A SCALABLE AND FLEXIBLE FRAMEWORK FOR GAUSSIAN PROCESSES VIA MATRIX-VECTOR MULTIPLICATION

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Geoff Pleiss

August 2020

A SCALABLE AND FLEXIBLE FRAMEWORK FOR GAUSSIAN PROCESSES VIA MATRIX-VECTOR MULTIPLICATION

Geoff Pleiss, Ph.D.

Cornell University 2020

Gaussian processes (GPs) exhibit a classic tension of many machine learning methods: they possess desirable modelling capabilities yet suffer from important practical limitations. In many instances, GPs are able to offer well-calibrated uncertainty estimates, interpretable predictions, and the ability to encode prior knowledge. These properties have made them an indispensable tool for black-box optimization, time series forecasting, and high-risk applications like health care. Despite these benefits, GPs are typically not applied to datasets with more than a few thousand data points. This is in part due to an inference procedure that requires matrix inverses, determinants, and other expensive operations. Moreover, specialty models often require significant implementation efforts.

This thesis aims to alleviate these practical concerns through a single simple design decision. Taking inspiration from neural network libraries, we construct GP inference algorithms using only *matrix-vector multiplications* (MVMs) and other linear operations. This MVM-based approach simultaneously address several of these practical concerns: it reduces asymptotic complexity, effectively utilizes GPU hardware, and provides straight-forward implementations for many specialty GP models.

The chapters of this thesis each address a different aspect of Gaussian process inference. Chapter 3 introduces a MVM method for training Gaussian

process regression models (i.e. optimizing kernel/likelihood hyperparameters). This approach unifies several existing methods into a highly-parallel and stable algorithm. Chapter 4 focuses on making predictions with Gaussian processes. A memory-efficient cache, which can be computed through MVMs, significantly reduces the computation of predictive distributions. Chapter 5 introduces a multi-purpose MVM algorithm that can be used to draw samples from GP posteriors and perform approximate Gaussian process inference. All three of these methods offer speedups ranging from $4\times$ to $40\times$. Importantly, applying any of these algorithms to specialty models (e.g. multitask GPs and scalable approximations) simply requires a matrix-vector multiplication routine that exploits covariance structure afforded by the model.

The MVM methods from this thesis form the building blocks of the `GPyTorch` library, an open-sourced GP implementation designed for scalability and simple implementations. In the final chapter, we evaluate GPyTorch models on several large-scale regression datasets. Using the proposed MVM methods, we can apply exact Gaussian processes to datasets that are *2 orders of magnitude larger* than what has previously been reported—up to 1 million data points.

BIOGRAPHICAL SKETCH

Geoff Pleiss was born and raised in the Bay Area. He graduated from Olin College of Engineering in 2013 with a self-designed engineering degree, specializing in applied math and computing. After a brief stint as a software developer and consultant at Pivotal Labs in New York City, Geoff moved to Ithaca where he began his PhD work. At Cornell, Geoff studied machine learning under the advising of Kilian Q. Weinberger. During his PhD, Geoff worked at both Microsoft Research and ASAPP Inc. as a research intern. He is currently a postdoctoral researcher at Columbia University, working with John Cunningham.

This thesis is dedicated to my parents, Mike and Chris Pleiss, who have always been my role models in science. I cannot thank you enough for all of your love, encouragement, and support over the years.

(P.S. – Dad, hopefully this thesis convinces you that I'm not working on the same problems that you did back when you were a grad student.)

ACKNOWLEDGEMENTS

This thesis would not have been possible without the support of so many people. I would like to begin by thanking my advisor, Kilian Weinberger, who encouraged myself and Jake Gardner to go down this crazy Gaussian process rabbit hole. I am constantly humbled by Kilian's briliance and ability to synthesize many sub-areas of machine learning, while always finding simple and elegant solutions to research problems. Beyond his sound scientific advice, Kilian always offered much-needed support during the most challenging and frustrating moments of my PhD. Thank you for always being an inspiring mentor, a patient teacher, and a "funny guy."

I have been truly fortunate to work with many amazing collaborators. Andrew Gordon Wilson was a guiding force for much of my PhD research, and it was amazing to learn from his deep knowledge of Bayesian methods. Thank you for your encouragement and for the many thoughtful discussions about all things machine learning. To Karthik Sridharan, who always has remarkable insights no matter what the topic is. To Anil Damel and David Bindel, who have taught me much about scientific computing, and who I can trust to call me out when I say something wrong about numerics. To Martin Jankowiak and David Eriksson, who—in addition to their brilliance—have been tremendous fun to work with. (Thanks also for putting up with my coding/math errors.)

Jake Gardner has been an invaluable mentor and a trusty partner-in-crime for the research in this thesis. He introduced me to much of what I know about Gaussian processes and numerical linear algebra, and was very patient while I struggled through many of the harder concepts. The genesis of much of this research came from a summer where Jake and I decided to write our own Gaussian process library. Though Jake was the more senior student, our collaboration

back then (and now) felt like an equal partnership. It has been a true joy working with him, watching our small implementation blossom into several research projects. I look forward to continuing our work together.

The research presented in this thesis has been implemented as an open-source Gaussian process library called GPyTorch. I would like to thank so many wonderful contributors who have helped with the library's development and research. To Max Balandat, who has contributed more than anyone else (and who has taught me how to be a better Python developer). To Eytan Bakshi, who has facilitated a wonderful collaboration and helped expand the applicability of our work. To Ke Alex Wang, who is always there to help out with a pull request or an issue or a last minute NeurIPS sprint.

It has been a pleasure to be part of Kilian's extended lab group. To Felix Wu, for his never-give-up attitude; Chuan Guo, for his devotion to rigor; Yu Sun, for his jovial spirit; Gao Huang, for his amazing insightfulness; Matt Kusner, for his positivity and encouragement; Stephen Tyree, for our fun and effortless collaborations; Harry Chao, for our always stimulating conversations; Ruihan Wu, for her stellar mathematical abilities; Yan Wang, for his "surprisingly simple" solutions; Yurong You, for his computer vision wizardry; Tianyi Zhang, for his magical ability to balance 10 projects at once; and Varsha Kishore, for her willingness to join me on my crazy ideas.

Thank you to the Cornell computer science department, which is a supportive and thriving community. To Robert Kleinberg, for being a wonderful and carrying director of graduate studies. To Becky Stewart, who tirelessly fights for all PhD students and always responds promptly to my stressed-out emails.

My final thank you goes to Andrea Bruns, who's love and consistent support has made it possible for me to cross the finish line.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

CHAPTER 1

INTRODUCTION

The past decade has witnessed a wide-scale adoption of machine learning methods across numerous application domains. This surge is due to the confluence of several factors, of which we will highlight two. First, researchers have demonstrated the unparalleled **predictive capabilities** of several machine learning algorithms. At the same time, the community has developed algorithms that are increasingly **practical and easy-to-use**. Many models can be trained rapidly on consumer-level computer hardware [Howard, 2018], and high-quality software frameworks enable practitioners to rapidly develop new models. While machine learning's predictive successes have opened up new possibilities, its new-found ease-of-use has accelerated innovation and adoption.

Arguably, the machine learning algorithms which have had the broadest impact are the ones that seamlessly offer *both* predictive power and practicality. Deep neural networks perhaps best exemplify this trend. Recent innovations in network architecture [e.g. He et al., 2016, Vaswani et al., 2017, Devlin et al., 2019, Huang et al., 2019], optimization [e.g. Ioffe and Szegedy, 2015, Izmailov et al., 2018b], and theoretical understanding [e.g. Keskar et al., 2017, Jacot et al., 2018, Arora et al., 2019] have led to massive performance improvements on increasingly complex datasets. Moreover, these innovations have been complemented by the effective use of specialty compute hardware (such as GPUs and TPUs), the introduction of automatic differentiation [e.g. Paszke et al., 2017], and the development of several high-quality software implementations [e.g. Jia et al., 2014, Abadi et al., 2016, Paszke et al., 2019]. These pragmatic advances make it easy for practitioners to experiment with new models and architectures, which

has undoubtedly contributed to its profound and wide-spread successes [Goodfellow et al., 2016].

Gradient-boosted trees have a similar powerful-yet-practical story. Since their inception [Friedman, 2001, 2002], gradient boosted trees have excelled in many applications [e.g. Richardson et al., 2007, Burges, 2010]. The predictive power of these models is a product of several key attributes: for example, their remarkable generalization properties [Freund and Schapire, 1997, Schapire and Freund, 2013] and their ability to handle incomplete features [Friedman, 2001]. Equally important, these models are simple and computational efficient, in large part due to specialty parallel algorithms [e.g. Tyree et al., 2011, Ke et al., 2017] and easy-to-use software implementations such as XGBoost [Chen and Guestrin, 2016]. These advantages have made gradient-boosted decision trees a workhorse algorithm for many practitioners across application domains. According to a survey collected by Kaggle [2019], 75% of the responding data scientists regularly use gradient-boosted decision trees and the XGBoost software.

Nevertheless, for many machine learning algorithms there is still a trade-off between predictive potential and practical limitations. The focus of this thesis is **Gaussian process models** (GPs), which perhaps best exemplify this tension. Within the machine learning community, GPs have been well-regarded as a powerful model class with many desirable properties—such as calibrated uncertainty estimates and interpretable model priors. Recent work on hierarchical modelling [e.g. Damianou and Lawrence, 2013] and scalability [e.g. Wilson and Nickisch, 2015] have furthered their applicability to increasingly complex tasks. However, Gaussian processes have historically been relegated to small datasets, and the tools most commonly used for inference do not effectively utilize mod-

ern compute hardware. Using GPs requires significant implementation effort, as simple modifications like an additional output dimension might require different learning/inference procedures. These practical considerations hinder the adoption of GPs, while also limiting researchers' abilities to rapidly-prototype and make new developments. This thesis aims to address these limitations so that Gaussian processes can be powerful-*and*-practical models.

1.1 The Predictive Power of Gaussian Processes Models

Before addressing these issues, it is worth discussing why Gaussian processes are an invaluable model class for blackbox optimization [e.g. Snoek et al., 2012], robotics [e.g. Deisenroth and Rasmussen, 2011], health care [e.g. Schulam and Saria, 2015], and many other domains:

(1) Closed-form marginalization over hypotheses. Many machine learning algorithms (such as neural networks) construct a single model by optimizing over thousands or millions of parameters. Gaussian processes on the other hand marginalize over all possible predictive models $f(\cdot)$:

$$p_{\text{GP}}(y \mid \mathbf{x}) = \int_{f(\cdot)} p(y \mid \mathbf{x}, f(\cdot))\, p(f(\cdot))\ \mathrm{d}f(\cdot).$$

As a result, the predictions are less prone to overfitting [Rasmussen and Williams, 2006].

(2) Well-calibrated uncertainty estimates. The output of a Gaussian process is a predictive *distribution*, which incorporates both modelling uncertainty (e.g. how many different models could fit the data) and data uncertainty (e.g. how noisy are the training data). Consequentially, the predictive uncertainties tend to be very well calibrated to the data distribution.

(3) A flexible language for encoding prior knowledge. A Gaussian process' generalization capabilities are almost entirely determined by its modelling priors. Crucially, GP priors directly encode functional properties—such as smoothness, periodicity, or monotonicity—rather than beliefs about certain parameters. These functional properties are determined by the choice of *kernel function* (see Section 2.1.4). With the appropriate choice of prior, it is possible to generalize on datasets with as few as 10 observations [e.g. Rasmussen and Williams, 2006, Gardner et al., 2017].

(4) Interpretable predictions. The predictions from Gaussian processes (see Eqs. (2.4) and (2.5)) are not only expressive and powerful; they are also intuitive. If we view the GP's kernel function as a similarity/distance measure between two points, then the prediction at a given point $\mathbf{x}$ is simply an interpolation of nearby training points. The prediction's confidence interval is small when $\mathbf{x}$ is close to training points, and large when $\mathbf{x}$ is too far away for accurate interpolation.

These benefits are obviously applicable in the "small data" regime, where priors and marginalization are critical for meaningful predictive performance [Rasmussen and Ghahramani, 2001]. However, these properties also are beneficial for large datasets. Good uncertainty estimates and interpretable predictions are increasingly desirable for large-scale machine learning models. In addition, large datasets make it possible to use powerful families of covariance functions [Wilson and Adams, 2013, Wilson et al., 2016a, Benton et al., 2019] or hierarchical ("deep") GP models [Damianou and Lawrence, 2013, Salimbeni and Deisenroth, 2017, Jankowiak et al., 2020a].

1.2 Practical Concerns with Gaussian Processes Models

While Gaussian processes offer great predictive potential, there are several practical issues that hinder its use, especially on larger and more complex datasets.

1.2.1 Computational Complexity and Memory Requirements

Given N training data points, Gaussian process models naïvely require $\mathcal{O}(N^3)$ computation and $\mathcal{O}(N^2)$ storage. This complexity comes from computing a $N \times N$ covariance matrix of all training data and computing several non-linear operations (see Section 2.1.2). Historically, this has limited exact Gaussian process models to datasets with fewer than $1{,}000$ data points [Hensman et al., 2013].

1.2.2 Use of Modern Compute Hardware

It is worth noting that this $\mathcal{O}(N^3)$ computational complexity is not necessarily insurmountable given modern computational hardware. For example, some deep learning models require many more floating point operations (FLOPs) than large-scale GPs. A 264-layer DenseNet model [Huang et al., 2017], common on many computer vision tasks, requires 2.4×10^{18} FLOPs to train on 1.2 million images. Such a model would probably require months to train on standard CPUs, yet can be trained on 8 GPUs in a matter of hours [Howard, 2018].

Given the effectiveness of GPU acceleration on large neural networks, one might expect similar performance for large-scale Gaussian processes. Unfortunately, many GP implementations rely on the Cholesky factorization (see Sec-

tion 2.1.2), which does not benefit as readily from modern compute hardware. GPUs are designed for massively-parallelizable operations such as matrix-multiplication (which is the primary numerical operation of neural networks). A matrix-multiply between two $1{,}000 \times 1{,}000$ matrices is $10{,}000$ times faster on a GPU than on a CPU![1] The Cholesky algorithm on the other hand is inherently sequential and affords minimal parallelization; factorizing a $1{,}000 \times 1{,}000$ matrix is only 10 times faster on GPU than on CPU. This is why we cannot expect neural-network-level speedups for Cholesky-based GPs. Moreover, the Cholesky factorization requires $\mathcal{O}(N^2)$ storage. This amounts to a terabyte of memory for $N = 1{,}000{,}000$—well beyond the capacity of most GPU clusters.

1.2.3 Choosing Appropriate Approximations

To reduce the computational and memory burden, researchers have proposed numerous methods that approximate Gaussian processes with simpler models. Such models employ low-rank or structured approximations of the $N \times N$ matrices (see Section 2.1.5). Numerous advances have made these approximate methods more powerful while retaining manageable asymptotic complexities.

However, choosing a suitable approximation involves many design choices. All approximate methods introduce hyperparameters that control the speed/accuracy trade-off, while also making assumptions that might not be well suited to certain datasets. For example, variational approaches [e.g. Titsias, 2009, Hensman et al., 2013]—which are a popular general-purpose approximation—tend to overestimate the observational noise, leading to worse predictive uncertain-

[1] As measured on a NVIDIA GTX 1070 GPU versus an 8-core Intel i7 CPU.

ties [Turner and Sahani, 2011, Bauer et al., 2016]. Structured interpolation methods [Wilson and Nickisch, 2015] alleviate these biases, yet tend to be limited to low-dimensional problems. While some theoretical guarantees can guide these design decisions [Burt et al., 2019], choosing a good approximate model is ultimately dataset specific and can require expert knowledge.

1.2.4 Implementation and Programmability

One compelling advantage of neural networks is their modularity. Creating a novel neural network architecture requires significant thought and experimentation; however, *implementing* new architectures requires very little software engineering effort. Seemingly complex models like DenseNets [Huang et al., 2017] and Transformers [Vaswani et al., 2017] have surprisingly simple implementations using compositional layers. Small modifications, such as adding an additional output dimension, often require only a single additional line of code.

Gaussian processes on the other hand require significant implementation effort. Often, the *model* and the *learning/inference procedures* are tightly coupled. As an example, consider a Gaussian process with multiple output dimensions [Bonilla et al., 2008]. While this model and a standard (single-output) GP are seemingly similar, they require completely different implementations. The additional output dimension changes the structure of the prior covariance matrix (see Section 3.4), modifying the equations used for efficient inference. In the popular GPy [2012] software package, multi-output GPs and standard GPs are implemented as separate models, with multi-output GPs requiring an additional 100 lines of code. Compared to the one-line change for multi-output

neural networks, GPs are significantly more difficult to implement.

1.3 Outline of Contributions

This thesis introduces a framework that addresses these issues without sacrificing the desirable properties of GPs. Our approach is centered on a *single* critical design decision: taking inspiration from neural networks, we build GP training and inference algorithms *using only matrix-multiplication* and element-wise operations. As we will demonstrate, this reduces the asymptotic complexity of GPs, improves their GPU utilization, expands the applicability of exact methods, and simplifies implementation of specialty models. The following chapters introduce the components of our matrix-multiplication-based framework:

- In **Chapter 3**, we introduce the **BlackBox Matrix $\times$ Matrix (BBMM)** approach for training Gaussian process regression models. BBMM uses a modified version of preconditioned conjugate gradients (mBCG) that reduces GP training to a series of *matrix-multiplications*. We demonstrate that this approach effectively uses GPU acceleration and is up to $30\times$ faster than existing training methods. Additionally, we show that implementing specialty GP models with BBMM only requires writing an efficient kernel matrix-multiplication routine.
- **Chapter 4** focuses on making predictions with Gaussian processes. We introduce an algorithm—**LancZos Variance Estimates (LOVE)**—that efficiently pre-computes many of the terms required for predictions. As with BBMM training, LOVE relies entirely on *matrix-multiplication*, which is especially beneficial for models with fast kernel routines. After a simple pre-

computation, computing GP predictions is *linear* in the amount of training data, or $\mathcal{O}(1)$ time if used in conjunction with the structured kernel interpolation method [Wilson and Nickisch, 2015].

- **Chapter 5** focuses on Gaussian process models with non-Gaussian likelihood functions—i.e. GPs that are used to model heavy-tailed noise, arrival processes, or classification problems. Unlike with GP regression, these models necessitate the use of approximate Bayesian inference methods. We introduce a matrix-multiplication method based on **Contour Integral Quadrature (CIQ)** which can be used to optimize a re-parameterized variational training objective. On several large-scale spatial datasets, this approach enables faster optimization and higher-fidelity approximations than existing methods. We also demonstrate that this CIQ method can be used to efficiently sample from GP posteriors.
- This thesis culminates with **Chapter 6**, which utilizes the prior chapter's methods to scale GP regression to extremely large datasets. Combining BBMM and LOVE with partitioned matrix-multiplication routines, we demonstrate that Gaussian processes can be trained *without approximation* on datasets with *over 1 million data points*. GPU-acceleration makes these large-scale GPs roughly as fast as approximate methods, despite their larger asymptotic complexity. We perform the first-ever comparison of exact GPs against scalable approximations on datasets with 10^6 data points, showing dramatic performance improvements.

Finally, we package together these contributions into **GPyTorch**,[2] an open-source implementation of BBMM, LOVE, and CIQ. GPyTorch can be used to

[2] `http://github.com/cornellius-gp/gpytorch`

build small-scale or large-scale GPs with flexible neural-network-like building blocks. Moreover, the package seamlessly integrates with the PyTorch [Paszke et al., 2019], Pyro [Bingham et al., 2019], and BoTorch [Balandat et al., 2019] packages to combine GPs with neural networks, probabilistic models, and blackbox optimizers. Throughout this thesis, we will discuss how the various algorithms (BBMM, LOVE, and CIQ) are implemented in GPyTorch, and how a practitioner can build on top of them to develop novel GP models.

We begin with a brief overview of Gaussian processes, common kernel functions, and scalable GP approximations. Additionally, we introduce Krylov-subspace methods—a family of numerical algorithms that compute matrix functions through matrix-vector products—which form the foundation of our GP framework.

CHAPTER 2

BACKGROUND

For a majority of this thesis we will assume that we are building predictive models for supervised regression problems. As a running example, imagine that we wish to predict a child's future weight $y \in \mathbb{R}$ from some features about the child $\mathbf{x} \in \mathbb{R}^D$—such as the their current weight, their parents' income level, etc. We assume that the child's height y can be explained by 1) some latent function $f(\mathbf{x})$ of the features and 2) some observational noise ϵ:

$$y = f(\mathbf{x}) + \epsilon, \quad \epsilon \sim \mathcal{N}\left[0, \sigma^2_{\text{obs}}\right]. \tag{2.1}$$

In particular, we assume the observational noise follows some Gaussian distribution with variance σ^2_{obs}. This setup is one of the most studied problems in machine learning, and there are numerous different approaches to tackling this problem. In this thesis we concern ourselves with the approach of **Gaussian process regression**, which is a **non-parametric** and **Bayesian** approach. It is non-parametric because the class of possible functions $f(\cdot)$ is defined directly through pairs of training data points rather than through some prescribed functional form (e.g. quadratic functions, neural networks, etc.). It is Bayesian because the choice of $f(\cdot)$ is marginalized out.

2.1 Gaussian Process Regression

In the machine learning literature the term *Gaussian process* describes two different objects. In one context it refers to a distribution over functions that can define a prior over $f(\cdot)$. It can also refer the class of predictive models that make use of Gaussian process priors on $f(\cdot)$. As we will soon see, these two classes of

objects are effectively the same. However, for clarity in this section we will explicitly distinguish when "Gaussian process" is describing a prior distribution over functions and when it is describing a class of predictive models.

2.1.1 Gaussian Process Distributions

A **Gaussian process distribution** $f(\cdot) \sim \mathcal{GP}$ extends the multivariate-Gaussian distribution from finite-dimensional vectors to (infinite-dimensional) functions. It is defined by a **mean function** $\mu(\cdot)$ and a **covariance function** or **kernel function** $k(\cdot, \cdot)$.[1] Given N data points $\mathbf{X} = [\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}] \in \mathbb{R}^{N \times D}$, the distribution over $\mathbf{f} = [f(\mathbf{x}^{(1)}), \ldots, f(\mathbf{x}^{(N)})] \in \mathbb{R}^N$ is a multivariate Normal distribution:

$$p(\mathbf{f}) = \mathcal{N}[\mathbf{f}; \boldsymbol{\mu}_{\mathbf{X}}, \mathbf{K}_{\mathbf{XX}}], \tag{2.2}$$

where $f^{(i)} = f(\mathbf{x}^{(i)})$, $\mu_{\mathbf{X}}^{(i)} = \mu(\mathbf{x}^{(i)})$ and $K_{\mathbf{XX}}^{(i,j)} = k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$. (We will be using the above short-hand notations $\mathbf{f}$, $\boldsymbol{\mu}_{\mathbf{X}}$, and $\mathbf{K}_{\mathbf{XX}}$ throughout the remainder of this thesis.) The matrix $\mathbf{K}_{\mathbf{XX}} \in \mathbb{R}^{N \times N}$ is often referred to as the **kernel matrix** of $\mathbf{X}$.

$\mu(\cdot)$ can be any real-valued function, though it is common simply to choose the zero-function (i.e. $\mu(\cdot) = 0$). The covariance function $k(\cdot, \cdot)$ must be a valid kernel function, which means that the kernel matrix $\mathbf{K}$ must be positive definite. See Section 2.1.4 for common choices of $k(\cdot, \cdot)$.

[1] Throughout this thesis we will use the two terms interchangeably.

2.1.2 Gaussian Process Regression Models

Recall our high-level regression model from Eq. (2.1): $y = f(\mathbf{x}) + \epsilon$. **Gaussian process regression models** are a class of predictive models where

(1) $f(\cdot)$ is modelled by a Gaussian process prior: $f(\cdot) \sim \mathcal{GP}[\mu(\cdot), k(\cdot,\cdot)]$, and

(2) ϵ is modelled by a Gaussian noise distribution: $\epsilon \sim \mathcal{N}[0, \sigma^2_{\text{obs}}]$.

Together, these two items define the prior distribution and the likelihood, respectively. To make predictions on previously-unseen test points $\mathbf{x}^*, y^*$ (e.g. predict the future weight of a child), we *condition* the prior model on a set of previously-seen training data $\mathbf{X}, \mathbf{y}$ (e.g. the weights/features of other children). In total the predictive model is fully defined by:

(1) the mean function $\mu(\cdot)$ and covariance function $k(\cdot,\cdot)$ of the GP prior,

(2) the amount of observational noise σ^2_{obs}, and

(3) training data $\mathcal{D}_{\text{train}} = (\mathbf{X}, \mathbf{y})$.

Note that the only learnable parameters of this model are σ^2_{obs} and whatever parameters are required by the mean/covariance functions.

The predictive distribution. In many supervised regression paradigms (e.g. neural networks, ridge regression, etc.), it is common to learn a single latent function $f^*(\cdot)$ that best fits the training data $\mathcal{D}_{\text{train}}$. Under such a setup, the predictive distribution for a test point $(\mathbf{x}^*, y^*)$ is given by $p(y^*|f^*(\mathbf{x}^*)) = \mathcal{N}[y^*; f^*(\mathbf{x}^*), \sigma^2_{\text{obs}}]$. However, Gaussian process regression models *marginalize out*

the choice of $f(\cdot)$. Under this setup, the predictive distribution is given by:

$$p\left(y^* \mid \mathbf{x}^*, \mathcal{D}_{\text{train}}\right) = \int_{f(\cdot)} p\left(y \mid f(\mathbf{x}^*)\right)\; p\left(f(\mathbf{x}^*) \mid \mathbf{x}^*, \mathcal{D}_{\text{train}}\right)\; \mathrm{d}f(\cdot).$$

This predictive distribution happens to be computable in closed form. Given the Gaussian process prior on $f(\cdot)$ and the Gaussian noise observation model for $p(y^*|f(\mathbf{x}^*))$, the prediction $p(y^*|\mathbf{x}^*, \mathcal{D}_{\text{train}})$ is a Gaussian distribution:

$$p\left(y^* \mid \mathbf{x}^*, \mathcal{D}_{\text{train}}\right) = \mathcal{N}\left[y^*; \mu^*\left(\mathbf{x}^*\right), \text{Var}^*\left(\mathbf{x}^*\right)\right] \tag{2.3}$$

where $\mu^*\left(\cdot\right)$ and $\text{Var}^*\left(\cdot\right)$ are given by:

$$\mu^*\left(\mathbf{x}^*\right) = \mu\left(\mathbf{x}^*\right) + \mathbf{k}_{\mathbf{X}\mathbf{x}^*}^\top \left(\mathbf{K}_{\mathbf{X}\mathbf{X}} + \sigma_{\text{obs}}^2 \mathbf{I}\right)^{-1} \left(\mathbf{y} - \boldsymbol{\mu}\right) \tag{2.4}$$

$$\text{Var}^*\left(\mathbf{x}^*\right) = k(\mathbf{x}^*, \mathbf{x}^*) + \sigma_{\text{obs}}^2 - \mathbf{k}_{\mathbf{X}\mathbf{x}^*}^\top \left(\mathbf{K}_{\mathbf{X}\mathbf{X}} + \sigma_{\text{obs}}^2 \mathbf{I}\right)^{-1} \mathbf{k}_{\mathbf{X}\mathbf{x}^*}. \tag{2.5}$$

Here the shorthand $\mathbf{k}_{\mathbf{X}\mathbf{x}^*} \in \mathbb{R}^N$ is the vector of covariances between test point $\mathbf{x}^*$ and all training points $\mathbf{X} = [\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}]$. Note that, under Eqs. (2.4) and (2.5), the Gaussian process' prediction depends on **(1)** the labels of the training data **(2)** the similarities between $\mathbf{x}*$ and other training data (as determined by their prior covariances $\mathbf{k}_{\mathbf{X}\mathbf{x}^*}$).

Short derivation of the predictive distribution. To understand where Eq. (2.4) and Eq. (2.5) come from, we start by writing the joint prior distribution $p[\mathbf{f}, f(\mathbf{x}^*)]$. Under the Gaussian process prior of $f(\cdot)$ and Eq. (2.2), this joint distribution will be a multivariate-Gaussian.

$$p\left(\begin{bmatrix}\mathbf{f} \\ f(\mathbf{x}^*)\end{bmatrix} \middle| \middle| \begin{bmatrix}\mathbf{X} \\ \mathbf{x}^*\end{bmatrix}\right) = \mathcal{N}\left[\begin{bmatrix}\mathbf{f} \\ f(\mathbf{x}^*)\end{bmatrix}; \begin{bmatrix}\boldsymbol{\mu} \\ \mu(\mathbf{x}^*)\end{bmatrix}, \begin{bmatrix}\mathbf{K}_{\mathbf{X}\mathbf{X}} & \mathbf{k}_{\mathbf{X}\mathbf{x}^*} \\ \mathbf{k}_{\mathbf{X}\mathbf{x}^*}^\top & k(\mathbf{x}^*, \mathbf{x}^*)\end{bmatrix}\right].$$

We then compute the joint marginal likelihood $p([\mathbf{y}, y^*] \mid [\mathbf{X}, \mathbf{x}^*])$ by integrating out the dependence on $p([\mathbf{f}, f(\mathbf{x}^*)])$. Under the Gaussian noise observation

model of Eq. (2.1), it happens that the marginal likelihood is also multivariate Gaussian and can be computed in closed form.

$$p\left(\begin{bmatrix}\mathbf{y}\\ y^*\end{bmatrix}\,\middle|\,\begin{bmatrix}\mathbf{X}\\ \mathbf{x}^*\end{bmatrix}\right) = \int p\left(\begin{bmatrix}\mathbf{y}\\ y^*\end{bmatrix}\,\middle|\,\begin{bmatrix}\mathbf{f}\\ f(\mathbf{x}^*)\end{bmatrix}\right) p\left(\begin{bmatrix}\mathbf{f}\\ f(\mathbf{x}^*)\end{bmatrix}\,\middle|\,\begin{bmatrix}\mathbf{X}\\ \mathbf{x}^*\end{bmatrix}\right) \mathrm{d}\begin{bmatrix}\mathbf{f}\\ f(\mathbf{x}^*)\end{bmatrix}$$
$$= \mathcal{N}\left(\begin{bmatrix}\mathbf{y}\\ y^*\end{bmatrix}; \begin{bmatrix}\boldsymbol{\mu}\\ \mu(\mathbf{x}^*)\end{bmatrix}, \begin{bmatrix}\mathbf{K}_{\mathbf{XX}} + \sigma^2_{\text{obs}}\mathbf{I} & \mathbf{k}_{\mathbf{X}\mathbf{x}^*}\\ \mathbf{k}_{\mathbf{X}\mathbf{x}^*}^\top & k(\mathbf{x}^*,\mathbf{x}^*) + \sigma^2_{\text{obs}}\end{bmatrix}\right).$$

where the σ^2_{obs} terms come from the observational noise (which we assume is independent for all data points). Therefore, the predictive distribution $p(y^* \mid \mathbf{x}^*, \mathcal{D}_{\text{train}})$ is simply the conditional of a multivariate Gaussian. Applying standard Gaussian conditioning rules [see e.g. Bishop, 2006, Rasmussen and Williams, 2006] results in Eq. (2.4) and Eq. (2.5).

Predictions on multiple test points. If there are multiple test points $\mathbf{x}^*$, $\mathbf{x}^{*\prime}$ of interest, the predictions $p([y^*, y^{*\prime}] \mid [\mathbf{x}^*, \mathbf{x}^{*\prime}], \mathcal{D}_{\text{train}})$ are jointly Gaussian distributed. The mean is given by Eq. (2.4) and the covariance between the two points is given as

$$\text{Cov}^*\left(\mathbf{x}^*, \mathbf{x}^{*\prime}\right) = k(\mathbf{x}^*, \mathbf{x}^{*\prime}) - \mathbf{k}_{\mathbf{X}\mathbf{x}^*}^\top (\mathbf{K}_{\mathbf{XX}} + \sigma^2_{\text{obs}}\mathbf{I})^{-1}\mathbf{k}_{\mathbf{X}\mathbf{x}^{*\prime}}.$$

In fact, any (finite) set of test points will be jointly Gaussian with the above covariance. Recalling our definition from Section 2.1.1, this implies that the predictive distribution is a Gaussian process distribution.

2.1.3 Training Gaussian Process Models

Gaussian process regression models are non-parametric, which leaves us few terms that need to be learned. However, we wish to pick the mean func-

tion, kernel function, and observational noise that best fit our training data $\mathcal{D}_{\text{train}} = (\mathbf{X}, \mathbf{y})$. Let $\boldsymbol{\theta}$ be the set of learnable **hyperparameters** of these functions. For example, $\boldsymbol{\theta}$ might include the lengthscale ℓ and outputscale o^2 of the kernel function (see Section 2.1.4) as well as the observational noise parameter σ_{obs} of the likelihood. In more complex Gaussian process models, $\boldsymbol{\theta}$ may also include inducing point locations [Titsias, 2009] or neural network parameters for deep kernel learning [Wilson et al., 2016a]. We can measure the fit of these parameters through the **marginal log likelihood** of the Gaussian process applied to the training data $\mathbf{X}, \mathbf{y}$:

$$\begin{aligned} -\log p(\mathbf{y} \mid \mathbf{X}, \boldsymbol{\theta}) &= -\log \mathcal{N}\left[\mathbf{y}; \boldsymbol{\mu}, \widehat{\mathbf{K}}_{\mathbf{XX}}\right] \\ &\propto \log \left|\widehat{\mathbf{K}}_{\mathbf{XX}}\right| + (\mathbf{y}-\boldsymbol{\mu})^\top \widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}(\mathbf{y}-\boldsymbol{\mu}), \end{aligned} \tag{2.6}$$

where $\widehat{\mathbf{K}}_{\mathbf{XX}}$ is shorthand for the training kernel matrix with added observational noise: $\widehat{\mathbf{K}}_{\mathbf{XX}} = \mathbf{K}_{\mathbf{XX}} + \sigma_{\text{obs}}^2 \mathbf{I}$. Note that the dependence on $\boldsymbol{\theta}$ is implicitly absorbed into $\boldsymbol{\mu}$ and $\widehat{\mathbf{K}}_{\mathbf{XX}}$. We can use Eq. (2.6) to learn the parameters $\boldsymbol{\theta}$ via gradient-based optimization or sampling. Its derivative is given by

$$\frac{\partial - \log p(\mathbf{y} \mid \mathbf{X}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \propto \operatorname{Tr}\left(\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1} \frac{\partial \widehat{\mathbf{K}}_{\mathbf{XX}}}{\partial \boldsymbol{\theta}}\right) - (\mathbf{y}-\boldsymbol{\mu})^\top \widehat{\mathbf{K}}_{\mathbf{XX}}^{-1} \frac{\partial \widehat{\mathbf{K}}_{\mathbf{XX}}}{\partial \boldsymbol{\theta}} \widehat{\mathbf{K}}_{\mathbf{XX}}^{-1} (\mathbf{y}-\boldsymbol{\mu}). \tag{2.7}$$

We will discuss how to efficiently compute these terms in Chapter 3.

2.1.4 Common Covariance Functions

Of the three objects that define a Gaussian process regression model—the prior mean function, the prior covariance function, and the observed noise parameter—the covariance function arguably has the largest impact on the predictive distribution. While the parameters of any particular kernel function

can most often be learned through gradient descent, the practitioner must first choose the class of covariance functions to optimize over. Choosing the right covariance function class will largely determine how the Gaussian process regression model generalizes to unseen data points. This is especially crucial in the small-data regime: if the covariance function properly encodes prior information, then the model can generalize well even when there are few training points.

In the running example of predicting a child's weight, there are several known functional properties that we can encode a priori. For example, a child's weight will (in many cases) be roughly monotonic with respect to time. Additionally, children who grow up in similar household settings will be more likely to have similar weights. We will discuss how both of these properties can be encoded by choosing an appropriate kernel function.

Common kernel functions. Here we list some common and simple kernel functions that will be used throughout this thesis. We will describe their functional form and their properties:

- The **Radial Basis Function (RBF)** kernel, or squared exponential kernel is one of the most commonly-used covariance functions. It's functional form is given by

$$k_{\text{RBF}}(\mathbf{x}, \mathbf{x}') = o^2 \exp\left(\sum_{d=1}^{D} \frac{\left(x^{(d)} - x^{(d)\prime}\right)^2}{\ell_d^2}\right)$$

  where o^2 is referred to as the **outputscale** parameter and the $\ell_1, \ldots, \ell_D$ are the **lengthscales** for each dimension $d \in [1, D]$. When the ℓ_d parameters differ across all dimensions, the kernel is referred to as the **RBF-ARD** kernel, where ARD stands for **automatic relevance determination** .

The RBF kernel is commonly considered to be a "general" kernel, since it is able to universally approximate any function given enough training data [Micchelli et al., 2006]. However, functions drawn from $f \sim \mathcal{GP}\left[\mu, k_{\text{RBF}}\right]$ are infinitely differentiable, which may impose an unreasonable smoothness constraint in some applications [Stein, 2012].

- The **Matérn** kernel is another popular kernel which is arguably as general as the RBF kernel yet without the strict smoothness properties. The kernel introduces a parameter ν that determines how differentiable its sampled functions are. More concretely, the covariance between any two points is given as:

$$k_{\text{Mat}}^{(\nu)}(\mathbf{x}, \mathbf{x}') = o^2 \frac{2^{1-\nu}}{\gamma(\nu)} \left(\sqrt{2\nu} \sum_{d=1}^{D} \frac{|x^{(d)} - x^{(d)\prime}|}{\ell_d} \right)^{\nu} K_\nu \left(\sqrt{2\nu} \sum_{d=1}^{D} \frac{|x^{(d)} - x^{(d)\prime}|}{\ell_d} \right),$$

  where o^2 and ℓ_d are analogous to the parameters from the RBF kernel, and where K_ν is a modified Bessel function [Rasmussen and Williams, 2006]. Functions sampled from $f \sim \mathcal{GP}\left[\mu, k_{\text{mat}}^{(\nu)}\right]$ are $\lceil \nu \rceil - 1$-times differentiable. As $\nu \to \infty$ we recover the RBF kernel.

  Typically, o^2 and ℓ_d are learned through gradient descent whereas ν is preselected and fixed. Common values for ν are $3/2$ and $5/2$, both of which have more succinct closed-form expressions. These kernels are often referred to as **Matérn-3/2** and **Matérn-5/2**, respectively.

- The **Linear** kernel is one of the least expressive kernels, as it only has support for linear functions. However, it is commonly used as a building block for more complex kernels, as we will discuss shortly. It is given by:

$$k_{\text{Lin}}(\mathbf{x}, \mathbf{x}') = b^2 + o^2(\mathbf{x} - \mathbf{c})^\top(\mathbf{x} - \mathbf{c}),$$

  where b, $\mathbf{c}$, and o are learnable parameters.

There are many other kernels that can encode other functional structures such as periodicity (using the periodic kernel), multiple lengthscales (using the rational quadratic kernel), or erratic/noisy dynamics (using Ornstein-Uhlenbeck kernel). See [Rasmussen and Williams, 2006] for more discussion on basic kernel types.

Composing kernels. Kernels can be composed together to combine functional properties. The two most common methods for composition are addition: $k(\mathbf{x}, \mathbf{x}') = k_1(\mathbf{x}, \mathbf{x}') + k_2(\mathbf{x}, \mathbf{x}')$ and multiplication: $k(\mathbf{x}, \mathbf{x}') = k_1(\mathbf{x}, \mathbf{x}')k_2(\mathbf{x}, \mathbf{x}')$. The sub-kernels k_1 and k_2 can operate on any subset of the dimensions of $\mathbf{x}$. In our running example, we may wish to model our data with the kernel

$$k(\mathbf{x}, \mathbf{x}') = k_{\text{Lin}}(\mathbf{x}_{\text{time}}, \mathbf{x}'_{\text{time}})k_{\text{RBF}}(\mathbf{x}_{\text{time}}, \mathbf{x}'_{\text{time}}) + k_{\text{RBF}}(\mathbf{x}_{\text{household}}, \mathbf{x}'_{\text{household}})$$

where $\mathbf{x}_{\text{time}}$ and $\mathbf{x}_{\text{household}}$ are the time and household features of $\mathbf{x}$ respectively. This would encode our first prior assumption (a child's weight increases near-monotonically with time, but is not strictly linear) and our second prior assumption (children from a similar household environments should have similar development).

Stationary kernels. The RBF and Matérn kernels belong to an important class of kernels known as **stationary kernels**. This class more generally includes all kernels that can be written as function of the difference between two points:

$$k_{\text{stationary}}(\mathbf{x}, \mathbf{x}') = g(|\mathbf{x} - \mathbf{x}'|), \tag{2.8}$$

for some function g.[2] This class of kernels has many important properties; see [e.g. Rasmussen and Williams, 2006, Wilson and Adams, 2013] for more details. For the purpose of this thesis, we draw attention to stationary kernels as they allow for potential computational savings. For example, the computational complexity of GPs with stationary kernels is reduced when the data lie on a regularly-spaced grid [Cunningham et al., 2008, Saatçi, 2012] or can be approximated by grid points [Wilson, 2014, Wilson and Nickisch, 2015].

Spectral and Deep kernels. With complex datasets, it may be too difficult to manually encode a specific functional structure. In such settings, it is better to automatically learn functional structure directly from the data using a highly-parametric kernel, such as the **spectral-mixture** kernel [Wilson and Adams, 2013] or a **deep kernel** [Wilson et al., 2016a]. At a high level, these kernels learn the g function in Eq. (2.8) using a mixture model or a neural network.

The spectral-mixture kernel (in one dimension) is defined by a mixture of Q functions:

$$k_{\text{SM}}(x, x') = \sum_{q=1}^{Q} w_q \exp\left(-2\pi^2(x - x')^2 v_q\right) \cos\left(-2\pi(x - x')^2 \mu_q\right),$$

where w_q, v_q, and μ_q are learnable parameters. It is well-suited for interpolation and extrapolation tasks of time series or spatial data [Wilson and Adams, 2013]. Deep kernels [Wilson et al., 2016a,b, Calandra et al., 2016] are kernels where the inputs $\mathbf{x}$, $\mathbf{x}'$ are first transformed using a deep neural network Φ:

$$k_{\text{deep}}(x, x') = k_{\text{base}}\left(\Phi(\mathbf{x}), \Phi(\mathbf{x}')\right),$$

[2] Note that the linear kernel cannot be written in this way and is therefore non-stationary.

where $k_{\text{base}}(\cdot, \cdot)$ is a simpler base kernel (usually the RBF kernel). For both cases, the parameters of the mixture model/neural network are learned by optimizing the marginal log likelihood in Eq. (2.6).

2.1.5 Scalable Gaussian Processes

As we will discuss in detail in the next chapters, computing the marginal log likelihood (Eq. (2.6)) and the predictive distribution (Eqs. (2.4) and (2.5)) has historically been computationally prohibitive for large datasets. This motivates the development of several approximate Gaussian process models [e.g. Quiñonero-Candela and Rasmussen, 2005, Snelson and Ghahramani, 2006, Rahimi and Recht, 2008, Titsias, 2009, Hensman et al., 2013, Wilson and Nickisch, 2015, Izmailov et al., 2018a, Gardner et al., 2018b, Evans and Nair, 2018] that trade off exactness for computational scalability. Many of these methods approximate the training kernel matrix $\mathbf{K_{XX}}$ with a structured or low-rank matrix $\widetilde{\mathbf{K}}_{\mathbf{XX}}$ that affords faster matrix operations and requires less storage. Here we highlight some notable variants, though it is by no means an exhaustive list—see [Liu et al., 2020] for a more thorough review.

Inducing Points. A common mechanism for scalable models is to introduce a set of pseudo-inputs, or **inducing points**. The kernel function for (non-inducing) inputs $\mathbf{x}$, $\mathbf{x}'$ is approximated by interpolating against the kernel matrix for the pseudo-inputs $\mathbf{Z} = [\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(M)}]$.

$$
\begin{aligned}
k(\mathbf{x}, \mathbf{x}') \approx \widetilde{k}(\mathbf{x}, \mathbf{x}') &= \overbrace{\left(\mathbf{k}_{\mathbf{Zx}}^\top \mathbf{K}_{\mathbf{ZZ}}^{-1}\right)}^{\text{interpolation}} \mathbf{K_{ZZ}} \overbrace{\left(\mathbf{K}_{\mathbf{ZZ}}^{-1} \mathbf{k}_{\mathbf{Zx}'}\right)}^{\text{interpolation}} \\
&= \mathbf{k}_{\mathbf{Zx}}^\top \mathbf{K}_{\mathbf{ZZ}}^{-1} \mathbf{k}_{\mathbf{Zx}'}. \qquad (2.9)
\end{aligned}
$$

In its simplest form, it is common to choose $M \ll N$ inducing points. The approximate training kernel matrix $\widetilde{\mathbf{K}}_{\mathbf{XX}} \approx \mathbf{K}_{\mathbf{XZ}}\mathbf{K}_{\mathbf{ZZ}}^{-1}\mathbf{K}_{\mathbf{ZX}}$ will then have low-rank structure, allowing for efficient computation of Eqs. (2.4) to (2.7). The locations of the inducing points $\mathbf{Z}$ are additional parameters to optimize via Eq. (2.6) (typically with gradient descent).

Here we have presented inducing points as a mechanism for approximating kernel matrices. Under this view, we can use the same training/prediction equations as presented in Sections 2.1.2 and 2.1.3—replacing the kernel matrices with their corresponding approximations. We do note it is also possible to motivate/derive inducing point methods in other ways—e.g. through a greedy approximation [Smola and Bartlett, 2001] or a variational bound [Titsias, 2009, Hensman et al., 2013]. However, under these interpretations it is less straightforward to derive connections to Eqs. (2.4) to (2.7).

Variants. The basic formula of Eq. (2.9) can be extended to create a variety of approximate Gaussian process regression methods. We briefly present three common variants here—all of which use $M \ll N$ inducing points and learn their locations through gradient descent of Eq. (2.6). In the next section we will discuss Kernel Interpolation for Scalable Structured GPs (KISS-GP) [Wilson and Nickisch, 2015]—a variant which utilizes inducing points in a unique way.

- **Subset of Regressors (SoR)** [Silverman, 1985, Smola and Bartlett, 2001] is arguably the most standard variant. Here, the kernel matrix $\mathbf{K}_{\mathbf{XX}}$ and all kernel vectors $\mathbf{k}_{\mathbf{Xx}^*}$ are replaced with their approximate variants:

$$\widetilde{\mathbf{K}}_{\mathbf{XX}} \approx \mathbf{K}_{\mathbf{XZ}}\mathbf{K}_{\mathbf{ZZ}}^{-1}\mathbf{K}_{\mathbf{ZX}}, \quad \widetilde{\mathbf{k}}_{\mathbf{Xx}^*} \approx \mathbf{K}_{\mathbf{XZ}}\mathbf{K}_{\mathbf{ZZ}}^{-1}\mathbf{k}_{\mathbf{Xx}^*}.$$

- **Fully Independent Training Conditional (FITC)** [Snelson and Ghahramani, 2006] can be interpreted as an extension of SoR where the approximate training kernel matrix has a diagonal correction term $\mathbf{\Lambda}$.

$$\widetilde{\mathbf{K}}_{\mathbf{XX}} \approx \mathbf{K}_{\mathbf{XZ}}\mathbf{K}_{\mathbf{ZZ}}^{-1}\mathbf{K}_{\mathbf{ZX}} + \mathbf{\Lambda}, \quad \Lambda^{(i,i)} = k(\mathbf{x}^{(i)}, \mathbf{x}^{(i)}) - \mathbf{k}_{\mathbf{Z}\mathbf{x}^{(i)}}^{\top}\mathbf{K}_{\mathbf{ZZ}}^{-1}\mathbf{k}_{\mathbf{Z}\mathbf{x}^{(i)}}.$$

  The diagonal correction term is motivated through a connection to approximate Bayesian inference.

- **Sparse Gaussian Process Regression (SGPR)** [Titsias, 2009] uses the same kernel approximation as SoR. However, when learning the inducing point locations and other parameters, SGPR augments the marginal log likelihood of Eq. (2.6) with a diagonal correction term:

$$\mathcal{L}(\mathbf{y} \mid \mathbf{X}, \boldsymbol{\theta}) = -\log p(\mathbf{y} \mid \mathbf{X}, \boldsymbol{\theta}) + \sum_{i=1}^{N} \left(k(\mathbf{x}^{(i)}, \mathbf{x}^{(i)}) - \mathbf{k}_{\mathbf{Z}\mathbf{x}^{(i)}}^{\top}\mathbf{K}_{\mathbf{ZZ}}^{-1}\mathbf{k}_{\mathbf{Z}\mathbf{x}^{(i)}} \right).$$

  The diagonal correction term can be derived through a connection to variational methods.

We will discuss how to exploit this low-rank kernel matrix structure for faster inference in the next chapter.

2.1.6 Kernel Interpolation for Scalable Structured Gaussian Processes (KISS-GP)

Kernel Interpolation for Scalable Structured Gaussian Processes (KISS-GP) [Wilson and Nickisch, 2015] utilizes inducing points in a fundamentally different way than SoR, FITC, or SGPR. While the above methods tend to use $M \ll N$

inducing points, Wilson and Nickisch [2015] demonstrate that it is possible to set $M \gg N$ by making the following restrictions:

(1) the interpolation terms in Eq. (2.9) are approximated by *sparse interpolation vectors* $\mathbf{w_x} \approx \mathbf{K_{ZZ}^{-1}} \mathbf{k_{Zx}}$;

(2) the inducing points $\mathbf{Z} = [\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(M)}]$ must lie on a *regularly-spaced grid*; and

(3) the kernel function is *stationary* (e.g. RBF, Matérn, etc.).

The resulting approximate kernel matrix has nice algebraic structure that can be exploited for efficient storage and fast computations. In particular, the approximate kernel matrix $\widetilde{\mathbf{K}}_{\mathbf{XX}}$ affords fast *matrix-vector multiplications* (MVMs), which will translate to fast Gaussian process training and inference (using the methods described in Chapters 3 and 4).

In more detail, KISS-GP assumes that a data point $\mathbf{x}$ is well-approximated as a *local interpolation* of $\mathbf{Z}$. Using cubic interpolation [Keys, 1981], $\mathbf{x}$ is expressed in terms of its 4 closest inducing points, and the interpolation weights are captured in the sparse vector $\mathbf{w_x}$. The $\mathbf{w_x}$ vectors approximate the training kernel matrix $\mathbf{K_{XX}} \approx \tilde{\mathbf{K}}_{\mathbf{XX}}$ via Eq. (2.9):

$$\begin{aligned} \widetilde{\mathbf{K}}_{\mathbf{XX}} &= \left(\overbrace{\mathbf{K_{ZX}^\top K_{ZZ}^{-1}}}^{\approx \mathbf{W_X^\top}} \right) \mathbf{K_{ZZ}} \left(\overbrace{\mathbf{K_{ZZ}^{-1} K_{ZX}}}^{\approx \mathbf{W_X}} \right) \\ &\approx \mathbf{W_X^\top}\, \mathbf{K_{ZZ}}\, \mathbf{W_X}. \end{aligned} \tag{2.10}$$

Here, $\mathbf{W_X} = [\mathbf{w_{x^{(1)}}}, \ldots, \mathbf{w_{x^{(N)}}}]$ contains the interpolation vectors for all $\mathbf{x}^{(i)}$. Performing a matrix-vector multiplication with $\widetilde{\mathbf{K}}_{\mathbf{XX}}$ (i.e. $(\mathbf{W_X^\top K_{ZZ} W_X})\mathbf{b}$ for any vector $\mathbf{b}$) takes *near-linear* time. To see why, a MVM with the matrix

$\mathbf{W_X}$ requires $\mathcal{O}(N)$ time due to the $\mathcal{O}(N)$ sparsity of $\mathbf{W_X}$. Moreover, restrictions 2 and 3 (stationary kernel/grided inducing points) endow the inducing kernel matrix $\mathbf{K_{ZZ}}$ with Toeplitz structure, which can be exploited for $\mathcal{O}(M)$ storage and $\mathcal{O}(M \log M)$ MVMs [see Wilson et al., 2015, for details]. Thus, if we perform $(\mathbf{W_X^\top K_{ZZ} W_X})\mathbf{b}$ from right-to-left, the entire multiplication takes $\mathcal{O}(N + M \log M)$ time. As we will demonstrate in the next section, exploiting these fasts MVMs results in a near-linear training time complexity for KISS-GP.

Computing predictive means. One advantage of KISS-GP's structure is the ability to perform constant time predictive mean calculations [Wilson et al., 2015]. Substituting the KISS-GP approximate kernel into Eq. (2.4) and assuming a prior mean of 0 for notational brevity, the predictive mean is given by

$$\mu^*(\mathbf{x}^*) = \mathbf{w}_{\mathbf{x}^*}^\top \underbrace{\mathbf{K_{ZZ}} \mathbf{W_X} (\mathbf{W_X^\top K_{ZZ} W_X} + \sigma_{\text{obs}}^2 \mathbf{I})^{-1} \mathbf{y}}_{\mathbf{a}'}. \tag{2.11}$$

(Here blue highlights computations that don't depend on test data.) Because $\mathbf{w}_{\mathbf{x}^*}$ is the only term in Eq. (2.11) that depends on $\mathbf{x}^*$, the remainder of the equation (denoted as $\mathbf{a}'$) can be pre-computed: $\mu^*(\mathbf{x}^*) = \mathbf{w}_{\mathbf{x}^*}^\top \mathbf{a}'$. After pre-computing $\mathbf{a}'$, the multiplication $\mathbf{w}_{\mathbf{x}^*}^\top \mathbf{a}'$ requires $\mathcal{O}(1)$ time, as $\mathbf{w}_{\mathbf{x}^*}$ is sparse and has only four nonzero elements.

