Using Hierarchical Matrices to SKI Faster

Course Project - CS 6220, Spring 2020

Maxwell Jenquin

May 21, 2020

This course project probes the application of hierarchical rank-structured matrix factorizations to a kernel-approximation framework designed primarily for Gaussian process models.

1 Background and Context

This section provides an overview of the thought process behind this project and the ideas it builds on.

1.1 Gaussian Process Models

A Gaussian process (or GP) is defined as a stochastic process such that any finite subset of its random variables has a joint multivariate Gaussian distribution. This convenient mathematical form allows us to interpret them as function-valued random variables, namely

$$f \sim \mathcal{GP}(\mu(\cdot), k(\cdot,\cdot)) \Leftrightarrow f(x_1)\cdots f(x_n) \sim \mathcal{N}(\mu(\boldsymbol{x}), k(\boldsymbol{x},\boldsymbol{x})).$$

Here the function f is distributed as a Gaussian process with mean defined by the function μ, and covariance defined by the kernel function k. By convention μ is typically set to zero, but the choice of k determines the properties of samples drawn from the distribution (here $k(\boldsymbol{x},\boldsymbol{x})$ is a square matrix with i,jth entry $k(x_i,x_j)$). Using convenient properties of Gaussian distributions, this construction allows us to employ GPs as data models - given observations $(\boldsymbol{x},\boldsymbol{y})$, consider functions f distributed as a GP and constrained by $f(x_i) \sim y_i + \mathcal{N}(0,\sigma^2)$. Then a distribution over predictions at some set x^* can be computed as

$$\begin{aligned} y^*|x^*, \boldsymbol{x}, \boldsymbol{y}, \sigma^2 &\sim \mathcal{N}(f^*, \mathrm{cov}(f(x^*))) \\ \text{where } f^* &= \mu(x^*) + k(x^*,\boldsymbol{x})\big[k(\boldsymbol{x},\boldsymbol{x}) + \sigma^2 I\big]^{-1}\boldsymbol{y} \\ \text{and } \mathrm{cov}(f(x^*)) &= k(x^*,x^*) - k(x^*,\boldsymbol{x})\big[k(\boldsymbol{x},\boldsymbol{x}) + \sigma^2 I\big]^{-1} k(\boldsymbol{x},x^*) \end{aligned}$$

This perspective allows us to apply GP models to data using any valid covariance function k (meaning any function $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ which is symmetric and positive semidefinite, given a domain $\Omega \in \mathbb{R}^d$). As mentioned, the kernel defines the properties of samples from the distribution over functions, making it the primary design choice in the modeling process. Typically, kernels with adjustable *hyperparameters* θ are chosen and those hyperparameters are optimized with respect to the log-likelihood of the distribution over observed data,

$$\log p(\boldsymbol{y}|\theta) \propto -\big[\boldsymbol{y}^T(K_\theta + \sigma^2 I)^{-1}\boldsymbol{y} + \log|K_\theta + \sigma^2 I|\big]$$

where K_θ refers to $k(\boldsymbol{x},\boldsymbol{x};\theta)$. Intuitively, the first term describes how well-fit to data the model is, and the second term serves as a complexity penalty [15]. Evaluating this quantity repeatedly is the primary computational task involved in using a Gaussian process model, whether for hyperparameter optimization or for marginalization of hyperparameters via numerical integration, and costs $\mathcal{O}(n^3)$ work in the most general case. For a more thorough discussion of GP models and their applications, see [15].

1.2 Structured Kernel Interpolation

Indeed the $\mathcal{O}(n^3)$ complexity involved in using a GP model severely restricts their application to large problems or real-time contexts. Many approximative and structure-exploiting techniques have been developed over the past few decades to reduce work and storage required [10]. In this project, I focused on one such methodology which exploits multiplicative structure in k: structured kernel interpolation, or SKI [16].

The primary assumption of this approach is that for $x \in \mathbb{R}^d$ we use a kernel that is made of a product of 1-dimensional kernels, split along domain axes. That is,

$$k(x, x') = \prod_{\ell=1}^{d} k_\ell(x_\ell, x'_\ell).$$

With this assumption, any grid-valued input set $\boldsymbol{x} = \boldsymbol{x}_1 \times \cdots \times \boldsymbol{x}_d$ produces a kernel matrix K which can be decomposed into a Kronecker product:

$$K = K_1 \otimes \cdots \otimes K_d$$

Note that if the size of K_ℓ is $n_\ell \times n_\ell$, then the size of K is $n \times n$ with $n = \prod n_\ell$. With K in this form its eigendecomposition can be found efficiently, since it is the Kronecker product of the eigendecompositions of the component matrices,

$$K = K_1 \otimes \cdots \otimes K_d = Q_1 V_1 Q_1^T \otimes \cdots \otimes Q_d V_d Q_d^T = \bigotimes_{\ell=1}^{d} Q_\ell \bigotimes_{\ell=1}^{d} V_\ell \left(\bigotimes_{\ell=1}^{d} Q_\ell \right)^T = QVQ^T.$$

Computing d smaller eigendecompositions costs $\mathcal{O}(n_1^3 + \cdots + n_d^3)$ as opposed to the default cost of $\mathcal{O}(n_1^3 n_2^3 \cdots n_d^3)$, which represents a large improvement in complexity. Kronecker structure can also be exploited for fast matrix-vector products by repeatedly restructuring the vector and applying one of the component matrices, for a cost of $\mathcal{O}(n_1^2 + \cdots + n_d^2)$ as opposed to the default cost of $\mathcal{O}(n^2) = \mathcal{O}(n_1^2 n_2^2 \cdots n_d^2)$.

In addition, log-determinants can be calculated using eigenvalues λ_i of the matrix K:

$$\log |K + \sigma^2 I| = \sum_{i=1}^{n} \log(\lambda_i + \sigma^2).$$

Furthermore, fast matrix-vector products allow linear solves with diagonal correction to be done quickly using an iterative algorithm, namely linear conjugate gradients. An alternative method of computing linear solves exploits the eigendecomposition in a similar fashion to the log-determinant computation, namely

$$(QVQ^T + \sigma^2 I)^{-1} \boldsymbol{y} = Q(V + \sigma^2 I)^{-1} Q^T \boldsymbol{y}$$

although in practice SKI uses an iterative solver instead.

These savings are significant to the context of GP methods, and because a multiplicative kernel is a design choice SKI can be employed to great effect in any moderate-dimensional context with grid-spaced input data. However, such an input pattern is very restrictive, and so the final component of SKI is to use local interpolation to induce Kronecker structure:

$$K_{\boldsymbol{x},\boldsymbol{x}} \approx W K_{U,U} W^T$$

where W is an extremely entry-sparse matrix which locally interpolates grid-spaced covariance matrix $K_{U,U}$ to the true input positions (for example W might be generated by local cubic interpolation). This matrix approximation improves as grid density increases, and a simple correction to the log-determinant calculation provides the necessary accuracy (eigenvalues of $K_{U,U}$ need only be rescaled by n/m, where m is the number of grid points in U). These approximations and structure-exploiting techniques comprise SKI. Adjustments to complexity in terms of m and many more details about SKI are provided in [16], but are not relevant to the project and so are omitted here. Since the original paper positing SKI, Lanczoz-based approximation methods for log-determinant approximation have come into favor for their flexibility and relative speed [14, 5], but that is out of scope for this project.

