $n = 13,514$  vs.  $n = 20,000$ . Again the same. .... 28

# Introduction

Randomized algorithms are now a common way to find low-rank approximations of large matrices. The randomized SVD method by Halko, Martinsson, and Tropp [HMT11] is the standard way to compute a rank- $k$  approximation of a matrix. It costs much less than a full SVD, and it remains an active research topic today [TW23]. Some methods try to make the random sketch step even faster. The Subsampled Randomized Hadamard Transform (SRHT) [BG13, Tro11] uses a structured transform instead of a plain random matrix. In theory this needs fewer operations:  $O(n \log n)$  instead of  $O(n\ell)$ . Another method, randQB\_EI [YGL18], builds the same kind of low-rank basis step by step. It picks the rank automatically to reach a target accuracy, instead of asking for a fixed rank up front.

This thesis compares these three methods, Gaussian rSVD, SRHT-rSVD, and randQB\_EI, on matrices that come from partial differential equations (PDEs).

The term “PDE matrix” can mean two different things. It can mean a **discretization** matrix, like a finite-element stiffness matrix. Or it can mean a **boundary-integral kernel** matrix, the kind used in boundary-element or fast-multipole methods. We show with real data (Section 4.1) that only the second kind has low numerical rank. A discretization matrix comes from a local operator, it is sparse, and there is no reason for it to compress well. A boundary-integral kernel matrix is dense, but it is smooth away from the diagonal. This smoothness is exactly why hierarchical/H-matrix methods and the fast multipole method work [GR87, Hac99]. This difference is well known in the numerical linear algebra field, but the two meanings are still easy to conflate in informal use. Showing this difference with real downloaded data, instead of only stating it from theory, is one contribution of this thesis, and it gives the first research question. **RQ0**: do the two kinds of “PDE matrix” really differ in numerical rank, and by how much (Section 4.1)?

Once we have a matrix that compresses well, three more questions remain. **RQ1**: how do the three methods compare on accuracy, runtime, and memory, at the same target rank or tolerance (Section 4.2)? **RQ2**: does SRHT’s better time complexity actually make it faster in practice than Gaussian rSVD (Section 4.3)? **RQ3**: does the rank or power-iteration budget needed for a fixed accuracy change with the problem size (Section 4.6)?

The rest of the thesis is organized as follows. Chapter 2 sets up notation, states the classical results the thesis leans on, explains why random sketching works at all, and gives each of the three algorithms as pseudocode. Chapter 3 describes the matrices, the metrics, and the experimental protocol. Chapter 4 answers RQ0 through RQ3. Chapter 5 gives practical advice and states what this work does not establish, and Chapter 6 lists the directions we think is worth taking next.

**Contributions.** This is an empirical study rather than a theoretical one. Gaussian rSVD, SRHT-rSVD, and randQB\_EI are all existing, published algorithms. What this thesis adds is:

1. A controlled, repeatable comparison of the three methods on a matrix type, boundary-integral kernel matrices, that truly has low rank, at two sizes ( $n = 13,514$  and  $n = 20,000$ ).
2. Power iteration added to SRHT, which the base algorithm does not have. This removes an unfair advantage that Gaussian rSVD would otherwise have, so both methods get the same accuracy boost before they are compared.

3. A clear, data-backed answer to which “PDE matrix” type actually works with randomized low-rank methods. We use real downloaded SuiteSparse data for the negative case (RQ0) and a tested physical construction for the positive case.
4. A robustness check on the RQ2 answer. We compare a plain, uncompiled implementation of the SRHT sketch against an in-place version compiled with Numba, so that an implementation effect can be separated from an algorithmic one, and we count the floating-point operations of both sketch steps to locate where the remaining gap comes from (Section 4.3).

## Chapter 2

# Background

This section fixes notation, states the two classical results the whole thesis leans on (the SVD and the Eckart–Young–Mirsky theorem), explains why a random sketch captures a matrix’s range at all, and then gives each of the three algorithms as pseudocode together with its cost. Section 2.9 explains the property of boundary-integral kernel matrices that makes any of this worth doing at all.

### 2.1 Notation and two classical results

Matrices are capital letters, vectors lower case.  $A^T$  is the transpose,  $\|\cdot\|_F$  the Frobenius norm and  $\|\cdot\|_2$  the spectral norm:

$$\|A\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n a_{ij}^2}, \quad \|A\|_2 = \max_{x \neq 0} \frac{\|Ax\|_2}{\|x\|_2}. \quad (1)$$

**Singular value decomposition.** Every  $A \in \mathbb{R}^{m \times n}$  of rank  $r$  can be written [GV13]

$$A = U \Sigma V^T = \sum_{j=1}^r \sigma_j u_j v_j^T, \quad (2)$$

where  $U \in \mathbb{R}^{m \times r}$  and  $V \in \mathbb{R}^{n \times r}$  have orthonormal columns and  $\Sigma = \text{diag}(\sigma_1, \dots, \sigma_r)$  holds the singular values in decreasing order,  $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r > 0$ . Truncating Equation 2 after  $k$  terms gives the **rank- $k$  truncated SVD**  $A_k = \sum_{j=1}^k \sigma_j u_j v_j^T$ .

**Eckart–Young–Mirsky.**  $A_k$  is the best rank- $k$  approximation of  $A$  in both norms. Eckart and Young proved this for the Frobenius norm [EY36], and Mirsky later extended it to every unitarily invariant norm, the spectral norm included [Mir60]. For any  $B$  with  $\text{rank}(B) \leq k$ ,

$$\|A - B\|_F \geq \|A - A_k\|_F = \left( \sum_{j>k} \sigma_j^2 \right)^{1/2}, \quad \|A - B\|_2 \geq \|A - A_k\|_2 = \sigma_{k+1}. \quad (3)$$

Every “vs. optimal” number reported in Chapter 4 is the ratio of a method’s measured error to the right-hand side of Equation 3. It is the cleanest yardstick available, since it asks not “is this error small?” but “how much worse is this than the best possible answer at the same rank?”

**Numerical rank.** Equation 3 also gives the definition we use when we ask whether a matrix “has low rank”. The **numerical rank at tolerance  $\varepsilon$**  is the smallest  $k$  whose truncation is within a relative  $\varepsilon$ :

$$\text{rank}_\varepsilon(A) = \min \left\{ k : \|A - A_k\|_F \leq \varepsilon \|A\|_F \right\}. \quad (4)$$

A matrix is worth compressing when  $\text{rank}_\varepsilon(A) \ll \min(m, n)$  at a tolerance of practical interest. Section 4.1 measures exactly Equation 4 for two real finite-element matrices and finds it is **not** small.

### 2.2 Symbols used in this thesis

Table 1 collects every symbol that appears more than once, so that any equation in the thesis can be read without hunting backwards for a definition.

| Symbol                     | Meaning                                                                  |
|----------------------------|--------------------------------------------------------------------------|
| $A$                        | the matrix being approximated, of size $m \times n$                      |
| $m, n$                     | number of rows and columns of $A$                                        |
| $\hat{A}$                  | the low-rank approximation returned by a method, $\hat{A} = U\Sigma V^T$ |
| $A_k$                      | the best rank- $k$ approximation, from the truncated SVD (Equation 2)    |
| $\sigma_j$                 | the $j$ th largest singular value of $A$                                 |
| $U, \Sigma, V$             | factors of the SVD: $A = U\Sigma V^T$ (Equation 2)                       |
| $k$                        | target rank requested from a fixed-rank method                           |
| $p$                        | oversampling: extra sketch columns beyond $k$ . We use $p = 10$          |
| $\ell$                     | sketch size, $\ell = k + p$                                              |
| $q$                        | number of power iterations (Equation 8)                                  |
| $b$                        | block size used by randQB_EI (Algorithm 3)                               |
| $\varepsilon$              | accuracy tolerance requested from a fixed-precision method               |
| $\Omega$                   | Gaussian random test matrix, size $n \times \ell$                        |
| $\Theta$                   | the SRHT sketching matrix (Equation 9)                                   |
| $H, D, R$                  | Hadamard matrix, random sign matrix, row-selection matrix in Equation 9  |
| $n_{\text{pad}}$           | $n$ rounded up to a power of two, needed by the FWHT                     |
| $Q$                        | matrix with orthonormal columns approximating the range of $A$           |
| $B$                        | the small factor $B = Q^T A$ in a QB factorization                       |
| $\eta$                     | separation ratio between the two point clouds (Equation 17)              |
| $\ \cdot\ _F, \ \cdot\ _2$ | Frobenius norm and spectral norm                                         |

Table 1: Symbols used throughout the thesis.

## 2.3 Terminology

The terms below are standard in the randomized numerical linear algebra literature, but they are used heavily in this thesis and several of them mean something narrower here than in everyday use.

**Low-rank approximation** a factorization  $\hat{A} = U\Sigma V^T$  in which  $U$  and  $V$  have only  $k \ll \min(m, n)$  columns, so that storing  $\hat{A}$  costs  $O((m+n)k)$  instead of  $O(mn)$ . The goal is to make  $\|A - \hat{A}\|$  small while keeping  $k$  small.

**Numerical rank** the smallest  $k$  at which the truncated SVD is within a relative tolerance of  $A$ , as defined in Equation 4. A matrix can have full **exact** rank and still have small numerical rank; it is the latter that decides whether compression is worth doing.

**Sketch** a small random projection of  $A$ , such as  $Y = A\Omega$ . The columns of  $Y$  are random samples from the range of  $A$ , and the point of a sketch is that a few of them already span nearly the same subspace as the leading singular vectors.

**Range (column space)** the set of vectors  $Ax$  for all  $x$ . Finding a small orthonormal basis  $Q$  for (most of) the range is Stage A of every method here, and is where all the approximation error comes from (Equation 5).

**Oversampling** drawing  $p$  more sketch columns than the target rank  $k$ . The extra columns cost little and make the range estimate far more reliable; the bounds Equation 6 and Equation 7 both improve as  $p$  grows.

**Power iteration** replacing  $A$  by  $(AA^T)^q A$  before sketching, which raises each singular value to the power  $2q + 1$  (Equation 8) and so separates the leading singular directions from the trailing ones. It buys accuracy at the cost of extra passes over  $A$ .

**Fixed-rank method** one that takes  $k$  as input and returns whatever error results. Gaussian rSVD and SRHT-rSVD are fixed-rank.

**Fixed-precision method** one that takes a tolerance  $\varepsilon$  as input and returns whatever rank is needed to reach it. randQB\_EI is fixed-precision. Section 3.3 explains how the two kinds are compared.

**QB factorization** the intermediate form  $A \approx QB$  with  $Q$  orthonormal and  $B = Q^T A$ . All three methods produce one; converting it to an SVD costs only a small SVD of  $B$ .

**Subspace embedding** a random matrix  $\Theta$  that nearly preserves the lengths of all vectors inside a fixed low-dimensional subspace (Equation 11). This is the property that makes a structured sketch such as the SRHT usable in place of a Gaussian one.

**Green's function** the kernel describing the influence of a point source on a point target for a given differential operator. For the Laplace equation in three dimensions it is Equation 14, and its smoothness away from the diagonal is the reason boundary-integral matrices compress.

**Separation ratio** the ratio  $\eta$  of the distance between two point clusters to their radius. Larger  $\eta$  means better-separated clusters and faster singular value decay (Equation 16). Hierarchical matrix methods call a block with large enough  $\eta$  **admissible**.

**Fast Walsh–Hadamard Transform (FWHT)** the  $O(n \log n)$  algorithm for applying the Hadamard matrix Equation 10 without forming it, analogous to the FFT for the Fourier matrix. It is the step that is supposed to make SRHT faster than a dense sketch.

**BLAS and dgemm** the standard dense linear algebra interface, and its double-precision matrix–matrix multiply. Implementations are multi-threaded and heavily tuned, which is why Section 4.3 finds a dense matrix multiply outrunning an asymptotically cheaper transform.

## 2.4 Why a random sketch finds the range

Computing Equation 2 costs  $O(mn \min(m, n))$ , which is out of reach for the matrices we care about here. The randomized approach replaces it with a two-stage scheme [HMT11]; [MT20] is the standard modern survey of the area.

**Stage A** find a tall, thin matrix  $Q$  with orthonormal columns such that  $A \approx QQ^T A$ . That is, find an approximate basis for the range of  $A$ .

**Stage B** form the small matrix  $B = Q^T A$ , factor **that** cheaply, and push the factors back through  $Q$ .

Stage B is exact given  $Q$ : if  $B = Q^T A$  has SVD  $B = \tilde{U}\Sigma V^T$  then  $Q\tilde{U}\Sigma V^T = QQ^T A$ , so all of the approximation error comes from Stage A:

$$\|A - Q\Sigma V^T \tilde{U}^T\|_\xi = \|A - QQ^T A\|_\xi, \quad \xi \in \{F, 2\}. \quad (5)$$

The whole game is therefore Stage A. The randomized trick is to probe the range with random vectors. That random projections preserve geometry well enough for this to work goes back to Johnson and Lindenstrauss [JL84], and Sarlós first applied the idea to low-rank matrix approximation [Sar06]. Draw  $\ell = k + p$  random vectors  $\omega_1, \dots, \omega_\ell$  and form  $y_i = A\omega_i$ . Each  $y_i$  is a random sample from the range of  $A$ , and because  $A$  stretches most along its leading singular directions, the  $y_i$  line up with those directions. With  $\ell$  samples slightly larger than  $k$ , the span

of  $y_1, \dots, y_\ell$  captures the leading rank- $k$  subspace with high probability. The extra  $p$  vectors are the **oversampling**, and they matter: with  $p = 0$  the estimate is unreliable, while a small fixed  $p$  (we use  $p = 10$ ) makes it robust.

Halko, Martinsson and Tropp make this precise. With a Gaussian test matrix, target rank  $k \geq 2$  and oversampling  $p \geq 2$ , the expected errors are bounded by [HMT11]

$$\mathbb{E} \|A - QQ^T A\|_F \leq \left(1 + \frac{k}{p-1}\right)^{1/2} \left(\sum_{j>k} \sigma_j^2\right)^{1/2}, \quad (6)$$

$$\mathbb{E} \|A - QQ^T A\|_2 \leq \left(1 + \sqrt{\frac{k}{p-1}}\right) \sigma_{k+1} + \frac{e\sqrt{k+p}}{p} \left(\sum_{j>k} \sigma_j^2\right)^{1/2}. \quad (7)$$

Compare these with Equation 3. The Frobenius bound in Equation 6 is the optimal error times a factor that depends only on  $k$  and  $p$ . The spectral bound in Equation 7 is the optimal error  $\sigma_{k+1}$  times a similar factor, **plus** an extra term proportional to the whole Frobenius tail. That additive term has no counterpart in Equation 6, and it is the reason a randomized sketch should be expected to look relatively worse when scored in the spectral norm than in the Frobenius norm. Section 4.2 measures this effect and finds it clearly.

**Power iteration.** This device is due to Rokhlin, Szlam and Tygert [RST09]. When the singular values decay slowly, the trailing terms in Equation 6 and Equation 7 are large and the sketch is poor. The fix is to sketch a matrix with the same singular vectors but faster-decaying singular values. For an integer  $q$ , set

$$A_{(q)} = (AA^T)^q A, \quad \text{so that} \quad \sigma_j(A_{(q)}) = \sigma_j(A)^{2q+1}. \quad (8)$$

Sketching  $A_{(q)}$  instead of  $A$  leaves the singular subspaces untouched but raises every ratio  $\sigma_{k+1}/\sigma_k$  to the power  $2q + 1$ , pushing the trailing terms down sharply. Each extra  $q$  costs two more passes over  $A$ . In floating point,  $A_{(q)}$  must never be formed explicitly: the small singular values underflow. Instead one alternates multiplication and re-orthonormalization, which is what step 4 of Algorithm 1 does.