2.1.7 Stochastic Variational Inference for Non-Conjugate/Large-Scale GPs

For most of this thesis we will focus on Gaussian process regression with a (conjugate) Gaussian observation model—$p(y \mid f(x)) = \mathcal{N}[y; f(x), \sigma_{\text{obs}}^2]$. Under this

setting, exact Bayesian inference is tractable, which results in the closed-form predictive distribution given by Eqs. (2.4) and (2.5). Here we will briefly examine the case where exact inference is intractable, which occurs when

(1) using a non-Gaussian observation model (e.g. a Bernioulli likelihood for GP classification, a Student-T likelihood for heavy-tailed regression); or

(2) using a dataset that is too large (or cumbersome) to fit into memory.

These settings necessitate *approximate Bayesian inference methods* to estimate the predictive distribution. Here, we would note a subtle but important distinction between "approximate inference" and the "scalable approximations" presented in Section 2.1.5. Scalable GP approximations like KISS-GP perform exact GP inference but approximate the kernel matrix operations.[3] Approximate inference GPs on the other hand approximate the intractable predictive distribution with a simple (often Gaussian) distribution.

There are several approaches to approximate non-conjugate GP predictive distributions [Minka, 2001, Rasmussen and Williams, 2006, Hensman et al., 2015a, Li et al., 2015]. Here we introduce a family of methods that rely on stochastic variational approximations.

Stochastic variational Gaussian processes (SVGP) [Hensman et al., 2013, 2015b, Matthews et al., 2016] are a popular class of models that approximate GP

[3]We note that the SGPR method [Titsias, 2009] is in some sense a "hybrid" method. It is motivated and derived via variational inference, and therefore can be considered an approximate inference method. However, as demonstrated in Section 2.1.5, it can also be viewed through the lens of exact GP inference with a SoR approximated kernel. In this thesis we consider it under the lens of "scalable exact GPs." We reserve the "approximate inference" label for models that can handle non-conjugate observation models.

posteriors via stochastic variational inference. This method has several compelling properties: **(1)** it uses $M \ll N$ inducing points to easily trade off speed for predictive fidelity; and **(2)** it can be used in conjunction with stochastic gradient optimization for memory savings.

Similarly to SoR/FITC/SGPR, these models introduce a set of $M \ll N$ inducing points $\mathbf{Z} = [\mathbf{z}_1, \ \ldots, \ \mathbf{z}_M]$. Let the random variable $\mathbf{u} = [f(\mathbf{z}_1, \ \ldots, \ f(\mathbf{z}_M)]$ represent the function evaluated at each inducing point. SGPR approximates the joint posterior distribution $p(f(\cdot), \mathbf{u}, \mid \mathbf{X}, \mathbf{y})$ with a Gaussian process *variational distribution* $q(f(\cdot), \mathbf{u})$:

$$\begin{aligned} p(f(\cdot), \mathbf{u} \mid \mathbf{X}, \mathbf{y}) &\approx q(f(\cdot), \mathbf{u}) \\ &= p(f(\cdot) \mid \mathbf{u}) q(\mathbf{u}), \quad q(\mathbf{u}) = \mathcal{N}\left[\mathbf{m}, \mathbf{S}\right], \end{aligned}$$

where $\mathbf{m} \in \mathbb{R}^M$ and $\mathbf{S} \in \mathbb{R}^{M \times M}$ are learnable *variational parameters*. Under this approximation, note that the conditional variational distribution $q(f(\cdot) \mid \mathbf{u})$ matches the prior conditional distribution. Marginalizing out $\mathbf{u}$ gives us the approximate Gaussian predictive distribution $q(f(\mathbf{x}))$:

$$\begin{aligned} q\left(f(\mathbf{x})\right) &= \mathop{\mathbb{E}}_{q(\mathbf{u})}\left[q(f(\mathbf{x}), \mathbf{u})\right] \triangleq \mathop{\mathbb{E}}_{q(\mathbf{u})}\left[\, p\left(f(\mathbf{x}) \mid \mathbf{u}\right) \right] \\ &= \mathcal{N}\left[\mathbf{x}^*; \mu^*_{\text{aprx}}\left(\mathbf{x}\right), \text{Var}^*_{\text{aprx}}\left(\mathbf{x}\right)\right] \\ \mu^*_{\text{aprx}}\left(\mathbf{x}\right) &= \mathbf{k}_{\mathbf{Zx}}^\top \mathbf{K}_{\mathbf{ZZ}}^{-1} \mathbf{m} \qquad (2.12) \\ \text{Var}^*_{\text{aprx}}\left(\mathbf{x}\right) &= k(\mathbf{x}, \mathbf{x}) - \mathbf{k}_{\mathbf{Zx}}^\top \mathbf{K}_{\mathbf{ZZ}}^{-1} \left(\mathbf{K}_{\mathbf{ZZ}} - \mathbf{S}\right) \mathbf{K}_{\mathbf{ZZ}}^{-1} \mathbf{k}_{\mathbf{Zx}} \qquad (2.13) \end{aligned}$$

where $\mu^*_{\text{aprx}}(\cdot)$ and $\text{Var}^*_{\text{aprx}}(\cdot)$ are the (approximate) posterior mean and covariance functions. The parameters $\mathbf{m}$ and $\mathbf{S}$ (as well as the inducing locations $\mathbf{Z}$ and kernel/likelihood hyperparameters) are optimized to minimize the KL-divergence between the approximate posterior and the true posterior $p(f(\cdot), \mathbf{u} \mid \mathbf{X}, \mathbf{y})$. This objective function is captured by the evidence-lower

bound (ELBO):

$$-\mathcal{L}_{\text{ELBO}} = -\sum_{i=1}^{N} \mathop{\mathbb{E}}_{q(f(\mathbf{x}^{(i)}))} \left[\log p(y^{(i)} \mid f(\mathbf{x}^{(i)})) \right] + \text{KL}\left[q(\mathbf{u}) \| p(\mathbf{u}) \right]. \tag{2.14}$$

Here, $p(y^{(i)} \mid f(\mathbf{x}^{(i)})$ is the (potentially) non-conjugate likelihood function. The expectation $\mathbb{E}_{q(f(\mathbf{x}^{(i)}))} \left[\log p(y^{(i)} \mid f(\mathbf{x}^{(i)})) \right]$ can be approximated with Monte Carlo integration [Wilson et al., 2016b] or with Gauss-Hermite quadrature [Hensman et al., 2015b]. The KL divergence term is given by

$$\text{KL}\left[q(\mathbf{u}) \| p(\mathbf{u}) \right] = \frac{1}{2}\Big(\mathbf{m}^\top \mathbf{K}_{\mathbf{ZZ}}^{-1} \mathbf{m} + \text{Tr}\left(\mathbf{K}_{\mathbf{ZZ}}^{-1} \mathbf{S} \right) - \log |\mathbf{K}_{\mathbf{ZZ}}^{-1} \mathbf{S}| - M \Big).$$

Eq. (2.14) is typically optimized using gradient descent or natural gradient descent methods [Hensman et al., 2012, Salimbeni et al., 2018b]. Note that, because the ELBO factorizes as a sum over data points, we can approximate the sum over a minibatch of data. This ability to use stochastic optimization is an advantage of SVGP, as it does not require large datasets to be stored in memory.

To form predictions at test time, we simply use Eq. (2.12) and Eq. (2.13) to compute $q(f(\mathbf{x}^*))$ for a test point $\mathbf{x}^*$. The predictive distribution $q(\mathbf{y} \mid \mathbf{x}^*) = \mathbb{E}_{q(f(\mathbf{x}^*))} \left[p(y^* \mid f(\mathbf{x}^*)) \right]$ can again be approximated via sampling or quadrature. Closed-form predictive distributions $q(y^* \mid \mathbf{x}^*)$ exist for Gaussian and Bernoulli observation models [Rasmussen and Williams, 2006].

2.1.8 Summary of Notation

The notation of this section is summarized in Table 2.1. It will be used throughout the remaining chapters.

Table 2.1: Summary of Gaussian process notation.

Notation	Domain	Description
N	$\mathbb{Z}$	Number of training data points
D	$\mathbb{Z}$	Dimensionality of inputs
M	$\mathbb{Z}$	Number of inducing points
$\mathbf{X}$	$\mathbb{R}^{N\times D}$	Training features
$\mathbf{y}$	$\mathbb{R}^{N}$	Training targets
$\mathbf{x}^*$	$\mathbb{R}^{D}$	Features of a test data point
y^*	$\mathbb{R}$	Target of a test data point
$\mu(\cdot)$	$\mathbb{R}^D \to \mathbb{R}$	The GP (prior) mean function
$k(\cdot,\cdot)$	$\mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$	The GP (prior) covariance/kernel function
σ^2_{obs}	$\mathbb{R}$	Observational variance of the Gaussian likelihood
$f(\cdot)$	$\mathbb{R}^D \to \mathbb{R}$	Latent function (modelled with a GP prior)
$\boldsymbol{\theta}$		Parameters of kernel, likelihood, etc.
$\boldsymbol{\mu}_{\mathbf{X}}$	$\mathbb{R}^{N}$	(Prior) mean vector for training data $\mathbf{X}$
$\mathbf{K}_{\mathbf{XX}}$	$\mathbb{R}^{N\times N}$	(Prior) kernel matrix for training data $\mathbf{X}$
$\widehat{\mathbf{K}}_{\mathbf{XX}}$	$\mathbb{R}^{N\times N}$	(Prior) kernel matrix $\mathbf{K}_{\mathbf{XX}}$ plus observational noise $\sigma^2_{\text{obs}}\mathbf{I}$
$\mathbf{k}_{\mathbf{Xx}^*}$	$\mathbb{R}^{N}$	Prior covariance between training data $\mathbf{X}$ and point $\mathbf{x}^*$
$\mathbf{f}$	$\mathbb{R}^{N}$	Latent function evaluated on training data $\mathbf{X}$
$\mathbf{Z}$	$\mathbb{R}^{M\times D}$	Locations of inducing points
$\mathbf{K}_{\mathbf{ZZ}}$	$\mathbb{R}^{M\times M}$	(Prior) kernel matrix for inducing points $\mathbf{Z}$
$\mathbf{k}_{\mathbf{Zx}}$	$\mathbb{R}^{M}$	Prior covariance between inducing points $\mathbf{Z}$ and point $\mathbf{x}$
$\mathbf{W}_{\mathbf{X}}$	$\mathbb{R}^{N\times M}$	Sparse inducing-interpolation matrix for training data $\mathbf{X}$ (using Eq. (2.10))
$\mathbf{w}_{\mathbf{x}}$	$\mathbb{R}^{M}$	Sparse inducing-interpolation vector for point $\mathbf{x}$ (using Eq. (2.10))
$\widetilde{k}(\cdot,\cdot)$	$\mathbb{R}^N \times \mathbb{R}^N \to \mathbb{R}$	Approximate kernel function (using Eq. (2.9))
$\widetilde{\mathbf{K}}_{\mathbf{XX}}$	$\mathbb{R}^{N\times N}$	Approximate training kernel matrix (using Eq. (2.9))
$\mu^*(\cdot)$	$\mathbb{R}^D \to \mathbb{R}$	Predictive mean function of the GP model
$\text{Var}^*(\cdot)$	$\mathbb{R}^D \to \mathbb{R}$	Predictive variance function of the GP model
$\text{Cov}^*(\cdot,\cdot)$	$\mathbb{R}^D \to \mathbb{R}$	Predictive co-variance function of the GP model

2.2 The Cholesky Factorization and its Pivoted Variant

The Cholesky factorization decomposes a positive definite matrix $\mathbf{K}$ as $\mathbf{L}\mathbf{L}^\top$, where $\mathbf{L}$ is lower triangular. It is typically used to compute matrix solves and determinants of positive definite matrices. Historically, it has been the primary numerical tool for GP training and predictions.

In the remaining chapters we will propose alternative numerical tools for Gaussian processes. However, we introduce the Cholesky algorithm here to better understand its numerical properties and limitations. Additionally, we introduce a pivoted version of the Cholesky factorization, which will be used as a building block for a preconditioner introduced in Section 3.3.

2.2.1 The Cholesky Factorization

The Cholesky decomposition is defined by a recursive algorithm:

$$\begin{bmatrix} \mathbf{K}^{(11)} & \mathbf{K}^{(12)} \\ \mathbf{K}^{(12)^\top} & \mathbf{K}^{(22)} \end{bmatrix} = \begin{bmatrix} \mathbf{L}^{(11)} & \mathbf{0} \\ \mathbf{L}^{(11)^{-1}}\mathbf{K}^{(12)} & \mathbf{L}^{(22)} \end{bmatrix} \begin{bmatrix} \mathbf{L}^{(11)^\top} & \mathbf{K}^{(12)^\top}\mathbf{L}^{(11)^{-\top}} \\ \mathbf{0} & \mathbf{L}^{(22)^\top} \end{bmatrix} \tag{2.15}$$

where $\mathbf{L}^{(11)}$ is the Cholesky factor of $\mathbf{K}^{(11)}$ and $\mathbf{L}^{(22)}$ is the Cholesky factor of the Schur compliment $\mathbf{S}_2 = \left(\mathbf{K}^{(22)} - \mathbf{K}^{(12)^\top}\mathbf{L}^{(11)^{-\top}}\mathbf{L}^{(11)^{-1}}\mathbf{K}^{(12)}\right)$. The base case is for 1×1 matrices: $\text{Chol}(K) = \sqrt{K}$.

Runtime and space. The Cholesky algorithm is inherently sequential, requiring N steps to decompose an $N \times N$ matrix. Each step $i \in [1, N]$ computes the Schur compliment of a $(N - i) \times (N - i)$ matrix, which takes $\mathcal{O}(i^2)$ time (see [Golub and Van Loan, 2012, Sec. 4.2] for a complete derivation). Thus in total

the factorization takes $\mathcal{O}(N^3)$ time. Storing the Cholesky factor requires $\mathcal{O}(N^2)$ space. It is worth noting that in general these computation and storage requirements cannot be improved upon even when $\mathbf{K}$ has exploitable structure (e.g. Toeplitz). Moreover, this sequential nature of the recursive algorithm prevents it from exploiting the full possibilities of GPU acceleration.

2.2.2 An Iterative View of the Cholesky Factorization

The Cholesky factorization is considered a **direct numerical method**. In other words, the recursion defined by Eq. (2.15) is run to completion before computing any solves/determinants. Any computations performed with the Cholesky factor $\mathbf{L}$ are considered to be "exact" up to numerical round-off errors. Alternatively, we can view the Cholesky decomposition as an **iterative method**, where each iteration produces a higher rank approximation to the matrix $\mathbf{K}$. In particular, if $\mathbf{K} = \begin{bmatrix} K^{(11)}, & \mathbf{b}^\top; & \mathbf{b}, & \mathbf{K}^{(22)} \end{bmatrix}$, then $\mathbf{L}^{(11)} = \sqrt{K^{(11)}}$, $\mathbf{L}^{(21)} = \left(1/\sqrt{K^{(11)}}\right)\mathbf{b}$, and the Schur complement is $\mathbf{S}_2 = \mathbf{K}^{(22)} - \left(1/K^{(11)}\right)\mathbf{b}\mathbf{b}^\top$. Therefore:

$$\begin{aligned} \mathbf{K} &= \overbrace{\begin{bmatrix} \sqrt{K^{(11)}} \\ \frac{1}{\sqrt{K^{(11)}}}\mathbf{b} \end{bmatrix}}^{\widehat{\boldsymbol{\ell}}_1} \overbrace{\begin{bmatrix} \sqrt{K^{(11)}} & \frac{1}{\sqrt{K^{(11)}}}\mathbf{b}^\top \end{bmatrix}}^{\widehat{\boldsymbol{\ell}}_1^\top} + \begin{bmatrix} 0 & 0 \\ 0 & \mathbf{S}_2 \end{bmatrix} \\ &\triangleq \widehat{\boldsymbol{\ell}}_1\widehat{\boldsymbol{\ell}}_1^\top + \begin{bmatrix} 0 & 0 \\ 0 & \mathbf{S}_2 \end{bmatrix}. \end{aligned} \tag{2.16}$$

Running this same procedure on $\mathbf{S}_2$ gives us $\mathbf{S}_2 = \widehat{\boldsymbol{\ell}}_2\widehat{\boldsymbol{\ell}}_2^\top + [0, \ 0; \quad 0, \ \mathbf{S}_3]$:

$$\mathbf{K} = \widehat{\boldsymbol{\ell}}_1\widehat{\boldsymbol{\ell}}_1^\top + \begin{bmatrix} 0 \\ \widehat{\boldsymbol{\ell}}_2 \end{bmatrix} \begin{bmatrix} 0 & \widehat{\boldsymbol{\ell}}_2^\top \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & \mathbf{S}_3 \end{bmatrix}$$

After R recursive iterations, defining $\boldsymbol{\ell}_i = \left[\mathbf{0};\ \widehat{\boldsymbol{\ell}}_i\right]$ we obtain

$$\mathbf{K} = \sum_{i=1}^{R} \boldsymbol{\ell}_i \boldsymbol{\ell}_i^\top + \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{S}_{R+1} \end{bmatrix}, \tag{2.17}$$

where each $\boldsymbol{\ell}_i$ has exactly $i-1$ leading zero ($\boldsymbol{\ell}_i = [0,\ \ldots,\ 0,\ \widehat{\boldsymbol{\ell}}_i]$). The $\sum_{i=1}^{R} \boldsymbol{\ell}_i \boldsymbol{\ell}_i^\top$ sum of outer product matrices gives us a low-rank approximation to $\mathbf{K}$, with

$$\left\| \mathbf{K} - \sum_{i=1}^{R} \boldsymbol{\ell}_i \boldsymbol{\ell}_i^\top \right\|_2 = \left\| \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{S}_{R+1} \end{bmatrix} \right\|_2 . \tag{2.18}$$

After all N iterations the sum of the outer products is exact: $\mathbf{K} = \sum_{i=1}^{N} \boldsymbol{\ell}_i \boldsymbol{\ell}_i^\top$.

2.2.3 The Partial Pivoted Cholesky Factorization

To improve the low rank approximation in Eq. (2.17), one natural goal is to minimize the norm of the Schur complement $\|\mathbf{S}_i\|$ at each iteration. Harbrecht et al. [2012] suggest a greedy approach, permuting the rows and columns of $\mathbf{S}_i$ so that the upper-leftmost entry in $\mathbf{S}_i$ is the maximum diagonal element. In the first step (where $\mathbf{S}_1 = \mathbf{K}$), this amounts to replacing $\mathbf{K}$ with $\boldsymbol{\Pi}_1 \mathbf{K} \boldsymbol{\Pi}_1^\top$, where $\boldsymbol{\Pi}_1$ is a permutation matrix that swaps the first row/column with whichever row/column corresponds to the maximum diagonal element. Thus:

$$\boldsymbol{\Pi}_1 \mathbf{K} \boldsymbol{\Pi}_1^\top = \boldsymbol{\ell}_1 \boldsymbol{\ell}_1^\top + \begin{bmatrix} 0 & 0 \\ 0 & \bar{\mathbf{S}}_2 \end{bmatrix}.$$

To proceed, one can apply the same pivoting rule to $\mathbf{S}_2$ to achieve $\boldsymbol{\Pi}_2$:

$$(\boldsymbol{\Pi}_2 \boldsymbol{\Pi}_1)\, \mathbf{K}\, \left(\boldsymbol{\Pi}_1^\top \boldsymbol{\Pi}_2^\top\right) = \boldsymbol{\Pi}_2 \boldsymbol{\ell}_1 \boldsymbol{\ell}_1^\top \boldsymbol{\Pi}_2^\top + \boldsymbol{\ell}_2 \boldsymbol{\ell}_2^\top + \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \bar{\mathbf{S}}_3 \end{bmatrix}.$$

In general, after R steps, we obtain:

$$\mathbf{K} = \sum_{i=1}^{R} \left(\prod_{j=1}^{i} \mathbf{\Pi}_j \right) \boldsymbol{\ell}_i \boldsymbol{\ell}_i \left(\prod_{j=1}^{i} \mathbf{\Pi}_{i-j+1}^{\top} \right) + \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \bar{\mathbf{S}}_{R+1} \end{bmatrix}. \tag{2.19}$$

Collecting the $\left(\prod_{j=1}^{i} \mathbf{\Pi}_j \right) \boldsymbol{\ell}_i$ vectors into a matrix $\bar{\mathbf{L}}_R$ gives us a rank-R approximation of $\mathbf{K}$:

$$\mathbf{K} = \bar{\mathbf{L}}_R \bar{\mathbf{L}}_R^{\top} + \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \bar{\mathbf{S}}_{R+1} \end{bmatrix} \approx \bar{\mathbf{L}}_R \bar{\mathbf{L}}_R^{\top}. \tag{2.20}$$

$\bar{\mathbf{L}}_R \in \mathbb{R}^{N \times R}$ is referred to as the **partial pivoted Cholesky factor** of $\mathbf{K}$.

Properties. For matrices with rapidly decaying spectra, the partial pivoted Cholesky factor is a remarkably effective approximation. Harbrecht et al. [2012] prove an *exponential* convergence guarantee for matrices with exponentially-decaying eigenvalues:

Theorem 2.1 (Partial Pivoted Cholesky Convergence [Harbrecht et al., 2012])**.** *If the first R eigenvalues $\lambda_1, \ldots, \lambda_R$ of a positive-definite $\mathbf{K} \in \mathbb{R}^{N \times N}$ satisfy $(4^i \lambda_i) \leq \mathcal{O}(e^{-Bi})$ for some $B > 0$, then the rank-R pivoted Cholesky decomposition $\bar{\mathbf{L}}_R \bar{\mathbf{L}}_R^{\top}$ gives*

$$\operatorname{Tr}\left(\mathbf{K} - \bar{\mathbf{L}}_R \bar{\mathbf{L}}_R^{\top} \right) \leq \mathcal{O}\left(N e^{-BR} \right).$$

Runtime and space. The pivoted Cholesky algorithm adds two additional steps to each Cholesky iteration: **(1)** computing the Schur compliment's diagonal $\texttt{diag}(\mathbf{S}_{i+1})$ (to determine which elements to pivot), and **(2)** pivoting $\mathbf{S}_{i+1}$ and existing vectors $\boldsymbol{\ell}_1, \ldots, \boldsymbol{\ell}_i$. Each addition is $\mathcal{O}(N)$ extra time (see [Harbrecht et al., 2012, Thm. 1]). Therefore, the rank-R pivoted Cholesky factor is not much different than R iterations of Cholesky, requiring $\mathcal{O}(NR^2)$ time and

$\mathcal{O}(NR)$ space. Importantly, the pivoted Cholesky algorithm can produce low-rank approximations *without explicitly computing* $\mathbf{K}$. This is useful if $\mathbf{K}$ is structured or sparse and requires $o(N^2)$ storage. All that's required is a routine for computing the diagonal of $\mathbf{K}$ and an arbitrary row $\mathbf{a}^{(i)}$. When $\mathbf{K}$ is not explicitly computed, the complexity is $\mathcal{O}(R^2\rho(\mathbf{K}))$, where $\rho(\mathbf{K})$ is the time to compute a row and/or a diagonal (see Algorithm 2.1). For most structured/low-rank matrices it will be the case that $\rho(\mathbf{K}) \approx \mathcal{O}(N)$.

Algorithm 2.1: Partial (rank-R) pivoted Cholesky decomposition.

Input : $\mathbf{d_K}$ – diagonal of $\mathbf{K}$.
`row_K(i)` – function for computing the i^{th} of matrix $\mathbf{K}$.
Output: $\bar{\mathbf{L}}_{\text{R}}$ – the partial (rank-R) pivoted Cholesky factor.

$\boldsymbol{\pi} \leftarrow [1,\ 2,\ \ldots,\ N]$ `// Vector representing permutation matrix` $\boldsymbol{\Pi}_i$.
$\mathbf{c}_0 \leftarrow \mathbf{0}$
$\bar{\mathbf{d}}_{\mathbf{K}} \leftarrow \mathbf{d_K}$ `// Permuted diagonal.`
for $i \leftarrow 1$ **to** R **do**
  `// Find index of largest entry in permutted` $\mathbf{d_K}$.
  $m \leftarrow \arg\max_{j \in [i,N]} \bar{\mathbf{d}}_{\mathbf{K}}^{(\pi(j))}$
  `// Update permutation.`
  $\pi^{(i)}, \pi^{(m)} \leftarrow \pi^{(m)}, \pi^{(i)}$
  `// Get next row (according to permutation).`
  $\mathbf{a}^{(\pi(i))} \leftarrow$ `row_K(` $\pi(i)$`)`
  `// Cholesky iteration (with permuted indices).`
  $\boldsymbol{\ell}^{(i)} \leftarrow \mathbf{0}$
  $\ell^{(i,\pi(i))} \leftarrow \sqrt{\mathbf{d}_{\mathbf{K}}^{(\pi(i))}}$
  `// This inner for-loop can be vectorized for paralleism.`
  for $j \leftarrow (i+1)$ **to** N **do**
    $\ell^{(i,\pi(j))} \leftarrow \left(\mathbf{a}^{(\pi(i),\pi(j))} - \sum_{k=1}^{i-1}\left(\ell^{(k,\pi(i))}\ell^{(k,\pi(j))}\right)\right) / \ell^{(i,\pi(i))}$
    $d^{(\pi(j))} \leftarrow d^{(\pi(j))} - \left(\ell^{(i,\pi(i))}\ell^{(i,\pi(j))}\right)$
  end
end
return $\mathbf{L}_R = \left[\boldsymbol{\ell}^{(1)},\ \ldots,\ \boldsymbol{\ell}^{(R)}\right]$

2.3 Matrix-Vector Multiplication (MVM) Algorithms for Computing Linear Solves and Other Matrix Functions

The primary focus of this thesis is avoiding the Cholesky decomposition for GP learning and inference. Recall that the Cholesky decomposition is primarily used to compute matrix solves and log determinants:

$$\mathbf{L}\mathbf{L}^\top = \widehat{\mathbf{K}}_{\mathbf{XX}}, \quad (\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1})\mathbf{y} = \mathbf{L}^{-\top}\mathbf{L}^{-1}\mathbf{y}, \quad \log|\widehat{\mathbf{K}}_{\mathbf{XX}}| = 2\sum_{i=1}^{N}\log L^{(ii)}$$

We will instead compute $(\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1})\mathbf{y}$ and $\log|\widehat{\mathbf{K}}_{\mathbf{XX}}|$ (and other non-linear terms) through algorithms that only utilize *matrix-vector multiplication* (**MVMs**) and other simple vector operations. This approach, as we will demonstrate, has several advantages: **(1)** it effectively utilizes GPU acceleration, **(2)** it reduces memory requirements, and **(3)** it simplifies the implementation of specialty models. We will rely on a family of iterative algorithms known as **Krylov subspace methods** [e.g. Saad, 2003, Van der Vorst, 2003], which were originally developed for large sparse matrices. Importantly, these algorithms can compute non-linear terms *without explicitly computing the matrix* $\widehat{\mathbf{K}}_{\mathbf{XX}}$. Instead, these methods only access $\widehat{\mathbf{K}}_{\mathbf{XX}}$ through MVMs.

The next three chapters propose GP-specific Krylov subspace algorithms for training and inference. In this remainder of this section, we introduce three common Krylov methods that we use as building blocks: **linear conjugate gradients** (CG), **Lanczos tridiagonalization**, and **MINRES**.

Intuition. At first glance, it may not be obvious how matrix-vector multiplications—a linear operation—can compute non-linear operations like

$\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{b}$ or $\log|\widehat{\mathbf{K}}_{\mathbf{XX}}|$. To motivate the use of MVM-based algorithms, we will begin by reformulating matrix solves and log determinants to make their MVM connection obvious. First, we will view $\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{y}$ through an optimization lens. Since $\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}$ is positive definite, we note that any matrix solve is the solution to the following convex problem:

$$\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{y} = \underset{\mathbf{c}\in\mathbb{R}^N}{\arg\min}\Big[\frac{1}{2}\,\mathbf{c}^\top(\widehat{\mathbf{K}}_{\mathbf{XX}})\,\mathbf{c} - \mathbf{c}^\top\mathbf{y}\Big], \tag{2.21}$$

The gradient of the objective function is $(\widehat{\mathbf{K}}_{\mathbf{XX}})\,\mathbf{c} - \mathbf{y}$. Thus, one could compute $\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{y}$ with gradient descent, and each gradient step only accesses $\widehat{\mathbf{K}}_{\mathbf{XX}}$ through a MVM with the current solution $\mathbf{c}$. Rather than using vanilla gradient descent however, we will instead use a special decent method (conjugate gradients) that carefully chooses its updates for faster convergence.

The connection between log determinants and MVMs is less straightforward. At a high-level, we note that the eigenvalues of $\widehat{\mathbf{K}}_{\mathbf{XX}}$ can be found through MVM iterations. For example, the *power method* computes the largest eigenvector by running the iteration $\mathbf{c}_{j+1} = (\widehat{\mathbf{K}}_{\mathbf{XX}})\mathbf{c}_j$ until convergence.[4] In practice computing a complete eigendecomposition of $\widehat{\mathbf{K}}_{\mathbf{XX}}$ through power iteration would be a highly inefficient way to compute log determinants. We will instead compute these terms through stochastic Lanczos quadrature [Ubaru et al., 2017] which similarly uses MVMs but avoids a full eigendecomposition.

With these intuitions in place, we will now introduce linear conjugate gradients, Lanczos tridiagonalization, and MINRES. These methods expand upon the

[4]Let $\lambda_i, \mathbf{v}_i$ be the i^{th} eigenvalue/vector of $\widehat{\mathbf{K}}_{\mathbf{XX}}$. If we write the initial vector as a linear combination of the eigenvectors $\mathbf{c}_0 = \sum_{i=1}^N w_i\mathbf{v}_i$, then the iterations simply scale the weights w_i by a factor λ_i: $\mathbf{c}_j = \sum_{i=1}^N w_i\lambda_i^j\mathbf{v}_i$. After enough iterations, the largest eigenvalue λ_1^j will dominate all others and so $\mathbf{c}_j \approx w_1\lambda_1^j\mathbf{v}_1$. See [e.g. Golub and Van Loan, 2012, Ch. 8] for details.

simple iterative procedures we outline above, yet converge to the true solutions in many fewer iterations.

2.3.1 Linear Conjugate Gradients

Linear conjugate gradients (CG) [Hestenes et al., 1952] is an iterative method for solving the optimization problem posed in Eq. (2.21) if $\mathbf{K}$ is positive definite. At a high level, it is similar to gradient descent: each iteration updates an existing solution by taking a step in a "search" direction. Unlike gradient descent, a good solution to $\mathbf{K}^{-1}\mathbf{b}$ is often found in $\ll N$ iterations, where N is the size of the matrix $\mathbf{K}$. Moreover, each iteration requires only a single MVM with $\mathbf{K}$.

Here we offer a high level overview of the algorithm and its convergence properties. Because CG is notoriously confusing and difficult to intuit, we refer the reader to Shewchuk [1994] for a gentler introduction.

Overview. Conjugate gradients differs from vanilla gradient descent in two ways: its' choice of the decent direction and its' choice of step size. Standard gradient descent takes a constant step φ in the direction of the gradient $\mathbf{r}_j$ at every iteration:

$$\mathbf{c}_{j+1}^{(GD)} = \mathbf{c}_j - \varphi \mathbf{r}_j, \quad \mathbf{r}_j = \mathbf{K}\mathbf{c}_j - \mathbf{b}.$$

($\mathbf{r}_j$ is also referred to as the "residual" of the current solution.) This approach is inefficient because multiple iterations might take steps in the same direction.

On the other hand, the descent directions taken by conjugate gradients $\mathbf{d}_1, \ldots, \mathbf{d}_J$ are *mutually conjugate* to one another. This means that all search directions are orthogonal with respect to the inner product defined by $\mathbf{K}$—i.e. for

all $j \neq j'$, we have $\langle \mathbf{d}_j, \mathbf{d}_{j'} \rangle_{\mathbf{K}} = \mathbf{d}_j^\top \mathbf{K} \mathbf{d}_{j'} = 0$. A consequence of this is that no two iterations move the solution $\mathbf{c}_{j+1}$ in the same direction. Each iteration takes a variable step size α_j that corresponds to the optimal step in the $\mathbf{d}_j$ direction. The resulting iterations are:

$$\mathbf{c}_{j+1}^{(CG)} = \mathbf{c}_j - \alpha_j \mathbf{d}_j.$$

where $\mathbf{d}_j$ and α_j are determined using formulas described below. The N conjugate descent vectors $\mathbf{d}_1, \ldots, \mathbf{d}_N$ form a basis of $\mathbb{R}^N$, and therefore we can describe the solution as $\mathbf{K}^{-1}\mathbf{b} = \mathbf{c}_0 - \sum_{j=1}^{N} \alpha_j \mathbf{d}_j$. In other words, these iterations find the true solution $\mathbf{K}^{-1}\mathbf{b}$ in at most N iterations, though often $\ll N$ iterations finds a very close approximation.

The algorithm. By carefully choosing the conjugate descent directions $\mathbf{d}_1, \ldots, \mathbf{d}_N$, the CG iterations can be computed through an extremely efficient recurrence relation. At iteration j, the descent direction $\mathbf{d}_j$ is given by taking the residual $\mathbf{r}_j = \mathbf{K}\mathbf{c}_j - \mathbf{b}$ and making it conjugate to all previous descent directions:

$$\mathbf{d}_j = \mathbf{r}_j - \sum_{\ell=1}^{j-1} \frac{\mathbf{d}_\ell^\top \mathbf{K} \mathbf{r}_j}{\mathbf{d}_\ell^\top \mathbf{K} \mathbf{d}_\ell} \mathbf{d}_\ell.$$

(The above equation is very similar to Gram-Schmidt orthogonalization, except that the resulting vectors are *conjugate* rather than *orthogonal*.) In practice, the sum reduces down to a single term $(\mathbf{d}_{j-1}^\top \mathbf{K} \mathbf{r}_j)/(\mathbf{d}_{j-1}^\top \mathbf{K} \mathbf{d}_{j-1})\mathbf{d}_{j-1}$ since the residual already happens to be conjugate to all other search directions (see [Shewchuk, 1994, Sec. 7-8] for a complete derivation). Moreover, the residual $\mathbf{r}_j$ and the step size α_j can be computed through inner products of existing terms. As a result, each iteration of CG requires only a single MVM ($\mathbf{K}\mathbf{d}_{j-1}$) and a few additional vector inner products. The entire algorithm is thus a recurrence re-

Algorithm 2.2: Standard conjugate gradients (CG).

Input : `mvm_K(·)` – function for matrix-vector multiplication (MVM) with matrix $\mathbf{K}$
$\mathbf{b}$ – vector to solve against
Output: $\mathbf{c} = \mathbf{K}^{-1}\mathbf{b}$.

$\mathbf{c}_0 \leftarrow \mathbf{0}$ `// Current solution`
$\mathbf{r}_0 \leftarrow$ `mvm_K(`$\mathbf{c}_0$`)` - $\mathbf{b}$ `// Current residual`
$\mathbf{d}_0 \leftarrow \mathbf{r}_0$ `// ''Descent'' direction for next solution`

for $j \leftarrow 1$ **to** J **do**
  $\mathbf{v}_j \leftarrow$ `mvm_K(` $\mathbf{d}_{j-1}$`)`
  $\alpha_j \leftarrow (\mathbf{r}_{j-1}^\top \mathbf{r}_{j-1})/(\mathbf{d}_{j-1}^\top \mathbf{v}_j)$
  $\mathbf{c}_j \leftarrow \mathbf{c}_{j-1} + \alpha_j \mathbf{d}_{j-1}$
  $\mathbf{r}_j \leftarrow \mathbf{r}_{j-1} - \alpha_j \mathbf{v}_j$
  if $\|\mathbf{r}_j\|_2 <$ tolerance **then return** $\mathbf{c}_j$;
  $\beta_j \leftarrow (\mathbf{r}_j^\top \mathbf{r}_j)/(\mathbf{r}_{j-1}^\top \mathbf{r}_{j-1})$
  $\mathbf{d}_j \leftarrow \mathbf{r}_j - \beta_j \mathbf{d}_{j-1}$
end

return $\mathbf{c}_J$

lation involving 3 vectors (see Algorithm 2.2), which makes CG very fast and memory efficient.

Convergence. While N iterations of conjugate gradients finds the exact solution, often $J \ll N$ iterations find a solution *within numerical precision* of the true solution. Through some numerical manipulation [Shewchuk, 1994, Ch. 9], one can show that the solution $\mathbf{c}_J$ at iteration J is optimal with respect to norm induced by the matrix $\mathbf{K}$:

$$\begin{aligned} \mathbf{c}_J &= \underset{\mathbf{c} \in \mathcal{K}_J(\mathbf{K},\mathbf{b})}{\arg\min} \frac{1}{2}\mathbf{c}^\top \mathbf{K}\mathbf{c} - \mathbf{c}^\top \mathbf{b} \\ &= \underset{\mathbf{c} \in \mathcal{K}_J(\mathbf{K},\mathbf{b})}{\arg\min} \|\mathbf{c} - (\mathbf{K}^{-1}\mathbf{b})\|_{\mathbf{K}}, \end{aligned} \tag{2.22}$$

where $\mathcal{K}_J(\mathbf{K}, \mathbf{b})$—the domain that $\mathbf{c}_J$ is optimal over—is referred to as the J^{th} **Krylov subpace** of $\mathbf{K}$ and $\mathbf{b}$.

$$\mathcal{K}_J(\mathbf{K}, \mathbf{b}) = \left\{\mathbf{b},\ \mathbf{K}\mathbf{b},\ \mathbf{K}^2\mathbf{b},\ \ldots,\ \mathbf{K}^{J-1}\mathbf{b}\right\}. \tag{2.23}$$

(This is why CG is referred to as a "Krylov Subspace Method".) Note the connection between Eq. (2.23) and MVMs: the vectors spanning $\mathcal{K}_J(\mathbf{K}, \mathbf{b})$ can be formed by applying a MVM to the previous vector: $\mathbf{K}^J\mathbf{b} = \mathbf{K}\left(\mathbf{K}^{J-1}\mathbf{b}\right)$. The CG search directions $\mathbf{d}_1, \ldots, \mathbf{d}_J$ lie in the Krylov subspace, and so the matrix-vector multiplication $\mathbf{K}\mathbf{d}_{J-1}$ in Algorithm 2.2 expands $\mathcal{K}_J(\mathbf{K}, \mathbf{b})$ to the next subspace $\mathcal{K}_{J+1}(\mathbf{K}, \mathbf{b})$.

Eqs. (2.22) and (2.23) explain why CG rapidly converges. Since the Krylov subspace contains powers of $\mathbf{K}$ applied to $\mathbf{b}$, we can interpret Eq. (2.22) as an *optimal polynomial approximation* of $\mathbf{K}^{-1}$:

$$\mathbf{c}_J = \underset{\text{poly}_J(\mathbf{K})}{\arg\min} \|\text{poly}_J(\mathbf{K})\mathbf{b} - (\mathbf{K}^{-1}\mathbf{b})\|_{\mathbf{K}},$$

where $\text{poly}_J(\mathbf{K})$ is a J^{th}-degree polynomial. From this perspective, one can derive CG's famous (though loose) convergence bound (see [e.g. Shewchuk, 1994, Ch. 9]):

Theorem 2.2 (Convergence of CG)**.** *Let $\mathbf{K}$ be a $N \times N$ positive definite matrix and let $\mathbf{b}$ be a vector in $\mathbb{R}^N$. After J iterations of CG, the difference between $\mathbf{c}_J$ and the true solution $\mathbf{K}^{-1}\mathbf{b}$ is bounded by:*

$$\|\mathbf{c}_J - (\mathbf{K}^{-1}\mathbf{b})\|_{\mathbf{K}} \leq 2\left[\frac{\sqrt{\kappa(\mathbf{K})} - 1}{\sqrt{\kappa(\mathbf{K})} + 1}\right]^J \|\mathbf{b}\|_{\mathbf{K}},$$

where $\kappa(\mathbf{K}) = \|\mathbf{K}\|_2/\|\mathbf{K}^{-1}\|_2$ is the condition number *of $\mathbf{K}$.*

Thus CG converges exponentially to the true solve $\mathbf{K}\mathbf{b}^{-1}$. This bound depends on the condition number of $\mathbf{K}$, which is often quite large for GP kernel matrices.

In practice however, convergence can be much faster if, for example, the eigenvalues of $\mathbf{K}$ are clustered [Saad, 2003]. We find that a few hundred iterations of CG often produces solves within 3-5 decimal places for most kernel matrices, even when $N \geq 100{,}000$.

Preconditioning. A commonly-used approach to accelerate CG convergence is to lower the $\kappa(\mathbf{K})$ condition number in Theorem 2.2. This can be accomplished through **preconditioning**, which introduces a matrix $\mathbf{P}$ to solve the related linear system

$$\mathbf{P}^{-1}\mathbf{K}\mathbf{c} = \mathbf{P}^{-1}\mathbf{b}$$

instead of $\mathbf{K}^{-1}\mathbf{b}$. Both systems have the same solution $\mathbf{c}$, but the preconditioned system's convergence depends on the conditioning of $\mathbf{P}^{-1}\mathbf{K}$ rather than that of $\mathbf{K}$. The algorithm for *preconditioned conjugate gradients* (PCG) is essentially the same as vanilla CG, with the additional step of applying $\mathbf{P}^{-1}$ to the MVMs (i.e. $\mathbf{P}^{-1}\mathbf{K}\mathbf{d}_{j-1}$) and residuals (i.e. $\mathbf{P}^{-1}\mathbf{r}_j$). See Algorithm 2.3 for details.

Choosing a preconditioner $\mathbf{P}^{-1}$ is a trade-off between computational efficiency and effectiveness. Trivially, the best preconditioner is $\mathbf{P}^{-1} = \mathbf{K}^{-1}$ which would reduce the condition number in Theorem 2.2 to $\kappa = 1$. However this is obviously not a practical choice—if we already had a way to efficiently compute $\mathbf{K}^{-1}$ there would be no need to run CG! The most effective preconditioners have simple-to-compute inverses (e.g. diagonal matrices), yet are able to closely approximate $\mathbf{K}^{-1}$. A common (though often ineffective) choice of $\mathbf{P}$ is the *Jacobi preconditioner*, which is simply the diagonal of $\mathbf{K}$ (i.e. $\mathbf{P} = \text{diag}(\mathbf{K})$).

The design of effective preconditioners is an active area of research and is too extensive to adequately review here. We refer the reader to [Saad, 2003] which

Algorithm 2.3: Preconditioned conjugate gradients (PCG). Terms in violet differ from standard CG (Algorithm 2.2).

Input : `mvm_K(·)` – function for matrix-vector multiplication (MVM) with matrix $\mathbf{K}$
$\mathbf{b}$ – vector to solve against
$\mathbf{P}^{-1}(\cdot)$ – function for preconditioner
Output: $\mathbf{c} = \mathbf{K}^{-1}\mathbf{b}$.

$\mathbf{c}_0 \leftarrow \mathbf{0}$ `// Current solution`
$\mathbf{r}_0 \leftarrow$ `mvm_K(`$\mathbf{c}_0$`)` - $\mathbf{b}$ `// Current residual`
$\mathbf{z}_0 \leftarrow \mathbf{P}^{-1}(\mathbf{r}_0)$ `// Preconditioned residual`
$\mathbf{d}_0 \leftarrow \mathbf{z}_0$ `// ''Descent'' direction for next solution`

for $j \leftarrow 1$ **to** J **do**
  $\mathbf{v}_j \leftarrow$ `mvm_K(`$\mathbf{d}_{j-1}$`)`
  $\alpha_j \leftarrow (\mathbf{r}_{j-1}^\top \mathbf{z}_{j-1})/(\mathbf{d}_{j-1}^\top \mathbf{v}_j)$
  $\mathbf{c}_j \leftarrow \mathbf{c}_{j-1} + \alpha_j \mathbf{d}_{j-1}$
  $\mathbf{r}_j \leftarrow \mathbf{r}_{j-1} - \alpha_j \mathbf{v}_j$
  if $\|\mathbf{r}_j\|_2 <$ tolerance **then return** $\mathbf{c}_j$;
  $\mathbf{z}_j \leftarrow \mathbf{P}^{-1}(\mathbf{r}_j)$
  $\beta_j \leftarrow (\mathbf{z}_j^\top \mathbf{z}_j)/(\mathbf{z}_{j-1}^\top \mathbf{z}_{j-1})$
  $\mathbf{d}_j \leftarrow \mathbf{z}_j - \beta_j \mathbf{d}_{j-1}$
end

return $\mathbf{c}_J$

devotes two chapters to different preconditioning methods.

2.3.2 Lanczos Tridiagonalization

The Lanczos [1950] algorithm iteratively produces a partial tridiagonalization of the symmetric matrix $\mathbf{K} \in \mathbb{R}^{N \times N}$. At iteration J, $\mathbf{K}$ is factorized as

$$\mathbf{K}\mathbf{Q}_J = \mathbf{Q}_J \mathbf{T}_J + \mathbf{r}_J \mathbf{e}_J^\top, \tag{2.24}$$

where $\mathbf{Q}_J \in \mathbb{R}^{N \times J}$ has orthonormal columns, $\mathbf{T}_J \in \mathbb{R}^{J \times J}$ is symmetric tridiagonal, and $(\mathbf{r}_J \mathbf{e}_J^\top) \in \mathbb{R}^{N \times J}$ is a residual term involving the unit vector $\mathbf{e}_J$. After N iterations, the residual term will be zero and we recover the complete tridiago-

nalization $\mathbf{K} = \mathbf{Q}_N \mathbf{T}_N \mathbf{Q}_N^\top$. In practice however, the complete tridiagonalization is almost never computed. The matrices $\mathbf{Q}_J, \mathbf{T}_J$ after $J \ll N$ iterations can approximate functions involving $\mathbf{K}$ with high degrees of accuracy. We will discuss how $\mathbf{Q}_J$ and $\mathbf{T}_J$ can approximate log determinants and matrix solves after introducing the algorithm itself.

The algorithm. Given an initial vector $\mathbf{b}$, J iterations of the Lanczos algorithm form an orthonormal basis of the J^{th} Krylov subspace (Eq. (2.23)). Applying Gram-Schmidt orthogonalization to $\left[\mathbf{b},\ \mathbf{K}\mathbf{b},\ \ldots,\ \mathbf{K}^{J-1}\mathbf{b}\right]$ produces the columns of $\mathbf{Q}_J = \left[\mathbf{b}/\|\mathbf{b}\|,\ \mathbf{q}^{(2)},\ \ldots,\ \mathbf{q}^{(J)}\right]$, and the orthogonalization coefficients are collected into $\mathbf{T}_J$. Because $\mathbf{K}$ is symmetric, each vector only needs to be orthogonalized against the two preceding vectors, which results in the tridiagonal structure of $\mathbf{T}$ [Golub and Van Loan, 2012]. In practice, the columns of $\mathbf{Q}_J$ and the sub-diagonal/diagonal entries of $\mathbf{T}_j$ can be computed using a simple iterative procedure (Algorithm 2.4). Each iteration produces a new column of $\mathbf{Q}_J$, requiring a single MVM with $\mathbf{K}$. In total J iterations requires J MVMs.

In practice, Algorithm 2.4 can be numerically unstable after many iterations of J (e.g. $J > 50$). This is because the columns of $\mathbf{Q}_J$ lose orthogonality due to round-off errors. We discuss ways to overcome these instabilities in Section 4.6.

Using Lanczos to estimate $f(\mathbf{K})\mathbf{b}$. Lanczos tridiagonalization is a general-purpose algorithm, as it can be used to compute an approximation for any $f(\mathbf{K})\mathbf{b}$, where $f(\cdot)$ is a matrix function (i.e. matrix inverse or matrix logarithm). First, we note that, if $\mathbf{Q}_N \mathbf{T}_N \mathbf{Q}_N^\top = \mathbf{K}$ is a complete tridiagonalization, then $\mathbf{T}_N$

Algorithm 2.4: Lanczos tridiagonalization.

Input : `mvm_K()` – function for matrix-vector multiplication (MVM) with matrix $\mathbf{K}$
$\mathbf{b}$ – initial (probe) vector
J – number of iterations

Output: $\mathbf{Q}_J, \mathbf{T}_J$ – orthonormal and tridiagonal matrices

$\mathbf{q}^{(1)} \leftarrow \mathbf{b}/\|\mathbf{b}\|$ `// Current column of` $\mathbf{Q}_J$
$\mathbf{v} \leftarrow \texttt{mvm\_}\mathbf{K}(\mathbf{q}^{(1)})$ `// Buffer for MVM output`
$\gamma^{(1)} \leftarrow \mathbf{q}^{(1)\top}\mathbf{v}$ `// Current main diagonal entry of` $\mathbf{T}_J$
$\mathbf{v} \leftarrow \mathbf{v} - \gamma^{(1)}\mathbf{q}^{(1)}$
$\delta^{(1)} \leftarrow \|\mathbf{v}\|$ `// Current sub-diagonal entry of` $\mathbf{T}_J$

for $j \leftarrow 2$ **to** J **do**
  `// Run Gram-Schmidt orth. against previous two` $\mathbf{Q}_J$ `vectors`
  $\mathbf{q}^{(j)} \leftarrow \mathbf{v}/\delta^{(j-1)}$
  $\mathbf{v} \leftarrow \texttt{mvm\_}\mathbf{K}(\mathbf{q}^{(j)}) - \delta^{(j-1)}\mathbf{q}^{(j-1)}$
  $\gamma^{(j)} \leftarrow \mathbf{q}^{(j)\top}\mathbf{v}$
  $\mathbf{v} \leftarrow \mathbf{v} - \gamma^{(j)}\mathbf{q}^{(j)}$
  $\delta^{(j)} \leftarrow \|\mathbf{v}\|$
end
`// Collect orthonormal columns` $\mathbf{q}^{(j)}$ `and tridiagonal entries` $\gamma^{(j)}$, $\delta^{(j)}$
$\mathbf{Q}_J \leftarrow \left[\mathbf{q}^{(1)}, \ \ldots, \ \mathbf{q}^{(J)}\right]$
$\forall j \in [1, J] \quad T_J^{(j,j)} \leftarrow \gamma^{(j)}$
$\forall j \in [1, J-1] \quad T_J^{(j,j-1)} \leftarrow \delta_{(j)}$
$\forall j \in [1, J-1] \quad T_J^{(j-1,j)} \leftarrow \delta_{(j)}$

return $\mathbf{Q}_J$, $\mathbf{T}_J$

and $\mathbf{K}$ are similar matrices and therefore:

$$f(\mathbf{K}) = \mathbf{Q}_N\big[f(\mathbf{T}_N)\big]\mathbf{Q}_N^\top. \tag{2.25}$$

Typically, the Lanczos algorithm is used only to compute a *partial* tridiagonalization $\mathbf{Q}_J$, $\mathbf{T}_J$ with $J \ll N$. However, this partial tridiagonalization can be used to approximate $f(\mathbf{K})$ applied to the initial Lanczos vector $\mathbf{b}$:

$$\begin{aligned} f(\mathbf{K})\mathbf{b} &\approx \Big(\mathbf{Q}_J\big[f(\mathbf{T}_J)\big]\mathbf{Q}_J^\top\Big)\mathbf{b}. \\ &= \|\mathbf{b}\|_2\Big(\mathbf{Q}_J\big[f(\mathbf{T}_J)\big]\Big)\mathbf{e}_1 \triangleq \mathbf{c}_J, \end{aligned} \tag{2.26}$$

where the second line holds because $\mathbf{q}_1 = \mathbf{b}/\|\mathbf{b}\|$. The estimate $\mathbf{c}_J$ tends to converge exponentially as J increases, though the rate of convergence depends on the conditioning of $\mathbf{K}$. Moreover, the computational complexity of Eq. (2.26) is quite small since we can take advantage of the tridiagonal structure of $\mathbf{T}_J$.

For many functions of interest, there are special variants of Eq. (2.26) that offer more efficient computation. We now outline two special cases that can be applied to Gaussian process inference.

Estimating log determinants. Ubaru et al. [2017] introduce a method to produce unbiased estimates of log determinants using Lanczos tridiagonalization and *stochastic trace estimation* [Hutchinson, 1990, Avron and Toledo, 2011, Fitzsimons et al., 2018]. They note that the log determinant can be re-written as:

$$\begin{aligned}\log|\mathbf{K}| &= \operatorname{Tr}(\log\mathbf{K}) \\ &\approx \frac{1}{T}\sum_{i=1}^{T}\mathbf{z}^{(i)\top}(\log\mathbf{K})\,\mathbf{z}^{(i)}. \end{aligned} \tag{2.27}$$

where $\log\mathbf{K}$ denotes the matrix logarithm, and $\mathbf{z}^{(i)}$ are i.i.d random variables with zero mean and unit covariance (e.g. $\mathbf{z}^{(i)} \sim \mathcal{N}[0, \mathbf{I}]$). It can be seen with a little mathematical manipulation that the second line is an unbiased estimate of the trace operator [Hutchinson, 1990].

