École polytechnique fédérale de Lausanne

# Project N°2 Report

## Randomized Nyström

*Authors:*
Tara Fjellman
Amal Seddas

*Professor:*
Laura Grigori

# Contents

# 1 Introduction

In this project, we study the randomized Nyström approximation algorithm for the low-rank approximation of a positive-semidefinite matrix $A \in \mathbb{R}^{n\times n}$. There are several important applications for this theory, such as image processing, PCA, or solving integral equations. However, the most common practical setting are kernel methods for large-scale machine-learning problems. Since these algorithms scale at least quadratic in the number of data points, low-rank approximations are essential to obtain reasonable storage usage and computational costs.

Our goal is to efficiently parallelize this alogrithm. Key aspects of our investigations are numerical stability, scalability, performance (in terms of runtime), and the approximation of the leading $k$ singular values of $A$.

# 2 Randomized Nyström low rank approximation

The goal of the Randomized Nyström algorithm is to represent a given matrix $A \in \mathbb{R}^{n\times n}$ by some lower rank $k < n$ approximation. The starting point of the algorithm is a rank-$\ell$ approximation of $A$, in the form

$$A_{Nyst} := (A\Omega)(\Omega^T A\Omega)^\dagger(\Omega^T A) \in \mathbb{R}^{n\times n},$$

where $(\Omega^T A\Omega)^\dagger$ is the pseudoinverse of $\Omega^T A\Omega$ and $\Omega \in \mathbb{R}^{n\times\ell}$ is a sketching matrix. We assume $k < \ell < n$. To further reduce the rank of $A_{Nyst}$ to $k$, we have two choices: approximate only the core matrix $\Omega^T A\Omega$ or the whole $A_{Nyst}$. In this project, we focus on the second approach. Algorithm **??** shows how this can be achieved.

**Algorithm 1** Randomized Nyström approximation using the Cholesky decomposition

1: Let $C = A\Omega$ and $B = \Omega^T C$.
2: Compute the Cholesky decomposition: $B = LL^T$.
3: Solve the linear system using back substitution: $Z = C(L^T)^{-1}$.
4: Perform QR decomposition: $Z = QR$.
5: Compute the singular value decomposition (SVD) of $R$ and truncate:

$$R = U\Sigma V^T \approx \tilde{U}_k\tilde{\Sigma}_k\tilde{V}_k^T.$$

6: Set $\tilde{U}_k = QU_k$ and return $\tilde{U}_k\tilde{\Sigma}_k^2\tilde{U}_k^T \approx A_{Nyst}$.

To show why the result of algorithm **??** is in fact a rank-$k$ approximation of $A_{Nyst}$, we write

$$\begin{aligned}\tilde{U}_k\tilde{\Sigma}_k^2\tilde{U}_k^T &= QU_k\tilde{\Sigma}_k\Sigma_k^T\tilde{U}_k^T Q^T \\ &\approx QR(VV^T)R^T Q^T = ZZ^T = A\Omega(L^T)^{-1}(L)^{-1}\Omega^T A^T \\ &= A\Omega(LL^T)^{-1}\Omega^T A = A\Omega(\Omega^T A\Omega)^{-1}\Omega^T A.\end{aligned}$$

However, here we have the inverse of $\Omega^T A\Omega$ instead of its pseudoinverse. Unfortunately, This means Algorithm **??** fails in Step 2 if $\Omega^T A\Omega$ is numerically singular. In this case, as in [4], we replace $L$ by a square root of $B$ in SVD form. Specifically, we write $B = U_B\Sigma_B U_B^T$ (since $B$ is SPSD), and set $L = U_B\sqrt{\Sigma_B}U_B^T$. By construction, we have

$$LL^T = U_B\sqrt{\Sigma_B}U_B^T(U_B\sqrt{\Sigma_B}U_B^T)^T = U_B\Sigma_B U_B^T = B.$$

We then replace $(L^T)^{-1}$ with $(L^T)^\dagger$. This can be simply calculated as

$$(L^T)^\dagger = (U_B\sqrt{\Sigma_B}U_B^T)^\dagger,$$

where $\sqrt{\Sigma_B}$ is a diagonal matrix whose entries are given by the square roots of those of $\Sigma_B$.

