Fast Gaussian processes for spatiotemporal data

via hierarchical matrix compression

by

Junhyung Chang

A thesis submitted in partial fulfillment

Of the requirements for the degree of

Master of Science

Courant Institute of Mathematical Sciences

New York University

May, 2022

_____(signature line)

Michael O'Neil

Abstract

For 1-d and 2-d input data, which are the types of data that are encountered in spatiotemporal data analysis, many kernels generate matrices that are hierarchically low-rank. Hence, hierarchical matrix compression schemes can be utilized for kernel methods including Gaussian process regression. The various types of hierarchical matrix factorization methods based on how the matrix is tessellated, how each block is compressed, which blocks to compress, etc., are reviewed. The HODLR factorization, recursive skeletonization factorization, and matrix peeling are implemented, as they are particularly useful for Gaussian processes. A full Gaussian process regression workflow based on hierarchical matrix factorizations is developed in this thesis.

Acknowledgements

I am greatly thankful to the members of my family for giving me all the love and support one can wish for.

I am also deeply grateful to professor O'Neil, first, for his genuine willingness to help, and second, for all the insightful and valuable pieces of advice regarding numerical analysis and life overall. At least one problem in my life would always be resolved after a conversation with professor O'Neil.

List of Tables

List of Figures

Contents

Introduction

This thesis aims to explore the various matrix compression techniques that can be used for Gaussian process regression. These compression schemes typically lead to a matrix factorization that can be directly applied or inverted, hence these methods are often called direct solvers (in contrast to iterative solvers).

This thesis is organized into two parts. In part I, linear algebraic fundamentals are reviewed (sections 1, 2), then hierarchical matrices and some useful hierarchical matrix factorizations are introduced in section 3. In part II, the idea behind Gaussian process regression is developed, and some computational issues that commonly arise in applications are discussed in section 4. Then, the topic of finding the best fit hyperparameters using a maximum likelihood framework is included in section 5. Finally, section 6 contains information on computing the predictive distribution accompanied by numerical experiments.

Mostly 1-d, and 2-d input data are concerned in this thesis. Kernel matrices generated by high dimensional data ($d > 2$) typically have higher rank than matrices generated from low dimensional data. Since hierarchical solvers require low-rank, they typically perform the best on 1-d (time series) or 2-d (spatial) problems. Developing efficient direct solvers for 3-d data requires a more sophisticated compression scheme that utilizes knowledge of the underlying physics of the problem. This rank growth in kernel matrices is one manifestation of the curse of dimensionality, and is explored further in section 4.1.4.

Another important theme is the use of randomization to accelerate certain matrix compression techniques when the system size is large. Randomized algorithms are introduced throughout part I, and numerical experiments are contained in part II.

Part I Preliminaries: Numerical Linear Algebra

1 Low-rank approximations

Basic linear algebra subprograms (BLAS) involving dense matrices are often computationally expensive. For a dense matrix $A \in \mathbb{R}^{n\times n}$, BLAS level 2 routines such as matrix-vector multiplies typically cost $\mathcal{O}(n^2)$ flops, and level 3 routines such as matrix-matrix multiplies, matrix inversion, and computing matrix factorizations typically cost $\mathcal{O}(n^3)$ flops if implemented naively. Fortunately, many matrices that arise in engineering and science applications have special structure which allows the operations above to be performed in much less time. Some examples of matrices with special structure include:

- Sparse matrices.
- (Numerically) low-rank matrices. A matrix $A \in \mathbb{R}^{m\times n}$ has numerical ε-rank k if

$$\inf\{\|A - X\| \ : \ \text{rank}(X) = k\} \leq \varepsilon. \tag{1.1}$$

  It is not often the case that a matrix has exact rank $k < \min(m, n)$, even in the case of rapid spectral decay. This is because the smaller singular values are usually near machine precision, but not exactly zero. When a matrix is numerically low-rank, a truncation of a rank-revealing factorization can approximate the matrix very well, and can also accelerate matrix computations. A particularly insightful rank-revealing factorization that exists for every matrix is the singular value decomposition (svd). Consider the truncated svd of length k:

$$\underset{m\times n}{A} \approx A_k = \underset{m\times k}{U} \ \underset{k\times k}{S} \ \underset{k\times n}{V}^{\top} = \sum_{i=1}^{k} \sigma_i u_i v_i^{\top}, \tag{1.2}$$

  where $k \leq \min(m, n)$, and u_i, v_i are the i-th columns of U and V respectively. The Eckart-Young-Mirsky theorem [11] states that A_k in (1.2) is the best rank-k approximation in both operator and Frobenius norms, and also provides exact errors in both norms as well. Looking at the sum in equation (1.2), a rank-k matrix can be well approximated using just k pairs of singular vectors. Hence, low-rank matrices are often referred to as data-sparse, even though the matrix A itself is dense.

- Fast transform matrices such as the fast Fourier transform (FFT) matrix, fast Walsh-Hadamard transform matrix, etc. Applying these transforms to a single vector typically has $\mathcal{O}(n \log n)$ complexity.
- Kernel matrices that arise from Green's functions of a differential operator. In this case, the fast multipole method (FMM) can be applied to perform a matrix-vector multiply in $\mathcal{O}(n)$ operations.

In this chapter, several low-rank approximation techniques that are widely used in Gaussian process regression, and other data science applications are explored.

1.1 Randomized SVD

1.1.1 Computing the range-finder

Let $K \in \mathbb{C}^{m \times n}$, $m \leq n$ be any matrix of numerical rank $r < m$. The key idea of any randomized approximate factorization is to find an approximate orthonormal basis Q for the range of K using a randomized scheme. Once Q is found, then K can be approximated by the projection of K onto the approximate range spanned by Q, namely

$$K \approx QQ^*K. \tag{1.3}$$

From the expression (1.3), one can form various low-rank factorizations of K such as the SVD, QR factorization, interpolative decomposition, etc., via standard deterministic linear algebra routines. For example, a simple algorithm from [20], [31] which uses (1.3) to form an approximate SVD of K is reviewed in section 1.1.2.

The main computational hurdle of randomized methods is finding the approximate basis Q, which the authors of [20], [31] call "Stage A". The idea is to draw a random matrix $\Omega \in \mathbb{C}^{m \times \ell}$, where ℓ is the target rank plus an oversampling parameter p, namely $\ell = r + p$, and form

$$Y = K\Omega. \tag{1.4}$$

In essence, each column of Y is in the range of K, and the columns will be linearly independent almost surely, due to the random structure of Ω. One can further orthonormalize the columns of Y via e.g. Gram-Schmidt to form the approximate orthonormal basis Q.

It is clear that the main computational challenge is the matrix product $K\Omega$ in (1.4), which costs $\mathcal{O}(mn\ell)$ flops. In addition, K may even be too large to store in core memory. Hence, a fast matrix-vector multiplying routine is usually necessary for such range-finding methods to be applied to matrices where $m, n > \mathcal{O}(10^5)$.

One method to accelerate the dense matrix-matrix multiply in (1.4) is to use an FMM-type fast summation algorithm when the matrix K is generated by a kernel function which is also a Green's function of a linear differential operator. Alternatively, an iterative matrix-matrix multiply can be performed in parallel if advanced computing resources are available.

Furthermore, it also turns out that the distribution of Ω can be chosen to accelerate the matrix product (1.4). If one simply draws the entries of Ω from a standard normal distribution, the naive matrix product costs $\mathcal{O}(mn\ell)$ flops. However, this

can be reduced to $\mathcal{O}(mn \log \ell)$ flops when Ω is structured in a specific way. The idea is to construct Ω so that $K\Omega$ first introduces randomness only to the columns of K, and then further 'mixes up' the randomness by applying a near-linear time transformation matrix such as the fast Johnson-Lindenstrauss transform, subsampled random Fourier transform, fast Walsh-Hadamard transform, etc. [49].

1.1.2 Randomized approximate SVD

Once the range-finder Q is computed, the matrix K can be approximated in the form (1.3). If one takes the SVD of Q^*K, namely $Q^*K = \hat{U}SV^*$, then (1.3) can be re-written as

$$K \approx Q\hat{U}SV^* = USV^*. \tag{1.5}$$

It is clear that $U = Q\hat{U}$ is unitary. This processes is named "Stage B" in [20], [31].

From (1.3), one can also find a generic low-rank approximation

$$K \approx \tilde{U}\tilde{V}^*, \tag{1.6}$$

where $\tilde{U} \in \mathbb{C}^{m\times r}$, and $\tilde{V} \in \mathbb{C}^{n\times r}$. This can be done e.g., by taking $\tilde{U} = Q$, and $\tilde{V}^* = Q^*K$. The approximation (1.6) will be used to compress the off-diagonal low-rank blocks in the HODLR factorization in section 6.2.1.

1.1.3 A single-pass algorithm for the randomized eigenvalue decomposition

While deriving (1.5), one needs to form the matrix Q^*K, and compute its SVD. This is the second time that the full matrix K is accessed because K was already used once during "Stage A", while forming Q. The second call for K can be avoided by using a method introduced in [20], [31]. There are two versions to this method, for Hermitian and non-Hermitian matrices respectively. In this thesis, the matrix K will always be Hermitian only self-adjoint kernels are considered. Therefore, the Hermitian single-pass algorithm is sufficient for the purposes of this thesis.

Observing (1.3), one can take the complex transpose, and use the fact that K is Hermitian to write

$$KQQ^* \approx K \approx QQ^*K. \tag{1.7}$$

From (1.7), one obtains

$$K \approx QQ^*KQQ^*. \tag{1.8}$$

Let

$$C = Q^*KQ, \tag{1.9}$$

and take its eigenvalue decomposition $C = \hat{U}D\hat{U}^*$. Then, the desired approximate decomposition of K is

$$K \approx Q\hat{U}D\hat{U}^*Q^* = UDU^*, \tag{1.10}$$

where $U = Q\hat{U}$.

Since the goal is not to use K in the computation of C in (1.9), one can alter-

natively multiply the matrix $Q^*\Omega \in \mathbb{C}^{r\times r}$ to the right of (1.9) to get

$$C(Q^*\Omega) = Q^*KQQ^*\Omega. \tag{1.11}$$

Since $K \approx KQQ^*$, and $Y = K\Omega$ is already computed in equation (1.4) during "Stage A", one can re-write (1.11) as

$$C(Q^*\Omega) \approx Q^*K\Omega = Q^*Y. \tag{1.12}$$

Thus C can be computed without using K by solving the system (1.12), which can be done quickly, since C, $Q^*\Omega$, and Q^*Y are all $r \times r$ matrices.

This method speeds up the computation of the eigenvalue decomposition, but is less accurate than the double-pass algorithm in section 1.1.2.

1.1.4 Computational complexity

Let T_{mult} denote the cost of computing Ax. It is explained in [20] that the range-finding stage costs $(k+p)T_{mult} + \mathcal{O}(k^2m)$ flops, and the factorization stage costs $(k+p)T_{mult} + \mathcal{O}(k^2(m+n))$, totalling in

$$T_{total} = 2(k+p)T_{mult} + \mathcal{O}(k^2(m+n)). \tag{1.13}$$

Given a matrix-vector multiply scheme with $T_{mult} \sim \mathcal{O}(m+n)$, which is often attainable when A is sufficiently sparse or structured, the complexity of the randomized svd is asymptotically linear in both m and n.

1.2 Rank-revealing factorizations, and computing the interpolative decomposition (ID)

A rank-revealing factorization of a matrix is often used to determine the numerical rank of a matrix, and also to compress a low-rank matrix. For a matrix $A \in \mathbb{R}^{m\times n}$, the SVD is the "best" low-rank approximation in terms of error, but is often expensive to compute. An alternative approach is the rank-revealing QR factorization

$$A\Pi = QR = Q\begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix}, \tag{1.14}$$

where $Q \in \mathbb{R}^{m\times m}$ is unitary, and $R \in \mathbb{R}^{m\times n}$ is upper triangular. Algorithms for finding Π so that the larger singular values are condensed in R_{11} include the Gu and Eisenstat algorithm [18].

Another alternative is the rank-revealing LU factorization

$$\Pi_1 A \Pi_2 = LU. \tag{1.15}$$

The rank-revealing factorization that is used most extensively in this thesis is the interpolative decomposition (ID). If an $m \times n$ matrix A has rank $k < \min(m, k)$,

then there exist factorization that "interpolates" k rows or columns by

$$\underset{m\times n}{A} = \underset{m\times k}{A(:, J_k)}\ \underset{k\times n}{X} \tag{1.16}$$

$$\underset{m\times n}{A} = \underset{m\times k}{Y}\ \underset{k\times n}{A(I_k, :)} \tag{1.17}$$

$$\underset{m\times n}{A} = \underset{m\times k}{Y}\ \underset{k\times k}{A(I_k, J_k)}\ \underset{k\times n}{X}\ , \tag{1.18}$$

where $A(\cdot, \cdot)$ is the Matlab notation for selecting a subset of rows, and columns of A. The one-sided ID in equations (1.16), (1.17) is used in the recursive skeletonization factorization.

Following the notation in [45], the ID can be computed immediately from a rank-revealing, economy-sized QR factorization

$$A\Pi = QR = Q[R_1\ R_2], \tag{1.19}$$

where $A \in \mathbb{R}^{m\times k}$ $(k < m)$, $Q \in \mathbb{R}^{m\times k}$ is unitary, and $R \in \mathbb{R}^{k\times n}$ is upper triangular, and Π is a permutation matrix that orders the columns so that the first k pivots, or "skeleton" columns, are at the left. It follows that $R_1 \in \mathbb{R}^{k\times k}$ is invertible, and (1.19) gives us the ID

$$A\Pi = [QR_1 \quad QR_1(R_1^{-1}R_2)] := [A(:, s) \quad A(:, s)T] = A(:, s)[I \quad T]. \tag{1.20}$$

Here, s denotes the index set of skeleton columns. If r is the index set of residual columns, then (1.20) states that

$$A(:, r) \approx A(:, s)T, \tag{1.21}$$

which is a form that is used extensively in the recursive skeletonization factorization.

The advantage of using the ID for compressing low-rank matrices is that one of the factors is simply a subset of the rows or columns of A. Hence it preserves the information of the matrix A, which is useful in situations where A is a discretization of an operator on a specific domain. In section 3.4, it is shown that the ID allows a particularly simple expression for some Hierarchical matrix factorizations.

More general information on compressing low-rank matrices can be found in [9].

1.3 CUR approximation algorithms

A CUR approximation of a matrix A is a particularly simple rank-revealing approximate factorization of the form

$$A = CUR, \tag{1.22}$$

where C is a subset of the columns of A, and R is a subset of the rows of A. CUR approximation is popular due to the fact that it is faster to compute than svd or QR based algorithms, and also because it is structure-preserving like the ID. In fact, taking, e.g., UR as X in section 1.2, the CUR factorization immediately gives the ID.

It must be noted, however, that it is difficult to find the optimal indices for the CUR factorization, hence the result of the approximation is often sub-optimal. However, there are both deterministic and randomized techniques that can reliably compute CUR approximations.

A variety of both deterministic and randomized methods for computing CUR approximations can be found in [28], [31], [45].

1.3.1 Adaptive cross approximation (ACA)

Adaptive cross approximation is a method of computing a CUR approximation of the form

$$A \approx A(:, J_{skel})(A(I_{skel}, J_{skel}))^{-1} A(I_{skel}, :), \tag{1.23}$$

where I_{skel} and J_{skel} are index sets of column and row skeletons respectively. The indices are selected based on a maximum absolute value search over the entries of the matrix. The search begins from the entry with the largest absolute value. The row and column corresponding to the entry are deleted via a rank one update

$$A' = A - \frac{1}{A(i,j)} A(:, j) A(i, :). \tag{1.24}$$

The search continues over the remaining entries, and the process is repeated until the maximum value found is less than the prescribed tolerance. This method is fast and very reliable, but relies on having access to all of the entries in A, which may be impossible when A cannot be stored in core memory.

For a matrix $A \in \mathbb{R}^{1024 \times 1024}$ generated by a squared exponential kernel, table 1.1 shows the accuracy achieved by ACA given a tolerance.

`tol`	$\lvert I_{skel} \rvert$	error
1e-3	4	8.8401e-03
1e-6	7	1.0372e-05
1e-9	10	1.3055e-09
1e-12	12	1.6540e-12
1e-15	14	5.2273e-15

Table 1.1: Error of ACA, and length of index set I_{skel} for increasing tolerance values.

2 Matrix identities

2.1 Inverting a block partitioned matrix

The following inversion lemma is used to invert a HODLR factorized matrix in section 3.3, and also to compute the posterior distribution of a Gaussian process in section 4.1.2.

Definition 2.1 (Schur complement)**.** Consider a block partitioned square matrix

$$A = \begin{pmatrix} P & Q \\ R & S \end{pmatrix}. \tag{2.25}$$

If P^{-1} exists, the Schur complement of block P is defined as

$$A/P := S - RP^{-1}Q. \tag{2.26}$$

Remark 2.1. The Schur complement A/P arises from performing block Gaussian elimination on A.