1.3 Toeplitz Methods and Kernel Choices

While SKI itself provides large improvements to the computational complexity of GP inference, significant further improvements are possible and indeed simple to provide if two other conditions are met. First, the multiplicative components of the kernel, k_ℓ, must be *stationary*, meaning they depend only on the distance between inputs, and not on input position itself. Such kernels are very commonly employed; indeed the most widely used kernel in practice is a simple smoothing function, the Gaussian or squared-exponential kernel

$$k_{\text{SE}}(x, x') = \sigma^2 \exp\bigg(- \frac{|x - x'|^2}{2\omega^2} \bigg)$$

where the hyperparameters $\theta = (\sigma, \omega)$ represent variance (average distance of sample functions from their mean) and lengthscale of variation (how fast a sample function varies relative to the input space). The majority of commonly-employed kernels are in fact stationary, including the standard periodic kernel (used to produce function samples which are exactly periodic), Matérn kernel family, and the rational quadratic kernel. Furthermore, sums and products of stationary kernels are themselves stationary, so the restriction of this first condition is not difficult to retain in most situations. Still, some useful kernels are nonstationary, and in problems where signal varies qualitatively over the domain stationarity can be a confounding factor in model fit and predictive power.

The second condition is that the grid U must be evenly spaced in each dimension (although that spacing need not be uniform across dimensions, as each Kronecker component can be treated independently). This requires that the domain be connected, and that input data be reasonably well-distributed over that domain (so that the interpolation done by W is meaningful). Still, because U is a design choice in the same sense that k is, this restriction is often simple to obey and situations where it is not appropriate are generally straightforward to identify.

If both of these conditions are met, then the component matrices K_ℓ take on Toeplitz structure, meaning they are constant along diagonals. This occurs because regular grid spacing means $|x_i - x_j| = |i - j|D$ for some fixed distance D, meaning the fixed index spacing along a diagonal of $k_\ell(\boldsymbol{x}, \boldsymbol{x})$ provides constant values. Toeplitz matrices can be embedded into an associated circulant matrix (circulant matrices are Toeplitz but with a cyclic value pattern along the anti-diagonal) of twice the size, and then that matrix can be diagonalized by the discrete Fourier transform (DFT) in $\mathcal{O}(n \log n)$ time.

From the fact that circulant (and therefore Toeplitz) matrices can be diagonalized by the DFT we have access to very fast ($\mathcal{O}(n_\ell \log n_\ell)$) matrix-vector multiplications, as well as much faster log-determinant computation. In particular, given the first column c of a circulant matrix and a vector x we can apply the circulant matrix to x via

$$y = \mathcal{F}_n^{-1}\big(\mathcal{F}(c) \odot \mathcal{F}(x)\big)$$

where $\odot$ represents Hadamard (elementwise) multiplication.

Aside: Two kernels we will employ in this project are the Matérn 3/2 kernel and the polynomial kernel, below. The Matérn kernel is stationary, while the polynomial kernel is not.

$$\begin{aligned} k_{\text{Mat}}(x, x') &= \sigma^2 \bigg(1 + \frac{\sqrt{3}|x - x'|}{\rho} \bigg) \exp\bigg(- \frac{\sqrt{3}|x - x'|}{\rho} \bigg), \\ k_{\text{P}}(x, x') &= (\alpha x^T x' + \beta)^\gamma. \end{aligned}$$

1.4 Hierarchical Rank-Structured Factorizations

The next piece of this puzzle is a class of approaches to kernel methods which have been employed primarily in the world of PDE and related problems. Stemming largely from concepts such as those found in the fast multipole method [6], rank-structured decompositions seek to exploit the fact that long-range interactions carry less information and have less effect on many physical problems than close-range interactions. Hierarchical matrix factorizations use the rank-structure that arises from this property to well-approximate matrices in a form that allows them to be used much more quickly for common operations.

Hierarchical matrix theory [2] describes matrices with recursive structure, typically rank structure. The most useful matrix factorizations arising from this field often permute rows and columns of the original matrix freely, sometimes by decomposing the domain via a KD-tree, allowing low-rank blocks to be approximated by some outer product recursively. Two such matrix factorizations are described next.

1.4.1 HODLR Matrix Factorization

Hierarchical Off-Diagonal Low-Rank (or HODLR) matrices are structured such that when partitioned as

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

the diagonal blocks have full rank and the off-diagonal blocks have low rank. This rank structure is maintained when partitioning the diagonal blocks themselves in the same way, down to some level of minimum granularity (typically the level at which dense linear algebra is considered sufficiently inexpensive, in practice) [1, 7].

Constructing a HODLR factorization for a given matrix requires an efficient factorization scheme for the off-diagonal blocks. While the choice of particular algorithm varies between implementations, *hm-toolbox* [12] uses column-pivoted Householder QR with a spectral norm error measure for dense factorization, although it provides the option of using the truncated SVD instead. With this subroutine taken for granted, a HODLR factorization of p levels can be computed recursively. In the following 3-level example, it is clear that dense blocks A_{jj} must be inverted explicitly, but all other necessary inverses use the results of that computation alongside the Sherman-Morrison-Woodbury identity, which is often useful for linear solves involving low-rank components.

$$K = \begin{bmatrix} A_{11} & 0 & 0 & 0 \\ 0 & A_{22} & 0 & 0 \\ 0 & 0 & A_{33} & 0 \\ 0 & 0 & 0 & A_{44} \end{bmatrix} \begin{bmatrix} I_{n/4} & A_{11}^{-1} U_1^{(2)} V_1^{(2)\,T} & 0 & 0 \\ A_{22}^{-1} V_1^{(2)} U_1^{(2)\,T} & I_{n/4} & 0 & 0 \\ 0 & 0 & I_{n/4} & A_{33}^{-1} U_2^{(2)} V_2^{(2)\,T} \\ 0 & 0 & A_{44}^{-1} V_2^{(2)} U_2^{(2)\,T} & I_{n/4} \end{bmatrix} \begin{bmatrix} I_{n/2} & A_1^{-1} U_1^{(1)} V_1^{(1)\,T} \\ A_2^{-1} V_1^{(1)} U_1^{(1)\,T} & I_{n/2} \end{bmatrix}.$$

With a factorization depth of $\mathcal{O}(\log n)$, a HODLR matrix can be factored in $\mathcal{O}(n \log^2 n)$ work, after which the inverse and determinants are computable in $\mathcal{O}(n \log n)$ work [12, 1]. Note that matrix-vector products are also improved to the same complexity.

1.4.2 Recursive Skeletonization Factorization

Recursive skeletonization is a hierarchical matrix factorization procedure that relies more heavily on concepts arising from the fast multipole method and PDE problems than the HODLR factorization. It carries more overhead and uses a much more complex rank structure. The first approaches to the main idea used additive factorizations [11, 9], while more recent formulations use a multiplicative description [13] (which is used in this project). Due to its complexity the details of this factorization are not provided here, see [13] for a complete description.