# 3 Sketching and Sketching Matrices

In this section, we provide a detailed overview of sketching and the sketching matrices utilized in this project. The goal of sketching is to embed the high-dimensional matrix $A \in \mathbb{R}^{n\times n}$ into a reduced-dimensional space described by $A\Omega$ while preserving essential geometric properties of the data. An mentioned earlier it requires a tall and skinny sketching matrix $\Omega \in \mathbb{R}^{n\times \ell}$. More formally, given any two vectors $x$ and $y$ in the high-dimensional space, their sketched counterparts $\hat{x}$ and $\hat{y}$ should approximately preserve their inner product:

$$|\langle \hat{x}, \hat{y}\rangle - \langle x, y\rangle| \le \varepsilon \|x\|_2\|y\|_2 \tag{2}$$

where $\varepsilon > 0$ is a small approximation error. However, achieving this exact preservation for all $x$ and $y$ is generally infeasible due to the reduced dimensionality $\ell$. Instead, $\Omega$ is typically regarded as a random matrix, ensuring that **??** holds with high probability $1-\delta$, where $\delta < 1$.

In this project, we employ the following two types of sketching matrices.

## 3.1 Gaussian sketching

The Gaussian sketching matrix has entries of $\Omega$ drawn independently from a standard normal distribution.

This method is such that for an input matrix $A \in \mathbb{R}^{n\times d}$ and for any vector $x \in \mathbb{R}^d$:

$$(1-\varepsilon)\|Ax\|_2^2 \le \|A\Omega x\|_2^2 \le (1+\varepsilon)\|Ax\|_2^2,$$

with high probability, where $\varepsilon > 0$ is a small approximation error. This property ensures that pairwise distances between points in the projected space are approximately preserved.

The Gaussian sketching matrix can also be viewed as a random transformation that approximately satisfies the following subspace embedding property for any subspace $T \subseteq \mathbb{R}^d$:

$$(1-\varepsilon)\|v\|_2^2 \le \|\Omega v\|_2^2 \le (1+\varepsilon)\|v\|_2^2, \quad \forall v \in T.$$

### 3.2 Block SRHT (BSRHT) sketching

BSRHT is a version of the Subsampled Randomized Hadamard Transform (SRHT) specifically designed for distributed architectures. For $n$ a power of two, the SRHT can be defined as $\Omega^T = \sqrt{n/\ell}RHD$, where: $H \in \mathbb{R}^{n\times n}$ is the normalized Walsh-Hadamard matrix; $D \in \mathbb{R}^{n\times n}$ is a diagonal matrix with i.i.d. random variables $\sim$ Uniform($\pm 1$); and $R \in \mathbb{R}^{\ell\times n}$ is a subset of $\ell$ randomly sampled rows from the $n \times n$ identity matrix. Now, for $P$ different processors, BSRHT can be constructed block-wise from the SRHT as:

$$\Omega^T = \begin{pmatrix} \Omega_1^T \\ \Omega_2^T \\ \vdots \\ \Omega_P^T \end{pmatrix} = \sqrt{\frac{n}{P\ell}} \begin{pmatrix} D_{L1} & \cdots & D_{LP} \end{pmatrix} \begin{pmatrix} RH & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & RH \end{pmatrix} \begin{pmatrix} D_{R1} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & D_{RP} \end{pmatrix}, \tag{2}$$

with: $H \in \mathbb{R}^{n/P\times \ell/P}$ being the normalized Walsh-Hadamard matrix; $D_{Li} \in \mathbb{R}^{n/P\times n/P}$, $D_{Ri} \in \mathbb{R}^{n/P\times n/P}$ being diagonal matrices with i.i.d. Rademacher entries $\pm 1$; and $R \in \mathbb{R}^{\ell\times n/P}$ being a uniform sampling matrix, sampling along the rows.