Lemma 2.1. If $M := (A/P)^{-1}$ exists, then

$$A^{-1} = \begin{pmatrix} \tilde{P} & \tilde{Q} \\ \tilde{R} & \tilde{S} \end{pmatrix}, \tag{2.27}$$

where

$$\tilde{P} = P^{-1} + P^{-1}QMRP^{-1}, \tag{2.28}$$
$$\tilde{Q} = -P^{-1}QM, \tag{2.29}$$
$$\tilde{R} = -MRP^{-1} \tag{2.30}$$
$$\tilde{S} = M, \tag{2.31}$$

2.2 Sherman-Morrison-Woodbury inversion formulas

There exist formulas to efficiently invert a rank-k update of a matrix A. First presented is the original inversion formula by Sherman, Morrison, and Woodbury. Next, a variation of the formula is introduced.

2.2.1 The matrix inversion lemma

The most general form of the formula is due to Woodbury [34]. This version is useful for inverting a matrix factorized by the HODLR factorization in section 3.3.

Lemma 2.2 (Woodbury matrix identity)**.** For $\underset{n\times n}{A}$, $\underset{n\times k}{U}$, $\underset{k\times k}{C}$, $\underset{n\times k}{V}$, and A invertible,

$$(A + UCV^*)^{-1} = A^{-1} - A^{-1}U(C^{-1} + V^*A^{-1}U)^{-1}V^*A^{-1}. \tag{2.32}$$

In particular, if $C = I$, and $k = 1$, then

$$(A + UV^*)^{-1} = A^{-1} - \frac{A^{-1}UV^*A^{-1}}{1 + V^*A^{-1}U}, \tag{2.33}$$

which is the form introduced by Sherman and Morrison [42].

The proof of this lemma involves the block inversion formula in (2.27).

Remark 2.2. The following lemma is a variation of the Woodbury identity which is used for inverting an additive matrix factorization such as the recursive skeletonization [32].

Lemma 2.3. (A variation of the Woodbury identity) If A invertible, and admits the factorization

$$\underset{n\times n}{A} = \underset{n\times k}{U}\ \underset{k\times k}{\tilde{A}}\ \underset{k\times n}{V^*} + \underset{n\times n}{D}, \tag{2.34}$$

then

$$\underset{n\times n}{A^{-1}} = \underset{n\times k}{E}\ \underset{k\times k}{(A+\hat{D})^{-1}}\ \underset{k\times n}{F^*} + \underset{n\times n}{G}, \tag{2.35}$$

where

$$\hat{D} = (V^*DU)^{-1}, \tag{2.36}$$
$$E = D^{-1}U\hat{D}, \tag{2.37}$$
$$F = (\hat{D}V^*D^{-1})^*, \tag{2.38}$$
$$G = D^{-1} - D^{-1}U\hat{D}V^*D^{-1}, \tag{2.39}$$

if all the inverses above exist. Also, $\operatorname{rank}(G) = N - K$.

The following proof follows [33, §13].

Proof. First observe that for two invertible matrices X, Y of the same size,

$$(X^{-1} + Y^{-1}) = Y - Y(X+Y)^{-1}Y, \tag{2.40}$$

which is simply the first version of the Woodbury identity (2.32), where $A = X^{-1}$, $U = I$, $C = Y^{-1}$, $V = I$. Next, use $X = \tilde{A}$, $Y = (V^*D^{-1}U)^{-1}$ for (2.40), and plug in the results into (2.32) to get (2.35). □

2.3 Weinstein–Aronszajn (Sylvester) determinant identity

When $A \in \mathbb{R}^{m\times m}$ is a low-rank update of the identity matrix, i.e.,

$$A = I_m + UV^\top, \tag{2.41}$$

where $U, V \in \mathbb{R}^{m\times k}$, $k < n$, the determinant of A can be computed in an efficient way by the following lemma.

Lemma 2.4 (Weinstein–Aronszajn identity)**.**

$$\det(I_m + UV^\top) = \det(I_k + V^\top U). \tag{2.42}$$

That is, the determinant of a large $(m \times m)$ matrix can be replaced with the determinant of the small $(k \times k)$ matrix. This lemma is used to find the determinant of a HODLR factorized matrix in section 3.3.

3 Hierarchical matrices

3.1 Taxonomy of $\mathcal{H}$-matrices

Matrices that arise from discretizing a boundary integral operator, or evaluating a kernel function often have hierarchical rank structure. Although these matrices are dense, matrix operations such as matrix-vector multiplication, and solving a system of equations can be performed in near-linear time by exploiting the hierarchical rank structure.

In this thesis, a class of rank-structured matrices known as $\mathcal{H}$-matrices (Hierarchical matrices) are considered. When an $\mathcal{H}$-matrix is partitioned into blocks, some of the off-diagonal blocks are of (numerical) low-rank, hence can be compressed using a low-rank approximation.

For the full taxonomy of $\mathcal{H}$-matrices, the concept of admissibility must be introduced. In weakly admissible formats, all off diagonal blocks are compressed at each level. Strongly admissibile formats refer to compressing only a subset of off-diagonal blocks at each level.

Also, $\mathcal{H}$-matrices can be categorized by the types of low-rank factors used to compress the off-diagonal blocks. The HODLR format compresses each off-diagonal block separately using low-rank factors. Under stricter conditions, the off-diagonal blocks on the same block-row can be expressed using common low-rank factors, which leads to the HBS format.

Using the terminology defined above, Hierarchical matrices can be categorized as in figures 3.1 and 3.2, which are based on the works of Martinsson [33], and Ambikasaran [1].

	Weakly admissible	Strongly admissible
General basis functions	HODLR	$\mathcal{H}$-matrices (not HODLR or $\mathcal{H}^2$)
Nested basis functions	HBS/HSS	$\mathcal{H}^2$-matrices

Figure 3.1: Types of hierarchical matrices

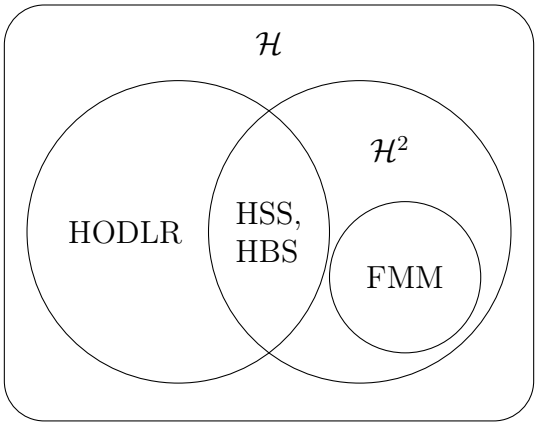

Figure 3.2: Relationship between $\mathcal{H}$-matrices

3.2 k-d tree structure and indexing notation

In order to utilize the isometric variance property of certain kernels, it is important to pre-order the data based on location. In particular, data points with close input should be clustered together. This can be done by ordering the data based on a k-d tree structure.

To make the exposition simple, it is convenient to introduce an index system. Denote the input space as Ω, and let $\mathcal{I}_\Omega$ be the index set of all inputs. In later sections, $\mathcal{I}_{\ell;j}$ will denote the index set of the j-th subdomain at level ℓ. Note that for the HODLR decomposition, the levels proceed from coarse to fine, which means that the size of the index sets will become smaller as the level increases. On the other hand, for the recursive skeletonization factorization, the levels proceed from fine to coarse, which means the size of the index sets become larger as the level increases. The finest nodes are often called the leaf nodes. If a node is a subset of a greater node, the node is called a child of the greater parent node. Sibling nodes are nodes that share parent nodes.

In this thesis, only perfect trees are considered. A perfect tree is a binary structure where each node has exactly two child nodes, and the leaf nodes are all in the same level. In this setting, a k-d tree sorting can be simply done by first sorting the data set with respect to the first dimension and equally dividing the domain into the size of leaf nodes. Next, each node in the data set is sorted separately with respect to the second dimension, and are again equally divided into the size of leaf nodes. This process is repeated up until dimension k. A detailed review of various k-d tree data structures can be found in [41].

3.3 The hierarchical off-diagonal low-rank (HODLR) framework

Definition 3.1 (HODLR condition)**.** Let A be a matrix of size $n \times n$, and let k be an integer such that $k < n$. We say that A is a HODLR matrix with rank k if either of the following conditions holds:

- A is itself of size at most $2k \times 2k$.
- If A is partitioned into four equal sized blocks,

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix},$$

  then A_{12} and A_{21} have rank at most k and A_{11} and A_{22} are also HODLR matrices of rank k.

The off-diagonal low rank blocks can be compressed in the form

$$A_{ij} = UV^*, \tag{3.43}$$

where $U, V \in \mathbb{C}^{n \times \ell}$, and ℓ is typically the target rank that is determined by the

desired precision. Hence, the matrix A is approximated as

$$A \approx \begin{pmatrix} A_{11} & UV^* \\ VU^* & A_{22} \end{pmatrix},$$

which can be viewed as a low-rank update to a block diagonal matrix:

$$A \approx \begin{pmatrix} A_{11} & 0 \\ 0 & A_{22} \end{pmatrix} + \begin{pmatrix} 0 & U \\ V & 0 \end{pmatrix} \begin{pmatrix} U^* & 0 \\ 0 & V^* \end{pmatrix}. \tag{3.44}$$

The inverse of A can be efficiently computed via the Woodbury matrix identity (2.32). Since A is a HODLR matrix, the off diagonal blocks of A_{11}, and A_{22} can be compressed once again, which leads to a hierarchical compression scheme.

3.3.1 The HODLR factorization and solver

An effective way to invert a hierarchically compressed matrix is to construct a HODLR factorization, which can be done by recursively factoring out the block diagonals from the previous level. For example, the first level of this factorization is

$$\begin{aligned} A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} &= \begin{pmatrix} A_{11} & 0 \\ 0 & A_{22} \end{pmatrix} \begin{pmatrix} I & A_{11}^{-1}A_{12} \\ A_{22}^{-1}A_{21} & I \end{pmatrix} \\ &\approx \begin{pmatrix} A_{11} & 0 \\ 0 & A_{22} \end{pmatrix} \begin{pmatrix} I & A_{11}^{-1}UV^* \\ A_{22}^{-1}VU^* & I \end{pmatrix}. \end{aligned} \tag{3.45}$$

One can repeat this procedure to obtain a two-level factorization by splitting the diagonal blocks A_{11} and A_{22} into four smaller blocks, then factoring out the four smaller diagonal blocks. In the end, the matrix A is decomposed into a chain of factors

$$A = A_\kappa A_k A_{k-1} \dots A_1, \tag{3.46}$$

and the inverse of A is

$$A^{-1} = A_1^{-1} \dots A_{k-1}^{-1} A_k^{-1} A_\kappa^{-1}. \tag{3.47}$$

In the whole process, the only matrices that need to be stored are the low-rank factors $U_i^{(j)}, V_i^{(j)}$. For example, the factor A_1 looks like

$$\begin{pmatrix} I & A_{11}^{-1}UV^* \\ A_{22}^{-1}VU^* & I \end{pmatrix}.$$

The computation of A_{11}^{-1}, A_{22}^{-1} require information from all the factors $A_2, \dots, A_\kappa$, but storing all of the full inverse diagonal blocks is infeasible, because they quickly become too large. Only the low-rank factors U and V are to be stored, and the full inverse matrices should never be formed explicitly. This is because the Woodbury matrix identity (2.32) can be applied repeatedly only using the low-rank factors U and V. That is, instead of forming each inverse of the diagonal blocks, one applies the matrices occurring in the Woodbury formula to all the remaining $U_i^{(j)}, V_i^{(j)}$'s in the right locations, hence updating the low-rank matrices in the remaining factors

of the HODLR factorization.

Within the HODLR factorization algorithm, finding a low-rank approximation for the off diagonal matrices in the form of equation (3.43) requires the most computational work, especially because the first few levels of compression involve very large matrices. Common techniques for compressing the off-diagonal blocks include adaptive cross approximation (ACA) in section 1.3, and the interpolative decomposition in section 1.2. SVD based low-rank approximation algorithms typically scale like $\mathcal{O}(n^3)$, which make them infeasible for large matrices. However, randomized versions such as the randomized approximate SVD in section 1.1, or the matrix peeling algorithm in section 3.5 can be implemented efficiently. Also, an approximate spectral decomposition can be obtained by computing a truncated eigenfunction expansion of the kernel function, which is discussed in section 6.3. A comparison of such compression schemes is made section 7.

3.3.2 Computational complexity

Assuming that an $\mathcal{O}(n \log n)$ factorization scheme is used for the compression of the off diagonal blocks, repeating over $\kappa \approx \log_2 n$ levels results in an $\mathcal{O}(n \log^2 n)$ algorithm for the HODLR factorization.

3.3.3 Computing the determinant

Suppose a HODLR factorization of $C \in \mathbb{R}^{m \times m}$ is obtained:

$$C \approx C_\kappa C_{\kappa-1} \dots C_0. \tag{3.48}$$

Since each factor $C_i \in \mathbb{R}^{m \times m},\ i = 0, \dots, \kappa_1$ is a low-rank update of the identity, the Weinstein-Aronszajn identity in (2.42) can be used to compute the determinants of K_i can be computed in $\mathcal{O}(n)$ time each, which amounts to $\mathcal{O}(\kappa n)$ in total. Considering that $k \sim \log_2(n)$, the determinant computation has $\mathcal{O}(n \log n)$ complexity.

Algorithm 3.1 is a pseudocode for computing the inverse of a HODLR factorization of a Gaussian process covariance matrix, along with the log determinant. This method is used to compute the score functions for hyperparameter MLE, the posterior distribution, and the GP density in part II of this thesis.

3.4 Recursive skeletonization factorization (RSF)

Following the exposition of Ho & Ying [22], a recursive skeletonization factorization (RSF) of the matrix A is of the form

$$A = LDU, \tag{3.49}$$

where L and U are products of unit block-triangular matrices and permutation matrices, and D is a block diagonal matrix. When A is symmetric positive semi-definite, which is the case for covariance matrices, the factorization becomes a generalized block LDL decomposition, i.e., $A = LDL^\top$. RSF allows fast matrix-vector multiplication and inversion via sparse matrix operations. Once an RSF of A is obtained,

Algorithm 3.1 Applying inverse HODLR factorization to a vector

Require: p: Size of smallest diagonal block in K_κ
Require: ϵ: Precision of low-rank approximation
Require: Matrix entry evaluation routine (e.g. kernel function)
Require: training data $(\mathbf{x}, \mathbf{y})$

$\kappa \leftarrow \lfloor \log_2(n/p) \rfloor$
for $j = 1$ to κ **do**
    for $i = 1$ to 2^j **do**
        Compute low rank factorizations U_i^j, V_i^j of all off diagonal blocks
    end for
end for
Form block diagonal matrix K_κ from block diagonals of K, then apply inverse of each block to $\mathbf{y}$, and every remaining $U_i^{(l)}$ or $V_i^{(l)}$ $(l < \kappa)$ in the right place.
for $j = \kappa$ to 1 **do**
    for $i = 1$ to 2^j **do**
        Compute inverse of i-th block in K_j via Woodbury matrix identity, and apply to $\widetilde{\mathbf{y}}$ and every remaining $U_i^{(l)}$ or $V_i^{(l)}$ $(l < j)$ in the right place. $\{\widetilde{\mathbf{y}}$ is all the previous inverse matrices applied to $\mathbf{y}\}$
    end for
end for

it is also straightforward to compute $\det(A) = \det(D)$, and the cholesky factor $R = D^{1/2}L^\top$. Hence, RSF performs the necessary tasks that are required for Gaussian process regression. The types of computational tasks required for Gaussian process regression is discussed further in section 4.2.

It is worth noting that the RSF can be formulated in a way that is mathematically equivalent to recursive skeletonization (RS) [32, 9]. The difference is that RSF is a multiplicative factorization that relies on sparse matrix operations, whereas RS is an additive factorization which uses the Woodbury matrix identity to compute the inverse. The RSF is better suited for the purpose of this thesis since the multiplicative format of the RSF makes it easy to compute the log determinant and square root matrix of A. On the other hand, the RSF requires certain diagonal sub-blocks to be invertible (which will nevertheless be the case in the experiments conducted in this thesis), which is a somewhat stricter requirement than that of RS. RSF is also the factorization that is used in Minden et al. [36] for Gaussian processes MLE.

Let Ω denote the set of all input data, or degrees of freedom (DOFs). Once the data is ordered in a k-d tree format as in section 3.2, each DOF $x_j \in \Omega,\ j = 1, \ldots, N$ is given an additional label $s = \{0, 1\}$ denoting whether x_j is in the skeleton set (active $\iff s = 1$), or in the residual set (inactive $\iff s = 0$). Hence, the data matrix X has total dimension $N \times (d + 1)$. Let $\mathcal{I}_{\ell;i}$ denote the index set of the i-th subdomain at level ℓ. The length of $\mathcal{I}_{\ell;i}$ is determined by the number of levels L, which is pre-specified. More specifically in a perfect tree setting, $|\mathcal{I}_{\ell;i}|$ becomes twice as large after each level, and in the L-th (last) level, there are two subdomains, i.e., $\mathcal{I}_\Omega = \mathcal{I}_{L;1} \cup \mathcal{I}_{L;2}$. Hence, $|\mathcal{I}_{\ell;i}| = N/2^{(L-\ell+1)}$. Also define $\hat{\mathcal{I}}_{\ell;i}$ to be the set of indices corresponding to active DOFs in the i-th subdomain at level ℓ.