## 2.5 Gaussian randomized SVD

The baseline method applies the above directly with a Gaussian test matrix.

Algorithm 1: Gaussian randomized SVD [HMT11]

---

**in:**  $A \in \mathbb{R}^{m \times n}$ , target rank  $k$ , oversampling  $p$ , power iterations  $q$

1.  $\Omega \leftarrow$  Gaussian random matrix of size  $n \times (k + p)$
2.  $Y \leftarrow A\Omega$  ▷ sketch of the column space
3.  $Q \leftarrow \text{qr}(Y)$  ▷ orthonormal basis, Stage A
4. **for**  $i = 1, \dots, q$  **do**  $\tilde{Q} \leftarrow \text{qr}(A^T Q)$ ,  $Q \leftarrow \text{qr}(A\tilde{Q})$
5.  $B \leftarrow Q^T A$  ▷ Stage B
6.  $[\tilde{U}, \Sigma, V^T] \leftarrow \text{svd}(B)$

**out:**  $U = Q\tilde{U}$ , truncated with  $\Sigma, V$  to  $k$  columns

---

Cost is dominated by step 2, a dense  $m \times n$  by  $n \times \ell$  product costing  $O(mn\ell)$  operations, plus  $O(m\ell^2)$  for the QR and  $O(mn\ell)$  again for step 5. Power iteration multiplies the pass count by roughly  $2q + 1$ .

## 2.6 SRHT randomized SVD

Step 2 of Algorithm 1 is the expensive part, and it treats  $\Omega$  as an unstructured dense matrix. The Subsampled Randomized Hadamard Transform replaces  $\Omega$  with a structured matrix that can be applied in  $O(n \log n)$  time per vector rather than  $O(n\ell)$ . The construction is due to Ailon and Chazelle, who introduced it as a fast Johnson–Lindenstrauss transform [AC09]; its use for low-rank approximation was analyzed by Boutsidis and Gittens [BG13] and by Tropp [Tro11]. Define

$$\Theta = \sqrt{n/\ell} RHD \in \mathbb{R}^{\ell \times n}, \quad (9)$$

where  $D \in \mathbb{R}^{n \times n}$  is diagonal with independent random signs  $\pm 1$ ,  $H \in \mathbb{R}^{n \times n}$  is the normalized Walsh–Hadamard matrix, and  $R \in \mathbb{R}^{\ell \times n}$  selects  $\ell$  rows uniformly at random. The Walsh–Hadamard matrix is defined recursively for  $n$  a power of two:

$$H_1 = [1], \quad H_{2m} = \frac{1}{\sqrt{2}} \begin{pmatrix} H_m & H_m \\ H_m & -H_m \end{pmatrix}. \quad (10)$$

The recursion in Equation 10 is what makes  $Hx$  cheap: it never needs to be formed. The Fast Walsh–Hadamard Transform [FA76] evaluates  $Hx$  with  $\log_2 n$  passes of additions and subtractions,  $O(n \log n)$  work in total, in the same way the FFT exploits the structure of the Fourier matrix. When  $n$  is not a power of two the input is zero-padded up to the next one, which is why our implementation works with  $n_{\text{pad}} = 2^{\lceil \log_2 n \rceil}$ .

The random signs  $D$  matter. Without them, a matrix whose mass sits in a few coordinates would survive row sampling badly. Multiplying by  $D$  and then  $H$  spreads that mass evenly across all coordinates, so that sampling  $\ell$  rows afterwards loses little. Tropp [Tro11] shows that  $\Theta$  is, with high probability, a **subspace embedding**: for a fixed  $k$ -dimensional subspace with orthonormal basis  $V$ , and  $\ell$  large enough (up to logarithmic factors,  $\ell = O(k \log k)$ ),

$$(1 - \delta) \|Vx\|_2 \leq \|\Theta Vx\|_2 \leq (1 + \delta) \|Vx\|_2 \quad \text{for all } x. \quad (11)$$

Equation 11 says  $\Theta$  distorts lengths inside that subspace only slightly, which is exactly what Stage A needs. Boutsidis and Gittens [BG13] turn this into low-rank approximation bounds comparable to Equation 6 and Equation 7.

Algorithm 2: SRHT randomized SVD, with the power-iteration step added here

---

**in:**  $A \in \mathbb{R}^{m \times n}$ , target rank  $k$ , oversampling  $p$ , power iterations  $q$

1.  $D \leftarrow$  diagonal of random signs;  $R \leftarrow \ell = k + p$  row indices sampled uniformly
2.  $Y \leftarrow A(\sqrt{n/\ell} RHD)^T$  ▷ via FWHT,  $O(mn \log n)$
3.  $Q \leftarrow \text{qr}(Y)$
4. **for**  $i = 1, \dots, q$  **do**  $\tilde{Q} \leftarrow \text{qr}(A^T Q)$ ,  $Q \leftarrow \text{qr}(A\tilde{Q})$  ▷ our addition
5.  $B \leftarrow Q^T A$
6.  $[\tilde{U}, \Sigma, V^T] \leftarrow \text{svd}(B)$

**out:**  $U = Q\tilde{U}$ , truncated with  $\Sigma, V$  to  $k$  columns

---

Step 4 is not part of the published SRHT algorithm. We add it so that both methods have access to the same accuracy tool, and we verify in Section 4.3 that it reproduces Gaussian rSVD’s accuracy exactly at matched  $q$ . Note what the comparison then looks like on paper: step 2 drops from  $O(mn\ell)$  to  $O(mn \log n)$ , and every other step is identical to Algorithm 1. Whenever  $\ell >$

$\log n$ , SRHT should win. At  $n = 13,514$  and  $k = 200$  that predicts a sizeable advantage, since  $\ell = 210$  against  $\log_2 n_{\text{pad}} = 14$ . Section 4.3 tests whether this happens.

## 2.7 randQB\_EI: fixed precision instead of fixed rank

Both methods above need  $k$  chosen in advance. Often the real requirement is the opposite: reach a target accuracy  $\varepsilon$  and use whatever rank that takes. randQB\_EI [YGL18] builds  $Q$  and  $B = Q^T A$  a block of  $b$  columns at a time, and stops when a running error estimate drops below tolerance.

The reason this is cheap rests on one identity. If  $Q$  has orthonormal columns and  $B = Q^T A$ , then

$$\|A - QB\|_F^2 = \|A\|_F^2 - \|B\|_F^2. \quad (12)$$

To see it, expand the square as a trace and use  $Q^T Q = I$ :

$$\|A - QB\|_F^2 = \|A\|_F^2 - 2 \operatorname{tr}(A^T QB) + \|QB\|_F^2. \quad (13)$$

Now  $\operatorname{tr}(A^T QB) = \operatorname{tr}(B^T B) = \|B\|_F^2$  because  $B = Q^T A$ , and  $\|QB\|_F = \|B\|_F$  because  $Q$  has orthonormal columns. Both of the last two terms collapse to  $\|B\|_F^2$ , leaving Equation 12.

Equation 12 is the whole trick. The error after each block can be tracked by subtracting the Frobenius mass of the new rows of  $B$  from a running total, a few hundred floating-point operations, instead of forming the residual  $A - QB$ , which would cost as much as the factorization itself and, for a large matrix, would not fit into memory. Yu, Gu and Li call this the **error indicator**, the “EI” in the name.

Algorithm 3: randQB\_EI [YGL18]

---

**in:**  $A \in \mathbb{R}^{m \times n}$ , tolerance  $\varepsilon$ , block size  $b$ , rank cap  $k_{\max}$

1.  $Q \leftarrow [], B \leftarrow [], E \leftarrow \|A\|_F^2$  ▷ running error, see Equation 12
2. **while**  $E > \varepsilon^2 \|A\|_F^2$  **do**
3.    $\Omega_i \leftarrow$  Gaussian random matrix of size  $n \times b$
4.    $Y_i \leftarrow A\Omega_i - Q(B\Omega_i)$  ▷ project out what is already captured
5.    $Q_i \leftarrow \operatorname{qr}(Y_i)$ ; re-orthogonalize  $Q_i$  against  $Q$  and re-factor
6.    $B_i \leftarrow Q_i^T A$
7.    $Q \leftarrow [Q, Q_i]$ ,  $B \leftarrow \begin{pmatrix} B \\ B_i \end{pmatrix}$ ,  $E \leftarrow E - \|B_i\|_F^2$
8.   **if** number of columns  $\geq k_{\max}$ : stop and report failure

**out:**  $Q, B$  with  $\|A - QB\|_F \leq \varepsilon \|A\|_F$ ; convert to SVD via  $\operatorname{svd}(B)$

---

The price of choosing the rank automatically is that each block carries no oversampling margin of its own beyond  $b$ . At a matched rank, randQB\_EI should therefore be slightly less accurate than a fixed-rank method that spends  $p$  extra columns, and Section 4.2 measures that penalty.

## 2.8 Cost summary

Table 2 collects the leading costs. It is worth being explicit that these are operation counts, not predictions of runtime: a count says nothing about how fast a machine retires those operations, and Section 4.3 is in large part a demonstration of that distinction.

| Method        | Sketch step | Rest                  | Input    |
|---------------|-------------|-----------------------|----------|
| Gaussian rSVD | $O(mn\ell)$ | $O(mn\ell + m\ell^2)$ | rank $k$ |

| Method    | Sketch step               | Rest                  | Input                   |
|-----------|---------------------------|-----------------------|-------------------------|
| SRHT-rSVD | $O(mn \log n)$            | $O(mn\ell + m\ell^2)$ | rank $k$                |
| randQB_EI | $O(mnk)$ total, in blocks | included              | tolerance $\varepsilon$ |

Table 2: Leading operation counts, with  $\ell = k + p$ . Power iteration multiplies the pass count by about  $2q + 1$  for the first two methods. Only the sketch step differs between Gaussian rSVD and SRHT-rSVD; that single column is what Section 4.3 puts to the test.

## 2.9 Why boundary-integral kernel matrices compress well

A matrix has low numerical rank in the sense of Equation 4 when its singular values drop off fast. Whether that happens depends on where the matrix comes from.

A finite-element (FEM) stiffness matrix comes from discretizing a differential operator, such as the Laplacian. This kind of operator is **local**: each row of the matrix only involves a few nearby mesh points, and most entries are exactly zero. There is no reason for such a matrix to have low numerical rank, and Section 4.1 confirms this on two real matrices.

A boundary-integral kernel matrix is different. Its entries sample a **Green’s function** between two sets of points. For the Laplace equation in three dimensions the relevant kernel is

$$K(x, y) = \frac{1}{4\pi\|x - y\|_2}, \quad (14)$$

and the matrix is  $A_{ij} = K(x_i, y_j)$  for targets  $x_i$  in one cluster and sources  $y_j$  in another. Suppose the source cluster has radius  $r$  and is centred at  $y_0$ , and the target cluster is a distance  $R$  away, with

$$\eta = \frac{R}{r} > 1 \quad (\text{the separation ratio}) . \quad (15)$$

Expanding Equation 14 about  $y_0$  in spherical harmonics (the multipole expansion behind Rokhlin’s fast integral-equation solver [Rok85] and the fast multipole method [GR87]) gives a **separable** approximation, a sum of products of a function of  $x$  alone with a function of  $y$  alone:

$$K(x, y) \approx \sum_{t=1}^P \varphi_t(x) \psi_t(y), \quad \text{with error } O(\eta^{-P}). \quad (16)$$

A sum of  $P$  separable terms is exactly a rank- $P$  matrix. So Equation 16 says the matrix block is within  $O(\eta^{-P})$  of a rank- $P$  matrix: the singular values decay geometrically, at a rate set by  $\eta$ . Hierarchical (H-matrix) methods [Hac99, Beb08] build on the same fact, splitting a matrix into blocks and compressing every block whose two point sets are far enough apart to make  $\eta$  comfortably larger than one, while leaving near-diagonal blocks dense.

Two consequences matter for this thesis. First, this is the mechanism that makes low-rank methods worth applying at all, and it is a property of boundary-integral operators, not of “PDE matrices” generally. Second, the rate in Equation 16 depends on  $\eta$ , a purely geometric quantity, and not on how many points discretize each cluster. Refining the discretization adds rows and columns without changing  $\eta$ , so the singular value decay should be unchanged. That is a testable prediction, and Section 4.6 tests it.

## Chapter 3

# Methods

### 3.1 Matrices

**Real FEM stiffness matrices (the negative case, RQ0).** We use two matrices from the SuiteSparse Matrix Collection [DH11]: `pde900` ( $n = 900$ , Bai group) and `poisson3Da` ( $n = 13,514$ , FEMLAB group). Both are downloaded, not built by us. For `pde900`, the ground-truth SVD comes from a direct dense SVD, which is cheap at this size. For `poisson3Da`, we use SuiteSparse’s own precomputed SVD file instead of computing one ourselves.

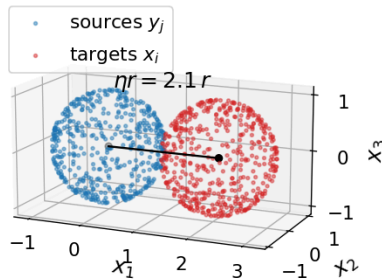
**Boundary-integral kernel matrix (the main case, RQ1–RQ3).** This is the matrix that Section 2.9 predicts should compress, and it is the main test case of thesis. We build it by sampling the Laplace kernel Equation 14 between two point clouds. Source points  $y_1, \dots, y_n$  are drawn uniformly on the surface of a sphere of radius  $r$  centred at the origin, and target points  $x_1, \dots, x_m$  uniformly on a sphere of the same radius centred at distance  $\eta r$  along the first axis:

$$y_j \in \{y : \|y\|_2 = r\}, \quad x_i \in \{x : \|x - \eta r e_1\|_2 = r\}, \quad A_{ij} = \frac{1}{4\pi \|x_i - y_j\|_2}. \quad (17)$$

Uniform sampling on the sphere is done by drawing a standard Gaussian vector in  $\mathbb{R}^3$  and normalizing it, which gives a rotationally invariant distribution. The separation ratio is  $\eta = 2.1$  throughout, so the two spheres are disjoint ( $\eta > 2$ ) but not far apart. This choice is deliberate: pushing  $\eta$  higher would make the decay in Equation 16 so fast that every method would look equally good at tiny rank, and the comparison would say nothing. We verified at both  $n = 900$  and  $n = 13,514$  that  $\eta = 2.1$  gives a decay comparable to the synthetic exponential-decay control below: fast, but slow enough that the rank sweep has somewhere to go.

Figure 1 draws the construction, and shows what that choice costs. The right panel sweeps  $\eta$  at  $n = 900$ : at  $\eta = 5$  the matrix is numerically rank-36 and every method would reach machine precision immediately, at  $\eta = 3$  rank-69, while at our  $\eta = 2.1$  the error is still falling at  $k = 100$ . So  $\eta = 2.1$  is the hardest of the three, and the only one of them on which a rank sweep can distinguish anything.