To compute each of the $\mathbf{z}^{(i)}(\log\mathbf{K})\,\mathbf{z}^{(i)}$ terms, we turn to Eq. (2.25). Let $\mathbf{Q}_J^{(i)}$, $\mathbf{T}_J^{(i)}$ correspond to the Lanczos matrices with initial vector $\mathbf{q}_1^{(i)} = \mathbf{z}^{(i)}/\|\mathbf{z}^{(i)}\|$. Then we apply Eq. (2.26):

$$\begin{aligned}\mathbf{z}^{(i)\top}(\log\mathbf{K})\,\mathbf{z}^{(i)} &\approx \mathbf{z}^{(i)\top}\left(\mathbf{Q}_J^{(i)}\left[\log\mathbf{T}_J^{(i)}\right]\mathbf{Q}_J^{(i)\top}\right)\mathbf{z}^{(i)} \\ &= \|\mathbf{z}^{(i)}\|_2^2\left(\mathbf{e}_1^\top\left[\log\mathbf{T}_J^{(i)}\right]\mathbf{e}_1\right). \end{aligned} \tag{2.28}$$

Combining Eq. (2.27) and Eq. (2.28) gives us an unbiased estimate of $\log|\mathbf{K}|$. This estimate, referred to as the **stochastic Lanczos quadrature** estimate of $\log|\mathbf{K}|$, converges exponentially in J (the number of Lanczos iterations):

Theorem 2.3 (Corrolary 4.5 of Ubaru et al. [2017]). *Let $\mathbf{K} \in \mathbb{R}^{N\times N}$ be a positive definite matrix with condition number $\kappa(\mathbf{K})$. Suppose we estimate $\Gamma \approx \log|\mathbf{K}|$ using Eqs.* (2.27) *and* (2.28) *with:*

- $J \geq \frac{1}{4}\sqrt{3\kappa(\mathbf{K})}\log\left(\mathcal{O}(\kappa(\mathbf{K}))/\epsilon\right)$ *iterations of Lanczos,*[5] *and*
- $T \geq \frac{32}{\epsilon^2}\log\left(\frac{2}{\delta}\right)$ *random $\mathbf{z}^{(i)} \sim \mathcal{N}[\mathbf{0},\mathbf{I}]$ vectors.*[6]

Then the error of the stochastic Lanczos quadrature estimate Γ is probabilistically bounded by:

$$Pr\left[\left|\log|\mathbf{K}| - \Gamma\right| \leq \epsilon N\right] \geq (1-\delta)\,.$$

Computing matrix solves. Another common application of Lanczos tridiagonalization is computing $\mathbf{K}^{-1}\mathbf{b}$. Let $\mathbf{Q}_J$ and $\mathbf{T}_J$ be the Lanczos matrices using $\mathbf{b}$ as the initial vector (i.e. $\mathbf{q}_1 = \mathbf{b}/\|\mathbf{b}\|$). We can approximate the matrix solve using Eq. (2.26):

$$\mathbf{K}^{-1}\mathbf{b} \approx \|\mathbf{b}\|_2\mathbf{Q}_J\mathbf{T}_J^{-1}\mathbf{e}_1 \tag{2.29}$$

where again $\mathbf{e}_1 \in \mathbb{R}^J$ is the first unit vector $[1,0,0,\ldots,0]$.

[5] The exact value for the $\mathcal{O}(\kappa(\mathbf{K}))$ constant is $\left(5\kappa(\mathbf{K})\log(2(\kappa(\mathbf{K})+1))/\sqrt{2(\kappa(\mathbf{K})+1)}\right)$.

[6] The constant originally used by Ubaru et al. [2017] assumes that the $\mathbf{z}^{(i)}$ are Rademacher random variables rather than Gaussian random variables. The variance of stochastic trace estimation with Gaussian variables is bounded with a factor of 32 rather than 24 (see [Roosta-Khorasani and Ascher, 2015, Eqs. 4 and 5]), and so we have adjusted the constant accordingly.

2.3.3 Connection between CG and Lanczos

The Lanczos and CG algorithms are very closely related, as they are both Krylov subspace methods. Here we show that the two algorithms can be derived from one another. We will exploit this fact in the next two chapters.

Deriving CG from Lanczos. It can be shown that, if $\mathbf{K}$ is positive definite, then the solution in Eq. (2.29) is *exactly* the same as the CG solution after J iterations. In fact, one way to derive Algorithm 2.2 is by showing that Eq. (2.29) can be reduced to the three-term CG recurrence. In practice, the CG algorithm tends to be preferred over Eq. (2.29) for computing $\mathbf{K}^{-1}\mathbf{b}$. The advantages of CG are

(1) CG only stores 3 vectors at any given iteration, whereas Eq. (2.29) requires storing the $\mathbf{Q}_J \in \mathbb{R}^{N\times J}$ matrix;

(2) CG is numerically stable, whereas the Lanczos vectors lose orthogonality (after say $J > 50$ iterations); and

(3) CG is more easily preconditioned.

However, one advantage of Lanczos is that the $\mathbf{Q}_J$ and $\mathbf{T}_J$ matrices can be used to jump-start subsequent solves $\mathbf{K}^{-1}\mathbf{b}'$. Parlett [1980], Saad [1987], and Schneider and Willsky [2001] argue that subsequent solves can be approximated as

$$\mathbf{K}^{-1}\mathbf{b}' \approx \left(\mathbf{Q}_J \mathbf{T}_J^{-1} \mathbf{Q}_J^\top\right) \mathbf{b}', \tag{2.30}$$

where $\mathbf{Q}_J$ and $\mathbf{T}_J$ come from a previous solve.[7] We will use this in Chapter 4.

[7] Alternatively, one could use $\left(\mathbf{Q}_J \mathbf{T}_J^{-1} \mathbf{Q}_J^\top\right) \mathbf{b}'$ as an "initial guess" to the solve $\mathbf{K}^{-1}\mathbf{b}'$ that can be refined with additional CG iterations.

Deriving Lanczos from CG. Additionally, one can recover part of the Lanczos tridiagonalization from conjugate gradients. Saad [2003] and others show that it is possible to recover the $\mathbf{T}_J$ tridigonal Lanczos matrix by reusing the α_j and β_j coefficients generated in the CG iterations (see Algorithm 2.3).

Observation 2.1 (Recovering Lanczos tridiagonal matrices from PCG [Saad, 2003])**.** *Assume we use J iterations of standard preconditioned conjugate gradients to solve $\mathbf{K}^{-1}\mathbf{b}$ with preconditioner $\mathbf{P}$. Let $\alpha_1, \ldots, \alpha_p$ and $\beta_1, \ldots, \beta_p$ be the scalar coefficients from each iteration (defined in Algorithm 2.3). The matrix*

$$\begin{bmatrix} \frac{1}{\alpha_1} & \frac{\sqrt{\beta_1}}{\alpha_1} & & & 0 \\ \frac{\sqrt{\beta_1}}{\alpha_1} & \frac{1}{\alpha_2} + \frac{\beta_1}{\alpha_1} & \frac{\sqrt{\beta_2}}{\alpha_2} & & \\ & \frac{\sqrt{\beta_2}}{\alpha_2} & \frac{1}{\alpha_3} + \frac{\beta_2}{\alpha_2} & \frac{\sqrt{\beta_3}}{\alpha_3} & \\ & & \ddots & \ddots & \frac{\sqrt{\beta_{m-1}}}{\alpha_{m-1}} \\ 0 & & & \frac{\sqrt{\beta_{m-1}}}{\alpha_{m-1}} & \frac{1}{\alpha_m} + \frac{\beta_{m-1}}{\alpha_{m-1}} \end{bmatrix}$$

is equal to the Lanczos tridiagonal matrix $\mathbf{T}_J$, formed by running J iterations of Lanczos to achieve $(\mathbf{P}^{-1}\mathbf{K})\,\mathbf{Q}_J = \mathbf{Q}_J\mathbf{T}_J + \mathbf{r}\mathbf{e}_J^\top$) with probe vector $\mathbf{b}$.

(See [Saad, 2003], Section 6.7.3.) In other words, we can recover the Lanczos tridiagonal matrix $\mathbf{T}_J$ simply by running CG. However, the orthonormal matrix $\mathbf{Q}_J$ cannot be as easily derived as a CG byproduct.

2.3.4 MINRES

The method of minimum residuals (MINRES) [Paige and Saunders, 1975] is an alternative to linear conjugate gradients, with the advantage that it can be applied to indefinite symmetric matrices. Paige and Saunders [1975] formulate

MINRES to solve the least-squares problem $\arg\min_{\mathbf{c}} \|\mathbf{K}\mathbf{c} - \mathbf{b}\|_2$. Each iteration J produces a solution $\mathbf{c}_J$ which is optimal within the J^{th} Krylov subspace:

$$\mathbf{c}_J^{(\text{MINRES})} = \underset{\mathbf{c} \in \mathcal{K}_J(\mathbf{K}, \mathbf{b})}{\arg\min} \|\mathbf{K}\mathbf{c} - \mathbf{b}\|_2. \tag{2.31}$$

Note the primary differences between Eq. (2.31) and the CG optimality equation (Eq. (2.22)): CG optimizes the *error* with respect to the $\mathbf{K}$ norm, while MINRES optimizes the *residual* with respect to the 2 norm. This is why MINRES can be applied to indefinite matrices while CG cannot.

Through some mathematical manipulation, one can reformulate Eq. (2.31) as an unconstrained optimization problem:

$$\mathbf{c}_J^{(\text{MINRES})} = \|\mathbf{b}\|_2 \mathbf{Q}_J \mathbf{z}_J, \quad \mathbf{z}_J = \underset{\mathbf{z} \in \mathbb{R}^J}{\arg\min} \left\| \left(\widetilde{\mathbf{T}}_J \right) \mathbf{z} - \mathbf{e}_1 \right\|_2, \quad \widetilde{\mathbf{T}}_J \begin{bmatrix} \mathbf{T}_J \\ \|\mathbf{r}_J\|_2 \mathbf{e}_J^\top \end{bmatrix}, \tag{2.32}$$

where $\mathbf{e}_1, \mathbf{e}_J$ are unit vectors, and $\mathbf{Q}_J$, $\mathbf{T}_J$, and $\mathbf{r}_J$ are the outputs from the Lanczos algorithm. Since Eq. (2.32) is a least-squares problem, we can write its analytic solution using the QR factorization of $\widetilde{\mathbf{T}}_J = \boldsymbol{\mathcal{Q}}_J \mathbf{R}_J$:

$$\mathbf{c}_J^{(\text{MINRES})} = \|\mathbf{b}\|_2 \, \mathbf{Q}_J \left(\mathbf{R}_J^{-1} \boldsymbol{\mathcal{Q}}_J^\top \right) \mathbf{e}_1. \tag{2.33}$$

One way to perform MINRES is first running J iterations of the Lanczos algorithm, computing $\widetilde{\mathbf{T}}_J = \boldsymbol{\mathcal{Q}}_J \mathbf{R}_J$, and then plugging the resulting $\mathbf{Q}_J$, $\boldsymbol{\mathcal{Q}}_J$, and $\mathbf{R}_J$ into Eq. (2.33). Paige and Saunders instead introduce a vector recurrence to iteratively compute $\mathbf{c}_J^{(\text{MINRES})}$. This recurrence relation, which is given by Algorithm 2.5 and broadly described below, is exactly equivalent to Eq. (2.33); however it uses careful bookkeeping to avoid storing any $N \times J$ terms.

First we note that the $\widetilde{\mathbf{T}}_J$ matrices are formed recursively, and thus their QR

factorizations are also recursive:

$$\boldsymbol{\mathcal{Q}}^\top \widetilde{\mathbf{T}}_J = \begin{bmatrix} \boldsymbol{\mathcal{Q}}_{J-1}^\top & \boldsymbol{\mathcal{Q}}^{\top(J,1:J-1)} \\ \boldsymbol{\mathcal{Q}}^{\top(1:J-1,J+1)} & \mathcal{Q}^{(J,J+1)} \end{bmatrix} \begin{bmatrix} \widetilde{\mathbf{T}}_{J-1} & \mathbf{t}^{(J)} \\ \mathbf{0}^\top & \|\mathbf{r}_J\| \end{bmatrix} = \begin{bmatrix} \mathbf{R}_{J-1} & \mathbf{r}^{(J,1:J-1)} \\ \mathbf{0} & R^{(J,J)} \end{bmatrix} = \mathbf{R}_J$$

where $\mathbf{t}^{(J)}$ and $[\mathbf{r}^{(J,1:J-1)}; R^{(J,J)}]$ are the last columns of $\mathbf{T}_J$ and $\mathbf{R}_J$ respectively. Moreover, if we recursively form $\mathbf{R}_J^{-1}$ as

$$\mathbf{R}_J^{-1} = \begin{bmatrix} \mathbf{R}_{J-1} & \mathbf{r}^{(J,1:J-1)} \\ \mathbf{0} & R^{(J,J)} \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{R}_{J-1}^{-1} & \left(\mathbf{R}_{J-1}^{-1}\mathbf{r}^{(J,1:J-1)}\right)/R^{(J,J)} \\ \mathbf{0} & 1/R^{(J,J)} \end{bmatrix},$$

then Eq. (2.33) can be re-written in a decent-style update:

$$\begin{aligned} \mathbf{c}_J^{(\text{MINRES})} &= \|\mathbf{b}\|_2 \begin{bmatrix} \mathbf{Q}_{J-1}\mathbf{q}^{(J)} \end{bmatrix} \begin{bmatrix} \mathbf{R}_{J-1}^{-1} & \frac{\mathbf{R}_{J-1}^{-1}\mathbf{r}^{(J,1:J-1)}}{R^{(J,J)}} \\ \mathbf{0} & 1/R^{(J,J)} \end{bmatrix} \begin{bmatrix} \boldsymbol{\mathcal{Q}}_{J-1}^\top & \boldsymbol{\mathcal{Q}}^{\top(J,1:J-1)} \\ \boldsymbol{\mathcal{Q}}^{\top(1:J-1,J+1)} & \mathcal{Q}^{(J,J+1)} \end{bmatrix} \mathbf{e}_1 \\ &= \|\mathbf{b}\|_2 \begin{bmatrix} \mathbf{Q}_{J-1}\mathbf{R}_{J-1}^{-1} & \frac{\mathbf{Q}_{J-1}\mathbf{R}_{J-1}^{-1}\mathbf{r}^{(J,1:J-1)}}{R^{(J,J)}} \\ \mathbf{0} & 1/R^{(J,J)}\mathbf{q}_{J-1} \end{bmatrix} \begin{bmatrix} \boldsymbol{\mathcal{Q}}_{J-1}^\top \mathbf{e}_1 \\ \mathcal{Q}^{\top(1,J+1)} \end{bmatrix} \\ &= \underbrace{\left(\|\mathbf{b}\|_2 \mathbf{Q}_{J-1}\mathbf{R}_{J-1}^{-1}\boldsymbol{\mathcal{Q}}_{J-1}\mathbf{e}_1\right)}_{\mathbf{c}_{J-1}^{(\text{MINRES})}} + \underbrace{\frac{\|\mathbf{b}\|_2 \mathcal{Q}^{\top(1,J+1)}}{R^{(J,J)}}}_{\varphi_J} \underbrace{\begin{bmatrix} \mathbf{Q}_{J-1}\mathbf{R}_{J-1}^{-1}\mathbf{r}^{(J,1:J-1)} \\ \mathbf{q}_{J-1} \end{bmatrix}}_{\mathbf{d}_J}. \end{aligned} \tag{2.34}$$

Thus $\mathbf{c}_J^{(\text{MINRES})} = \mathbf{c}_{J-1}^{(\text{MINRES})} + \varphi_J \mathbf{d}_J$. We note that $\mathbf{r}^{(J,1:J-1)}$, which is the next entry in the QR factorization of $\widetilde{\mathbf{T}}_J$, can be cheaply computed using Givens rotations (see [e.g. Golub and Van Loan, 2012, Ch. 11.4.1]). Moreover, only the last two entries of $\mathbf{r}^{(J,1:J-1)}$ will be non-zero (due to the tridiagonal structure of $\widetilde{\mathbf{T}}_J$. Consequentially, we only need to store the last two vectors of $\mathbf{Q}_{J-1}\mathbf{R}_{J-1}^{-1}$, which again can be computed recursively.

The recurrence requires the storage of ≈ 6 vectors. As with CG, each iteration requires a single MVM with $\mathbf{K}$ (to form the next Lanczos vector $\mathbf{q}_J$); all subsequent operations are $\mathcal{O}(N)$. The entire procedure is given by Algorithm 2.5.

Algorithm 2.5: Method of minimum residuals (MINRES).

Input : `mvm_K(·)` – function for MVM with matrix $\mathbf{K}$
$\mathbf{b}$ – vector to solve against
Output: $\mathbf{c} = \mathbf{K}^{-1}\mathbf{b}$.

$\mathbf{c}_1 \leftarrow \mathbf{0}$ // Current solution.
$\mathbf{d}_1, \mathbf{d}_0 \leftarrow \mathbf{0}$ // Current & prev. ''search'' direction.
$\varphi_2 \leftarrow \|\mathbf{b}\|_2$ // Current ''step'' size.

$\mathbf{q}_1 \leftarrow \mathbf{b}/\|\mathbf{b}\|_2$ // Current Lanczos vector.
$\mathbf{v}_1 \leftarrow \texttt{mvm\_}\mathbf{K}(\mathbf{q}_0)$ // Buffer for MVM output.
$\delta_1 \leftarrow \|\mathbf{b}\|_2$ // Current Lanczos residual/sub-diagonal.
$\delta_0 \leftarrow 1$ // Prev. Lanczos residual/sub-diagonal.
$\eta_1 \leftarrow 1$ // Current scaling term.
$\eta_0 \leftarrow 0$ // Prev. scaling term.

for $j \leftarrow 2$ **to** J **do**
  // Run one iter of Lanczos. Gets next vector of $\mathbf{Q}$ matrix, and next diag/sub-diag (γ, δ) entries of $\mathbf{T}$ matrix.
  $\mathbf{q}_j \leftarrow \mathbf{v}_j/\delta_j$
  $\mathbf{v}_j \leftarrow \texttt{mvm\_}\mathbf{K}(\mathbf{q}_j) - \delta_j \mathbf{q}_{j-1}$
  $\gamma_j \leftarrow \mathbf{q}_j \mathbf{v}_j$
  $\mathbf{v}_j \leftarrow \mathbf{v}_j - \gamma_j \mathbf{q}_j$
  $\delta_j \leftarrow \|\mathbf{v}_j\|$
  // Compute the next $\mathbf{r}^{(J)}$ (part of QR) via Givens rotations. There are three non-0 entries: $\mathbf{R}^{(J,J-2:J)} = [\epsilon_J, \zeta_J, \eta_J]$.
  $\epsilon_j \leftarrow \delta_{j-1}\left(\delta_{j-2}/\sqrt{\delta_{j-2}^2 + \eta_{j-2}^2}\right)$
  $\zeta_j \leftarrow \delta_{j-1}\left(\eta_{j-2}/\sqrt{\delta_{j-2}^2 + \eta_{j-2}^2}\right)$
  $\eta_j \leftarrow \gamma_j\left(\eta_{j-1}/\sqrt{\delta_{j-1}^2 + \eta_{j-1}^2}\right) + \zeta_j\left(\delta_{j-1}/\sqrt{\delta_{j-1}^2 + \eta_{j-1}^2}\right)$
  $\zeta_j \leftarrow \zeta_j\left(\eta_{j-1}/\sqrt{\delta_{j-1}^2 + \eta_{j-1}^2}\right) + \gamma_j\left(\delta_{j-1}/\sqrt{\delta_{j-1}^2 + \eta_{j-1}^2}\right)$
  $\eta_j \leftarrow \eta_j\left(\eta_j/\sqrt{\delta_j^2 + \eta_j^2}\right)$
  // Compute ''step'' size $\varphi_J = \boldsymbol{\mathcal{Q}}^{(1,J+1)}/R^{(J,J)}$.
  $\varphi_j \leftarrow \varphi_{j-1}\left(\delta_{j-1}/\sqrt{\delta_{j-1}^2 + \eta_{j-1}^2}\right)\left(\eta_j/\sqrt{\delta_j^2 + \eta_j^2}\right)$
  // Update the current solution based on the $\mathbf{r}^{(J)}$ entries $(\epsilon_J, \zeta_J, \eta_J)$ and previous search vectors $\mathbf{d}_{j-1}$, $\mathbf{d}_{j-2}$.
  $\mathbf{d}_j \leftarrow (\mathbf{q} - \zeta_j \mathbf{d}_{j-1} - \epsilon_j \mathbf{d}_{j-2})/\eta_j$
  $\mathbf{c}_j \leftarrow \mathbf{c}_{j-1} + \varphi_j \mathbf{d}_j$
end

return $\|\mathbf{b}\|_2\, \mathbf{c}_j$

CHAPTER 3

GAUSSIAN PROCESS TRAINING VIA BLACK-BOX MATRIX × MATRIX INFERENCE

3.1 Introduction

Training a Gaussian process regression model ensures that the mean function, kernel function, and likelihood are well suited to a given dataset. It is worth noting that GPs have few learnable parameters, especially compared to highly-parametric models like neural networks. Nevertheless, the mean, kernel, and likelihood parameters greatly influence the performance of the model and therefore should be well-chosen. We will denote this set of learnable parameters by the vector $\boldsymbol{\theta}$.

Typically, $\boldsymbol{\theta}$ is learned by optimizing the GP marginal log likelihood (Eq. (2.6)) via a gradient descent method [Rasmussen and Williams, 2006]. Alternatively, $\boldsymbol{\theta}$ can be inferred via eliptic slice sampling [Murray et al., 2010], gradient-based samplers [Havasi et al., 2018], or other MCMC methods. Eq. (2.6) can be also used to choose an appropriate kernel function via Bayesian model selection [Rasmussen and Williams, 2006, Duvenaud et al., 2013]. Regardless of the desired mechanism—optimization, sampling, or model selection—training a Gaussian process will require repeatedly computing the GP marginal log likelihood and its derivative (i.e. $\approx 50 - 100$ times).

Most GP implementations compute the marginal log likelihood and its derivative using the Cholesky factor of the training kernel matrix $\mathbf{L}\mathbf{L}^\top = \widehat{\mathbf{K}}_{\mathbf{XX}}$. As described in Chapter 1 this has numerous drawbacks. Besides its $\mathcal{O}(N^3)$ time

complexity and $\mathcal{O}(N^2)$ space requirements, the Cholesky factorization is an inherently sequential algorithm that does not effectively utilize modern hardware like GPUs. Scalable approximations can remedy these concerns to some extent, yet such approximations tend to require significant implementation efforts.

In this chapter, we introduce a highly efficient framework for Gaussian process training. Whereas previous approaches require the user to provide routines for computing the GP marginal log likelihood for a sufficiently complex model, our framework only requires access to a blackbox routine that performs matrix-matrix multiplications with the kernel matrix and its derivative. Accordingly, we refer to our method as Black-Box Matrix × Matrix (BBMM) Inference.

In contrast to the Cholesky decomposition, matrix multiplication fully utilizes GPU acceleration. We will demonstrate that this approach also significantly eases implementation for a wide class of specialty GP models. In particular, we make the following contributions:

(1) We introduce a modified *batched* version of linear conjugate gradients (mBCG) that provides all computations necessary for both the marginal likelihood and its derivatives. mBCG uses matrix-matrix multiplications that more efficiently utilize modern hardware than both existing Cholesky and MVM based training. It also circumvents several critical space complexity and numerical stability issues present in existing MVM methods. Perhaps most notably, mBCG reduces the time complexity of exact GP inference from $\mathcal{O}(N^3)$ to $\mathcal{O}(N^2)$.

(2) We introduce a method for *preconditioning* this modified conjugate gradients algorithm based on the partial pivoted Cholesky decomposition [Harbrecht et al., 2012]. All required operations with this preconditioner are

efficient, and in practice require negligible time.

(3) We empirically demonstrate the efficacy of BBMM for medium-scale exact GPs and several scalable methods. On datasets with up to $3{,}000$ data points, we show that exact GPs with BBMM are up to $20\times$ faster than Cholesky-based GPs. Moreover, the popular SKI [Wilson and Nickisch, 2015] and SGPR [Titsias, 2009] frameworks with BBMM achieve up to $15\times$ and $4\times$ speedups (respectively) on datasets as large as $500{,}000$ data points. We also demonstrate that BBMM performs linear solves with higher accuracy than Cholesky in single-precision arithmetic.

In addition, we discuss how BBMM is incorporated into the GPyTorch software package. We demonstrate that BBMM enables simple implementations of exact GPs, multi-output GPs, and specialty models like the SKI and SGPR approximations.

Related work. Recently, a number of researchers have proposed alternatives to Cholesky-based training, instead relying on iterative numerical algorithms to compute Eqs. (2.6) and (2.7) [Cunningham et al., 2008, Murray, 2009, Saatçi, 2012, Wilson, 2014, Wilson and Nickisch, 2015, Cutajar et al., 2016, Dong et al., 2017, Gardner et al., 2018b]. These approaches rely on the conjugate gradients and Lanczos algorithms described in Section 2.3, both of which perform matrix-vector multiplication (MVMs) with the training kernel matrix. One key advantage is that MVM approaches can exploit algebraic structure for increased computational efficiencies.

The method proposed in this chapter builds upon this prior work in a few notable ways. Firstly, the mBCG method computes all training terms through a

single iterative method. This is in contrast to existing methods that use separate algorithms for matrix solves (conjugate gradients) and log determinants (Lanczos quadrature). Secondly, our BBMM framework is designed to be an *general purpose* training approach. Many of the existing MVM-based methods are employed only when the training kernel matrix has structure that affords $< \mathcal{O}(N^2)$ MVM routines. (The notable exception is the work of Cutajar et al. [2016], which uses MVM techniques for standard un-structured GPs.) Our approach, which has a special focus on GPU acceleration, is designed for the general case ($\mathcal{O}(N^2)$ MVMs) as well as for special cases when there is exploitable structure.

3.2 Gaussian Process Training Through Matrix Multiplication

The goal of this chapter is to replace existing training strategies with a unified framework that utilizes modern hardware efficiently. We additionally desire that complex GP models can be used in a blackbox manner without additional training rules. To this end, our method reduces the bulk of GP inference to one of the most efficiently-parallelized computations: *matrix-matrix multiplication*. We call our method Black-Box Matrix × Matrix inference (BBMM) because it only requires a user to specify a matrix multiply routine for the kernel $\widehat{\mathbf{K}}_{\mathbf{XX}}(\cdot)$ and its derivative $\frac{\partial \widehat{\mathbf{K}}_{\mathbf{XX}}}{\partial \boldsymbol{\theta}}(\cdot)$.

Required operations. To train a GP we must compute the marginal log likelihood (Eq. (2.6)) and its derivative (Eq. (2.7)). We rewrite the equations here,

assuming a prior mean of zero for brevity:

$$
\begin{aligned}
-\log p(\mathbf{y} \mid \mathbf{X}, \boldsymbol{\theta}) &\propto \log \left|\widehat{\mathbf{K}}_{\mathbf{XX}}\right| + \mathbf{y}^\top \widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{y}, \\
\frac{\partial -\log p(\mathbf{y} \mid \mathbf{X}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} &\propto \operatorname{Tr}\left(\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\frac{\partial \widehat{\mathbf{K}}_{\mathbf{XX}}}{\partial \boldsymbol{\theta}}\right) - \mathbf{y}^\top \widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\frac{\partial \widehat{\mathbf{K}}_{\mathbf{XX}}}{\partial \boldsymbol{\theta}}\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{y},
\end{aligned}
$$

where again $\widehat{\mathbf{K}}_{\mathbf{XX}}$ is the training kernel matrix plus observational noise ($\widehat{\mathbf{K}}_{\mathbf{XX}} = \mathbf{K}_{\mathbf{XX}} + \sigma^2_{\text{obs}}\mathbf{I}$). These equations have three operations that dominate their time complexity:

(1) the linear solve $\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{y}$,

(2) the log determinant $\log|\widehat{\mathbf{K}}_{\mathbf{XX}}|$, and

(3) the trace term $\operatorname{Tr}(\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\frac{\partial \widehat{\mathbf{K}}_{\mathbf{XX}}}{\partial \boldsymbol{\theta}})$.

The Cholesky decomposition is used in many GP implementations to compute these three terms. This procedure factorizes $\widehat{\mathbf{K}}_{\mathbf{XX}}$ as $\mathbf{L}\mathbf{L}^\top$, where $\mathbf{L}$ is lower triangular. As discussed in Section 2.2.3, computing $\mathbf{L}$ requires $\mathcal{O}(N^3)$ time and $\mathcal{O}(N^2)$ memory. After computing this factorization, matrix solves and log determinants take $\mathcal{O}(N^2)$ and $\mathcal{O}(N)$ time respectively. In general, this asymptotic complexity cannot be reduced even if $\widehat{\mathbf{K}}_{\mathbf{XX}}$ has nice structure (e.g. Toeplitz). Furthermore, its recursive nature makes the Cholesky algorithm less amenable to GPU acceleration since GPUs are designed to parallelize matrix-vector multiplications. Nguyen et al. [2019] aim to alleviate these costs through distributed computing, yet their approach requires quadratic communication costs and quadratic memory.

MVM-based training methods. As described in Section 2.3, $\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{y}$ can also be computed using *conjugate gradients* (CG) [Cunningham et al., 2008, Cutajar

et al., 2016], and the other two quantities can be computed using the Lanczos tridiagonalization algorithm [Ubaru et al., 2017, Dong et al., 2017] or entropic estimates [Fitzsimons et al., 2018]. These MVM-based methods are asymptotically faster and more space efficient than Cholesky based methods [Wilson and Nickisch, 2015, Dong et al., 2017]. Additionally, these methods are able to exploit algebraic structure in the data for further efficiencies [Cunningham et al., 2008, Saatçi, 2012, Wilson and Nickisch, 2015]. We aim to expand on these existing methods. In particular, our goals are to

(1) introduce a unified MVM algorithm that simultaneously computes all terms to improve parallelization/GPU utilization; and

(2) avoid using Lanczos tridiagonalization for the second two terms, as it suffers from numerical instabilities [Golub and Van Loan, 2012].

3.2.1 Modified Batched Conjugate Gradients (mBCG)

For this purpose, we introduce a *modified Batched Conjugate Gradients* (mBCG) algorithm. Standard conjugate gradients takes as input a vector $\mathbf{y}$ and, after J iterations, outputs an approximate solve $\mathbf{c}_J \approx \widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{y}$ (with equality when $J = N$). We modify conjugate gradients to **(1)** perform linear solves with multiple right hand sides simultaneously; and **(2)** return tridiagonal matrices corresponding to partial Lanczos tridiagonalizations of $\widehat{\mathbf{K}}_{\mathbf{XX}}$. Specifically, mBCG takes as input a matrix $\left[\mathbf{y},\ \mathbf{z}^{(1)},\ \cdots,\ \mathbf{z}^{(T)}\right]$, and outputs:

$$\left[\mathbf{c}^{(0)},\ \mathbf{c}^{(1)},\ \cdots,\ \mathbf{c}^{(T)}\right] = \widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\left[\mathbf{y},\ \mathbf{z}^{(1)},\ \cdots,\ \mathbf{z}^{(T)}\right], \quad \mathbf{T}^{(1)}, \ldots, \mathbf{T}^{(T)} \tag{3.1}$$

where $\mathbf{T}^{(1)}, \ldots, \mathbf{T}^{(T)}$ are the partial Lanczos tridiagonalizations of $\widehat{\mathbf{K}}_{\mathbf{XX}}$ with respect to the vectors $\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(T)}$ (see Section 2.3.2).

Using mBCG for GP training. Before describing the details of the mBCG algorithm, we will first discuss how its outputs can be used to compute the three GP training terms: $\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{y}$, $\mathrm{Tr}(\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\frac{\partial\widehat{\mathbf{K}}_{\mathbf{XX}}}{\partial\boldsymbol{\theta}})$, and $\log|\widehat{\mathbf{K}}_{\mathbf{XX}}|$.

(1) $\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{y}$ is equal to $\mathbf{c}^{(0)}$ in Eq. (3.1), directly returned from mBCG.

(2) $\mathrm{Tr}(\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\frac{\partial\widehat{\mathbf{K}}_{\mathbf{XX}}}{\partial\boldsymbol{\theta}})$ can be approximated using *stochastic trace estimation* [Hutchinson, 1990, Fitzsimons et al., 2018], which treats this term as a sum of linear solves. Given i.i.d. random variables $\mathbf{z}^{(1)},\ldots,\mathbf{z}^{(T)}$ so that $\mathbb{E}\left[\mathbf{z}^{(i)}\right] = 0$ and $\mathbb{E}\left[\mathbf{z}^{(i)}\mathbf{z}^{(i)^\top}\right] = \mathbf{I}$, the matrix trace can be written as $\mathrm{Tr}(\mathbf{A}) = \mathbb{E}\left[\mathbf{z}^{(i)^\top}\mathbf{A}\mathbf{z}^{(i)}\right]$. Thus,

$$\begin{aligned}\mathrm{Tr}\left(\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\frac{\partial\widehat{\mathbf{K}}_{\mathbf{XX}}}{\partial\boldsymbol{\theta}}\right) &= \mathbb{E}\left[\mathbf{z}^{(i)^\top}\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\frac{\partial\widehat{\mathbf{K}}_{\mathbf{XX}}}{\partial\boldsymbol{\theta}}\mathbf{z}^{(i)}\right] \\ &\approx \frac{1}{T}\sum_{i=1}^{T}\left(\mathbf{z}^{(i)^\top}\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\right)\left(\frac{\partial\widehat{\mathbf{K}}_{\mathbf{XX}}}{\partial\boldsymbol{\theta}}\mathbf{z}^{(i)}\right) \qquad (3.2)\end{aligned}$$

is an unbiased estimator of the derivative. This computation motivates the $\mathbf{z}^{(1)},\ldots,\mathbf{z}^{(T)}$ terms in Eq. (3.1). A single matrix multiply with the derivative $\frac{\partial\widehat{\mathbf{K}}_{\mathbf{XX}}}{\partial\boldsymbol{\theta}}[\mathbf{z}^{(1)},\ \ldots,\ \mathbf{z}^{(T)}]$ yields the remaining terms.

(3) $\log|\widehat{\mathbf{K}}_{\mathbf{XX}}|$ can be estimated using the stochastic Lanczos quadrature routine of Ubaru et al. [2017], as described in Section 2.3.2. To briefly summarize, this approach approximates the matrix logarithm as

$$(\log\mathbf{A}) \approx \mathbf{z}^{(i)}\mathbf{Q}^{(i)}\left(\log\mathbf{T}^{(i)}\right)\mathbf{Q}^{(i)^\top}\mathbf{z}^{(i)},$$

where $\mathbf{Q}^{(i)}$ and $\mathbf{T}^{(i)}$ are the orthogonal and tridiagonal matrices from Lanczos with initial vector $\mathbf{z}^{(i)}$. Combining stochastic trace estimation with this

approximation gives us

$$\log|\widehat{\mathbf{K}}_{\mathbf{XX}}| = \operatorname{Tr}\left(\log\widehat{\mathbf{K}}_{\mathbf{XX}}\right) \approx \frac{1}{T}\sum_{i=1}^{T}\mathbf{z}^{(i)\top}\mathbf{Q}^{(i)}\left(\log\mathbf{T}^{(i)}\right)\mathbf{Q}^{(i)\top}\mathbf{z}^{(i)}$$
$$= \frac{1}{T}\sum_{i=1}^{T}\|\mathbf{z}^{(i)}\|_2\,\mathbf{e}^{(1)\top}\left(\log\mathbf{T}^{(i)}\right)\mathbf{e}^{(1)},$$

where $\mathbf{e}^{(1)} = [1, 0, , \ldots, 0]$. (The reduction in the second line comes from the orthogonality of $\mathbf{Q}^{(i)}$ with $\mathbf{z}^{(i)}/\|\mathbf{z}^{(i)}\|_2$ being its first column.) Therefore, we can estimate the log determinant of $\widehat{\mathbf{K}}_{\mathbf{XX}}$ simply by computing logarithms of the tridiagonal matrices returned by mBCG in Eq. (3.1).

We note that our derivative and log determinant estimates are also proposed by Cutajar et al. [2016] and Dong et al. [2017], respectively. Notably, Cutajar et al. [2016] does not return a log determinant estimate and therefore their approach cannot be used for sampling $\boldsymbol{\theta}$ or Bayesian model selection. We further differ from Cutajar et al. [2016] in that we use batched operations to compute all terms simultaneously. We differ from Dong et al. [2017] in that we avoid the explicit Lanczos tridiagonalization algorithm (Algorithm 2.4) and thus circumvent its storage and numerical stability issues [Golub and Van Loan, 2012].

Now that we have motivated the terms produced by mBCG, we will present the algorithm itself.

The mBCG algorithm, presented in Algorithm 3.1, makes two changes to the standard conjugate gradients algorithm (Algorithm 2.3). In particular, it performs multiple solves $\mathbf{A}^{-1}\mathbf{B} = \left[\mathbf{A}^{-1}\mathbf{b}^{(0)}, \ \ldots, \ , \mathbf{A}^{-1}\mathbf{b}^{(T)}\right]$ simultaneously using **matrix-matrix multiplication** (MMM), and it also returns Lanczos tridiagonalization matrices $\mathbf{T}^{(1)}, \ \ldots, \ \mathbf{T}^{(T)}$ associated with each of the solves.

Algorithm 3.1: Modified batch conjugate gradients (mBCG). Terms in green represent matrix operations that were originally vector operations in standard PCG (Algorithm 2.3). Terms in red compute the Lanczos tridiagonalization matrices.

Input : $\texttt{mmm\_K}(\cdot)$ – function for matrix-matrix multiplication with matrix $\mathbf{K}$
$\mathbf{B}$ – $N \times (T+1)$ matrix to solve against
$\widehat{\mathbf{P}}^{-1}(\cdot)$ – func. for preconditioner

Output: $\mathbf{C} = \mathbf{K}^{-1}\mathbf{B}$, $\mathbf{T}_1, \ldots, \mathbf{T}_T$ (tridiag. matrices for $\mathbf{b}^{(2)}, \ldots, \mathbf{b}^{(T+1)}$).

$\mathbf{C}_0 \leftarrow \mathbf{0}$ // Current solutions
$\mathbf{R}_0 \leftarrow \texttt{mmm\_K}(\mathbf{C}_0) - \mathbf{B}$ // Current errors
$\mathbf{Z}_0 \leftarrow \widehat{\mathbf{P}}^{-1}(\mathbf{R}_0)$ // Preconditioned errors
$\mathbf{D}_0 \leftarrow \mathbf{Z}_0$ // ''Search'' directions for next solutions
$\mathbf{T}_1, \ldots \mathbf{T}_T \leftarrow 0$ // Tridiag matrices

for $j \leftarrow 1$ **to** J **do**
- $\mathbf{V}_j \leftarrow \texttt{mmm\_K}(\mathbf{D}_{j-1})$
- $\boldsymbol{\alpha}_j \leftarrow (\mathbf{R}_{j-1} \circ \mathbf{Z}_{j-1})^\top \mathbf{1} / (\mathbf{D}_{j-1} \circ \mathbf{V}_j)^\top \mathbf{1}$
- $\mathbf{C}_j \leftarrow \mathbf{C}_{j-1} + \texttt{diag}(\boldsymbol{\alpha}_j)\ \mathbf{D}_{j-1}$
- $\mathbf{R}_j \leftarrow \mathbf{R}_{j-1} - \texttt{diag}(\boldsymbol{\alpha}_j)\ \mathbf{V}_j$
- **if** $\forall i \quad \left\|\mathbf{r}_j^{(i)}\right\|_2 <$ tolerance **then return** $\mathbf{C}_j$;
- $\mathbf{Z}_j \leftarrow \widehat{\mathbf{P}}^{-1}(\mathbf{R}_j)$
- $\boldsymbol{\beta}_j \leftarrow (\mathbf{Z}_j \circ \mathbf{Z}_j)^\top \mathbf{1} / (\mathbf{Z}_{j-1} \circ \mathbf{Z}_{j-1})^\top \mathbf{1}$
- $\mathbf{D}_j \leftarrow \mathbf{Z}_j - \texttt{diag}(\boldsymbol{\beta}_j)\ \mathbf{D}_{j-1}$
- $\forall i \quad T_i^{(j,j)} \leftarrow 1/\boldsymbol{\alpha}_j^{(i)} + \boldsymbol{\beta}_{j-1}^{(i)} / \boldsymbol{\alpha}_{j-1}^{(i)}$
- $\forall i \quad T_i^{(j-1,j)} \leftarrow \sqrt{\boldsymbol{\beta}_{j-1}^{(i)}} / \boldsymbol{\alpha}_j^{(i)}$
- $\forall i \quad T_i^{(j,j-1)} \leftarrow \sqrt{\boldsymbol{\beta}_{j-1}^{(i)}} / \boldsymbol{\alpha}_j^{(i)}$

end
return $\mathbf{C}_J, \mathbf{T}_1, \ldots \mathbf{T}_T$

The majority of the lines in Algorithm 3.1 are direct adaptations of lines from Algorithm 2.3 to handle multiple vectors simultaneously. We denote these lines in green. For example, performing a matrix-matrix multiply $\mathbf{AB}$ is equivalent to performing a matrix-vector multiply $\mathbf{A}\mathbf{b}^{(i)}$ for each column of $\mathbf{B}$. Thus we can replace multiple MVM calls with a single MMM call. In standard PCG, there are two scalar coefficient used during each iteration j: α_j and β_j (see Algorithm 2.3). Note that each solve $\mathbf{c}^{(0)}, \ldots, \mathbf{c}^{(T)}$ in mBCG uses different scalar val-

ues. Therefore, mBCG replaces the scalers with *two coefficient vectors*: $\boldsymbol{\alpha}_j \in \mathbb{R}^{T+1}$ and $\boldsymbol{\beta}_j \in \mathbb{R}^{T+1}$, where each of the vector entries corresponds to a single solve. There are two types of operations involving these coefficients:

(1) updates (e.g. $\boldsymbol{\alpha}_j \leftarrow (\mathbf{R}_{j-1} \circ \mathbf{Z}_{j-1})^\top \mathbf{1} / (\mathbf{D}_{j-1} \circ \mathbf{V}_j)^\top \mathbf{1}$) and

(2) scalaing (e.g. $\mathbf{C}_j \leftarrow \mathbf{C}_{j-1} + \texttt{diag}(\boldsymbol{\alpha}_j)\ \mathbf{D}_{j-1}$).

The update rules are batched versions of the update rules in the standard CG algorithm. For example:

$$\begin{bmatrix} \alpha_j^{(0)} \\ \vdots \\ \alpha_j^{(T)} \end{bmatrix} = \frac{(\mathbf{R}_{j-1} \circ \mathbf{Z}_{j-1})^\top \mathbf{1}}{(\mathbf{D}_{j-1} \circ \mathbf{V}_j)^\top \mathbf{1}} = \begin{bmatrix} \frac{\left(\mathbf{r}_{j-1}^{(0)} \circ \mathbf{z}_{j-1}^{(0)}\right)\mathbf{1}}{\left(\mathbf{d}_{j-1}^{(0)} \circ \mathbf{v}_j^{(0)}\right)\mathbf{1}} \\ \vdots \\ \frac{\left(\mathbf{r}_{j-1}^{(T)} \circ \mathbf{z}_{j-1}^{(T)}\right)\mathbf{1}}{\left(\mathbf{d}_{j-1}^{(T)} \circ \mathbf{v}_j^{(T)}\right)\mathbf{1}} \end{bmatrix} = \begin{bmatrix} \frac{\left(\mathbf{r}_{j-1}^{(0)\top} \mathbf{z}_{j-1}^{(0)}\right)}{\left(\mathbf{d}_{j-1}^{(0)\top} \mathbf{v}_j^{(0)}\right)} \\ \vdots \\ \frac{\left(\mathbf{r}_{j-1}^{(T)\top} \mathbf{z}_{j-1}^{(T)}\right)}{\left(\mathbf{d}_{j-1}^{(T)\top} \mathbf{v}_j^{(T)}\right)} \end{bmatrix},$$

Similarly, for scaling,

$$\begin{aligned} \begin{bmatrix} \mathbf{c}_j^{(0)} & \cdots & \mathbf{c}_j^{(T)} \end{bmatrix} &= \mathbf{C}_{j-1} + \text{diag}(\alpha_j)\mathbf{D}_{j-1} \\ &= \begin{bmatrix} \mathbf{c}_{j-1}^{(0)} & \cdots & \mathbf{c}_{j-1}^{(T)} \end{bmatrix} + \begin{bmatrix} \alpha_j^{(0)}\mathbf{d}_{j-1}^{(0)} & \cdots & \alpha_j^{(T)}\mathbf{d}_{j-1}^{(T)}. \end{bmatrix} \end{aligned}$$

In summary, mBCG is therefore able to perform all solve operations in batch, which enables it to use GPU parallelism.

To compute the Lanczos tridiagonal matrices that correspond to inputs $\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(T)}$, mBCG adapts a technique from Saad [2003]. From Observation 2.1, the diagonal and subdiagonal entries of $\mathbf{T}^{(1)}, \ldots, \mathbf{T}^{(T)}$ are simple deterministic transforms of the $\boldsymbol{\alpha}_j$ and $\boldsymbol{\beta}_j$ coefficients from mBCG. The final three lines in red in Algorithm 3.1 use the $\boldsymbol{\alpha}_j$ and $\boldsymbol{\beta}_j$ coefficients to iteratively compute the Lanczos matrices from Observation 2.1. Notably, these matrices can be formed with $\mathcal{O}(T)$ extra computation, and we are able to avoid running the Lanczos algorithm.

3.2.2 Runtime and Space

As shown above, we are able to approximate all training terms from *a single call to mBCG*. These approximations improve with the number of mBCG iterations. Each iteration requires one matrix multiplication with $\widehat{\mathbf{K}}_{\mathbf{XX}}$ and element-wise operations on $N \times T$ matrices. Therefore, J iterations of mBCG requires $\mathcal{O}(NT)$ space and $\mathcal{O}(J\,\Xi(\widehat{\mathbf{K}}_{\mathbf{XX}}))$ time, where $\Xi(\widehat{\mathbf{K}}_{\mathbf{XX}})$ is the time to multiply $\widehat{\mathbf{K}}_{\mathbf{XX}}$ by a $N \times (T+1)$ matrix. This takes $\mathcal{O}(N^2T)$ time with a standard matrix. It is worth noting that this is a lower asymptotic complexity than standard Cholesky-based training, which is $\mathcal{O}(N^3)$. Therefore, *BBMM offers a computational speedup for exact GPs*. As we will show in Section 3.4, this time complexity can be further reduced with structured data or sparse GP approximations.

After using mBCG to produce the solves and tridiagonal matrices, recovering the three training terms takes little additional time and space. $\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{y}$ requires no additional computation because it is the first output of the algorithm. The $\operatorname{Tr}(\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\frac{\partial \widehat{\mathbf{K}}_{\mathbf{XX}}}{\partial \boldsymbol{\theta}})$ estimate is the inner product of the $\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{z}^{(i)}$ solves with the $\frac{\partial \widehat{\mathbf{K}}_{\mathbf{XX}}}{\partial \boldsymbol{\theta}}[\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(T)}]$ vectors. This only requires an additional $\Xi(\frac{\partial \widehat{\mathbf{K}}_{\mathbf{XX}}}{\partial \boldsymbol{\theta}}) + \mathcal{O}(NT)$ time and $\mathcal{O}(NT)$ space.[1] Computing $\log|\widehat{\mathbf{K}}_{\mathbf{XX}}|$ dominates the post-mBCG running time; however, it is negligible assuming $J \ll N$ iterations of mBCG. To compute the log determinant estimate, we must compute $\mathbf{e}^{(1)^\top} \log \mathbf{T}^{(i)} \mathbf{e}^{(1)}$ for each i, which requires eigendecomposing the $\mathbf{T}^{(i)}$ matrices. This costs $\mathcal{O}(TJ^2)$ time,[2] which again is significantly less than the running time of mBCG.

[1] We assume that $\Xi(\frac{\partial \widehat{\mathbf{K}}_{\mathbf{XX}}}{\partial \boldsymbol{\theta}}) \approx \Xi(\widehat{\mathbf{K}}_{\mathbf{XX}})$, which is true for exact GPs and GP approximations.
[2] By exploiting tridiagonal structure.

3.3 Preconditioning

While each iteration of mBCG performs large parallel matrix × matrix operations, the iterations themselves are sequential. A natural goal for better utilizing hardware is to trade off fewer sequential steps for slightly more effort per step. We accomplish this goal using *preconditioning* [e.g. Demmel, 1997, Saad, 2003, Van der Vorst, 2003, Golub and Van Loan, 2012], which introduces a matrix $\mathbf{P}$ to solve the related linear system

$$\left(\widehat{\mathbf{P}}^{-\frac{1}{2}}\widehat{\mathbf{K}}_{\mathbf{XX}}\widehat{\mathbf{P}}^{-\frac{1}{2}}\right)\mathbf{C} = \widehat{\mathbf{P}}^{-\frac{1}{2}}\left[\mathbf{y},\ ,\mathbf{z}^{(1)},\ ,\ldots,\ \mathbf{z}^{(T)}\right]. \tag{3.3}$$

instead of $\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\left[\mathbf{y},\ ,\mathbf{z}^{(1)},\ ,\ldots,\ \mathbf{z}^{(T)}\right]$. Both systems are guaranteed to have the same solution, but the preconditioned system's convergence depends on the conditioning of $\widehat{\mathbf{P}}^{-\frac{1}{2}}\widehat{\mathbf{K}}_{\mathbf{XX}}\widehat{\mathbf{P}}^{-\frac{1}{2}}$ rather than that of $\widehat{\mathbf{K}}_{\mathbf{XX}}$. Despite the matrix square roots in Eq. (3.3), preconditioned CG/mBCG only need access to $\widehat{\mathbf{P}}$ and its inverse $\widehat{\mathbf{P}}^{-1}$ (see Algorithm 2.3).

3.3.1 Modifying mBCG for Preconditioning

We have to make some special adjustments to BBMM algorithm in order to use preconditioning. In particular, the input to the preconditioned mBCG algorithm should be the vectors

$$\left[\mathbf{y},\ \mathbf{z}^{(1)},\ \cdots,\ \mathbf{z}^{(T)}\right],\quad \mathbf{z}^{(i)} \sim \mathcal{N}\left[\mathbf{0}, \widehat{\mathbf{P}}\right], \tag{3.4}$$

which produces the solves

$$\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\left[\mathbf{y},\ \mathbf{z}^{(1)},\ \cdots,\ \mathbf{z}^{(T)}\right],\quad \mathbf{z}^{(i)} \sim \mathcal{N}\left[\mathbf{0}, \widehat{\mathbf{P}}\right].$$

The difference between Eq. (3.4) and the original (non-preconditioned) input (Eq. (3.1)) is that the $\mathbf{z}^{(i)}$ probe vectors have covariance $\mathbb{E}\left[\mathbf{z}^{(i)}\mathbf{z}^{(i)\top}\right] = \widehat{\mathbf{P}}$ (rather than unit variance). To understand why this is the case, recall from Section 3.2 that our log determinant estimate is given by:

$$\log\left|\widehat{\mathbf{K}}_{\mathbf{XX}}\right| \approx \mathop{\mathbb{E}}_{\mathbf{z}^{(i)}\sim\mathcal{N}[0,\mathbf{I}]}\left[\mathbf{z}^{(i)\top}\mathbf{Q}^{(i)}\left(\log\mathbf{T}^{(i)}\right)\mathbf{Q}^{(i)\top}\mathbf{z}^{(i)}\right],$$

If we precondition mBCG with $\widehat{\mathbf{P}}$, then the $\mathbf{T}^{(i)}$ matrices will correspond to the *preconditioned system* ($\widehat{\mathbf{P}}^{-\frac{1}{2}}\widehat{\mathbf{K}}_{\mathbf{XX}}\widehat{\mathbf{P}}^{-\frac{1}{2}}$) and *preconditioned probe vectors* ($\widehat{\mathbf{P}}^{-\frac{1}{2}}\mathbf{z}^{(i)}$). Consequentially, the stochastic Lanczos quadrature estimate will return

$$\log\left|\widehat{\mathbf{P}}^{-\frac{1}{2}}\widehat{\mathbf{K}}_{\mathbf{XX}}\widehat{\mathbf{P}}^{-\frac{1}{2}}\right| \approx \mathop{\mathbb{E}}_{\mathbf{z}^{(i)}\sim\mathcal{N}[0,\widehat{\mathbf{P}}]}\left[\left(\mathbf{z}^{(i)\top}\widehat{\mathbf{P}}^{-\frac{1}{2}}\right)\mathbf{Q}^{(i)}\left(\log\mathbf{T}^{(i)}\right)\mathbf{Q}^{(i)\top}\left(\widehat{\mathbf{P}}^{-\frac{1}{2}}\mathbf{z}^{(i)}\right)\right]. \quad (3.5)$$

By using $\mathbf{z}^{(i)} \sim \mathcal{N}[\mathbf{0}, \widehat{\mathbf{P}}]$ as probe vectors: the resulting preconditioned vectors $\widehat{\mathbf{P}}^{-\frac{1}{2}}\mathbf{z}^{(i)}$ will be samples from $\mathcal{N}[\mathbf{0}, \mathbf{I}]$, which is the requirement for a stochastic trace estimate.