For a matrix whose entries come from an "admissible" function, roughly speaking one with the property that well-separated subdomains interact with low numerical rank, the core idea of this factorization is to identify far-field interactions and compress them, as with any such factorization. The choice of compression method used here is the *interpolative decomposition*, which partitions columns of the relevant interaction submatrix into skeleton columns and redundant columns. Redundant columns are then approximated by interpolation from the set of skeleton columns, and if the low-rank assumption is correct then this interpolation can be made ϵ-accurate while the number of skeleton columns remains low.

Another key idea in the recursive skeletonization procedure is the use of a proxy surface to avoid some direct computation involving far-field interactions. The spatial domain is decomposed hierarchically via a KD-tree, and around each subdomain Green's identity is used to compress far-field interactions onto a uniform (spherical) surface surrounding that subdomain. Outgoing interactions can also be compressed in the same way, greatly reducing the size of interpolative decomposition required via the *proxy trick* [3].

Employing the interpolative decomposition wherever low-rank compression is necessary, recursive skeletonization proceeds by block-elimination, first permuting rows and columns to collect points within interacting regions, then by multiplication by block-updates to the identity matrix (which are all that is necessary for the interpolative decomposition, once the skeleton sets are determined). This procedure iterates hierarchically, with skeleton sets defined at each level of the domain decomposition. The result is a block-diagonal matrix containing "remainders" of the recursive block elimination procedure, multiplied on either side by one permutation matrix and a large product of skeletonization matrices, each of which is decomposable into one permutation matrix and a product of two noninteracting block updates to the identity (from the interpolative decomposition).

Time complexity for this decomposition is fairly involved, but with a few assumptions (namely that skeleton sets grow only as some power of domain decomposition level, and that there are on the order of $\log n$ skeleton points at the lowest level) it can be shown that matrix vector products, linear solves, and even factorization cost and memory requirement scale as $\mathcal{O}(n)$, with constants depending on tolerance ϵ and problem dimensionality. Here it is important to note that the initial tree decomposition of the domain costs $\mathcal{O}(n \log n)$, but in practice this cost is negligible compared to factorization.

2 Core Concept and Theory

As described in the introduction, SKI uses a specifically structured kernel function and interpolation to grid-spaced data to generate Kronecker structure in the kernel matrix. While this is a significant runtime improvement on its own, it only competes with the fastest approaches to GP models when additional Toeplitz structure is exploited. Because such structure restricts problem domains to a *uniform* grid and stationary component kernels $k_\ell(\cdot, \cdot)$, this project's goal is to speed up default SKI while avoiding such restrictions.

Toeplitz structure is, in the context of kernel matrices, an algebraic (and therefore very brittle) property. That is, any tiny deviation from assumptions such as a "nearly" stationary kernel or a not-quite-uniform grid removes the property entirely. Meanwhile typical kernel functions are endowed with much more robust properties which can be exploited for faster inference. As in the case of Green's functions in PDE problems, the majority of useful kernels describe strong correlations between nearby points and significantly weaker correlations between distant regions of the problem domain. That is, they exhibit *spatial decay*, although notable exceptions are periodic and quasiperiodic kernels. Here it should also be noted that most nonstationary kernels only exhibit such decay in certain coordinate subspaces (e.g. the linear kernel $k(x, x') \propto \langle x, x' \rangle$ decays as angle between domain points moves towards orthogonality).

Spatial decay yields lower signal for long-range interactions, reducing the detail of information provided by submatrices associated with those interactions. This is a hallmark of rank-structured matrices, and so decompositions such as HODLR and RS should be well-adapted to exploiting that property. Not every kernel exhibits hierarchical rank structure, and some (again, the periodic kernel is a good example) explicitly break the assumptions necessary for theoretical performance of the two algorithms, but many useful kernels do - and some of those kernels are nonstationary, where Toeplitz methods are not applicable.

Therefore, the core concept of this project is to apply HODLR and RS decompositions to SKI and test their performance relative to default and Toeplitz methods. In particular the two metrics that are of interest are how each method performs as input dimensionality increases with a fixed number of grid points per axis, and how each method performs in fixed dimensionality as input density increases.

2.1 RS SKI and HODLR SKI

Assume that inputs are spaced on a grid and that our kernel function splits multiplicatively. Then it is clear that the most efficient way to use the RS and HODLR decompositions within the SKI framework is to apply them to each 1-dimensional component matrix, and use fast matrix-vector products and determinant computations to further speed up the default SKI methodology. Because no neat way to make a diagonal correction to a linear solve exists we continue using linear conjugate gradient iteration for that purpose (exploiting fast matrix-vector products along the way). However we would prefer to avoid computing an eigendecomposition for the log-determinant, even though it can be sped up in an iterative algorithm using access to fast matrix-vector products. Therefore we lose default SKI's log-determinant methodology, and must construct our own variation.

Because HODLR and RS factorizations provide fast determinant calculation, our best path forward is to make a diagonal correction to the determinant of the Kronecker product. The matrix determinant lemma states

$$\det(A + uv^T) = (1 + v^T A^{-1} u) \det(A).$$

Using this fact, we can see the path to a correction quite easily, using only the diagonal of the kernel matrix's inverse:

$$\begin{aligned}
\det\left(\bigotimes_{\ell=1}^{d} K_\ell + \sigma^2 I\right) &= \det\left(\bigotimes_{\ell=1}^{d} K_\ell + \sum_{j=1}^{n} \sigma^2 e_j e_j^T\right) \\
&= \det\left(\bigotimes_{\ell=1}^{d} K_\ell + \sum_{j=2}^{n} \sigma^2 e_j e_j^T\right)(1 + \sigma^2 e_1^T K^{-1} e_1) \\
&= \det\left(\bigotimes_{\ell=1}^{d} K_\ell + \sum_{j=2}^{n} \sigma^2 e_j e_j^T\right)(1 + \sigma^2 K^{-1}(1,1)) \\
&= \det\left(\bigotimes_{\ell=1}^{d} K_\ell + \sum_{j=3}^{n} \sigma^2 e_j e_j^T\right)(1 + \sigma^2 K_{11}^{-1})(1 + \sigma^2 K_{22}^{-1}) \\
&= \cdots \\
&= \det\left(\bigotimes_{\ell=1}^{d} K_\ell\right) \prod_{j=1}^{n} (1 + \sigma^2 K_{jj}^{-1})
\end{aligned}$$

Furthermore, the determinant of a Kronecker product can be expressed in terms of determinants of component matrices - if K_ℓ is of size $n_\ell \times n_\ell$, then

$$\det\left(\bigotimes_{\ell=1}^{d} K_\ell\right) = \prod_{\ell=1}^{d} \det(K_\ell)^{n_\ell}.$$

Using these two facts as well as the fact that our goal is the log-determinant, we come up with the following general approach for computing that quantity, in both the HODLR and RS case:

$$\log\det\left(\bigotimes_{\ell=1}^{d} K_\ell + \sigma^2 I\right) = \text{sum}(\log(\mathbf{1} + \sigma^2 \text{diag}(K^{-1}))) + \sum_{\ell=1}^{d} n_\ell \log\det(K_\ell)$$

where the first term is simply the sum of all elements in a vector of ones plus the scaled diagonal of K^{-1}.