This structure is particularly suited for distributed computations, as it allows for parallelism while maintaining the theoretical properties of the SRHT.

# 4 Parallelisation

We parallelise the algorithm by distributing among the processors the computation of $A\Omega, \Omega^T A\Omega$, and all other matrix operations with complexities at least proportional to $n$. Since parallelisation of QR factorisation was the topic of the first project we included the relevant functions for it in the code folder. The parallelisation of matrix products was however implemented from scratch. It was done as described in algorithm **??**, by distributing matrices smartly among the processors.

**Algorithm 2** Computes the matrix product of two matrices $D$ and $E$ in parallel.

**Require:** $q = \sqrt{P}$ is an integer, $D \in \mathbb{R}^{m\times m}$ distributed such that rank $i$ has $D_{i//q,i \bmod q}$ and $E \in \mathbb{R}^{m\times n}$ distributed such that rank $i$ has $E_{i//q}$. FullProd is true if we also want to compute $E^T DE$.

**Ensure:** $F = DE$ (and $G = E^T F$ too if FullProd is true).

$F_{ij} \leftarrow P_{ij}E_j$
**if** FullProd **then**
  $G_{ij} \leftarrow E_i^F P_{ij}$
**end if**
Row-wise Sum-reduce : $F_i \leftarrow \sum_j^q F_{ij}$
Column-wise Gather on rank 0 : $F \leftarrow [F_1^T, ..., F_q^T]^T$
**if** FullProd **then**
  Sum-reduce : $G \leftarrow \sum_{i,j}^{q,q} G_{ij}$
**end if**

This version of the matrix product allows for a fluid computation of both the $A\Omega, \Omega^T A\Omega$ products involved in the sketching and general matrix products between two matrices which appear in the computation of the rank-$k$ approximation.

The pseudo-code for the parallelisation of the randomized Nyström algorithm is then described in algorithm **??**.

**Algorithm 3** Randomized Nyström algorithm. The syntax was adapted from an Overleaf example [1].

**Require:** $A$ is an $n \times n$ symmetric positive semidefinite matrix, $\Omega$ is a sketching matrix of size $n \times l$, and $k$ is the rank of the approximation
**Ensure:** $[A_{Nyst}]_k$, the rank-$k$ randomized Nyström approximation of $A$.

```
C, B ← AΩ, Ω^T AΩ                          ▷ With algorithm 2 with FullProd=True
L, Failed ← Cholesky(B)                    ▷ Locally on rank 0
if Failed then
    U, Λ ← EigDecomp(B)                    ▷ Locally on rank 0
    Q, R ← QR(C)                           ▷ Using TSQR
    Û_k ← QU(:, 1 : k)                     ▷ With algorithm 2 with FullProd=False
    [A_Nyst]_k ← Û_k Λ(1 : k, 1 : k)^+ Û_k^T   ▷ With algorithm 2 with FullProd=False
else
    Z ← CL^{-T}                            ▷ Computed by substitution : LZ^T = C^T
    Q, R ← QR(Z)                           ▷ Using TSQR
    U_k, Σ_k, V_k ← TruncSVD(R)            ▷ Locally on rank 0
    Û_k ← QU(:, 1 : k)                     ▷ With algorithm 2 with FullProd=False
    [A_Nyst]_k ← Û_k Σ^2(1 : k, 1 : k) Û_k^T   ▷ With algorithm 2 with FullProd=False
end if
```

# 5 Experimental procedure

The presented results were obtained by running our scripts on the Helvetios cluster and averaging over different runs to make results more robust and interpretable.

In general, due to the BSRHT making use of the Hadamard transformation, $n$ was made vary as powers of 2 only. In reality one can always zero-pad the data to ensure that the Hadamard transformation can be applied. This was however avoided for performance analysis since it would have introduced unnecessary variation in the measurements.