Estimating $\log|\widehat{\mathbf{K}}_{\mathbf{XX}}|$ from preconditioned mBCG. To compute $\log|\widehat{\mathbf{K}}_{\mathbf{XX}}|$ from Eq. (3.5), we note that

$$\log\left|\widehat{\mathbf{K}}_{\mathbf{XX}}\right| = \log\left|\widehat{\mathbf{P}}^{-\frac{1}{2}}\widehat{\mathbf{K}}_{\mathbf{XX}}\widehat{\mathbf{P}}^{-\frac{1}{2}}\right| + \log\left|\widehat{\mathbf{P}}\right|.$$

We estimate the first term using stochastic Lanczos quadrature (Eq. (3.5)) on the preconditioned system, and then "correct" this estimate with the log determinant of the preconditioner. Note that this will still be an unbiased estimate of $\log|\widehat{\mathbf{K}}_{\mathbf{XX}}|$ if we can exactly compute $\log|\widehat{\mathbf{P}}|$.

Estimating $\mathrm{Tr}(\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}(\partial\widehat{\mathbf{K}}_{\mathbf{XX}}/\partial\boldsymbol{\theta}))$ from preconditioned mBCG. To estimate $\mathrm{Tr}(\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}(\partial\widehat{\mathbf{K}}_{\mathbf{XX}}/\partial\boldsymbol{\theta}))$ from the new probe vectors $\mathbf{z}^{(i)} \sim \mathcal{N}[\mathbf{0}, \widehat{\mathbf{P}}]$, we note that

we can form a stochastic trace estimate from the following:

$$\operatorname{Tr}\left(\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\frac{\partial\widehat{\mathbf{K}}_{\mathbf{XX}}}{\partial\boldsymbol{\theta}}\right) = \operatorname{Tr}\left(\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\frac{\partial\widehat{\mathbf{K}}_{\mathbf{XX}}}{\partial\boldsymbol{\theta}}\underset{\mathbf{z}^{(i)}\sim\mathcal{N}[0,\widehat{\mathbf{P}}]}{\mathbb{E}}\left[\widehat{\mathbf{P}}^{-1}\mathbf{z}^{(i)}\mathbf{z}^{(i)\top}\right]\right)$$
$$\approx \underset{\mathbf{z}^{(i)}\sim\mathcal{N}[0,\widehat{\mathbf{P}}]}{\mathbb{E}}\left[\left(\mathbf{z}^{(i)}\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\right)\left(\frac{\partial\widehat{\mathbf{K}}_{\mathbf{XX}}}{\partial\boldsymbol{\theta}}\,\widehat{\mathbf{P}}^{-1}\mathbf{z}^{(i)}\right)\right]. \tag{3.6}$$

The only differences between Eq. (3.6) and the non-preconditioned trace estimate in Eq. (3.2) are **(1)** we use $\mathbf{z}^{(i)} \sim \mathcal{N}[\mathbf{0}, \widehat{\mathbf{P}}]$ as probe vectors, and **(2)** the derivative term $\partial\widehat{\mathbf{K}}_{\mathbf{XX}}/\partial\boldsymbol{\theta}$ is applied to the vectors $\widehat{\mathbf{P}}^{-1}\mathbf{z}^{(i)}$.

Requirements of mBCG preconditioners. Based on the above discussion, we observe three requirements of any preconditioner $\widehat{\mathbf{P}}$ for mBCG. First, in order to ensure that preconditioning operations do not dominate the running time of Algorithm 3.1, the preconditioner should afford roughly linear-time solves and linear space. Second, we should be able to efficiently compute the log determinant of the preconditioner $\log|\widehat{\mathbf{P}}|$ to "correct" the log determinant estimate in Eq. (3.5). Finally, we should be able to efficiently sample probe vectors $\mathbf{z}^{(i)}$ from the distribution $\mathcal{N}[\mathbf{0}, \widehat{\mathbf{P}}]$.

3.3.2 The Partial Pivoted Cholesky Preconditioner for mBCG

For one possible preconditioner, we turn to the **partial pivoted Cholesky decomposition** (as introduced in Section 2.2.3). The pivoted Cholesky algorithm allows us to compute a rank-R approximation ($R \ll N$) of the training kernel matrix $\mathbf{K}_{\mathbf{XX}} \approx \bar{\mathbf{L}}_R\bar{\mathbf{L}}_R^\top$. Our mBCG preconditioner will be

$$\widehat{\mathbf{P}}_R = \bar{\mathbf{L}}_R\bar{\mathbf{L}}_R^\top + \sigma_{\text{obs}}^2\mathbf{I}, \tag{3.7}$$

where σ^2_{obs} is the Gaussian likelihood's noise term. Intuitively, if $\bar{\mathbf{L}}_R\bar{\mathbf{L}}_R^\top$ is a good low-rank approximation of $\mathbf{K_{XX}}$, then $(\bar{\mathbf{L}}_R\bar{\mathbf{L}}_R^\top + \sigma^2_{\text{obs}}\mathbf{I})^{-1}\widehat{\mathbf{K}}_{\mathbf{XX}} \approx \mathbf{I}$.

Unlike the standard Cholesky decomposition, which computes an exact factorization in N iterations, the partial pivoted Cholesky decomposition produces a *rank-R* factorization in $R \ll N$ iterations, and therefore does not share its asymptotic concerns. Moreover, it meets the requirements outlined above:

Observation 3.1 (Properties of the rank-R pivoted Cholesky preconditioner)**.**

(1) *$\bar{\mathbf{L}}_R$ can be computed in $\mathcal{O}(\texttt{row\_K}(\widehat{\mathbf{K}}_{\mathbf{XX}})\, R^2)$ time, where $\texttt{row\_K}(\widehat{\mathbf{K}}_{\mathbf{XX}})$ is the time required to retrieve a single row of $\widehat{\mathbf{K}}_{\mathbf{XX}}$ (see Section 2.2.3).*

(2) *Storing $\bar{\mathbf{L}}_R$ requires $\mathcal{O}(NR)$ memory.*

(3) *Linear solves with $\widehat{\mathbf{P}}_R = \bar{\mathbf{L}}_R\bar{\mathbf{L}}_R^\top + \sigma^2_{obs}\mathbf{I}$ can be performed in $\mathcal{O}(NR^2)$ time using the Woodbury matrix formula.*[3]

(4) *The log determinant of $\widehat{\mathbf{P}}_R$ can be computed in $\mathcal{O}(NR^2)$ time using the matrix determinant lemma.*[4]

(5) *Samples from $\mathcal{N}[\mathbf{0}, \widehat{\mathbf{P}}_R]$ can be drawn with $\mathcal{O}(NR^2)$ computation using the reparameterization trick [Kingma and Welling, 2014].*[5]

---

[3]The Woodbury matrix formula is a $\mathcal{O}(R^2N)$ formula for "rank-R plus diagonal" solves:

$$\left(\bar{\mathbf{L}}_R\bar{\mathbf{L}}_R^\top + \sigma^2_{\text{obs}}\mathbf{I}\right)^{-1}\mathbf{b} = \sigma^{-2}_{\text{obs}}\mathbf{b} - \sigma^{-4}_{\text{obs}}\,\bar{\mathbf{L}}_R\left(\mathbf{I} - \sigma^{-2}_{\text{obs}}\bar{\mathbf{L}}_R^\top\bar{\mathbf{L}}_R\right)^{-1}\bar{\mathbf{L}}_R^\top\mathbf{b}.$$

[4]The matrix determinant lemma is an analog of the Woodbury formula for determinants:

$$\log|\bar{\mathbf{L}}_R\bar{\mathbf{L}}_R^\top + \sigma^2_{\text{obs}}\mathbf{I}| = \log|\mathbf{I} - \sigma^{-2}_{\text{obs}}\bar{\mathbf{L}}_R^\top\bar{\mathbf{L}}_R| + 2N\log\sigma_{\text{obs}}.$$

[5]Draw standard normal vectors $\boldsymbol{\epsilon}'_1 \in \mathbb{R}^R$ and $\boldsymbol{\epsilon}'_2 \in \mathbb{R}^N$. By the reparameterization trick, $\left(\bar{\mathbf{L}}_R\boldsymbol{\epsilon}'_1 + \sigma_{\text{obs}}\boldsymbol{\epsilon}'_2\right)$ is a sample from $\mathcal{N}[\mathbf{0}, (\bar{\mathbf{L}}_R\bar{\mathbf{L}}_R^\top + \sigma^2_{\text{obs}}\mathbf{I})]$.

Assuming $R \ll N$ (for example, $R \approx 5$), computing and using $\widehat{\mathbf{P}}$ is less expensive than a single matrix multiplication with $\widehat{\mathbf{K}}_{\mathbf{XX}}$. While the $R \approx 5$ iterations required to compute $\bar{\mathbf{L}}_R$ are inherently sequential, we note that this is far fewer iterations than the standard Cholesky factorization.

Perhaps more important than its runtime are its convergence properties. In general, the low-rank nature of $\widehat{\mathbf{P}}$ is an ideal choice for many kernel matrices with rapidly decaying spectra. Such kernels tend to be horribly conditioned yet are well approximated by low rank matrices. We empirically demonstrate in Section 3.5 that $\widehat{\mathbf{P}}$ significantly improves the convergence of mBCG for several kernels. Below we discuss theoretical guarantees for certain classes of kernels.

Theoretical analysis. Kernels with rapidly decaying eigenvalues (i.e. kernels that are well approximated by low-rank matrices) will see the largest improvements from the partial pivoted Cholesky preconditioner. Based on the work of Harbrecht et al. [2012], we can prove the following lemma about kernel condition numbers:

Lemma 3.1. *Let $\bar{\mathbf{L}}_R$ be the rank-R pivoted Cholesky factor of kernel matrix $\mathbf{K}_{\mathbf{XX}} \in \mathbb{R}^{N \times N}$. If the first R eigenvalues $\lambda_1, \ldots, \lambda_R$ of $\mathbf{K}_{\mathbf{XX}}$ satisfy*

$$4^i \lambda_i \leq \mathcal{O}\left(e^{-Bi}\right), \quad i \in \{1, \ \ldots, \ R\}, \tag{3.8}$$

for some $B > 0$, then the condition number $\kappa(\widehat{\mathbf{P}}^{-1}\widehat{\mathbf{K}}_{\mathbf{XX}}) \triangleq \|\widehat{\mathbf{P}}_k^{-1}\widehat{\mathbf{K}}_{\mathbf{XX}}\|_2 \|\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\widehat{\mathbf{P}}_k\|_2$ satisfies the following bound:

$$\kappa\left(\widehat{\mathbf{P}}^{-1}\widehat{\mathbf{K}}_{\mathbf{XX}}\right) \leq \left(1 + \mathcal{O}\left(\sigma_{obs}^{-2} N e^{-BR}\right)\right)^2$$

where $\widehat{\mathbf{P}} = \left(\bar{\mathbf{L}}_R \bar{\mathbf{L}}_R^\top + \sigma_{obs}^2 \mathbf{I}\right)$ and $\widehat{\mathbf{K}}_{\mathbf{XX}} = (\mathbf{K}_{\mathbf{XX}} + \sigma_{obs}^2 \mathbf{I})$.

(See Appendix A.1 for a proof.) It so happens that the exponentially-decaying eigenvalue assumption actually holds for certain classes of kernels. For example, the eigenvalues of univariate RBF kernels are guaranteed to decay *super-exponentially* (see Appendix A.2). In our experiments we observe improved conditioning for other kernels as well (Section 3.5).

Using Lemma 3.1, we can prove the following statements about the solves/log determinant estimates from preconditioned mBCG:

Theorem 3.1 (Convergence of solves from preconditioned mBCG)**.** *Let* $\mathbf{K_{XX}} \in \mathbb{R}^{N \times N}$ *be a* $N \times N$ *kernel that satisfies the eigenvalue condition of Eq.* (3.8)*, and let* $\bar{\mathbf{L}}_R$ *be its rank-*R *pivoted Cholesky factor. After* J *iterations of mBCG with preconditioner* $\widehat{\mathbf{P}} = (\bar{\mathbf{L}}_R \bar{\mathbf{L}}_R^\top + \sigma_{obs}^2 \mathbf{I})$*, the difference between* $\mathbf{c}_J$ *and true solution* $\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{y}$ *is bounded by:*

$$\left\| \widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{y} - \mathbf{c}_J \right\|_{\widehat{\mathbf{K}}_{\mathbf{XX}}} \leq \left[\frac{1}{1 + \mathcal{O}(\sigma_{obs}^2 e^{RB}/N)} \right]^J \left\| \widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{y} \right\|_{\widehat{\mathbf{K}}_{\mathbf{XX}}},$$

where $\widehat{\mathbf{K}}_{\mathbf{XX}} = (\mathbf{K_{XX}} + \sigma_{obs}^2 \mathbf{I})$ *and* $B > 0$ *is a constant.*

Theorem 3.2 (Convergence of log determinants from preconditioned mBCG)**.** *Assume* $\mathbf{K_{XX}} \in \mathbb{R}^{N \times N}$ *satisfies the eigenvalue condition of Eq.* (3.8)*. Suppose we estimate* $\Gamma \approx \log |\widehat{\mathbf{P}}^{-1}\widehat{\mathbf{K}}_{\mathbf{XX}}|$ *using Eq.* (3.5) *with:*

- $J \geq \mathcal{O}\left[(1 + \sigma_{obs}^{-2} N e^{-BR}) \log\left((1 + \sigma_{obs}^{-2} N e^{-BR})/\epsilon\right)\right]$ *iterations of mBCG (for some constant* $B > 0$*), and*
- $T \geq \frac{32}{\epsilon^2} \log\left(\frac{2}{\delta}\right)$ *random* $\mathbf{z}^{(i)} \sim \mathcal{N}[\mathbf{0}, \widehat{\mathbf{P}}]$ *vectors.*

Then the error of the stochastic Lanczos quadrature estimate Γ *is probabilistically bounded by:*

$$Pr\left[\left|\log |\widehat{\mathbf{P}}^{-1}\widehat{\mathbf{K}}_{\mathbf{XX}}| - \Gamma\right| \leq \epsilon N\right] \geq (1 - \delta).$$

(See Appendix A.1 for proofs.) Theorem 3.1 implies that the mBCG solves—used to compute both $\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{y}$ and $\mathrm{Tr}(\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}(\partial\widehat{\mathbf{K}}_{\mathbf{XX}}/\partial\boldsymbol{\theta}))$—will converge *exponentially* quicker as the rank of the partial pivoted Cholesky decomposition increases. Theorem 3.2 implies that the number of iterations needed to accurately estimate $\log|\widehat{\mathbf{P}}^{-1}\widehat{\mathbf{K}}_{\mathbf{XX}}|$ also decreases quickly as R increases. Furthermore, in the limit as $R \to N$ we have that $\log|\widehat{\mathbf{K}}_{\mathbf{XX}}| = \log|\widehat{\mathbf{P}}|$. Since our calculation of $\log|\widehat{\mathbf{P}}|$ is exact, the final estimate of $\log|\widehat{\mathbf{K}}_{\mathbf{XX}}|$ will have less stochasticity.

Related work. Cutajar et al. [2016] explore preconditioned conjugate gradients for GP training, using various sparse GP methods (as well as some classical methods) as preconditioners. Bach [2013] uses the pivoted Cholesky decomposition as a low-rank approximation to kernel matrices. However, Bach [2013] treats this decomposition as an approximate training method, whereas we use the decomposition primarily as a preconditioner and thus avoid any loss of accuracy from the low rank approximation.

3.4 Programmability with BBMM

We have discussed how the BBMM framework is more hardware efficient than existing training methods and avoids numerical instabilities with Lanczos. Another key advantage of BBMM is that it can easily be adapted to complex GP models or structured GP approximations.

Indeed BBMM is *blackbox* by nature, only requiring a routine to perform matrix-multiplications with the kernel matrix and its derivative. Here we provide examples of how existing GP models and scalable approximations can be

easily implemented in this framework. The implementations of many specialty models require at most *50 lines of Python code*.

3.4.1 GPyTorch's `LazyTensor` Construct

In GPyTorch, we use the construct of a `LazyTensor` object (or lazily-evaluated tensor) to represent kernel matrices $\mathbf{K_{XX}}$. A `LazyTensor` represents a (potentially) structured matrix $\mathbf{K} \in \mathbb{R}^{N \times N}$ that is defined through some $\leq \mathcal{O}(N^2)$ representation $\mathbf{r}$. As an example, a diagonal matrix $\mathbf{K}$ can be represented by its diagonal vector. Consequentially, the representation $\mathbf{r}$ for the `DiagLazyTensor` class is simply a $\mathbb{R}^N$ vector of diagonal entries.

Each `LazyTensor` sub-class defines a `_matmul(·)` method (for performing $\mathbf{K}(\mathbf{B})$ for some matrix $\mathbf{B}$) and a `_deriv(·,·)` method for computing $\partial \operatorname{Tr}(\mathbf{A}^\top \mathbf{K} \mathbf{B}) / \partial \mathbf{r}$, where $\mathbf{A}$ and $\mathbf{B}$ are arbitrary matrices. The `_matmul(·)` method is used by mBCG, exploiting any structure of the `LazyTensor` for fast matrix multiplication. The `_deriv(·,·)` function can be used to compute an unbiased estimate of Eq. (2.7):

- The $\frac{1}{T} \sum_{i=1}^{T} \left(\mathbf{z}^{(i)\top} \widehat{\mathbf{K}}_{\mathbf{XX}}^{-1} \right) \left(\frac{\partial \widehat{\mathbf{K}}_{\mathbf{XX}}}{\partial \boldsymbol{\theta}} \mathbf{z}^{(i)} \right)$ term, which approximates the trace term in Eq. (2.7), can be rewritten as

  $$\frac{1}{T} \frac{\partial \operatorname{Tr}((\mathbf{Z}^\top \widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}) \widehat{\mathbf{K}}_{\mathbf{XX}}(\mathbf{Z}))}{\partial \mathbf{r}} \frac{\partial \mathbf{r}}{\partial \boldsymbol{\theta}},$$

  where $\mathbf{Z} = [\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(T)}]$. This can be computed by calling `_deriv(`$\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{Z}$`,` $\mathbf{Z}$`)`.

- The $(\mathbf{y}^\top \widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}) \frac{\partial \widehat{\mathbf{K}}_{\mathbf{XX}}}{\partial \boldsymbol{\theta}} (\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1} \mathbf{y})$ term in Eq. (2.7) can be rewritten as

  $$\frac{\partial (\mathbf{y}^\top \widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}) \widehat{\mathbf{K}}_{\mathbf{XX}} (\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1} \mathbf{y})}{\partial \mathbf{r}} \frac{\partial \mathbf{r}}{\partial \boldsymbol{\theta}},$$

which can be computed by calling `_deriv(`$\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{y}$`, `$\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{y}$`)`.

While all `LazyTensor` subclasses define `_matmul(·)` and `_deriv(·, ·)`, the superclass defines additional methods necessary for GP training, such as

- `inv_matmul(·)` (for computing matrix solves $\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{y}$)
- `logdet()` (for computing the log determinant $\log|\widehat{\mathbf{K}}_{\mathbf{XX}}|$)
- `inv_quad_logdet(·)` (for simultaneously computing $\mathbf{y}^\top\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{y}$ and $\log|\widehat{\mathbf{K}}_{\mathbf{XX}}|$).

Each of these methods runs the mBCG algorithm using the subclass' `_matmul(·)` method. Moreover, each method uses `_deriv(·,·)` in conjunction with PyTorch's autograd [Paszke et al., 2017] to compute the appropriate derivatives. In GPyTorch, we use the `inv_quad_logdet(·)` method to compute Eq. (2.6)—as it can compute all log likelihood and derivative terms with a single mBCG call.

For standard GP regression (with no approximations), there is no exploitable structure for the kernel matrix. We therefore use the aptly named `NonLazyTensor`, which simply represents an arbitrary matrix—i.e. $\mathbf{r} = \mathbf{K}_{\mathbf{XX}}$. The `_matmul(·)` and `_deriv(·, ·)` routines are straightforward:

```
class NonLazyTensor(LazyTensor):
    def __init__(self, matrix):
        self.matrix = matrix

    def _matmul(self, B):
        return self.matrix @ B

    def _deriv(self, A, B):
        # d(tr(A^T K B)) / d(K) = A B^T
        return A @ B.transpose(-1, -2)
```

For a full GP code example that uses `LazyTensors` under the hood, see Appendix D.1 or `http://github.com/cornellius-gp/gpytorch`.

3.4.2 Examples of `LazyTensor`s and Specialty GP Models

Bayesian linear regression can be viewed as GP regression with the special kernel matrix

$$\mathbf{K}_{\mathbf{XX}}^{(\text{lin})} = \mathbf{X}\mathbf{X}^\top.$$

A matrix multiply with this kernel against an $N \times T$ matrix $\mathbf{B}$, $(\mathbf{X}\mathbf{X}^\top)\mathbf{V}$ requires $\mathcal{O}(TND)$ time. Therefore, BBMM requires $\mathcal{O}(JTND)$ time, and is exact in $\mathcal{O}(TND^2)$ time. This running time complexity matches existing efficient algorithms for Bayesian linear regression, *with no additional derivation*.

In GPyTorch's, the `LinearKernel` class outputs a `RootLazyTensor`:

```
class RootLazyTensor(LazyTensor):
    """ The output of gpytorch.kernels.LinearKernel """
    def __init__(self, X):
        self.X = X  # X is a (N X D) matrix

    def _matmul(self, B):
        # If B is a (N X T) matrix, this is O(T N D)
        return (self.X @ self.X.transpose(-1, -2) @ B

    def _deriv(self, A, B):
        # d(tr(A^T K B)) / d(X) = A B^T X + B A^T X
        # if A and B are (N X T), this is O(T N D)
        return A @ B.transpose(-1, -2) @ self.X + \
            B @ A.transpose(-1, -2) @ self.X
```

Note that the `RootLazyTensor`'s `_matmul(·)` function encodes the efficient MVM necessary for $\mathcal{O}(JTND)$ inference.

Multi-task Gaussian processes [Bonilla et al., 2008] use a kernel function that combines covariance between inputs $\mathbf{x}, \mathbf{x}'$ and covariance between tasks c, c': $k^{(\text{input})}(\mathbf{x}, \mathbf{x}')\ k^{(\text{task})}(c, c')$. Constructing the kernel matrix over all training inputs $\mathbf{X}$ and tasks $[1, C]$ results in Kronecker structure:

$$\mathbf{K}^{(\text{multi})}_{\mathbf{XX},\mathbf{cc}} = \mathbf{K}^{(\text{task})}_{\mathbf{XX}} \otimes \mathbf{K}^{(\text{task})}_{\mathbf{cc}},$$

where $\otimes$ represents the Kronecker product. Though $\mathbf{K}^{(\text{multi})}_{\mathbf{XX},\mathbf{cc}}$ is a $(NC) \times (NC)$ matrix (for C tasks), exploiting the Kronecker structure results in $\mathcal{O}(NC^2 + N^2C)$ matrix-vector multiplications. (This can be reduced further if $\mathbf{K}^{(\text{input})}_{\mathbf{XX}}$ is structured and affords fast multiplications.) Thus, multi-task Gaussian processes can be implemented simply by using a Kronecker matrix multiplication routine, with a time complexity that matches state-of-the-art [Bonilla et al., 2008].

The `MultitaskKernel` object in GPyTorch returns a `KroneckerProductLazyTensor`. (The forward and backward pass methods are adopted from Saatçi [2012].) Here, we assume that the $\mathbf{K}^{(\text{input})}_{\mathbf{XX}}$ and $\mathbf{K}^{(\text{task})}_{\mathbf{cc}}$ matrices are themselves represented as `LazyTensors`.

```
class KroneckerProductLazyTensor(LazyTensor):
    """ The output of gpytorch.kernels.MultitaskKernel """
    def __init__(self, K_input, K_task):
        self.K_input = K_input  # a (N X N) LazyTensor
        self.K_task = K_task  # a (C X C) LazyTensor

    def _matmul(self, B):
        # Kronecker prod, adapted from Saatci, 2012
        # O(N C^2 + N C^2), less if inputs are structured
        # B is (CN X T)
        out = B.clone().view(self.K_task.size(-1), -1)  # C X NT
        out = self.K_task._matmul(out)
        out = out.view(self.K_task.size(-1), -1, B.size(-1))
        out = out.transpose(-3, -2)  # N X C X T
        out = out.view(self.K_input.size(-1), -1)  # N X CT
        out = self.K_input._matmul(out)
        out = out.view(-1, B.size(-1))  # CN X T
        return out

    def _deriv(self, A, B):
        # Kronecker deriv
        # O(N C^2 + N C^2), less if inputs are structured
        # See Saatci, 2012, or github.com/cornellius-gp/gpytorch
```

Importantly, implementing multitask GPs in GPyTorch simply requires changing the kernel module (which changes the `LazyTensor`)—no additional implementation is required. This is demonstrated by the code example in Appendix D.2.

Compositions of kernels can often be handled automatically. For example, given a BBMM routine for $\mathbf{K}_1, \mathbf{K}_2, \mathbf{K}_3$, we can automatically perform $(\mathbf{K}_1\mathbf{K}_2 + \mathbf{K}_3)\mathbf{B} = \mathbf{K}_1(\mathbf{K}_2\mathbf{B}) + \mathbf{K}_3\mathbf{B}$. To handle the sum of two kernels, we can define a `SumLazyTensor` that takes two `LazyTensor`s and distributes their matrix-multiplication. For the product of kernels, one can use the method of Gardner et al. [2018b].

```
class SumLazyTensor(LazyTensor):
    def __init__(self, *Ks):
        self.Ks = Ks  # list of (N X N) LazyTensors

    def _matmul(self, B):
        return sum(K._matmul(B) for K in self.Ks)

    def _deriv(self, A, B):
        return [K._deriv(A, B) for K in self.Ks]
```

Sparse Gaussian Process Regression (SGPR) [Titsias, 2009] and many other sparse GP techniques [Quiñonero-Candela and Rasmussen, 2005, Snelson and Ghahramani, 2006, Hensman et al., 2013] use the subset of regressors (SoR) approximation for the kernel (see Section 2.1.5):

$$\mathbf{K}_{\mathbf{XX}}^{(\text{SGPR})} = \mathbf{K}_{\mathbf{XZ}}\mathbf{K}_{\mathbf{ZZ}}^{-1}\mathbf{K}_{\mathbf{ZX}}.$$

Performing a matrix-matrix multiply with this matrix requires $\mathcal{O}(TNM + TM^3)$ time by distributing the vector multiply and grouping terms correctly. This computation is *asymptotically faster* than the $\mathcal{O}(NM^2 + M^3)$ time required by Cholesky based inference. Augmenting the SoR approximation with a diagonal correction, e.g. as in FITC [Snelson and Ghahramani, 2006], is similarly straightforward. (We omit the code example here: see `http://github.com/cornellius-gp/gpytorch`.)

KISS-GP [Wilson and Nickisch, 2015]—see Section 2.1.6—also known as SKI, is an inducing point method designed to provide fast matrix vector multiplies (MVMs) for use with Krylov subspace methods. KISS-GP is thus a natural candidate for BBMM and can benefit greatly from hardware acceleration. The KISS-GP approximation applied to the training covariance matrix gives us

$$\mathbf{K}_{\mathbf{XX}}^{(\text{KISS-GP})} = \mathbf{W}_{\mathbf{X}}^{\top}\mathbf{K}_{\mathbf{ZZ}}\mathbf{W}_{\mathbf{X}}.$$

where $\mathbf{W}$ is a $\mathcal{O}(N)$ sparse interpolation matrix and $\mathbf{K_{ZZ}}$ has Toeplitz structure for $\mathcal{O}(M \log M)$ matrix-vector multiplies. Thus KISS-GP provides $\mathcal{O}(TN + TM \log M)$ matrix-matrix multiplies. (We omit the code example here.)

3.4.3 `LazyTensor`s and Pivoted Cholesky Preconditioning

In order to compute the partial pivoted Cholesky preconditioner for arbitrary `LazyTensors`, we need routines for **(1)** computing the row of a matrix, and **(2)** computing the matrix diagonal. Each `LazyTensor` can optionally implement its own `_get_row(·)` method; however, a simple default implementation uses the `_matmul` function: $\mathbf{k}^{(i)} = \mathbf{K_{XX}}\mathbf{e}_i$ ($\mathbf{e}_i$ is the i^{th} unit vector). There is no standard default for the `diagonal()` function; however, they tend to be straightforward to write. We include some examples for various `LazyTensor` sub-classes below.

```
# For NonLazyTensor
def diagonal(self):
    return K.diagonal(dim1=-1, dim2=-2)

# For RootLazyTensor
def diagonal(self):
    # diag(X X^T) = (X \cdot X) e_1
    # where \cdot is elementwise multiplication
    return self.X.pow(2).sum(dim=-1)

# For KroneckerProductLazyTensor
def diagonal(self):
    # A neat trick: the KP-diagonal is a flattened
    # outer product of the K_input/K_task diagonals
    return torch.ger(
        self.K_input.diagonal(), self.K_task.diagonal(),
    ).view(-1)

# For SumLazyTensor
def diagonal(self):
    return sum(K.diag for K in self.Ks)
```

3.5 Results

We evaluate the BBMM framework, demonstrating:

(1) the BBMM inference engine provides a substantial speed benefit over Cholesky based inference and standard MVM-based CG inference, especially for GPU computing;

(2) BBMM achieves comparable or better test error than Cholesky inference; and

(3) preconditioning provides a substantial improvement in the efficiency of our approach.

Baseline methods. We test BBMM on three types of GPs: **(1) Exact** GP models; **(2) SGPR** models [Titsias, 2009]; and **(3) KISS-GP** models with deep kernels [Wilson and Nickisch, 2015, Wilson et al., 2016a]. For Exact GPs and SGPR models, we compare BBMM against Cholesky-based training (as implemented in GPFlow [Matthews et al., 2017]). Since KISS-GP is not intended for Cholesky inference, we compare BBMM to the inference procedure of Dong et al. [2017]. This procedure differers from BBMM in that it computes $\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{y}$ without a preconditioner and computes $\log|\widehat{\mathbf{K}}_{\mathbf{XX}}|$ and its derivative with the Lanczos algorithm.

Datasets. We test Exact GPs on five datasets from the UCI dataset repository [Asuncion and Newman, 2007] that have up to $N = 3{,}500$ training examples. We test SGPR on larger datasets (N up to $50{,}000$). For KISS-GP we test five large UCI datasets (N up to $515{,}000$).

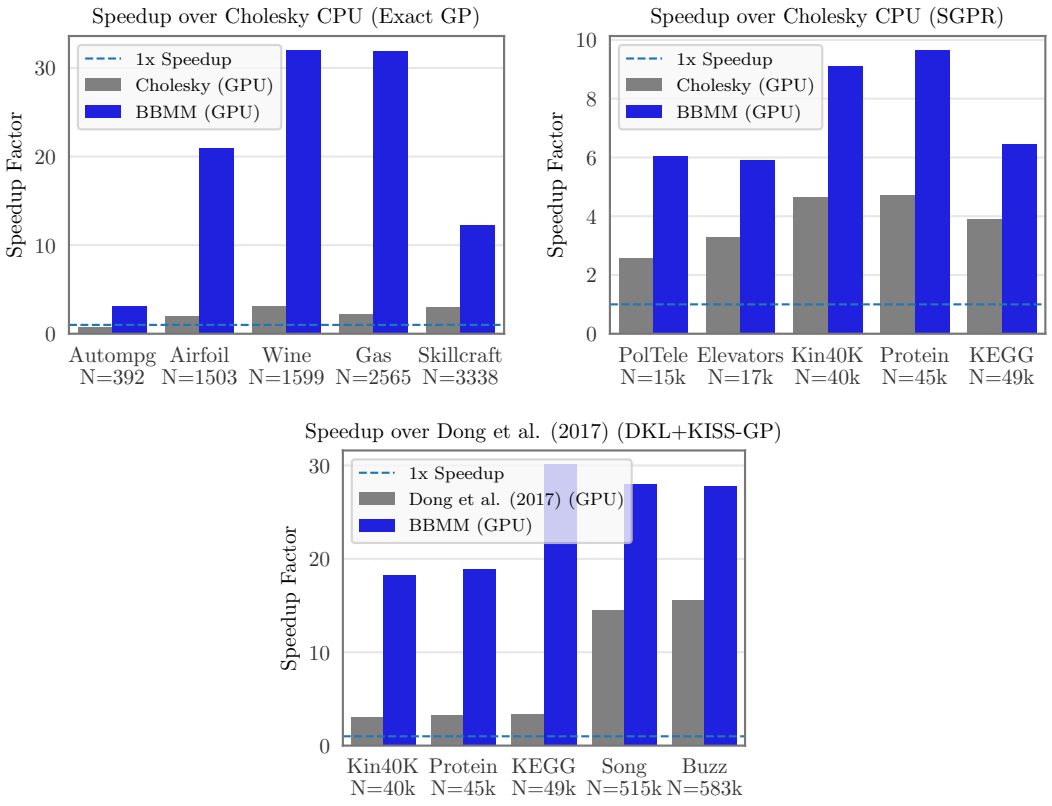

Figure 3.1: Speedup of GPU-accelerated GP training. BBMM is in blue; competing GPU methods are in gray. **Left:** Exact GP speedup over CPU Cholesky-based training. **Middle:** SGPR [Titsias, 2009, Hensman et al., 2013] speedup over CPU Cholesky-based training. **Right:** KISS-GP+DKL [Wilson and Nickisch, 2015, Wilson et al., 2016a] speedup over CPU training of Dong et al. [2017].

Experiment details. All methods use the Adam optimizer [Kingma and Ba, 2015] with identical hyperparameters. In BBMM experiments we use rank $R=5$ pivoted Cholesky preconditioners unless otherwise stated. We use a maximum of $J = 20$ iterations of CG for each solve, and we use $T = 10$ probe vectors filled with Gaussian random variables to estimate the log determinant and trace terms. SGPR models use $M = 300$ inducing points. KISS-GP models use $M = 10{,}000$ inducing points and the deep kernels described in [Wilson et al., 2016a]. All speed experiments are run on an Intel Xeon E5-2650 CPU and a NVIDIA Titan XP GPU.

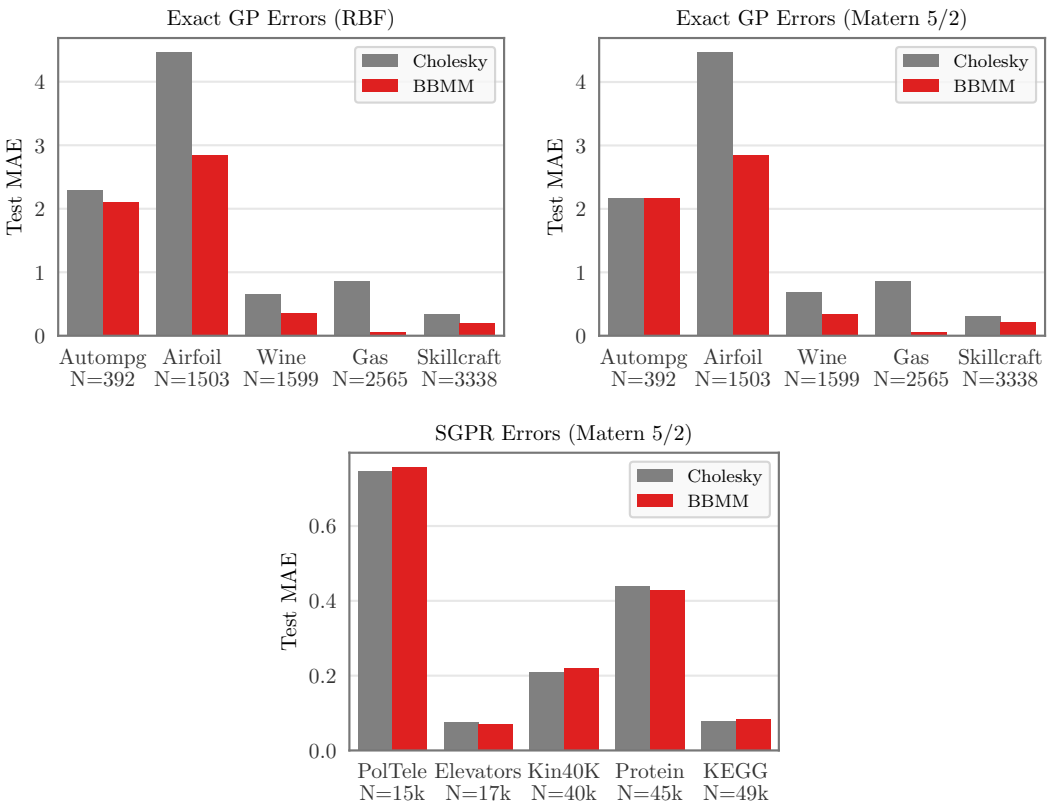

Figure 3.2: Predictive error comparison of mBCG versus Cholesky (mean average error). The left two plots compare errors of Exact GPs with RBF and Matérn-5/2 kernels, and the final plot compares error of SGPR models with a Matérn-5/2 kernel.

Speed comparison. Fig. 3.1 shows the speedup obtained by GPU-accelerated BBMM over CPU-based training methods (Cholesky for Exact/SGPR, Dong et al. [2017] for KISS-GP). BBMM is up to *32 times faster* than Exact/KISS-GP CPU training, and up to 10 times faster than SGPR CPU training. The largest speedups occur on the biggest datasets, since smaller datasets experience larger GPU overhead. Notably, BBMM achieves a much larger speedup than GPU accelerated Cholesky methods (Exact, SGPR), which only achieve a roughly $4\times$ speedup. This result underscores the fact that Cholesky methods are not as well suited for GPU acceleration. For KISS-GP models, BBMM performs better than the GPU-accelerated method of Dong et al. [2017]. This speedup is because

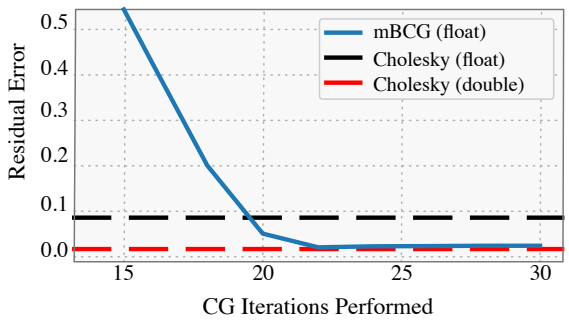

Figure 3.3: Solve error of mBCG versus Cholesky.

BBMM is able to calculate all inference terms in parallel, while Dong et al. [2017] computes the terms in series.

Predictive error comparison. Computing predictive means requires the solve $\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{y}$. Therefore, the PCG algorithm can be used to compute this term with preconditioning and GPU acceleration. In Fig. 3.2 we compare the mean average error (MAE) of BBMM models versus Cholesky models. We demonstrate results using both the RBF kernel and a Matérn-5/2 kernel. Across all datasets, our method is at least as accurate in terms of final test MAE. On a few datasets (e.g. Gas, Airfoil, and Wine with Exact GPs) BBMM even improves final test error. The Cholesky decomposition is known to have numerical issues resulting from extremely small eigenvalues. For example, Cholesky methods frequently add noise (or "jitter") to the diagonal of the kernel matrix for numerical stability. It is possible to reduce the numerical instabilities with double precision (see Fig. 3.3); however, this requires an increased amount of computation. BBMM on the other hand avoids adding this noise, without requiring double precision.

Preconditioning. To demonstrate the effectiveness of preconditioning, we train deep RBF and deep Matérn-5/2 kernels on two datasets (Protein and

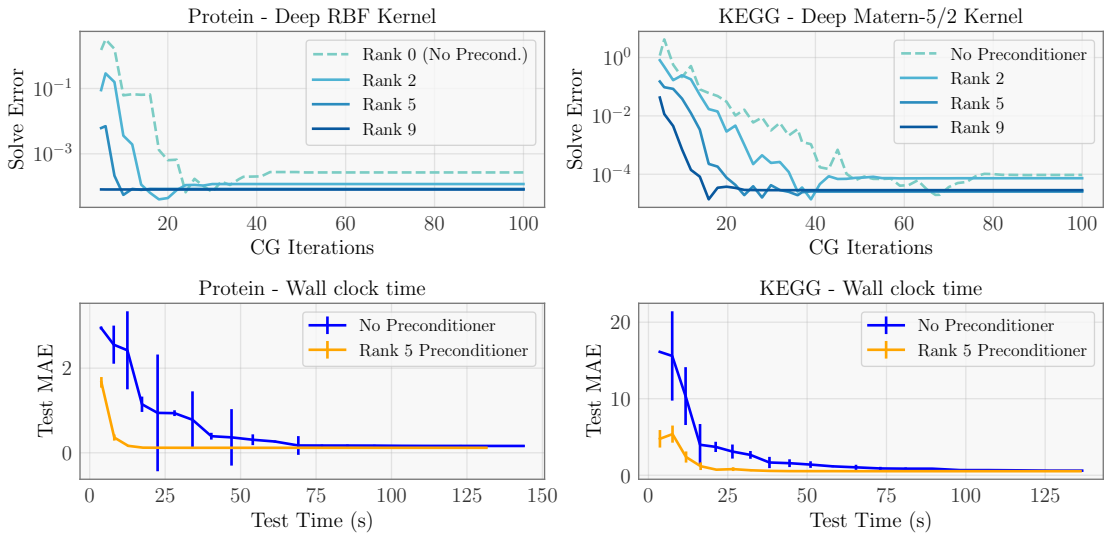

Figure 3.4: Effect of partial pivoted Cholesky preconditioning. **Top:** mBCG residual $\|\widehat{\mathbf{K}}_{\mathbf{XX}}\mathbf{c} - \mathbf{y}\|/\|\mathbf{y}\|$ as a function of mBCG iterations. **Bottom:** Test set mean average error (MAE) as a function of mBCG wall-clock time. Solves are computed using no preconditioner, rank $R = 2$, $R = 5$, and $R = 9$ pivoted Cholesky preconditioners using deep RBF and deep Matérn kernels. On the 2 datasets tested (Protein and KEGG), preconditioning accelerates convergence.

KEGG) and evaluate the solve error of mBCG. We measure the relative residual $\|\widehat{\mathbf{K}}_{\mathbf{XX}}\mathbf{c} - \mathbf{y}\|/\|\mathbf{y}\|$ as a function of the number of mBCG iterations performed. We compare using no preconditioner, as well as rank $R = 2$, $R = 5$, and $R = 9$ partial pivoted Cholesky preconditioners. The results are in the top of Fig. 3.4. As expected based on our theoretical intuitions, increasing the rank of the preconditioner substantially reduces the number of mBCG iterations required to achieve convergence.

In Fig. 3.4 (bottom), we confirm that these more accurate solves indeed result in faster training. We plot the test MAE of preconditioned/non-preconditioned mBCG as a function of wall-time.[6] We observe that a rank-5 preconditioner is indeed sufficient—the solves converge up to 5 times faster than without precon-

[6]Wall-clock time is varied by changing the number of CG iterations.

ditioning. Consequentially, we recommend always using the partial pivoted Cholesky preconditioner with BBMM. It has virtually no wall-clock overhead and rapidly accelerates convergence.

3.6 Discussion

In this chapter, we have presented a novel algorithm for Gaussian process training (BBMM) based on blackbox matrix-matrix multiplication routines with kernel matrices. Below we discuss our findings and discuss extensions of BBMM that will be explored in future chapters.

Avoiding the Cholesky decomposition. An important takeaway of this chapter is that it is beneficial to avoid the Cholesky decomposition for GP training, even when no structured approximations are made. We will explore the exact GP setting in more detail in Chapter 6 and further demonstrate the computational benefits of BBMM.

The Cholesky decomposition performs a large amount of computation to get a linear solve when fast iterative methods suffice. Ultimately, the Cholesky decomposition of a full matrix takes $\mathcal{O}(N^3)$ time while CG takes $\mathcal{O}(N^2)$ time. Indeed, as shown in Fig. 3.3, CG may even provide *better* linear solves than the Cholesky decomposition.

Non-Gaussian likelihoods. When GP models are used with non-conjugate likelihoods (e.g. for classification or heavy-tailed noise models), we cannot compute the GP's marginal log likelihood (i.e. Eqs. (2.6) and (2.7) do not apply).

We must instead use variational approximations of the marginal log likelihood, which require a different set of training and inference equations. We will discuss how MVM techniques can be applied to this problem in Chapter 5.

Computing predictive distributions. The focus of this chapter has been optimizing the GP marginal log likelihood. The next chapter will focus on making predictions after a GP has been trained. The equations for computing predictive distributions—Eqs. (2.4) and (2.5)—require matrix solves. In the next chapter, we will introduce a MVM based method that *pre-computes and caches* most of the required CG computation.

CHAPTER 4

GAUSSIAN PROCESS PREDICTIONS VIA LANCZOS VARIANCE ESTIMATES

4.1 Introduction

Applying a Gaussian process model to previously-unseen test data returns a *predictive distribution* rather than a point prediction. Gaussian process predictive distribution are non-parametric and naturally adapt to the amount of training data. As a result, these predictions tend to be well-calibrated—even for data points that lie far away from previously-seen training data [Rasmussen and Williams, 2006, Wilson, 2014]. This property is crucial for applications where incorrect predictions could have catastrophic consequences, such as in medicine [Schulam and Saria, 2017] or large-scale robotics [Deisenroth et al., 2015].

This non-parametric formulation unfortunately comes with a computational downside. Computing predictions with a GP requires performing linear solves with the $N \times N$ training covariance matrix (see Eqs. (2.4) and (2.5)). BBMM and recent advances in inducing point methods can be used to reduce some of these computational requirements. Using a simple caching strategy (described in the next section), predictive means can be computed in $\mathcal{O}(N)$ time. This computation can be reduced to $\mathcal{O}(1)$ when used in conjunction with Kernel Interpolation for Scalable Structured GP models (KISS-GP, see Section 2.1.6).

Unfortunately, predictive uncertainties remain a computational bottleneck, even with BBMM. The predictive variance $\text{Var}^*(\mathbf{x}^*)$ requires computing $\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{k}_{\mathbf{X}\mathbf{x}^*}$ (see Eq. (2.5)), which depends on the test point $\mathbf{x}^*$ and therefore cannot

be computed upfront. Matching the complexity of predictive mean inference without sacrificing accuracy has remained an open problem. The majority of this chapter is therefore dedicated to reducing the computational requirements of predictive variances.

We provide a matrix-vector multiplication (MVM) solution based on the tridiagonalization algorithm of Lanczos [1950]. We express the predictive covariance between $\mathbf{x}^*$ and $\mathbf{x}^{*\prime}$ as $\mathbf{k}_{\mathbf{X}\mathbf{x}^*}^\top \mathbf{C}\, \mathbf{k}_{\mathbf{X}\mathbf{x}^*}$, where $\mathbf{C} \approx \widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}$ is a low-rank $N \times N$ approximation. Using the Lanczos algorithm, we can efficiently form this low-rank decomposition using $J \ll N$ matrix-vector multiplications with $\widehat{\mathbf{K}}_{\mathbf{XX}}$. After this one-time upfront computation, all variances can be computed in *linear time* – $\mathcal{O}(JN)$ – per (co)-variance entry. When used in conjunction with KISS-GP, this complexity can be further reduced to *constant time*, and posterior samples can be drawn in *linear time*.

We refer to this method as LanczOs Variance Estimates, or LOVE for short. LOVE has the lowest asymptotic complexity for computing predictive (co)-variances with GPs. We empirically validate LOVE on seven datasets and find that it consistently provides substantial speedups over existing methods *without sacrificing accuracy*. Variances and samples are accurate to within four decimals, and can be computed *up to 18,000 times faster.*

4.2 Motivation

Eq. (2.4) and Eq. (2.5) describe how to compute predictive means and (co)-variances (restated below, assuming a prior mean of $\mathbf{0}$ for brevity):

$$\mu^* (\mathbf{x}^*) = \mathbf{k}_{\mathbf{X}\mathbf{x}^*}^\top \widehat{\mathbf{K}}_{\mathbf{X}\mathbf{X}}^{-1} \mathbf{y} \tag{4.1}$$

$$\text{Cov}^* (\mathbf{x}^*, \mathbf{x}^{*\prime}) = k(\mathbf{x}^*, \mathbf{x}^{*\prime}) - \mathbf{k}_{\mathbf{X}\mathbf{x}^*}^\top \widehat{\mathbf{K}}_{\mathbf{X}\mathbf{X}}^{-1} \mathbf{k}_{\mathbf{X}\mathbf{x}^{*\prime}} + \sigma_{\text{obs}}^2. \tag{4.2}$$

The terms in blue only depend on the training data. If we are making a single prediction with a GP model, then both of these terms only require a single call to conjugate gradients ($\widehat{\mathbf{K}}_{\mathbf{X}\mathbf{X}}^{-1} \mathbf{k}_{\mathbf{X}\mathbf{x}^*}^\top$). However, if we are making predictions on thousands of test points, these repeated CG calls may become prohibitively expensive, even when using the mBCG algorithm to parallelize the solves. Our goal is to *pre-compute and cache* the most computationally intensive parts of these equations. After this pre-computation, subsequent predictions ideally should be computationally cheap—i.e. $\mathcal{O}(N)$ or less.

4.2.1 Computing Predictive Means

Before discussing predictive (co)-variances, which will be the primary focus of this chapter, we will first discuss a pre-computation strategy for predictive means: Since the $\widehat{\mathbf{K}}_{\mathbf{X}\mathbf{X}}^{-1} \mathbf{y} \triangleq \mathbf{a}$ term only depends on training data, it can be cached and re-used for all predictive means. Each subsequent predictive mean is simply the inner product between the $\mathbf{k}_{\mathbf{X}\mathbf{x}^*}$ vector and the pre-computed $\mathbf{a}$ vector, which in general takes $\mathcal{O}(N)$ time.

This cost can be reduced even further for KISS-GP models, as discussed in Section 2.1.6 and [Wilson et al., 2015]. Recall that KISS-GP approximates the

training and test covariances as:

$$\tilde{\mathbf{k}}_{\mathbf{X}\mathbf{x}^*} = \mathbf{W}_{\mathbf{X}}^\top \mathbf{K}_{\mathbf{Z}\mathbf{Z}} \mathbf{w}_{\mathbf{x}^*}, \quad \tilde{\mathbf{K}}_{\mathbf{X}\mathbf{X}} = \mathbf{W}_{\mathbf{X}}^\top \mathbf{K}_{\mathbf{Z}\mathbf{Z}} \mathbf{W}_{\mathbf{X}},$$

where $\mathbf{K}_{\mathbf{Z}\mathbf{Z}} \in \mathbb{R}^{M \times M}$ is the Toeplitz inducing kernel matrix, $\mathbf{W}_{\mathbf{X}} \in \mathbb{R}^{M \times N}$ is the sparse interpolation for training points $\mathbf{X}$, and $\mathbf{w}_{\mathbf{x}^*} \in \mathbb{R}^M$ is the sparse interpolation for $\mathbf{x}^*$. Plugging these approximations into Eq. (4.1) gives us

$$\mu^* (\mathbf{x}^*) = \mathbf{w}_{\mathbf{x}^*}^\top \underbrace{\mathbf{K}_{\mathbf{Z}\mathbf{Z}} \mathbf{W}_{\mathbf{X}} (\mathbf{W}_{\mathbf{X}}^\top \mathbf{K}_{\mathbf{Z}\mathbf{Z}} \mathbf{W}_{\mathbf{X}} + \sigma_{\text{obs}}^2 \mathbf{I})^{-1} \mathbf{y}}_{\mathbf{a}'}.$$

where again the blue terms only depend on training data. After pre-computing the $\mathbf{a}'$ vector, all subsequent means are the inner product between $\mathbf{a}'$ and the $\mathcal{O}(1)$ *sparse* $\mathbf{w}_{\mathbf{x}}^*$ vector. These computations are $\mathcal{O}(1)$.

4.2.2 Computing (Co)-Variances without Pre-Computation

The predictive (co)-variances are more computationally challenging, as the only term in Eq. (4.2) that does not depend on test data is $\widehat{\mathbf{K}}_{\mathbf{X}\mathbf{X}}^{-1}$. A common pre-computation is to form the Cholesky factorization $\mathbf{L}\mathbf{L}^\top = \widehat{\mathbf{K}}_{\mathbf{X}\mathbf{X}}$ ($\mathcal{O}(N^3)$ time, $\mathcal{O}(N^2)$ memory). After factorization, all subsequent solves take $\mathcal{O}(N^2)$ time. However, this cubic dependence on N and quadratic memory may be infeasible for $N \geq 10{,}000$.