While it may seem prohibitively expensive to directly compute the inverse of the full covariance matrix, note that Kronecker structure makes this much easier. In fact, both necessary operations "commute" with the Kronecker product - matrix inversion and taking the diagonal. That is,

$$\left(\bigotimes_{\ell=1}^{d} K_\ell\right)^{-1} = \bigotimes_{\ell=1}^{d} K_\ell^{-1}, \text{ and } \text{diag}\left(\bigotimes_{\ell=1}^{d} K_\ell\right) = \bigotimes_{\ell=1}^{d} \text{diag}(K_\ell).$$

As such it is straightforward to compute this correction by extracting the diagonal of each component's inverse and taking the Kronecker product of the results. Furthermore the implementation of RS used in this project provides a fast method for extracting the diagonal of the inverse, at less expense than forming the entire inverse matrix.

2.2 HODLR-Toeplitz SKI

The HODLR implementation used in this project includes a special constructor for Toeplitz matrices. While it did not change the methodology employed, this constructor was compared to the default HODLR constructor in experiments where Toeplitz structure was present. Based on a cursory exploration of the codebase, it does not seem that the Toeplitz structure is taken particular advantage of beyond faster factorization, but it should be included in the list of altered methodologies employed in this project.

2.3 Complexity

No discussion of these methods would be complete without describing their asymptotic complexity. However, since a number of the useful results depend on conditions that may or may not be met by any given kernel matrix (e.g. the number of hierarchical levels and the exact properties of the kernel function) it should be noted that the discussion here is limited to regimes where the complexity described for the factorizations in question holds.

It should also be noted that specific implementations of these ideas may vary in computational complexity, so this discussion uses the figures provided by the authors of the implementations used in practice.

In the following discussion we will omit the tree decomposition cost required for RS factorization, since in practice it is smaller than the factorization itself (as mentioned in section 1.4.2). Also note that we omit the Toeplitz-HODLR SKI methodology, since less information is provided by the author of [12] regarding its complexity, and it is included in experiments mostly out of personal curiosity, since Toeplitz methods are already "sufficiently fast".

2.3.1 HODLR SKI Complexity

Suppose that the d-dimensional grid has n_ℓ values on axis ℓ, and that a HODLR decomposition of each component matrix K_ℓ will be of rank r in each off-diagonal component that is approximated by a low-rank matrix. Then each component factorization comes at a cost of $\mathcal{O}(kn_\ell^2)$, and matrix-vector products scale as $\mathcal{O}(kn_\ell \log n_\ell)$. However, the cost of determinants is not made entirely clear by the author of [12] - a foray into the source code tells us it is based on the LU decomposition, which for a HODLR matrix costs $\mathcal{O}(kn_\ell \log^2 n_\ell)$.

In the end, HODLR SKI's complexity is dominated by the factorization cost, $\mathcal{O}\big(k \sum n_\ell^2\big)$. However in practice this cost carries much less overhead than the RS factorization, and so for attainable problem sizes a practical scaling may be closer to $\mathcal{O}\big(k \sum n_\ell \log n_\ell\big)$ or $\mathcal{O}\big(k \sum n_\ell \log^2 n_\ell\big)$.

It is reasonable to assume that for most problems the input grid has a similar number of values per axis, that is $n_\ell \approx \sqrt[d]{n}$. In that case complexity scales as $\mathcal{O}\big(kdn^{\frac{2}{d}}\big)$, but the post-factorization components of computation scale as $\mathcal{O}\big(\frac{k}{d}\sqrt[d]{n} \log^2 n)$ and $\mathcal{O}\big(k\sqrt[d]{n} \log n)$.

The only place Toeplitz-HODLR structure seems to improve these complexity results is in construction, where it costs only $\mathcal{O}(kn_\ell \log n_\ell)$ per component.

2.3.2 RS SKI Complexity

It is much more straightforward to determine complexity in the RS-factored case. Supposing again that the d-dimensional grid has n_ℓ values on axis ℓ, each component factorization comes at a cost of $\mathcal{O}(n_\ell)$, and once factored determinants and matrix-vector products also scale as $\mathcal{O}(n_\ell)$. In addition, [8] states in the codebase documentation that the special function `rskelf_diag`, which extracts the diagonal of an RS-factored matrix or its inverse, has typical complexity of the same as the constructor itself. Therefore the overall cost for factorization, log-determinant computation, and linear solves via (hopefully preconditioned) conjugate gradients scales asymptotically as $\mathcal{O}\big(\sum n_\ell\big)$.

If again we assume that for most problems the input grid has a similar number of values per axis, that is $n_\ell \approx \sqrt[d]{n}$. In that case, complexity would scale as $\mathcal{O}(d\sqrt[d]{n})$, a spectacular theoretical performance. If instead we assume that each axis contains approximately m unique values, then even as dimension scales up we have complexity $\mathcal{O}(d\sqrt[d]{m^d}) = \mathcal{O}(dm)$, which is remarkable. Furthermore this complexity result is independent of the dimensionality constants associated with the RS factorization, since each such factorization is over a 1-dimensional domain.

In reality, the large amount of overhead required by the RS factorization dominates the cost for any reasonably sized problem. Still, asymptotic results such as these are exciting on their own and more than enough reason to test this methodology on large input sets.

3 Implementation and Experiments

In this section is a description of my finalized implementation of SKI with HODLR and RS factorizations, as well as a detailed description of the computational experiments done to test these approaches. Both required several iterations to be mathematically reasonable and relatively efficient. As such all that is presented here is their final form, although this portion of the project proved more difficult than any other.

3.1 Implementation

Code for this project can be found at `github.com/MaxJenquin/DataSparse-Project`. HODLR factorization implementation can be found at `github.com/numpi/hm-toolbox`, and RS factorization implementation can be found at `github.com/klho/FLAM`.

SKI is most completely implemented in the Python library GPyTorch, built on the popular machine learning library PyTorch. However, in that form it is both abstruse and challenging to augment without intimate knowledge of the PyTorch platform. In addition, adding HODLR and RS functionality to an existing SKI codebase would require a detailed implementation of the algorithms themselves in PyTorch, which would have become the central part of the project due to the time required.

Instead, I chose to use existing implementations of HODLR and RS factorizations in Matlab, and integrate those into a basic SKI implementation built from scratch. In particular, from the HODLR and HSS codebase known as hm-toolbox [12] I used the default HODLR factorization and associated subroutines, as well as the Toeplitz HODLR factorization. From the recursive skeletonization codebase known as FLAM [8] I used the multiplicative RS factorization and associated subroutines.

A full-fledged SKI implementation requires a number of elements, some of which were too complex to allow time for iteration of my experiments and methods. A full-fledged implementation would also involve some object-oriented programming in Matlab, which is to say a "SKI-covariance" class that serves the purpose of the more rudimentary code I used to assemble my experiments. Below is a full accounting of what my relatively lightweight SKI implementation entailed.