## 5.1 Datasets

Following the approach of [3], we consider two types of datasets to evaluate the performance of our methods: synthetic datasets and those derived from the MNIST dataset. For the synthetic datasets, we construct diagonal matrices with positive eigenvalues and different rates of decay for the singular values. We explore both *polynomial decay* and *exponential decay*, with rates categorized as *slow*, *medium*,

and *fast*. The diagonal structure allows us to easily select the singular values and define an effective rank $R$ for the matrices. This design provides controlled conditions for studying spectral properties and their influence on algorithmic performance.

For the datasets derived from MNIST, we normalize the data to have values in the range $[0, 1]$ and use a subset of $n$ data points. A similarity matrix is then constructed using the radial basis function (RBF), defined as: $A_{ij} = e^{-\|x_i - x_j\|^2/c^2}$, where $x_i$ and $x_j$ are data samples, and $c$ is a tunable parameter controlling the penalty for dissimilarity. This results in a symmetric, dense $n \times n$ matrix, where $c$ determines how rapidly similarity decreases with distance.

We visualize the singular value decay of these datasets in Figure **??**, which highlights the differences between the decay patterns for the synthetic matrices and the RBF-based similarity matrices derived from MNIST.

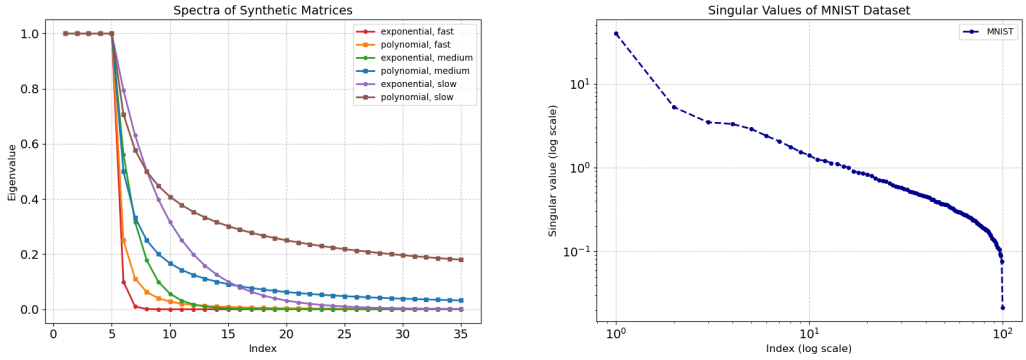


Figure 1: Singular value decay of synthetic matrices (left) and the MNIST-based similarity matrix (right).

## 5.2 Stability analysis

This section evaluates the numerical performance and stability of Gaussian Sketching and Block Subsampled Randomized Hadamard Transform (BSRHT) for low-rank matrix approximations. The trace-relative error, defined as

$$\frac{\|A - A_{\text{approx}}\|_*}{\|A\|_*}, \tag{1}$$

where $\|\cdot\|_*$ represents the nuclear norm, is used to quantify the quality of the approximations. Smaller trace-relative error values indicate higher approximation quality. Additionally, the alignment of numerical results with theoretical guarantees established by the Oblivious Subspace Embedding (OSE) property is examined.

The theoretical framework ensures that both Gaussian Sketching and BSRHT satisfy the $(\epsilon, \delta, d)$ OSE property. For Gaussian Sketching, the embedding dimension $l$ is given by:

$$l = \mathcal{O}\left(\epsilon^{-2}(d + \ln\frac{1}{\delta})\right), \tag{2}$$

ensuring accurate approximations for any target rank $d$ with high probability. For BSRHT, the embedding dimension is:

$$l = \mathcal{O}\left(\epsilon^{-2}(d + \ln\frac{n}{\delta})\ln\frac{d}{\delta}\right), \tag{3}$$

making it scalable and efficient in distributed environments.