Sources and targets on two spheres of radius  $r$ , centres  $\eta r$  apart (500 of each shown)



Wider separation means faster decay (the  $\eta^{-P}$  rate of the multipole expansion),  $n = 900$

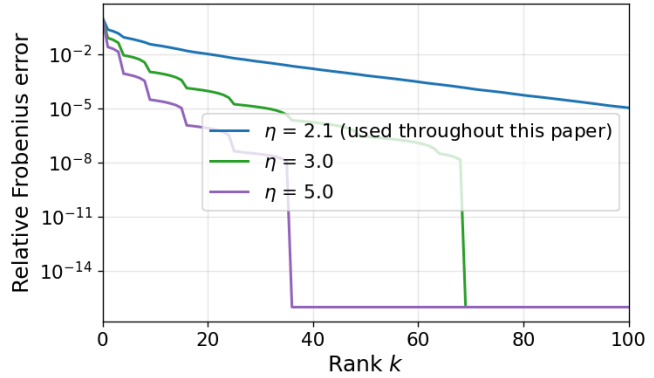


Figure 1: Left: the construction of Equation 17, sources on one sphere, targets on another of the same radius, centres  $\eta r$  apart (500 points of each shown; the experiments use thousands). Right: best rank- $k$  error at three separation ratios,  $n = 900$ . A wider gap gives faster decay, exactly as the  $O(\eta^{-P})$  rate of Equation 16 predicts. The flat floors at  $\eta = 3$  and  $\eta = 5$  are machine precision: past that rank there is nothing left to approximate.

Note that Equation 17 fixes  $\eta$  independently of  $n$ . By Equation 16 the singular value decay is then also independent of  $n$ , which is the prediction Section 4.6 tests by rebuilding the same geometry at  $n = 20,000$ .

**Synthetic control spectra.** As a sanity check, we also test on synthetic matrices with a known, exact singular-value spectrum: exponential decay ( $\sigma_j = e^{-0.1j}$ ), polynomial decay ( $\sigma_j = j^{-1}$ ), and a noisy, near-flat spectrum. Each is built as  $A = U \text{diag}(\sigma)V^T$ , with  $U$  and  $V$  random orthogonal matrices. Since we choose the singular values ourselves, the ground truth is exact and free, no SVD needed. These matrices are  $n = 2000$  and dense. They only serve as a check that the three algorithms agree with theory, before we trust their results on the boundary-integral matrix (see the “matches the synthetic control test” paragraph in Section 4.2).

Table 3 collects every matrix used anywhere in this thesis, with how its ground truth was obtained. The last column is worth reading down: only the two smallest matrices admit a full dense SVD at all, which is the practical reason randomized methods are of interest in the first place.

| Matrix               | $n$    | Storage        | Origin                 | Ground truth           |
|----------------------|--------|----------------|------------------------|------------------------|
| pde900               | 900    | sparse, 0.54%  | SuiteSparse, Bai       | full dense SVD         |
| poisson3Da           | 13,514 | sparse, 0.19%  | SuiteSparse, FEMLAB    | SuiteSparse’s own SVD  |
| BIE kernel           | 900    | dense, 6 MB    | built, Equation 17     | full dense SVD         |
| BIE kernel           | 13,514 | dense, 1.46 GB | built, Equation 17     | top 250 via svds       |
| BIE kernel           | 20,000 | dense, 3.20 GB | built, Equation 17     | top 250 via svds       |
| Synthetic $\times 3$ | 2000   | dense, 32 MB   | built, chosen spectrum | spectrum known exactly |

Table 3: Every matrix used in this thesis. The two pde900/poisson3Da rows are downloaded real-world data and carry the RQ0 negative result; the boundary-integral rows are the main test case; the synthetic row is the control layer. Densities are nonzeros as a fraction of  $n^2$  (4,380 and 352,762 nonzeros); dense sizes are float64 storage. The synthetic matrices are three separate spectra (exponential, polynomial, near-flat) sharing one size.

### 3.2 Metrics

For every trial we record:

**Relative Frobenius error**  $\|A - \hat{A}\|_F / \|A\|_F$ , where  $\hat{A} = U\Sigma V^T$  is the returned factorization.

**Relative spectral error**  $\|A - \hat{A}\|_2 / \|A\|_2$ .

**Runtime** wall-clock time around the algorithm call only, not counting I/O or metric computation.

**Peak memory** via Python’s tracemalloc, measured over the same interval. This tracks Python-level allocation, not the operating system’s view of process memory. We state this once here instead of repeating it at every result.

Both error metrics need care at these sizes. The naive route, building  $\hat{A}$ , subtracting, and taking a norm, is never used here, because  $\hat{A}$  is dense even when  $A$  is sparse. For a matrix the size of SuiteSparse’s poisson3Db ( $85,623 \times 85,623$ , not used in this thesis) a dense residual would need tens of gigabytes. Instead both errors are computed from the factors directly.

For the Frobenius error, expanding the square and using the orthonormality of  $U$  and  $V$  gives an expression in which the only object touching  $A$  is a thin product:

$$\|A - \hat{A}\|_F^2 = \|A\|_F^2 - 2 \sum_{j=1}^k \sigma_j u_j^T A v_j + \sum_{j=1}^k \sigma_j^2. \quad (18)$$

Computing Equation 18 costs one  $AV$  product of size  $m \times k$  plus  $O(mk)$  work, and  $\|A\|_F^2$  is available in one pass (for sparse  $A$ , from its stored entries). No  $m \times n$  intermediate is ever formed.

For the spectral error we never form the residual either. We give the residual to a sparse eigensolver as an **implicit operator** defined only by how it multiplies vectors,

$$x \mapsto Ax - U(\Sigma(V^T x)), \quad z \mapsto A^T z - V(\Sigma(U^T z)), \quad (19)$$

and ask for its single largest singular value. Every step in Equation 19 is a matrix-vector product or a thin product, so the memory cost is  $O(m + n + k)$  beyond  $A$  itself.

Every configuration, a given rank  $k$  or a given tolerance  $\varepsilon$ , is run with 12 independent random seeds, and we report the mean and standard deviation across those 12 runs. Seeds are fixed integers 0, 1, ..., 11 so that any run reproduces exactly.

### 3.3 Comparing a fixed-rank method against a fixed-precision method

Gaussian rSVD and SRHT-rSVD take a rank  $k$  and report an error. randQB\_EI takes a tolerance  $\varepsilon$  and reports a rank. To put all three on the same plot (error against rank), we do the following. For randQB\_EI, we sweep a list of  $\varepsilon$  values and record the rank it lands on for each one. We calibrate this list against the matrix’s own measured spectrum first; reusing a list of  $\varepsilon$  values from a different matrix does not work, since the same  $\varepsilon$  can map to very different ranks depending on the spectrum. For the “vs. optimal” column in Table 8, we look up what the exact SVD’s error would be at that same rank (the Eckart–Young optimal), from the matrix’s own ground-truth spectrum. This lookup happens after the fact, when we build the plots and tables. It does not change how randQB\_EI itself runs.

### 3.4 Ground truth for large matrices

At  $n = 900$ , a full dense SVD is cheap and gives an exact ground truth for every rank. At  $n = 13,514$  and  $n = 20,000$ , a full dense SVD takes too long (we measure 637 seconds at  $n = 13,514$  in Table 17) and is not needed, since we only test ranks up to 200. Instead we compute the top 250 singular values with `scipy.sparse.linalg.svds` (one to two minutes). This is exact for every rank we test: the Eckart–Young error at rank  $k$  only depends on the singular values **below** rank  $k$ , and all of those sit inside the top 250 we compute.

### 3.5 The SRHT power-iteration extension

The base SRHT algorithm, as published, has no power-iteration step. We add one, using the same scheme Gaussian rSVD already uses (Algorithm 1); the result is step 4 of Algorithm 2. After the SRHT sketch gives an orthonormal basis  $Q$ , we repeat  $q$  times:

$$Q_t = \text{orth}(A^T Q), \quad Q = \text{orth}(A Q_t) \quad (20)$$

using a QR decomposition for  $\text{orth}(\cdot)$  at each step, to keep the numbers stable. This is the same scheme Gaussian rSVD uses, just applied to the SRHT-sketched basis instead of the Gaussian-sketched one. We check this against Gaussian rSVD’s power iteration directly: at matched  $q$ , the two methods should reach the same accuracy, and Section 4.3 confirms they do, to six decimal places.

For large or sparse inputs, our SRHT code also accepts a `sketch_batch_size` argument that runs the Hadamard transform in row batches instead of all at once. This bounds the memory the sketch step uses, at no real cost in runtime (Section 4.3) – we use it at  $n = 20,000$  to keep the sketch step’s memory well within what this machine has available.

### 3.6 Tools, software and reproducibility

Table 4 lists everything used to produce the results in this thesis. All three algorithms are our own implementations in Python, using NumPy [Har+20] and SciPy [Vir+20] for the dense linear algebra, Numba [LPS15] for the compiled transform, and Matplotlib [Hun07] for the figures. Dense products, QR factorizations and small SVDs therefore go through the same OpenBLAS 0.3.30 backend for every method, so no method is advantaged by calling a better library than another. The one exception is the Fast Walsh–Hadamard Transform, which is not a BLAS operation and has no library equivalent; it is our own code, compiled with Numba. Section 4.3 argues that this asymmetry is unavoidable rather than unfair, since there is no `dgemm` for a Hadamard transform, but readers should know it is there.

Timing and memory results depend on the machine, and RQ2’s conclusion depends on them in particular, so we state the platform: an AMD EPYC 7763 with 2 physical cores (4 hardware threads) and 15 GB of RAM, running Python 3.13 with NumPy 2.3.5, SciPy 1.16.3 and Numba 0.67.0. OpenBLAS uses all available threads; our Hadamard transform is single-threaded. On this machine a well-tuned double-precision `dgemm` reaches roughly 80–90 GFLOP/s, which is close to the theoretical peak of two AVX2 cores, so the BLAS baseline in Section 4.3 should be read as near the best this hardware can do rather than as a loosely optimized reference.

| Tool                | Version | Used for                                                       |
|---------------------|---------|----------------------------------------------------------------|
| Python              | 3.13    | all experiment code                                            |
| NumPy               | 2.3.5   | dense linear algebra, random number generation                 |
| SciPy               | 1.16.3  | sparse matrices, svds truncated SVD, <code>.mat</code> loading |
| Numba               | 0.67.0  | JIT compilation of the Fast Walsh–Hadamard Transform           |
| OpenBLAS            | 0.3.30  | the BLAS/LAPACK backend behind NumPy and SciPy                 |
| pandas              | 2.3.3   | aggregating trial results into the tables below                |
| Matplotlib          | 3.10.8  | all figures                                                    |
| pytest              | 9.0.2   | the correctness test suite (58 tests)                          |
| Jupyter / nbconvert | 7.16.6  | running the experiment notebooks                               |
| uv                  | 0.12    | creating the environment from pinned versions                  |
| Typst               | 0.15.1  | typesetting this document                                      |
| SuiteSparse         | –       | source of the two real FEM matrices                            |

Table 4: Software used. Versions are pinned in `requirements.txt` so the environment can be rebuilt exactly.

Every experiment is driven by a script that fixes all random seeds, so re-running it reproduces the tables in this thesis exactly. Table 5 says which script produces which result, and how many repetitions stand behind it. The main sweeps use 12 seeds; the parameter sweeps use 8; the three measurements that involve a full dense SVD or a 637-second baseline use fewer, because the cost of a repetition there is minutes rather than milliseconds. Correctness of the three implementations is checked by a 58-test suite, which includes a direct comparison of our Fast Walsh–Hadamard Transform against an explicitly constructed Hadamard matrix (Appendix

A), regression tests on sparse input, and a check that SRHT-rSVD with power iteration matches Gaussian rSVD at matched  $q$ .

| Script                                                   | Repeats          | Produces                                                                              |
|----------------------------------------------------------|------------------|---------------------------------------------------------------------------------------|
| benchmark_synthetic.py and the three algorithm notebooks | 12, 8 for sweeps | Table 9,<br>Table 11,<br>Table 18,<br>Table 19,<br>Figure 9,<br>Figure 10             |
| pde900_pilot.ipynb                                       | 12               | Table 7                                                                               |
| bie_kernel_main_sweep_script.py                          | 12               | Table 8,<br>Table 12,<br>Table 10,<br>Table 16,<br>Figure 4,<br>Figure 5,<br>Figure 7 |
| bie_kernel_n20000_scaling_script.py                      | 12               | Table 20,<br>Figure 11,<br>Figure 12                                                  |
| sketch_throughput.py                                     | 3                | Table 15,<br>Figure 6                                                                 |
| fwht_implementation_cost.py                              | 3                | Table 13                                                                              |
| exact_svd_speedup.py                                     | 5 small, 3 large | Table 17,<br>Figure 8                                                                 |
| scripts/make_paper_figures.py                            | –                | Figure 1,<br>Figure 3,<br>and the<br>plotting for<br>the figures<br>above             |

Table 5: Which script produces which result. Scripts live in `src/experiments/` unless a path is given. The last one draws figures from CSVs the others write; it computes nothing of its own except the geometry panels, so a figure can always be redrawn without repeating an experiment.

# Results

This section answers the four research questions in turn. Unless we say otherwise, every rank-sweep test uses oversampling  $p = 10$  and 12 random seeds. Error bars in the figures show  $\pm 1$  standard deviation across seeds.

## 4.1 FEM stiffness matrices resist low-rank approximation

Before we test the three algorithms on a matrix type that should compress well, we first checked whether real finite-element (FEM) matrices, often just called “PDE matrices,” actually have low numerical rank. They do not. Table 6 shows two real matrices from the SuiteSparse Matrix Collection [DH11].

| Matrix     | Size $n$ | Rank needed          | Result                                |
|------------|----------|----------------------|---------------------------------------|
| pde900     | 900      | 772 / 900 (86%)      | for relative Frobenius error $< 10\%$ |
| poisson3Da | 13,514   | 5,000 / 13,514 (37%) | still leaves 57% relative error       |

Table 6: Rank needed for a modest accuracy target, on two real FEM stiffness matrices. Neither one compresses well at any practical rank. `poisson3Da`’s exact answer comes from SuiteSparse’s own precomputed SVD; `pde900`’s comes from a direct dense SVD (possible at  $n = 900$ ).

Figure 2 shows the same fact directly. Plotted on a common vertical scale, both FEM matrices are almost flat: after 250 singular values `pde900` has dropped only to about half of  $\sigma_1$ , and `poisson3Da` barely moves at all. The boundary-integral kernel matrix, on the same axes, falls through six orders of magnitude over the same range. This is the difference between a matrix that compresses and one that does not, and it is visible without any further analysis.