Implemented:

- Kronecker matrix-vector product routine
  - Customized unfolding and refolding functions for component application
  - Various corrections for dealing with different input formats (default, Toeplitz, HODLR, RS each require a small amount of special treatment)
- Linear conjugate gradient method for iteratively computing linear solves
- Toeplitz matrix-vector multiplication
- Log-determinant correction routine
- Various small computational tasks

Not Implemented:

- SKI class for consistency and re-usability
- Customized iterative eigendecomposition algorithm, e.g. the QZ algorithm, to take advantage of fast matrix-vector products in the Toeplitz case
- Preconditioning for linear conjugate gradient method
- Interpolation routines and handling of interpolation matrices

For the eigendecomposition, I used Matlab's built-in `eig` function, because implementing a full adaptive QZ algorithm with a matrix-vector product function as input would have been relatively time-consuming. For the same reason (and others) I avoided implementing preconditioning for the linear conjugate gradient subroutine.

Partway through my iterated adjustment of experiments, I realized how large of a role preconditioning plays in the fast convergence of linear conjugate gradient solves, especially for kernelized covariance problems of the type that SKI is built to handle. In fact SKI matrices are in general not simple to precondition [4], and while building preconditioner matrices is vital to a full-fledged implementation of SKI, I decided that because each case (default SKI, Toeplitz SKI, HODLR SKI, and RS SKI) would require different approaches for comparable gains, I would instead focus on ironing out details in experiment design and log-determinant computation.

As such the results section provides careful treatment of timing for the conjugate gradient step and the rest of the necessary computations. Perhaps the most meaningful result is the timing associated with decomposition and log-determinant calculation, although per-iteration linear conjugate gradient timing is also informative.

Ultimately, this resulted in an experimental flaw. Due to the lack of QZ algorithm implementation with allowance for Toeplitz matrix-vector products and the pivot away from presenting timing results involving full conjugate gradient solves to presenting per-iteration timing results separately, Toeplitz and default SKI were reduced to the same procedure for the primary timing results presented (e.g. figure 1, figure 4).

For simplicity, inputs were assumed to already lie on an appropriate grid in all experiments, since that aspect of each methodology is unchanged between approaches used here. Therefore implementing interpolation was not necessary.

3.2 Experiments

The goal of the experiments done for this project was to well-probe the performance of each tested approach in several generic cases. No particular real-world problem was solved, in favor of more time spent pushing the size limit boundaries of manufactured problems. Several dichotomies were put to the test, detailed below.

3.2.1 Stationarity

One major shortcoming of the Toeplitz SKI framework is that it only works with stationary kernels. Therefore in order to compare RS SKI and HODLR SKI to the fastest default SKI methodology at least one experiment needed to be done with a stationary kernel. For this purpose the Matérn 3/2 kernel was chosen.

At the same time, one major advantage of RS and HODLR SKI is that they can be employed on nonstationary component kernels, so long as rank structure can still be found. Parallel to the experiments with stationary kernels, then, similar experiments with a nonstationary kernel were needed in order to compare the performance of RS, HODLR and default SKI to each other and to their performance in the stationary case.

3.2.2 Domain Geometry

The other major shortcoming of the Toeplitz SKI framework is that it requires the underlying grid be uniform on each axis. This means that data must come from a single contiguous region, which can preclude certain data sets where gaps in observation occur or introduce unnecessary levels of error when data are not uniformly distributed over the domain. In order to compare it to RS and HODLR SKI, at least one experiment needed to be done on a uniform grid. In particular, the uniform grid used had between $10S$ and $15S$ points per axis, and was over the domain $[-1, 1]^d$ for a d-dimensional problem of data scale S.

Meanwhile, RS and HODLR SKI have the potential for their performance to actually be improved on a domain comprised of well-separated subdomains, especially RS SKI due to the fact that it explicitly partitions inputs spatially and uses that information to exploit rank structure. To compare performance among methods in this case and against the uniform grid case, at least one experiment with well-separated subdomains needed to be done. In particular, the nonuniform grid used was over the domain $[-1, -0.7]^d \cup [0.7, 1]^d$, with between $5S$ and $7S$ randomly-chosen points per axis per subdomain, for a total of between $10S$ and $14S$ points per axis. Again, these details are for a

d-dimensional problem of data scale S.

3.2.3 Data Size

Two ways to vary the size of input data were explored, each with a different goal. One experiment fixed dimensionality at $d = 3$, and iterated over data scales S as detailed above. This served to keep default and Toeplitz SKI in their range of comfort, where their performance is not yet degraded by the all-too-familiar curse of dimensionality.

The second experiment was designed to address the fact that SKI typically functions best on low to moderate dimensional problems. This shortcoming is universal to nearly all GP methodologies, and undoubtedly the concepts in this project are not powerful enough to overcome the curse of dimensionality, but regardless performance as dimension grows is an important metric for applicability. For these experiments, data scale S was fixed at 1, and dimensionality d was increased.

3.2.4 The Experiments Themselves

In light of the above considerations, four experiments were constructed.

The first experiment tested default, Toeplitz, RS, HODLR and HODLR-Toeplitz SKI for a 3-dimensional problem on the uniform grid domain, increasing S until problem sizes ballooned dramatically. Solve times for various tasks, problem sizes, and conjugate gradient iterations required were all recorded. The kernel function for this experiment was the stationary Matérn 3/2 kernel.

The second experiment was identical to the first, except with S fixed at 1 and problem dimensionality d increasing until problem sizes increased dramatically. The same information was also recorded for this experiment. The kernel function for this experiment was the stationary Matérn 3/2 kernel.

The third experiment tested default, RS and HODLR SKI for a 3-dimensional problem on the non-uniform grid domain, increasing S in the same manner as problem 1. The same information was recorded. The kernel function for this experiment was the product of the stationary Matérn 3/2 kernel and the nonstationary polynomial kernel.

The fourth and final experiment tested default, RS and HODLR SKI on the non-uniform grid domain, fixing S at 1 and increasing the problem dimensionality d as in experiment 2. The same information was also recorded here. The kernel function for this experiment was the product of the stationary Matérn 3/2 kernel and the nonstationary polynomial kernel.

Between the four of them, these experiments provide a relatively complete view of the behavior of these proposed methodologies in different situations.

3.2.5 Two Further Experiments

It should be noted that experiments were severely limited by the fact that conjugate gradient runtime was massive (due in large part to the lack of preconditioning). Hoping to use that information instead of wasting computational time, I implemented a 10,000 iteration limit after which point the algorithm terminated early without convergence (a lower limit such as 1000 iterations let only the smallest of tested systems converge, hence the large value of 10,000).

While analyzing results and determining the bottleneck to be in the conjugate gradient iteration due to lack of preconditioning, the limitations of that iteration on problem dimension were painfully obvious with regards to runtime. To extend the analysis a few dimensions further, pared-down dimension-scaling experiments were constructed, although they only reached $d = 8$ before Matlab's memory allocation system was pushed to its limit. These experiments exactly mirrored the dimension scaling experiments in the uniform/stationary and nonuniform/nonstationary cases, except with extended problem dimension ranges and only the log-determinant and factorization steps timed. In favor of realistic problem constraints (e.g. 10-15 grid points per axis), the relationship between problem size and dimensionality was not altered to push above $d = 8$.

4 Numerical Results and Discussion

Due to the poor behavior of linear conjugate gradient iteration without preconditioning for systems such as these, full results for solve times are not presented. Instead, factorization and log-determinant computation times are plotted, and time-per-iteration of conjugate gradients are described as well. Then, if appropriate preconditioning makes iterations required for convergence nearly uniform across all methodologies this alternative measurement is a valid proxy for time required for linear solves via conjugate gradients. Intuitively such preconditioning should be possible, since the underlying matrix represented by each factorization is the same.