The steps of the algorithm are written out below. Unlike the HODLR factorization in section 3.3, which proceeds from coarser levels to finer levels, RSF proceeds from fine to coarse.

Level $\ell = 1$:

In the first level, all DOFs are active, i.e., $\cup_i \hat{\mathcal{I}}_{1;i} = \Omega$. For simplicity, consider a two-level factorization of a matrix $A \in \mathbb{R}^{n \times n}$. Then $|\mathcal{I}_{1:i}| = n/4$. For each subdomain $\mathcal{I}_{1;i}$, sparsity is introduced via the following steps.

First, consider the sub-block $\tilde{A} := A(\mathcal{I}^c_{1:1}, \mathcal{I}_{1:1})$. Using the ID in section 1.2, this sub-block can be decomposed into

$$\tilde{A}\tilde{P} = \tilde{A}_{1:k}[I \quad T], \tag{3.50}$$

where $\tilde{P}$ is a permutation matrix that re-orders the DOFs in $\mathcal{I}_{1;1}$ so that the k skeleton points come first. Let P denote the $n \times n$ permutation matrix that has $\tilde{P}$ in the $(\mathcal{I}_{1;1}, \mathcal{I}_{1;1})$ block. After the rows and columns of $\tilde{A}$ are sorted, consider a block partitioning of A as in figure 3.3.

$$P^\top A P \quad = \quad \begin{array}{|c|c|c|l} \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \\ \hline A_{pp} & A_{ps} & A_{pr} & \mathcal{I}^c_{1;1} \\ \hline A_{sp} & A_{ss} & A_{sr} & \\ \hline A_{rp} & A_{rs} & A_{rr} & \mathcal{I}_{1;1}\tilde{P} \\ \hline \end{array}$$

Figure 3.3: The DOFs in $\mathcal{I}_{1;1}$ are re-ordered so that the k skeleton DOFs come at the beginning.

The s in the labels of the A sub-blocks stands for "skeleton", and r stands for "residual". The goal is to eliminate the blocks A_{pr}, A_{sr}, A_{rp}, and A_{rs} using block Gaussian elimination. Since $A_{pr} \approx A_{ps}T$ by the ID, the A_{pr}, and A_{rp} blocks can be eliminated with a block-row and a block-column operation as depicted in figure 3.4.

The block-column operation matrix Q is

$$Q = \begin{bmatrix} I & & \\ & I & -T \\ & & I \end{bmatrix}.$$

$$Q^\top P^\top APQ \quad \approx \quad \begin{array}{|c|c|c|} \hline & & \\ A_{pp} & A_{ps} & \mathbf{0} \\ & & \\ \hline A_{sp} & A_{ss} & B_{sr} \\ \hline \mathbf{0} & B_{rs} & B_{rr} \\ \hline \end{array}$$

Figure 3.4: After eliminating A_{pr} and A_{rp}.

The three modified blocks are

$$\begin{aligned} B_{sr} &= A_{sr} - A_{ss}T \\ B_{rs} &= A_{rs} - T^\top A_{ss} \\ B_{ss} &= A_{rr} - T^\top A_{sr} - A_{rs}T + T^\top A_{ss}T. \end{aligned}$$

Now, take the LDL decomposition of B_{rr}, namely

$$B_{rr} = L\hat{D}L^\top. \tag{3.51}$$

Note that the L here is not the same as the one in the expository equation (3.49). $\hat{D}$ is a diagonal matrix. Next, the blocks B_{rs} and B_{sr} are eliminated with a block-row and a block-column operation as depicted in figure 3.5.
The block-row operation matrix S is (assuming B_{rr} is invertible),

$$S = \begin{bmatrix} I & & \\ & I & \\ & -L^{-\top}\hat{D}^{-1}L^{-1}B_{rs} & L^{-\top} \end{bmatrix} = \begin{bmatrix} I & & \\ & I & \\ & & L^{-\top} \end{bmatrix} \begin{bmatrix} I & & \\ & I & \\ & -\hat{D}^{-1}L^{-1}B_{rs} & I \end{bmatrix}. \tag{3.52}$$

The modified sub-block is

$$B_{ss} = A_{ss} - B_{sr}B_{rr}^{-1}B_{rs}. \tag{3.53}$$

It is now clear that the advantage of using the ID for eliminating the A_{pr} and A_{rp} blocks is that it leaves A_{ps} and A_{sp} unchanged. As the algorithm proceeds throughout a single level, more zero rows will be introduced to A_{ps}, and a sparsified version of A_{ps} can be used again in later levels.

$$S^\top Q^\top P^\top APQS \quad \approx \quad \begin{array}{|c|c|c|} \hline A_{pp} & A_{ps} & \mathbf{0} \\ \hline A_{sp} & B_{ss} & \mathbf{0} \\ \hline \mathbf{0} & \mathbf{0} & \hat{D} \\ \hline \end{array}$$

Figure 3.5: After eliminating B_{sr} and B_{rs}. $\mathcal{I}_{1;1}$ is sparsified, and $\hat{D}$ is decoupled from the system.

The remaining sub-domains $\mathcal{I}_{1;i},\ i = 2, 3, 4$ can be sparsified in the same way. If A_{ps} is stored, then it is important to cancel out the rows of A_{ps} that correspond to the residual DOFs of the following sub-domains.

Levels $\ell = 2, \ldots, L$:

In the proceeding levels, the process is identical, except for the fact that compression occurs within the parent nodes. The decoupled DOFs no longer need to be considered. Since only the active DOFs are used, the size of the block B_{ss} is different than that in level 1. For a uniformly generated 2-d data set with $N = 2^{10}$, the active DOFs after each level are depicted in figure 3.6. The sparsity pattern of the sparsified matrix after each level is depicted in figure 3.7.

The final factorization can be written succinctly as

$$A \approx \hat{L} D \hat{L}^\top, \tag{3.54}$$

where D is a block diagonal matrix that is nearly diagonal, and

$$\hat{L} = \prod_{\ell=1}^{L} \left(\prod_{i=1}^{2^{L-\ell+1}} S_{\ell;i}^{-1} Q_{\ell;i}^{-1} P_{\ell;i}^{-1} \right). \tag{3.55}$$

Note that $\hat{L}$ is not necessarily lower triangular, but is a product of triangular matrices and permutation matrices.

The block-row and block-column operation matrices $S_{\ell;i}$ and $Q_{\ell;i}$ that occur in each step are fast to apply via sparse multiplication, and easy to invert by negating the off diagonal blocks. The permutation matrices are also fast to apply, and easy to invert by simply taking the transpose. The (nearly) diagonal matrix D can be

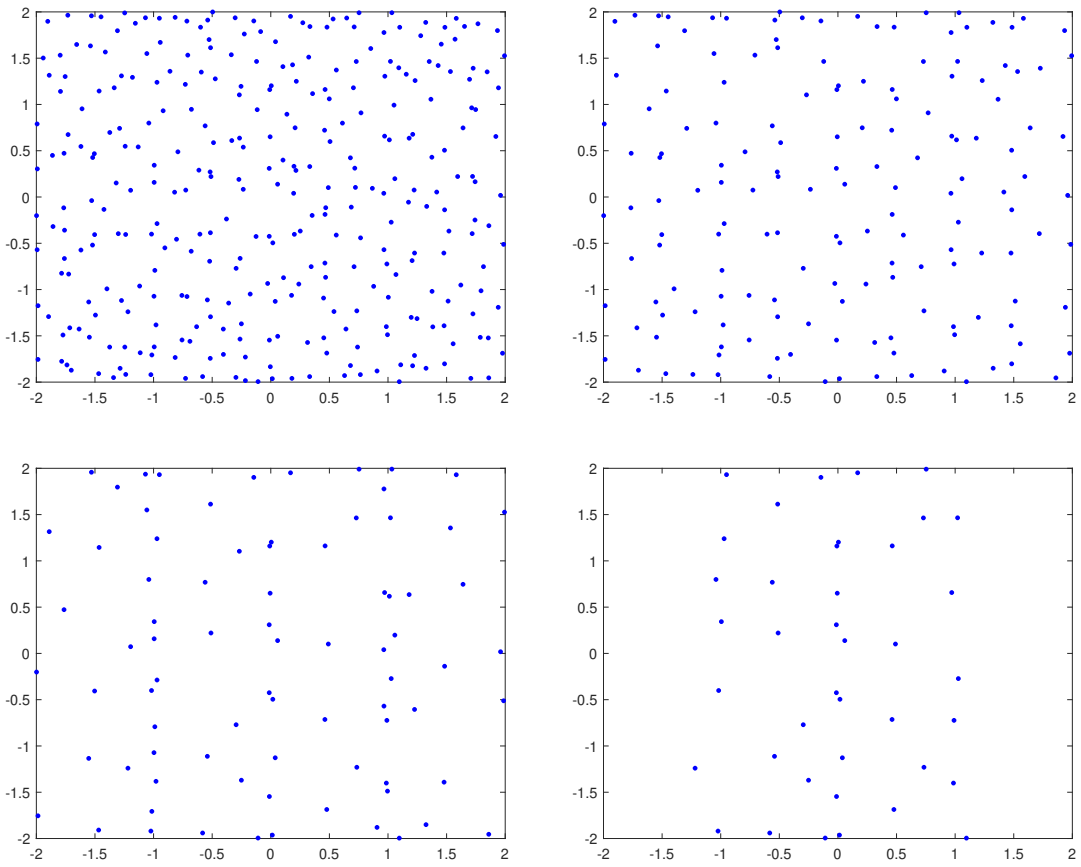

Figure 3.6: A 2-d example of active DOFs after levels 1, 2, 3, 4 of RSF respectively. Note that the remaining DOFs at each level appear to be close to the boundary of each subdomain.

applied and inverted efficiently in $\mathcal{O}(n)$ operations.

3.4.1 Computational complexity and accelerating the recursive skeletonization factorization

Both the RSF and RS are $\mathcal{O}(n^2)$ algorithims if implemented naively [29]. The proxy surface compression technique in [32] accelerates the algorithm to $\mathcal{O}(n^{3/2})$. The additive RS algorithm can be further accelerated by randomization such as in [33, §17.3], which has $\mathcal{O}(n)$ complexity, provided a fast $\mathcal{O}(n)$ matrix-vector multiplication scheme such as the FMM is available.

3.4.2 Higher dimensions

For higher dimensional data, the rank of the Gram matrix is much higher than one-dimensional data, which makes it harder to achieve accurate results using hierarchical matrix factorizations. This is often referred to as the curse of dimensionality, and is demonstrated in section 4.1.4. For 3-d data, the key is to compress further the off-diagonal blocks with lower rank. Some hierarchical matrix factorizations such

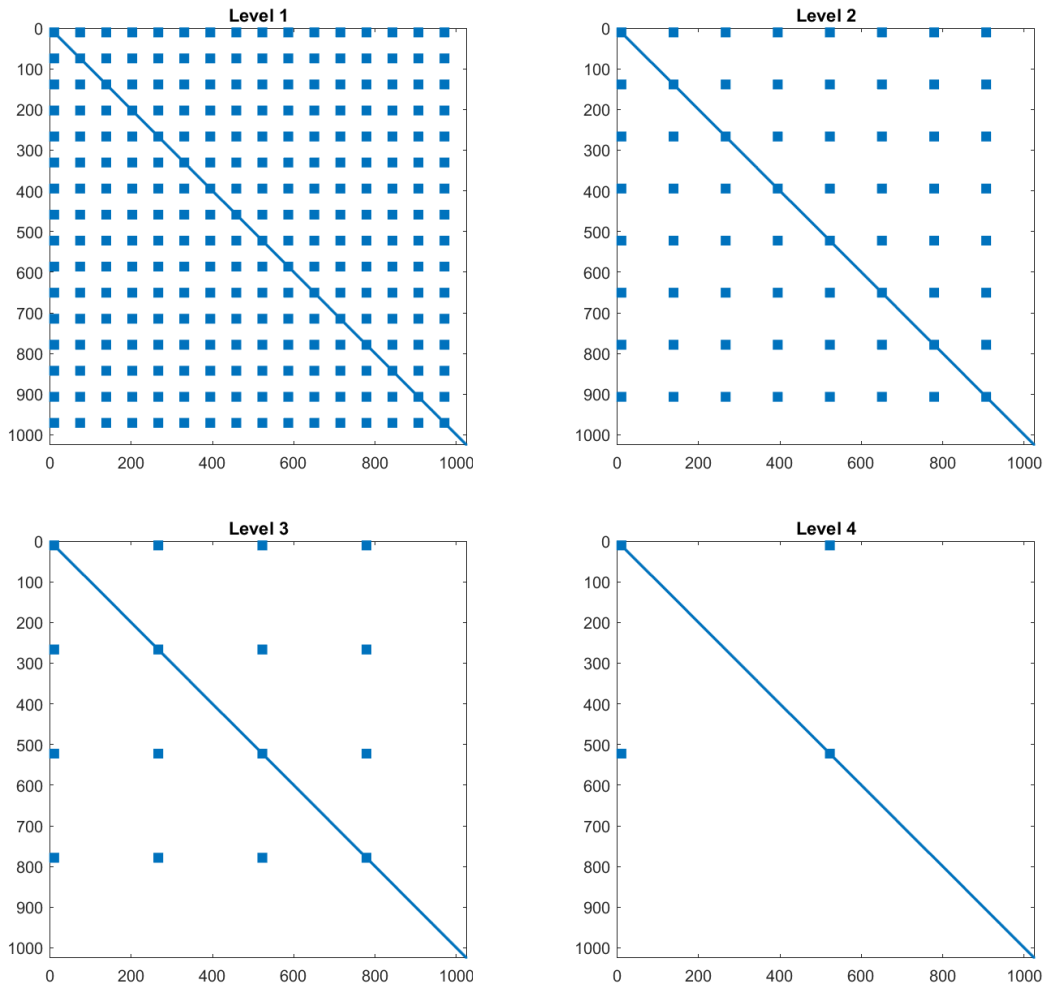

Figure 3.7: Sparsity pattern of the near diagonal matrix after levels 1, 2, 3, 4 of RSF respectively. Note that the off diagonal blocks can be combined into a larger diagonal block by a single additional row and column permutation.

as the hierarchical interpolative factorization in [22] can be considered. If physical knowledge of the data is known, and can be exploited, certain blocks can be chosen to be highly compressed.

3.4.3 Computing the determinant

The LU factors do not contribute to the determinant, because they are products of unit block-triangular matrices which have determinant 1, and block-diagonal matrices with blocks that are unitary, hence also have determinant 1. Therefore, the determinant is obtained by computing the determinant of the innermost near-diagonal matrix D.

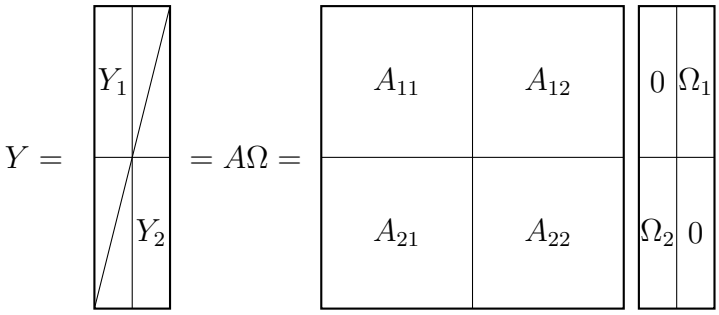

Figure 3.8: The range-finding process in Level 1 of matrix peeling. The goal is to simultaneously find the approximate bases of A_{12} and A_{21}, which are Y_1, and Y_2 respectively.

3.5 Trace estimation via matrix peeling

As mentioned in section 3.4, additive hierarchical matrix factorizations such as recursive skeletonization were not employed in this thesis to solve a system $Cx = b$ since they do not allow a fast computation of $\log\det(C)$. However, an additive hierarchical factorization known as the "matrix peeling algorithm" can be used instead to estimate the trace of a matrix when the matrix can only be accessed through matrix-vector multiplies. Trace estimation is a central task in the maximum likelihood estimation of kernel hyperparameters, and will be discussed in detail in chapter 5. Here, the matrix peeling algorithm is introduced.

As the name suggests, the goal of matrix peeling is to compress off-diagonal blocks, and subtract their effects from the original matrix to extract the diagonal blocks. Matrix peeling can be applied to both the HODLR framework, and the HBS framework. Here, the idea is developed using the HODLR framework. Unlike the recursive skeletonization factorization of section 3.4, and similar to the HODLR factorization of section 3.3, matrix peeling progresses top-down, namely, from coarse to fine boxes.

Matrix peeling compresses the off-diagonal sibling boxes at each level simultaneously using a randomized low-rank approximation scheme described in 1.1, which is from [20]. The following summary of the algorithm follows [30]. Again, a perfect tree structure is used.