It is possible to obtain some computational savings when using inducing point methods. For example, if we replace $\mathbf{k}_{\mathbf{X}\mathbf{x}^*}$ and $\widehat{\mathbf{K}}_{\mathbf{X}\mathbf{X}}$ with their corresponding KISS-GP approximations in Eq. (4.2), then we have:

$$\begin{aligned} \text{Cov}^* (\mathbf{x}^*, \mathbf{x}^{*\prime}) \approx \sigma_{\text{obs}}^2 + k(\mathbf{x}^*, \mathbf{x}^{*\prime}) \\ - \mathbf{w}_{\mathbf{x}^*}^\top \underbrace{\mathbf{K}_{\mathbf{Z}\mathbf{Z}} \mathbf{W}_{\mathbf{X}} \left(\mathbf{W}_{\mathbf{X}}^\top \mathbf{K}_{\mathbf{Z}\mathbf{Z}} \mathbf{W}_{\mathbf{X}} + \sigma_{\text{obs}}^2 \mathbf{I} \right)^{-1} \mathbf{W}_{\mathbf{X}}^\top \mathbf{K}_{\mathbf{Z}\mathbf{Z}}}_{\mathbf{C}} \mathbf{w}_{\mathbf{x}^{*\prime}}. \end{aligned} \tag{4.3}$$

$\mathbf{C}$, the braced portion of Eq. (4.3), does not depend on the test points $\mathbf{x}_i^*$, $\mathbf{x}_j^*$ and therefore can be pre-computed during training. The primary cost of this pre-computation is the M solves with $(\mathbf{W}_{\mathbf{X}}^\top \mathbf{K}_{\mathbf{ZZ}} \mathbf{W}_{\mathbf{X}} + \sigma_{\text{obs}}^2 \mathbf{I})$: one for each column vector in $\mathbf{W}_{\mathbf{X}}^\top \mathbf{K}_{\mathbf{ZZ}}$, each of which takes $\mathcal{O}(N + M \log M)$ time with CG (see Section 2.1.6 or Wilson and Nickisch [2015]). The total time for this pre-computation is therefore $\mathcal{O}(MN + M^2 \log M)$ (the time for M solves). After pre-computation, Eq. (4.3) becomes

$$\text{Cov}^* \left(\mathbf{x}^*, \mathbf{x}^{*\prime}\right) \approx k_{\mathbf{x}^*\mathbf{x}^{*\prime}} - \mathbf{w}_{\mathbf{x}^*}^\top \, \mathbf{C} \, \mathbf{w}_{\mathbf{x}^{*\prime}} + \sigma_{\text{obs}}^2 \tag{4.4}$$

As $\mathbf{w}_{\mathbf{x}^*}$ contains only four nonzero elements, the inner product of $\mathbf{w}_i^*$ with $\mathbf{C}$ takes $\mathcal{O}(M)$ time. Thus predictive covariances with Eq. (4.4) are $\mathcal{O}(M)$ after pre-computation.

Although this technique offers computational savings over the Cholesky method, the quadratic dependence on M in the pre-computation phase is a computational bottleneck. In contrast, all other operations with KISS-GP require at most linear storage and near-linear time. Indeed, one of the hallmarks of KISS-GP is the ability to use a very large number of inducing points $M = \Theta(N)$ so that kernel computations are nearly exact.

4.3 LanczOs Variance Estimates (LOVE)

We propose to overcome these limitations through an altered pre-computation step. In particular, we can approximate $\widehat{\mathbf{K}}_{\mathbf{XX}}$ in Eq. (4.2) as a low rank matrix. Letting $\mathbf{R}$ be a $J \times N$ matrix such that $\mathbf{R}^\top \mathbf{R} \approx \widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}$, we rewrite Eq. (4.2) as:

$$\text{Cov}^* \left(\mathbf{x}^*, \mathbf{x}^{*\prime}\right) = k(\mathbf{x}^*, \mathbf{x}^{*\prime}) - (\mathbf{R}\mathbf{k}_{\mathbf{X}\mathbf{x}^*})^\top (\mathbf{R}\mathbf{k}_{\mathbf{X}\mathbf{x}^{*\prime}}) + \sigma_{\text{obs}}^2 \tag{4.5}$$

Algorithm 4.1: LanczOs Variance Estimates (LOVE). Terms in blue only depend on training data.

Input : $\mathbf{k}_{\mathbf{X}\mathbf{x}^*}$, $\mathbf{k}_{\mathbf{X}\mathbf{x}^{*\prime}}$ – covariance vectors for $\mathbf{x}^*$, $\mathbf{x}^{*\prime}$
$k(\mathbf{x}^*, \mathbf{x}^{*\prime})$ – prior covariance between $\mathbf{x}^*$, $\mathbf{x}^{*\prime}$
`mvm_`$\widehat{\mathbf{K}}_{\mathbf{XX}}$`()` – func. that performs MVMs with $\widehat{\mathbf{K}}_{\mathbf{XX}}$
J – number of Lanczos iterations

Output: Approximate predictive covariance $\mathrm{Cov}^*(\mathbf{x}^*, \mathbf{x}^{*\prime})$.

if $\mathbf{R}$ *has not previously been computed* **then**

$\mathbf{Q}, \mathbf{T} \leftarrow$ `lanczos`$_J$`(` `mvm_`$\widehat{\mathbf{K}}_{\mathbf{XX}}$, $\mathbf{k}_{\mathbf{X}\mathbf{x}^*}$`)`
`//` J `iter. of Lanczos w/ matrix` $\widehat{\mathbf{K}}_{\mathbf{XX}}$ `and probe vec.` $\mathbf{k}_{\mathbf{X}\mathbf{x}^*}$
$\mathbf{L}_{\mathbf{T}} \leftarrow$ `cholesky_factor(` $\mathbf{T}$`)`
$\mathbf{R} \leftarrow$ `trinagular_solve(` $\mathbf{Q}^\top, \mathbf{L}_{\mathbf{T}}$`)`

end

`//` $\mathbf{R} \in \mathbb{R}^{J \times N}$, $J \ll N$.

return $k(\mathbf{x}^*, \mathbf{x}^{*\prime}) - (\mathbf{k}_{\mathbf{X}\mathbf{x}^*}^\top \mathbf{R}^\top)(\mathbf{R}\mathbf{k}_{\mathbf{X}\mathbf{x}^{*\prime}})$

Variance computations with Eq. (4.5) take $\mathcal{O}(JN)$ time.

An MVM-based low-rank approximation with Lanczos. There are many possible ways to form a low-rank approximation of $\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}$. Our proposed method will make use of the Lanczos algorithm from Section 2.3.2, which will generate the low-rank approximation through matrix-vector multiplication (MVMs). As we will demonstrate in Section 4.5, the Lanczos low-rank approximation rapidly converges to the true inverse.

Recall from Section 2.3.2 that J iterations of Lanczos tridiagonalization approximate matrix solves:

$$\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{b} \approx \mathbf{Q}\mathbf{T}^{-1}\mathbf{Q}^\top\mathbf{b},$$

where the orthonormal matrix $\mathbf{Q} \in \mathbb{R}^{N \times J}$ and tridiagonal matrix $\mathbf{T} \in \mathbb{R}^{J \times J}$ are computed with respect to probe vector $\mathbf{b}$. As argued by Parlett [1980], Saad [1987], and Schneider and Willsky [2001], the $\mathbf{Q}$ and $\mathbf{T}$ matrices can be used

Table 4.1: Asymptotic complexities of predictive (co)-variances (N training points, M inducing points, J Lanczos/CG iterations).

Method	**Pre-computation** (time)	(storage)	**Computing variances** (time)
Standard GP	$\mathcal{O}(N^3)$	$\mathcal{O}(N^2)$	$\mathcal{O}(N^2)$
SGPR	$\mathcal{O}(NM^2)$	$\mathcal{O}(M^2)$	$\mathcal{O}(M^2)$
KISS-GP	–	–	$\mathcal{O}(J(N + M \log M))$
Standard GP (w/ LOVE)	$\mathcal{O}(JN^2)$	$\mathcal{O}(JN)$	$\mathcal{O}(JN)$
KISS-GP (w/ LOVE)	$\mathcal{O}(J(N + M \log M))$	$\mathcal{O}(JM)$	$\mathcal{O}(J)$

to approximate subsequent solves $\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{b}' \approx \mathbf{Q}\mathbf{T}^{-1}\mathbf{Q}^\top\mathbf{b}'$. We exploit this fact and use $\mathbf{Q}\mathbf{T}^{-1}\mathbf{Q}^\top$ to be a general-purpose approximation to $\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}$. By running $J \ll N$ Lanczos iterations (e.g. $J \approx 100$), the resulting approximation will be low-rank. In particular, we have

$$\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1} \approx \overbrace{\mathbf{Q}\mathbf{T}^{-1}\mathbf{Q}^\top}^{\text{apply Lanczos}} = \underbrace{\left(\mathbf{Q}\mathbf{L}_{\mathbf{T}}^{-\top}\right)}_{\mathbf{R}^\top}\underbrace{\left(\mathbf{L}_{\mathbf{T}}^{-1}\mathbf{Q}^\top\right)}_{\mathbf{R}}$$

where $\mathbf{L}_\mathbf{T}$ is the Cholesky factor of $\mathbf{T}$. Applying Lanczos to $\widehat{\mathbf{K}}_{\mathbf{XX}}$ requires J MVMs for a total of $\mathcal{O}(J\xi(\widehat{\mathbf{K}}_{\mathbf{XX}}))$ time ($\xi(\widehat{\mathbf{K}}_{\mathbf{XX}})$ is the complexity of one MVM with $\widehat{\mathbf{K}}_{\mathbf{XX}}$, which is nominally $\mathcal{O}(N^2)$). Computing and applying the Cholesky factor $\mathbf{L}_\mathbf{T}$ is $\mathcal{O}(J)$ time due to the tridiagonal structure of $\mathbf{T}$.

In total, the entire pre-computation phase takes $\mathcal{O}(JN^2)$ time for standard GPs. This is the same amount of time of a single marginal likelihood computation using BBMM. After pre-computation, each covariance takes $\mathcal{O}(JN)$ time. We refer to this fast covariance approximation algorithm as **LanczOs Variance Estimates**, or **LOVE** for short. It is summarized in Algorithm 4.1 and Table 4.1.

J, the size of the low-rank approximation, depends on the conditioning of $\widehat{\mathbf{K}}_{\mathbf{XX}}$ and not its size. Empirically, we find that $J = 100$ is sufficient for most matrices with $N \leq 20{,}000$; therefore J can be considered to be constant.

Table 4.2: Asymptotic complexities of posterior sampling (N training points, M inducing points, J Lanczos/CG iterations, S samples, T test points).

Method	**Pre-computation** (time)	(storage)	**Drawing S samples** (time)
Standard GP	–	–	$\mathcal{O}(TN^2 + T^2(N+S) + T^3)$
SGPR	–	–	$\mathcal{O}(TM^2 + T^2(M+S) + T^3)$
KISS-GP	–	–	$\mathcal{O}(JT(N + M\log M) + T^2(M+S) + T^3)$
KISS-GP (w/ LOVE)	$\mathcal{O}(J(N + M\log M))$	$\mathcal{O}(JM)$	$\mathcal{O}(JS(T+M))$

4.3.1 Programmability

Because LOVE is a MVM-based algorithm, it affords the same modularity as BBMM. When LOVE is used in conjunction with scalable GP approximations/multitask models, we can take advantage of fast kernel MVMs for a $o(N^2)$ asymptotic complexity. In GPyTorch we use the `LazyTensor` construct from Section 3.4 to adapt LOVE to specialty models. The same `_matmul` function we use for mBCG can also be used for fast (co)-variances with LOVE.

4.4 LOVE with KISS-GP

In this section, we demonstrate that LOVE is an especially compelling algorithm for the scalable KISS-GP framework. With a few modifications to Algorithm 4.1, KISS-GP + LOVE can achieve *constant-time covariance approximations* and *linear time posterior samples*.

Algorithm 4.2: LOVE + KISS-GP for constant-time predictive variances. Terms in blue only depend on training data.

Input : $\mathbf{w}_{\mathbf{x}^*}$, $\mathbf{w}_{\mathbf{x}^{*\prime}}$ – interpolation vectors for $\mathbf{x}^*$, $\mathbf{x}^{*\prime}$
$k(\mathbf{x}^*, \mathbf{x}^{*\prime})$ – prior covariance between $\mathbf{x}^*$, $\mathbf{x}^{*\prime}$
$\mathbf{b} = \frac{1}{M}\mathbf{W}_{\mathbf{X}}^\top \mathbf{K}_{\mathbf{ZZ}}\mathbf{1}$ – average col. of $\mathbf{W}_{\mathbf{X}}^\top \mathbf{K}_{\mathbf{ZZ}}$
`mvm_`$\widehat{\mathbf{K}}_{\mathbf{XX}}$`()` – func. that performs MVMs with $\widehat{\mathbf{K}}_{\mathbf{XX}}$
`mvm_`$\mathbf{K}_{\mathbf{ZX}}$`()` – func. that performs MVMs with $\mathbf{K}_{\mathbf{ZX}}$

Output: Approximate predictive covariance $\text{Cov}^*(\mathbf{x}^*, \mathbf{x}^{*\prime})$.

if $\widetilde{\mathbf{R}}$ *has not previously been computed* **then**
  $\mathbf{Q}, \mathbf{T} \leftarrow$ `lanczos`$_J$`(` `mvm_`$\widehat{\mathbf{K}}_{\mathbf{XX}}$, $\mathbf{b}$`)`
    `//` J `iter. of Lanczos w/ matrix` $\widehat{\mathbf{K}}_{\mathbf{XX}}$ `and probe vec.` $\mathbf{b}$
  $\mathbf{L}_{\mathbf{T}} \leftarrow$ `cholesky_factor(` $\mathbf{T}$`)`
  $\widetilde{\mathbf{R}} \leftarrow$ `trinagular_solve(` `mvm_`$\mathbf{K}_{\mathbf{ZX}}$`(` $\mathbf{Q}$`),` $\mathbf{L}_{\mathbf{T}}$`);`
   `//` $\widetilde{\mathbf{R}} = \mathbf{L}_{\mathbf{T}}^{-1}\mathbf{Q}^\top \mathbf{W}_{\mathbf{X}}^\top \mathbf{K}_{\mathbf{ZZ}}$
end
`//` $\widetilde{\mathbf{R}} \in \mathbb{R}^{J \times M}$, $J \ll M$.
$\mathbf{v} \leftarrow$ `sparse_mm(` $\widetilde{\mathbf{R}}$, $\mathbf{w}_{\mathbf{x}}^*$`)`
$\mathbf{v}' \leftarrow$ `sparse_mm(` $\widetilde{\mathbf{R}}$, $\mathbf{w}_{\mathbf{x}}^{*\prime}$`)`
return $k_{\mathbf{x}_i^*, \mathbf{x}_j^*} - \mathbf{v}^T\mathbf{v}'$

4.4.1 Constant-Time (Co)-Variances with KISS-GP + LOVE

The KISS-GP approximation $\mathbf{W}_{\mathbf{X}}^\top \mathbf{K}_{\mathbf{ZZ}} \mathbf{W}_{\mathbf{X}}$ allows us to make additional precomputations to further reduce test-time complexity. In particular,

$$\begin{aligned}
\text{Cov}^*(\mathbf{x}^*, \mathbf{x}^{*\prime}) &\approx k(\mathbf{x}^*, \mathbf{x}^{*\prime}) - \mathbf{k}_{\mathbf{X}\mathbf{x}^*}^\top \mathbf{R}^\top \mathbf{R} \mathbf{k}_{\mathbf{X}\mathbf{x}^{*\prime}} + \sigma_{\text{obs}}^2 \\
&\approx k(\mathbf{x}^*, \mathbf{x}^{*\prime}) - \big(\mathbf{w}_{\mathbf{x}^*}^\top \underbrace{\mathbf{K}_{\mathbf{ZZ}}\mathbf{W}_{\mathbf{X}})\,\mathbf{R}^\top}_{\widetilde{\mathbf{R}}^\top} \underbrace{\mathbf{R}\,(\mathbf{W}_{\mathbf{X}}^\top \mathbf{K}_{\mathbf{ZZ}}}_{\widetilde{\mathbf{R}}}\, \mathbf{w}_{\mathbf{x}^{*\prime}}\big) + \sigma_{\text{obs}}^2 \\
&\approx k(\mathbf{x}^*, \mathbf{x}^{*\prime}) - \left(\widetilde{\mathbf{R}}\mathbf{w}_{\mathbf{x}^*}^\top\right)^\top \left(\widetilde{\mathbf{R}}\mathbf{w}_{\mathbf{x}^{*\prime}}^\top\right) + \sigma_{\text{obs}}^2 \qquad (4.6)
\end{aligned}$$

The matrix $\widetilde{\mathbf{R}} = \mathbf{R}\mathbf{K}_{\mathbf{ZZ}}\mathbf{W}_{\mathbf{X}}$ is a $J \times M$ matrix. Variance computations with Eq. (4.6) take $\mathcal{O}(J)$ time due to the sparsity of $\mathbf{w}_{\mathbf{x}^*}$ and $\mathbf{w}_{\mathbf{x}^{*\prime}}$. Taking J to be a constant (as $J \approx 100$ suffices for most kernel matrices), KISS-GP covariance computations with Eq. (4.6) take *constant time*.

Moreover, this additional work to compute $\widetilde{\mathbf{R}}$ from $\mathbf{R}$ takes negligible time. The complexity of computing $\mathbf{R}$ is $\mathcal{O}(J(N + M \log M))$, as Lanczos requires J MVMs with $\widehat{\mathbf{K}}_{\mathbf{XX}}$ and KISS-GP affords $\mathcal{O}(N + M \log M)$ MVMs. Forming $\widetilde{\mathbf{R}}$ requires multiplying the $J \times N$ $\mathbf{R}$ matrix by $\mathbf{K}_{\mathbf{ZZ}}$ and $\mathbf{W}_{\mathbf{X}}$, which also takes $\mathcal{O}(J(N + M \log M))$ time. It is summarized in Algorithm 4.2 and Table 4.1.

In addition to these fast predictive (co)-variances, LOVE + KISS-GP offers two additional speedups that are specific to KISS-GP models.

4.4.2 Predictive Distribution Sampling with LOVE + KISS-GP

When used in conjunction with KISS-GP, LOVE can also be used to compute samples from the posterior covariance matrix. This is a very common operation: in Bayesian optimization, several popular acquisition functions—such as predictive entropy search [Hernández-Lobato et al., 2014], max-value entropy search [Wang and Jegelka, 2017], and knowledge gradient [Frazier et al., 2009]—require posterior sampling.

Let $\mathbf{X}^* = [\mathbf{x}_1^*, \ldots, \mathbf{x}_T^*]$ be a test set of T points. To draw samples from $p(f(\mathbf{x}_1^*), \ldots, f(\mathbf{x}_T^*) \mid \mathcal{D})$—the posterior function evaluated on $\mathbf{x}_1^*, \ldots, \mathbf{x}_T^*$, the cross-covariance terms (i.e. $\text{Cov}^*\left(\mathbf{x}_i^*, \mathbf{x}_j^*\right)$) are necessary in addition to the variance terms ($\text{Var}^*\left(\mathbf{x}_i^*\right)$). We sample $\boldsymbol{\epsilon} \sim p(f(\mathbf{x}_1^*), \ldots, f(\mathbf{x}_T^*) \,|\, \mathcal{D})$ through the reparameterization trick [Kingma and Welling, 2014, Rezende et al., 2014]:

$$\boldsymbol{\mu}^* = \begin{bmatrix} \mu^*\left(\mathbf{x}_1^*\right) \\ \vdots \\ \mu^*\left(\mathbf{x}_1^*\right) \end{bmatrix}, \quad \mathbf{COV}^* = \begin{bmatrix} \text{Cov}^*\left(\mathbf{x}_1^*, \mathbf{x}_1^*\right) & \cdots & \text{Cov}^*\left(\mathbf{x}_1^*, \mathbf{x}_T^*\right) \\ & \ddots & \\ \text{Cov}^*\left(\mathbf{x}_T^*, \mathbf{x}_1^*\right) & \cdots & \text{Cov}^*\left(\mathbf{x}_T^*, \mathbf{x}_T^*\right) \end{bmatrix},$$

$$\boldsymbol{\epsilon} = \boldsymbol{\mu}^* + \mathbf{S}\boldsymbol{\epsilon}' \sim p(\mathbf{f}(\mathbf{x}_1^*), \ldots, \mathbf{f}(\mathbf{x}_T^*) \,|\, \mathcal{D}) \tag{4.7}$$

where $\boldsymbol{\epsilon}' \sim \mathcal{N}[\mathbf{0}, \mathbf{I}]$ and $\mathbf{S}$ is some matrix such that $\mathbf{S}\mathbf{S}^\top = \mathbf{COV}^*$. Typically $\mathbf{S}\mathbf{S}^\top$ is taken to be the Cholesky decomposition of the posterior covariance matrix. Computing this decomposition incurs a $\mathcal{O}(T^3)$ cost on top of the $\mathcal{O}(T^2)$ covariance evaluations.

A fast KISS-GP sampling matrix. We use LOVE and KISS-GP to rewrite Eq. (4.6) as

$$\begin{aligned}
\mathbf{COV}^* &\approx \mathbf{K}_{\mathbf{X}^*,\mathbf{X}^*} - \overbrace{\mathbf{W}_{\mathbf{X}^*}^\top \left(\widetilde{\mathbf{R}}^\top \widetilde{\mathbf{R}}\right) \mathbf{W}_{\mathbf{X}^*}}^{\text{LOVE + KISS-GP approximation}} \\
&\approx \overbrace{\mathbf{W}_{\mathbf{X}^*}^\top \mathbf{K}_{\mathbf{Z}\mathbf{Z}} \mathbf{W}_{\mathbf{X}^*}}^{\text{KISS-GP approximation}} - \mathbf{W}_{\mathbf{X}^*}^\top \left(\widetilde{\mathbf{R}}^\top \widetilde{\mathbf{R}}\right) \mathbf{W}_{\mathbf{X}^*} \\
&= \mathbf{W}_{\mathbf{X}^*}^\top \left(\mathbf{K}_{\mathbf{Z}\mathbf{Z}} - \widetilde{\mathbf{R}}^\top \widetilde{\mathbf{R}}\right) \mathbf{W}_{\mathbf{X}^*} \qquad (4.8)
\end{aligned}$$

where $\mathbf{W}_{\mathbf{X}^*} = [\mathbf{w}_{\mathbf{x}_1^*}, \ldots, \mathbf{w}_{\mathbf{x}_T^*}]$ is the interpolation matrix for test points. We have thus reduced the full covariance matrix to a test-independent term $(\mathbf{K}_{\mathbf{Z}\mathbf{Z}} - \widetilde{\mathbf{R}}^\top \widetilde{\mathbf{R}})$ that can be pre-computed. We apply the Lanczos algorithm on this term during pre-computation to obtain a rank-J approximation:

$$\mathbf{K}_{\mathbf{Z}\mathbf{Z}} - \widetilde{\mathbf{R}}^\top \widetilde{\mathbf{R}}' \approx \mathbf{Q}'\mathbf{T}'\mathbf{Q}'^\top, \qquad (4.9)$$

where again $\mathbf{Q}' \in \mathbb{R}^{M \times J}$ is orthonormal and $\mathbf{T}' \in \mathbb{R}^{J \times J}$ is tridiagonal. This Lanczos decomposition $\mathbf{Q}'\mathbf{T}'\mathbf{Q}'^\top$ requires J matrix-vector multiplies with $\mathbf{K}_{\mathbf{Z}\mathbf{Z}} - \widetilde{\mathbf{R}}^\top \widetilde{\mathbf{R}}$, each of which requires $\mathcal{O}(M \log M)$ time. Substituting Eq. (4.9) into Eq. (4.8), we get:

$$\begin{aligned}
\mathbf{COV}^* &\approx \mathbf{W}_{\mathbf{X}^*}^\top \left(\mathbf{Q}'\mathbf{T}'\mathbf{Q}'^\top\right) \mathbf{W}_{\mathbf{X}^*} \\
&= \mathbf{W}_{\mathbf{X}^*}^\top \left(\mathbf{Q}'\mathbf{L}_{\mathbf{T}'}\right) \left(\mathbf{L}_{\mathbf{T}'^\top}\mathbf{Q}'^\top\right) \mathbf{W}_{\mathbf{X}^*} \qquad (4.10)
\end{aligned}$$

where $\mathbf{L}_{\mathbf{T}'}$ is the Cholesky factor of $\mathbf{T}'$ (a $\mathcal{O}(J)$ operation due to the tridiagonal structure). Setting $\mathbf{S} = \mathbf{Q}'\mathbf{L}_{\mathbf{T}}$, we see that $\mathbf{COV}^* = (\mathbf{W}_{\mathbf{X}^*}^\top \mathbf{S})(\mathbf{W}_{\mathbf{X}^*}^\top \mathbf{S})^\top$. Moreover,

$\mathbf{S} \in \mathbb{R}^{M \times J}$ can be pre-computed and cached since it does not depend on test data. In total, pre-computing $\mathbf{S}$ takes $\mathcal{O}(JM \log M + MJ^2)$ time in addition to what is required for fast variances. A matrix-vector multiplication with $(\mathbf{W}_{\mathbf{X}}^\top \mathbf{S})$ takes $\mathcal{O}(TJ)$ time due to the $\mathcal{O}(T)$ sparsity of $\mathbf{W}_{\mathbf{X}}$. Therefore, drawing S samples (corresponding to S different values of $\boldsymbol{\epsilon}'$) takes $\mathcal{O}(SJ(T+M))$ time (see Table 4.2)—a *linear* dependence on T.

4.4.3 Extension to Additive KISS-GP Kernel Compositions

LOVE is applicable even when the KISS-GP approximation is used with an additive composition of kernels,

$$\widetilde{k}(\mathbf{x}, \mathbf{x}') = \mathbf{w}_{\mathbf{x}}^{(1)\top} \mathbf{K}_{\mathbf{ZZ}}^{(1)} \mathbf{w}_{\mathbf{x}'}^{(1)} + \ldots + \mathbf{w}_{\mathbf{x}}^{(D)\top} \mathbf{K}_{\mathbf{ZZ}}^{(D)} \mathbf{w}_{\mathbf{x}'}^{(D)}.$$

Additive structure has been a focus in several Bayesian optimization settings, since the cumulative regret of additive models depends linearly on the number of dimensions [Kandasamy et al., 2015, Wang et al., 2017, Gardner et al., 2017, Wang and Jegelka, 2017]. Additionally, deep kernel learning GPs [Wilson et al., 2016a,b] typically use sums of one-dimensional kernel functions. To apply LOVE, we note that the KISS-GP additive composition can be re-written as

$$\widetilde{k}(\mathbf{x}, \mathbf{x}') = \begin{bmatrix} \mathbf{w}_{\mathbf{x}}^{(1)} \\ \vdots \\ \mathbf{w}_{\mathbf{x}}^{(D)} \end{bmatrix}^\top \begin{bmatrix} \mathbf{K}_{\mathbf{ZZ}}^{(1)} & \ldots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \ldots & \mathbf{K}_{\mathbf{ZZ}}^{(D)} \end{bmatrix} \begin{bmatrix} \mathbf{w}_{\mathbf{x}}^{(1)} \\ \vdots \\ \mathbf{w}_{\mathbf{x}'}^{(D)} \end{bmatrix}. \tag{4.11}$$

The block matrices in Eq. (4.11) are analogs of their 1-dimensional counterparts in Eq. (2.10). Therefore, we can directly apply Algorithm 4.2, replacing $\mathbf{W}_{\mathbf{X}}$, $\mathbf{w}_{\mathbf{x}}^*$, $\mathbf{w}_{\mathbf{x}}^{*\prime}$, and $\mathbf{K}_{\mathbf{ZZ}}$ with their block forms. The block $\mathbf{w}_{\mathbf{x}}^*$, $\mathbf{w}_{\mathbf{x}}^{*\prime}$ vectors are $\mathcal{O}(D)$-sparse, and therefore interpolation takes $\mathcal{O}(D)$ time. MVMs with the block $\mathbf{K}_{\mathbf{ZZ}}$ matrix

take $\mathcal{O}(DM \log M)$ time by exploiting the block-diagonal structure. With D additive components, predictive variance computations cost only a factor $\mathcal{O}(D)$ more than their 1-dimensional counterparts.

4.5 Results

In this section we demonstrate the effectiveness and speed of KISS-GP + LOVE, both at computing predictive variances and also at posterior sampling. Our goal is to show that 1) LOVE produces uncertainties and samples that are indistinguishable from the state-of-the-art, and 2) that LOVE offers substantial speed improvements. All experiments in this section use LOVE in conjunction with KISS-GP models. See Chapter 6 for results where LOVE is used with standard Gaussian processes.

In the following experiments, all LOVE low-rank approximations use $J = 50$ Lanczos iterations and KISS-GP models use $M = 10{,}000$ inducing points unless otherwise stated. We optimize models with ADAM [Kingma and Ba, 2015] and a learning rate of 0.1. All timing experiments are performed on a GTX 1070 GPU. Exact GPs, KISS-GP models, and LOVE are implemented in our GPyTorch software. SGPR models are implemented in GPFlow [Matthews et al., 2017].

4.5.1 Predictive Variances

We measure the accuracy and speed of LOVE + KISS-GP variances. In all experiments, we compare against Exact GP (**Exact**) variances (without LOVE) and standard KISS-GP variances (**KISS-GP w/o LOVE**). (Note that we do not com-

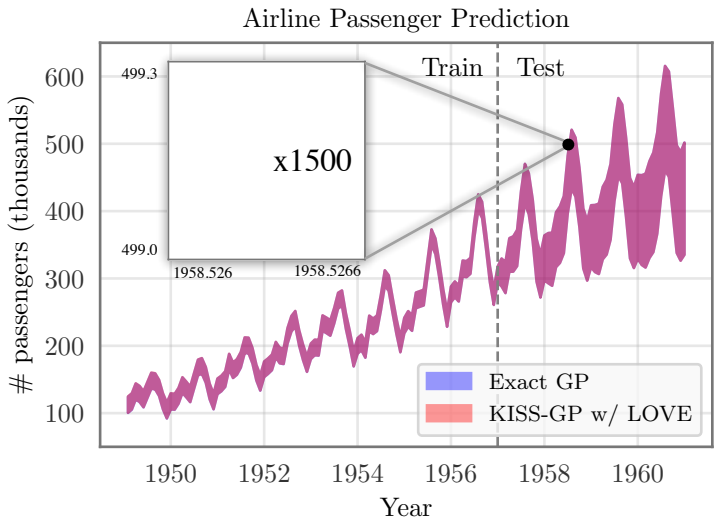

Figure 4.1: Comparison of LOVE predictive variances versus exact predictive variances on airline passenger extrapolation. The LOVE variances are accurate within 10^{-4}.

pare predictive means, as LOVE only affects variance computations.) We report the scaled mean absolute error (SMAE)[1] [Rasmussen and Williams, 2006] of LOVE + KISS-GP variances compared against these baselines. For each dataset, we optimize the hyperparameters of a KISS-GP model. We then apply the same hyperparameters to each baseline model.

One-dimensional example. We first demonstrate LOVE on a complex one-dimensional regression task. The airline passenger dataset (**Airline**) measures the average monthly number of passengers from 1949 to 1961 [Hyndman, 2005]. We aim to extrapolate the numbers for the final 4 years (48 measurements) given the first 8 years (96 measurements). Accurate extrapolation on this dataset requires a kernel function capable of expressing various patterns, such as the spectral mixture (SM) kernel [Wilson and Adams, 2013].

[1]Mean absolute error divided by the variance of y.

Table 4.3: Speedup and accuracy of LOVE + KISS-GP for predictive variances (Deep RBF Kernels). Accuracy is measured by Scaled Mean Average Error. (N is the number of data, D is the dimensionality.)

Dataset			**Variance SMAE**			
Name	N	D	(vs KISS-GP w/o LOVE)	(vs Exact GP)	(from scratch)	(after pre-comp.)
Airfoil	1,503	6	1.30×10^{-5}	7.01×10^{-5}	4×	84×
Skillcraft	3,338	19	2.00×10^{-7}	2.86×10^{-4}	25×	167×
Parkinsons	5,875	20	8.80×10^{-5}	5.18×10^{-3}	46×	443×
PoleTele	15,000	26	2.90×10^{-5}	1.08×10^{-3}	78×	1178×
Elevators	16,599	18	1.20×10^{-6}	–	64×	1017×
Kin40k	40,000	8	3.90×10^{-7}	–	31×	2065×
Protein	45,730	9	5.30×10^{-5}	–	44×	1151×

Dataset			**Speedup over SGPR**			
Name	N	D	(from scratch) $M = 100$	(after pre-comp.) $M = 100$	(from scratch) $M = 1000$	(after pre-comp.) $M = 1000$
Airfoil	1,503	6	3×	49×	9×	183×
Skillcraft	3,338	19	4×	70×	17×	110×
Parkinsons	5,875	20	3×	33×	16×	152×
PoleTele	15,000	26	1.5×	40×	21×	343×
Elevators	16,599	18	2×	31×	20×	316×
Kin40k	40,000	8	8×	81×	12×	798×
Protein	45,730	9	10×	109×	20×	520×

We compute the variances for Exact GP and KISS-GP + LOVE models with a 10-mixture SM kernel. In Fig. 4.1, we see that the LOVE + KISS-GP confidence intervals match the Exact GP's intervals extremely well. The SMAE of LOVE's predicted variances (compared against Exact GP variances) is 1.29×10^{-4}. Although not shown in the plot, we confirm the reliability of these predictions by computing the log-likelihood of the test data. We compare the LOVE + KISS-GP model to an Exact GP, a KISS-GP model without LOVE, and a sparse variational GP (SGPR) model with $M = 1000$ inducing points. All methods achieve nearly identical log-likelihoods, ranging from -221 to -222.

Large datasets. We measure the accuracy of LOVE variances on several large-scale regression benchmarks from the UCI repository [Asuncion and Newman, 2007]. Each of the models use deep RBF kernels (DKL) with the architectures described in [Wilson et al., 2016a]. In Table 4.3, we report the SMAE of the LOVE + KISS-GP variances compared against the two baselines. On all datasets, we find that LOVE + KISS-GP matches KISS-GP w/o LOVE to at least 5 decimal places. Furthermore, LOVE + KISS-GP is able to approximate Exact variances with no more than than 10^{-3} error. Therefore, using LOVE to compute variances results in *almost no loss in accuracy*.

Speedup. In Table 4.3 we compare the variance computation speed of a KISS-GP model with and without LOVE on the UCI datasets. In addition, we compare against SGPR models (with non-deep RBF kernels), a competitive scalable GP approach. On all datasets, we measure the time to compute variances **from scratch**, which includes the cost of pre-computation. In addition, we report the speed **after pre-computing** any terms that aren't specific to test points. We see in Table 4.3 that KISS-GP + LOVE yields a substantial speedup over KISS-GP without LOVE. The speedup is between $4\times$ and $44\times$, even when accounting for LOVE's precomputation. After pre-computation, LOVE is *up to* $2{,}000\times$ *faster*. Additionally, LOVE + KISS-GP is significantly faster than SGPR models. For SGPR models with $M = 100$ inducing points, the LOVE + KISS-GP model (with $M = 10{,}000$ inducing points) is up to $10\times$ faster before pre-computation and $100\times$ faster after. Compared against $M = 1000$ SGPR models, LOVE + KISS-GP is up to $20\times/500\times$ faster before/after precomputation. The biggest improvements are obtained on the largest datasets since LOVE, unlike other methods, is independent of dataset size at test time.

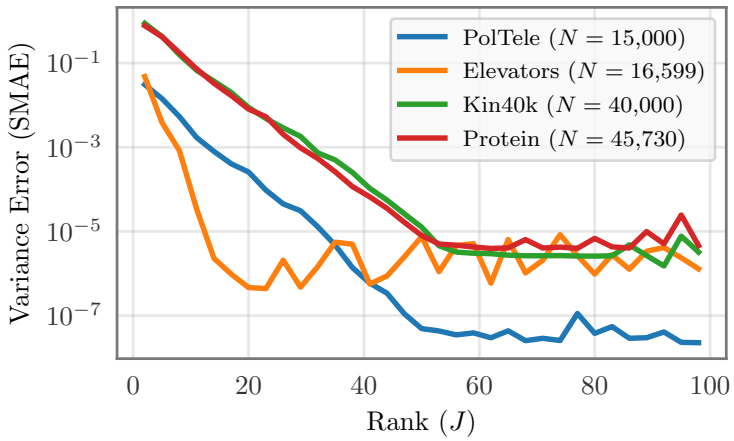

Figure 4.2: LOVE variance error as a function of Lanczos iterations (KISS-GP model, $M = 10{,}000$. Protein, Kin40k, PoleTele, and Elevators UCI datasets).

Accuracy vs. Lanczos iterations. In Fig. 4.2, we measure the accuracy of LOVE as a function of the number of Lanczos iterations (J), which corresponds to the rank of the $\widetilde{\mathbf{R}}$ matrix in Eq. (4.6). We train a KISS-GP model with a deep RBF kernel on the four largest datasets from Table 4.3, measuring the SMAE of LOVE + KISS-GP's predictive variances.[2] As seen in Fig. 4.2, error decreases *exponentially* with the number of Lanczos iterations, up until roughly 50 iterations. After roughly 50 iterations, the error levels off, though this may be an artifact of floating-point precision (which may also cause small subsequent fluctuations).

4.5.2 Sampling

We evaluate the quality of posterior samples drawn with LOVE + KISS-GP as described in Section 4.4.2. We compare against three baselines: sampling from an **Exact GP** using the Cholesky decomposition, sampling from a **SGPR** model with Cholesky, and sampling from an exact GP with random **Fourier features**

[2] As measured against the variances from KISS-GP w/o LOVE.

Table 4.4: Accuracy and computation time of drawing samples from the posterior distribution.

Dataset	Sample Covariance Error				
	Exact GP w/ Cholesky	**Fourier Features**	**SGPR** ($M = 100$)	**SGPR** ($M = 1000$)	**LOVE + KISS-GP**
PolTele	$\mathbf{8.8 \times 10^{-4}}$	1.8×10^{-3}	4.9×10^{-3}	2.7×10^{-3}	$\mathbf{7.5 \times 10^{-4}}$
Elevators	$\mathbf{2.6 \times 10^{-7}}$	3.1×10^{-4}	8.7×10^{-6}	3.6×10^{-6}	$\mathbf{5.5 \times 10^{-7}}$
BO (Eggholder)	$\mathbf{7.7 \times 10^{-4}}$	1.5×10^{-3}	$\mathbf{8.1 \times 10^{-4}}$	–	$\mathbf{8.0 \times 10^{-5}}$
BO (Styblinski-Tang)	$\mathbf{5.4 \times 10^{-4}}$	7.3×10^{-3}	$\mathbf{5.2 \times 10^{-4}}$	–	$\mathbf{5.2 \times 10^{-4}}$

Dataset	Speedup over Exact GP w/ Cholesky				
	Fourier Features	**SGPR** ($M = 100$)	**SGPR** ($M = 1000$)	**LOVE + KISS-GP** (from scratch)	**LOVE + KISS-GP** (after pre-comp.)
PolTele	$22\times$	$24\times$	$3\times$	$21\times$	$\mathbf{881}\times$
Elevators	$31\times$	$33\times$	$4\times$	$25\times$	$\mathbf{1062}\times$
BO (Eggholder)	$16\times$	$8\times$	–	$19\times$	$\mathbf{775}\times$
BO (Styblinski-Tang)	$11\times$	$8\times$	–	$42\times$	$\mathbf{18{,}100}\times$

[Rahimi and Recht, 2008]. For Fourier features, we use 5000 random features—the maximum number of features without exhausting available GPU memory. The model hyperparameters are learned on an Exact GP and then shared across all baselines. We use two UCI datasets (PolTele and Eleveators) as well as two Bayesian optimization (BO) benchmark functions (Eggholder: 2 dimensional, and Styblinski-Tang: 10 dimensional). For the BO functions, we evaluate the model after 100 queries of max-value entropy search [Wang and Jegelka, 2017]. For Eggholder, we use a standard (non-deep) RBF kernel; for Syblinski-Tang we use the additive kernel suggested by Kandasamy et al. [2015].[3]

[3]The Syblinski-Tang KISS-GP model uses the sum of 10 RBF kernels—one for each dimension—and $M = 100$ inducing points.

Sampling accuracy. In Table 4.4 we evaluate the accuracy of the different sampling methods. We draw $S = 1000$ samples at $T = 10{,}000$ test locations and compare the empirical covariance matrix with the true posterior covariance. The reported numbers are element-wise mean absolute errors. It is worth noting that all methods incur some sampling error—even when sampling with an Exact GP. Nevertheless, Exact GPs, SGPR, and LOVE produce very accurate sample covariance matrices. In particular, Exact GPs and LOVE achieve between 1 and 3 orders of magnitude less error than the Random Fourier Feature method.

Speed. Though LOVE, Exact GPs, and SGPR have similar accuracies, LOVE is much faster. Even when accounting for LOVE's pre-computation time (i.e. sampling "from scratch"), LOVE is comparable to Fourier features and SGPR in terms of speed. At test time (i.e. after pre-computation), LOVE is up to $18{,}000$ times faster than Cholesky.

Bayesian optimization. Many acquisition functions in Bayesian optimization rely on sampling from the GP posterior. For example, max-value entropy search [Wang and Jegelka, 2017] uses samples to estimate the function's maximum value $p(y_{\text{max}} \mid \mathcal{D})$. In Fig. 4.3, we visualize each method's sampled distribution of $p(y_{\text{max}} \mid \mathcal{D})$ for the Eggholder function. We plot kernel density estimates of this distribution after 100 BO iterations.[4] Since the Exact GP sampling method uses the Cholesky decomposition, its sampled max-value distribution can be considered closest to ground truth. The Fourier feature distribution differs significantly from the Exact GP distribution. In contrast, LOVE very closely resembles

[4]Using the max-value entropy search acquisition function [Wang and Jegelka, 2017].

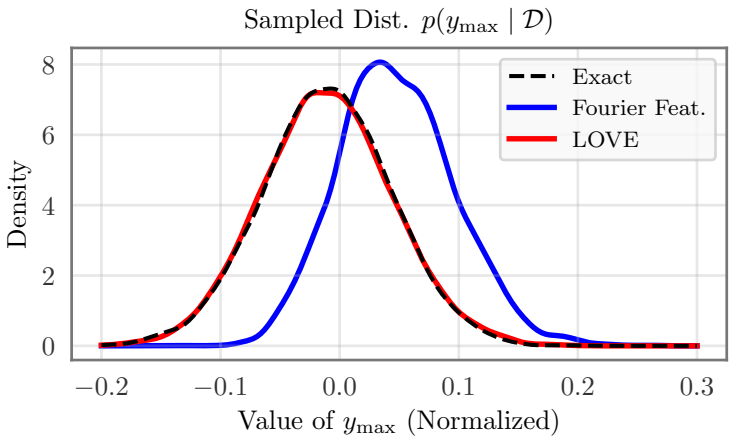

Figure 4.3: Comparison of LOVE versus Random Fourier Features for Bayesian optimization (BO) via max-value entropy search. Each line represents an approximation of the maximum value distribution $p(y_{\max} \mid \mathcal{D})$ using different sampling methods (exact sampling, Random Fourier Features, and LOVE + KISS-GP). Samples drawn with LOVE+KISSGP closely match samples drawn using an Exact GP. (Dataset: (normalized) Eggholder function after 100 iterations of BO.)

Exact GPs, yet can be computed up to $700\times$ faster (Table 4.4).

4.6 Discussion

This chapter has introduced the LOVE algorithm for exact GPs and KISS-GP models. Whereas the running times of previous methods depend on dataset size, LOVE + KISS-GP provides *constant time* and near-exact predictive variances. It is worth noting that LOVE is fully compatible with other inducing point techniques as well, as the algorithm is defined by a matrix-vector multiplication routine. For example, one could apply LOVE to SGPR using the efficient MVM described in Section 3.4. Since SGPR makes a rank $M \ll N$ approximation to the kernel, setting $J = M$ recovers the $\mathcal{O}(NM^2)$ precomputation time and $\mathcal{O}(M^2)$ prediction time of standard SGPR variances [Titsias, 2009].

Ensuring Lanczos solves are accurate. Given a matrix $\widehat{\mathbf{K}}_{\mathbf{XX}}$, the Lanczos decomposition $\mathbf{QTQ}^\top$ is designed to approximate the solve $\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{b}$, where $\mathbf{b}$ is the first column of $\mathbf{Q}$. As argued in Section 2.3.2, the $\mathbf{Q}$ and $\mathbf{T}$ matrices can usually be re-used to approximate the solves $\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}(\mathbf{W}_\mathbf{X}^\top\mathbf{K}_{\mathbf{ZZ}}) \approx \mathbf{QT}^{-1}\mathbf{Q}^\top(\mathbf{W}_\mathbf{X}^\top\mathbf{K}_{\mathbf{ZZ}})$. This property of the Lanczos algorithm is why LOVE can compute fast predictive variances. While this method usually produces accurate solves, the solves will not be accurate if some columns of $(\mathbf{W}_\mathbf{X}^\top\mathbf{K}_{\mathbf{ZZ}})$ are (nearly) orthogonal to the columns of $\mathbf{Q}$. Importantly, we can easily check the accuracy of the solves by measuring the residual norm:

$$\|\widehat{\mathbf{K}}_{\mathbf{XX}} \underbrace{(\mathbf{RR}^\top)}_{\approx\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}} \mathbf{k}_{\mathbf{Xx}^*} - \mathbf{k}_{\mathbf{Xx}^*}\|_2.$$

Note that this convergence check only requires an additional MVM with $\widehat{\mathbf{K}}_{\mathbf{XX}}$. If the residual is not sufficiently small for some vector $\mathbf{k}_{\mathbf{Xx}^*}$, we can run conjugate gradients to refine the solve $\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{k}_{\mathbf{Xx}^*}$ using $\mathbf{RR}^\top\mathbf{k}_{\mathbf{Xx}^*}$ as an initial starting vector. In practice, we find that these countermeasures are almost never necessary with LOVE—the Lanczos solves are almost always accurate.

Numerical stability of Lanczos. A practical concern for LOVE is round-off errors that may affect the Lanczos algorithm. In particular it is common for the vectors of $\mathbf{Q}$ to lose orthogonality [Paige, 1970, Simon, 1984, Golub and Van Loan, 2012], resulting in an incorrect decomposition. To correct for this, several methods such as full reorthogonalization and partial or selective reorthogonalization exist [e.g. Golub and Van Loan, 2012]. In our implementation, we use full reorthogonalization when a loss of orthogonality is detected. In practice, the cost of this correction is absorbed by the parallel performance of the GPU and does not impact the final running time.

Sampling without KISS-GP. LOVE in conjunction with KISS-GP makes it possible to efficiently draw samples from a GP posterior. This has potential in many applications like Bayesian optimization and model-based reinforcement learning [e.g., Deisenroth and Rasmussen, 2011, Hernández-Lobato et al., 2014, Wang and Jegelka, 2017], which typically rely on parametric approximations or finite basis approaches for approximate sampling. However, it is worth noting that this sampling technique cannot be applied to exact GP models, as the derivation of LOVE sampling requires an inducing point approximation to the prior test covariance $\mathbf{K}_{\mathbf{X}^*\mathbf{X}^*}$ (see Eq. (4.8)). In the next chapter we will address this limitation and introduce a $\mathcal{O}(N^2)$ sampling algorithm for exact Gaussian process models.

A complete MVM-based framework for Gaussian process regression. These past two chapters have presented MVM-based methods for training GPs (using BBMM) and computing predictive distributions (using LOVE). Both of these algorithms can be readily applied to regression models with Gaussian likelihoods. In Chapter 6 we will utilize these methods to push beyond current limits of exact GP models. Before discussing those results, we will first introduce one final MVM-based method for non-conjugate GP models (e.g. classification GPs).

CHAPTER 5

VARIATIONAL GAUSSIAN PROCESSES INFERENCE AND BAYESIAN OPTIMIZATION VIA CONTOUR INTEGRAL QUADRATURE

5.1 Introduction

The BBMM and LOVE algorithms address training and predictions with GP regression models. However, other GP tasks—such as black-box optimization and non-conjugate inference (e.g. classification)—require two additional kernel matrix operations.

- **Sampling** from GP posteriors—as described in Section 4.4.2—is a common inference operation especially in the context of Bayesian optimization [Thompson, 1933, Frazier et al., 2009, Wang and Jegelka, 2017]. Since we cannot directly sample a function $f(\cdot) \sim p(f(\cdot) \mid \mathcal{D})$, it is more common to draw samples from $f(\cdot)$ evaluated on a finite test set $\mathbf{X}^* = [\mathbf{x}_1^*, \ldots, \mathbf{x}_T^*]$ If $\mathcal{N}[\boldsymbol{\mu}^*, \mathbf{COV}^*]$ is the GP posterior evaluated on $\mathbf{X}^*$, then we can draw samples $\boldsymbol{\epsilon}'$ via the reparameterization trick [Kingma and Welling, 2014, Rezende et al., 2014]:

$$\boldsymbol{\epsilon} = \boldsymbol{\mu}^* + \left(\mathbf{COV}^{*\frac{1}{2}}\right) \boldsymbol{\epsilon}'. \tag{5.1}$$

  where $\boldsymbol{\epsilon}' \sim \mathcal{N}[\mathbf{0}, \mathbf{I}]$ is a standard normal vector.

- **"Whitening"** is essentially the inverse of the sampling operation—and it is often necessary for non-conjugate GP approximations. If $\mathbf{b}$ is a random variable with mean $\boldsymbol{\mu}$ and covariance $\mathbf{K}$ (e.g. a sample from a GP prior), then the whitening operation gives us a coordinate transformation

$$\mathbf{b}' = \mathbf{K}^{-\frac{1}{2}} (\mathbf{b} - \boldsymbol{\mu}) \tag{5.2}$$

so that the resultant vector $\mathbf{b}'$ has zero mean and unit covariance. As we will discuss in Section 5.4.1, the whitening operation can significantly accelerate the convergence of variational Gaussian process inference [Kuss and Rasmussen, 2005, Hensman et al., 2013, Matthews, 2017].

Note that Eqs. (5.1) and (5.2) both require applying the matrix square root (or its inverse) to a vector: $\mathbf{K}^{1/2}\mathbf{b}$. In practice, $\mathbf{K}^{1/2}$ can be replaced with any matrix $\mathbf{R}$ such that $\mathbf{R}\mathbf{R}^\top = \mathbf{K}$. All such $\mathbf{R}$ matrices are equivalent to $\mathbf{K}^{1/2}$ up to an orthonormal rotation [Golub and Van Loan, 2012].

Existing methods for sampling and whitening typically rely on the Cholesky factorization: $\mathbf{K} = \mathbf{L}\mathbf{L}^\top$. However, computing the Cholesky factor requires $\mathcal{O}(N^3)$ time and $\mathcal{O}(N^2)$ memory for an $N \times N$ covariance matrix $\mathbf{K}$. To avoid this large complexity, randomized algorithms [Rahimi and Recht, 2008, Mutny and Krause, 2018], low-rank/sparse approximations [Hensman et al., 2017, Wilson et al., 2020], or alternative distributions [Wang and Jegelka, 2017] are often used to approximate the sampling and whitening operations. The previous chapter offers an efficient sampling method for KISS-GP models; however, this method is not generally applicable to other classes of GP models (see Section 4.4.2).

In this chapter, we make several contributions to address these issues:

- We introduce a MVM approach for computing $\mathbf{K}^{\pm 1/2}\mathbf{b}$. The approach uses an insight from Hale et al. [2008] that expresses the matrix square root as a sum of shifted matrix inverses.
- To efficiently compute these shifted inverses, we leverage a modified version of the MINRES algorithm [Paige and Saunders, 1975] that performs *multiple shifted solves* through a single iteration of MVMs. We demonstrate

that, surprisingly, **multi-shift MINRES (msMINRES)** convergence can be accelerated with a *single* preconditioner despite the presence of multiple shifts. Achieving 4 or 5 decimal places of accuracy typically requires *fewer than 100 matrix-vector multiplications*, which can be highly accelerated through GPUs.

- We derive a scalable backward pass for $\mathbf{K}^{\pm 1/2}\mathbf{b}$ that enables our approach to be used as part of learning and optimization.
- We apply our $\mathbf{K}^{-1/2}\mathbf{b}$ and $\mathbf{K}^{1/2}\mathbf{b}$ routines to two applications: **(1)** variational Gaussian processes with up to $M = 10^4$ inducing points (where we additionally introduce a $\mathcal{O}(M^2)$ MVM-based natural gradient update); and **(2)** sampling from Gaussian process posteriors in the context of Bayesian optimization with up to $50{,}000$ candidate points.

5.2 Contour Integral Quadrature (CIQ) via Matrix-Vector Multiplication

In this section we develop a MVM method to compute $\mathbf{K}^{\pm 1/2}\mathbf{b}$ for sampling and whitening. Our approach scales better than existing methods (e.g. Cholesky) by: **(1)** reducing computation from $\mathcal{O}(N^3)$ to $\mathcal{O}(N^2)$; **(2)** more effectively using GPU acceleration; and **(3)** affording an efficient gradient computation.