It is important to note here that due to the fact that the accelerated QZ algorithm was not implemented, Toeplitz and default SKI require the same computation for factorization and log-determinant calculation! This oversight calls into question how competitive our rank-structured factorization SKI methods are with Toeplitz SKI, although comparisons to default SKI are still entirely valid. For plots of this type, Toeplitz and default methods should be exactly overlaid (in general the Toeplitz data covers the default data).

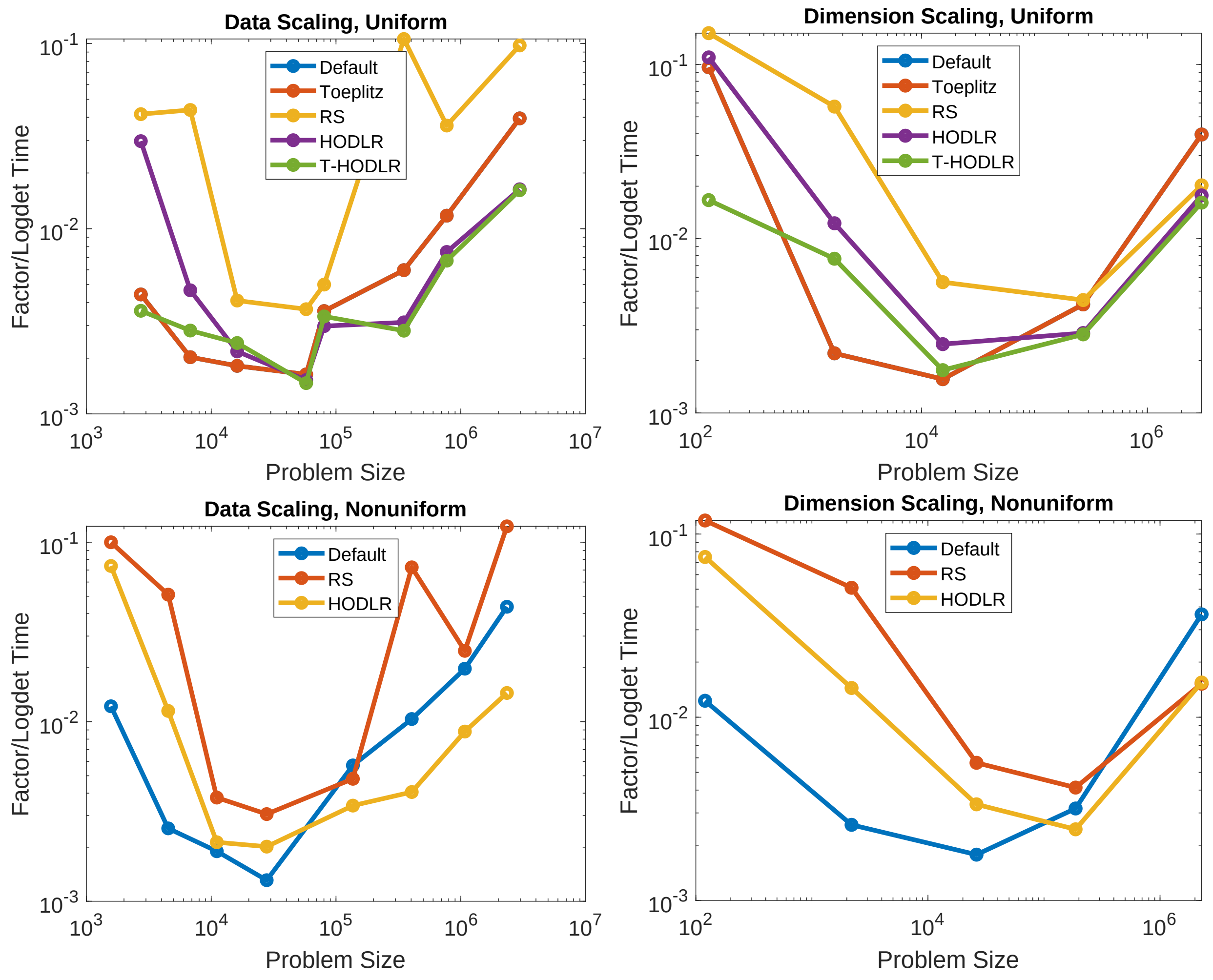

Figure 1: Factorization plus log-determinant computation times versus problem size for the four main experiments. Clockwise from top left: uniform grid with varying data density, uniform grid with varying problem dimensionality, nonuniform grid with varying problem dimensionality, and nonuniform grid with varying data density.

In figures 1 and 4, factorization and log-determinant computation times are plotted against problem size for each experiment. These results indicate that while function calls and factorization time slow HODLR SKI down for small problems, it and Toeplitz-HODLR SKI outperform the competition for large problems, regardless of how problem size was scaled (dimension vs density).

Meanwhile, looking at figures 1 and the center plot of figure 3 it is clear that RS SKI performs much better when problems are scaled via dimensionality than via density - perhaps explained in part by the fact that the lengthscale parameter of the underlying kernel function did not change, and therefore denser inputs may have begun to violate the rank-structure assumption made by the RS factorization. Still, its performance on high-dimension problems caught up to that of HODLR by the end of the primary experiments in figure 1, an encouraging result.