Level 1:

First, generate a random matrix $\Omega \in \mathbb{R}^{n\times k}$, where k is the target rank plus an oversampling parameter. Ω has zero blocks on the diagonal. Then, split a HODLR matrix $A \in \mathbb{R}^{n\times n}$ into quadtree sub-blocks, and multiply A to Ω to get the approximate basis matrix Y. This procedure is depicted in figure 3.8.

Next, orthonormalize the columns of Y_1, and Y_2 in figure 3.8 to get the orthogonal bases U_1 and U_2. These blocks are concatenated into a matrix

$$U = \begin{bmatrix} 0 & U_1 \\ U_2 & 0 \end{bmatrix}. \tag{3.56}$$

Let $A^{(\ell)}$ denote the off-diagonal sibling sub-matrix of A at level ℓ, namely the matrix

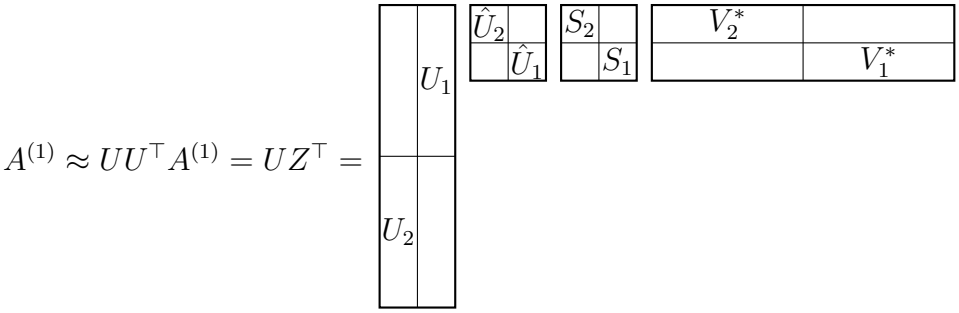

Figure 3.9: The factorization process in Level 1 of matrix peeling.

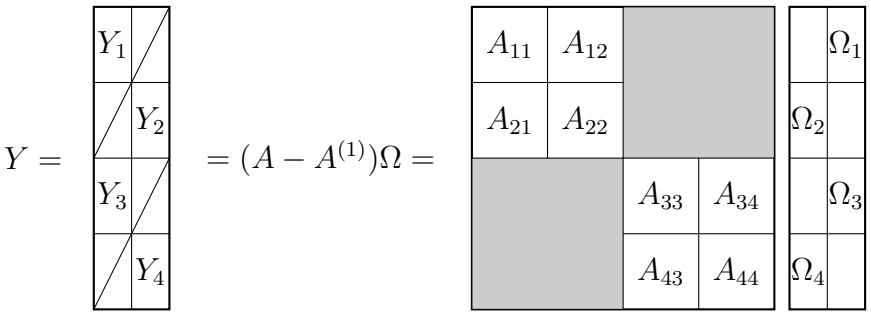

Figure 3.10: The range-finding process in Level 2 of matrix peeling. The goal is to simultaneously find the approximate bases of $A_{12}, A_{21}, A_{34}, A_{43}$, which are Y_1, Y_2, Y_3, Y_4 respectively. Note that the gray area is suppressed by subtracting $A^{(1)}$ from A.

with only the off-diagonal sibling blocks that are to be compressed at level ℓ. From the approximation

$$A^{(1)} \approx UU^\top A^{(1)}, \tag{3.57}$$

compute $Z = A^\top U$, then take the economy-sized svd of each nonzero block of $Z^\top$ to obtain the factorization in 3.9.

This can be used to find an approximate rank-revealing factorization of both off-diagonal siblings A_{12} and A_{21}, namely

$$A^{(1)} = \tilde{U}SV, \tag{3.58}$$

where $\tilde{U} = U\hat{U}$. Hence in $A - A^{(1)}$, the diagonal blocks A_{11}, and A_{22} are extracted, and the off-diagonal blocks are suppressed.

Higher levels:

At level $\ell > 1$, the first step is again to find the approximate basis matrices of the off-diagonal sibling boxes $\{A_\tau\}$ that are not yet suppressed. Assuming $\ell = 2$, build a random matrix Ω as depicted in figure 3.10, and apply $A - \sum_{i=1}^{\ell-1} A^{(i)}$ to Ω. The $A^{(i)}$, $i = 1, \ldots, \ell - 1$ must be subtracted from A, otherwise the matrix Y in figure 3.10 is incorrectly altered by the effects of the parent-level off-diagonal blocks. This means that the low-rank factors of all of the $A^{(i)}$-s have to be stored, and called every time $A - \sum_{i=1}^{\ell-1} A^{(i)}$ is applied to a matrix. Note that $A^{(i)}$ is never formed explicitly inside the algorithm.

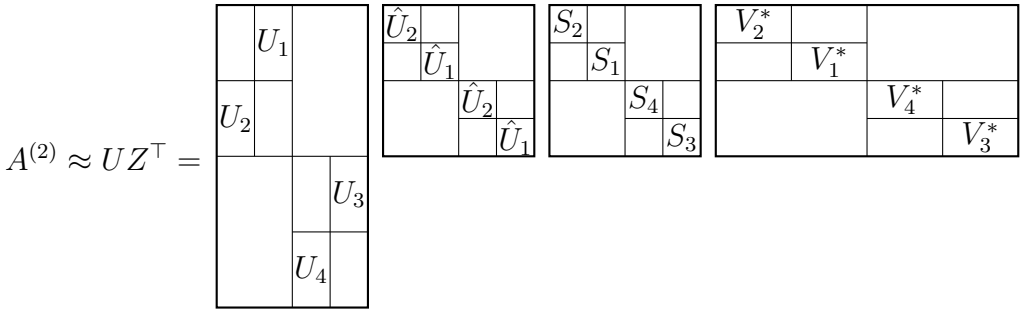

Figure 3.11: The factorization process in Level 2 of matrix peeling. Note that the low-rank factors are getting larger, but can be stored in sparse format to save memory. Also, in practice, the factors U and V are usually much taller and narrower than depicted in this figure.

The labeled sub-blocks of Y are orthonormalized, and the orthogonal bases $U_i,\ i = 1, \ldots, 4$ are obtained, and concatenated into a larger matrix U as in figure 3.11. Then, the off-diagonal sibling sub-matrix at level 2 can be approximated by

$$A - A^{(1)} = A^{(2)} \approx UU^\top (A - A^{(1)}). \tag{3.59}$$

Taking $Z = (A - A^{(1)})^\top U$, and computing the economy-sized svd of each non-zero block, $A^{(2)}$ can be approximated as in figure 3.11. This procedure is repeated over incresing levels until $A - \sum_{i=1}^{\ell-1} A^{(i)}$ has diagonal blocks of prescribed size.

The sparsity pattern of $A - \sum_{i=1}^{\ell} A^{(\ell)}$ up to a threshold is plotted in figure 3.12 for demonstrative purposes.

3.5.1 Extracting the trace

After L levels of peeling, the matrix $A - \sum_{i=1}^{L} A^{(i)}$ will be approximately block diagonal as depicted in figure 3.12, although $A - \sum_{i=1}^{L} A^{(i)}$ is not explicitly formed. In order to extract the diagonal blocks, and hence the trace, construct a matrix $\Omega \in \mathbb{R}^{n \times m}$, where n is the size of the system, and m is the size of the boxes at the leaf level. For $j = 1, \ldots, n/m$, set each $\Omega((j-1)m+1 : jm, :)$ sub-block of Ω as the identity matrix of size m. Then, compute the product

$$D = \left(A - \sum_{i=1}^{L} A^{(i)} \right) \Omega, \tag{3.60}$$

which results in the (approximate) diagonal blocks of A being placed in the sub-blocks of D. Now, the trace can be computed recursively via

$$\mathrm{tr}(A) \approx \sum_{j=1}^{n/m} \mathrm{tr}(D((j-1)m+1 : jm, :)). \tag{3.61}$$

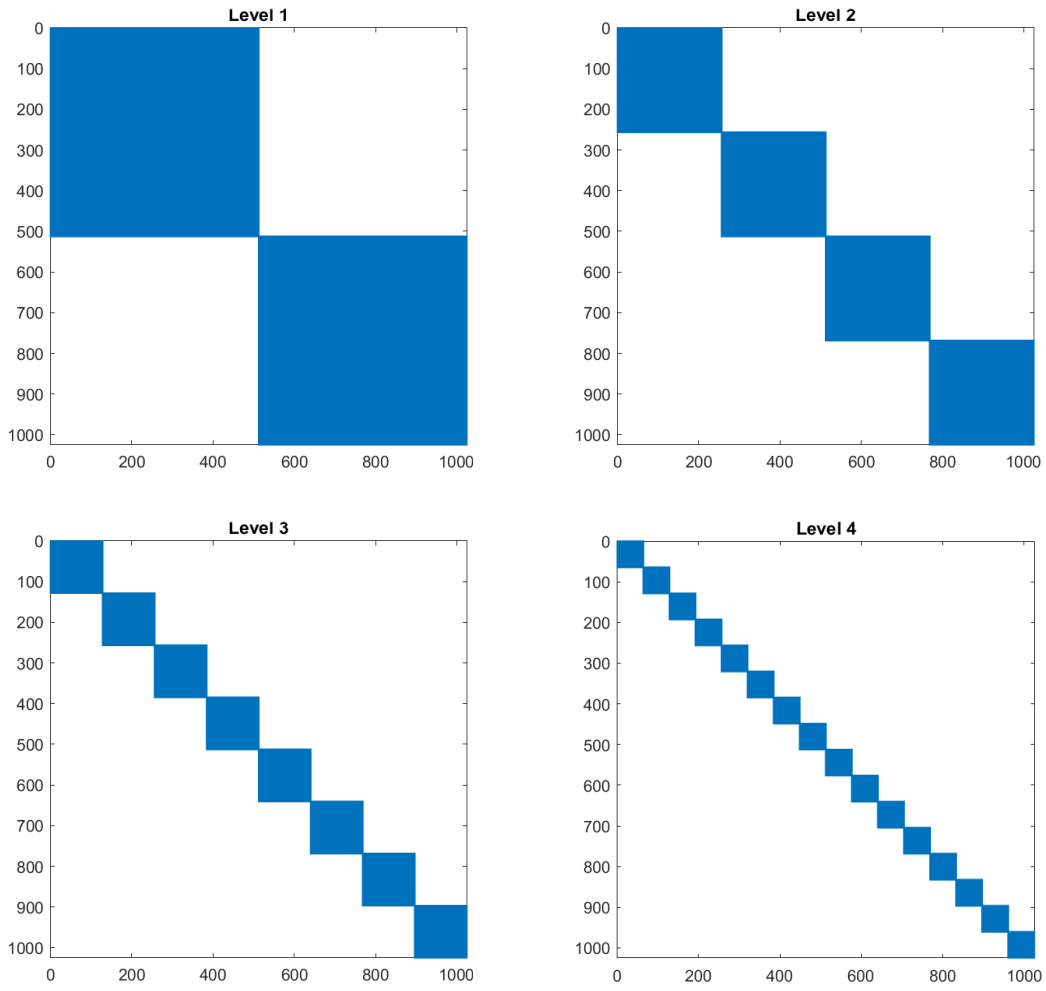

Figure 3.12: Sparsity pattern of $A - \sum_{i=1}^{\ell} A^{(i)}$ (up to a small threshold) after levels 1,2,3,4 of HODLR matrix peeling.

3.5.2 Computational complexity

Following [30], [33], let T_{mult} be the cost of computing a matrix-vector multiply, and T_{rand} be the cost of generating a pseudorandom number from a standard Gaussian distribution, and T_{flop} be the cost of a floating point operation. In the HBS setting, there are $\mathcal{O}(k \log n)$ matrix-vector multiplies, $\mathcal{O}(kn \log n)$ random number generations, and $\mathcal{O}(k^2 n \log n)$ flops, resulting in

$$T_{total} \sim T_{mult} \times k \log n + T_{rand} \times kn \log n + T_{flop} \times k^2 n \log n. \tag{3.62}$$

Provided there is access to a fast $\mathcal{O}(n)$ matrix-vector multiply scheme, the total asymptotic cost will be $\mathcal{O}(n \log n)$. Comparing this with the cost of randomized compression of HBS matrices in [33, §17.3], which is $\mathcal{O}(n)$, matrix peeling is more costly by a log factor, but has the advantage that it does not require the evaluation of individual matrix entries.

3.6 References

Additional resources regarding hierarchical matrices include [6, 19]. Resources for various Hierarchical matrix factorizations include [27, 47, 14, 23, 2, 51, 50, 15, 7].

Part II Fast Frameworks for Gaussian processes

4 Overview of Gaussian processes

4.1 Gaussian processes for statistical inference

Consider a stochastic process

$$\{f(x) \ : \ x \in \mathcal{X}\}, \tag{4.1}$$

where $\mathcal{X}$ is an index set, which will serve the role of input space in the inference setting. A stochastic process is Gaussian if every finite collection of the random variables

$$\mathbf{f} = (f(x_1), \ldots, f(x_m)) \tag{4.2}$$

is a multivariate Gaussian random variable. For practical applications, the entries of the covariance matrix of $\mathbf{f}$, namely $\Sigma = \mathrm{cov}(\mathbf{f}, \mathbf{f})$, are assumed to be generated by a rule

$$k(x_i, x_j) = \mathrm{cov}(f(x_i), f(x_j)), \tag{4.3}$$

where the function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ that describes this rule is commonly called a covariance function, or kernel. In general, a matrix that is generated by a kernel is called a Gram matrix.

For the purpose of statistical inference, Gaussian processes have several advantages. First, since the Gaussian is extremely well-studied, the predictive distribution of outputs, as well as the likelihood function can be expressed analytically. Secondly, the choice of kernel is the only assumption required to perform statistical tasks with Gaussian processes. This means the model adapts to any type of data without making assumptions about the true distribution of the data.

According to Rasmussen & Williams [38], there are two equivalent pathways leading to the same predictive distribution. One is the Bayesian linear model approach, and the other is the function-modeling approach. Both approaches are summarized below.

4.1.1 Bayesian linear model formulation

Consider a linear model with i.i.d. Gaussian noise that predicts the output y_* for test input values x_* using training data (X, y) defined by

$$X := [x_i, \ldots, x_n]^\top \in \mathbb{R}^{n \times m}, \quad \text{and} \quad y \in \mathbb{R}^n. \tag{4.4}$$

Namely, the model of interest is

$$y_i = \hat{y}_i + \varepsilon_i, \tag{4.5}$$

where

$$\hat{y}_i = \phi(x_i)^\top w, \quad \text{and} \quad \varepsilon_i \overset{\text{i.i.d.}}{\sim} \mathcal{N}(0, \sigma^2). \tag{4.6}$$

The vector

$$\phi(x_i) := [\phi_1(x_i), \ldots, \phi_k(x_i)]^\top \tag{4.7}$$

is the set of features, or basis functions evaluated at x_i. Also, the matrix of tabulated features is denoted as

$$\Phi(X) := [\phi(x_1), \ldots, \phi(x_n)]^\top \in \mathbb{R}^{n \times m}. \tag{4.8}$$

In ordinary least squares regression, the goal is to find the projections w that minimize a loss function. In contrast, the Bayesian approach aims to maximize the posterior density of w conditioned on $\Phi(X)$ and y, which is computed by Bayes' rule:

$$\mathrm{P}(w|X, y) = \frac{\mathrm{P}(y|X, w)\,\mathrm{P}(w)}{\mathrm{P}(y|X)}. \tag{4.9}$$

The Bayesian approach is directly related to Gaussian processes. In particular, due to the i.i.d. mean zero Gaussian noise,

$$\mathbb{E}[y_i] = \hat{y}_i = \phi(x_i)^\top w, \quad \mathrm{Var}[y_i] = \sigma^2. \tag{4.10}$$

Hence, the conditional density of y in (4.9) is expressed as

$$\begin{aligned} \mathrm{P}(y|X, w) &= \prod_{i=1}^{n} \mathrm{P}(y_i|X, w) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{\left(y_i - \phi(x_i)^\top w\right)^2}{2\sigma^2}\right) \\ &= \frac{1}{(2\pi\sigma^2)^{n/2}} \exp\left(-\frac{1}{2\sigma^2}|y - \Phi(X)^\top w|^2\right), \end{aligned} \tag{4.11}$$

which is the density of $\mathcal{N}(\Phi(X)^\top w, \sigma^2 I)$. If a zero-mean Gaussian prior is placed on w, namely,

$$w \sim \mathcal{N}(0, \Sigma), \tag{4.12}$$

then it follows from formula (4.9) that the posterior distribution of w is

$$\begin{aligned}
\mathrm{P}(w|X,y) &\propto \mathrm{P}(y|X,w)\,\mathrm{P}(w) \\
&= \exp\left(-\frac{1}{2\sigma^2}|y-\Phi(X)^\top w|^2\right)\exp\left(-\frac{1}{2}w^\top\Sigma^{-1}w\right) \\
&= \exp\left(-\frac{1}{2\sigma^2}(y^\top y - 2w^\top\Phi(X)y) - \frac{1}{2}w^\top\left(\frac{1}{\sigma^2}\Phi(X)\Phi(X)^\top + \Sigma^{-1}\right)w\right) \\
&\propto \exp\left(-\frac{1}{2}(w-\overline{w})\left(\frac{1}{\sigma^2}\Phi(X)\Phi(X)^\top + \Sigma^{-1}\right)(w-\overline{w})\right),
\end{aligned} \tag{4.13}$$

where $\overline{w} = \frac{1}{\sigma^2}\left(\frac{1}{\sigma^2}\Phi(X)\Phi(X)^\top + \Sigma^{-1}\right)\Phi(X)y$ is obtained by completing the square in equation (4.13). Following the notation of Rasmussen & Williams [38], define

$$A := \frac{1}{\sigma^2}\Phi(X)\Phi(X)^\top + \Sigma^{-1}. \tag{4.14}$$

Then the density in equation (4.13) gives

$$w|X,y \sim \mathcal{N}(\overline{w}, A^{-1}). \tag{4.15}$$

Since the posterior is Gaussian, the posterior mean is equal to its mode, hence the maximum a priori estimate for w is the posterior mean. Also, the maximum likelihood estimate of the weights is obtained by maximizing $\mathrm{P}(w|X,y)$.