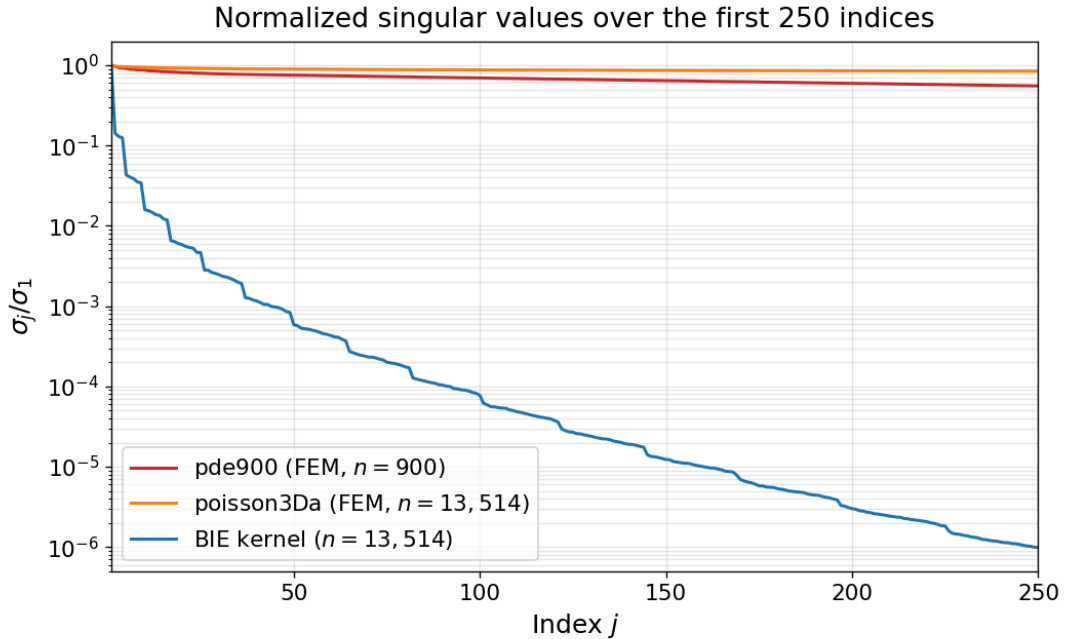


Figure 2: Normalized singular values  $\sigma_j/\sigma_1$  for the two real FEM matrices and the boundary-integral kernel matrix, over the first 250 indices. The FEM spectra are essentially flat; the boundary-integral spectrum decays geometrically, as Equation 16 predicts. `pde900` is from a full dense SVD, `poisson3Da` from SuiteSparse’s precomputed SVD, and the kernel matrix from the top 250 singular values.

Figure 2 plots singular values; Figure 3 plots the thing a user actually cares about, which is the error left over after keeping  $k$  of them. The two FEM matrices barely move: at  $k = 250$ , pde900 is still at 0.63 relative error and poisson3Da at 0.97, that is, throwing away everything past rank 250 costs poisson3Da almost all of its content. The boundary-integral matrix is at  $4.1 \times 10^{-6}$  by the same rank. The dotted line marks the 10% target from Table 6, which the kernel matrix crosses at rank 4 and neither FEM matrix crosses anywhere on this axis.

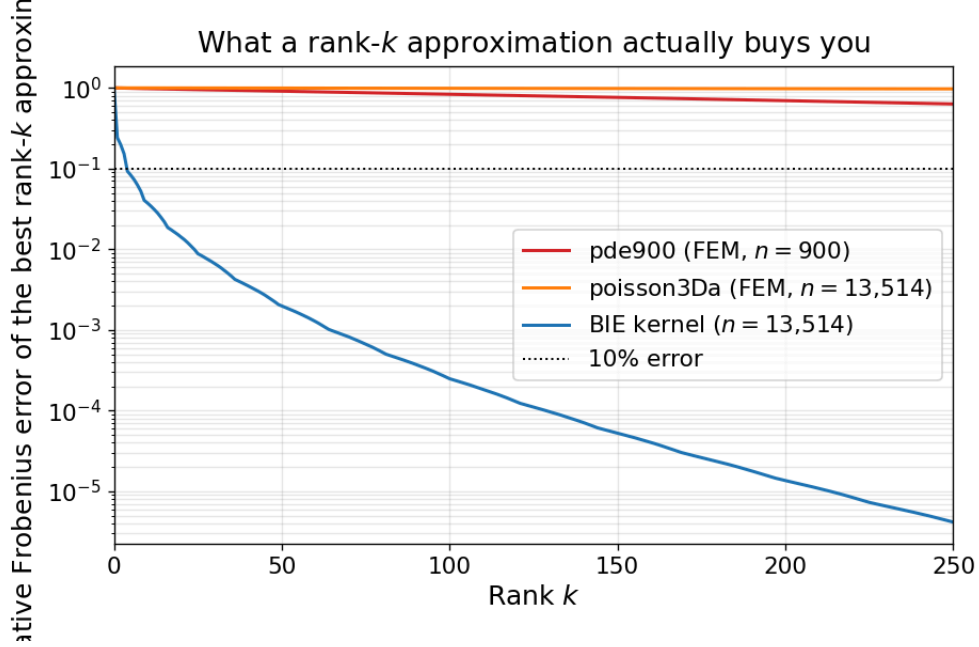


Figure 3: Relative Frobenius error of the best rank- $k$  approximation, the Eckart–Young optimum, so no algorithm can do better than these curves. Computed as  $\sqrt{(\|A\|_F^2 - \sum_{j \leq k} \sigma_j^2)} / \|A\|_F$ , which needs only the top singular values and never forms a residual. Same three matrices and same sources as Figure 2.

This matches the idea behind hierarchical/H-matrix methods and the fast multipole method [GR87, Hac99]. A discretized differential operator is a local, sparse operator, and there is no reason it should have low rank. A **boundary-integral** kernel matrix is different: it comes from a smooth Green’s function measured between two sets of well-separated points, and it decays fast because of that smoothness. This is why a boundary-integral kernel matrix, not a FEM stiffness matrix, is used as the main test case for RQ1 through RQ3 below.

**The randomized methods confirm this directly, on real sparse data.** Table 7 runs all three algorithms on pde900 itself, a genuine downloaded sparse matrix (4380 nonzeros, 0.54% density), not the constructed matrix used everywhere else in this thesis. Table 1 already shows the exact SVD needs rank 772 of 900 for 10% error, and the randomized methods land on the same conclusion without ever forming a dense residual. At  $k = 20$  the relative error is still 0.98, barely better than returning nothing at all, and even at  $k = 800$ , effectively the whole matrix, it only reaches 0.17–0.18.

| Algorithm     | $k$ | Rel. Frob. error | Runtime (s) | Peak mem. (MB) |
|---------------|-----|------------------|-------------|----------------|
| Gaussian rSVD | 20  | 0.980            | 0.042       | 0.8            |
| SRHT-rSVD     | 20  | 0.980            | 0.060       | 2.2            |
| Gaussian rSVD | 200 | 0.805            | 0.188       | 6.2            |
| SRHT-rSVD     | 200 | 0.805            | 0.253       | 9.9            |

| Algorithm     | $k$ | Rel. Frob. error | Runtime (s) | Peak mem. (MB) |
|---------------|-----|------------------|-------------|----------------|
| Gaussian rSVD | 800 | 0.173            | 0.491       | 27.9           |
| SRHT-rSVD     | 800 | 0.180            | 0.526       | 53.3           |

Table 7: Gaussian rSVD and SRHT-rSVD run directly on the sparse pde900 matrix, 12-seed means. Even at  $k = 800$  out of 900, almost no compression at all, the error stays above 0.17, confirming Table 6’s rank-needed number with the actual randomized algorithms rather than only the exact SVD.

This also validates something the boundary-integral matrix cannot, because it is dense by construction: that both algorithms’ sparse-input code paths give correct results, and that the SRHT power-iteration extension (Section 2.6) still matches Gaussian rSVD on a real sparse matrix. At  $q = 3$  the two report 0.9169 and 0.9169 (to four decimal places, the raw values differ by  $4 \times 10^{-5}$ , which is consistent with the different order of floating-point operations a sparse matmul takes compared to a dense one). Power iteration was already checked on the constructed boundary-integral matrix in Section 4.3, and agreeing again here, on real downloaded data with a completely different matrix structure, gives a stronger check than either result alone.

## 4.2 Accuracy comparison on boundary-integral kernel matrices

The main matrix here is a dense  $13,514 \times 13,514$  boundary-integral kernel matrix (a Laplace single-layer potential between two well-separated point clouds; see Methods). Figure 4 shows relative Frobenius error against target rank  $k$  for all three algorithms. It also shows the Eckart–Young optimal curve, computed from a truncated ground-truth spectrum (the top 250 singular values, via `scipy.sparse.linalg.svds`, exact for every rank tested here).

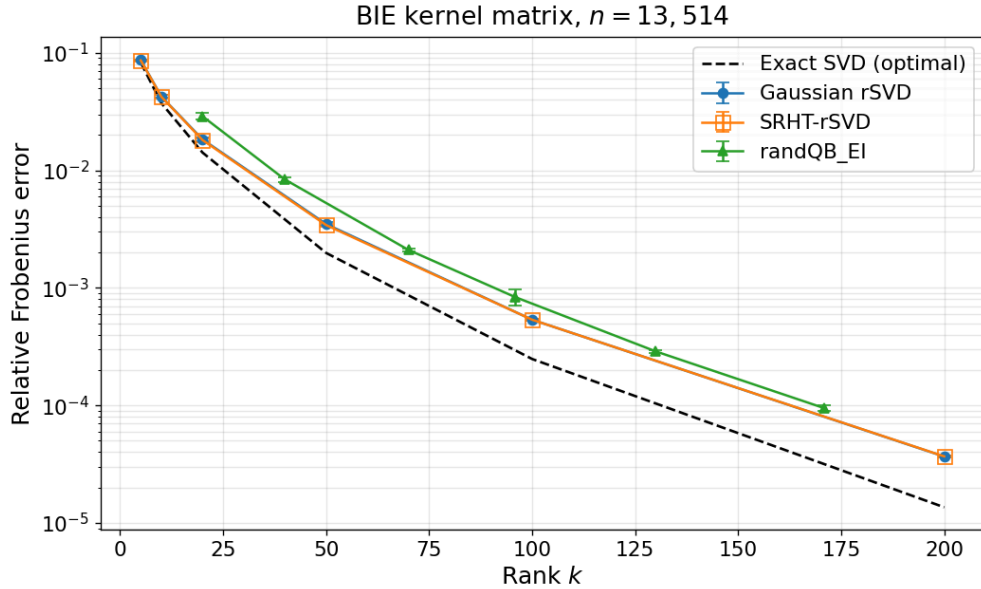


Figure 4: Relative Frobenius error vs. target rank, on the  $n = 13,514$  boundary-integral kernel matrix. Gaussian rSVD and SRHT-rSVD look the same at every rank. randQB\_EI stays a small, steady amount behind both. All three follow the Eckart–Young optimal curve closely, which shows the code is correct.

Table 8 gives the exact numbers. Gaussian rSVD and SRHT-rSVD agree almost exactly at every rank (for example at  $k = 200$ :  $3.67 \times 10^{-5}$  vs.  $3.68 \times 10^{-5}$ ), which makes sense, since both use a sketch of the same size,  $\ell = k + p$ . randQB\_EI is run as a fixed-precision sweep over its tolerance  $\varepsilon$ , and we report it at the rank it reaches. It stays behind both other methods by a

fairly steady  $1.4\text{--}1.6 \times$  across the whole range (comparing against Gaussian rSVD interpolated to randQB\_EI’s achieved rank). Again this is not surprising, since randQB\_EI’s blocks get no extra oversampling margin beyond the block size  $b$ .

| Algorithm                           | $k$  | Frob. error           | vs. opt.      | Spectral error        | vs. opt.      |
|-------------------------------------|------|-----------------------|---------------|-----------------------|---------------|
| Gaussian rSVD                       | 5    | $8.67 \times 10^{-2}$ | $1.05 \times$ | $4.14 \times 10^{-2}$ | $1.02 \times$ |
| SRHT-rSVD                           | 5    | $8.63 \times 10^{-2}$ | $1.04 \times$ | $4.10 \times 10^{-2}$ | $1.01 \times$ |
| Gaussian rSVD                       | 50   | $3.49 \times 10^{-3}$ | $1.76 \times$ | $1.51 \times 10^{-3}$ | $2.65 \times$ |
| SRHT-rSVD                           | 50   | $3.42 \times 10^{-3}$ | $1.73 \times$ | $1.43 \times 10^{-3}$ | $2.51 \times$ |
| Gaussian rSVD                       | 200  | $3.67 \times 10^{-5}$ | $2.70 \times$ | $1.23 \times 10^{-5}$ | $4.17 \times$ |
| SRHT-rSVD                           | 200  | $3.68 \times 10^{-5}$ | $2.71 \times$ | $1.25 \times 10^{-5}$ | $4.23 \times$ |
| randQB_EI ( $\varepsilon = 0.05$ )  | 20   | $2.88 \times 10^{-2}$ | $2.03 \times$ | $1.60 \times 10^{-2}$ | $2.88 \times$ |
| randQB_EI ( $\varepsilon = 0.001$ ) | 95.8 | $8.36 \times 10^{-4}$ | $2.82 \times$ | $3.34 \times 10^{-4}$ | $3.80 \times$ |

Table 8: Accuracy at chosen ranks,  $n = 13,514$  boundary-integral kernel matrix, 12-seed means (randQB\_EI at its achieved mean rank). “vs. opt.” is the ratio to the Eckart–Young optimal error at the same rank. Two things are worth noticing here: the gap to optimal grows with  $k$  for every method, since the fixed oversampling  $p = 10$  gives proportionally less slack at higher rank, and that gap is consistently worse in the spectral norm than in the Frobenius norm.

**The choice of norm matters more than the choice of method.** Reading down Table 8’s last two columns, the distance to the Eckart–Young optimum is always larger in the spectral norm than in the Frobenius norm, and the difference widens with rank: at  $k = 5$  the two norms agree almost exactly ( $1.05 \times$  vs.  $1.02 \times$  for Gaussian rSVD), but by  $k = 200$  the Frobenius gap is  $2.70 \times$  while the spectral gap is  $4.17 \times$ . This is what one should expect from a sketch with a fixed oversampling budget. The sketch captures the directions that carry most of the Frobenius mass first, so what it leaves behind weighs relatively more heavily in the spectral norm, which is set by the single largest missed direction. The practical reading is that the choice of error norm changes the apparent quality of all three methods by more than the choice between the methods does, so a reader comparing published rSVD results should check which norm is being reported before comparing numbers.

**Matches the synthetic control test.** The same pattern, Gaussian  $\approx$  SRHT, with randQB\_EI behind because it has no oversampling margin, also shows up on a synthetic exponential-decay spectrum at  $n = 2000$ , used here as a check before trusting the results on the boundary-integral matrix. Table 9 gives the numbers. But the size of randQB\_EI’s gap depends on the spectrum: about  $2.1 \times$  worse than Gaussian/SRHT at matched rank on the synthetic spectrum, against  $1.4\text{--}1.6 \times$  worse on the boundary-integral matrix. The direction of the effect stays the same, but its size does not.

| Algorithm                          | $k$       | Rel. Frobenius error   |
|------------------------------------|-----------|------------------------|
| Gaussian rSVD                      | 20        | $1.705 \times 10^{-1}$ |
| SRHT-rSVD                          | 20        | $1.668 \times 10^{-1}$ |
| Gaussian rSVD                      | 100       | $9.13 \times 10^{-5}$  |
| SRHT-rSVD                          | 100       | $8.87 \times 10^{-5}$  |
| randQB_EI ( $\varepsilon = 0.05$ ) | 50 (mean) | $2.22 \times 10^{-2}$  |
| randQB_EI ( $\varepsilon = 0.1$ )  | 40 (mean) | $5.57 \times 10^{-2}$  |

Table 9: Synthetic exponential-decay spectrum ( $n = 2000$ ), 12-seed means. Same pattern as Table 8: Gaussian and SRHT agree closely; randQB\_EI trails by a wider margin here than on the boundary-integral matrix.