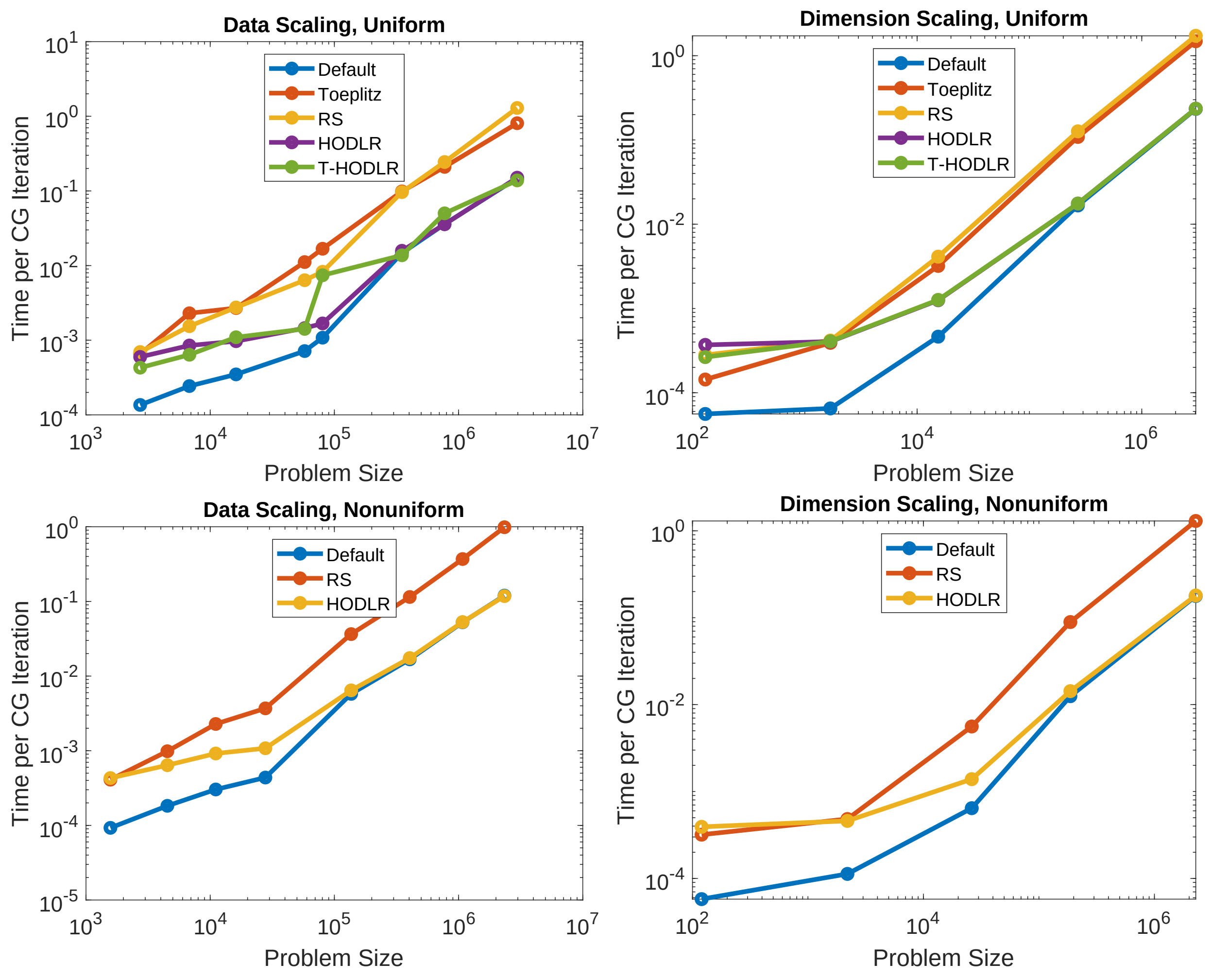

Figure 2: Single CG-iteration times versus problem size for the four main experiments. Clockwise from top left: Uniform grid with varying data density, Uniform grid with varying problem dimensionality, Nonuniform grid with varying problem dimensionality, and Nonuniform grid with varying data density.

Comparisons between the four main experiments in figure 3 reveal only that RS SKI has trouble with data density as mentioned previously, and that HODLR SKI has surprising trouble with nonuniform grid data scaling. This second result is fairly counterintuitive, since the off-diagonal blocks of component matrices in that well-separated case should be most amenable to the HODLR rank structure assumption, at least on the highest level of factorization. It is possible that far-from-uniform grid spacings played a role in this phenomenon, requiring further investigation.

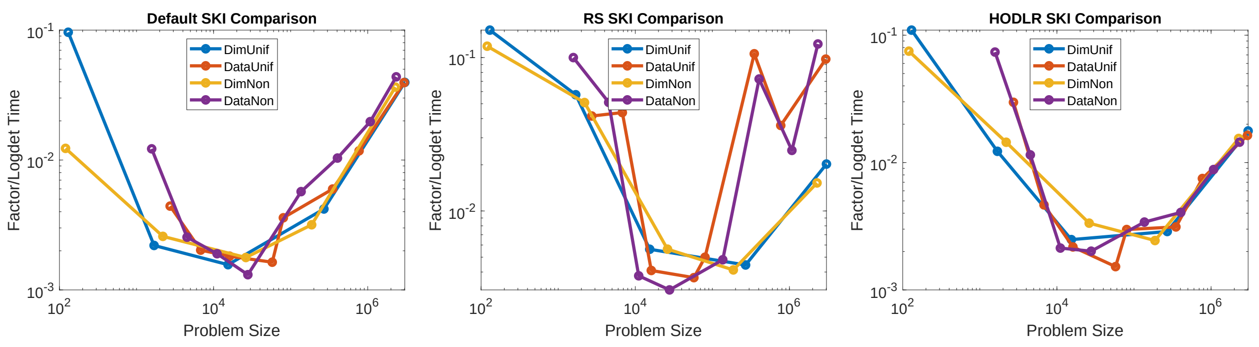

Figure 3: Uniform vs nonuniform grid experiment results for the three methodologies applicable to both cases. Left: default SKI, center: RS SKI, right: HODLR SKI. On each legend the four experiments are labeled according to the quantity they varied to scale the problem, and the grid spacing they used.

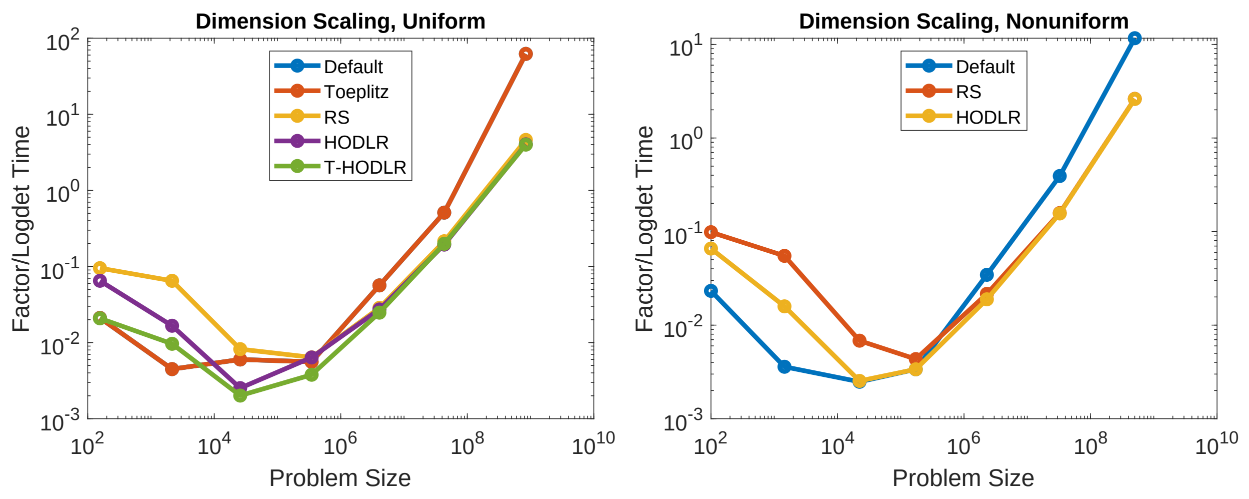

Figure 4: Extended experiment factorization plus log-determinant computation times versus problem size, scaled via problem dimensionality (as opposed to data density). Left: uniform grid, right: nonuniform grid.

Extending the dimensional scaling experiments further, figure 4 verifies that RS SKI catches up to HODLR SKI and keeps pace well as dimension continues to scale a bit further, while the gap between default methods and the factored methods increases. Meanwhile, little difference between uniform grids with stationary kernels and nonuniform grids with nonstationary kernels is observed for dimension-scaling problems, according to figure 5.