Both methods conform to the following error bound:

$$\|A - A_{\text{approx}}\|_* \leq (1+\epsilon)\|A - [[A]]_d\|_*, \tag{4}$$

where $[[A]]_d$ is the optimal rank-$d$ approximation of $A$.

**Slow Exponential Decay:** The results (Figure **??**, left) show that Gaussian outperforms BSRHT Sketching for all ranks when $l$ is small (e.g., $l = 15$ or $l = 20$). As $l$ increases for example $l = 170$ , we run into a problem :Tara fills .

**Fast Exponential Decay:** For fast decaying spectra (Figure **??**, right), Gaussian Sketching dominates at all ranks and embedding dimensions, showing superior accuracy. However, both Gaussian Sketching and BSRHT at $l = 37$ display unexpected behavior, where the relative error does not decrease smoothly as expected for larger embedding

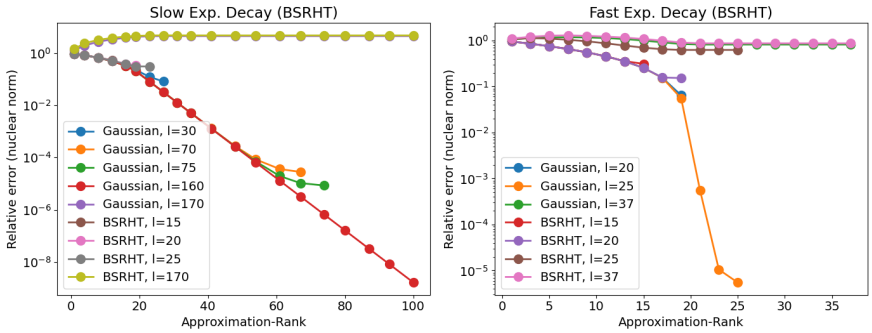


Figure 2

**Slow Polynomial Decay:** For slow exponential decay (Figure **??**, left), both Gaussian Sketching and BSRHT exhibit similar trends, with Gaussian Sketching showing slightly better performance. However, as $l$ increases, their performance becomes nearly identical.

**Fast Polynomial Decay:** n the fast polynomial decay scenario (Figure **??**, right), Gaussian Sketching consistently outperforms BSRHT across all ranks and embedding dimensions. The relative errors for Gaussian Sketching decrease significantly as the rank increases, particularly for $l = 100$ and $l = 300$. In contrast, BSRHT shows a slower reduction in error, and its performance plateaus at higher ranks.

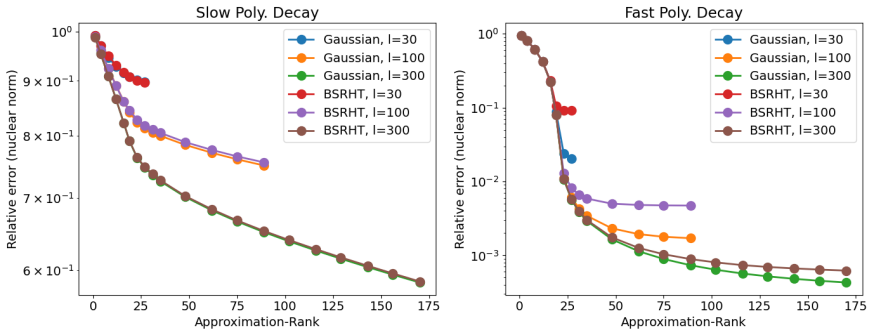


Figure 3

**MNIST** The two methods exhibit almost identical behavior, with a minor difference at lower ranks, which completely disappears as the rank increases, as shown in Figure **??**.