Contour Integral Quadrature (CIQ) A standard result from complex analysis is that $\mathbf{K}^{-1/2}$ can be expressed through Cauchy's integral formula:

$$\mathbf{K}^{-1/2} = \frac{1}{2\pi i} \oint_{\Gamma} \tau^{-1/2} \left(\tau \mathbf{I} - \mathbf{K}\right)^{-1} \, \mathrm{d}\tau,$$

where Γ is a closed contour in the complex plane that winds once around the spectrum of $\mathbf{K}^{-\frac{1}{2}}$ [Davies and Higham, 2005, Hale et al., 2008, Higham, 2008]. Applying a numerical quadrature scheme to the contour integral yields the rational approximations

$$\mathbf{K}^{-\frac{1}{2}} \approx \sum_{q=1}^{Q} w_q \left(t_q \mathbf{I} + \mathbf{K}\right)^{-1} \quad \text{and} \quad \mathbf{K}^{\frac{1}{2}} \approx \mathbf{K} \sum_{q=1}^{Q} w_q \left(t_q \mathbf{I} + \mathbf{K}\right)^{-1}, \tag{5.3}$$

where the weights w_q encapsulate the normalizing constant, quadrature weights, and the $t_q^{-\frac{1}{2}}$ terms. Hale et al. [2008] introduce a real-valued quadrature strategy based on a change-of-variables formulation (described in Appendix B) that converges extremely rapidly—often achieving full machine precision with only $Q \approx 20$ quadrature points. For the remainder of this chapter, applying Eq. (5.3) to compute $\mathbf{K}^{\pm 1/2}\mathbf{b}$ will be referred to as **Contour Integral Quadrature (CIQ)**.

5.2.1 An Efficient Matrix-Vector Multiplication Approach to CIQ with msMINRES

Using the quadrature method of Eq. (5.3) for whitening and sampling requires solving several shifted linear systems. To compute the shifted solves required by Eq. (5.3) we leverage a variant of the minimum residuals algorithm (MINRES) developed by Paige and Saunders [1975] (see Section 2.3.4).

msMINRES for multiple shifted solves. To efficiently compute *all* the shifted solves, we leverage techniques [e.g. Datta and Saad, 1991, Freund, 1990, Meerbergen, 2003] that exploit the shift-invariance property of Krylov subspaces:

i.e. $\mathcal{K}_J(\mathbf{K}, \mathbf{b}) = \mathcal{K}_J(t\mathbf{I} + \mathbf{K}, \mathbf{b})$. We introduce a variant of MINRES, which we refer to as **multi-shift MINRES** or **msMINRES**, that re-uses the same Krylov subspace vectors $[\mathbf{b},\ \mathbf{K}\mathbf{b},\ \ldots,\ \mathbf{K}^{J-1}\mathbf{b}]$ for all shifted solves $(t\mathbf{I} + \mathbf{K})^{-1}\mathbf{b}$. In other words, using msMINRES we can get all $(t_q\mathbf{I} + \mathbf{K})^{-1}\mathbf{b}$ *essentially for free*, i.e. only requiring J MVMs for the Krylov subspace $\mathcal{K}_J(\mathbf{K}, \mathbf{b})$. As with standard MINRES, the msMINRES procedure for computing $(t_q\mathbf{I} + \mathbf{K})^{-1}$ from $[\mathbf{b},\ \mathbf{K}\mathbf{b},\ \ldots,\ \mathbf{K}^{J-1}\mathbf{b}]$ can be reduced to a simple vector recurrence.

First, recall from Section 2.3.4 that the MINRES solution for $\mathbf{K}^{-1}\mathbf{b}$ can be formed through Lanczos (see Eq. (2.33)):

$$\mathbf{K}^{-1}\mathbf{b} \approx \|\mathbf{b}\|_2\, \mathbf{Q}_J \left(\mathbf{R}_J^{-1}\boldsymbol{\mathcal{Q}}_J^\top\right) \mathbf{e}_1, \quad \boldsymbol{\mathcal{Q}}_J\mathbf{R}_J = \begin{bmatrix} \mathbf{T}_J \\ \|\mathbf{r}_J\|_2\mathbf{e}_J^\top \end{bmatrix}, \tag{5.4}$$

where $\mathbf{Q}_J$, $\mathbf{T}_J$, and $\mathbf{r}_J$ are the outputs from J steps of the Lanczos algorithm with initial vector $\mathbf{b}$:

$$\mathbf{K}\mathbf{Q}_J = \mathbf{Q}_J\mathbf{T}_J + \mathbf{r}_J\mathbf{e}_J^\top.$$

To adapt MINRES to multiple shifts (i.e. msMINRES), we exploit a well-established fact about the shift invariance of Krylov subspaces:

Observation 5.1. *Let* $\mathbf{K}\mathbf{Q}_J = \mathbf{Q}_J\mathbf{T}_J + \mathbf{r}_J\mathbf{e}_J^\top$ *be the Lanczos factorization for* $\mathbf{K}$ *given the initial vector* $\mathbf{b}$*. Then*

$$(\mathbf{K} + t\mathbf{I})\mathbf{Q}_J = \mathbf{Q}_J(\mathbf{T}_J + t\mathbf{I}) + \mathbf{r}_J\mathbf{e}_J^\top$$

is the Lanczos factorization for matrix $(\mathbf{K} + t\mathbf{I})$ *with initial vector* $\mathbf{b}$*.*

Consequently, we can re-use the $\mathbf{Q}_J$ and $\mathbf{T}_J$ Lanczos matrices to compute *multiple shifted solves.*

$$(\mathbf{K} + t\mathbf{I})^{-1}\mathbf{b} \approx \|\mathbf{b}\|_2\, \mathbf{Q}_J \left(\mathbf{R}_J^{(t)-1}\boldsymbol{\mathcal{Q}}_J^{(t)\top}\right) \mathbf{e}_1, \quad \boldsymbol{\mathcal{Q}}_J^{(t)}\mathbf{R}_J^{(t)} = \begin{bmatrix} \mathbf{T}_J + t\mathbf{I} \\ \|\mathbf{r}_J\|_2\mathbf{e}_J^\top \end{bmatrix}, \tag{5.5}$$

Assuming $\mathbf{Q}$ and $\mathbf{T}$ have been previously computed, Eq. (5.5) requires no additional MVMs with $\mathbf{K}$. We refer to this multi-shift formulation as **Multi-Shift MINRES**, or **msMINRES**.

A simple vector recurrence for msMINRES. Just as with standard MINRES, Eq. (5.5) can also be computed via a vector recurrence. We can derive a msMINRES algorithm simply by modifying the existing MINRES recurrence. Before the QR step in Algorithm 2.5, we add t to the Lanczos diagonal terms ($\gamma_j + t$, where $\gamma_j = T^{(j,j)}$). This can be extended to simultaneously handle *multiple shifts* $t_1, \ldots, t_Q$. Each shift would compute its own QR factorization, its own step size $\varphi_j^{(t_q)}$, and its own search vector $\mathbf{d}_j^{(t_q)}$. However, all shifts share the same Lanczos vectors $\mathbf{q}_j$ and therefore share the same MVMs. The operations for each shift can be vectorized for efficient parallelization.

To summarize: the resulting algorithm—msMINRES—gives us approximations to $(t_1\mathbf{I}+\mathbf{K})^{-1}\mathbf{b}$, …, $(t_Q\mathbf{I}+\mathbf{K})^{-1}$ *essentially for free* by leveraging the information we needed anyway to compute $\mathbf{K}^{-1}\mathbf{b}$. The full vector recurrence is outlined in Algorithm 5.1. Its computational properties will be highlighted below.

5.2.2 Computational Complexity and Convergence Analysis of msMINRES-CIQ

Pairing Eq. (5.3) with msMINRES is an efficient algorithm for computing $\mathbf{K}^{1/2}\mathbf{b}$ and $\mathbf{K}^{-1/2}\mathbf{b}$. Algorithm 5.2 summarizes this approach; below we highlight its computational properties:

Property 5.1 (Computation/Memory of msMINRES-CIQ)**.** *J iterations of*

Algorithm 5.1: Multi-shift MINRES (msMINRES). Differences from MINRES (Alg. 2.5) are in green. Green `for` loops are parallelizable.

Input : `mvm_K(·)` – function for MVM with matrix $\mathbf{K}$
$\mathbf{b}$ – vector to solve against
$t_1, \ldots, t_Q$ – shifts
Output: $\mathbf{c}_1 = (\mathbf{K} + t_1)^{-1}\mathbf{b}, \ldots, \mathbf{c}_Q = (\mathbf{K} + t_Q)^{-1}\mathbf{b}$.

$\mathbf{q}_1 \leftarrow \mathbf{b}/\|\mathbf{b}\|_2$ // Current Lanczos vector.
$\mathbf{v}_1 \leftarrow$ `mvm_K(` $\mathbf{q}_0$ `)` // Buffer for MVM output.
$\delta_1 \leftarrow \|\mathbf{b}\|_2$, $\delta_0 \leftarrow 1$ // Current/prev. Lanczos residual/sub-diagonal.
for $q \leftarrow 1$ **to** Q **do**
  $\mathbf{c}_1^{(q)} \leftarrow \mathbf{0}$ // Current solution.
  $\mathbf{d}_1^{(q)}, \mathbf{d}_0^{(q)} \leftarrow \mathbf{0}$ // Current & prev. ''search'' direction.
  $\varphi_2^{(q)} \leftarrow \|\mathbf{b}\|_2$ // Current ''step'' size.
  $\eta_1^{(q)} \leftarrow 1$, $\eta_0^{(q)} \leftarrow 0$ // Current/prev. scaling term.
end
for $j \leftarrow 2$ **to** J **do**
  $\mathbf{q}_j \leftarrow \mathbf{v}_j/\delta_j$
  $\mathbf{v}_j \leftarrow$ `mvm_K(` $\mathbf{q}_j$ `)` $-\delta_j \mathbf{q}_{j-1}$
  $\gamma_j \leftarrow \mathbf{q}_j \mathbf{v}_j$
  $\mathbf{v}_j \leftarrow \mathbf{v}_j - \gamma_j \mathbf{q}_j$
  $\delta_j \leftarrow \|\mathbf{v}_j\|$
  for $q \leftarrow 1$ **to** Q **do**
    $\epsilon_j^{(q)} \leftarrow \delta_{j-1}\left(\delta_{j-2}/\sqrt{\delta_{j-2}^2 + \eta_{j-2}^{(q)2}}\right)$
    $\zeta_j^{(q)} \leftarrow \delta_{j-1}\left(\eta_{j-2}^{(q)}/\sqrt{\delta_{j-2}^2 + \eta_{j-2}^{(q)2}}\right)$
    $\eta_j^{(q)} \leftarrow (\gamma_j + t_q)\left(\eta_{j-1}^{(q)}/\sqrt{\delta_{j-1}^2 + \eta_{j-1}^{(q)2}}\right) + \zeta_j^{(q)}\left(\delta_{j-1}/\sqrt{\delta_{j-1}^2 + \eta_{j-1}^{(q)2}}\right)$
    $\zeta_j^{(q)} \leftarrow \zeta_j^{(q)}\left(\eta_{j-1}^{(q)}/\sqrt{\delta_{j-1}^2 + \eta_{j-1}^{(q)2}}\right) + (\gamma_j + t_q)\left(\delta_{j-1}/\sqrt{\delta_{j-1}^2 + \eta_{j-1}^{(q)2}}\right)$
    $\eta_j^{(q)} \leftarrow \eta_j^{(q)}\left(\eta_j^{(q)}/\sqrt{\delta_j^2 + \eta_j^{(q)2}}\right)$
    $\varphi_j^{(q)} \leftarrow \varphi_{j-1}^{(q)}\left(\delta_{j-1}/\sqrt{\delta_{j-1}^2 + \eta_{j-1}^{(q)2}}\right)\left(\eta_j^{(q)}/\sqrt{\delta_j^2 + \eta_j^{(q)2}}\right)$
    $\mathbf{d}_j^{(q)} \leftarrow \left(\mathbf{q} - \zeta_j^{(q)}\mathbf{d}_{j-1}^{(q)} - \epsilon_j^{(q)}\mathbf{d}_{j-2}^{(q)}\right)/\eta_j^{(q)}$
    $\mathbf{c}_j^{(q)} \leftarrow \mathbf{c}_{j-1}^{(q)} + \varphi_j^{(q)}\mathbf{d}_j^{(q)}$
  end
end
return $\|\mathbf{b}\|_2\, \mathbf{c}_j$

Algorithm 5.2: Computing $\mathbf{K}^{-\frac{1}{2}}\mathbf{b}$ with MVM-based Contour Integral Quadrature (CIQ).

Input : `mvm_K(·)` – function for a matrix-vector multiplication (MVM) with matrix $\mathbf{K}$
$\mathbf{b}$ – right hand side
J – number of `msMINRES` iterations
Q – number of quad. points

Output: $\mathbf{a} \approx \mathbf{K}^{-\frac{1}{2}}\mathbf{b}$

$[w_1, \ldots, w_Q], [t_1, \ldots, t_Q] \leftarrow$ `compute_quad( mvm_K(·), Q)` `// Weights` (w_i) `and shifts` (t_i) `for quadrature -- details in Appendix B.`

$(t_1\mathbf{I} + \mathbf{K})^{-1}\mathbf{b}, \ldots (t_Q\mathbf{I} + \mathbf{K})^{-1}\mathbf{b} \leftarrow$ `msMINRES( mvm_K(·), b,` $J, t_1, \ldots, t_Q$`)`
`// Shifted MINRES computes all solves simultaneously.`

return $\sum_{q=1}^{Q} w_q \left(t_q\mathbf{I} + \mathbf{K}\right)^{-1}\mathbf{b}$ `// CIQ estimate of` $\frac{1}{2\pi i}\int \tau^{-1/2}(\tau\mathbf{I} - \mathbf{K})^{-1}\mathbf{b}\,\mathrm{d}\tau = \mathbf{K}^{-1/2}\mathbf{b}$

msMINRES requires exactly J MVMs with the input matrix $\mathbf{K}$, regardless of the number of quadrature points Q. The resulting runtime of msMINRES-CIQ is $\mathcal{O}(J\xi(\mathbf{K}))$, where $\xi(\mathbf{K})$ is the time to perform a MVM with $\mathbf{K}$. The memory requirement is $\mathcal{O}(QN)$ in addition to what is required to store $\mathbf{K}$.

For arbitrary positive semi-definite $N \times N$ matrices, the runtime of msMINRES-CIQ is $\mathcal{O}(JN^2)$, where often $J \ll N$. Below we bound its error:

Theorem 5.1. *Let $\mathbf{K} \succ 0$ and $\mathbf{b}$ be inputs to msMINRES-CIQ, producing $\mathbf{a}_J \approx \mathbf{K}^{1/2}\mathbf{b}$ after J iterations with Q quadrature points. The difference between $\mathbf{a}_J$ and $\mathbf{K}^{1/2}\mathbf{b}$ is bounded by:*

$$\left\|\mathbf{a}_J - \mathbf{K}^{\frac{1}{2}}\mathbf{b}\right\|_2 \leq \overbrace{\mathcal{O}\left(\exp\left(-\frac{2Q\pi^2}{\log\kappa(\mathbf{K}) + 3}\right)\right)}^{\textit{Quadrature error}} + \overbrace{\frac{2Q\log\left(5\sqrt{\kappa(\mathbf{K})}\right)\kappa(\mathbf{K})\sqrt{\lambda_{min}}}{\pi}\left(\frac{\sqrt{\kappa(\mathbf{K})} - 1}{\sqrt{\kappa(\mathbf{K})} + 1}\right)^{J-1}\|\mathbf{b}\|_2}^{\textit{msMINRES error}}.$$

where $\lambda_{max}, \lambda_{min}$ are the max and min eigenvalues of $\mathbf{K}$, and $\kappa(\mathbf{K}) \equiv \frac{\lambda_{max}}{\lambda_{min}}$ is the condition number.

For $\mathbf{a}'_J \approx \mathbf{K}^{-1/2}\mathbf{b}$, the bound incurs an additional factor of $1/\lambda_{\min}$. (See Appendix B.2 for proofs.) Theorem 5.1 suggests that error in computing $(t_q\mathbf{I} + \mathbf{K})^{-1}\mathbf{b}$ will be the primary source of error as the quadrature error decays rapidly with Q. In many of our applications the rapid convergence of Krylov subspace methods for linear solves is well established, allowing for accurate solutions if desired. For covariance matrices up to $N = 50{,}000$, often $Q = 8$ and $J \leq 100$ suffices for 4 decimal places of accuracy and J can be further reduced with preconditioning (see Section 5.2.4).

5.2.3 Efficient Vector-Jacobi Products for Backpropagation

In certain applications, such as variational Gaussian process inference, we have to compute gradients of the $\mathbf{K}^{-1/2}\mathbf{b}$ operation. This requires the vector-Jacobian product $\mathbf{v}^\top(\partial\mathbf{K}^{-1/2}\mathbf{b}/\partial\mathbf{K})$, where $\mathbf{v}$ is the back-propagated gradient. The form of the Jacobian is the solution to a Lyapunov equation, which requires expensive iterative methods or solving a $N^2 \times N^2$ Kronecker sum $(\mathbf{K}^{1/2} \oplus \mathbf{K}^{1/2})^{-1}$. Both of these options are much slower than the forward pass and are impractical for large N. Fortunately, our quadrature formulation affords a computationally efficient approximation to this vector-Jacobian product. If we back-propagate directly through each term in Eq. (5.3), we have

$$\begin{aligned}
\mathbf{v}^\top \left(\frac{\partial \mathbf{K}^{-1/2}\mathbf{b}}{\partial \mathbf{K}}\right) &= \frac{\partial \mathbf{v}^\top \mathbf{K}^{-1/2}\mathbf{b}}{\partial \mathbf{K}} \\
&\approx \frac{\partial \sum_{q=1}^{Q} w_q \left(\mathbf{v}^\top (t_q\mathbf{I} + \mathbf{K})^{-1}\mathbf{b}\right)}{\partial \mathbf{K}} = \\
&\approx -\frac{1}{2}\sum_{q=1}^{Q} w_q \left(t_q\mathbf{I} + \mathbf{K}\right)^{-1} \left(\mathbf{v}\mathbf{b}^\top + \mathbf{b}\mathbf{v}^\top\right) \left(t_q\mathbf{I} + \mathbf{K}\right)^{-1}. \qquad (5.6)
\end{aligned}$$

Since the forward pass computes the solves with $\mathbf{b}$, the only additional work needed for the backward pass is computing the shifted solves $(t_q\mathbf{I} + \mathbf{K})^{-1}\mathbf{v}$, which can be computed with another call to the msMINRES algorithm. Thus the backward pass takes only $\mathcal{O}(J\xi(\mathbf{K}))$ (e.g. $\mathcal{O}(JN^2)$) time.

Programmability. As with BBMM and LOVE, msMINRES-CIQ can take full advantage of GPyTorch's `LazyTensor` construct. The algorithm for msMINRES-CIQ (Algorithm 5.2) accesses $\mathbf{K}$ through Lanczos and msMINRES—each of which only requires black-box access to a MVM routine. Therefore, msMINRES-CIQ can be implemented for specialty GP models using the efficient `_matmul` function of `LazyTensor`s. Moreover, back-propagation via Eq. (5.6) can be implemented using the `_deriv` function of LazyTensors. Recall that the `_deriv(`$\mathbf{A}, \mathbf{B}$`)` function computes $\partial \operatorname{Tr}(\mathbf{A}^\top \mathbf{K}\mathbf{B})/\partial \mathbf{r}$, where $\mathbf{r}$ is a representation of $\mathbf{K}$. If we define $\mathbf{A}$ and $\mathbf{B}$ as

$$\mathbf{A} = \begin{bmatrix} w_1 \left(t_q 1\mathbf{I} - \mathbf{K}\right)^{-1} \mathbf{v} & \dots & w_Q \left(t_Q \mathbf{I} - \mathbf{K}\right)^{-1} \mathbf{v} \end{bmatrix},$$
$$\mathbf{B} = \begin{bmatrix} \left(t_q 1\mathbf{I} - \mathbf{K}\right)^{-1} \mathbf{b} & \dots & \left(t_Q \mathbf{I} - \mathbf{K}\right)^{-1} \mathbf{b} \end{bmatrix},$$

then the vector-Jacobi product in Eq. (5.6) can be re-written as $\operatorname{Tr}(\mathbf{A}\mathbf{B}^\top) =$ `_deriv(`$\mathbf{A}, \mathbf{B}$`)`. This allows us to easily implement efficient CIQ variants for structured matrices.

5.2.4 Preconditioning

To improve the convergence of Theorem 5.1, we can introduce a preconditioner $\mathbf{P}$ where $\mathbf{P}^{-1}\mathbf{K} \approx \mathbf{I}$. For standard MINRES, applying a preconditioner

is straightforward. We simply use MINRES to solve the system

$$\left(\mathbf{P}^{-1/2}\mathbf{K}\mathbf{P}^{-1/2}\right)\mathbf{P}^{1/2}\mathbf{c} = \mathbf{P}^{-1/2}\mathbf{b},$$

which has the same solution $\mathbf{c}$ as the original system. In practice the preconditioned MINRES vector recurrence does not need access to $\mathbf{P}^{-1/2}$—it only needs access to $\mathbf{P}^{-1}$ (see [Choi, 2006, Ch. 3.4] for details).

However, it is not immediately straightforward to apply preconditioning to msMINRES, as preconditioners break the shift-invariance property that is necessary for the $\mathcal{O}(JN^2)$ shifted solves [Jegerlehner, 1996, Aune et al., 2013]. More specifically, if we apply $\mathbf{P}$ to msMINRES, then we obtain the solves

$$\mathbf{P}^{-1/2}(\mathbf{P}^{-1/2}\mathbf{K}\mathbf{P}^{-1/2} + t_q\mathbf{I})^{-1}(\mathbf{P}^{-1/2}\mathbf{b}).$$

Plugging these shifted solves into the quadrature equation Eq. (5.3) therefore gives us

$$\widetilde{\mathbf{a}}_J \approx \mathbf{P}^{-\frac{1}{2}}(\mathbf{P}^{-\frac{1}{2}}\mathbf{K}\mathbf{P}^{-\frac{1}{2}})^{-\frac{1}{2}}(\mathbf{P}^{-\frac{1}{2}}\mathbf{b}). \tag{5.7}$$

In general, we cannot recover $\mathbf{K}^{-1/2}$ from Eq. (5.7). Nevertheless, we can still obtain preconditioned solutions that are equivalent to $\mathbf{K}^{-1/2}\mathbf{b}$ and $\mathbf{K}^{1/2}\mathbf{b}$ up to an orthogonal rotation. Let $\mathbf{R} = \mathbf{K}\mathbf{P}^{-1/2}(\mathbf{P}^{-1/2}\mathbf{K}\mathbf{P}^{-1/2})^{-1/2}$. We have that

$$\mathbf{R}\mathbf{R}^\top = \mathbf{K}\left(\mathbf{P}^{-\frac{1}{2}}(\mathbf{P}^{-\frac{1}{2}}\mathbf{K}\mathbf{P}^{-\frac{1}{2}})^{-\frac{1}{2}}\right)\left((\mathbf{P}^{-\frac{1}{2}}\mathbf{K}\mathbf{P}^{-\frac{1}{2}})^{-\frac{1}{2}}\mathbf{P}^{-\frac{1}{2}}\right)\mathbf{K} = \mathbf{K}.$$

Thus $\mathbf{R}$ is equivalent to $\mathbf{K}^{1/2}$ up to orthogonal rotation. We can compute $\mathbf{R}\mathbf{b}$ (e.g. for sampling) by applying Eq. (5.7) to the initial vector $\mathbf{P}^{1/2}\mathbf{b}$:

$$\mathbf{R}\mathbf{b} = \mathbf{K}\underbrace{\left[\mathbf{P}^{-\frac{1}{2}}(\mathbf{P}^{-\frac{1}{2}}\mathbf{K}\mathbf{P}^{-\frac{1}{2}})^{-\frac{1}{2}}\mathbf{P}^{-\frac{1}{2}}\right]\left(\mathbf{P}^{\frac{1}{2}}\mathbf{b}\right)}_{\text{Applying preconditioned msMINRES to } \mathbf{P}^{1/2}\mathbf{b}}. \tag{5.8}$$

Similarly, $\mathbf{R}' = \mathbf{P}^{-1/2}\left(\mathbf{P}^{-1/2}\mathbf{K}\mathbf{P}^{-1/2}\right)^{-1/2}$ is equivalent to $\mathbf{K}^{-1/2}$ up to orthogonal rotation:

$$\mathbf{R}'\mathbf{R}'^\top = \left(\mathbf{P}^{-\frac{1}{2}}(\mathbf{P}^{-\frac{1}{2}}\mathbf{K}\mathbf{P}^{-\frac{1}{2}})^{-\frac{1}{2}}\right)\left((\mathbf{P}^{-\frac{1}{2}}\mathbf{K}\mathbf{P}^{-\frac{1}{2}})^{-\frac{1}{2}}\mathbf{P}^{-\frac{1}{2}}\right) = \mathbf{K}^{-1}.$$

We can compute $\mathbf{R}'\mathbf{b}$ (e.g. for whitening) via:

$$\mathbf{R}'\mathbf{b} = \underbrace{\left[\mathbf{P}^{-\frac{1}{2}}(\mathbf{P}^{-\frac{1}{2}}\mathbf{K}\mathbf{P}^{-\frac{1}{2}})^{-\frac{1}{2}}\mathbf{P}^{-\frac{1}{2}}\right]\left(\mathbf{P}^{\frac{1}{2}}\mathbf{b}\right)}_{\text{Applying preconditioned msMINRES to } \mathbf{P}^{1/2}\mathbf{b}}. \tag{5.9}$$

Crucially, the convergence of Eqs. (5.8) and (5.9) depends on the conditioning $\kappa(\mathbf{P}^{-1}\mathbf{K}) \ll \kappa(\mathbf{K})$.

As with standard MINRES, msMINRES only requires access to $\mathbf{P}^{-1}$, not $\mathbf{P}^{-1/2}$. Note however that Eqs. (5.8) and (5.9) both require multiplies with $\mathbf{P}^{1/2}$. If a preconditioner $\mathbf{P}$ does not readily decompose into $\mathbf{P}^{1/2}\mathbf{P}^{1/2}$, we can simply run the CIQ algorithm on $\mathbf{P}$ to compute $\mathbf{P}^{1/2}\mathbf{b}$. Thus our requirements for a preconditioner are: **(1)** it affords efficient solves (ideally $o(N^2)$), and **(2)** it affords efficient MVMs (also ideally $o(N^2)$) for computing $\mathbf{P}^{1/2}\mathbf{b}$ via msMINRES-CIQ. Note that these requirements are met by the partial pivoted Cholesky preconditioner proposed in Section 3.3.

5.2.5 Related Work

Other Krylov methods for $\mathbf{K}^{1/2}\mathbf{b}$ and $\mathbf{K}^{-1/2}\mathbf{b}$, often via polynomial approximations [e.g. Higham, 2008], have been explored. Chow and Saad [2014] compute $\mathbf{K}^{1/2}\mathbf{b}$ via a preconditioned Lanczos algorithm. Unlike msMINRES, however, they require storage of the entire Krylov subspace. Moreover this approach does not afford a simple gradient computation. More similar to our work is [Aune et al., 2013, 2014], which uses the quadrature formulation of Eq. (5.3) in conjunction with a shifted conjugate gradients solver. We expand upon their method by: **(1)** introducing a simple gradient computation, **(2)** proving a convergence guarantee, and **(3)** enabling the use of simple preconditioners.

5.3 Benchmarking msMINRES-CIQ

In this section we empirically measure the convergence and speedup of msMINRES-CIQ applied to several types of covariance matrices.

Convergence of CIQ. In Fig. 5.1 we measure the relative error of computing $\mathbf{K}^{1/2}\mathbf{b}$ with msMINRES-CIQ on random matrices.[1] We vary **(1)** the number of quadrature points Q; **(2)** the size of the matrix N; and **(3)** the conditioning of the matrix. The figure displays results for matrices with spectra that decay as $\lambda_t = 1/\sqrt{t}$, $\lambda_t = 1/t$, $\lambda_t = 1/t^2$, and $\lambda_t = e^{-t}$, as well as for one-dimensional RBF and Matérn kernel matrices (formed with random data), which have near-exponentially decaying spectra. Consequently, the $1/\sqrt{t}$ matrices are relatively well-conditioned, while the RBF/Matérn kernels are relatively ill-conditioned. Nevertheless, in all cases msMINRES-CIQ achieves 10^{-4} relative error with only $Q = 8$ quadrature points, regardless of the size of the matrix. Approximation algorithms like randomized SVD on the other hand incur an order of magnitude more error (Fig. 5.2); a rank of $R = 1{,}024$ is unable to reduce the relative error to a single decimal point.

To further compare msMINRES-CIQ to randomized methods, Fig. 5.3 plots the empirical covariance matrix of $1{,}000$ Gaussian samples drawn from a Gaussian process prior $\mathcal{N}[\mathbf{0}, \mathbf{K}]$. We construct the RBF covariance matrices $\mathbf{K}$ using subsets of the Protein and Kin40k datasets [Asuncion and Newman, 2007]. We note that all methods incur some sampling error, regardless of the subset size

[1] msMINRES is stopped after achieving a relative residual of 10^{-4} or after reaching $J = 400$ iterations.

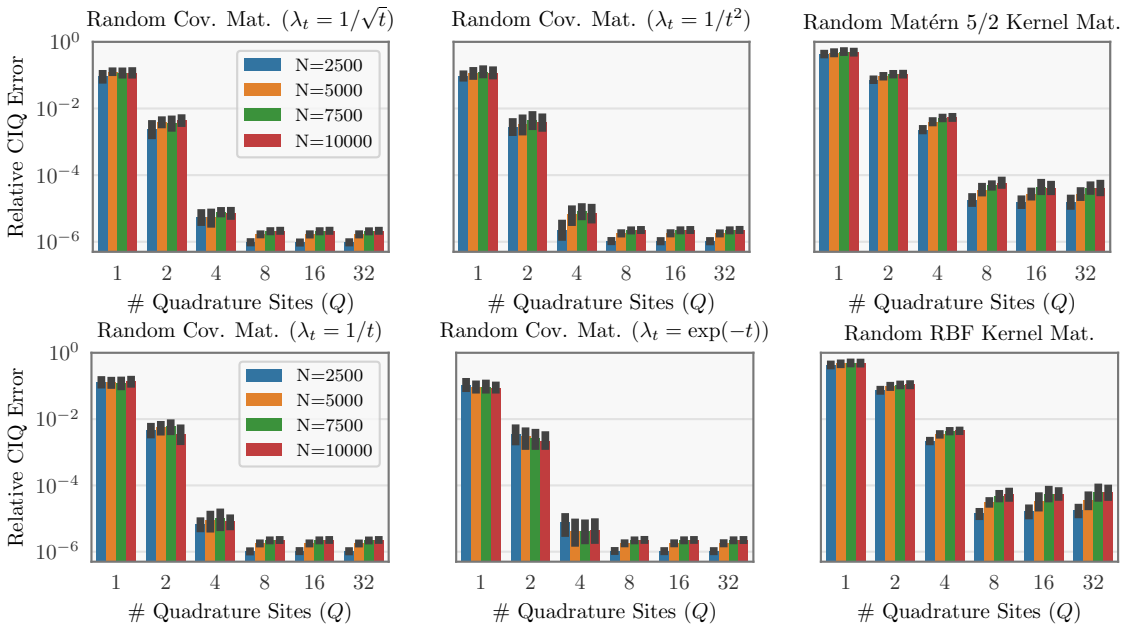

Figure 5.1: msMINRES-CIQ relative error at computing $\mathbf{K}^{1/2}\mathbf{b}$ as a function of number of quadrature points Q. We test random matrices with eigenvalues that scale as $\lambda_t = 1/\sqrt{t}$ (top left), $\lambda_t = 1/t$ (bottom left), $\lambda_t = 1/t^2$ (top middle), and $\lambda_t = e^{-t}$ (bottom middle). Additionally, we test random Matérn/RBF kernel matrices (top right/bottom right). In all cases $Q = 8$ achieves $< 10^{-4}$ error.

(N). msMINRES-CIQ and Cholesky-based sampling tend to have very similar empirical covariance error. On the other hand, the Random Fourier Features method [Rahimi and Recht, 2008] (with $1{,}000$ random features) incurs errors up to $2\times$ as large. This additional error is due to the randomness in the RFF approximation.

Typically $J = 100$ msMINRES iterations suffices for convergence; however this number can be lowered with preconditioning. To demonstrate this, we construct random $N \times N$ Matérn/RBF kernels $\mathbf{K}$, applying CIQ to a set of N orthonormal vectors ($[\mathbf{K}^{1/2}\mathbf{b}_1, \ldots, \mathbf{K}^{1/2}\mathbf{b}_N]$), and compute the empirical covariance. In Fig. 5.4 we plot the number of msMINRES iterations needed to achieve a relative error of 10^{-6}. The pivoted Cholesky preconditioner of Section 3.3—which forms a low-rank approximation of $\mathbf{K}$—accelerates conver-

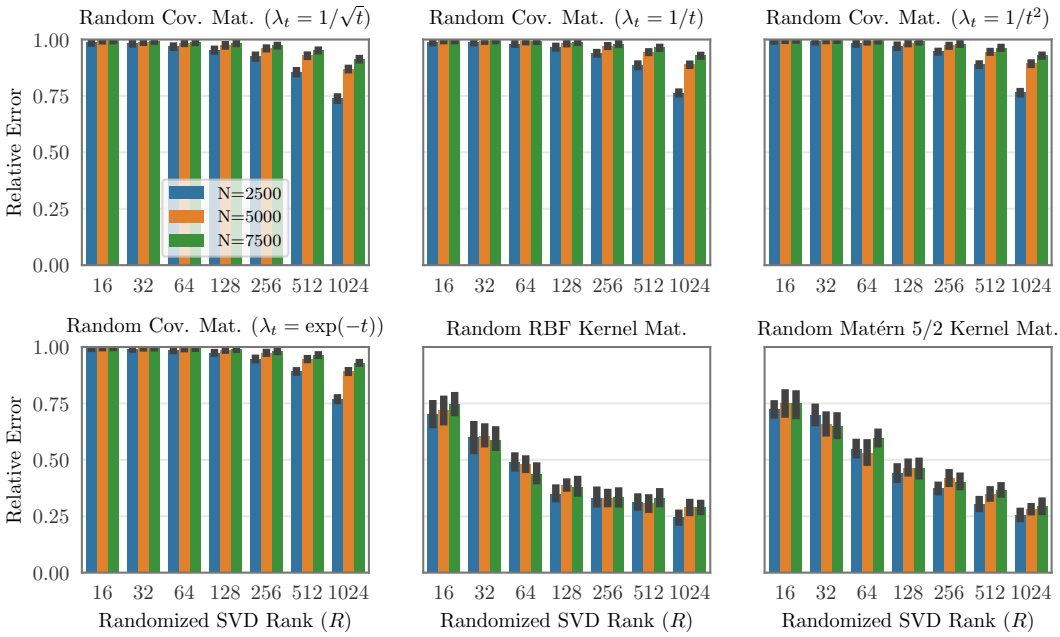

Figure 5.2: Randomized SVD relative error at computing $\mathbf{K}^{1/2}\mathbf{b}$ as a function of approximation rank R. In all cases, randomized SVD is unable to achieve a relative error better than about 0.25.

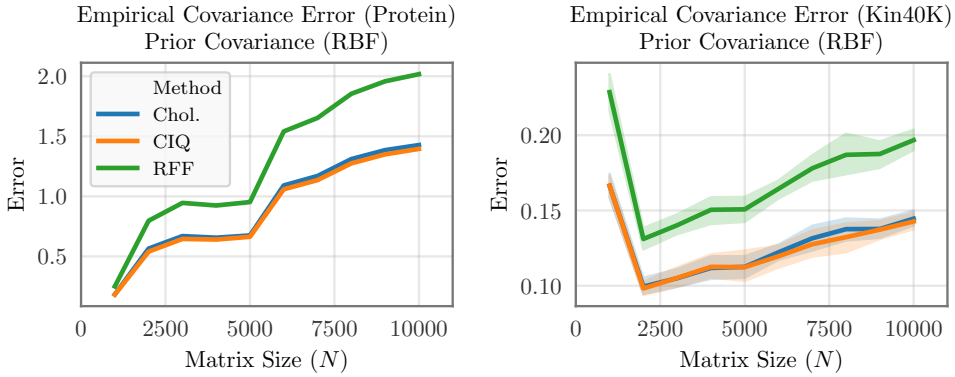

Figure 5.3: Empirical covariance error (relative norm) of various sampling methods (Cholesky, msMINRES-CIQ, and $1{,}000$ Random Fourier Features [Rahimi and Recht, 2008]). Empirical covariances are measured from $1{,}000$ samples. RBF matrices are constructed from data in the Protein and Kin40k datasets [Asuncion and Newman, 2007].

gence of msMINRES. Without preconditioning (i.e. rank=0), $J = 100$ iterations are required for $N = 7{,}500$ matrices. With rank-100/rank-400 preconditioners, iterations are cut by a factor of two/four.

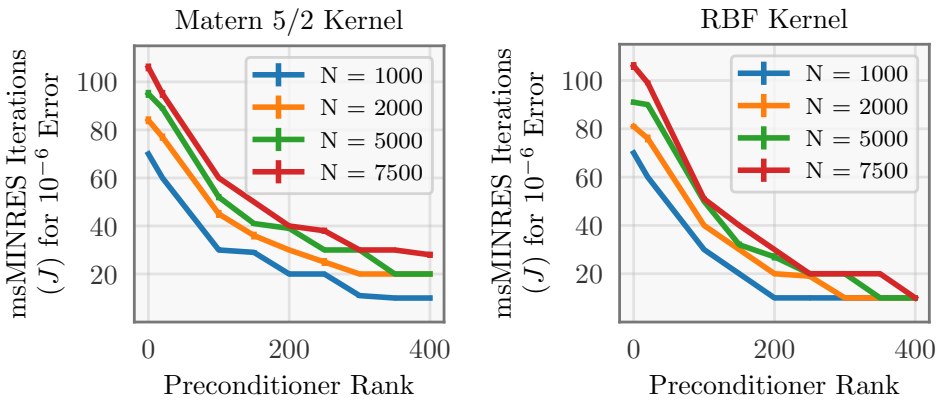

Figure 5.4: Effect of preconditioning msMINRES-CIQ (random Matérn and RBF kernels with a pivoted Cholesky preconditioner). Larger preconditioners reduce the number of msMINRES iterations required to reach 10^{-6} error.

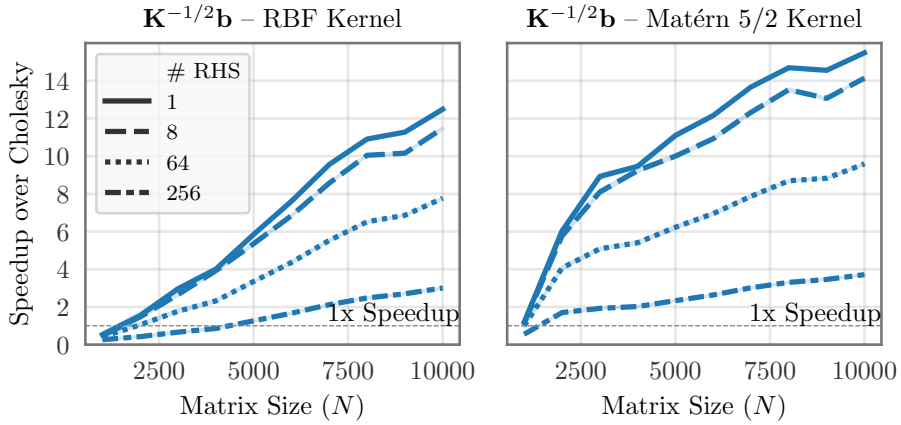

Figure 5.5: Speedup of msMINRES-CIQ over Cholesky when computing forward/backward passes of $\mathbf{K}^{-\frac{1}{2}}\mathbf{b}$ w/ varying number of right-hand-sides (RHS).

Speedup over Cholesky. We compare the wall-clock speedup of msMINRES-CIQ over Cholesky in Fig. 5.5 on RBF/Matérn kernels.[2] We compute $\mathbf{K}^{-1/2}\mathbf{b}$ and its derivative on multiple right-hand-side (RHS) vectors. As N increases, msMINRES-CIQ incurs a larger speedup (up to $15\times$ faster than Cholesky). This speedup is less pronounced when computing many RHSs simultaneously, as

[2] $Q = 8$. msMINRES is stopped after a residual of 10^{-4}. Kernels are formed using data from the Kin40k dataset [Asuncion and Newman, 2007]. Timings are performed on a NVIDIA 1070 GPU.

the cubic complexity of Cholesky is amortized across each RHS. Nevertheless, msMINRES-CIQ is advantageous for matrices larger than $N = 3{,}000$ even when simultaneously whitening 256 vectors.

5.4 Applications

In previous sections we showed, both theoretically and empirically, that msMINRES-CIQ accurately computes $\mathbf{K}^{\pm 1/2}\mathbf{b}$ while scaling better than traditional (Cholesky-based) methods. In this section we demonstrate applications of this increased speed and scalability. In particular, we show that using CIQ in conjunction with variational Gaussian processes and Bayesian optimization facilitates higher-fidelity models that can be applied to large-scale problems.

5.4.1 Whitened Stochastic Variational Gaussian Processes

As a first application, we demonstrate that the msMINRES-CIQ whitening procedure $\mathbf{K}^{-1/2}\mathbf{b}$ can increase the fidelity of **stochastic variational Gaussian processes (SVGP)** [Hensman et al., 2013, 2015b, Matthews, 2017]. Recall from Section 2.1.7 that these models are used for non-conjugate likelihoods (e.g. binary classification) or for large datasets that do not fit into memory. SVGP forms an approximate posterior $p(f(\mathbf{x}) \mid \mathbf{X}, \mathbf{y}) \approx q(f(\mathbf{x})) = \mathbb{E}_{q(\mathbf{u})}\left[p\left(f(\mathbf{x}) \mid \mathbf{u}\right)\right]$, where $\mathbf{u} \in \mathbb{R}^M$ are inducing function values (see [Hensman et al., 2015b, Matthews, 2017] for a detailed derivation). $q\left(\mathbf{u}\right)$ is a Gaussian variational distribution parameterized by mean $\mathbf{m} \in \mathbb{R}^M$ and covariance $\mathbf{S} \in \mathbb{R}^{M \times M}$. $\mathbf{m}$ and $\mathbf{S}$ (as well as the model's kernel/likelihood hyperparameters) are chosen to maximize the

variational ELBO:

$$\mathcal{L}_{\text{ELBO}}\big\{q(\mathbf{u}) = \mathcal{N}\left[\mathbf{m}, \mathbf{S}\right]\big\} = \sum_{i=1}^{N} \mathbb{E}_{q(f(\mathbf{x}^{(i)}))}\left[\log p(y^{(i)} \mid f(\mathbf{x}^{(i)}))\right] - \mathrm{KL}\left[\, q(\mathbf{u}) \| p(\mathbf{u}) \,\right].$$

Rather than directly learning $\mathbf{m}$ and $\mathbf{S}$, it is more common to learn the *whitened parameters* [Kuss and Rasmussen, 2005, Matthews, 2017]:

$$\mathbf{m}' = \mathbf{K}_{\mathbf{ZZ}}^{-1/2}\mathbf{m}, \quad \mathbf{S}' = \mathbf{K}_{\mathbf{ZZ}}^{-1/2}\mathbf{S}\mathbf{K}_{\mathbf{ZZ}}^{-1/2}.$$

Under these coordinates, the KL divergence term is $\frac{1}{2}(\mathbf{m}'^{\top}\mathbf{m}' + \mathrm{Tr}(\mathbf{S}') - \log|\mathbf{S}'| - M)$, which doesn't depend on $p(\mathbf{u})$ and therefore is relatively simple to optimize. The posterior distribution $q(f(\mathbf{x})) = \mathcal{N}[\mu^*_{\text{aprx}}(\mathbf{x}), \mathrm{Var}^*_{\text{aprx}}(\mathbf{x})]$ is given by

$$\begin{aligned} \mu^*_{\text{aprx}}(\mathbf{x}) &= \mathbf{k}_{\mathbf{Zx}}^{\top}\mathbf{K}_{\mathbf{ZZ}}^{-\frac{1}{2}}\mathbf{m}', \\ \mathrm{Var}^*_{\text{aprx}}(\mathbf{x}) &= k(\mathbf{x},\mathbf{x}) - \mathbf{k}_{\mathbf{Zx}}^{\top}\mathbf{K}_{\mathbf{ZZ}}^{-\frac{1}{2}}\left(\mathbf{I} - \mathbf{S}'\right)\mathbf{K}_{\mathbf{ZZ}}^{-\frac{1}{2}}\mathbf{k}_{\mathbf{Zx}}. \end{aligned} \tag{5.10}$$

Time and space complexity. During training, we repeatedly compute the ELBO and its derivative, which requires computing Eq. (5.10) and its derivative for a minibatch of data points. Optimization typically requires up to $10{,}000$ iterations of training [e.g. Salimbeni et al., 2018b]. We note that $\mathbf{K}_{\mathbf{ZZ}}^{-1/2}\mathbf{b}$ (and its derivative) is the most expensive numerical operation during each ELBO computation. If we use Cholesky to compute this operation, the time complexity of SVGP training is $\mathcal{O}(M^3)$. On the other hand, msMINRES-CIQ-based SVGP training is only $\mathcal{O}(JM^2)$, where J is the number of msMINRES iterations. Both methods require $\mathcal{O}(M^2)$ storage for the $\mathbf{m}'$ and $\mathbf{S}'$ parameters.

Natural gradient descent with msMINRES-CIQ. The size of the variational parameters $\mathbf{m}'$ and $\mathbf{S}'$ grows quadratically with M. This poses a challenging optimization problem for standard gradient descent methods. To adapt to the

large M regime, we rely on **natural gradient descent (NGD)** to optimize $\mathbf{m}'$ and $\mathbf{S}'$ [e.g. Hensman et al., 2012, Salimbeni et al., 2018b]. At a high level, these methods perform the updates $[\mathbf{m},\ \mathbf{S}] \leftarrow [\mathbf{m},\ \mathbf{S}] - \varphi \boldsymbol{\mathcal{F}}^{-1} \nabla \mathcal{L}_{\text{ELBO}}$, where φ is a step size, $\nabla \mathcal{L}_{\text{ELBO}}$ is the ELBO gradient, and $\boldsymbol{\mathcal{F}}$ is the Fisher information matrix of the variational parameters. Naïvely, each NGD step requires $\mathcal{O}(M^3)$ computations with $\mathbf{m}'$ and $\mathbf{S}'$, which would dominate the cost of CIQ-based SVGP. Fortunately, we can derive a natural gradient update that only relies on matrix solves with $\mathbf{S}'$, which take $\mathcal{O}(JM^2)$ time using preconditioned conjugate gradients. Therefore, using NGD incurs the same *quadratic* asymptotic complexity as msMINRES-CIQ. See Appendix C for the $\mathcal{O}(M^2)$ NGD update equations.

Experimental details. We compare msMINRES-CIQ-SVGP against Cholesky-SVGP on 3 large-scale datasets: a GIS dataset (**3dRoad**, $D = 2$), a monthly precipitation dataset (**Precipitation**, $D = 3$), and a tree cover dataset (**Covtype**, $D = 54$). Each task has between $N = 70{,}000$ and $N = 500{,}000$ training data points. For 3dRoad we use a Gaussian observation model. The Precipitation dataset has noisier observations; therefore we apply a Student-T observation model. Finally, we reduce the CovType dataset to a binary classification problem and apply a Bernoulli observation model. We train models with $10^3 \leq M \leq 10^4$. Each dataset is randomly split into 75% training, 10% validation, and 15% testing sets; $\mathbf{x}$ and y values are scaled to be zero mean and unit variance. All models use a constant mean and a Matérn 5/2 kernel, with lengthscales initialized to 0.01 and inducing points initialized by K-means clustering. Each model is trained for 20 epochs with a minibatch size of 256.[3] We alternate

[3]The batch size is 512 on the Covtype dataset due to its larger size.

between optimizing $\mathbf{m}'/\mathbf{S}'$ and the other parameters, using NGD for the former and Adam [Kingma and Ba, 2015] for the latter. Each optimizer uses an initial learning rate of 0.01[4], decayed by $10\times$ at epochs 1, 5, 10, and 15. For CIQ we use $Q = 15$ quadrature points. msMINRES terminates when the $\mathbf{c}_j$ vectors achieve a relative norm of 0.001 or after $J = 200$ iterations. Results are averaged over three trials.

The 3dRoad and CovType datasets are available from the UCI repository [Asuncion and Newman, 2007]. For 3dRoad, we only use the first two features—corresponding to latitude and longitude. For CovType, we reduce the 7-way classification problem to a binary problem (`Cover_Type` $\in \{2, 3\}$ versus `Cover_Type` $\in \{0, 1, 4, 5, 6\}$). The Precipitation dataset is available from the IRI/LDEO Climate Data Library.[5] This spatio-temporal dataset aims to predict the "WASP" index (Weighted Anomaly Standardized Precipitation) at various latitudes/longitudes. Each data point corresponds to the WASP index for a given year (between 2010 and 2019)—which is the average of monthly WASP indices. In total, there are 10 years and $10{,}127$ latitude/longitude coordinates, for a total dataset size of $101{,}270$.

Results. The two methods achieve very similar test-set negative log likelihood (Fig. 5.6) and RMSE (Fig. 5.7). We note that there are small differences in the optimization dynamics, which is to be expected since $\mathbf{K}_{\mathbf{ZZ}}^{-1/2}\mathbf{k}_{\mathbf{Zx}}$ can differ by an orthogonal transformation when computed with msMINRES-CIQ versus

[4]On the Precipitation dataset, the initial learning rate is 0.005 for NGD stability with the Student-T likelihood.

[5]`http://iridl.ldeo.columbia.edu/maproom/Global/Precipitation/WASP_Indices.html`

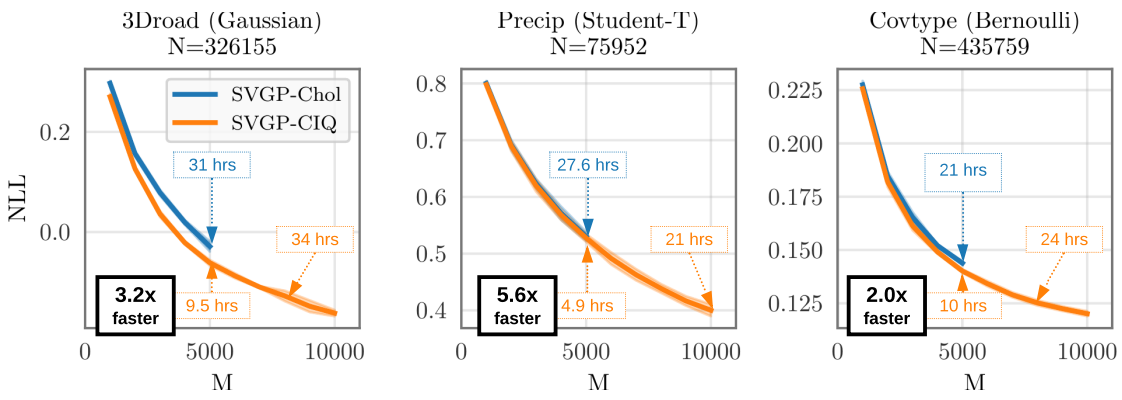

Figure 5.6: Negative log likelihood (NLL) of Cholesky versus msMINRES-CIQ SVGP models. **Left:** 3dRoad dataset ($N = 326155, D = 2$, Gaussian likelihood). **Middle:** Precipitation dataset ($N = 75952, D = 3$, Student-T likelihood). **Right:** CoverType dataset ($N = 435759, D = 54$, Bernoulli likelihood). NLL improves with more inducing points (M), and Cholesky and msMINRES-CIQ models have similar performance. However msMINRES-CIQ models train faster than their Cholesky counterparts.

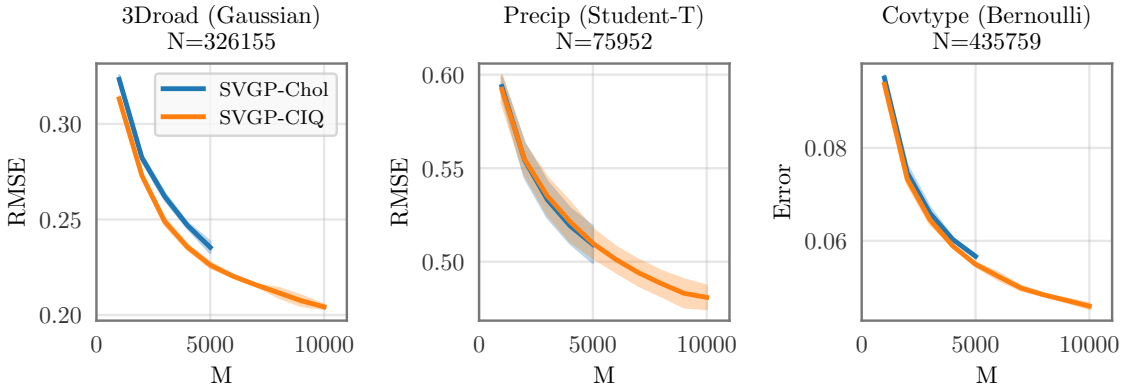

Figure 5.7: Error of Cholesky versus msMINRES-CIQ SVGP models. **Left:** 3dRoad dataset RMSE ($N = 326155, D = 2$, Gaussian likelihood). **Middle:** Precipitation dataset RMSE ($N = 75952, D = 3$, Student-T likelihood). **Right:** CoverType dataset $0/1$ error ($N = 435759, D = 54$, Bernoulli likelihood).