**randQB\_EI’s tolerance is a promise it keeps, and usually overshoots.** The point of a fixed-precision interface is that the caller names an error and gets it. Table 10 checks whether that happens on the main matrix. It does, in one direction: at every tolerance we tried, the error delivered is below the error requested, never above. But the margin is large at loose tolerances, requesting  $\varepsilon = 0.05$  delivers 0.029, roughly twice the accuracy requested, and it narrows as the tolerance tightens, until at  $\varepsilon = 10^{-4}$  the delivered error  $9.5 \times 10^{-5}$  sits just under the request. Two causes push in the same direction. The method can only stop on a multiple of the block size  $b = 10$ , so it overshoots by up to one block (the same effect Table 19 isolates), and its error indicator is an upper bound on the true error, not an estimate of it. A caller who needs a tight rank should read this as follows: the tolerance passed in is a ceiling, and at loose tolerances the caller pays for accuracy that was not requested.

| Requested $\varepsilon$ | Achieved rank | Error delivered       | Ratio | Runtime (s) |
|-------------------------|---------------|-----------------------|-------|-------------|
| 0.05                    | 20.0          | $2.88 \times 10^{-2}$ | 0.58  | 0.79        |
| 0.01                    | 40.0          | $8.35 \times 10^{-3}$ | 0.84  | 1.30        |
| 0.003                   | 70.0          | $2.10 \times 10^{-3}$ | 0.70  | 2.09        |
| 0.001                   | 95.8          | $8.36 \times 10^{-4}$ | 0.84  | 2.80        |
| 0.0003                  | 130.0         | $2.88 \times 10^{-4}$ | 0.96  | 3.73        |
| 0.0001                  | 170.8         | $9.48 \times 10^{-5}$ | 0.95  | 4.86        |

Table 10: randQB\_EI on the  $n = 13,514$  boundary-integral kernel matrix, 12-seed means,  $b = 10$ ,  $k_{\max} = 400$ . All 72 trials converged. “Ratio” is the delivered error as a fraction of the requested tolerance: below 1 everywhere, so the tolerance is never violated, but well below 1 at the loose end. Peak memory was 1393 MB at every tolerance, unchanged by the rank reached.

**A fixed-precision method can fail outright where a fixed-rank one cannot.** The synthetic layer also includes a noisy, near-flat spectrum, deliberately built with no exploitable gap. On it, Gaussian rSVD and SRHT-rSVD both degrade gracefully, they still return an answer, just a poor one: relative error 0.987 at  $k = 50$  and 0.948 even at  $k = 200$ , against the 1.0 that returning nothing at all would give. randQB\_EI does not degrade in this way, it fails to produce an answer at all. Table 11 shows this is not limited to one hard case: at every tolerance we tried, all 12 seeds hit the rank cap ( $k_{\max} = 300$ ) before reaching the flat spectrum’s target error and raised the RuntimeError in step 8 of Algorithm 3, with zero convergences out of 60 trials. The polynomial-decay spectrum, on which a fixed-rank method always returns something, still fails randQB\_EI outright at the tightest tolerance,  $\varepsilon = 0.05$ : reaching that accuracy needs more rank than the 300-column budget allows, and there is no partial credit. Either the block-by-block loop reaches tolerance within budget, or the caller gets an exception instead of a low-quality answer.

| Spectrum    | $\varepsilon = 0.05$ | $\varepsilon = 0.10$ | $\varepsilon = 0.20$ | $\varepsilon = 0.30$ | $\varepsilon = 0.50$ |
|-------------|----------------------|----------------------|----------------------|----------------------|----------------------|
| Exponential | 12/12                | 12/12                | 12/12                | 12/12                | 12/12                |
| Polynomial  | 0/12                 | 12/12                | 12/12                | 12/12                | 12/12                |
| Flat        | 0/12                 | 0/12                 | 0/12                 | 0/12                 | 0/12                 |

Table 11: randQB\_EI’s convergence rate (out of 12 seeds) at  $n = 2000$ ,  $b = 10$ ,  $k_{\max} = 300$ . A failure here is not a bad answer, it is no answer at all: the call raises instead of returning a factorization.

This is a real, practical gap in the “use randQB\_EI for a fixed accuracy” advice in Chapter 5: it only holds if the requested tolerance is reachable within whatever rank budget the caller has set. A fixed-rank method never has this failure mode, it always returns **something** at the requested  $k$ , however inaccurate. A caller who does not know in advance whether a tolerance is reachable should set a generous  $k_{\max}$  and be ready to catch the exception, not assume the call will succeed.

### 4.3 Runtime, memory, and the SRHT speed question

SRHT’s sketch step has a better time complexity than Gaussian rSVD’s dense projection:  $O(n \log n)$  for the structured Hadamard transform, against  $O(n\ell)$  for a dense random projection. So in theory it should be the faster of the two. Figure 5 shows this advantage does not show up at the sizes we tested.

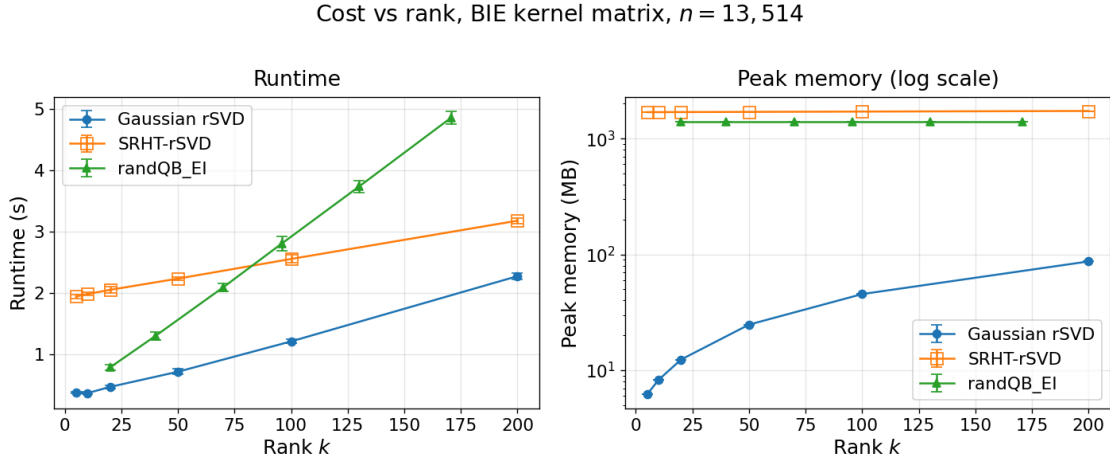


Figure 5: Runtime and peak memory vs. target rank,  $n = 13,514$ . SRHT-rSVD is slower and uses far more memory than Gaussian rSVD at every rank we tested, even though it should be cheaper in theory.

| Algorithm     | $k$ | Runtime (s) | Peak mem. (MB) | vs. Gaussian                          |
|---------------|-----|-------------|----------------|---------------------------------------|
| Gaussian rSVD | 5   | 0.383       | 6.2            | –                                     |
| SRHT-rSVD     | 5   | 1.946       | 1692.5         | $5.1 \times$ time, $273 \times$ mem.  |
| Gaussian rSVD | 50  | 0.722       | 24.8           | –                                     |
| SRHT-rSVD     | 50  | 2.236       | 1701.7         | $3.1 \times$ time, $68.6 \times$ mem. |
| Gaussian rSVD | 200 | 2.275       | 87.0           | –                                     |
| SRHT-rSVD     | 200 | 3.177       | 1732.7         | $1.4 \times$ time, $19.9 \times$ mem. |

Table 12: Runtime and memory at  $n = 13,514$ , 12-seed means. SRHT’s cost stays almost flat across  $k$ , because it comes mostly from a fixed-size sketch step that does not depend on the target rank. So its disadvantage gets smaller as  $k$  grows and Gaussian’s cost catches up.

**randQB\_EI sits between the two on cost, and its memory does not depend on the rank it reaches.** Its runtime grows with the rank it settles on, as one would expect from an incremental method: 0.79s at achieved rank 20, up to 4.86s at rank 171, slower than Gaussian rSVD at every matched rank, but on the same order. Its peak memory is 1393 MB at every tolerance we tried, whether it stops at rank 20 or rank 171. That number is close to the size of one dense copy of the matrix itself (1.46 GB), which is what dominates it: the growing  $Q$  and  $B$  factors are small next to a single full-size intermediate. So on memory, randQB\_EI is

much closer to SRHT-rSVD than to Gaussian rSVD, and the ranking on memory is Gaussian  $\ll$  randQB\_EI  $<$  SRHT at this size.

**Is this gap an implementation artifact?** We checked this directly. The Fast Walsh–Hadamard Transform (FWHT) can be written in more than one way, and the two ways differ a great deal in speed even though they compute exactly the same result. A plain implementation built from ordinary NumPy array operations, rebuilding the whole array at every one of the 14 stages with `numpy.concatenate`, and taking a hidden full copy through a transpose before reshaping, is the version this project started with. The version we report throughout this thesis instead updates its array in place and is compiled with Numba (Appendix A). Both versions were checked against an independent Hadamard-matrix reference and give bit-identical output, so the gap that remains after this optimization is a property of the comparison itself, not measurement noise or an unoptimized implementation.

Table 13 measures the two versions side by side rather than leaving this as a recollection. Both were run in the same process on the same matrix, and the script asserts they produce the same sketch before timing either. The rewrite is worth  $7.8\text{--}8.8\times$  in time and  $4.1\text{--}4.9\times$  in peak memory, consistently at both sizes. The size of that gap is the real point here: had we compared Gaussian rSVD against the original version, SRHT would have looked  $16.7\times$  slower at  $n = 13,514$  rather than  $2.1\times$ , and we would have reported a conclusion about our own code instead of about the two sketching methods.

| $n$    | Version | Time (s) | Peak mem. | Time saved  | Memory saved |
|--------|---------|----------|-----------|-------------|--------------|
| 2000   | pre-fix | 0.206    | 164 MB    |             |              |
| 2000   | current | 0.023    | 39 MB     | $8.8\times$ | $4.1\times$  |
| 13,514 | pre-fix | 15.04    | 8857 MB   |             |              |
| 13,514 | current | 1.92     | 1817 MB   | $7.8\times$ | $4.9\times$  |

Table 13: The sketch step under the FWHT implementation used throughout this thesis and under the one it replaced, at  $l = 210$ , mean of 3 calls. The pre-fix code is taken verbatim from this project’s own version history, not rewritten from a description of it. For scale, Gaussian rSVD’s sketch on the same matrix takes 0.90s and 22.7 MB (Table 15), so the pre-fix version was  $16.7\times$  slower and  $390\times$  heavier where the current one is  $2.1\times$  and  $80\times$ .

**The same pattern shows up at every size we tested, not just  $n = 13,514$ .** Table 14 gives the SRHT-vs-Gaussian ratio at three sizes: the  $n = 2000$  synthetic exponential spectrum (Chapter 3),  $n = 13,514$  (Table 12), and  $n = 20,000$  (Section 4.6). SRHT’s disadvantage is largest at small rank and shrinks as  $k$  grows, at every size, but SRHT is never faster, and never close to matching Gaussian’s memory use, anywhere we tested.

| Matrix                   | $k$ | Time ratio  | Memory ratio |
|--------------------------|-----|-------------|--------------|
| Synthetic, $n = 2000$    | 5   | $8.9\times$ | $23.0\times$ |
| Synthetic, $n = 2000$    | 50  | $3.2\times$ | $6.0\times$  |
| Synthetic, $n = 2000$    | 200 | $1.2\times$ | $2.0\times$  |
| BIE kernel, $n = 13,514$ | 5   | $5.1\times$ | $273\times$  |
| BIE kernel, $n = 13,514$ | 50  | $3.1\times$ | $68.6\times$ |
| BIE kernel, $n = 13,514$ | 200 | $1.4\times$ | $19.9\times$ |
| BIE kernel, $n = 20,000$ | 5   | $7.7\times$ | $27.8\times$ |

| Matrix                   | $k$ | Time ratio   | Memory ratio |
|--------------------------|-----|--------------|--------------|
| BIE kernel, $n = 20,000$ | 50  | $4.4 \times$ | $7.1 \times$ |
| BIE kernel, $n = 20,000$ | 200 | $1.9 \times$ | $2.2 \times$ |

Table 14: SRHT-rSVD’s runtime and peak memory, as a multiple of Gaussian rSVD’s, at every size tested in this thesis. SRHT is always slower and always uses more memory, though the gap narrows at higher rank since SRHT’s sketch cost is roughly fixed while Gaussian’s grows with  $k$ . One caveat on the memory column: the  $n = 20,000$  runs use the batched sketch described in Section 4.6, while the two smaller sizes do not, so the memory ratios in the last three rows are much smaller than they would be without batching and should not be read as SRHT’s overhead falling away with size. The time ratios are unaffected.

A check of the floating-point operation count rules out “too much math” as the reason. To measure this cleanly we timed the sketch step on its own, separately from the QR and SVD work that both methods share, at  $k = 200$  ( $l = 210$ ). Figure 6 shows the result in three bars, and Table 15 gives the numbers.

Sketch step only, BIE kernel matrix  $n=13,514$ ,  $l=210$

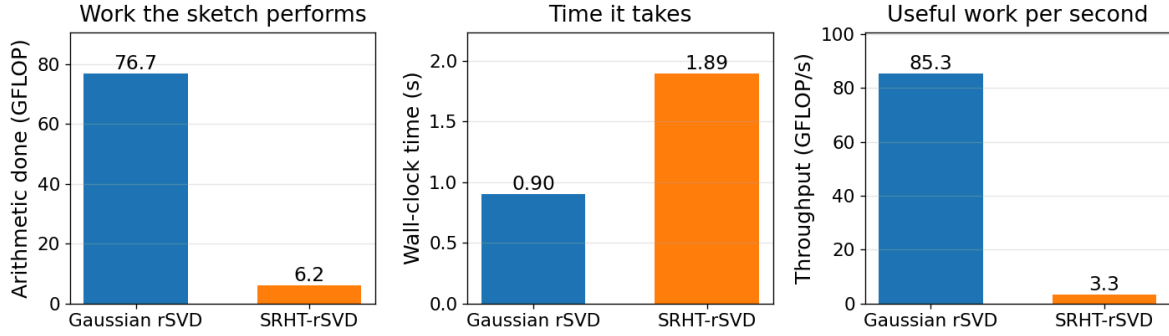


Figure 6: The sketch step alone,  $n = 13,514$ ,  $l = 210$ . SRHT-rSVD does far less arithmetic than Gaussian rSVD and still takes twice as long, because it retires operations at a small fraction of the rate. The left two panels are what the complexity analysis predicts and what the clock actually says; the right panel is the distance between them.