In most cases however, instead of explicitly computing an estimate of the weights w, the primary interest is to use $P(w|X,y)$ compute the predictive distribution, which is the average of the outputs with respect to all possible weights:

$$\mathrm{P}(y_*|x_*,X,y) = \int \mathrm{P}(y_*|x_*,w)\,\mathrm{P}(w|X,y)\,\mathrm{d}w. \tag{4.16}$$

Once again, the density is obtained by completing the square which is recognized as the density of the Gaussian

$$y_*|x_*,X,y \;\sim\; \mathcal{N}\left(\frac{1}{\sigma^2}\phi_*^\top A^{-1}\Phi y,\; \phi_*^\top A^{-1}\phi_*\right), \tag{4.17}$$

where $\Phi = \Phi(X)$, and $\phi_* = \phi(x_*)$. The mean in expression (4.17) contains the term $\sigma^{-2}A^{-1}\Phi$, which can be written as

$$\begin{aligned}
\sigma^{-2}A^{-1}\Phi &= \sigma^{-2}A^{-1}\Phi(\Phi^\top\Sigma\Phi + \sigma^2 I)(\Phi^\top\Sigma\Phi + \sigma^2 I)^{-1} \\
&= A^{-1}A\Sigma\Phi(\Phi^\top\Sigma\Phi + \sigma^2 I)^{-1} \\
&= \Sigma\Phi(\Phi^\top\Sigma\Phi + \sigma^2 I)^{-1}.
\end{aligned} \tag{4.18}$$

Therefore, the mean in expression (4.17) can be written as $\phi_*^\top\Sigma\Phi(\Phi^\top\Sigma\Phi + \sigma^2 I)^{-1}y$. Furthermore, A^{-1} can be computed using the Sherman-Woodbury-Morrison formula

in (2.32) to get

$$A^{-1} = \Sigma - \Sigma\Phi(\sigma^2 I + \Phi^\top \Sigma\Phi)^{-1}\Phi^\top\Sigma. \tag{4.19}$$

Plugging this into the variance term, expression (4.17) becomes

$$\begin{aligned} y_*|x_*, X, y \ \sim \ & \mathcal{N}(\phi_*^\top\Sigma\Phi(\Phi^\top\Sigma\Phi + \sigma^2 I)^{-1}y, \\ & \phi_*^\top\Sigma\phi_* - \phi_*^\top\Sigma\Phi(\sigma^2 I + \Phi^\top\Sigma\Phi)^{-1}\Phi^\top\Sigma\phi_*). \end{aligned} \tag{4.20}$$

This form shows that the features only enter the expression (4.20) through $\Phi^\top\Sigma\Phi$, $\phi_*^\top\Sigma\Phi$, or $\phi_*^\top\Sigma\phi_*$, which alludes to the significance of the matrix function $\phi(x)^\top\Sigma\phi(x')$.

4.1.2 Function space formulation

Alternatively, the predictive distribution of y_* can be constructed by directly viewing y_* as a realization of a random vector f_* from a Gaussian process. If one first specifies a kernel of the Gaussian process, a predictive distribution of f_* is obtained by directly computing the conditional distribution of f_* on the training points and test input without any reference to weights or parameters. In Rasmussen & Williams [38], this approach is referred to as the "function-space view", since it aims to directly construct the distribution of the random function, rather than estimate the parameters of an underlying model.

In this approach, a kernel $k(x, x') = \mathrm{cov}(f(x), f(x'))$ is specified, and given a dataset (y, X), the the covariance matrix of the observed data is denoted as

$$\mathrm{cov}(y, y) = \sigma^2 I + K(X, X), \tag{4.21}$$

where σ is the standard deviation of white Gaussian noise. It is worth noting that the covariance of a pair of output values is determined by the input values.

Denoting the test input as X_* and the target random vector as f_*, consider the joint distribution of y and f_*, namely

$$\begin{bmatrix} y \\ f_* \end{bmatrix} \ \sim \ \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} \sigma^2 I + K & K_* \\ K_*^\top & K_{**} \end{bmatrix}\right), \tag{4.22}$$

where $K := K(X, X), K_* := K(X, X_*)$, and $K_{**} := K(X_*, X_*)$. The prior mean functions need not be zero, but is set to zero for convenience. Incorporating nonzero mean functions is straightforward, as seen in Rasmussen & Williams [38, §2.7].

The distribution of $y_*|x_*, X, y$, i.e., the predictive (conditional) distribution, which is computed using the block-wise inversion lemma in section 2.1 is

$$\begin{aligned} f_*|x_*, X, y \ \sim \ & \mathcal{N}(K_*^\top(K + \sigma^2 I)^{-1}y, \\ & K_{**} - K_*^\top(K + \sigma^2 I)^{-1}K_*). \end{aligned} \tag{4.23}$$

Comparing the expressions (4.20) and (4.23), it is clear that the function space approach and the linear model approach are equivalent if

$$k(x, x') = \phi(x)^\top\Sigma\phi(x'). \tag{4.24}$$

This motivates the relationship between the kernel $k(x, x')$ and the matrix function $\phi(x)\Sigma\phi(x')$, which is sometimes called the *equivalent kernel*. The relationship between basis functions and kernels is further explored in section 6.3.

4.1.3 Hyperparameter maximum likelihood estimation

Another important task is finding the optimal hyperparameters of the kernel. While it is sometimes adequate to choose the hyperparameters based on physical knowledge of the underlying model, most of the time the hyperparameters must be learned using the given data. This task is commonly performed using a maximum likelihood framework.

Given a kernel $k(x_1, x_2; \theta)$, where θ is the vector of hyperparametrs, and a vector of observations y that is from a random vector $f \sim \mathcal{N}(0, C = K + \sigma^2 I)$, the marginal log-likelihood is shown in [38] to be

$$\ell(\theta) := \log \mathrm{P}(y|\theta) = -\frac{1}{2} y^\top C^{-1} y - \frac{1}{2} \log |C| - \frac{n}{2} \log 2\pi. \tag{4.25}$$

In order use a first-order optimization routine, the gradient of the log-likelihood is also required. Each component of the gradient $g(\theta) \in \mathbb{R}^p$ is computed analytically by

$$g_i(\theta) := \frac{\partial \ell(\theta)}{\partial \theta_i} = \frac{1}{2} y^\top C^{-1} C_i C^{-1} y - \frac{1}{2} \operatorname{tr}(C^{-1} C_i), \tag{4.26}$$

using the identities

$$\frac{\partial}{\partial \theta_i} C^{-1} = -C^{-1} \frac{\partial C}{\partial \theta_i} C^{-1}, \tag{4.27}$$

and

$$\frac{\partial}{\partial \theta_i} \log |C| = \operatorname{tr}\left(K^{-1} \frac{\partial K}{\partial \theta} \right), \quad \text{if } C \text{ is positive definite.} \tag{4.28}$$

The equations $g_i(\theta) = 0$ are often referred to as score equations.

4.1.4 Kernels, Gram matrices, and the curse of dimensionality

The construction of the posterior density in section 4.1.2 alludes to the fact that a Gaussian process is completely described by a kernel, and a mean function.

Kernels that solely depend on $\tau := x - x'$ are called stationary kernels. Qualitatively, a stationary kernel represents the assumed similarity of outputs for input data based on how "close" they are, where the notion of closeness is determined by $\|\tau\|$. If x and x' enter only through $r := \|\tau\|_2 = \|x - x'\|_2$, where $\|\cdot\|_2$ is the Euclidean norm, then the kernel is called isotropic. For stationary kernels that are sufficiently smooth, the resulting Gram matrix is often highly low-rank.

Another advantage of using a stationary kernel is that it leads to a separable Gram matrix.

There are also kernels that are not stationary. For example, the feature space construction of the posterior distribution in equation (4.20) leads to the kernel $k(x, x') = \phi(x)^\top \Sigma \phi(x')$. Defining $\Phi \in \mathbb{R}^{n \times m}$ to be the matrix of m features evaluated

at n points, then the resulting Gram matrix becomes

$$K = (\Sigma^{\frac{1}{2}}\Phi)^\top(\Sigma^{\frac{1}{2}}\Phi).$$

Since $\Sigma^{\frac{1}{2}}\Phi \in \mathbb{R}^{n\times m}$, the rank of this Gram matrix is at most m.

Also, the family of piecewise polynomial kernels with compact support often gives rise to a sparse Gram matrix, for which efficient sparse linear algebra routines can be used.

A kernel $k : \mathcal{X} \times \mathcal{X} \to \mathbb{C}$ is a valid covariance function if it is a function which is positive semidefinite, i.e.,

$$\int k(\mathbf{x}, \mathbf{x}')f(\mathbf{x})f(\mathbf{x}')\,\mathrm{d}\mu(\mathbf{x})\,\mathrm{d}\mu(\mathbf{x}') \geq 0$$

for all $f \in L_2(\mathcal{X}, \mu)$. Hence, a Gram matrix corresponding to a covariance matrix is also positive semidefinite.

A detailed list of covariance functions commonly used for statistical inference and their properties are included in [38, §4].

Hierarchical compression schemes are effective for the Gram matrices that arise from the following families of covariance functions. Note that for other covariance functions such as dot product kernels, or compactly-supported kernels, there exists other efficient methods to facilitate computation, especially when the matrix is fully separable, or sparse.

Below is a very brief list of commonly used isotropic kernels. Here, $r = \|x - x'\|_2$ invariably.

- The squared exponential kernel (also known as the radial basis function kernel)
  - $\to$ $k(r) = \exp\left(-\dfrac{r^2}{2\ell^2}\right)$
  - $\to$ Hyperparameters: ℓ (characteristic length-scale)
- The Matérn kernel
  - $\to$ $k(r) = \dfrac{1}{2^{\nu-1}\Gamma(\nu)}\left(\dfrac{\sqrt{2\nu}}{\ell}r\right)^\nu K_\nu\left(\dfrac{\sqrt{2\nu}}{\ell}r\right)$, where $K_\nu(z)$ is the modified Bessel function of the second kind.
  - $\to$ Hyperparameters: $\nu, \ell > 0$
- The exponential kernel
  - $\to$ $k(r) = \exp\left(-\dfrac{r}{\ell}\right)$
  - $\to$ Hyperparameters: ℓ
- The rational quadratic kernel
  - $\to$ $k(r) = \left(1 + \dfrac{r^2}{2\alpha\ell^2}\right)^{-\alpha}$

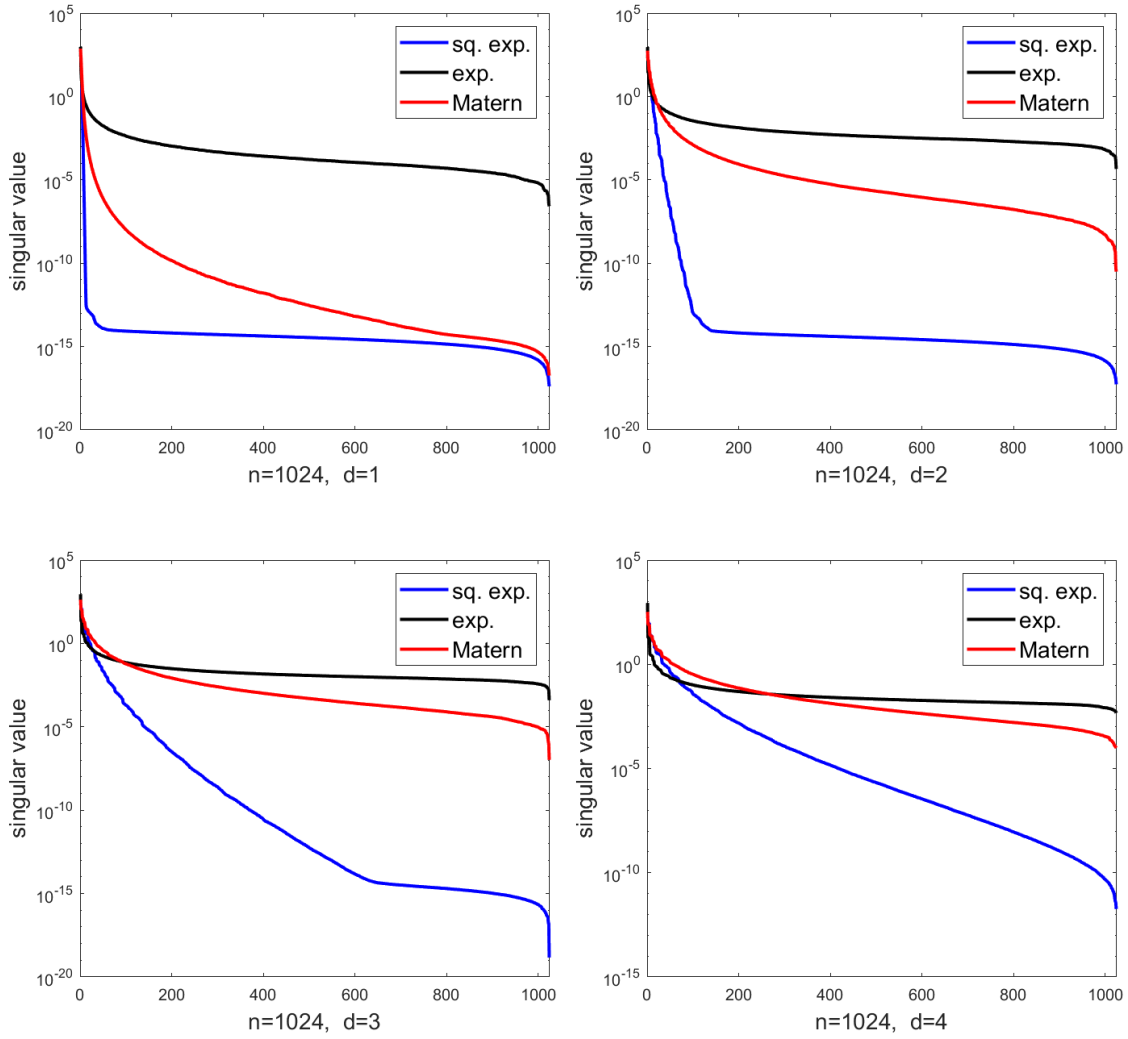

Figure 4.1: The spectral decay of 1024×1024 Gram matrices generated by the squared exponential kernel ($\ell = 1$), exponential kernel ($\ell = 1$), and the Matérn kernel ($\ell = 1, \nu = 5/2$) as the dimension of the data increases from 1 to 4.

→ Hyperparameters: ℓ, α

- The inverse multiquadratic kernel

  → $k(r) = \dfrac{1}{\sqrt{1+r^2}}$ (This is a special case of the rational quadratic kernel with parameters $\ell = 1$, $\alpha = 1/2$).

- The biharmonic kernel

  → $k(r) = r^2 \log |r|$

Figure 4.1 displays the spectral decay of Gram matrices arising from a few different kernels as the dimension increases. It is worth noting that the squared expo-

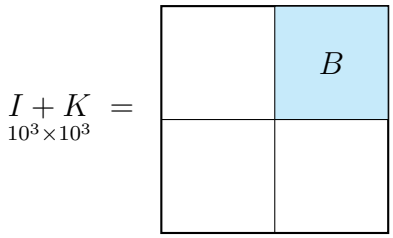

Figure 4.2: A 1000×1000 Gram matrix K generated by evaluating $k(x, x')$ at 1000 uniformly drawn points from the interval $[-0.5, 0.5]$. Table 4.1 compares the rank growth of the 500×500 off-diagonal block B for increasing d.

d	1	2	3	4	5
squared exponential ($\ell = 1$)	7	41	178	466	500
exponential ($\ell = 1$)	1	162	422	500	500
Matérn ($\nu = 5/2$, $\ell = 1$)	3	177	390	500	500

Table 4.1: The rank growth of the 500×500 off-diagonal block B in figure 4.2 for increasing d. The rank was computed using Matlab's `rank` function.

nential kernel shows very rapid spectral decay for $d = 1$, and is moderately rapid even for dimension $d = 2$. Although not included in figure 4.1, the Gram matrix associated with the rational quadratic kernel has similar spectral decay pattern as the squared exponential kernel. The Matérn kernel seems to be heavily affected by growth of dimension. The spectrum of the exponential kernel does not decay much, but is interestingly not affected much by growth of dimension. Figure 4.2 and table 4.1 below demonstrate the curse of dimensionality in terms of the rank growth of the blocks of the Gram matrix.