Cholesky. The key difference is the training time: with $M = 5{,}000$ inducing points, msMINRES-CIQ models are up to *5.6x faster* than Cholesky models (on a Titan RTX GPU). Moreover, msMINRES-CIQ models with $M = 8{,}000$-$10{,}000$ take roughly the same amount of time as $M = 5{,}000$ Cholesky models. Note we do not train $M > 5{,}000$ Cholesky models as doing so would require 2-10 days.

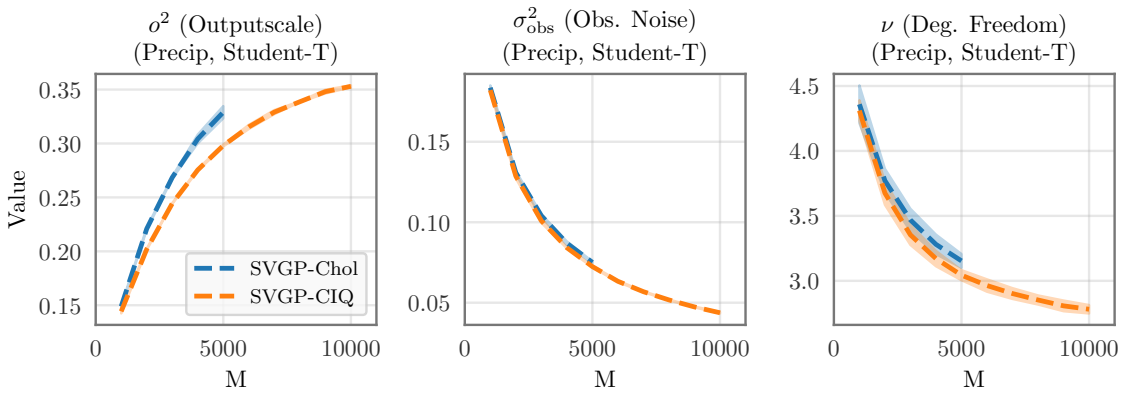

Figure 5.8: Hyperparameters as a function of inducing points (M) for Chol-SVGP and msMINRES-CIQ-SVGP (Precipitation dataset, Student-T likelihood). As M increases, the kernel outputscale (left) also increases. At the same time, the estimated observational noise (middle) decreases as does the estimated degrees of freedom (right), reflecting a heavier-tailed noise distribution. This suggests that, with larger M, SVGP models can find more signal in the data.

Effects of increased inducing points. We find that accuracy improves with increased M on all datasets. Scaling from $M = 5{,}000$ to $M = 10{,}000$ reduces test-set NLL by 0.1 nats on the 3dRoad and Precipitation datasets. We find similar reductions in predictive error. By scaling more readily to large M, msMINRES-CIQ enables high-fidelity variational approximations that would be computationally prohibitive with Cholesky. We also find that increasing M changes the values of the learned kernel/likelihood hyperparameters. In Fig. 5.8 we plot the learned hyperparameters of the Precipitation SVGP models: **(1)** o^2 (the kernel outputscale)—which roughly corresponds to variance explained as "signal" in the data; **(2)** σ^2_{obs}—which roughly corresponds to variance explained away as observational noise; and **(3)** ν (degrees of freedom)—which controls the tails of the noise model (lower ν corresponds to heavier tails). As M increases, we find that the observational noise parameter decreases by a factor of 4—down from 0.19 to 0.05—while the ν parameter also decreases. Models with larger M values can more closely approximate the true posterior [Hensman et al., 2013]; there-

fore, we expect that the larger-M likelihoods more closely correspond to the "true" parameters. This confirms findings from Bauer et al. [2016], who argue that variational approximations with small M tend to overestimate the amount of noise in datasets.

5.4.2 Posterior Sampling for Bayesian Optimization

The second application of msMINRES-CIQ we explore is Gaussian process posterior sampling in the context of Bayesian optimization (BO) [e.g. Snoek et al., 2012]. Many acquisition functions require drawing samples from posteriors [e.g. Frazier et al., 2009, Hernández-Lobato et al., 2014, Wang and Jegelka, 2017]. One canonical example is **Thompson Sampling** (TS) [Thompson, 1933, Hernández-Lobato et al., 2017, Kandasamy et al., 2018]. TS trades off exploitation of existing minima for exploration of new potential minima. TS chooses the next query point $\widetilde{\mathbf{x}}$ as the minimizer of a sample drawn from the posterior. Let $\mathbf{X}^* = [\mathbf{x}_1^*, \ldots, \mathbf{x}_T^*]$ be a *candidate set* of possible query points. To choose the next query point $\widetilde{\mathbf{x}}$, TS computes

$$\widetilde{\mathbf{x}} = \arg\min \left(\boldsymbol{\mu}^*(\mathbf{X}^*) + \mathbf{COV}^*(\mathbf{X}^*)^{\frac{1}{2}} \boldsymbol{\epsilon} \right), \quad \boldsymbol{\epsilon} \sim \mathcal{N}\left[\mathbf{0}, \mathbf{I}\right]. \tag{5.11}$$

where $\boldsymbol{\mu}^*(\mathbf{X}^*)$ and $\mathbf{COV}^*(\mathbf{X}^*)$ are the posterior mean and covariance of the Gaussian process at the candidate set. The candidate set is often chosen using a space-filling design, e.g. a Sobol sequence. The search space grows exponentially with the dimension; therefore, we need large values of T to more densely cover the search space for better optimization performance. Using Cholesky to compute Eq. (5.11) incurs a $\mathcal{O}(T^3)$ computational cost and $\mathcal{O}(T^2)$ memory, which severely limits the size of T. In comparison, msMINRES-CIQ only requires $\mathcal{O}(T^2)$ computation and $\mathcal{O}(T)$ memory.

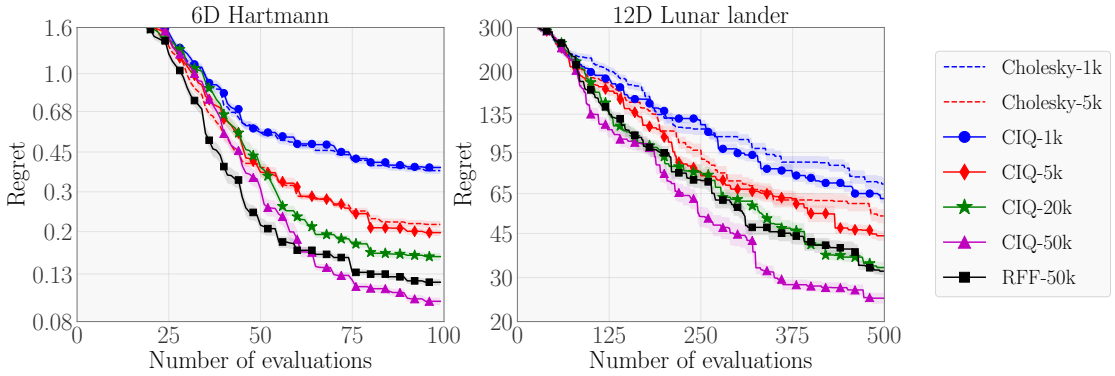

Figure 5.9: A comparison of sampling methods for Bayesian Optimization (BO). BO is applied to the (**left**) Hartmann ($D = 6$) and (**right**) Lunar Lander ($D = 12$) problems. Methods: Cholesky-$\langle T \rangle$ draws posterior samples with Cholesky at T candidate points. CIQ-$\langle T \rangle$ draws posterior samples with msMINRES-CIQ. RFF-50k uses random Fourier features to draw approximate posterior samples at $50{,}000$ candidate points. Larger T results in better optimization. msMINRES-CIQ enables scaling to $T \geq 50{,}000$. Each plot shows mean regret with standard error in log-scale based on 30 replications.

Experimental details. The 6-dimensional **Hartmann** function is a classical test problem in global optimization.[6] There are 6 local minima and a global minimum with value -3.32237. We use a total of 100 evaluations with 10 initial points. The 10 initial points are generated using a Latin hypercube design and we use a batch size of 5. In each iteration, we draw 5 samples and select 5 new trials to evaluate in parallel.

We consider the same setup and controller as in [Eriksson et al., 2019] for the 12-dimensional **Lunar Lander** problem.[7] The goal is to learn a controller that minimizes fuel consumption and distance to a given landing target while also preventing crashes. The state of the lunar lander is given by its angle, position, and time derivatives. Given this state

[6] `https://www.sfu.ca/~ssurjano/hart6.html`
[7] From the Open-AI gym: `https://gym.openai.com/envs/LunarLander-v2`

vector, the controller chooses one of the following four actions: $a \in$ {do nothing, booster left, booster right, booster down}. The objective is the average final reward over a fixed constant set of 50 randomly generated terrains, initial positions, and initial velocities. The optimal controller achieves an average reward of ≈ 309 over the 50 environments.

For both problems, we use a Matérn-$5/2$ kernel with ARD and a constant mean function. The domain is scaled to $[0,1]^d$ and we standardize the function values before fitting the Gaussian process. The kernel hyperparameters are optimized using L-BFGS-B and we use the following bounds: (lengthscale) $\ell \in [0.01, 2.0\,]$, (signal variance) $s^2 \in [0.05, 50.0]$, (noise variance) $\sigma^2 \in [1e-6, 1e-2]$. Additionally, we place a horseshoe prior on the noise variance as recommended in [Snoek et al., 2012]. We add $1\mathrm{e}{-4}$ to the diagonal of the kernel matrix to improve the conditioning and use a preconditioner of rank 200 for CIQ.

Baselines. We measure the performance of TS as a function of the candidate set size T and consider $T \in \{1{,}000, 5{,}000, 20{,}000, 50{,}000\}$. We run Cholesky (**Cholesky**-T) for $T \in \{1{,}000, 5{,}000\}$ and msMINRES-CIQ (**CIQ**-T). Note that it would be very challenging and impractical to use Cholesky with $T \geq 10{,}000$, due to its quadratic memory and cubic time complexity. In addition to Cholesky and CIQ, we also compare to random Fourier features (RFF) [Rahimi and Recht, 2008] with $1{,}000$ random features.

Optimization performance. We plot the mean regret with standard error based on 30 replications in Fig. 5.9. By increasing $T = 1{,}000$ to $T = 50{,}000$, the final regret achieved by CIQ is significantly lower on both problems. Large candidate sets have previously only been possible with approximate sampling

methods like RFF. We note, however, that RFF with $T = 50{,}000$ is outperformed by CIQ-50k on both problems.

5.5 Discussion

We have introduced msMINRES-CIQ—a MVM-based method for computing $\mathbf{K}^{1/2}\mathbf{b}$ and $\mathbf{K}^{-1/2}\mathbf{b}$. In sampling and whitening applications, msMINRES-CIQ can be used as a $\mathcal{O}(N^2)$ drop-in replacement for the $\mathcal{O}(N^3)$ Cholesky decomposition. Its scalability and GPU utilization enable us to use more inducing points with SVGP models and larger candidate sets in Bayesian optimization. In all applications, such increased fidelity results in better performance.

Advantages and disadvantages. One advantage of the Cholesky decomposition is its reusability. As discussed in Section 5.3, the cubic cost of computing $\mathbf{L}\mathbf{L}^\top$ is amortized when drawing $\mathcal{O}(M)$ samples or whitening $\mathcal{O}(M)$ vectors. Conversely, applying msMINRES-CIQ to $\mathcal{O}(M)$ vectors would incur a $\mathcal{O}(M^3)$ cost, eroding its computational benefits. Thus, our method is primarily advantageous in scenarios with a small number of right hand sides or where $\mathbf{K}$ is too large to apply Cholesky. We also emphasize that msMINRES-CIQ—like all Krylov methods—can take advantage of fast MVMs. Though this chapter focuses on applying this algorithm to dense matrices, we suggest that future work explore applications involving sparse or structured matrices.

Scaling beyond $M = 10{,}000$ and $T = 50{,}000$. It has been common to use only $M \approx 1{,}000$ inducing points with SVGP models. In this chapter, we have used

an order of magnitude more inducing points which results in demonstrably better predictive performance. As M continues to grow, the primary bottleneck of msMINRES-CIQ-SVGP becomes the quadratic memory costs of the variational parameters $\mathbf{m}'$ and $\mathbf{S}'$. While a scalable approximation of $\mathbf{K_{ZZ}}$ (with fast MVMs) can reduce the computational cost of whitening, the $\mathbf{S}'$ matrix in general does not afford space-saving structure. Efficient variational parameterizations will be necessary to scale to even larger M. This has been the topic of some recent work [Wilson et al., 2016b, Cheng and Boots, 2017, Salimbeni et al., 2018a, Shi et al., 2020].

Scaling msMINRES-CIQ Thompson sampling beyond $T = 50{,}000$ is to some extent more straightforward, as it does not require storing learnable parameters. Nevertheless, scaling T will naturally require more computation, which may result in the acquisition function becoming increasingly computationally demanding. The next section introduces a simple strategy to utilize multiple GPUs/distributed resources to alleviate this bottleneck.

CHAPTER 6

SCALING EXACT GAUSSIAN PROCESSES TO MILLIONS OF DATA POINTS

6.1 Introduction

In this final chapter, we combine the proposed methods from earlier chapters to scale exact GPs well beyond what has previously been achieved. In particular, we train a Gaussian process *on over a million data points* without the use of scalable approximations. Such a result would be intractable with standard GP implementations that rely on the Cholesky decomposition. On the other hand, BBMM and LOVE **(1)** effectively utilize GPU acceleration via matrix multiplication (Section 3.5); **(2)** achieve exponential convergence using a partial pivoted Cholesky preconditioner (Section 3.3); **(3)** require relatively few iterations to achieve convergence (Sections 3.5 and 4.5); and **(4)** more accurately solve linear systems than Cholesky-based approaches (Section 3.5).

While the past chapters address the computational efficiency of BBMM and LOVE, we must still address the memory bottleneck. Exact GPs, unlike their scalable counterparts, make use of dense $N \times N$ training kernel matrices which naïvely require quadratic memory. To overcome this limitation, we partition and distribute kernel MVMs across GPUs in a Map-Reduce style fashion. This reduces the memory requirement for GPs down to $\mathcal{O}(N)$, permitting scaling beyond $N \approx 10^4$. Our implementation uses the KeOps library [Charlier et al., 2020] and a custom multi-GPU wrapper to implement these memory-efficient MVMs.

In addition, we introduce a number of practical heuristics to accelerate training and maximally utilize parallelization. With a single GPU, exact GPs can be trained in seconds for $N \approx 10^4$, minutes for $N \approx 10^5$, and hours for $N \approx 10^6$. After training, exact GP models can *make predictions in milliseconds* using the LOVE method presented in Chapter 4.

Exact GPs vs scalable approximations. A natural question is whether exact Gaussian processes are desirable for such large datasets. Even with LOVE and BBMM, exact GPs require $\mathcal{O}(N^2)$ computation. Many approximate methods have been introduced to reduce this asymptotic complexity, relying on a mixture-of-experts [Deisenroth and Ng, 2015], inducing points [Snelson and Ghahramani, 2006, Titsias, 2009, Wilson and Nickisch, 2015, Gardner et al., 2018b], random feature expansions [Rahimi and Recht, 2008, Le et al., 2013, Yang et al., 2015], tensor decompositions [Izmailov et al., 2018a, Evans and Nair, 2018], or stochastic variational optimization [Hensman et al., 2013, 2015b, Wilson et al., 2016b, Cheng and Boots, 2017, Salimbeni et al., 2018a, Shi et al., 2020]. Recent analysis demonstrates good convergence rates for GP approximations under certain conditions [Burt et al., 2019]. Moreover, the BBMM, LOVE, and CIQ algorithms can be used with a wide variety of GP models with almost no additional implementation (see Section 3.4).

At the same time, there is reason to believe that exact methods are preferable over scalable GP approximations if they are computationally feasible. Every approximate method inherently makes biases and tradeoffs that may not work with every dataset [Turner and Sahani, 2011, Bauer et al., 2016]. Choosing an appropriate approximation is akin to a large hyperparameter search and may require expert knowledge. Exact GPs offer a simplicity that is more generally

applicable and—as we will demonstrate—often more accurate.

We benchmark on large regression datasets from the UCI repository [Asuncion and Newman, 2007]. We find exact GPs offer notably better performance than scalable approximations, often exceeding a two-fold reduction in root-mean-squared error. Exact GPs continue to benefit from the addition of new training points, a valuable conceptual finding in favor of non-parametric approaches. Moreover, our results clarify the relative performance of popular GP approximations against exact GPs in the $N \geq 100{,}000$ regime—a comparison which has previously been considered intractable.

6.2 Adapting BBMM and LOVE to Large-Scale Exact GPs

The BBMM and LOVE methods presented in Chapters 3 and 4 effectively utilize GPU acceleration and reduce the time complexity of GPs to $\mathcal{O}(N^2)$. As we will demonstrate, these advantages make it possible to scale exact GPs to very large datasets—up to two orders of magnitude larger than what is possible with Cholesky-based training/inference [Nguyen et al., 2019].

In this section, we make slight modifications to the BBMM and LOVE algorithms for large-scale GPs. Firstly, we reduce the MVM memory requirements to $\mathcal{O}(N)$ using partitioned and distributed kernel matrices. Secondly, we offer practical guidelines to speed up BBMM convergence for large datasets.

6.2.1 Reducing Memory Requiremnts to $\mathcal{O}(N)$

The primary input to the modified batched conjugate gradients (mBCG) algorithm of Section 3.2 is $\texttt{mvm\_}\widehat{\mathbf{K}}_{\mathbf{XX}}$, a black-box function that performs MVMs using the kernel matrix $\widehat{\mathbf{K}}_{\mathbf{XX}}$. Each iteration of mBCG updates four (batches) of vectors: $\mathbf{c}$ (the current solution), $\mathbf{r}$ (the current error), $\mathbf{d}$ (the "search" direction for the next solution), and $\mathbf{z}$ (a preconditioned error term)—see Algorithms 2.3 and 3.1 for details. Storing these vectors requires exactly $4N$ space. In addtion, the mBCG algorithm stores the tridiagonal matrices for stochastic Lanczos quadrature, which requires $\mathcal{O}(J)$ memory for J iterations of mBCG. Therefore, the quadratic space cost associated with MVM-based exact GPs comes from storing the matrix $\widehat{\mathbf{K}}_{\mathbf{XX}}$.

Typically, $\texttt{mvm\_}\widehat{\mathbf{K}}_{\mathbf{XX}}$ is implemented by first computing the full $N \times N$ kernel matrix $\widehat{\mathbf{K}}_{\mathbf{XX}}$. Although forming $\widehat{\mathbf{K}}_{\mathbf{XX}}$ requires $\mathcal{O}(N^2)$ memory, the *output* of the MVM $\widehat{\mathbf{K}}_{\mathbf{XX}}\mathbf{b}$ requires only $\mathcal{O}(N)$ memory. By using a *map-reduce style algorithm* for this computation, we can reduce the memory requirement of $\texttt{mvm\_}\widehat{\mathbf{K}}_{\mathbf{XX}}$.

Partitioned kernel MVMs. We first partition the data matrix $\mathbf{X} \in \mathbb{R}^{N \times D}$ into P partitions, each of which contains roughly N/P data points:

$$\mathbf{X} = \begin{bmatrix} \mathbf{X}^{(1)}; & \cdots & ; \mathbf{X}^{(P)} \end{bmatrix}$$

where we use ";" to denote row-wise concatenation. For each $\mathbf{X}^{(\ell)}$, we can compute $\widehat{\mathbf{K}}_{\mathbf{X}^{(\ell)}\mathbf{X}}$, which is a roughly $(N/P) \times N$ kernel matrix. We can now rewrite the training kernel matrix as a concatenation of the P partitions:

$$\widehat{\mathbf{K}}_{\mathbf{XX}} = \begin{bmatrix} \widehat{\mathbf{K}}_{\mathbf{X}^{(1)}\mathbf{X}}; & \cdots & ; \widehat{\mathbf{K}}_{\mathbf{X}^{(P)}\mathbf{X}} \end{bmatrix}.$$

Computing each partition requires access to the full training set $\mathbf{X}$, which we assume fits in memory. However, each partition $\widehat{\mathbf{K}}_{\mathbf{X}^{(\ell)}\mathbf{X}}$ contains only $1/P$ of the entries of the full kernel matrix. Rewriting the matrix-vector product $\widehat{\mathbf{K}}_{\mathbf{XX}}\mathbf{b}$ in terms of these partitions:

$$\widehat{\mathbf{K}}_{\mathbf{XX}}\mathbf{b} = \left[\widehat{\mathbf{K}}_{\mathbf{X}^{(1)}\mathbf{X}}\mathbf{b};\ \ \cdots\ \ ;\widehat{\mathbf{K}}_{\mathbf{X}^{(P)}\mathbf{X}}\mathbf{b}\right],$$

we see that this MVM can be computed in smaller components by separately computing each $\widehat{\mathbf{K}}_{\mathbf{X}^{(\ell)}\mathbf{X}}\mathbf{b}$ and concatenating the results. We discard each kernel partition $\widehat{\mathbf{K}}_{\mathbf{X}^{(\ell)}\mathbf{X}}$ once its MVM has been computed. This MVM only allocates new memory to temporarily store the $(N/P) \times N$ kernel matrix partition $\widehat{\mathbf{K}}_{\mathbf{X}^{(\ell)}\mathbf{X}}$. The other memory requirements (i.e. storing $\mathbf{X}$ and $\mathbf{b}$) are linear in N.

This algorithm allows us to reduce memory usage in exchange for additional yet easily parallelizable computations. If $P = 1$ then we have the naïve $\mathcal{O}(N^2)$ memory MVM procedure. As $P \to N$, mBCG will only require $\mathcal{O}(N)$ memory.

(After developing our own custom partitioned-MVM approach, we were made aware of the KeOps software package [Charlier et al., 2020] which uses low-level GPU code to speed up partitioned MVMs. With this library, partitioned MVMs can even be faster than standard MVMs.)

Distributed parallel MVMs. MVM-based training/inference can easily take advantage of multiple GPUs or distributed computational resources. Each MVM partition $\widehat{\mathbf{K}}_{\mathbf{X}^{(l)}\mathbf{X}}\mathbf{b}$ can be performed on a different device. Additionally, we note that distributed/parallel MVMs require only $O(N)$ communication between devices (i.e. the memory required to share $\mathbf{X}$ and the partial output vector $\widehat{\mathbf{K}}_{\mathbf{X}^{(l)}\mathbf{X}}\mathbf{b}$). In contrast, distributing the Cholesky decomposition requires $\mathcal{O}(N^2)$ communication [Nguyen et al., 2019].

6.2.2 Practical Considerations

Preconditioning. As in Chapter 3, we use the partial pivoted Cholesky preconditioner for training and computing predictive means. In Chapter 3 the preconditioner size is typically limited to $R = 5$; however, we find that preconditioners of size $R = 100$ provide a noticeable speed improvement for large GPs. Computing a rank R partial pivoted Cholesky preconditioner requires only R kernel matrix rows: a $\mathcal{O}(N)$ space dependence. While each partitioned MVM computes the kernel sub-matrices from scratch, the preconditioner is computed only once. Therefore, for large values of N it can be efficient to increase the size of the preconditioner to reduce the number of mBCG iterations.

mBCG convergence criteria. Importantly, mBCG is *not* an approximate method for performing linear solves. Rather, it is a method that performs solves to a specified tolerance. If this tolerance is sufficiently tight, solve are exact up to machine precision. Thus, it is analogous to using gradient descent for convex optimization problems (which is in fact what is happening).

At test time, we find that (nearly) exact solves $\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{y}$ are critical for good predictive means. Therefore, we set the convergence criterion of CG to be $\|\widehat{\mathbf{K}}_{\mathbf{XX}}\mathbf{c} - \mathbf{y}\|_2 / \|\mathbf{y}\|_2 \leq 0.001$, where $\mathbf{c}$ is the solution from mBCG. For hyper-parameter optimization, we find that less strict tolerances are surprisingly sufficient. A looser convergence criterion of up to $\|\widehat{\mathbf{K}}_{\mathbf{XX}}\mathbf{c} - \mathbf{y}\|_2 / \|\mathbf{y}\|_2 = 1$ has little impact on final model performance.

Pre-training. On large datasets, each training iteration (i.e. each call to mBCG) may take several minutes. We can reduce the total number of training iterations

by first initializing the hyperparameters to sensible defaults. As a simple initialization strategy, we pre-train exact Gaussian process models on a subset of training data (e.g. $N = 10{,}000$). After this initialization, only a few iterations (i.e. < 5 gradient descent steps) are necessary on the full dataset (see Fig. 6.2).

6.3 Results

We compare exact GPs against widely-used approximate methods on large-scale datasets from the UCI repository [Asuncion and Newman, 2007]. These results are the first-ever comparison of exact versus approximate GPs on $N \gg 10^5$. Our experiments demonstrate that exact GPs: **(1)** outperform popular approximate GPs methods on many benchmarking datasets; **(2)** compute thousands of test-point predictions in milliseconds, even when $N > 10^6$; **(3)** utilize all available data when making predictions; and **(4)** achieve linear training speedups when using multiple GPUs.

We compare exact GPs against two scalable GP approximations: Sparse Gaussian Process Regression (SGPR) [Titsias, 2009] and Stochastic Variational Gaussian Processes (SVGP) [Hensman et al., 2013]. These methods are widely popular and general applicable, enabling a comparison over a wide range of datasets. Unless otherwise stated, we use $M = 512$ for SGPR and $M = 1{,}024$ for SVGP, which are common values for these methods [Matthews et al., 2017].

Experiment details. Each dataset is randomly split into 64% training, 16% validation, and 20% testing sets. Data are scaled to be mean 0 and standard deviation 1 as measured by the training set. We use a constant prior mean and a

Matérn 3/2 kernel with a shared lengthscale for each dimension.

For exact GPs: we pre-train the model's hyperparameters using a subset of $10{,}000$ randomly selected training points. The sub-sampled model is optimized with 10 steps of L-BFGS [Liu and Nocedal, 1989] and 10 steps of Adam [Kingma and Ba, 2015] with step sizes of 0.1. After pre-training, we run 3 additional iterations of Adam on the full dataset. For SGPR models: we optimize hyperparameters with 100 iterations of Adam, learning rate of 0.1. For SVGP models: we jointly optimize the variational parameters and hyperparameters with Adam—using a learning rate of 0.01 and a minibatch size of $1{,}024$—for 100 epochs.

Exact GPs and SGPR are trained with BBMM, using a rank-100 partial pivoted-Cholesky preconditioner. During training, the mBCG convergence tolerance is set to $\|\widehat{\mathbf{K}}_{\mathbf{XX}}\mathbf{c} - \mathbf{y}\|_2 / \|\mathbf{y}\|_2 = 1$. At test time, the mBCG tolerance is set to 0.001. We use a rank-100 LOVE approximation of $\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}$ to compute predictive variances. On the HouseElectric dataset, the likelihood's observational noise is constrained to be ≥ 0.1 in order to regularize the poorly conditioned kernel matrix. We use the KeOps library [Charlier et al., 2020] in conjunction with our GPyTorch BBMM/LOVE implementations to perform partitioned kernel MVMs.

Accuracy. Table 6.1 displays the test set RMSEs and negative log likelihoods (NLLs) of exact GPs and their approximate counterparts. We find that exact GPs achieve lower error than approximate methods on nearly every dataset. Notably, on certain datasets like 3dRoad, exact GPs achieve a half or even a quarter of the error of approximate methods.

Moreover, we also see approximate GP performance is dataset dependent.

Table 6.1: Performance of exact GPs and scalable approximations on large UCI datasets (shared-lengthscale Matérn 3/2 kernels). All results are averaged over 3 trials; $\pm$ corresponds to 1 standard deviation. (We are unable to scale SGPR to HouseElectric due to its memory requirements when $M = 512$.) **Top:** test set root mean square error (RMSE). **Bottom:** test set negative log likelihood (NLL).

			RMSE		
Dataset	N	D	**Exact GP** (BBMM)	**SGPR** ($M=512$)	**SVGP** ($M=1{,}024$)
PoleTele	9,600	26	$\mathbf{0.151} \pm 0.012$	0.217 ± 0.002	0.215 ± 0.002
Elevators	10,623	18	$\mathbf{0.394} \pm 0.006$	0.437 ± 0.018	0.399 ± 0.009
Bike	11,122	17	$\mathbf{0.220} \pm 0.002$	0.362 ± 0.004	0.303 ± 0.004
Kin40K	25,600	8	$\mathbf{0.099} \pm 0.001$	0.273 ± 0.025	0.268 ± 0.022
Protein	29,267	9	$\mathbf{0.536} \pm 0.012$	0.656 ± 0.010	0.668 ± 0.005
KeggDirected	31,248	20	$\mathbf{0.086} \pm 0.005$	0.104 ± 0.003	0.096 ± 0.001
CTslice	34,240	385	0.262 ± 0.448	$\mathbf{0.218} \pm 0.011$	1.003 ± 0.005
KEGGU	40,708	27	$\mathbf{0.118} \pm 0.000$	0.130 ± 0.001	0.124 ± 0.002
3dRoad	278,319	3	$\mathbf{0.101} \pm 0.007$	0.661 ± 0.010	0.481 ± 0.002
Song	329,820	90	0.807 ± 0.024	$\mathbf{0.803} \pm 0.002$	0.998 ± 0.000
Buzz	373,280	77	$\mathbf{0.288} \pm 0.018$	0.300 ± 0.004	0.304 ± 0.012
HouseElectric	1,311,539	9	$\mathbf{0.055} \pm 0.000$	—	0.084 ± 0.005

			NLL		
Dataset	N	D	**Exact GP** (BBMM)	**SGPR** ($M=512$)	**SVGP** ($M=1{,}024$)
PoleTele	9,600	26	$-\mathbf{0.180} \pm 0.036$	-0.094 ± 0.008	-0.001 ± 0.008
Elevators	10,623	18	0.619 ± 0.054	0.580 ± 0.060	$\mathbf{0.519} \pm 0.022$
Bike	11,122	17	$\mathbf{0.119} \pm 0.044$	0.291 ± 0.032	0.272 ± 0.018
Kin40K	25,600	8	$-\mathbf{0.258} \pm 0.084$	0.087 ± 0.067	0.236 ± 0.077
Protein	29,267	9	1.018 ± 0.056	$\mathbf{0.970} \pm 0.010$	1.035 ± 0.006
KeggDirected	31,248	20	-0.199 ± 0.381	$-\mathbf{1.123} \pm 0.016$	-0.940 ± 0.020
CTslice	34,240	385	$-\mathbf{0.894} \pm 0.188$	-0.073 ± 0.097	1.422 ± 0.005
KEGGU	40,708	27	-0.419 ± 0.027	$-\mathbf{0.984} \pm 0.012$	-0.666 ± 0.007
3dRoad	278,319	3	0.909 ± 0.001	0.943 ± 0.002	$\mathbf{0.697} \pm 0.002$
Song	329,820	90	$\mathbf{1.206} \pm 0.024$	1.213 ± 0.003	1.417 ± 0.000
Buzz	373,280	77	0.267 ± 0.028	$\mathbf{0.106} \pm 0.008$	0.224 ± 0.050
HouseElectric	1,311,539	9	$-\mathbf{0.152} \pm 0.001$	—	-1.010 ± 0.039

Neither SVGP nor SGPR consistently outperforms the other. Interestingly, dataset size/dimensionality do not seem to influence their relative performance.

For example, though Protein and Kin40K are similar in size and have similar dimensionality, the approximate methods perform worse on Kin40K (relative to the RMSE of exact GPs) than they do on Protein.

Training time. Table 6.2 (top) displays the training times for exact and approximate GPs. With the pre-training strategy, exact GPs on datasets with $N \leq 50{,}000$ can usually be trained in seconds. Datasets with $N \geq 50{,}000$ can be trained in minutes, while datasets with $N \geq 300{,}000$ require hours. The exact amount of training time is dataset dependent, which likely depends on the conditioning of the training kernel matrices. Remarkably, exact GPs can often be trained in *less time than their approximate counterparts*, despite having a larger asymptotic complexity. Approximate GPs tend to require more optimization iterations to properly place inducing points. We also hypothesize that exact GPs benefit more from GPU acceleration, as dense kernel matrices afford more MVM parallelism than their low-rank approximations.

Prediction time. At test time, we find that the speed of exact GPs is comparable to approximate methods at test time. Table 6.2 (bottom) displays the time to compute $1{,}000$ predictive means and variances. After the LOVE precomputation, exact GPs make predictions in *milliseconds*.

Training acceleration with multiple GPUs. As discussed in Section 6.2, MVMs for training and predictions can be distributed across multiple devices. Fig. 6.1 plots the speedup as more GPUs are used for training on the KEGGU, 3dRoad, Song, and Buzz datasets. (Speedups are measured on NVIDIA Tesla V100-SXM2-32GB-LS GPUs without using the KeOps library.) Each of these

Table 6.2: Wall-clock time comparison of exact GPs versus approximate GPs on large UCI datasets. Models are trained and evaluated on a single NVIDIA GTX 2080-TI GPU. (Asterisks (*) indicate measurements made using 8 V100 GPUs without KeOps.) **Top:** training time for exact GPs and scalable approximations. **Bottom:** precomputation and prediction times for exact GPs. "Precomputation" refers to the LOVE cache computation. "Prediction" refers to predictive distribution computations for $1{,}000$ test points.

			Training		
Dataset	N	D	**Exact GP** (BBMM)	**SGPR** ($M=512$)	**SVGP** ($M=1{,}024$)
PoleTele	9,600	26	**41.5 s ± 1.1**	69.5 s ± 20.5	68.7 s ± 4.1
Elevators	10,623	18	**41.0 s ± 0.7**	69.7 s ± 22.5	76.5 s ± 5.5
Bike	11,122	17	**41.2 s ± 0.9**	70.0 s ± 22.9	77.1 s ± 5.6
Kin40K	25,600	8	**42.7 s ± 2.7**	97.3 s ± 57.9	195.4 s ± 14.0
Protein	29,267	9	**47.9 s ± 10**	136.5 s ± 53.8	198.3 s ± 15.9
KeggDirected	31,248	20	**51.0 s ± 6.3**	132.0 s ± 65.6	228.2 s ± 22.9
CTslice	34,240	385	3.32 min ± 5.0	**2.16 min ± 0.99**	3.86 min ± 0.34
KEGGU	40,708	27	**0.790 min ± 0.14**	2.22 min ± 1.0	4.78 min ± 0.40
3dRoad	278,319	3	15.8 min ± 7.4	**12.0 min ± 5.5**	34.0 min ± 3.1
Song	329,820	90	**4.22 min ± 3.7**	7.88 min ± 3.1	39.6 min ± 3.1
Buzz	373,280	77	1.19 hr ± 0.39	**0.486 hr ± 0.30**	0.772 hr ± 0.05
HouseElectric	1,311,539	9	**1.20 hr ± 0.04**	—	6.12 hr ± 0.08

			Precomputation	**Prediction**		
Dataset	N	D	**Exact GP** (BBMM)	**Exact GP** (BBMM)	**SGPR** ($M=512$)	**SVGP** ($M=1{,}024$)
PoleTele	9,600	26	5.14 s	**6 ms**	**6 ms**	273 ms
Elevators	10,623	18	0.95 s	**7 ms**	**7 ms**	212 ms
Bike	11,122	17	0.38 s	**7 ms**	9 ms	182 ms
Kin40K	25,600	8	12.3 s	**11 ms**	12 ms	220 ms
Protein	29,267	9	7.53 s	14 ms	**9 ms**	146 ms
KeggDirected	31,248	20	8.06 s	**15 ms**	16 ms	143 ms
CTslice	34,240	385	7.57 s	**22 ms**	14 ms	133 ms
KEGGU	40,708	27	18.9 s	**18 ms**	13 ms	211 ms
3dRoad	278,319	3	118 m*	119 ms	**68 ms**	130 ms
Song	329,820	90	22.2 m*	123 ms	**99 ms**	134 ms
Buzz	373,280	77	42.6 m*	131 ms	**114 ms**	142 ms
HouseElectric	1,311,539	9	3.40 hr*	958 ms	—	**166 ms**

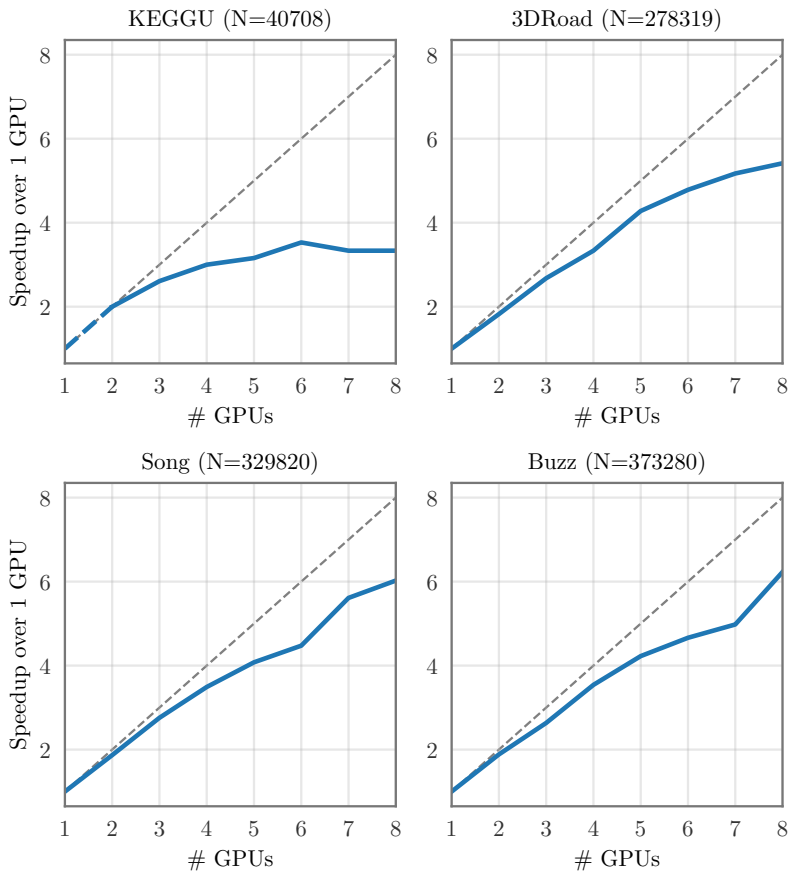

Figure 6.1: Speed of BBMM training using multi-GPU computation. On large datasets, exact GPs with BBMM achieve a near linear speedup with more GPUs. (Speedups are measured on NVIDIA Tesla V100-SXM2-32GB-LS GPUs.)

datasets achieve a nearly linear speedup up to 4 GPUs. The speedup is more pronounced for the two large datasets (3dRoad and Song).

Initialization. In Fig. 6.2 we compare GP models with and without our pre-training initialization scheme. The GPs with initialization are pre-trained on a $N = 10{,}000$ subset of the training data before running a final 3 iterations of Adam on the full dataset. The GPs without initialization are trained on the full dataset for 100 iterations of Adam. On all datasets, the pre-trained models achieve a comparable test set RMSE as the standard models. However, the pre-trained models require an *order of magnitude* less training time.

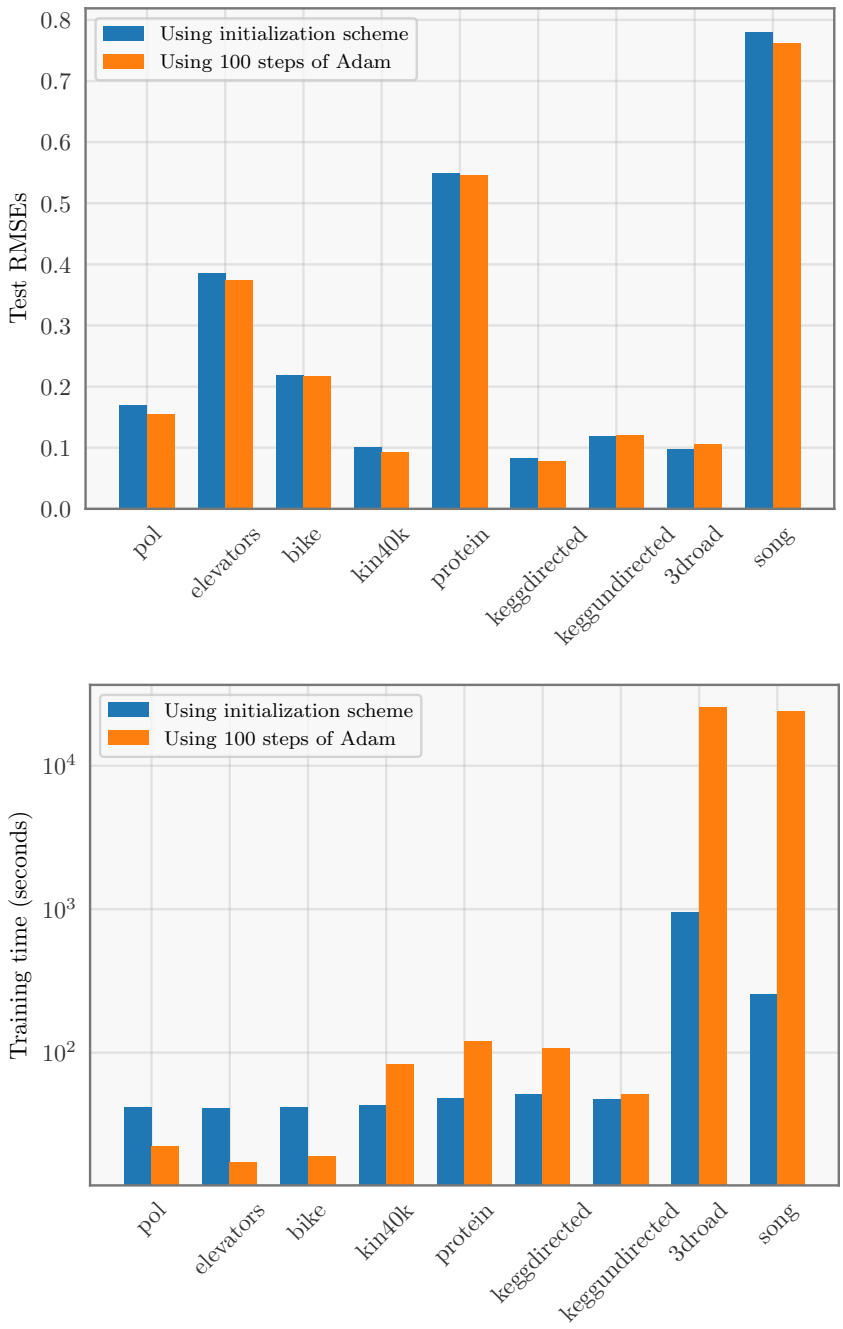

Figure 6.2: Effect of pre-training-based initialization on GP accuracy/timing.

6.4 Ablation Studies

With BBMM and LOVE, we can better understand how exact GPs scale to datasets with $N \gg 10^4$ compared to approximate GPs. Here, we demonstrate

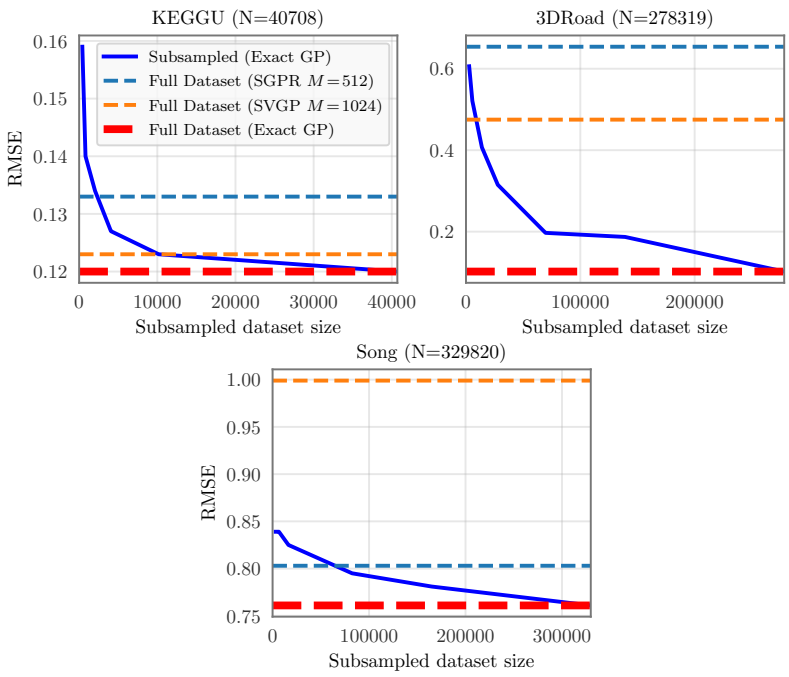

Figure 6.3: Effect of subsampling on exact GP performance, as measured by test set RMSE. Subsampled exact GPs outperform approximate GPs, even when only quarter of the training set is used. Exact GP error continues to decrease as data is added.

how the amount of data affects exact GP performance, and how the number of inducing points affects the performance of approximate GPs.

Do GPs need the entire dataset? As non-parametric models, Gaussian processes naturally adapt to the amount of data available. Fig. 6.3 shows an increase in accuracy as we increase the amount of training data on the KEGGU, 3dRoad, and Song datasets. For each dataset, we subsample a fraction of the data and plot the resulting test set RMSE as a function of training set size. As expected, the error decreases monotonically as we increase the subsample size. Fig. 6.3 also shows the performance of SGPR and SVGP models trained on the entire dataset. Strikingly, in all three cases, *an exact GP with less than a quarter of the training data outperforms approximate GPs trained on the entire dataset*.

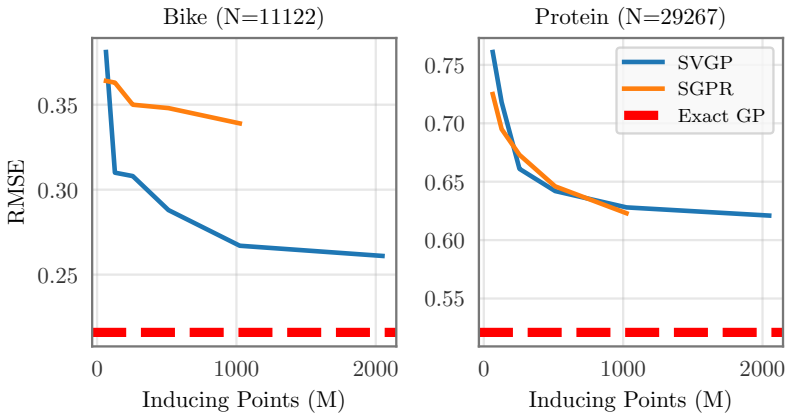

Figure 6.4: Error of SVGP and SGPR models as a function of inducing points (M). Both methods scale cubically with M. We are unable to run SGPR with more than $1{,}024$ inducing points on a single GPU. Exact GPs have lower error than both methods.

Would more inducing points help? The results in Table 6.1 naturally raise the question: "can approximate models with more inducing points recover the performance of exact methods?" In Fig. 6.4, we plot test set RMSE on two datasets, Bike and Protein, as a function of the number of inducing points. We note that in theory the performance of SVGP and SGPR should match exact GPs as $M \to N$ [Titsias, 2009, Hensman et al., 2013]. In practice however, the test RMSE of SGPR and SVGP remains well above exact GPs, and adding more inducing points has diminishing returns. We note that using M inducing points introduces a $M \times M$ matrix and a $\mathcal{O}(NM^2 + M^3)$ time complexity which makes it difficult to train SGPR with $M \gg 1024$ inducing points. It is possible to combine partitioned kernel MVMs with inducing-point methods to utilize even larger values of M. However, as Fig. 6.4 and Table 6.1 show, it may be preferable to use the extra computational resources to train an exact GP on more data rather than to train an approximate GP with more inducing points.

6.5 Discussion

Historically for Gaussian processes, "a large dataset is one that contains over a few thousand data points" [Hensman et al., 2013]. In this chapter, we have extended exact GPs far beyond what has previously been thought possible—applying GPs to datasets with over a million training examples. In our experiments, we find that exact GPs perform significantly better than approximate methods on large datasets, while requiring fewer design choices.

Is CG still exact? In the GP literature, *exact* GP training and inference typically refers to Cholesky-based inference with exact kernels [Rasmussen and Williams, 2006]. A natural question to ask is whether we can consider our approach "exact" in light of the fact that mBCG perform solves only up to a pre-specified error tolerance. However, unlike scalable GP approximations, the difference between a mBCG-based model and a theoretical model with "perfect" solves is precisely controlled by this error tolerance. We therefore consider mBCG "exact" in the context of mathematical optimization—namely that it computes solutions up to arbitrary numerical precision. In fact, mBCG-based methods can often be more precise than Cholesky based approaches in floating-point arithmetic due to fewer round-off errors, as demonstrated in Chapter 3 (Fig. 3.3).

When to approximate? There are many scalable GP approximations with varying statistical properties, advantages, and application regimes. We compare against the SVGP and SGPR methods due to their popularity and general applicability. There may be some regimes where other approximate methods outperform these two approaches. Our objective in this chapter is not to per-

form an exhaustive study of approximate methods, but rather to highlight that such comparisons are now possible.

Indeed, there are cases where approximate GP methods might still be preferable, especially when computational resources are limited. In certain regimes, such as low dimensional spaces, approximate methods like KISS-GP can achieve high degrees of accuracy in less time than exact GPs [Wilson and Nickisch, 2015]. Additionally, GP inference with non-conjugate likelihoods necessitates approximate Bayesian inference techniques (see Chapter 5).

Nevertheless, with efficient utilization of modern hardware, exact Gaussian processes are now an appealing option on substantially larger datasets than previously thought possible. We expect exact GPs to become even more scalable and accessible with continued advances in hardware design.

CHAPTER 7

CONCLUSION AND FUTURE DIRECTIONS

This thesis has presented a comprehensive framework for Gaussian process training, inference, and prediction. The algorithms proposed in Chapters 3 to 5 are based around a single central design decision: reduce all expensive matrix operations to parallelized matrix-vector multiplications (MVMs). Chapter 3 introduced the Black-Box Matrix × Matrix (BBMM) framework which computes GP training terms via a modified batched-conjugate gradients algorithm (mBCG). To enable fast predictions with GP models, Chapter 4 presented LanczOs Variance Estimates (LOVE), which computes an amortized cache of the predictive posterior. Finally, Chapter 5 introduced MVM-based Contour Integral Quadrature (msMINRES-CIQ) to "whiten" and "unwhiten" vectors with respect to a Gaussian covariance—enabling the MVM-based training of stochastic variational GP models and also allowing efficient posterior sampling.

The MVM theme simultaneously addresses several desiderata for Gaussian processes. As demonstrated in Chapter 3, MVM-based methods effectively utilize GPU hardware and reduce specialty implementations to ≤ 50 lines of code. This makes it easy for researchers to rapidly prototype and test novel models across a wide variety of datasets. MVM-based methods also lead to faster, more powerful models. Chapter 4 significantly reduces computational costs of GPs at test-time, while Chapter 5 scales up variational approximations and large-scale sampling, leading to better predictions and black-box optimization. Combined, these methods expand what is considered tractable for Gaussian processes, as demonstrated in the $N = 1{,}000{,}000$ exact GP experiments of Chapter 6.

7.1 Beyond Matrix-Vector Multiplication

The framework presented in this thesis makes GPs increasingly practical on massive datasets. Large-scale problems allow for more powerful classes of GP-based models, which in turn opens up many exciting research problems.

To increase the representational capacity of Gaussian processes, researchers have proposed highly-parametric kernels [Wilson and Adams, 2013, Wilson et al., 2016b], using GPs as components of larger pipelines [Schulam and Saria, 2015, Futoma et al., 2017], and hierarchical GP models [Wilson et al., 2012, Salimbeni and Deisenroth, 2017, Jankowiak et al., 2020a]. Of course, the additional complexity of these approaches may pose new training and inference challenges. Advances in large-scale optimization have mostly targeted the piecewise linear geometry of ReLU neural networks and may need to be adapted to the geometry of Gaussian process-based models. This is especially true for GP models that use alternative objective functions for learning [Sheth and Khardon, 2017, Knoblauch et al., 2019, Jankowiak et al., 2020b]. Moreover, hierarchical GP models are more computationally intensive than simpler models. Such models necessitate parametric approximations, as exact inference is intractable. Consequentially, increasing the fidelity of these models (e.g. stacking more layers, using more inducing points) increases the number of parameters, which may become an optimization or test-time bottleneck. It is worth noting that these problems are not unique to large-scale Gaussian processes—they are also issues of other large-scale machine learning models. Addressing these challenges in the context of GPs however is a relatively new area of research, as these models have only recently been considered practical.

7.2 Beyond Gaussian Processes

A key insight of this thesis is that non-linear operations on large-scale kernel matrices are surprisingly tractable when used in conjunction with GPU acceleration and efficient numerical techniques. While we motivate this finding through GPs, it is worth noting that the algorithms presented here are applicable to other classes of models. For example, a common relaxation to optimal transport problems is solved via Sinkhorn iterations [Cuturi, 2013], which rely on iterative MVMs with an exponentiated distance matrix. Second-order optimization is another application where large-scale solves are necessary. GPU-accelerated MVMs may make such methods applicable to higher dimensional problems [Koh and Liang, 2017].