SRHT’s sketch does a fixed 6.2 GFLOP: it transforms the whole padded matrix once, no matter what  $k$  is, then picks out a subset, while Gaussian’s dense sketch does 76.7 GFLOP at this  $l$ . So SRHT does  $12.4 \times$  **fewer** raw operations. It is still  $2.1 \times$  slower: 1.89s against 0.90s.

| Method        | Arithmetic         | Time (s)          | Throughput        | Peak mem.        |
|---------------|--------------------|-------------------|-------------------|------------------|
| Gaussian rSVD | 76.7 GFLOP         | 0.900             | 85.3 GFLOP/s      | 22.7 MB          |
| SRHT-rSVD     | 6.2 GFLOP          | 1.892             | 3.28 GFLOP/s      | 1817 MB          |
| ratio         | $12.4 \times$ less | $2.1 \times$ more | $26 \times$ lower | $80 \times$ more |

Table 15: Sketch step only,  $n = 13,514$ ,  $l = k + p = 210$ , mean of 3 timed calls after a warm-up call (the Numba kernel compiles on first use). Operation counts are analytic:  $2mn\ell$  for the dense product, and  $2mn_{\text{pad}} \log_2 n_{\text{pad}}$  for one padded Walsh–Hadamard transform, with  $n_{\text{pad}} = 16,384$ . Note the SRHT count does not contain  $l$  at all, it transforms everything and only keeps  $l$  rows.

The reason is throughput, not arithmetic. Gaussian’s sketch is a single BLAS dgemm call and sustains about 85 GFLOP/s on our machine. Our compiled Hadamard transform sustains only about 3.3 GFLOP/s, a  $26 \times$  gap in useful work per second. BLAS spreads a matrix multiply across threads, blocks it to fit the cache, and has decades of tuning behind it. A butterfly transform is harder to treat that way: its memory access pattern strides further apart at every

stage, and ours runs on a single thread. This is a hardware and library efficiency gap, not a mistake in the algorithm or a remaining bug in the code.

The memory column of Table 15 explains the other half of Table 12. The padded buffer is  $16,384 \times 13,514$  in `float64`, which is 1.77 GB, and it exists whether  $k$  is 5 or 200. That single allocation is essentially all of SRHT-rSVD’s peak memory, which is why that peak barely moves across the rank sweep while Gaussian rSVD’s rises from 6 MB to 87 MB.

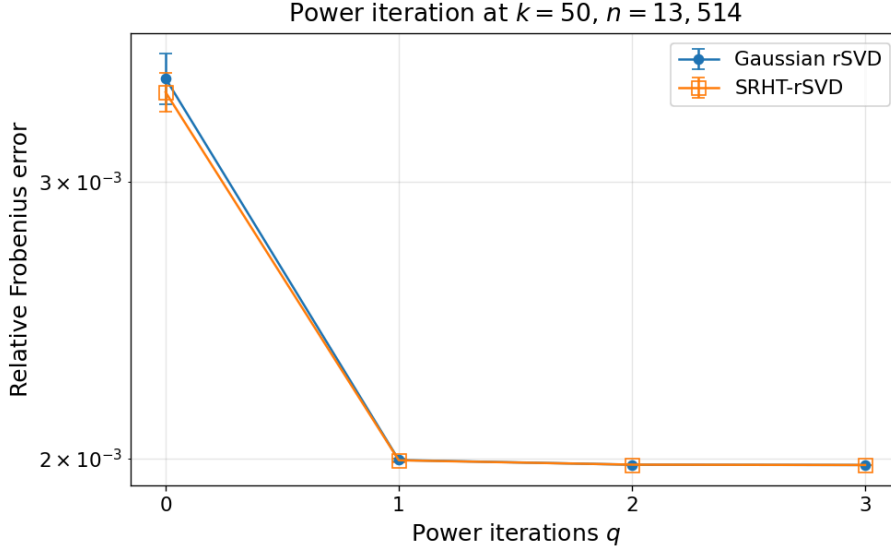


Figure 7: Power-iteration convergence at fixed rank  $k = 50$ ,  $n = 13,514$ . Gaussian rSVD and SRHT-rSVD look the same across  $q \in \{0, 1, 2, 3\}$ . This confirms the SRHT power-iteration extension, added to remove an unfair edge that Gaussian rSVD would otherwise have, gives correct results.

At  $q = 3$ , Gaussian rSVD and SRHT-rSVD report the same mean error to six decimal places (0.001982 vs. 0.001982). SRHT’s runtime is still higher (2.29s  $\rightarrow$  4.36s across  $q = 0..3$ , against Gaussian’s 0.72s  $\rightarrow$  2.80s), and its memory stays flat at about 1.70 GB no matter what  $q$  is. This is because power iteration only adds dense matrix multiplies on top of the already-sketched basis. It does not call the FWHT again.

Table 16 gives the full sweep. Reading across it, the practical message is that one power iteration is worth buying and a second is not:  $q = 0 \rightarrow 1$  cuts the Frobenius error by 43% and the spectral error by 62%, while  $q = 1 \rightarrow 3$  moves the Frobenius error by under 1%. Each extra  $q$  costs Gaussian rSVD roughly 0.7s at this size, so the second and third iterations buy almost nothing for a real price. The memory column shows a second effect: Gaussian rSVD’s peak rises from 24.8 MB to 43.4 MB at  $q = 1$  and then stops, because power iteration needs one extra full-height intermediate and no more, whichever  $q$  is chosen.

| Algorithm     | $q$ | Frob. error            | Spectral error         | Runtime (s) |
|---------------|-----|------------------------|------------------------|-------------|
| Gaussian rSVD | 0   | $3.488 \times 10^{-3}$ | $1.512 \times 10^{-3}$ | 0.72        |
| Gaussian rSVD | 1   | $1.996 \times 10^{-3}$ | $5.74 \times 10^{-4}$  | 1.45        |
| Gaussian rSVD | 2   | $1.983 \times 10^{-3}$ | $5.71 \times 10^{-4}$  | 2.11        |
| Gaussian rSVD | 3   | $1.982 \times 10^{-3}$ | $5.71 \times 10^{-4}$  | 2.80        |
| SRHT-rSVD     | 0   | $3.419 \times 10^{-3}$ | $1.431 \times 10^{-3}$ | 2.29        |
| SRHT-rSVD     | 1   | $1.995 \times 10^{-3}$ | $5.75 \times 10^{-4}$  | 2.96        |

| Algorithm | $q$ | Frob. error            | Spectral error        | Runtime (s) |
|-----------|-----|------------------------|-----------------------|-------------|
| SRHT-rSVD | 2   | $1.983 \times 10^{-3}$ | $5.71 \times 10^{-4}$ | 3.64        |
| SRHT-rSVD | 3   | $1.982 \times 10^{-3}$ | $5.71 \times 10^{-4}$ | 4.36        |

Table 16: Power iteration at fixed rank  $k = 50$ ,  $n = 13,514$ , 12-seed means. The two methods agree to three significant figures at every  $q$ , which is the check the SRHT power-iteration extension had to pass. Peak memory (not shown) is 24.8 MB for Gaussian rSVD at  $q = 0$  and 43.4 MB at every  $q \geq 1$ ; SRHT-rSVD stays at 1702 MB throughout, since its cost is dominated by the padded sketch rather than by power iteration.

#### 4.4 How much faster than a full SVD?

Everything so far compares the three randomized methods against each other. The comparison that motivates all of them is against the thing they replace: a full SVD of the same matrix. We measured it directly. At  $n = 13,514$ , `numpy.linalg.svd` with `full_matrices=False` takes 637 seconds and peaks at 4181 MB. The randomized methods answer the same question in a few seconds and a few megabytes.

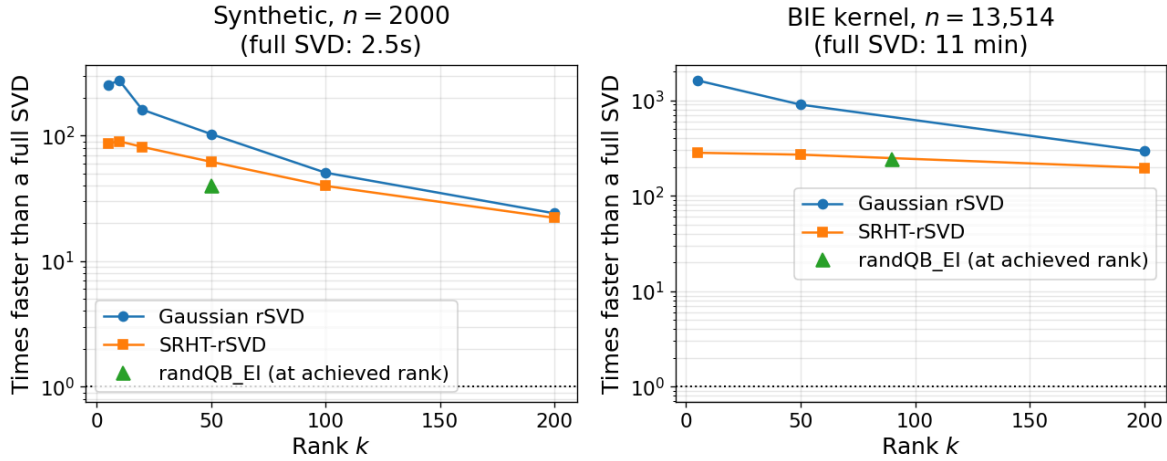


Figure 8: Runtime as a multiple of a full SVD's, at two sizes. Everything on these axes was measured in one process against one baseline per matrix, since a ratio is only meaningful if numerator and denominator were timed under the same conditions. The dotted line is break-even. The left panel averages over the three synthetic spectra, which barely differ: neither a full SVD's cost nor a fixed-rank sketch's depends on the spectrum.

Table 17 gives the numbers at the thesis's main size, and two things stand out here. First, the advantage shrinks as  $k$  grows, from  $1613 \times$  at  $k = 5$  to  $293 \times$  at  $k = 200$  for Gaussian rSVD, because the full SVD's cost is fixed while the sketch's grows with the rank. Second, the memory advantage is larger than the time advantage and shrinks the same way:  $675 \times$  less memory at  $k = 5$ ,  $48 \times$  at  $k = 200$ . For SRHT-rSVD the memory picture is quite different, and worse: its padded sketch buffer puts it within  $2.5 \times$  of a full SVD's peak no matter what rank is asked for, so at this size it saves time but hardly saves any memory at all.

| Method        | $k$ | Time (s) | vs. full SVD  | Peak mem. | vs. full SVD |
|---------------|-----|----------|---------------|-----------|--------------|
| Full SVD      | —   | 637.5    | —             | 4181 MB   | —            |
| Gaussian rSVD | 5   | 0.40     | $1613 \times$ | 6.2 MB    | $675 \times$ |
| Gaussian rSVD | 50  | 0.71     | $900 \times$  | 24.8 MB   | $169 \times$ |
| Gaussian rSVD | 200 | 2.17     | $293 \times$  | 87.0 MB   | $48 \times$  |

| Method                                | $k$ | Time (s) | vs. full SVD | Peak mem. | vs. full SVD |
|---------------------------------------|-----|----------|--------------|-----------|--------------|
| SRHT-rSVD                             | 5   | 2.26     | $282 \times$ | 1692 MB   | $2.5 \times$ |
| SRHT-rSVD                             | 50  | 2.36     | $270 \times$ | 1702 MB   | $2.5 \times$ |
| SRHT-rSVD                             | 200 | 3.24     | $196 \times$ | 1733 MB   | $2.4 \times$ |
| randQB_EI ( $\varepsilon = 10^{-3}$ ) | 90  | 2.64     | $242 \times$ | 1393 MB   | $3.0 \times$ |

Table 17: Against a full dense SVD of the same  $n = 13,514$  boundary-integral kernel matrix, 3-seed means. The full SVD row is a single measurement, at 637 seconds it is not something to repeat unnecessarily. The randomized runtimes and memory figures here reproduce Table 12’s to within a few percent, which is a useful check: those came from a separate benchmark run on a different day.

The same pattern holds at  $n = 2000$ , at a smaller scale: a full SVD takes 1.6–3.8s depending on the spectrum, against  $253 \times$  faster for Gaussian rSVD at  $k = 5$  and  $24 \times$  at  $k = 200$ . That the advantage is  $6\text{--}12 \times$  larger at  $n = 13,514$  than at  $n = 2000$  is exactly the shape one would expect: a full SVD costs  $O(n^3)$  while a rank- $k$  sketch costs about  $O(n^2k)$ , so the gap between them should widen with  $n$ , which is the whole argument for using these methods on large matrices in the first place.

One caveat belongs with these numbers. A full SVD returns the entire spectrum and both complete sets of singular vectors, while the randomized methods return  $k$  of each. When  $k$  is all that is needed, the comparison above is the relevant one. If the rest is needed too, no speedup applies, because the randomized methods cannot provide it at any price.

#### 4.5 How much do the tuning parameters matter?

Each method has one knob that has so far been held fixed: oversampling  $p$  for the two fixed-rank methods, and block size  $b$  for randQB\_EI. This section reports how sensitive the results are to those choices, measured on the synthetic polynomial-decay spectrum at  $n = 2000$ , the hardest of our synthetic cases, chosen because a slowly decaying spectrum is where oversampling should matter the most.

| $p$ | Gaussian err. | Gaussian time (s) | SRHT err. | SRHT time (s) |
|-----|---------------|-------------------|-----------|---------------|
| 2   | 0.1684        | 0.031             | 0.1668    | 0.103         |
| 5   | 0.1638        | 0.035             | 0.1631    | 0.112         |
| 10  | 0.1577        | 0.037             | 0.1575    | 0.117         |
| 20  | 0.1489        | 0.041             | 0.1480    | 0.121         |
| 40  | 0.1362        | 0.060             | 0.1363    | 0.138         |

Table 18: Oversampling sweep at fixed rank  $k = 50$ , synthetic polynomial-decay spectrum,  $n = 2000$ . Accuracy improves slowly and steadily with  $p$ ; doubling  $p$  from 10 to 20 buys about 6% error reduction for about 13% more time in Gaussian rSVD. The two methods track each other at every  $p$ , as they do at every  $k$ .

Oversampling sweep at  $k = 50$ , synthetic polynomial decay,  $n = 2000$

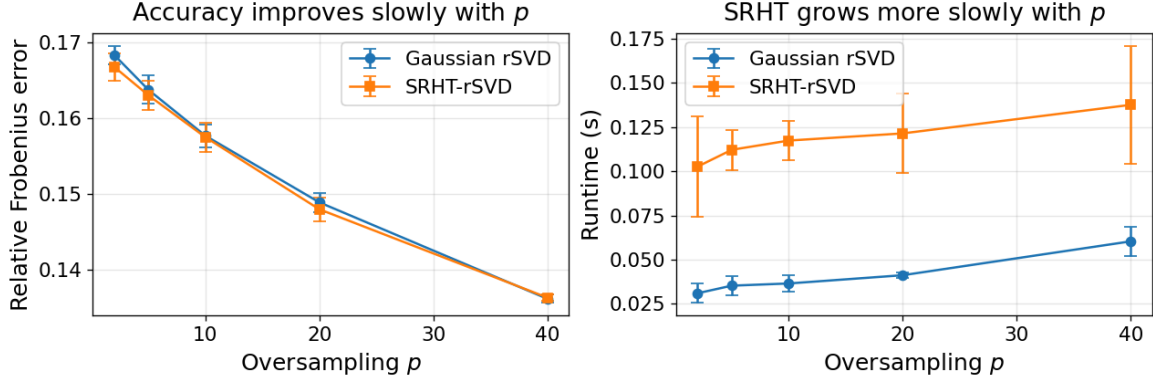


Figure 9: The same sweep as Table 18, drawn. Left: the two methods’ accuracy curves lie on top of each other at every  $p$ . Right: both get slower as  $p$  grows, but Gaussian rSVD’s time nearly doubles across the sweep (+95%) where SRHT-rSVD’s rises by a third (+34%), since SRHT’s sketch cost does not depend on  $\ell$ . Error bars are  $\pm 1$  standard deviation over 8 seeds.