Since hierarchical solvers rely on the low-rankness of off-diagonal blocks, the increase in dimension makes it increasingly difficult to apply a hierarchical solver. For high dimensional problems, which is mostly of interest in data science applications, a more advanced hierarchical compression scheme is required. Alternatively, one might consider using a iterative solver, or an analytical technique based on computing a series approximation of the kernel. More details on different approaches are included in section 4.2. A deeper analysis of the eigenvalues and eigenfunctions of kernels, and approximation techniques are covered in section 6.3.

4.2 Survey of fast algorithms for Gaussian processes

The predictive distribution of Gaussian process models in equation (4.23) is easy to express analytically, but involves inverting an $n \times n$ dense matrix, which is a major computational bottleneck costing $\mathcal{O}(n^3)$ floating point operations. This calls for an efficient way to compute (4.17) for large data. Another computation-heavy task is computing the Gaussian process log-likelihood (4.25) and its gradient (4.26) to compute the maximum likelihood estimates of the kernel hyperparameters. The computational challenges that occur for these tasks can be summarized as the following:

- Inverting the covariance matrix $C = \sigma^2 I + K$.
- When the exact Cholesky factor of C cannot be computed due to computational constraints, finding an approximate Cholesky factor R of C, i.e., $C \approx R^\top R$, can also greatly help compute terms of the form $y^\top C^{-1} y$ and $y^\top C^{-1} C_i C^{-1} y$, where the former appears in the posterior density function of f_*, and also the marginal log-likelihood function (4.25), and the latter appears in the score function (4.26).
- Computing the log determinant of C, which appears in the posterior density of f_*, and also the marginal log-likelihood function (4.25).
- Computing $\operatorname{tr}(C^{-1} C_i)$, from the gradient of the log-likelihood (4.26).

Several themes to efficiently handle these computations for large datasets are introduced in the following subsections.

4.2.1 Hierarchical matrix factorizations

For many commonly used kernels, the associated covariance matrix $C = \sigma^2 I + K$ has a hierarchical rank structure, whose definition is introduced in chapter 3. Qualitatively, the kernel represents a physical relationship between the input data.

Techniques to rapidly invert, or compute products of large, dense hierarchical matrices were actively developed for the purpose of solving PDEs arising in computational physics and engineering applications. In the past decade, these techniques involving hierarchical matrices started being used for statistical inference. In particular, kernel methods benefit from such techniques, since matrices that arise from kernels often have hierarchical rank structure.

Ambikasaran et al. [3] use a hierarchical off diagonal low rank (HODLR) factorization to compute the predictive distribution (4.23), and also introduces a method to compute the log determinant of the covariance matrix using the resulting hierarchical factorization. Minden et al. [36] tackle the problem of finding the best fit kernel hyperparameters by computing the log-likelihood and its gradient using a recursive skeletonization factorization, and matrix peeling for weakly-admissable matrices. Geoga et al. [13] use a HODLR factorization approach for the same task. Williams & Seeger [48] use the Nyström method to compress off diagonal blocks for similar tasks in various kernel methods.

A major advantage of this approach is that once a hierarchical matrix factorization is obtained, it can also be used to compute the determinant of the covariance matrix as well [3]. That is, prediction, density estimation, and model training can be performed together once a factorization is computed. In addition, a hierarchical matrix factorization can be used repeatedly in the case where testing has to be performed sequentially.

On the other hand, this approach becomes slower when the the Gram matrix is not sufficiently low-rank, which is often the case when dealing with Gaussian processes models of high dimension [36].

4.2.2 Analytic techniques

By expressing a Gaussian process using basis functions, it is also possible to obtain a low-rank approximation of the Gram matrix. Once a low-rank approximation of the form $K \approx UDU^\top$ is obtained (which is always possible since K is symmetric), it is straightforward to compute the inverse of $C = K + I_n$ via

$$\begin{aligned}[UDU^* + I_n]^{-1} &= [UDU^* + UI_rU^*]^{-1} \\ &= U[D + I_r]^{-1}U^*.\end{aligned} \tag{4.29}$$

Since $D + I_r$ is a $r \times r$ diagonal matrix, the computation of its inverse only requires $\mathcal{O}(r)$ floating point operations.

In order to obtain a low-rank approximation of the Gram matrix, Greengard & O'Neil [17] use the Karhunen-Loéve expansion of the Gaussian process. Greengard [16] further introduces methods to use Fourier series to approximate a Gaussian process. The Nyström method is used in Williams & Seeger [48]. Ambikasaran [1] introduces methods using Taylor series, multipole expansions, and interpolation techniques.

Analytical methods are typically very fast when the required length of the sum is short. However, such methods suffer from the curse of dimensionality, and also introduce a truncation error along with approximation error. The accuracy and error properties of analytical methods are explored in section 6.3.

4.2.3 Iterative methods

The computation-heavy term in the predictive distribution (4.23), namely $C^{-1}y$, can be computed by solving a system of equations involving the matrix C. Since C is a symmetric positive semidefinite matrix, a Cholesky solver is suitable for small models. However, the Cholesky factorization requires $\mathcal{O}(n^3)$ floating point operations, which is infeasible for larger problems, setting aside memory issues. Instead, a preconditioned conjugate gradient solver is adequate for large symmetric positive semidefinite systems of equations, which is demonstrated in Wang et al. [46].

The biggest advantage of using an iterative solver such as the conjugate gradient method is that it is highly parallelizable, and does not depend on the rank structure of C, although the result will be affected by the conditioning of C.

However, this approach does not facilitate the computation of the determinant of C. Hence $\log|C|$ in the log-likelihood function (4.25) typically needs to be computed by an approximate method.

4.2.4 Approximate methods

Aside from the approaches listed above, many approximation methods for Gaussian process models have also been introduced. While the previously mentioned methods also require some form of approximation, they still utilize the whole dataset, and the error can be controlled to high precision. By contrast, the methods in this category make an approximation by selecting a subset of the data. For instance, stochastic

methods are used to evaluate the log-likelihood in [43]. A matrix-free approach to computing the score equations (4.26) based on the Hutchinson trace estimator [24] is introduced in [4]. There is also a large literature on *covariance tapering*, e.g., [25], which is a method that aims to introduce sparsity in the covariance matrix.

In many instances, techniques originating from the deep learning literature have also been applied to large-scale Gaussian processes. For example, an interpolative decomposition is used to sample a covariance matrix in [12], a stochastic variational optimzation method is used to train a GP model in [8], and a mixture of experts technique is used in [10].

Also, in the case of Gaussian process classification, the resulting likelihood is non-Gaussian. Hence, it is unavoidable to approximate the integrals arising in the posterior computation. A Laplace approximation, e.g., in [39], aims to approximate the posterior density with that of a Gaussian. Other popular approaches to approximate the posterior density include expectation propagation (EP) [26], and Markov chain Monte Carlo (MCMC) [21]. A comparison of approximation methods for GP classification is in [37].

5 Fitting kernel hyperparameters

Many kernels have several associated parameters aside from the input, which can be interpreted as conveying physical information of the underlying problem. These parameters, which are often called hyperparameters, sometimes have to be estimated from the data using, e.g., a maximum likelihood framework with a first order optimization routine as in [36]. In Matlab, `fminunc` (unconstrained optimization) or `fmincon` (constrained optimization) are routines that can be used for this purpose.

5.1 Evaluating the marginal log-likelihood and its gradient

As was seen in section 4.1.3, the Gaussian marginal log-likelihood for the hyperparameters can be written as

$$\ell(\theta) := \log \mathrm{P}(y|\theta) = -\frac{1}{2} y^\top C^{-1} y - \frac{1}{2} \log |C| - \frac{n}{2} \log 2\pi. \tag{5.30}$$

The first term $-\frac{1}{2} y^\top C^{-1} y$ can be computed using a hierarchical solver such as the ones discussed in section 3. C^{-1} can be directly applied to y, or if a matrix R that satisfies $C \approx R^\top R$ can be computed efficiently, then computing $(R^{-1}y)^\top (R^{-1}y)$ can save time. It is worth noting that the HODLR factorization does not support computing the square root matrix, whereas the RSF automatically provides the square root matrix R.

Computing $\log |C|$ is supported by both the HODLR factorization, and RSF as mentioned in section 3.

The gradient of the marginal log-likelihood is stated again below.

$$g_i(\theta) := \frac{\partial \ell(\theta)}{\partial \theta_i} = \frac{1}{2} y^\top C^{-1} C_i C^{-1} y - \frac{1}{2} \operatorname{tr}(C^{-1} C_i). \tag{5.31}$$

The matrix derivative $C_i = \dfrac{\partial C}{\partial \theta_i}$ can be computed analytically for most kernels, and can be implemented easily. For example, for the squared exponential kernel

$$k(r; \ell) = \exp\left(-\frac{r^2}{2\ell^2}\right), \tag{5.32}$$

the derivative with respect to ℓ is

$$\frac{\partial k(r; \ell)}{\partial \ell} = \frac{r^2}{\ell^3} \exp\left(-\frac{r^2}{2\ell^2}\right). \tag{5.33}$$

Furthermore, if the square-root matrix R of C_i can be computed efficiently, then the first term in (5.31) can be computed via $(RC^{-1}y)^\top RC^{-1}y$. For the second term $-\dfrac{1}{2}\operatorname{tr}(C^{-1}C_i)$, first, $C^{-1}C_i$ can be computed using one of the hierarchical solvers, then the trace is estimated using matrix peeling.

5.1.1 Trace estimation

The problem of estimating $\operatorname{tr}(T(A))$, where $T(A) \in \mathbb{R}^{m \times m}$ is typically a black-box transformation of the matrix A, is a central problem not only in Gaussian process MLE, but also in topics such as spectral density computation, and log-determinant computation etc.

A popular approach for trace estimation is using Hutchinson's estimator [24]

$$\operatorname{tr}_{H_n}(A) := \frac{1}{n} \sum_{i=1}^{n} z_i^\top A z_i, \tag{5.34}$$

where $z_i \in \mathbb{R}^m$ is a random vector containing iid zero mean, unit variance sub-Gaussian entries. This estimator is based on the observation that if z has iid zero mean and unit variance entries,

$$\mathbb{E}(z^\top A z) = \mathbb{E}(\operatorname{tr}(z^\top A z)) = \mathbb{E}(\operatorname{tr}(A z z^\top)) = \operatorname{tr}(A \mathbb{E}(z z^\top)) = \operatorname{tr}(A). \tag{5.35}$$

For example, the original Hutchinson's estimator was based on $z_i \sim \{-1, 1\}^m$, and $z_i \sim \mathcal{N}(0, I_m)$ also performs similarly. Results in [40], [5] show that for z_i with sub-Gaussian entries, and A a positive semidefinite matrix, the required n to achieve

$$(1 - \varepsilon)\operatorname{tr}(A) \leq \operatorname{tr}_{H_n}(A) \leq (1 + \varepsilon)\operatorname{tr}(A) \quad \text{with probability } \geq 1 - \delta \tag{5.36}$$

is $n = \mathcal{O}(\log(1/\delta)/\varepsilon^2)$. A recent research [35] introduces an improved estimator that has the same error bound with $n = \mathcal{O}(\sqrt{\log(1/\delta)}/\varepsilon + \log(1/\delta))$.

A detailed comparison of the Hutchinson's estimator and the estimator based on matrix peeling is in [36].

5.2 Numerical results

The following experiments contain results produced by the matrix peeling algorithm with a naive $\mathcal{O}(n^2)$ dense matrix-vector multiply scheme. Note that in the presence of an $\mathcal{O}(n)$ fast multiplication scheme, or advanced computing resources, the complexity of matrix peeling can be improved to $\mathcal{O}(n \log n)$ as discussed in section 3.5.2.

The error is measured in the difference between the extracted trace and the true trace when it is available. Note that without direct access to individual matrix entries, the exact trace can only be accessed as long as the matrix can be stored.

d	n	Peel	Extract	Levels	$k+\ell$	error
1	16^2	5.9922e-02	2.4161e-02	2	12	2.8422e-14
	32^2	7.6296e-01	1.0228e-01	4	13	1.1369e-13
	64^2	1.3508e+01	8.1723e-01	6	15	3.6380e-12
	128^2	2.9173e+02	1.3940e+01	8	15	1.1823e-10
	256^2	5.4608e+03	2.3633e+02	10	15	
2	16^2	1.3363e-01	3.9090e-02	2	40	9.8681e-07
	32^2	2.8777e+00	2.7884e-01	4	40	1.7097e-04
	64^2	6.0081e+01	3.1813e+00	6	40	5.2320e-03
	128^2	1.1674e+03	4.8886e+01	8	40	4.4444e-03

Table 5.2: Run-time in seconds and error for computing the trace using matrix peeling

d	n	Iterations	Optimal ℓ	$\|\nabla f\|$
1	16^2	23	526.5503	5.3819e-06
	32^2	23	698.4499	9.9684e-06

Table 5.3: Number of iterations and value of estimated hyperparameter of the RBF kernel (characteristic length-scale ℓ) learned from the data using Matlab's `fminunc`. The data was generated uniformly on the interval $[-3, 3]$. The initial point x_0 was set to 1 for all cases.

6 Prediction

Once the kernel hyperparameters are learned from the data, the kernel can be used to compute the posterior distribution of the test outputs f^* (4.23), which is repeated once again:

$$\begin{aligned} f_*|x_*, X, y \;\sim\; \mathcal{N}(&K_*^\top (K+\sigma^2 I)^{-1} y, \\ &K_{**} - K_*^\top (K+\sigma^2 I)^{-1} K_*). \end{aligned} \tag{6.37}$$

6.1 Direct randomized approximate spectral decomposition approach

Using the randomized approximate svd in section 1.1, the Gram matrix can be directly compressed in the form

$$K \approx UDU^\top. \tag{6.38}$$

Using this, the task of computing $(K+I)^{-1}$ can be written as

$$\begin{aligned}(UDU^\top + I_n)^{-1} &= (UDU^\top + UI_rU^*)^{-1} \\ &= U(D+I_r)^{-1}U^\top.\end{aligned} \tag{6.39}$$

Since $D+I_r$ is a $r \times r$ diagonal matrix, the computation of its inverse is particularly simple.

The main computational challenge is the range-finding stage of the randomized svd mentioned in 1.1.1. The matrix-vector multiplications involving the matrix K is $\mathcal{O}(mn)$ if implemented naively, and quickly becomes slow. However, as mentioned in section 1.1.4, a fast matrix-multiply routine reduces the algorithm to be $\mathcal{O}(k^2(m+n))$ which is feasible for extremely large matrices.

6.1.1 Numerical results

The error is measured by the quality of the matrix factorization, namely $\|Cv - UDU^\top v\|/\|Cv\|$, where v is a uniformly generated random vector with unit norm. This is a proxy for $\|C - UDU^\top\|/\|C\|$. When n is large, it may also take a long time to compute Cv on a standard laptop. In this case, a random subset of the rows of C (1000 rows for all of the experiments henceforth) were chosen to compute the error. In the tables below, $k+\ell$ denotes the target rank k plus oversamling parameter p.

Table 6.4 shows the run time and errors for the simple randomized eigenvalue decomposition approach with the squared exponential (RBF) kernel.

Table 6.9 shows the run time and errors for the same method with the Matérn kernel. Information about the dimension, and hyperparameters are provided in each of the tables.