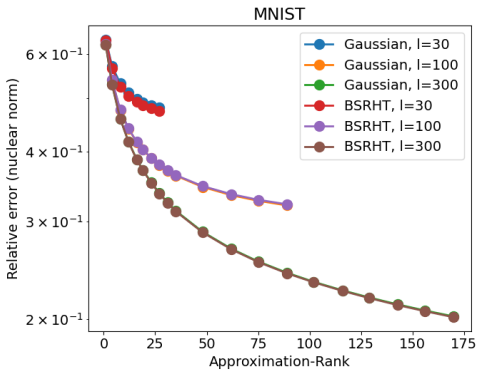


Figure 4

## 5.3 Performance analysis

For these results to have some general relevance, we tried considering realistic values of $k$ and $l$. More specifically, $k$ was taken often around $n/20$, while $l$ was taken logaritmically spaced.

When running in parallel, the number of processors was limited to values $P = 2^{2s}$ with $s \in \mathbb{N}$. This was due to perfect square constraint of the parallelised matrix multiplication and the BSRHT's use of the Hadamard transform. The values of $P$ we choose were 1,4,16,64. This also meant having to take $l \leq n/64$ integer for the TSQR algorithm to not yield an error for $P = 64$. We also recall that to see speed-up with TSQR we need the matrix to be tall-skinny. To have a good speed-up for a bigger $l$, while keeping numerical stability for ill-conditioned matrices another algo should be used.

Another implementation detail that should be mentioned is that we coded ourselves the Fast Hadamard Transform (starting from the python script on Wikipedia[2]). This was done to avoid using the only version we found online, that was extremely slow.

# 6 Algorithm Performance

## 6.1 Sequential Performance

To explore the sequential runtimes of the implemented algorithms, we decided to do two longitudinal studies : first varying $l$ for a selected value of $n$, and then varying $n$ for a selected $l$. The associated results are represented in **??**.

The main feature common to both plots is that the $k$ rank approximation part of the computation represents in most cases as small minority of runtime (as expected, since it should be of order $l^3 + nk^2$). This would of course change if one were to take big values of $k$ (i.e. if the data was really information-dense). It's runtimes also do not really seem to depend on the sketching method. This makes sense given the computations are the same regardless of which sketching method was used to obtain the $B$ and $C$ matrices. One could probably change this if one were to fully take advantage of the structure of the matrices involved in the BSRHT method (as one could use integer-float operations instead of float-float ones).

Looking more closely at both figures, it is clear that different behaviours are observed as a function of $l$ and $n$. We discuss each in the following paragraphs.

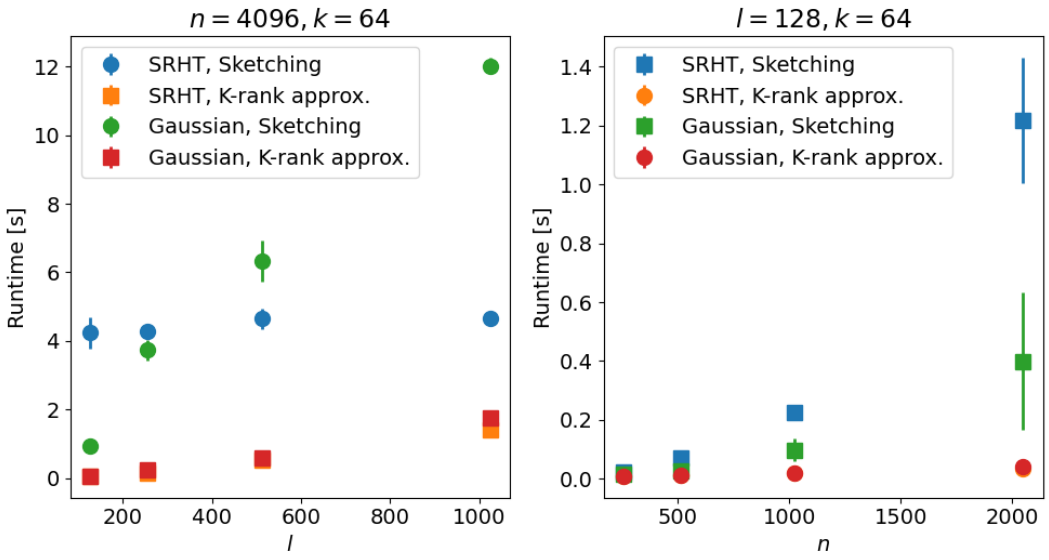


Figure 5: Runtimes associated to the tested sketching methods : as a function of $l$ (left) and as a function of $n$ (right). Runtimes are broken down in that associated to the computation of $A\Omega, \Omega^T A\Omega$ and that associated to the computation of the $k$ rank approximation.