A few things are worth drawing out of Table 18. Accuracy does improve monotonically with  $p$ , consistent with the  $k/(p-1)$  factor in Equation 6, but the improvement is gradual: going from  $p = 2$  to  $p = 40$ , a twentyfold increase, improves the error by only about 19%. Our default of  $p = 10$  sits in a comfortable part of that curve, and nothing in the thesis’s conclusions would change at  $p = 5$  or  $p = 20$ . Gaussian rSVD and SRHT-rSVD also agree at every  $p$ , just as they agree at every  $k$ , which is further evidence that the two differ only in cost, not in accuracy. Finally, the time cost of extra oversampling is much steeper for Gaussian rSVD (its sketch is  $O(mn\ell)$ , so it scales directly with  $p$ ) than for SRHT-rSVD (whose sketch cost is independent of  $\ell$ ). This is the one place in the thesis where SRHT’s structure shows a genuine advantage, though it is still not enough to overcome its much larger fixed cost.

| $b$ | Achieved rank | Rel. Frobenius error | Runtime (s) |
|-----|---------------|----------------------|-------------|
| 5   | 67.5          | 0.1463               | 0.156       |
| 10  | 70.0          | 0.1441               | 0.109       |
| 20  | 80.0          | 0.1341               | 0.115       |
| 40  | 80.0          | 0.1344               | 0.090       |

Table 19: Block-size sweep for randQB\_EI at fixed tolerance, synthetic polynomial-decay spectrum,  $n = 2000$ . Larger blocks overshoot the tolerance, they stop at a higher rank than needed, but run faster, because the method performs fewer reorthogonalization passes.

randQB\_EI block-size sweep, synthetic polynomial decay,  $n = 2000$

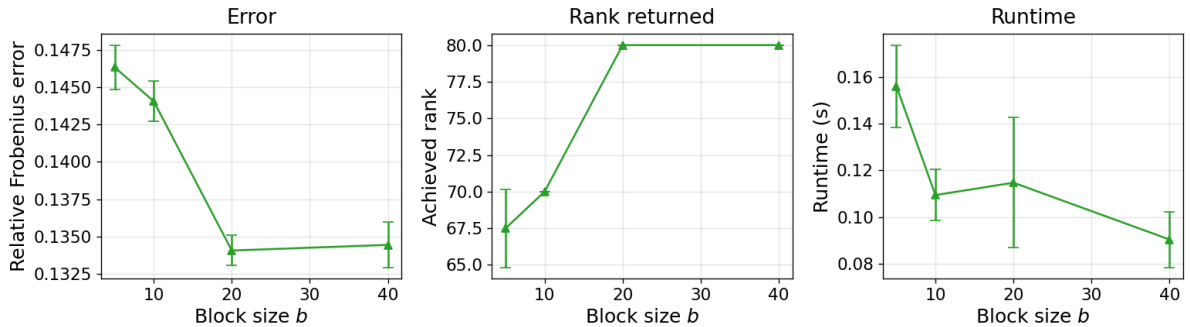


Figure 10: randQB\_EI’s block size, drawn from the same runs as Table 19. The middle panel is the trade-off: the method can only stop on a multiple of  $b$ , so  $b = 20$  and  $b = 40$  both return rank 80 where  $b = 5$  returns 67.5 on average. Error and runtime both move in the caller’s favour as  $b$  grows, since the cost of a large block is a rank that was not requested, not a worse answer. Error bars are  $\pm 1$  standard deviation over 8 seeds; the zero-width bars at  $b \geq 10$  mean all 8 seeds stopped at the same rank.

Table 19 shows the trade-off built into randQB\_EI’s block size. The method can only stop at a multiple of  $b$ , so a large  $b$  overshoots: at  $b = 40$  it returns rank 80 where  $b = 5$  returns rank 67.5 on average for nearly the same accuracy target. The overshoot is not wasted, the extra rank does buy a lower error, but it is not requested either, and a caller who wants a tight rank should use a small  $b$ . Runtime moves the other way, falling from 0.156s to 0.090s as  $b$  grows, because larger blocks mean fewer passes of reorthogonalization against the accumulated basis. Our default  $b = 10$  is a middle choice; the right value depends on whether the caller cares more about a tight rank or a fast answer.

#### 4.6 Scaling from $n = 13,514$ to $n = 20,000$

RQ3 asks whether the budget needed for a fixed accuracy target changes with problem size. We measure this two ways: the rank needed for a given accuracy (at fixed oversampling  $p = 10$ ), and the number of power iterations needed. Figure 11 and Figure 12 compare the  $n = 13,514$  results above against a  $n = 20,000$  boundary-integral kernel matrix, built the same way (same geometry, same separation ratio, a different random point-cloud draw).

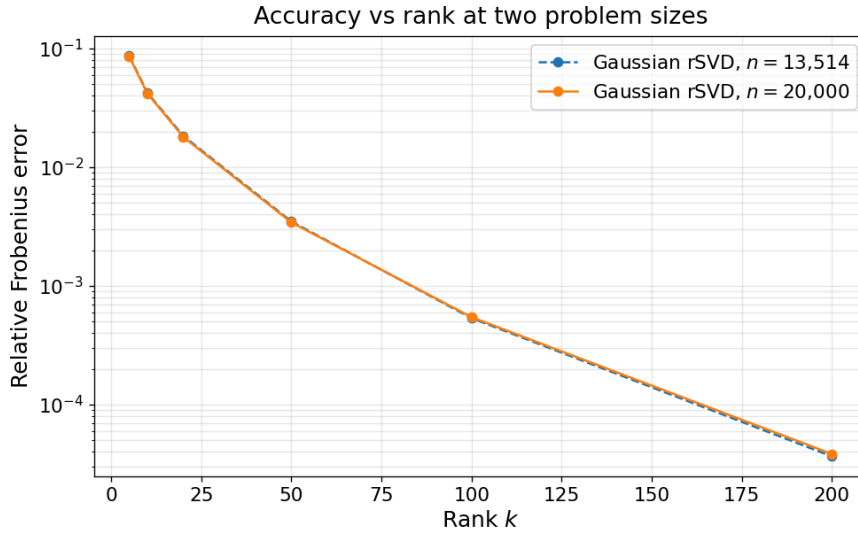


Figure 11: Accuracy vs. rank, Gaussian rSVD,  $n = 13,514$  vs.  $n = 20,000$ . The two curves look the same.

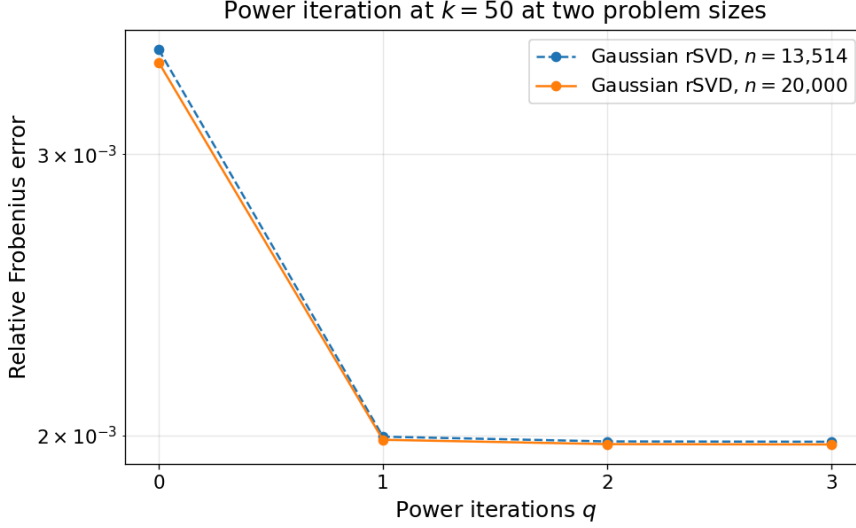


Figure 12: Power-iteration convergence at  $k = 50$ ,  $n = 13,514$  vs.  $n = 20,000$ . Again the same.

Table 20 gives the ratio of relative Frobenius error at matched rank between the two sizes. All six ranks fall between  $0.98$  and  $1.05 \times$  of each other, with no clear trend up or down.

| $k$ | Error, $n = 13,514$   | Error, $n = 20,000$   | Ratio |
|-----|-----------------------|-----------------------|-------|
| 5   | $8.67 \times 10^{-2}$ | $8.56 \times 10^{-2}$ | 0.986 |
| 10  | $4.25 \times 10^{-2}$ | $4.18 \times 10^{-2}$ | 0.984 |
| 20  | $1.83 \times 10^{-2}$ | $1.80 \times 10^{-2}$ | 0.979 |
| 50  | $3.49 \times 10^{-3}$ | $3.42 \times 10^{-3}$ | 0.982 |
| 100 | $5.35 \times 10^{-4}$ | $5.46 \times 10^{-4}$ | 1.019 |
| 200 | $3.67 \times 10^{-5}$ | $3.80 \times 10^{-5}$ | 1.048 |

Table 20: Relative Frobenius error at matched rank, Gaussian rSVD,  $n = 13,514$  vs.  $n = 20,000$ . The ratios stay close to 1.0 with no clear trend, that is, no size effect we can detect over this range.

This matches the idea that the boundary-integral kernel’s decay depends on the separation between the two point clouds, not on how many points are in each cloud. Adding more points at the same geometry does not change how fast the singular values fall off. SRHT’s memory cost does grow with problem size, through the padded dimension (the next power of two above  $n$ ). This meant we had to bound its sketch with a batched FWHT (`sketch_batch_size`) at  $n = 20,000$  to stay inside available memory. But that is a memory-engineering detail, not a change in the accuracy/budget relationship RQ3 asks about.

Two limits on this result are worth stating here rather than only in Section 5.2. First, only two sizes were tested here. That is enough to see no effect, but not enough to claim a general scaling law. Second, oversampling was held fixed at  $p = 10$  while the rank was varied, so what we show is that the **rank** needed for a given accuracy does not shift with size. We did not separately sweep  $p$  at both sizes, so we cannot say from this experiment whether the best **oversampling** value shifts with size.

# Discussion

## 5.1 Practical recommendations

This section collects the practical advice that follows from Section 4.1 through Section 4.6.

**Check what kind of “PDE matrix” is at hand before assuming it compresses.** A finite-element or finite-difference discretization matrix generally does not have low numerical rank, no matter how big the matrix or how fine the mesh. A boundary-integral kernel matrix, built from a smooth Green’s function between well-separated point sets, generally does. This difference matters more than the word “PDE” itself.

**Gaussian rSVD is the safe default at the sizes tested here.** At matched rank, it reaches the same accuracy as SRHT-rSVD, using far less memory ( $20\text{--}270 \times$  less at  $n = 13, 514$ ) and no more time, usually less. It is also simpler to build: no padding, no structured transform, no extra dependency. Its memory advantage is not only over SRHT: at  $n = 13, 514$  Gaussian rSVD peaks at  $6\text{--}87$  MB depending on rank, while randQB\_EI holds about 1.4 GB regardless of rank, so Gaussian rSVD is the only one of the three whose memory use stays small.

**SRHT is not yet worth it at these sizes, with the implementation tested here.** Its better time complexity is real in theory, but it did not turn into a real speed win, even for a compiled, in-place sketch step. It is only worth reaching for SRHT if a more heavily engineered implementation is on hand (multi-threaded, vectorized), or if one works at a scale much larger than tested here, where the  $O(n \log n)$  vs.  $O(n\ell)$  gap might finally beat BLAS’s efficiency edge.

**Use randQB\_EI for a fixed accuracy, not a fixed rank, but budget for the case where that accuracy is not reachable.** It costs a small, steady accuracy penalty against Gaussian/SRHT at the same rank, and its interface, pick a tolerance and get whatever rank reaches it, is the right tool when the right rank is not known ahead of time. Unlike the other two methods, though, it can fail outright: Table 11 shows it raising an exception on every trial for a hard spectrum, and even on an easy one once the tolerance is tight enough relative to the rank cap. Set  $k_{\max}$  generously and be ready to catch that failure, since a fixed-rank method never has it, because it always returns an answer at the rank asked for, however poor.

**One power iteration buys most of the benefit.** Going from  $q = 0$  to  $q = 1$  closes most of the accuracy gap to the exact SVD, for both Gaussian rSVD and SRHT-rSVD, on the boundary-integral matrix tested here (Figure 7). Later iterations help much less, while still costing extra passes over the matrix.

**Check which norm a reported error uses before comparing numbers.** All three methods sit further from the Eckart–Young optimum in the spectral norm than in the Frobenius norm, and the difference grows with rank (Table 8). At  $k = 200$ , the same Gaussian rSVD run is  $2.70 \times$  optimal in Frobenius norm but  $4.17 \times$  optimal in spectral norm. That spread is larger than the spread between the three methods, so a number quoted without its norm is hard to compare against.

Table 21 condenses the advice above into a lookup.

| Situation                | Use           | Because                                          |
|--------------------------|---------------|--------------------------------------------------|
| the target rank is known | Gaussian rSVD | same accuracy as SRHT, least memory of the three |

| Situation                                   | Use                            | Because                                                                                        |
|---------------------------------------------|--------------------------------|------------------------------------------------------------------------------------------------|
| the target accuracy is known, not the rank  | randQB_EI, generous $k_{\max}$ | only one of the three with that interface; can raise instead of returning                      |
| memory is limited                           | Gaussian rSVD                  | $20\text{--}270 \times$ below SRHT and $16\text{--}225 \times$ below randQB_EI at $n = 13,514$ |
| more accuracy is needed at the same rank    | one power iteration, $q = 1$   | cuts Frobenius error 43%; $q = 2, 3$ add under 1% more                                         |
| the matrix is FEM or finite-difference      | check the spectrum first       | no low-rank structure to exploit (Section 4.1)                                                 |
| SRHT's $O(n \log n)$ sketch looks appealing | measure before committing      | slower and heavier than Gaussian at every size tested (Table 14)                               |

Table 21: The recommendations above as a lookup table. Every entry is a finding from this thesis's own measurements at  $n = 2000$  to  $20,000$ , not general advice, see Section 5.2 for where it may stop holding.

## 5.2 Limitations

This section states what this work does not establish.

- The main matrices in Section 4.2 through Section 4.6 are **built** from a real physical kernel, not downloaded from a data repository. This is normal practice for boundary-integral benchmarks in the literature, but it is a different kind of “real” than `pde900` and `poisson3Da`, which are genuine downloaded matrices.
- We test only one kernel: the Laplace single-layer potential, between two well-separated spherical point clouds. Results may not carry over to other kernels (for example Helmholtz at high frequency), other geometries, or matrices where the two point sets sit close together (the “near-field” case). That last case is exactly why hierarchical methods split a matrix into blocks in the first place, instead of treating the whole matrix as low-rank.
- randQB\_EI's comparison against the other two methods always goes through a lookup step (Chapter 3), not a direct run at the same rank. This is a reasonable approximation, not an exact match.
- RQ3 varies the rank at fixed oversampling  $p = 10$ . We did not sweep  $p$  itself at both sizes, so we cannot say whether the best oversampling value shifts with problem size.
- Peak memory is measured with Python's `tracemalloc`, which tracks Python-level allocation, not full process memory.
- RQ3's scaling result rests on two matrix sizes. That is enough to see no effect over this range; it is not enough to claim a general scaling law.
- RQ0's negative finding rests on two matrices from one family of problems (Poisson-type). It is a real, checked result for these two matrices, not a proof that no FEM stiffness matrix can ever have low rank.
- RQ2's answer is scoped to the specific implementation tested here (Numba-compiled, single-threaded, on one machine). We did not try building a from-scratch implementation with the same level of engineering BLAS has (multi-threading, cache-blocking, vectorization); such an implementation might close the remaining gap further.