6.2 Hierarchical matrix factorization approach

6.2.1 HODLR factorization

The HODLR factorization of the covariance matrix $K(X,X)$ can be rapidly applied to a vector as discussed in section 3.3. Typically, the factorization is computed in the training phase, that is, while computing the log-likelihood function. Thus, this factorization can be used again to compute the posterior mean and variance. The timing and error associated with computing the HODLR factorization and the posterior distribution is in section 6.2.3. The error is measured by the quality of the factorization, i.e., $\|Cv - C_Hv\|/\|Cv\|$, where v is a uniformly generated random vector of norm 1, and C_H is the HODLR factorization of C.

d	ℓ	n	Stage A	Stage B	Invert	$k+p$	error
1	1	16^2	8.1414e-03	2.2306e-03	1.3758e-03	25	1.9952e-15
		32^2	5.5106e-02	7.2160e-04	3.6370e-04	25	1.9757e-14
		64^2	6.5045e-01	2.0360e-03	8.0130e-04	25	2.4012e-15
		128^2	9.5247e+00	4.6957e-03	1.8185e-03	25	6.5651e-15
		256^2	1.8059e+02	1.5815e-02	6.0505e-03	25	6.6119e-14
		512^2	2.6207e+03	4.8991e-02	2.8392e-02	25	8.5077e-14
2	1	16^2	2.6988e-02	2.6023e-03	2.3066e-03	100	2.2769e-05
		32^2	2.8729e-01	4.2543e-03	1.6171e-03	100	2.6345e-05
		64^2	3.8938e+00	7.6879e-03	3.5396e-03	100	2.6142e-05
		128^2	6.1767e+01	2.4406e-02	3.9177e-03	100	2.3383e-05
		256^2	7.1336e+02	1.1855e-01	1.5985e-02	100	3.8602e-05

Table 6.4: Run-time in seconds and error for computing posterior mean and variance using an eigenvalue decompositian via randomized SVD with squared exponential kernel.

d	ℓ	ν	n	Stage A	Stage B	Invert	$k+p$	error
1	1	5/2	16^2	5.7703e-02	2.6156e-03	1.3291e-03	45	$\mathcal{O}(10^{-6})$
			32^2	3.7997e-01	5.8160e-04	5.2900e-04	45	$\mathcal{O}(10^{-6})$
			64^2	5.6499e+00	1.0727e-03	7.6910e-04	45	$\mathcal{O}(10^{-6})$
			128^2	8.3203e+01	4.3308e-03	1.8921e-03	45	$\mathcal{O}(10^{-6})$
			256^2	1.2734e+03	2.2246e-02	1.0639e-02	45	$\mathcal{O}(10^{-6})$

Table 6.5: Run-time in seconds and error for computing posterior mean and variance using an eigenvalue decompositian via randomized SVD with Matérn kernel

6.2.2 Recursive skeletonization factorization

In the case of the RSF in section 3.4, the inverse factors can be formed directly during construction. Once the factorization is computed, the factors can be applied rapidly to a vector, hence can be used to compute the posterior distribution (6.37). The error is measured by the quality of the factorization, i.e., $\|Cv - C_R v\|/\|Cv\|$, where v is a uniform random vector of norm 1, and C_R is the RSF of C.

6.2.3 Numerical results

Tables 6.6 and 6.7 show the run time and errors of the HODLR factorization with the RBF kernel, and Matérn kernel respectively. Tables 6.8 and 6.9 show the run time and errors of the recursive skeletonization factorization with the RBF kernel, and Matérn kernel respectively.

d	ℓ	n	Compress	Invert	$k+p$	error
1	1	16^2	8.9363e-03	3.0572e-02	25	1.2808e-14
		32^2	6.4223e-02	1.1576e-01	25	2.9497e-15
		64^2	5.7460e-01	4.8035e-01	25	1.8343e-14
		128^2	7.4567e+00	2.1527e+00	25	2.3054e-14
		256^2	1.1956e+02	1.0847e+01	25	1.8229e-14
		512^2	5.8185e+02	2.9443e+01	25	
		1024^2	2.8447e+03	1.5031e+01	25	
2	1	16^2	9.0642e-03	1.1023e-01	35	6.8502e-05
		32^2	2.2445e-01	4.2361e-01	35	7.3941e-05
		64^2	3.2993e+00	1.7309e+00	35	1.0155e-04
		128^2	4.4982e+01	7.1992e+00	35	8.3263e-05
		256^2	6.1788e+02	3.0609e+01	35	9.7801e-05

Table 6.6: Run-time in seconds and error for computing posterior mean and variance using HODLR factorization with squared exponential kernel. Compared to the simple low rank approximation scheme in Table 6.4, the target rank required for a similar level of precision is much lower for HODLR when $d = 2$, which shows it is a more efficient compression scheme.

d	ℓ	ν	n	Compress	Invert	$k+p$	error
1	1	5/2	16^2	6.3903e-02	9.7910e-02	45	$\mathcal{O}(10^{-5})$
			32^2	7.0262e-01	4.0214e-01	45	$\mathcal{O}(10^{-5})$
			64^2	6.3467e+00	1.6789e+00	45	$\mathcal{O}(10^{-5})$
			128^2	9.2876e+01	5.9213e+00	45	$\mathcal{O}(10^{-5})$
			256^2	1.2750e+03	2.4691e+01	45	$\mathcal{O}(10^{-5})$
			512^2	5.3009e+03	1.1646e+02	45	$\mathcal{O}(10^{-5})$

Table 6.7: Run-time in seconds and error for computing posterior mean and variance using HODLR factorization with Matérn kernel

6.3 Analytical techniques for low-rank approximations

6.3.1 Computing the Karhunen-Loéve expansion of a GP

Another approach to finding a global low-rank approximation of K involves computing a low-order eigenfunction expansion of the integral operator $\mathcal{K} : L^2[-1,1] \to L^2[-1,1]$ defined by

$$\mathcal{K}f(x) = \int_{-1}^{1} k(x,x')f(x')dx'. \tag{6.40}$$

The interval $[-1,1]$ was chosen to make use of the canonical Gauss-Legendre quadrature nodes and weights, but the Gauss nodes can be scaled to incorporate any interval, for example $[-3,3]$, which is used for the experiments in section 7. Furthermore, one could generalize this operator to be on square integrable functions on a d-dimensional region D. If $u(x)$ is an eigenfunction of $\mathcal{K}$, and λ is the corresponding

d	ℓ	n	Compress	Apply	$\lvert\mathcal{I}_{\text{skel}}\rvert$	Levels	error
1	1	16^2	3.3735e-01	1.0284e-02	14	4	1.2316e-14
		32^2	5.6943e-01	2.0169e-02	14	4	7.4032e-14
		64^2	4.0954e+00	4.4472e-02	14	4	4.1333e-13
		128^2	1.8186e+02	4.5395e-01	15	4	4.8133e-12
2	1	16^2	9.9356e-02	8.6903e-03	45	2	4.5443e-05
		32^2	4.4229e-01	1.6060e-02	35	2	4.9252e-04
		64^2	8.7825e+00	4.8234e-02	55	4	3.9443e-04
		128^2	2.1939e+02	4.6907e-01	55	4	6.8105e-04

Table 6.8: Run-time in seconds and error for computing posterior mean and variance using recursive skeletonization factorization (RSF) with squared exponential kernel.

d	ℓ	ν	n	Compress	Apply	$\lvert\mathcal{I}_{\text{skel}}\rvert$	Levels	error
1	1	5/2	16^2	1.3534e-01	1.2311e-02	25	2	5.0860e-15
			32^2	7.3018e-01	1.5903e-02	25	2	3.0533e-14
			64^2	9.9857e+00	4.1975e-02	25	4	5.6823e-14
			128^2	2.2908e+02	4.0900e-01	25	4	1.8010e-13

Table 6.9: Run-time in seconds and error for computing posterior mean and variance using recursive skeletonization factorization (RSF) with Matérn kernel

eigenvalue, then

$$\lambda u(x) = \mathcal{K}u(x) = \int_{-1}^{1} k(x, x')u(x')dx'. \tag{6.41}$$

The key observation in [17] is that if f is a zero mean Gaussian process on D with covariance function k, then any order-m approximation

$$\hat{f}(x) = \sum_{i=1}^{m} \alpha_i g_i(x), \tag{6.42}$$

where $\alpha_i \sim \mathcal{N}(0, 1)$, has covariance function

$$\begin{aligned} \hat{k}(x, x') &= \mathbb{E}[\hat{f}(x)\hat{f}(x')] \\ &= \mathbb{E}\left[\left(\sum_{i=1}^{m} \alpha_i g_i(x)\right)\left(\sum_{i=1}^{m} \alpha_i g_i(x')\right)\right] \\ &= \sum_{i,j=1}^{m} g_i(x)g_j(x')\mathbb{E}[\alpha_i\alpha_j] \\ &= \sum_{i=1}^{m} g_i(x)g_i(x'). \end{aligned} \tag{6.43}$$

Moreover, it is shown in [44] that if the $g_i(x)$'s are chosen to be the basis functions of the Karhunen-Loéve (KL) expansion, namely $\varphi_i(x) := \sqrt{\lambda_i}u_i(x)$, where the $u_i(x)$'s are the normalized eigenfunctions of $\mathcal{K}$ corresponding to the eigenvalues λ_i's, which

Algorithm 6.2 Computing the KL-expansion

1: Form $n \times n$ discretization matrix A defined by $A_{i,j} = \sqrt{w_i w_j} k(x_i, x_j)$, where x_i, w_i's are the order-r Gauss nodes and weights.
2: Compute eigenvalue decomposition of A: $A = UDU^*$. The eigenvalues are on the diagonals of D
3: Scale the rows of U to form matrix of eigenfunctions evaluated at Gaussian nodes: $\hat{U}_{i,:} = U_{i,:}/\sqrt{w_i}$.
4: Use length-r Legendre polynomial approximation to find the eigenfunctions based on the columns of $\hat{U}$. Store the coefficients of the Legendre approximation in matrix $A = M\hat{U}$, where M is the matrix that transforms the evaluated function values into the coefficients.
5: The eigenfunctions are $u_i(x) = \sum_{j=1}^{r} A_{j,i} P_{j-1}(x)$ on $x \in [-1, 1]$, where $P_j(x)$ is the j-th order Legendre polynomial.
6: Set $\varphi_i(x) = \sqrt{\lambda_i} u_i(x)$.

are in decreasing order, then the covariance function $\hat{k}$ of $\hat{f}$ is the best approximation of the covariance function k of f in the L^2 norm. Mercer's theorem stated in e.g., [17] guarantees that

$$\sum_{i=1}^{r} \varphi_i(x)\varphi_i(x') \to k(x, x') \tag{6.44}$$

absolutely, and uniformly.

There are standard methods of finding the eigenvalues and eigenfunctions of $\mathcal{K}$, which involve a discretization matrix A that is obtained by discretizing $\mathcal{K}$ at the r-th order Gaussian nodes. Then, the eigenvalues of A are the first r eigenvalues of $\mathcal{K}$, and the eigenvectors of A are viewed as the eigenfunctions evaluated at each of the Gaussian nodes. A straightforward implementation is introduced in [17, Algorithm 1], which is summarized in algorithm 6.3.1.

This is an $\mathcal{O}(nr^2)$ algorithm on the interval, so it scales linearly with the system size. However, one must be aware of the additional error added in the discretization process, as well as the truncation error, which can be observed in section 6.3.3.

6.3.2 GP regression via KL-expansion

Once the order-m KL basis functions $(\varphi_i(x))_{i=1,\dots,m}$ are obtained, these can be used to form a low-rank approximation of the Gram matrix K. By (6.44), each entry in K can be approximated in the form

$$k(x, x') \approx \sum_{i=1}^{r} \varphi_i(x)\varphi_i(x'). \tag{6.45}$$

Defining $X := (\varphi_j(x_i))_{ij}$, then XX^* is a natural low-rank approximation of K, since each entry of XX^* is precisely the right hand side of (6.45). Hence, $[I + K]^{-1}$ is approximated by

$$[I + XX^*]^{-1}. \tag{6.46}$$

d	ℓ	n	KL-expand	Factor & invert	Length	error
1	1	16^2	9.7901e-01	8.3340e-02	25	3.2629e-11
		32^2	9.1814e-01	2.9757e-01	25	2.6403e-11
		64^2	8.7289e-01	1.1964e+00	25	2.3411e-11
		128^2	9.3278e-01	4.8108e+00	25	8.9366e-12
		256^2	9.1487e-01	2.0758e+01	25	6.1369e-12
		512^2	9.2822e-01	7.7226e+01	25	4.1503e-12
		1024^2	8.9800e-01	3.3264e+02	25	4.5741e-12

Table 6.10: Run-time in seconds and error for computing posterior mean and variance using KL-expansion based eigenvalue decomposition with squared exponential kernel

d	ℓ	ν	n	KL-expand	Factor & invert	Length	error
1	1	5/2	16^2	9.7533e-01	8.5563e-02	35	$\mathcal{O}(10^{-6})$
			32^2	8.7543e-01	3.1758e-01	35	$\mathcal{O}(10^{-6})$
			64^2	8.7446e-01	1.1730e+00	35	$\mathcal{O}(10^{-6})$
			128^2	1.0709e+00	4.9095e+00	35	$\mathcal{O}(10^{-6})$
			256^2	1.8818e+00	2.4444e+02	35	$\mathcal{O}(10^{-6})$
			512^2	1.6100e+00	1.0019e+03	35	$\mathcal{O}(10^{-6})$

Table 6.11: Run-time in seconds and error for computing posterior mean and variance using KL-expansion based eigenvalue decomposition with Matérn kernel

By computing the svd $X = UDV^*$, then (6.46) can be written as

$$[I + UD^2U^*]^{-1}. \tag{6.47}$$

One can apply the same technique used in (6.39) to compute the inverse quickly via

$$U[I_r + D^2]^{-1}U^*. \tag{6.48}$$

6.3.3 Numerical results

Here, the error is measured by $\|Kv - XX^\top v\|/\|K\|$, where v is again a uniformly generated random vector with unit norm, and the low-rank factor X is computed using the KL-expansion method described in section 6.3.2.

Tables 6.10 and 6.11 show the run time and error for the low-rank approximation obtained by the KL-expansion approach with the RBF kernel, and Matérn kernel respectively.

7 Results

The following plots demonstrate the effect of the hyperparameters on the variance of the Gaussian process with the squared exponential kernel (RBF kernel), and the Matérn kernel.

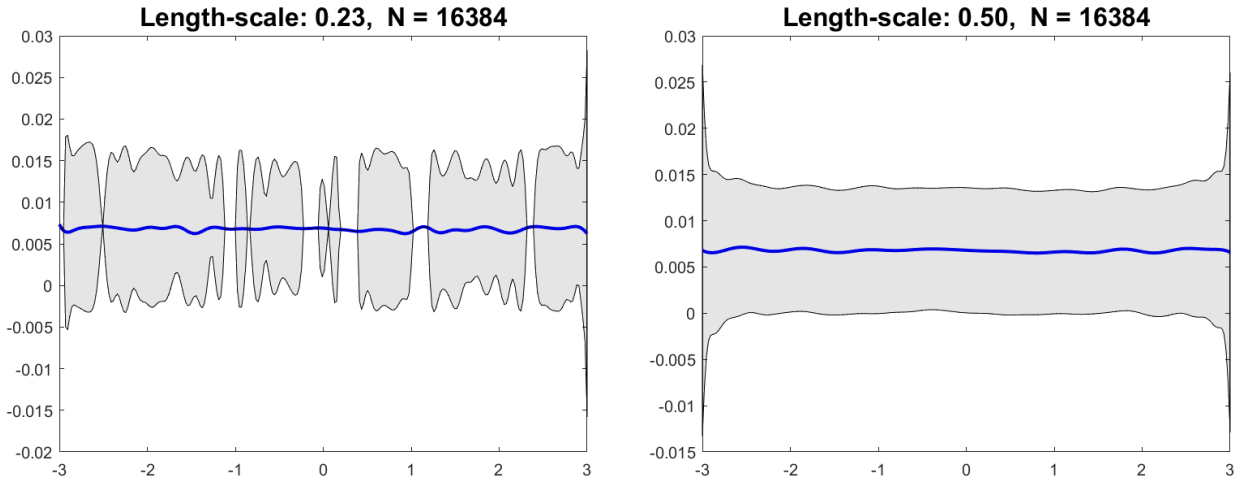

Figure 7.3: HODLR factorization is used to compute the posterior distribution of the Gaussian process with noise $\varepsilon \sim \mathcal{N}(0, 0.01)$, and the squared exponential (RBF) kernel. The posterior mean is the line in blue, and the gray area represents the pointwise 95 percent confidence interval of the test output. The increased characteristic length-scale parameter appears to smooth the variance.

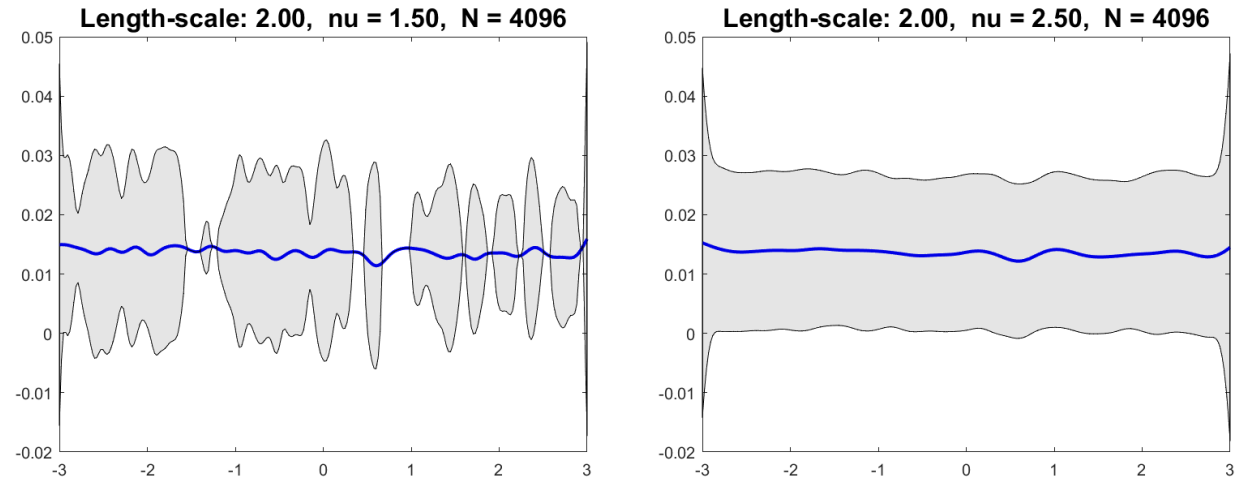

Figure 7.4: HODLR factorization is used to compute the posterior distribution of the Gaussian process with noise $\varepsilon \sim \mathcal{N}(0, 0.01)$, and the Matérn kernel. The posterior mean is the line in blue, and the gray area represents the pointwise 95 percent confidence interval of the test output. The increased parameter ν also appears to smooth the variance.