**$l$ variation** As a function of $l$ we observe extremely different behaviours from Gaussian and BSRHT sketchings. Indeed : the curve for BSRHT looks basically flat, while the Gaussian one increases steadily. This is expected given the complexities of the algorithms respectively are of order $n^2 \log_2 n$ and $nl^2 + ln^2$. This

gives a great advantage in using BSRHT if we are embedding information-dense spaces as it scales much better.

Coming back to the $k$-rank approximation runtimes, we notice that the expected considerable increase as a function of $l$ is observed, even if as mentioned it stays relatively small when compared to the sketching runtime.

**$n$ variation** As a function of $n$ both algorithms show significant increase. BSRHT shows faster runtime growth. For $n = 1024$ it seems to take twice the time, while for $n = 2048$ it seems to take three times as much. This is a perfect match due to extra $\log_2(n)$ term in its complexity. This gives a considerable advantage in using Gaussian sketching when handling information-sparse within high dimensional spaces.

As for the $k$-rank approximation runtimes, the linear increase as a function of $n$ is hard to see on the plot due to the different scale.

## 6.2 Parallel Performance

We now turn to the analysis of the parallel performance. The relevant plot for this section is **??**. For the small run we first observe that some speed-up is displayed as

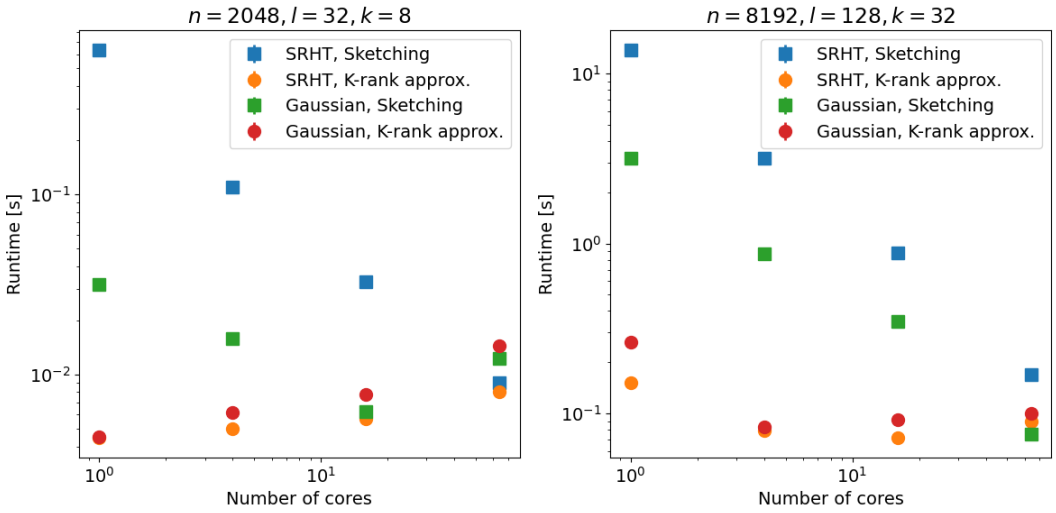


Figure 6: Runtimes associated to the tested sketching methods as a function of the number of cores : for a "small" example (left) and a "large" one (right). Runtimes are broken down in that associated to the computation of $A\Omega, \Omega^T A\Omega$ and that associated to the computation of the $k$ rank approximation.

a function of number of cores for both sketching methods. The speed-up is quasi-linear for BRSHT, while it saturates and back-fires for Gaussian sketching, due to the communication overhead. Given that the $k$-rank approximation part of the computation is smaller, it is most affected by the cost of communication, which actually dominates all throughout the core-number sweep. These behaviours are

expected given that : (a) the computation is rather small to be run in parallel; (b) BRSHT is slower than Gaussian sketching for small values of $l$.