- The SRHT power-iteration extension applies the standard scheme already used by Gaussian rSVD. It is included to equalize the comparison, and its correctness is checked experimentally (Section 4.3), but no separate theoretical analysis of it is given here.

## Chapter 6

# Future work

This work leaves several concrete directions open. They are listed roughly in order of how directly they follow from the results above.

**Settle RQ2 with a properly engineered transform.** The largest open question is that our Hadamard transform is single-threaded and not hand-tuned, while the BLAS baseline is both. A multi-threaded, explicitly vectorized, cache-blocked FWHT, or an existing optimized library implementation, would turn the present result (“SRHT loses on this implementation”) into a much stronger one either way. If a well-engineered SRHT still loses at these sizes, the finding becomes considerably more robust. If it wins instead, the crossover point itself is the interesting quantity.

**Find the crossover size.** Table 14 shows SRHT’s disadvantage shrinking as  $k$  grows but never disappearing over  $n = 2000$  to  $20,000$ . Since SRHT’s sketch cost is  $O(mn \log n)$  against Gaussian’s  $O(mn\ell)$ , the ratio should scale like  $\ell / \log n$ , which grows with  $k$  and only weakly with  $n$ . Testing  $n$  in the hundreds of thousands, with a matrix held implicitly rather than densely, would show whether a crossover exists on this hardware at all.

**Sweep oversampling across sizes.** As noted in Section 5.2, we varied rank at fixed  $p = 10$ , so RQ3’s answer covers the rank budget but not the oversampling budget. Sweeping  $p$  at both sizes would close that gap and is inexpensive to run.

**Other kernels and other geometries.** We test one kernel at one separation ratio. The natural extensions are (i) sweeping  $\eta$  to map measured numerical rank against the geometric decay rate predicted by Equation 16, which would turn the qualitative claim of Section 2.9 into a quantitative one; (ii) the Helmholtz kernel  $e^{i\kappa\|x-y\|} / (4\pi\|x-y\|)$ , where the rank is known to grow with the wavenumber  $\kappa$ , giving a case where low-rank methods should visibly degrade; and (iii) near-field blocks with  $\eta$  close to 1, where compression should fail. A method that behaves well across all three regimes is far better characterized than one tested at a single  $\eta$ .

**Extend to hierarchical matrices.** A single well-separated block is the building block of an H-matrix, not a substitute for one [Hac99]. The natural next step is to partition a full boundary-element matrix into a block tree, compress each admissible block with the winner of this comparison, and measure end-to-end accuracy and memory against a dense solve. That would also exercise the near-field case above, since near-diagonal blocks must be stored densely.

**Compare against Krylov methods.** Randomized block Krylov methods [TW23, MM15] often beat a plain randomized sketch on matrices whose spectrum decays slowly, at the cost of more passes over  $A$ . Our slowest case here, the noisy near-flat synthetic spectrum, is exactly the regime where that comparison would be informative.

**Adaptive rank for the fixed-rank methods.** randQB\_EI gets its stopping rule from the cheap error indicator Equation 12. The same identity applies to any QB-form factorization, so Gaussian rSVD and SRHT-rSVD could be given the same adaptive behaviour rather than requiring  $k$  in advance. That would remove the interface mismatch that forces the post-hoc rank lookup described in Chapter 3, and would make the three-way comparison exact rather than interpolated.

**Single-pass and GPU variants.** For matrices too large to store, single-pass sketching methods [TW23] trade accuracy for touching  $A$  only once. Separately, structured transforms

are a better fit for GPUs than the memory-bound picture in Section 4.3 suggests, so the ranking found here may not carry over to that hardware.

## Conclusion

This thesis set out to answer four questions about randomized low-rank approximation on PDE-derived matrices. Real finite-element matrices do not have low numerical rank, while boundary-integral kernel matrices, built from a smooth Green’s function, do (RQ0). On the boundary-integral matrix, Gaussian rSVD and SRHT-rSVD reach the same accuracy at matched rank, and randQB\_EI trails both by a margin that depend on the underlying spectrum (RQ1). SRHT’s better time complexity does not turn into a real speed advantage at the sizes tested here, even for a compiled, in-place implementation verified against an independent reference, and the remaining gap comes from BLAS’s hardware and library efficiency, not from a flaw in the algorithm (RQ2). The rank and the power-iteration budget needed for a fixed accuracy target do not change between  $n = 13,514$  and  $n = 20,000$  (RQ3).

None of these three algorithms is new. What this thesis adds is a controlled, reproducible comparison between them on a matrix class that truly has low rank, an SRHT power-iteration extension needed to make that comparison fair, and an answer to RQ2 that holds up against a plain implementation and across three different problem sizes, not just one single measurement. We hope this is useful groundwork for anyone deciding which randomized low-rank method to use on a similar problem.

## Appendix A

# Key implementations

This appendix gives the four pieces of code that carry the ideas of the thesis. They are lightly trimmed for readability: guard clauses for degenerate input, the optional power-iteration branch of `randQB_EI`, the one-dimensional input case of the SRHT sketch, and the wrapper that splits that sketch into row batches are omitted. Nothing that affects the numerics has been changed. The full source, the experiment scripts and the 58-test suite are in the accompanying project repository.

## A.1 The Fast Walsh–Hadamard Transform

This is the kernel that decides SRHT’s performance, and the subject of Section 4.3. It applies the Hadamard matrix Equation 10 in place, in  $\log_2 n$  butterfly stages, without ever forming the matrix. It is compiled with Numba; the scalar loop below becomes machine code on first call.

```
@numba.njit(nogil=True)
def _fwht_kernel(a):
    """In-place FWHT along axis 0 of a 2-D float64 C-contiguous array.
    a.shape[0] must be a power of two."""
    n, d = a.shape
    h = 1
    while h < n:
        step = 2 * h
        for i in range(0, n, step):
            for j in range(h):
                p = i + j
                q = p + h
                for col in range(d):
                    # innermost loop over the
                    # contiguous axis, for cache
                    x = a[p, col]
                    y = a[q, col]
                    a[p, col] = x + y
                    a[q, col] = x - y
            h *= 2
```

Two details matter here. The innermost loop runs over columns, the contiguous axis of a C-ordered array, so each pass streams memory rather than jumping around. And the butterfly reads both `a[p, col]` and `a[q, col]` into scalars before writing either, which is what allows the update to happen in place with no temporary array. A version built with `numpy.concatenate` to rebuild the array at every stage instead, a natural first way to write this loop, touches enough memory to dominate the whole function’s cost, and avoiding that is most of the speed difference reported in Section 4.3.

## A.2 The SRHT sketch

The transform above is applied inside the sketching operator, which multiplies by random signs, transforms, rescales, and subsamples rows, that is, it applies  $\Theta$  from Equation 9. The single-batch path is shown below. Around it sits a `sketch_batch_size` argument, omitted here, which bounds memory by feeding the input through this same path in row batches, and that is what makes  $n = 20,000$  feasible.

```
def apply(self, X):
    n, d = X.shape
    if self.n_pad != n:
        # zero-pad to a power of two
```

```

        Xp = np.zeros((self.n_pad, d)); Xp[:n, :] = X
    else:
        Xp = np.array(X, dtype=np.float64, order="C", copy=True)
    Xp *= self.signs[:, None]          # D: random +/-1
    _fwht_inplace(Xp)                  # H: Hadamard transform
    Xp /= np.sqrt(self.n_pad)          # normalization
    return self.scale * Xp[self.idx, :] # R: subsample l rows

```

### A.3 randQB\_EI's error indicator

The loop below is Algorithm 3. The line `E -= float(np.sum(Bi * Bi))` is the whole reason the method can choose its own rank so cheaply: by Equation 12, subtracting the Frobenius mass of each new block of  $B$  from a running total tracks the true approximation error exactly, with no residual ever formed.

```

def randqb_ei(A, eps, b=10, P=0, max_k=None, rng=None):
    A_sq = _sum_sq(A)                # ||A||_F^2
    tol = (eps ** 2) * A_sq
    Q = np.zeros((m, 0)); B = np.zeros((0, n))
    E = A_sq                          # running error, Equation 12
    while True:
        Oi = rng.standard_normal((n, b))
        AOi = A @ Oi
        Y = (AOi - Q @ (B @ Oi)) if Q.size else AOi  # remove captured part
        Qi, _ = np.linalg.qr(Y)
        if Q.size:                    # re-orthogonalize for stability
            Qi -= Q @ (Q.T @ Qi)
            Qi, _ = np.linalg.qr(Qi)
        Bi = Qi.T @ A
        Q = np.hstack([Q, Qi]); B = np.vstack([B, Bi])
        E -= float(np.sum(Bi * Bi))    # <-- the error indicator
        if E < tol:
            break
        if Q.shape[1] >= max_k:
            raise RuntimeError("hit the rank cap before reaching tolerance")
    return Q, B

```

### A.4 Error metrics without forming the residual

This is Equation 18 and Equation 19 in code. Both are needed because a dense residual is impossible at the sizes used here.

```

def frob_error_lowrank(A, U, S, Vt, norm_A=None):
    """||A - U diag(S) Vt||_F / ||A||_F, without forming the residual."""
    norm_A_sq = norm_A ** 2 if norm_A is not None else frob_sq(A)
    AV = A @ Vt.T                      # (m, k), thin
    cross = float(np.sum(S * np.einsum("ij,ij->j", U, AV)))
    diff_sq = max(norm_A_sq - 2 * cross + float(np.sum(S ** 2)), 0.0)
    return float(np.sqrt(diff_sq / norm_A_sq))    # Equation 18

def _residual_operator(A, U, S, Vt):
    """A - U diag(S) Vt as an implicit operator: matvecs only."""
    def matvec(x):
        x = np.asarray(x).ravel()
        return A @ x - U @ (S * (Vt @ x))        # Equation 19
    def rmatvec(x):
        x = np.asarray(x).ravel()

```

```
    return A.T @ x - Vt.T @ (S * (U.T @ x))  
    return LinearOperator(A.shape, matvec=matvec, rmatvec=rmatvec)
```

The spectral error is then the largest singular value of that operator, obtained with `svds(R, k=1)`, divided by  $\|A\|_2$ .

## References

- [HMT11] Halko, Nathan, Martinsson, Per-Gunnar, and Tropp, Joel A., Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions, *SIAM Review* **53** no. 2 (2011) 217–288.
- [TW23] Tropp, Joel A., and Webber, Robert J., Randomized Algorithms for Low-Rank Matrix Approximation: Design, Analysis, and Applications, arXiv preprint arXiv:2306.12418 (2023).
- [BG13] Boutsidis, Christos, and Gittens, Alex, Improved Matrix Algorithms via the Subsampled Randomized Hadamard Transform, *SIAM Journal on Matrix Analysis and Applications* **34** no. 3 (2013) 1301–1340.
- [Tro11] Tropp, Joel A., Improved Analysis of the Subsampled Randomized Hadamard Transform, *Advances in Adaptive Data Analysis* **3** nos. 1–2 (2011) 115–126.
- [YGL18] Yu, Wenjian, Gu, Yu, and Li, Yaohang, Efficient Randomized Algorithms for the Fixed-Precision Low-Rank Matrix Approximation, *SIAM Journal on Matrix Analysis and Applications* **39** no. 3 (2018) 1339–1359.
- [GR87] Greengard, Leslie, and Rokhlin, Vladimir, A Fast Algorithm for Particle Simulations, *Journal of Computational Physics* **73** no. 2 (1987) 325–348.
- [Hac99] Hackbusch, Wolfgang, A Sparse Matrix Arithmetic Based on H-Matrices. Part I: Introduction to H-Matrices, *Computing* **62** (1999) 89–108.
- [GV13] Golub, Gene H., and Loan, Charles F. Van, *Matrix Computations*, 4th ed., Johns Hopkins University Press, 2013.
- [EY36] Eckart, Carl, and Young, Gale, The Approximation of One Matrix by Another of Lower Rank, *Psychometrika* **1** no. 3 (1936) 211–218.
- [Mir60] Mirsky, Leon, Symmetric Gauge Functions and Unitarily Invariant Norms, *The Quarterly Journal of Mathematics* **11** no. 1 (1960) 50–59.
- [MT20] Martinsson, Per-Gunnar, and Tropp, Joel A., Randomized Numerical Linear Algebra: Foundations and Algorithms, *Acta Numerica* **29** (2020) 403–572.
- [JL84] Johnson, William B., and Lindenstrauss, Joram, Extensions of Lipschitz Mappings into a Hilbert Space, in *Conference in Modern Analysis and Probability*, American Mathematical Society, pp. 189–206.
- [Sar06] Sarlós, Tamás, Improved Approximation Algorithms for Large Matrices via Random Projections, in *47th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 143–152.
- [RST09] Rokhlin, Vladimir, Szlam, Arthur, and Tygert, Mark, A Randomized Algorithm for Principal Component Analysis, *SIAM Journal on Matrix Analysis and Applications* **31** no. 3 (2009) 1100–1124.
- [AC09] Ailon, Nir, and Chazelle, Bernard, The Fast Johnson–Lindenstrauss Transform and Approximate Nearest Neighbors, *SIAM Journal on Computing* **39** no. 1 (2009) 302–322.

- [FA76] Fino, Bernard J., and Algazi, V. Ralph, Unified Matrix Treatment of the Fast Walsh–Hadamard Transform, *IEEE Transactions on Computers* **C-25** no. 11 (1976) 1142–1146.
- [Rok85] Rokhlin, Vladimir, Rapid Solution of Integral Equations of Classical Potential Theory, *Journal of Computational Physics* **60** no. 2 (1985) 187–207.
- [Beb08] Bebendorf, Mario, *Hierarchical Matrices: A Means to Efficiently Solve Elliptic Boundary Value Problems*, vol. 63, Lecture Notes in Computational Science and Engineering, Springer, 2008.
- [DH11] Davis, Timothy A., and Hu, Yifan, The University of Florida Sparse Matrix Collection, *ACM Transactions on Mathematical Software* **38** no. 1 (2011) 1–25.
- [Har+20] Harris, Charles R., Millman, K. Jarrod, Walt, Stéfan J. van der, and others, Array Programming with NumPy, *Nature* **585** (2020) 357–362.
- [Vir+20] Virtanen, Pauli, Gommers, Ralf, Oliphant, Travis E., and others, SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python, *Nature Methods* **17** (2020) 261–272.
- [LPS15] Lam, Siu Kwan, Pitrou, Antoine, and Seibert, Stanley, Numba: A LLVM-Based Python JIT Compiler, in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC (LLVM '15)*, pp. 1–6.
- [Hun07] Hunter, John D., Matplotlib: A 2D Graphics Environment, *Computing in Science & Engineering* **9** no. 3 (2007) 90–95.
- [MM15] Musco, Cameron, and Musco, Christopher, Randomized Block Krylov Methods for Stronger and Faster Approximate Singular Value Decomposition, in *Advances in Neural Information Processing Systems 28 (NIPS)*, pp. 1396–1404.