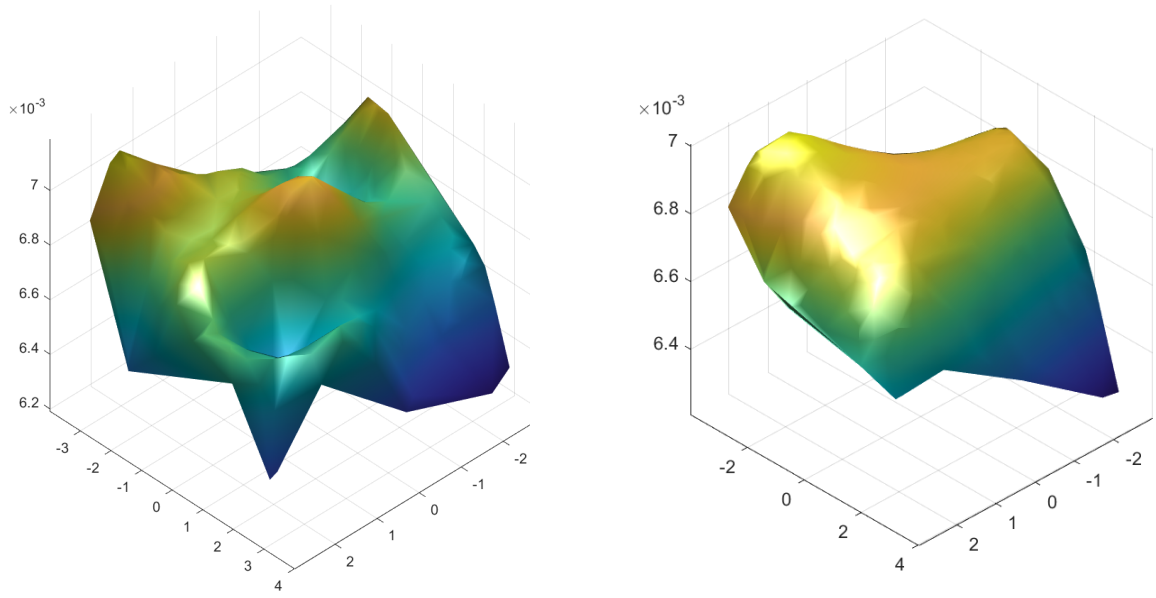

Figure 7.5: HODLR factorization is used to compute the posterior mean of the 2-d input Gaussian process with noise $\varepsilon \sim \mathcal{N}(0, 0.01)$, and the squared exponential (RBF) kernel. The 4096×2 data matrix was drawn uniformly from $[-3, 3] \times [-3, 3]$. In the plot on the left, the kernel has characteristic length-scale $\ell = 2$, and in the right plot, $\ell = 4$. The increased parameter ℓ appears to smooth out the mean function.

8 Conclusion

The HODLR framework and the recursive skeletonization factorization both efficiently compress kernel matrices from 1-d and 2-d data. The simple randomized spectral decomposition scheme for the whole matrix C is efficient in 1-d, but is less effective for 2-d data and higher, due to the unsophisticated compression method.

The KL-expansion based low-rank decomposition is the fastest ($\mathcal{O}(nr^2)$), even without advanced computing facilities. The error properties of the KL-expansion approach, and the extension to higher dimensions were not covered in this thesis due to a time constraint, but will be an interesting topic to explore in the future.

Other topics that are worth further investigation include randomized CUR algorithms and their error analysis, alternative optimization methods for hyperparameter MLE, and iterative methods for higher dimensions.

References

[1] S. Ambikasaran. "Fast algorithms for dense numerical linear algebra and applications". *Ph.D. dissertation, Stanford University* (2013) (cited on pp. 9, 33).

[2] S. Ambikasaran and E. Darve. "An $\mathcal{O}(N \log N)$ Fast Direct Solver for Partial Hierarchically Semi-Separable Matrices". *Journal of Scientific Computing* 57 (Dec. 2013). DOI: 10.1007/s10915-013-9714-z (cited on p. 23).

[3] S. Ambikasaran et al. "Fast Direct Methods for Gaussian Processes". *IEEE Transactions on Pattern Analysis and Machine Intelligence* 38.2 (2016), pp. 252–265. DOI: 10.1109/TPAMI.2015.2448083. URL: https://arxiv.org/abs/1403.6015 (cited on p. 32).

[4] M. Anitescu, J. Chen, and L. Wang. "A Matrix-Free Approach for Solving the Gaussian Process Maximum Likelihood Problem". 2011 (cited on p. 34).

[5] H. Avron and S. Toledo. "Randomized Algorithms for Estimating the Trace of an Implicit Symmetric Positive Semi-Definite Matrix". *J. ACM* 58.2 (Apr. 2011). ISSN: 0004-5411. DOI: 10.1145/1944345.1944349. URL: https://doi.org/10.1145/1944345.1944349 (cited on p. 35).

[6] J. Ballani and D. Kressner. "Matrices with Hierarchical Low-Rank Structures". Vol. 2173. Jan. 2016. ISBN: 978-3-319-49886-7. DOI: 10.1007/978-3-319-49887-4_3 (cited on p. 23).

[7] S. Chandrasekaran et al. "Some Fast Algorithms for Sequentially Semiseparable Representations". *SIAM Journal on Matrix Analysis and Applications* 27.2 (2005), pp. 341–364. DOI: 10.1137/S0895479802405884 (cited on p. 23).

[8] C.-A. Cheng and B. Boots. "Variational Inference for Gaussian Process Models with Linear Complexity". *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 2017, pp. 5190–5200 (cited on p. 34).

[9] H. Cheng et al. "On the Compression of Low Rank Matrices". *SIAM Journal on Scientific Computing* 26.4 (2005), pp. 1389–1404. DOI: 10.1137/030602678. URL: https://doi.org/10.1137/030602678 (cited on pp. 5, 13).

[10] M. Deisenroth and J. W. Ng. "Distributed Gaussian Processes". *Proceedings of the 32nd International Conference on Machine Learning, PMLR*. 37. 2015, pp. 1481–1490 (cited on p. 34).

[11] C. Eckart and G. Young. "The approximation of one matrix by another of lower rank". *Psychometrika* 1 (1936), pp. 211–218 (cited on p. 1).

[12] J. R. Gardner et al. "Product kernel interpolation for scalable gaussian processes". *AISTATS*. 2018, pp. 1407–1416 (cited on p. 34).

[13] C. J. Geoga, M. Anitescu, and M. L. Stein. "Scalable Gaussian Process Computations Using Hierarchical Matrices". *Journal of Computational and Graphical Statistics* 29 (2019), pp. 227–237 (cited on p. 32).

[14] A. Gillman, P. Young, and P.-G. Martinsson. "A direct solver with O(N) complexity for integral equations on one-dimensional domains". *Frontiers of Mathematics in China* 7 (May 2011). DOI: 10.1007/s11464-012-0188-3 (cited on p. 23).

[15] L. Greengard et al. "Fast direct solvers for integral equations in complex three-dimensional domains". *Acta Numerica* 18 (May 2009), pp. 243–275. DOI: 10.1017/S0962492906410011 (cited on p. 23).

[16] P. Greengard. "Efficient Fourier representations of families of Gaussian processes". *ArXiv* abs/2109.14081 (2021). URL: https://arxiv.org/abs/2109.14081 (cited on p. 33).

[17] P. Greengard and M. O'Neil. "Efficient reduced-rank methods for Gaussian processes with eigenfunction expansions". *ArXiv* (2021). URL: https://arxiv.org/abs/2108.05924v1 (cited on pp. 33, 40, 41).

[18] M. Gu and S. C. Eisenstat. "Efficient Algorithms for Computing a Strong Rank-Revealing QR Factorization". *SIAM Journal on Scientific Computing* 17.4 (1996), pp. 848–869. DOI: 10.1137/0917055. URL: https://doi.org/10.1137/0917055 (cited on p. 4).

[19] W. Hackbusch. "A Sparse Matrix Arithmetic Based on H-Matrices. Part I: Introduction to H-Matrices." *Computing* 62 (Apr. 1999), pp. 89–108. DOI: 10.1007/s006070050015 (cited on p. 23).

[20] N. Halko, P. G. Martinsson, and J. A. Tropp. "Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions". *SIAM Review* 53.2 (2011), pp. 217–288. URL: https://arxiv.org/abs/0909.4061 (cited on pp. 2–4, 19).

[21] J. Hensman et al. "MCMC for Variationally Sparse Gaussian Processes". *Advances in Neural Information Processing Systems*. Ed. by C. Cortes et al. Vol. 28. Curran Associates, Inc., 2015. URL: https://proceedings.neurips.cc/paper/2015/file/6b180037abbebea991d8b1232f8a8ca9-Paper.pdf (cited on p. 34).

[22] K. L. Ho and L. Ying. "Hierarchical interpolative factorization for elliptic operators: Integral equations". *Communications on Pure and Applied Mathematics* (2015) (cited on pp. 12, 18).

[23] K. L. Ho and L. Greengard. "A fast direct solver for structured linear systems by recursive skeletonization". *SIAM Journal on Scientific Computing* 34 (2012), pp. 2507–2532 (cited on p. 23).

[24] M. F. Hutchinson. "A stochastic estimator of the trace of the influence matrix for laplacian smoothing splines". *Communications in Statistics - Simulation and Computation* 19.2 (1990), pp. 433–450 (cited on pp. 34, 35).

[25] C. G. Kaufman, M. J. Schervish, and D. W. Nychka. "Covariance Tapering for Likelihood-Based Estimation in Large Spatial Data Sets". *Journal of the American Statistical Association* 103.2 (484 2008), pp. 1545–1555 (cited on p. 34).

[26] H.-C. Kim and Z. Ghahramani. "Bayesian Gaussian Process Classification with the EM-EP Algorithm". *IEEE Transactions on Pattern Analysis and Machine Intelligence* 28.12 (2006), pp. 1948–1959. DOI: 10.1109/TPAMI.2006.238 (cited on p. 34).

[27] L. Lin, J. Lu, and L. Ying. "Fast construction of hierarchical matrix representation from matrix–vector multiplication". *Journal of Computational Physics* 230 (2011), pp. 4071–4087 (cited on p. 23).

[28] M. W. Mahoney and P. Drineas. "CUR matrix decompositions for improved data analysis". *Proceedings of the National Academy of Sciences* 106.3 (2009), pp. 697–702. DOI: 10.1073/pnas.0803205106. URL: https://www.pnas.org/doi/abs/10.1073/pnas.0803205106 (cited on p. 6).

[29] P. G. Martinsson. "A Fast Randomized Algorithm for Computing a Hierarchically Semiseparable Representation of a Matrix". *SIAM Journal on Matrix Analysis and Applications* 32.4 (2011), pp. 1251–1274. DOI: 10.1137/100786617. URL: https://doi.org/10.1137/10078661 (cited on p. 17).

[30] P. G. Martinsson. "Compressing rank-structured matrices via randomized sampling". *SIAM Journal on Scientific Computing* 38 (2016), A1959–A1986 (cited on pp. 19, 22).

[31] P. G. Martinsson. "Randomized methods for matrix computations". *IAS/Park City Mathematics Series* (2018). URL: https://arxiv.org/abs/1607.01649 (cited on pp. 2, 3, 6).

[32] P. Martinsson and V. Rokhlin. "A fast direct solver for boundary integral equations in two dimensions". *Journal of Computational Physics* 205.1 (2005), pp. 1–23. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2004.10.033. URL: https://www.sciencedirect.com/science/article/pii/S0021999104004462 (cited on pp. 8, 13, 17).

[33] P.-G. Martinsson. *Fast Direct Solvers for Elliptic PDEs*. SIAM, 2019. ISBN: 978-1-61197-603-8. DOI: https://doi.org/10.1137/1.9781611976045 (cited on pp. 8, 9, 17, 22).

[34] A. W. Max. "Inverting modified matrices". *Memorandum Rept. 42, Statistical Research Group*. Princeton Univ., 1950, p. 4 (cited on p. 7).

[35] R. A. Meyer et al. "Hutch++: Optimal Stochastic Trace Estimation". *Proceedings of the SIAM Symposium on Simplicity in Algorithms (SOSA)* 2021 (2021), pp. 142–155. DOI: 10.1137/1.9781611976496.16 (cited on p. 35).

[36] V. Minden et al. "Fast Spatial Gaussian Process Maximum Likelihood Estimation via Skeletonization Factorizations". *SIAM Journal on Multiscale Modeling and Simulation* 15.4 (2017), pp. 1584–1611. DOI: https://doi.org/10.1137/17M1116477. URL: https://arxiv.org/abs/1603.08057 (cited on pp. 13, 32, 34, 36).

[37] H. Nickisch and C. E. Rasmussen. "Approximations for Binary Gaussian Process Classification". *Journal of Machine Learning Research* 9 (2008), pp. 2035–2078 (cited on p. 34).

[38] C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning*. the MIT Press, 2006 (cited on pp. 24, 26–29).

[39] J. Riihimaki and A. Vehtari. "Laplace Approximation for Logistic Gaussian Process Density Estimation and Regression". *Bayesian Analysis* 9.2 (2014), pp. 425–448 (cited on p. 34).

[40] F. Roosta-Khorasani and U. Ascher. "Improved Bounds on Sample Size for Implicit Matrix Trace Estimators". 15.5 (Oct. 2015), pp. 1187–1212. ISSN: 1615-3375. DOI: 10.1007/s10208-014-9220-1. URL: https://doi.org/10.1007/s10208-014-9220-1 (cited on p. 35).

[41] H. Samet. "The Quadtree and Related Hierarchical Data Structures". *ACM Computing Surveys* 16.2 (1984), pp. 187–260. URL: https://doi.org/10.1145/356924.356930 (cited on p. 10).

[42] J. Sherman and W. J. Morrison. "Adjustment of an Inverse Matrix Corresponding to a Change in One Element of a Given Matrix". *The Annals of Mathematical Statistics* 21.1 (1950), pp. 124–127. DOI: 10.1214/aoms/1177729893. URL: https://doi.org/10.1214/aoms/1177729893 (cited on p. 7).

[43] M. L. Stein, J. Chen, and M. Anitescu. "Stochastic Approximation of Score Functions for Gaussian Processes". *The Annals of Applied Statistics* 7.2 (2013), pp. 1162–1191 (cited on p. 34).

[44] L. N. Trefethen. *Approximation Theory and Approximation Practice*. SIAM, 2019. ISBN: 978-1-611975-93-2 (cited on p. 40).

[45] S. Voronin and P.-G. Martinsson. "Efficient algorithms for cur and interpolative matrix decompositions". *Advances in Computational Mathematics* 43 (June 2017), pp. 1–22. DOI: 10.1007/s10444-016-9494-8 (cited on pp. 5, 6).

[46] K. A. Wang et al. "Exact Gaussian Processes on a Million Data Points". *Advances in Neural Information Processing Systems*. 1312. 2019, pp. 14648–14659 (cited on p. 33).

[47] R. Wang, Y. Li, and E. Darve. "On The Numerical Rank Of Radial Basis Function Kernels In High Dimensions". *SIAM Journal on Matrix Analysis and Applications* 39.4 (2018), pp. 1810–1835 (cited on p. 23).

[48] C. Williams and M. Seeger. "Using the Nyström Method to Speed Up Kernel Machines". *Advances in Neural Information Processing Systems*. Ed. by T. Leen, T. Dietterich, and V. Tresp. Vol. 13. MIT Press, 2000 (cited on pp. 32, 33).

[49] F. Woolfe et al. "A fast randomized algorithm for the approximation of matrices". *Applied and Computational Harmonic Analysis* 25.3 (2008), pp. 335–366. ISSN: 1063-5203. DOI: https://doi.org/10.1016/j.acha.2007.12.002. URL: https://www.sciencedirect.com/science/article/pii/S1063520307001364 (cited on p. 3).

[50] J. Xia. "Randomized Sparse Direct Solvers". *SIAM Journal on Matrix Analysis and Applications* 34.1 (2013), pp. 197–227. DOI: 10.1137/12087116X (cited on p. 23).

[51] J. Xia et al. "Fast algorithms for hierarchically semiseparable matrices". *Numer. Linear Algebra Appl.* 17 (2010), pp. 953–976 (cited on p. 23).