For the specific parameter values selected, Gaussian sketching is a factor $\approx 10$ faster than BRSHT (except $P = 64$). For the large run the results are quite different. Communication is not an issue for the sketching part, meaning that quasi-linear speed-ups are observed for both methods. This means the parallelization is successful and scales well for big computations. We also observe initial speed-up for the $k$-rank approximation, though it is short-lived due to the computation still being quite small and due to that the matrix is not that tall-skinny (reducing the benefit in the QR decomposition).

# 7 Conclusion

In this project, we studied the randomized Nyström approximation algorithm for the low-rank approximation of a positive-semidefinite matrix $A \in \mathbb{R}^{n \times n}$. The goal was to efficiently parallelize this algorithm and investigate its numerical stability, scalability, performance (in terms of runtime), for both Gaussian and Block SRHT sketching methods and different decay patterns in $A$'s singular values.

The most important results were that Gaussian sketching is faster than BSRHT for small values of $l$ and that BSRHT scales better with $n$, while BSRHT scales much better with $l$ making it a must for information-dense data. The parallelization was successful and scaled well for big computations.

In terms of stability, Gaussian sketching was found to be the most stable method, consistently outperforming BSRHT across various decay patterns. However, both methods showed irregular behavior at certain embedding dimensions, which affected their performance.

Ultimately, choosing the method involves a trade-off between stability and scalability, where Gaussian sketching offers more stable results, and BSRHT provides better scalability with larger datasets

# 8 Appendix

## 8.1 Fast Walsh-Hadamard Transform

The optimized implementation of the Fast Walsh-Hadamard Transform (FWHT) presented here outperforms the SciPy version due to its focus on in-place computations and vectorized operations. Unlike the SciPy implementation, which often creates intermediate arrays and does not fully exploit in-place updates, this method minimizes memory usage by directly modifying the input matrix. This approach not only reduces overhead but also ensures better cache locality. Additionally, by processing blocks of rows iteratively and avoiding element-wise loops, the implementation leverages NumPy's highly optimized array operations, leading to a significant speedup (2-3x or more).

beginalgorithm

**Input:** Matrix $A$ of size $m \times n$, boolean flag copy **Output:** Transformed matrix $A$ $h \leftarrow 1$ copy $A \leftarrow$ copy of $A$ $h < m$ $i = 0$ to $m - 1$ with step $2h$ Perform batch updates: $A[i : i + h, :] \leftarrow A[i : i + h, :] + A[i + h : i + 2h, :]$ $A[i + h : i + 2h, :] \leftarrow A[i : i + h, :] - 2 \cdot A[i + h : i + 2h, :]$ $h \leftarrow 2h$ copy $A$

## Amal

ChatGPT was used to refine the English, generate code for the plots. Additionally, ChatGPT provided comments throughout the report and to the code to enhance readability.

## Tara

I personally use Github Copilot for coding purposes, as it can gain me some time by auto-completing lines. I however never keep lines I don't understand or find not relevant. I also sometimes used Chat GPT to brainstorm topics related to the debugging of the code.

# References

[1] 'Algorithms'. Accessed: Dec. 26, 2024. [Online]. Available: https://www.overleaf.com/learn/latex/Algorithms

[2] 'Fast Walsh–Hadamard transform - Wikiwand'. Accessed: Nov. 10, 2024. [Online]. Available: https://www.wikiwand.com/en/articles/Fast%20Walsh%E2%80%93Hadamard%20transform

[3] J. A. Tropp, A. Yurtsever, M. Udell, and V. Cevher, 'Fixed-Rank Approximation of a Positive-Semidefinite Matrix from Streaming Data'.

[4] L. Grigori, 'Randomized algorithms for low rank matrix approximation'.