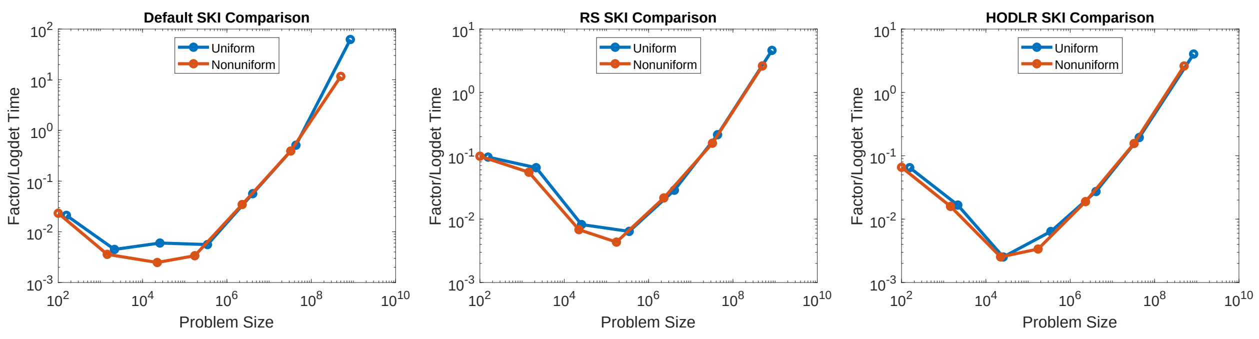

Figure 5: Extended experiment results, uniform vs nonuniform grid, for the three methodologies applicable to both cases. Left: default SKI, center: RS SKI, right: HODLR SKI. On each legend the two extra experiments are labeled according to the grid spacing they used.

Finally figure 2 yields the most surprising result of all, that default Kronecker multiplication proved the fastest method by which to do a single iteration of the conjugate gradient algorithm. Although HODLR and Toeplitz-HODLR catch up for large problems, this result is thoroughly counterintuitive, and is most likely explained by the poorly-optimized form in which the iteration routine was constructed. Memory management, fewer function calls, and a less rudimentary code structure would no doubt affect these results significantly, although it is questionable whether the RS implementation would be able to close this obvious gap in performance. Perhaps an implementation designed for the task would be more amenable to this analysis.

5 Conclusions and Future Exploration

This project has presented a cursory investigation of the application of rank-structured hierarchical matrix factorizations (HODLR and RS in particular) to the SKI framework for acceleration of kernel method operations. Two problem scalings were probed, as well as the relationship between kernel type and rank-structure assumptions, providing incomplete but compelling results that indicate this approach is worthwhile in at least a marginal sense.

While the results of this project were confounded by implementational issues, the upshot is that for large enough problems where rank-structure assumptions hold, these factorizations indeed improve speed of evaluation. Whether they truly compete with Toeplitz SKI methods is a question as yet unanswered, and the more likely answer is that they do not as a result of the overhead involved relative to the simplicity of Toeplitz methods. However, the asymptotic scaling of RS and the practical performance of HODLR are both strong indicators that for suffiently large problems where Toeplitz structure is not available, this approach could be quite useful. In particular the asymptotic scaling of these methods into high dimensional problems is intriguing, and much more testing in that arena is warranted.

Many possibilities for future work are indicated by this project and its results. Preconditioning schemes for Kronecker products of hierarchically factored matrices make this idea a practical reality, but the landscape of such preconditioners is complex and no doubt will require careful thought. Meanwhile, the relationship between kernel length scale, input density, and the rank-structure assumptions of these and other algorithms provide another direction of investigation, in order to explore the best option for different kernels and different situations.

Additionally, factorizations beyond RS and HODLR could outperform both, provided their overhead is low and their rank-structure assumptions hold true, and as more are developed for increasingly specialized applications the eventual landscape of a fully realized methodology built on this idea becomes quite specified, perhaps to the point of different factorizations for different Kronecker components, depending on the underlying component kernel and input set.

Finally, work towards the holy grail of accelerated GP inference, fast kernel matrix updates based on hyperparameter updates, may prove interesting for some kernel/decomposition combinations. While difficult in general, if some hierarchical decomposition is amenable to low-rank updates then further acceleration is possible using sparse updates to component matrices (therefore quite small sparse updates) and occasional refactorization is an exciting prospect when asymptotic scaling with dimension is so well-behaved. In addition Kronecker structure would no doubt help such a methodology scale to higher dimensions, as it has done throughout this project.

References

[1] Sivaram Ambikasaran, Daniel Foreman-Mackey, Leslie Greengard, David W Hogg, and Michael O'Neil. Fast direct methods for gaussian processes. *IEEE transactions on pattern analysis and machine intelligence*, 38(2):252–265, 2015.

[2] Steffen Börm, Lars Grasedyck, and Wolfgang Hackbusch. Introduction to hierarchical matrices with applications. *Engineering analysis with boundary elements*, 27(5):405–422, 2003.

[3] Hongwei Cheng, Zydrunas Gimbutas, Per-Gunnar Martinsson, and Vladimir Rokhlin. On the compression of low rank matrices. *SIAM Journal on Scientific Computing*, 26(4):1389–1404, 2005.

[4] Kurt Cutajar, Michael Osborne, John Cunningham, and Maurizio Filippone. Preconditioning kernel matrices. In *International Conference on Machine Learning*, pages 2529–2538, 2016.

[5] Kun Dong, David Eriksson, Hannes Nickisch, David Bindel, and Andrew G Wilson. Scalable log determinants for gaussian process kernel learning. In *Advances in Neural Information Processing Systems*, pages 6327–6337, 2017.

[6] Leslie Greengard and Vladimir Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 135(2):280–292, 1997.

[7] Wolfgang Hackbusch, Lars Grasedyck, and Steffen Börm. An introduction to hierarchical matrices. 2001.

[8] Kenneth L. Ho. Fast linear algebra in matlab (FLAM). Github. DOI: 10.5281/zenodo.1253581.

[9] Kenneth L Ho and Leslie Greengard. A fast direct solver for structured linear systems by recursive skeletonization. *SIAM Journal on Scientific Computing*, 34(5):A2507–A2532, 2012.

[10] Haitao Liu, Yew-Soon Ong, Xiaobo Shen, and Jianfei Cai. When gaussian process meets big data: A review of scalable gps. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[11] Per-Gunnar Martinsson and Vladimir Rokhlin. A fast direct solver for boundary integral equations in two dimensions. *Journal of Computational Physics*, 205(1):1–23, 2005.

[12] Stefano Massei, Leonardo Robol, and Daniel Kressner. hm-toolbox: Matlab software for hodlr and hss matrices. *SIAM Journal on Scientific Computing*, 42(2):C43–C68, 2020.

[13] Victor Minden, Kenneth L Ho, Anil Damle, and Lexing Ying. A recursive skeletonization factorization based on strong admissibility. *Multiscale Modeling & Simulation*, 15(2):768–796, 2017.

[14] Geoff Pleiss, Jacob R Gardner, Kilian Q Weinberger, and Andrew Gordon Wilson. Constant-time predictive distributions for gaussian processes. *arXiv preprint arXiv:1803.06058*, 2018.

[15] Christopher KI Williams and Carl Edward Rasmussen. *Gaussian processes for machine learning*, volume 2. MIT press Cambridge, MA, 2006.

[16] Andrew Wilson and Hannes Nickisch. Kernel interpolation for scalable structured gaussian processes (kiss-gp). In *International Conference on Machine Learning*, pages 1775–1784, 2015.