More generally, machine learning in recent years has shied away from complex matrix operations. Many modern algorithms instead derive expressive power through the composition of linear and element-wise functions [Goodfellow et al., 2016]. While deep neural networks demonstrate the merit of this approach, it is possible that incorporating more complex matrix operations could improve parameter efficiency and model capacity [Jankowiak et al., 2020a]. The ability to efficiently compute arbitrary functions of big matrices opens up possibilities well beyond large-scale Gaussian process models.

APPENDIX A

CONVERGENCE ANALYSIS OF PRECONDITIONED MBCG

A.1 Proof of Theorems in Section 3.3.2

Here we include theorems and proofs about the partial pivoted Cholesky preconditioner applied to mBCG. All theorems are restated from Section 3.3.2.

A.1.1 Proof of Lemma 3.1

Lemma 3.1 (Restated). *Let $\bar{\mathbf{L}}_R$ be the rank-R pivoted Cholesky factor of kernel matrix $\mathbf{K_{XX}} \in \mathbb{R}^{N\times N}$. If the first R eigenvalues $\lambda_1, \ldots, \lambda_R$ of $\mathbf{K_{XX}}$ satisfy*

$$4^i \lambda_i \leq \mathcal{O}\left(e^{-Bi}\right), \quad i \in \{1, \ \ldots, \ R\}, \tag{3.8}$$

for some $B > 0$, then the condition number $\kappa(\widehat{\mathbf{P}}^{-1}\widehat{\mathbf{K}}_{\mathbf{XX}}) \triangleq \|\widehat{\mathbf{P}}_k^{-1}\widehat{\mathbf{K}}_{\mathbf{XX}}\|_2 \|\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\widehat{\mathbf{P}}_k\|_2$ satisfies the following bound:

$$\kappa\left(\widehat{\mathbf{P}}^{-1}\widehat{\mathbf{K}}_{\mathbf{XX}}\right) \leq \left(1 + \mathcal{O}\left(\sigma_{obs}^{-2} N e^{-BR}\right)\right)^2$$

where $\widehat{\mathbf{P}} = \left(\bar{\mathbf{L}}_R \bar{\mathbf{L}}_R^\top + \sigma_{obs}^2 \mathbf{I}\right)$ and $\widehat{\mathbf{K}}_{\mathbf{XX}} = (\mathbf{K_{XX}} + \sigma_{obs}^2 \mathbf{I})$.

Proof. Let $\mathbf{E}$ be the difference between $\mathbf{K_{XX}}$ and its rank-R pivoted Cholesky approximation $\bar{\mathbf{L}}_R \bar{\mathbf{L}}_R^\top$:

$$\mathbf{E} = \mathbf{K_{XX}} - \bar{\mathbf{L}}_R \bar{\mathbf{L}}_R^\top = \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \bar{\mathbf{S}}_{R+1} \end{bmatrix}$$

where $\bar{\mathbf{S}}_{R+1}$ is the Schur compliment that arises as the Cholesky error term—Eq. (2.20). The error $\mathbf{E}$ is therefore a positive semi definite matrix. By definition, the condition number $\kappa(\widehat{\mathbf{P}}^{-1}\widehat{\mathbf{K}}_{\mathbf{XX}})$ is given by

$$\kappa\left(\widehat{\mathbf{P}}^{-1}\widehat{\mathbf{K}}_{\mathbf{XX}}\right) \triangleq \left\|\widehat{\mathbf{P}}^{-1}\widehat{\mathbf{K}}_{\mathbf{XX}}\right\|_2 \left\|\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\widehat{\mathbf{P}}\right\|_2$$

The left term can be rewritten as:

$$\begin{aligned}
\left\|\widehat{\mathbf{P}}^{-1}\widehat{\mathbf{K}}_{\mathbf{XX}}\right\|_2 &= \left\|\left(\bar{\mathbf{L}}_R\bar{\mathbf{L}}_R^\top + \sigma_{\text{obs}}^2\mathbf{I}\right)^{-1}\left(\mathbf{K}_{\mathbf{XX}} + \sigma_{\text{obs}}^2\mathbf{I}\right)\right\|_2 \\
&= \left\|\left(\bar{\mathbf{L}}_R\bar{\mathbf{L}}_R^\top + \sigma_{\text{obs}}^2\mathbf{I}\right)^{-1}\left(\bar{\mathbf{L}}_R\bar{\mathbf{L}}_R^\top + \mathbf{E} + \sigma_{\text{obs}}^2\mathbf{I}\right)\right\|_2 \\
&= \left\|\mathbf{I} + \left(\bar{\mathbf{L}}_R\bar{\mathbf{L}}_R^\top + \sigma_{\text{obs}}^2\mathbf{I}\right)^{-1}\mathbf{E}\right\|_2
\end{aligned}$$

Similarly, the right term is:

$$\begin{aligned}
\left\|\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\widehat{\mathbf{P}}\right\|_2 &= \left\|\left(\bar{\mathbf{L}}_R\bar{\mathbf{L}}_R^\top + \sigma_{\text{obs}}^2\mathbf{I}\right)\left(\mathbf{K}_{\mathbf{XX}} + \sigma_{\text{obs}}^2\mathbf{I}\right)^{-1}\right\|_2 \\
&= \left\|\left(\mathbf{K}_{\mathbf{XX}} - \mathbf{E} + \sigma_{\text{obs}}^2\mathbf{I}\right)\left(\mathbf{K}_{\mathbf{XX}} + \sigma_{\text{obs}}^2\mathbf{I}\right)^{-1}\right\|_2 \\
&= \left\|\mathbf{I} - \left(\mathbf{K}_{\mathbf{XX}} + \sigma_{\text{obs}}^2\mathbf{I}\right)^{-1}\mathbf{E}\right\|_2
\end{aligned}$$

Since $\mathbf{K}_{\mathbf{XX}}$ and $\bar{\mathbf{L}}_R\bar{\mathbf{L}}_R^\top$ are both positive semi-definite, $(\mathbf{K}_{\mathbf{XX}} + \sigma_{\text{obs}}^2)$ and $(\bar{\mathbf{L}}_R\bar{\mathbf{L}}_R^\top + \sigma_{\text{obs}}^2)$ will both have a minimum eigenvalue $\lambda_{\min} \geq \sigma_{\text{obs}}^2$. Therefore,

$$\left\|\left(\mathbf{K}_{\mathbf{XX}} + \sigma_{\text{obs}}^2\right)^{-1}\right\|_2 \leq \sigma_{\text{obs}}^{-2}, \quad \left\|\left(\bar{\mathbf{L}}_R\bar{\mathbf{L}}_R^\top + \sigma_{\text{obs}}^2\right)^{-1}\right\|_2 \leq \sigma_{\text{obs}}^{-2}.$$

Applying these bound, along with Cauchy-Schwarz and the triangle inequality, gives us

$$\begin{aligned}
\kappa\left(\widehat{\mathbf{P}}^{-1}\widehat{\mathbf{K}}_{\mathbf{XX}}\right) &\leq \left(1 + \left\|\left(\bar{\mathbf{L}}_R\bar{\mathbf{L}}_R^\top + \sigma_{\text{obs}}^2\mathbf{I}\right)^{-1}\right\|_2 \|\mathbf{E}\|_2\right)\left(1 + \left\|\left(\mathbf{K}_{\mathbf{XX}} + \sigma_{\text{obs}}^2\mathbf{I}\right)^{-1}\right\|_2 \|\mathbf{E}\|_2\right) \\
&\leq \left(1 + \sigma_{\text{obs}}^{-2}\|\mathbf{E}\|_2\right)\left(1 + \sigma_{\text{obs}}^{-2}\|\mathbf{E}\|_2\right) \\
&= \left(1 + \sigma_{\text{obs}}^{-2}\|\mathbf{E}\|_2\right)^2. \qquad \text{(A.1)}
\end{aligned}$$

Since $\mathbf{E}$ is positive semi-definite, we have that $\|\mathbf{E}\|_2 \leq \mathrm{Tr}(\mathbf{E})$. The eigenvalue condition from Eq. (3.8) allows us to bound $\mathrm{Tr}(\mathbf{E})$ using Theorem 2.1:

$$\|\mathbf{E}\|_2 \leq \mathrm{Tr}\left(\mathbf{E}\right) = \mathrm{Tr}\left(\mathbf{K_{XX}} - \bar{\mathbf{L}}_R\bar{\mathbf{L}}_R^\top\right) \leq \mathcal{O}\big(Ne^{-BR}\big)\,. \tag{A.2}$$

Plugging Eq. (A.2) into Eq. (A.1) completes the proof. □

A.1.2 Proof of Theorem 3.1

Theorem 3.1 (Restated). *Let $\mathbf{K_{XX}} \in \mathbb{R}^{N\times N}$ be a $N \times N$ kernel that satisfies the eigenvalue condition of Eq.* (3.8)*, and let $\bar{\mathbf{L}}_R$ be its rank-R pivoted Cholesky factor. After J iterations of mBCG with preconditioner $\widehat{\mathbf{P}} = (\bar{\mathbf{L}}_R\bar{\mathbf{L}}_R^\top + \sigma_{obs}^2\mathbf{I})$, the difference between $\mathbf{c}_J$ and true solution $\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{y}$ is bounded by:*

$$\left\|\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{y} - \mathbf{c}_J\right\|_{\widehat{\mathbf{K}}_{\mathbf{XX}}} \leq \left[\frac{1}{1+\mathcal{O}(\sigma_{obs}^2 e^{RB}/N)}\right]^J \left\|\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{y}\right\|_{\widehat{\mathbf{K}}_{\mathbf{XX}}},$$

where $\widehat{\mathbf{K}}_{\mathbf{XX}} = (\mathbf{K_{XX}} + \sigma_{obs}^2\mathbf{I})$ and $B > 0$ is a constant.

Proof. Since Eq. (3.8) holds, we can simply plug Lemma 3.1 into the standard CG convergence bound (Theorem 2.2):

$$\begin{aligned}
\left\|\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{y} - \mathbf{c}_J\right\|_{\widehat{\mathbf{K}}_{\mathbf{XX}}} &\leq 2\left[\frac{\sqrt{\kappa\left(\widehat{\mathbf{P}}^{-1}\widehat{\mathbf{K}}_{\mathbf{XX}}\right)}-1}{\sqrt{\kappa\left(\widehat{\mathbf{P}}^{-1}\widehat{\mathbf{K}}_{\mathbf{XX}}\right)}+1}\right]^J \left\|\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{y}\right\|_{\widehat{\mathbf{K}}_{\mathbf{XX}}} \\
&\leq 2\left[\frac{1+\mathcal{O}(\sigma_{\text{obs}}^{-2}Ne^{-BR})-1}{1+\mathcal{O}(\sigma_{\text{obs}}^{-2}Ne^{-BR})+1}\right]^J \left\|\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{y}\right\|_{\widehat{\mathbf{K}}_{\mathbf{XX}}} \\
&= \left[\frac{1}{1+\mathcal{O}(\sigma_{\text{obs}}^2 e^{RB}/N)}\right]^J \left\|\widehat{\mathbf{K}}_{\mathbf{XX}}^{-1}\mathbf{y}\right\|_{\widehat{\mathbf{K}}_{\mathbf{XX}}}.
\end{aligned}$$

□

A.1.3 Proof of Theorem 3.2

Theorem 3.2 (Restated). *Assume* $\mathbf{K_{XX}} \in \mathbb{R}^{N \times N}$ *satisfies the eigenvalue condition of Eq.* (3.8)*. Suppose we estimate* $\Gamma \approx \log|\widehat{\mathbf{P}}^{-1}\widehat{\mathbf{K}}_{\mathbf{XX}}|$ *using Eq.* (3.5) *with:*

- $J \geq \mathcal{O}\left[(1+\sigma_{obs}^{-2}Ne^{-BR})\log\left((1+\sigma_{obs}^{-2}Ne^{-BR})/\epsilon\right)\right]$ *iterations of mBCG (for some constant* $B > 0$*), and*
- $T \geq \frac{32}{\epsilon^2}\log\left(\frac{2}{\delta}\right)$ *random* $\mathbf{z}^{(i)} \sim \mathcal{N}[\mathbf{0}, \widehat{\mathbf{P}}]$ *vectors.*

Then the error of the stochastic Lanczos quadrature estimate Γ *is probabilistically bounded by:*

$$Pr\left[\left|\log|\widehat{\mathbf{P}}^{-1}\widehat{\mathbf{K}}_{\mathbf{XX}}| - \Gamma\right| \leq \epsilon N\right] \geq (1-\delta)\,.$$

Proof. Since Eq. (3.8) holds, we can simply plug Lemma 3.1 into the stochastic Lanczos quadrature bound of Ubaru et al. [2017] (Theorem 2.3). □

A.2 Applying Theorems 3.1 and 3.2 to Univariate RBF Kernels

Our convergence theory depends on the assumption that the eigenvalues of $\mathbf{K_{XX}}$ decay exponentially. A natural question is when this assumption holds in practice. It happens that one can prove a concrete bound on the eigenvalue distribution of univariate RBF kernels:

Lemma A.1 (Lemma 3 of Gardner et al. [2018a])**.** *Given* $x^{(1)}, \ldots, x^{(N)} \in [0,1]$*, the univariate RBF kernel matrix*[1] $\mathbf{K_{XX}} \in \mathbb{R}^{N \times N}$ *with* $K^{(ij)} = \exp\left(-(x^{(i)} - x^{(j)})^2/\ell^2\right)$

[1] Here we drop the multiplicative outputscale parameter o^2 without loss of generality.

has eigenvalues $\lambda_1, \ldots, \lambda_k, \ldots, \lambda_N$ *bounded by:*

$$\lambda_{2k+1} \leq 2Ne^{-\ell^2/4} I_{k+1}(\gamma/4) \sim \frac{2Ne^{-\ell^2/4}}{\sqrt{\pi}\ell} \left(\frac{e\ell^2}{8(k+1)} \right)^{k+1},$$

where I_{k+1} *denotes the modified Bessel function of the first kind with parameter* $k+1$*.*

In other words, the eigenvalues of univariate RBF kernels decay *super-exponentially*, meeting the requirements of Eq. (3.8) in Lemma 3.1. Therefore, the bounds given by Theorems 3.1 and 3.2 apply.

A proof of Lemma A.1 was included in a version of Chapter 3 that was published at NeurIPS 2018 [Gardner et al., 2018a]. The proof itself is the work of my co-authors David Bindel and Jacob R. Gardner—therefore I choose not to claim credit for it as part of this thesis. It can be found in Appendix E of [Gardner et al., 2018a].

We would also note that, while many kernels do not meet the eigenvalue criterion of Lemma 3.1, most kernels have rapidly decaying eigenvalues and therefore achieve significantly faster convergence with the partial pivoted Cholesky preconditioner. This is demonstrated by the empirical results in Section 3.5 and Section 6.3.

APPENDIX B

DETAILS ON MSMINRES-CONTOUR INTEGRAL QUADRATURE

B.1 Selecting Quadrature Locations and Weights

Here we briefly describe the quadrature formula derived by Hale et al. [2008] for use with Cauchy's integral formula. We refer the reader to the original publication for more details.

Assume that $\mathbf{K}$ is a positive definite matrix, and thus has real positive eigenvalues. Our goal is to approximate Cauchy's integral formula with a quadrature estimate:

$$f(\mathbf{K}) = \frac{1}{2\pi i} \oint_{\Gamma} f(\tau) \left(\tau \mathbf{I} - \mathbf{K}\right)^{-1} \, \mathrm{d}\tau \tag{B.1}$$

$$\approx \frac{1}{2\pi i} \sum_{q=1}^{Q} \widetilde{w}_q f(\tau_q) \left(\tau_q \mathbf{I} - \mathbf{K}\right)^{-1}, \tag{B.2}$$

where $f(\cdot)$ is analytic on and within Γ, and $\widetilde{w}_q$ and τ_q are quadrature weights and nodes respectively. Note that Eq. (B.1) holds true for any closed contour Γ in the complex plane that winds once (counterclockwise) around the spectrum of $\mathbf{K}$.

A naïve approach with uniformly-spaced quadrature. For now, assume that $\lambda_{\min}$ and $\lambda_{\max}$—the minimum and maximum eigenvalues of $\mathbf{K}$—are known. (We will later address how they can be efficiently estimated.) A naïve first approach to Eq. (B.2) is to uniformly place the quadrature locations in a circle that surrounds the eigenvalues and avoids crossing the negative real axis, where we

anticipate f may be singular:

$$\tau_q = \frac{\lambda_{\max} + \lambda_{\min}}{2} + \frac{\lambda_{\max}}{2} e^{2i\pi(q/Q)}, \quad \widetilde{w}_q = \frac{1}{Q}, \quad q = 0, 1, \ldots, Q-1.$$

This corresponds to a standard trapezoid quadrature rule. However, Hale et al. [2008] demonstrate that the convergence of this quadrature rule depends linearly on the condition number $\kappa(\mathbf{K}) = \lambda_{\max}/\lambda_{\min}$. In particular, this is because the integrand is only analytic in a narrow region around the chosen contour. As many kernel matrices tend to be approximately low-rank and therefore ill-conditioned, this simple quadrature rule requires large Q to achieve the desired numerical accuracy.

Improving convergence with conformal mappings. Rather than uniformly spacing the quadrature points, it makes more sense to place more quadrature points near $\lambda_{\min}$ and fewer near $\lambda_{\max}$. This can be accomplished by using the above trapezoid quadrature rule in a transformed parameter space that is "stretched" near $\lambda_{\min}$ and contracted near $\lambda_{\max}$. Mathematically, this is accomplished by applying a conformal mapping that moves the singularities to the upper and lower boundaries of a periodic rectangle. We may then apply the trapezoid rule along a contour traversing the middle of the rectangle—maximizing the region in which the function we are integrating is analytic around the contour.

B.1.1 A Specific Quadrature Formula for $f(\mathbf{K}) = \mathbf{K}^{-1/2}$

Hale et al. [2008] suggest performing a change of variables that projects Eq. (B.1) onto an annulus. Uniformly spaced quadrature points inside the annulus will

cluster near $\lambda_{\min}$ when projected back into the complex plane. This change of variables has a simple analytic formula involving Jacobi elliptic functions (see [Hale et al., 2008, Sec. 2] for details). In the special case of $f(\mathbf{K}) = \mathbf{K}^{-1/2}$, we can utilize an additional change of variables for an even more efficient quadrature formulation [Hale et al., 2008, Sec. 4]. Setting $\sigma = \tau^{1/2}$, we have

$$\begin{aligned}\mathbf{K}^{-\frac{1}{2}} &= \frac{1}{\pi i} \oint_{\Gamma_s} \left(\sigma^2 \mathbf{I} - \mathbf{K}\right)^{-1} \mathrm{d}\sigma. \\ &\approx \frac{1}{\pi i} \sum_{q=1}^{Q} \widetilde{w}_q \left(\sigma_q^2 \mathbf{I} - \mathbf{K}\right)^{-1}, \end{aligned} \tag{B.3}$$

where Γ_σ is a contour that surrounds the spectrum of $\mathbf{K}^{1/2}$. Since the integrand is symmetric with respect to the real axis, we only need to consider the imaginary portion of Γ_σ. Consequently, all the τ_q quadrature locations (back in the original space) will be real-valued and negative. Combining this square-root change-of-variables with the annulus change-of-variables results in the following quadrature weights/locations:

$$\begin{aligned}\sigma_q^2 &= \lambda_{\min}\Big(\mathrm{sn}(iu_q\mathcal{K}'(k) \mid k)\Big)^2, \\ \widetilde{w}_q &= -\frac{2\sqrt{\lambda_{\min}}}{\pi Q} \left[\mathcal{K}'(k) \;\; \mathrm{cn}\left(iu_q\mathcal{K}'(k) \mid k\right) \;\; \mathrm{dn}\left(iu_q\mathcal{K}'(k) \mid k\right)\right], \end{aligned} \tag{B.4}$$

where we adopt the following notation:

- $k = \sqrt{\lambda_{\min}/\lambda_{\max}} = 1/\sqrt{\kappa(\mathbf{K})}$;
- $\mathcal{K}'(k)$ is the complete elliptic integral of the first kind with respect to the complimentary elliptic modulus $k' = \sqrt{1-k^2}$;
- $u_q = \frac{1}{Q}(q - \frac{1}{2})$; and
- $\mathrm{sn}(\cdot \mid k)$, $\mathrm{cn}(\cdot \mid k)$, and $\mathrm{dn}(\cdot \mid k)$ are the Jacobi elliptic functions with respect to elliptic modulus k.

The weights $\widetilde{w}_q$ and locations σ_q^2 from Eq. (B.4) happen to be real-valued and negative. Setting $t_q = -\sigma_q^2$ and $w_q = -\widetilde{w}_q$ gives us:

$$\mathbf{K}^{-\frac{1}{2}} \approx \sum_{q=1}^{Q} w_q \left(t_q \mathbf{I} + \mathbf{K}\right)^{-1}, \quad w_q = -\widetilde{w}_q > 0, \quad t_q = -\sigma_q^2 > 0. \tag{B.5}$$

An immediate consequence of this is that the shifted matrices $(t_q \mathbf{I} + \mathbf{K})$ are all positive definite.

Convergence of the quadrature approximation. Due to the double change-of-variables, the convergence of this quadrature rule in Eq. (B.4) is extremely rapid—even for ill-conditioned matrices. Hale et al. prove the following error bound:

Lemma B.1 (Hale et al. [2008], Thm. 4.1)**.** *Let $t_1, \ldots, t_Q > 0$ and $w_1, \ldots, w_Q > 0$ be the locations and weights of Hale et al.'s quadrature procedure. The error of Eq.* (5.3) *is bounded by:*

$$\left\| \mathbf{K} \sum_{q=1}^{Q} w_q \left(t_q \mathbf{I} + \mathbf{K}\right)^{-1} - \mathbf{K}^{\frac{1}{2}} \right\|_2 \leq \mathcal{O}\left(\exp\left(-\frac{2Q\pi^2}{\log \kappa(\mathbf{K}) + 3} \right) \right),$$

where $\kappa(\mathbf{K}) = \lambda_{max}/\lambda_{min}$ is the condition number of $\mathbf{K}$.

Remarkably, the error of Eq. (5.3) is *logarithmically* dependent on the conditioning of $\mathbf{K}$. Consequently, $Q \approx 8$ quadrature points is even sufficient for ill-conditioned matrices (e.g. $\kappa(\mathbf{K}) \approx 10^4$).

B.1.2 Estimating the Minimum and Maximum Eigenvalues

The equations for the quadrature weights/locations depend on the extreme eigenvalues λ_{max} and λ_{min} of $\mathbf{K}$. Using the Lanczos algorithm [Lanczos, 1950],

we can obtain accurate estimates of these extreme eigenvalues using relatively few matrix-vector multiplies with $\mathbf{K}$. To estimate λ_{min} and λ_{max} from Lanczos, we perform an eigendecomposition of $\mathbf{T}_J$ (the tridiagonal Lanczos matrix after J iterations—see Section 2.3.2). If J is small (i.e. $J \approx 10$) then this eigendecomposition requires minimal computational resources. In fact, as $\mathbf{T}_J$ is tridiagonal invoking standard routines allows computation of all the eigenvalues in $\mathcal{O}(J^2)$ time. A well-known convergence result of the Lanczos algorithm is that the extreme eigenvalues of $\mathbf{T}_J$ tend to converge rapidly to λ_{min} and λ_{max} [e.g. Saad, 2003, Golub and Van Loan, 2012]. Since the Lanczos algorithm always produces underestimates of the largest eigenavlue and overestimates of the smallest it is reasonable to use slightly larger and smaller values in the construction of the quadrature scheme—as we see in Lemma B.1, the necessary number of quadrature nodes is insensitive to small overestimates of the condition number.

B.1.3 The Complete Quadrature Algorithm

Algorithm B.1 obtains the quadrature weights w_q and locations t_q corresponding to Eqs. (B.4) and (B.5). Computing these weights requires ≈ 10 matrix-vector multiplies with $\mathbf{K}$—corresponding to the Lanczos iterations—for a total time complexity of $\mathcal{O}(N)$. All computations involving elliptic integrals can be readily computed using routines available in e.g. the SciPy library.

Algorithm B.1: Computing w_q and t_q for Contour Integral Quadrature.

Input : `mvm_K(·)` – function for matrix-vector multiplication (MVM) with matrix $\mathbf{K}$
Q – number of quad. points

Output: $w_1, \ldots, w_Q, t_1, \ldots, t_Q$

```
// Estimate extreme eigenvalues with Lanczos.
```
$\_, \mathbf{T} \leftarrow$ `lanczos( mvm_K(·) )` `// Lanczos w/ rand. init. vector`
$\lambda_{\text{min}}, \cdots, \lambda_{\text{max}} \leftarrow$ `symeig(`$\mathbf{T}$`)`

```
// Compute elliptic integral of the first kind.
// We use the relation K'(k) = K(k'), where k' = sqrt(1 - k^2) is the
   complementary elliptic modulus.
```
$k^2 \leftarrow \lambda_{\text{min}} / \lambda_{\text{max}}$ `// The squared elliptic modulus.`
$k'^2 \leftarrow \sqrt{1 - k^2}$ `// The squared complementary elliptic modulus.`
$\mathtt{K}' \leftarrow$ `ellipke(`k'^2`)` `//` $\mathtt{K}' = \mathcal{K}'(k)$

```
// Compute each quadrature weight/location.
```
for $q \leftarrow 1$ **to** Q **do**
  $u_q \leftarrow (q - 1/2)/Q$
  `// Compute Jacobi elliptic fn's via Jacobi's imaginary transform.`
  `// First we compute` $\overline{\mathtt{sn}}_q = \mathtt{sn}(u_q \mathcal{K}'(k) | k')$, $\overline{\mathtt{cn}}_q = \mathtt{cn}(u_q \mathcal{K}'(k) | k')$, $\overline{\mathtt{dn}}_q = \mathtt{dn}(u_q \mathcal{K}'(k) | k')$.
  $\overline{\mathtt{sn}}_q, \overline{\mathtt{cn}}_q, \overline{\mathtt{dn}}_q \leftarrow$ `ellipj(`$u_q \mathtt{K}', k'^2$`)`
  `// Use identities to convert` $\overline{\mathtt{sn}}_q$, $\overline{\mathtt{cn}}_q$, $\overline{\mathtt{dn}}_q$ `values into`
  `//` $\mathtt{sn}_q = \mathtt{sn}(i u_q \mathcal{K}'(k) | k)$, $\mathtt{cn}_q = \mathtt{cn}(i u_q \mathcal{K}'(k) | k)$, $\mathtt{dn}_q = \mathtt{dn}(i u_q \mathcal{K}'(k) | k)$.
  $\mathtt{sn}_q \leftarrow i \left[\overline{\mathtt{sn}}_q / \overline{\mathtt{cn}}_q \right]$
  $\mathtt{dn}_q \leftarrow \left[\overline{\mathtt{dn}}_q / \overline{\mathtt{cn}}_q \right]$
  $\mathtt{cn}_q \leftarrow \left[1 / \overline{\mathtt{cn}}_q \right]$
  `// Quadrature weight` w_q `and location` t_q
  $w_q \leftarrow (-2\lambda_{\text{min}}^{1/2}) / (\pi Q)\, \mathtt{K}'\, \mathtt{cn}_q\, \mathtt{dn}_q$
  $t_q \leftarrow \lambda_{\text{min}} (\mathtt{sn}_q)^2$
end
return $w_1, \ldots, w_Q, t_1, \ldots, t_Q$

B.2 Proof of Theorem 5.1

To prove the convergence result in Theorem 5.1, we first prove the following lemmas.

Lemma B.2. *Let* $\mathbf{K} \succ 0$ *be symmetric positive definite and let shifts* $t_1, \ldots, t_Q > 0$ *be*

real-valued and positive. After J iterations of msMINRES, all shifted solve residuals are bounded by:

$$\left\|(\mathbf{K}+t_q\mathbf{I})\mathbf{c}_J^{(q)}-\mathbf{b}\right\|_2 \leq \left(\frac{\sqrt{\kappa(\mathbf{K}+t_q\mathbf{I})}-1}{a\sqrt{\kappa(\mathbf{K}+t_q\mathbf{I})}+1}\right)^J \|\mathbf{b}\|_2 \leq \left(\frac{\sqrt{\kappa(\mathbf{K})}-1}{\sqrt{\kappa(\mathbf{K})}+1}\right)^J \|\mathbf{b}\|_2,$$

where $\mathbf{b}$ is the vector to solve against, $\mathbf{c}_J^{(1)}$, ..., $\mathbf{c}^{(Q)}$ are the msMINRES outputs, and $\kappa(\mathbf{K})$ is the condition number of $\mathbf{K}$.

Proof. The convergence proof uses a polynomial bound, which is the standard approach for Krylov algorithms. See [e.g. Shewchuk, 1994, Trefethen and Bau III, 1997, Saad, 2003] for an analogous proof for the conjugate gradients method and [e.g. Greenbaum, 1997] for a treatment of MINRES applied to both positive definite and indefinite systems.

At iteration J, the msMINRES algorithm produces:

$$\mathbf{c}_J^{(q)} = \underset{\mathbf{c}^{(q)}\in\mathcal{K}_J(\mathbf{K},\mathbf{b})}{\arg\min}\left[\left\|(\mathbf{K}+t_q\mathbf{I})\mathbf{c}^{(q)}-\mathbf{b}\right\|_2\right], \quad q=1,\dots Q, \tag{B.6}$$

where without loss of generality we assume $\mathbf{c}_0^{(q)} = \mathbf{0}$ for simplicity. Using the fact that Krylov subspaces are shift invariant, we immediately have that

$$\mathbf{c}_J^{(q)} = \underset{\mathbf{c}^{(q)}\in\mathcal{K}_J(\mathbf{K}+t_q\mathbf{I},\mathbf{b})}{\arg\min}\left[\left\|(\mathbf{K}+t_q\mathbf{I})\mathbf{c}^{(q)}-\mathbf{b}\right\|_2\right], \quad q=1,\dots Q. \tag{B.7}$$

Since $(\mathbf{K}+t_q\mathbf{I}) \succ 0$ we may invoke a result on MINRES error bounds for symmetric positive definite matrices [Greenbaum, 1997, Chapter 3] to conclude that

$$\left\|(\mathbf{K}+t_q\mathbf{I})\mathbf{c}_J^{(q)}-\mathbf{b}\right\|_2 \leq \left(\frac{\sqrt{\kappa(\mathbf{K}+t_q\mathbf{I})}-1}{\sqrt{\kappa(\mathbf{K}+t_q\mathbf{I})}+1}\right)^J \|\mathbf{b}\|_2.$$

Observing that $\kappa(\mathbf{K}+t_q\mathbf{I}) \geq \kappa(\mathbf{K})$ for all q since $t_q > 0$ concludes the proof. □

Lemma B.2 is a very loose bound, as it doesn't assume anything about the spectrum of $\mathbf{K}$ (which is standard for generic Krylov method error bounds) and up-

per bounds the residual error for every shift using the most ill-conditioned system. In practice, we find that msMINRES converges for many covariance matrices with $J \approx 100$, even when the conditioning is on the order of $\kappa(\mathbf{K}) \approx 10^4$. This convergence can be further improved with preconditioning.

Lemma B.3. *For any positive definite* $\mathbf{K}$ *and positive* t*, we have*

$$\frac{\sqrt{\kappa(\mathbf{K}+t\mathbf{I})}-1}{\sqrt{\kappa(\mathbf{K}+t\mathbf{I})}+1} = \frac{\sqrt{\lambda_{max}+t}-\sqrt{\lambda_{min}+t}}{\sqrt{\lambda_{max}+t}+\sqrt{\lambda_{min}+t}} < \frac{\lambda_{max}}{4t}. \tag{B.8}$$

Proof. We can upper bound the numerator:

$$\begin{aligned}\sqrt{\lambda_{\text{max}}+t}-\sqrt{\lambda_{\text{min}}+t} &\leq \sqrt{\lambda_{\text{max}}+t}-\sqrt{t} \\ &= \sqrt{\lambda_{\text{max}}}\left(\sqrt{1+t/\lambda_{\text{max}}}-\sqrt{t/\lambda_{\text{max}}}\right) \\ &\leq \sqrt{\lambda_{\text{max}}}\frac{1}{2\sqrt{t/\lambda_{\text{max}}}} = \frac{\lambda_{\text{max}}}{2\sqrt{t}},\end{aligned}$$

where we have applied the standard inequality $\sqrt{(\cdot)+1}-\sqrt{(\cdot)} < \frac{1}{2\sqrt{(\cdot)}}$. The denominator can be (loosely) lower-bounded as $2\sqrt{t}$. Combining these two bounds completes the proof. □

Lemma B.4. *Let* σ_q^2 *and* $\widetilde{w}_q$ *be defined as in Eq.* (B.4)*. Then*

$$\sum_{q=1}^{Q}\frac{|w_q|}{|t_q|} = \sum_{q=1}^{Q}\frac{|\widetilde{w}_q|}{|\sigma_q^2|} < \frac{4Q\log\left(5\sqrt{\kappa(\mathbf{K})}\right)}{\pi\sqrt{\lambda_{min}}}$$

where $w_q = -\widetilde{w}_q$ *and* $t_q = -\sigma_q^2$ *as used in Eq.* (B.5)*.*

Proof. Using facts about elliptical integrals we have

$$\mathcal{K}'(k) < \log(1+4/k) \leq \log(5/k) \qquad k \in (0,1)$$

([Qiu et al., 1998, Thm. 1.7] and [Yang and Tian, 2019, Thm. 2])

$$\frac{\pi}{2} \leq \mathcal{K}(k) \qquad k \in [0,1] \qquad \text{([e.g. Qiu et al., 1998])}$$

where in the first statement we have used that $\mathcal{K}'(k) = \mathcal{K}(k')$. For Jacobi elliptic functions we have that

$$0 < \mathrm{dn}(u\mathcal{K}(k)|k) < 1 \qquad u \in (0,1),\ k \in (0,1) \qquad \text{([e.g. Meyer, 2001])}$$

$$0 < \mathrm{sn}(u\mathcal{K}(k)|k) < 1 \qquad u \in (0,1),\ k \in (0,1) \qquad \text{([e.g. Meyer, 2001])}$$

$$\mathrm{sn}(\pi u/2|0) < \mathrm{sn}(u\mathcal{K}(k)|k) < 1 \qquad u \in (0,1),\ k \in (0,1)$$

([Carlson and Todd, 1983, Thm. 1])

where in the last inequality we have used that $\mathcal{K}(0) = \pi/2$ [e.g. Abramowitz and Stegun, 1948]. Coupling the final inequality above with $\mathrm{sn}(\pi u/2|0) = \sin(\pi u/2)$ for $u \in (0,1)$ we have that

$$\sin(\pi u/2) < \mathrm{sn}(u\mathcal{K}(k)|k) < 1 \qquad u \in (0,1),\ k \in (0,1).$$

Now, for each q we have that

$$\begin{aligned}\frac{w_q}{t_q} = \frac{\widetilde{w}_q}{\sigma_q^2} &= \left(\frac{-2\sqrt{\lambda_{\min}}}{\pi Q \lambda_{\min}}\right) \frac{\mathcal{K}'(k)\mathrm{cn}\left(iu_q\mathcal{K}'(k) \mid k\right)\mathrm{dn}\left(iu_q\mathcal{K}'(k) \mid k\right)}{\mathrm{sn}(iu_q\mathcal{K}'(k) \mid k)^2} \\ &= \left(\frac{2\mathcal{K}'(k)}{\pi Q \lambda_{\min}}\right) \frac{\mathrm{dn}\left(u_q\mathcal{K}(k') \mid k'\right)}{\mathrm{sn}(u_q\mathcal{K}(k') \mid k')^2}\end{aligned}$$

(via Jacobi imaginary transforms [e.g. Abramowitz and Stegun, 1948])

Consequently, we may conclude that

$$\begin{aligned}\frac{|w_q|}{|t_q|} &= \left(\frac{2\mathcal{K}'(k)}{\pi Q \lambda_{\min}}\right) \frac{\mathrm{dn}\left(u_q\mathcal{K}(k') \mid k'\right)}{\mathrm{sn}(u_q\mathcal{K}(k') \mid k')^2} \\ &\leq \frac{2\log(5/k)}{\pi Q \lambda_{\min}} \left(\frac{1}{\sin^2(\pi u_q/2)}\right)\end{aligned}$$

where we note that all quantities on the right hand side are positive. Plugging in the values of $k = 1/\sqrt{\kappa(\mathbf{K})}$, $u_q = (q - 1/2)/Q$ and summing over u_q we see that

$$\sum_{q=1}^{Q} \frac{|w_q|}{|t_q|} < \sum_{q=1}^{Q} \frac{2\log\left(5\sqrt{\kappa(\mathbf{K})}\right)}{\pi Q \sqrt{\lambda_{\min}} \sin^2(\frac{\pi(q-1/2)}{2Q})}. \tag{B.9}$$

Through trigonometric identities: $\sum_{q=1}^{Q} 1/(Q \sin^2 \frac{\pi(q-1/2)}{2Q}) = 2Q$ and, therefore,

$$\sum_{q=1}^{Q} \frac{|w_q|}{|t_q|} < \frac{4Q \log\left(5\sqrt{\kappa(\mathbf{K})}\right)}{\pi\sqrt{\lambda_{\min}}}.$$

□

With these lemmas we are now able to prove Theorem 5.1:

Theorem 5.1 (Restated). *Let* $\mathbf{K} \succ 0$ *and* $\mathbf{b}$ *be inputs to msMINRES-CIQ, producing* $\mathbf{a}_J \approx \mathbf{K}^{1/2}\mathbf{b}$ *after* J *iterations with* Q *quadrature points. The difference between* $\mathbf{a}_J$ *and* $\mathbf{K}^{1/2}\mathbf{b}$ *is bounded by:*

$$\left\|\mathbf{a}_J - \mathbf{K}^{\frac{1}{2}}\mathbf{b}\right\|_2 \leq \overbrace{\mathcal{O}\left(\exp\left(-\frac{2Q\pi^2}{\log\kappa(\mathbf{K})+3}\right)\right)}^{\textit{Quadrature error}} + \overbrace{\frac{2Q\log\left(5\sqrt{\kappa(\mathbf{K})}\right)\kappa(\mathbf{K})\sqrt{\lambda_{min}}}{\pi}\left(\frac{\sqrt{\kappa(\mathbf{K})}-1}{\sqrt{\kappa(\mathbf{K})}+1}\right)^{J-1}\|\mathbf{b}\|_2}^{\textit{msMINRES error}}.$$

where $\lambda_{max}, \lambda_{min}$ *are the max and min eigenvalues of* $\mathbf{K}$*, and* $\kappa(\mathbf{K})$ *is the condition number of* $\mathbf{K}$*.*

Proof. First we note that the msMINRES-CIQ solution $\mathbf{a}_J$ can be written as $\sum_{i=1} w_q \mathbf{c}_J^{(q)}$, where $\mathbf{c}_J^{(q)}$ is the q^{th} shifted solve $\approx (t_q\mathbf{I} + \mathbf{K})^{-1}\mathbf{b}$ from msMINRES.

Applying the triangle inequality we have:

$$\begin{aligned}
\left\|\mathbf{a}_J - \mathbf{K}^{\frac{1}{2}}\mathbf{b}\right\|_2 = \Bigg\| &\overbrace{\sum_{q=1}^{Q} w_q \mathbf{c}_J^{(q)} - \left(\mathbf{K}\sum_{q=1}^{Q} w_q \left(t_q\mathbf{I}+\mathbf{K}\right)^{-1}\right)\mathbf{b}}^{\text{msMINRES error}} \\
&+ \underbrace{\left(\mathbf{K}\sum_{q=1}^{Q} w_q \left(t_q\mathbf{I}+\mathbf{K}\right)^{-1}\right)\mathbf{b} - \mathbf{K}^{\frac{1}{2}}\mathbf{b}}_{\text{Quadrature error}}\Bigg\|_2 \\
&\leq \sum_{q=1}^{Q} |w_q| \left\|\mathbf{c}_J^{(q)} - \mathbf{K}\left(t_q\mathbf{I}+\mathbf{K}\right)^{-1}\mathbf{b}\right\|_2 \\
&\quad + \left\|\mathbf{K}\left(\sum_{q=1}^{Q} w_q \left(t_q\mathbf{I}+\mathbf{K}\right)^{-1}\right)\mathbf{b} - \mathbf{K}^{\frac{1}{2}}\mathbf{b}\right\|_2 \qquad \text{(B.10)}
\end{aligned}$$

Plugging Lemma B.2 into the msMINRES part of the bound bound, we have:

$$\begin{aligned}
&\sum_{q=1}^{Q} |w_q| \left(\frac{\sqrt{\kappa(\mathbf{K}+t_q\mathbf{I})}-1}{\sqrt{\kappa(\mathbf{K}+t_q\mathbf{I})}+1}\right)^J \|\mathbf{b}\|_2 \\
&\quad\leq \sum_{q=1}^{Q} |w_q| \left(\frac{\sqrt{\kappa(\mathbf{K}+t_q\mathbf{I})}-1}{\sqrt{\kappa(\mathbf{K}+t_q\mathbf{I})}+1}\right)\left(\frac{\sqrt{\kappa(\mathbf{K})}-1}{\sqrt{\kappa(\mathbf{K})}+1}\right)^{J-1} \|\mathbf{b}\|_2 \quad \text{(via Lemma B.2)} \\
&\quad\leq \sum_{q=1}^{Q} |w_q| \left(\frac{\lambda_{\max}}{4t_q}\right)\left(\frac{\sqrt{\kappa(\mathbf{K})}-1}{\sqrt{\kappa(\mathbf{K})}+1}\right)^{J-1} \|\mathbf{b}\|_2 \quad \text{(via Lemma B.3)} \\
&\quad\leq \frac{2Q\log\left(5\sqrt{\kappa(\mathbf{K})}\right)\lambda_{\max}}{\pi\sqrt{\lambda_{\min}}}\left(\frac{\sqrt{\kappa(\mathbf{K})}-1}{\sqrt{\kappa(\mathbf{K})}+1}\right)^{J-1} \|\mathbf{b}\|_2 \quad \text{(via Lemma B.4)} \\
&\quad\leq \frac{2Q\log\left(5\sqrt{\kappa(\mathbf{K})}\right)\sqrt{\lambda_{\min}}\kappa(\mathbf{K})}{\pi}\left(\frac{\sqrt{\kappa(\mathbf{K})}-1}{\sqrt{\kappa(\mathbf{K})}+1}\right)^{J-1} \|\mathbf{b}\|_2 .
\end{aligned}$$

Plugging this bound and Lemma B.1 into Eq. (B.10) completes the proof. □

We can also prove this simple corollary:

Corollary B.1. *Let* $\mathbf{K} \succ 0$ *and* $\mathbf{b}$ *be the inputs to Algorithm 5.2, producing the output* $\mathbf{a}'_J \approx \mathbf{K}^{-1/2}\mathbf{b}$ *after* J *iterations with* Q *quadrature points. The difference between* $\mathbf{a}'_J$

and $\mathbf{K}^{-1/2}\mathbf{b}$ *is bounded by:*

$$\left\|\mathbf{a}'_J - \mathbf{K}^{-\frac{1}{2}}\mathbf{b}\right\|_2 \leq \overbrace{\mathcal{O}\left(\frac{1}{\lambda_{min}}\exp\left(-\frac{2Q\pi^2}{\log\kappa(\mathbf{K})+3}\right)\right)}^{\textit{Quadrature error}} + \overbrace{\frac{2Q\log\left(5\sqrt{\kappa(\mathbf{K})}\right)\kappa(\mathbf{K})}{\sqrt{\lambda_{min}}\pi}\left(\frac{\sqrt{\kappa(\mathbf{K})}-1}{\sqrt{\kappa(\mathbf{K})}+1}\right)^{J-1}\|\mathbf{b}\|_2}^{\textit{msMINRES error}}.$$

where $\lambda_{max}, \lambda_{min}$ *are the maximal and minimal eigenvalues of* $\mathbf{K}$*, and* $\kappa(\mathbf{K})$ *is the condition number of* $\mathbf{K}$*.*

Proof. Note that $\mathbf{a}'_J = \mathbf{K}^{-1}\mathbf{a}_J$, where $\mathbf{a}_J$ is the msMINRES-CIQ estimate of $\mathbf{K}^{1/2}\mathbf{b}$. Using the sub-multiplicative property of the induced matrix 2-norm we see that

$$\left\|\mathbf{a}'_J - \mathbf{K}^{-\frac{1}{2}}\mathbf{b}\right\|_2 \leq \left\|\mathbf{K}^{-1}\right\|_2 \left\|\mathbf{a}_J - \mathbf{K}^{\frac{1}{2}}\mathbf{b}\right\|_2 = \frac{1}{\lambda_{\min}}\left\|\mathbf{a}_J - \mathbf{K}^{\frac{1}{2}}\mathbf{b}\right\|_2,$$

where the final term is bounded by Theorem 5.1. □

APPENDIX C

DETAILS ON NATURAL GRADIENT DESCENT WITH CIQ-BASED SVGP

When performing variational inference, we must optimize the $\mathbf{m}'$ and $\mathbf{S}'$ parameters of the whitened variational distribution $q(\mathbf{u}') = \mathcal{N}[\mathbf{m}', \mathbf{S}']$. Rather than using standard gradient descent methods on these parameters, many have suggested that **natural gradient descent (NGD)** is better suited for variational inference [Hoffman et al., 2013, Hensman et al., 2012, Salimbeni et al., 2018b]. NGD performs the following update:

$$\begin{bmatrix} \mathbf{m}' & \mathbf{S}' \end{bmatrix} \leftarrow \begin{bmatrix} \mathbf{m}' & \mathbf{S}' \end{bmatrix} - \varphi \boldsymbol{\mathcal{F}}^{-1} \begin{bmatrix} \frac{\partial \text{ELBO}}{\partial \mathbf{m}'} & \frac{\partial \text{ELBO}}{\partial \mathbf{S}'} \end{bmatrix} \tag{C.1}$$

where φ is a step size, $\begin{bmatrix} \frac{\partial \text{ELBO}}{\partial \mathbf{m}'} & \frac{\partial \text{ELBO}}{\partial \mathbf{S}'} \end{bmatrix}$ is the ELBO gradient, and $\boldsymbol{\mathcal{F}}$ is the *Fisher information matrix* of the variational parameters. Conditioning the gradient with $\boldsymbol{\mathcal{F}}^{-1}$ results in descent directions that are better suited towards distributional parameters [Hoffman et al., 2013].

For Gaussian distributions (and other exponential family distributions) the Fisher information matrix does not need to be explicitly computed. Instead, there is a simple closed-form update that relies on different parameterizations of the Gaussian $\mathcal{N}[\mathbf{m}', \mathbf{S}']$:

$$\begin{bmatrix} \boldsymbol{\theta} & \boldsymbol{\Theta} \end{bmatrix} \leftarrow \begin{bmatrix} \boldsymbol{\theta} & \boldsymbol{\Theta} \end{bmatrix} - \varphi \begin{bmatrix} \frac{\partial \text{ELBO}}{\partial \boldsymbol{\eta}} & \frac{\partial \text{ELBO}}{\partial \mathbf{H}} \end{bmatrix}. \tag{C.2}$$

$[\boldsymbol{\theta},\ \boldsymbol{\Theta}]$ are the Gaussian's *natural parameters* and $[\boldsymbol{\eta},\ \mathbf{H}]$ are the Gaussian's *expectation parameters*:

$$\boldsymbol{\theta} = \mathbf{S}'^{-1}\mathbf{m}', \quad \boldsymbol{\Theta} = -\frac{1}{2}\mathbf{S}'^{-1},$$
$$\boldsymbol{\eta} = \mathbf{m}', \quad \mathbf{H} = \mathbf{m}'\mathbf{m}'^{\top} + \mathbf{S}'$$

In many NGD implementations, it is common to store the variational parameters via their natural representation ($\boldsymbol{\theta}$, $\boldsymbol{\Theta}$), compute the ELBO via the standard parameters ($\mathbf{m}'$, $\mathbf{S}'$), and then compute the derivative via the expectation parameters ($\boldsymbol{\eta}$, $\mathbf{H}$). Unfortunately, converting between these three parameterizations requires $\mathcal{O}(M^3)$ computation. (To see why this is the case, note that computing $\mathbf{S}'$ essentially requires inverting the $\boldsymbol{\Theta}$ matrix.)

A $\mathcal{O}(M^2)$ NGD update. In what follows, we will demonstrate that the ELBO and its derivative can be computed from $\boldsymbol{\theta}$ and $\boldsymbol{\Theta}$ in $\mathcal{O}(M^2)$ time via careful bookkeeping. Consequently, NGD updates have the same asymptotic complexity as the other computations required for SVGP. Recall that the ELBO is given by

$$\text{ELBO} = \overbrace{\sum_{i=1}^{N} \mathop{\mathbb{E}}_{q(f(\mathbf{x}^{(i)}))} \left[\log p(y^{(i)} \mid f(\mathbf{x}^{(i)})) \right]}^{\text{expected log likelihood}} - \mathrm{KL}\left[q(\mathbf{u}) \| p(\mathbf{u}) \right]$$

We will separately analyze the expected log likelihood and KL divergence computations.

C.1 The Expected Log Likelihood and its Gradient

Assume we are estimating the ELBO from a single data point $\mathbf{x}, y$. The expected log likelihood term of the ELBO is typically computed via Gauss-Hermite quadrature or Monte Carlo integration [Hensman et al., 2015b]:[1]

$$\mathop{\mathbb{E}}_{q(f(\mathbf{x})} \left[\log p(y \mid f(\mathbf{x})) \right] = \sum_{s=1}^{S} w_s p(y \mid f_s), \quad f_s = \mu^*_{\text{aprx}}(\mathbf{x}) + \text{Var}^*_{\text{aprx}}(\mathbf{x})^{1/2} \varepsilon_s$$

[1] It can also be computed analytically for Gaussian distributions [Hensman et al., 2013]. The analytic form achieves the same derivative decomposition as in Eq. (C.3) and so the following analysis will still apply.

where w_s are the quadrature weights (or $1/S$ for MC integration) and ε_s are the quadrature locations (or samples from $\mathcal{N}[0,1]$ for MC integration). Therefore, the variational parameters only interact with the expected log likelihood term via $\mu^*_{\text{aprx}}(\mathbf{x})$ and $\text{Var}^*_{\text{aprx}}(\mathbf{x})$. We can write its gradients via chain rule as:

$$\begin{aligned}\frac{\partial \mathbb{E}_{q(f(\mathbf{x}))}[\log p(y|f(\mathbf{x}))]}{\partial \boldsymbol{\eta}} &= c_1 \frac{\partial \mu^*_{\text{aprx}}(\mathbf{x})}{\partial \boldsymbol{\eta}} + c_2 \frac{\partial \text{Var}^*_{\text{aprx}}(\mathbf{x})}{\partial \boldsymbol{\eta}} \\ \frac{\partial \mathbb{E}_{q(f(\mathbf{x}))}[\log p(y|f(\mathbf{x}))]}{\partial \mathbf{H}} &= c_3 \frac{\partial \mu^*_{\text{aprx}}(\mathbf{x})}{\partial \mathbf{H}} + c_4 \frac{\partial \text{Var}^*_{\text{aprx}}(\mathbf{x})}{\partial \mathbf{H}}\end{aligned} \tag{C.3}$$

for some constants c_1, c_2, c_3, and c_4 that do not depend on the variational parameters. It thus suffices to show that the posterior mean/variance and their gradients can be computed from $\boldsymbol{\theta}$ and $\boldsymbol{\Theta}$ in $\mathcal{O}(M^2)$ time.

The predictive distribution and its gradient. All expensive computations involving $\boldsymbol{\theta}$ and $\boldsymbol{\Theta}$ are written in blue.

$\mu^*_{\text{aprx}}(\mathbf{x})$ and its derivative can be written as:

$$\begin{aligned}\mu^*_{\text{aprx}}(\mathbf{x}) &= \mathbf{k}_{\mathbf{Zx}}^\top \mathbf{K}_{\mathbf{ZZ}}^{-1/2} \mathbf{m}' && \text{(standard parameters)} \\ &= \mathbf{k}_{\mathbf{Zx}}^\top \mathbf{K}_{\mathbf{ZZ}}^{-1/2} \boldsymbol{\eta} && \text{(expectation parameters)} \\ &= \mathbf{k}_{\mathbf{Zx}}^\top \mathbf{K}_{\mathbf{ZZ}}^{-1/2} (-2\boldsymbol{\Theta})^{-1} \boldsymbol{\theta}, && \text{(C.4)} \\ \frac{\partial \mu^*_{\text{aprx}}(\mathbf{x})}{\partial \boldsymbol{\eta}} &= \mathbf{K}_{\mathbf{ZZ}}^{-1/2} \mathbf{k}_{\mathbf{Zx}}, && \text{(C.5)} \\ \frac{\partial \mu^*_{\text{aprx}}(\mathbf{x})}{\partial \mathbf{H}} &= \mathbf{0}.\end{aligned}$$

$\text{Var}^*_{\text{aprx}}(\mathbf{x})$ and its derivative can be written as:

$$\begin{aligned}
\text{Var}^*_{\text{aprx}}(\mathbf{x}) &= \mathbf{k}_{\mathbf{Z}\mathbf{x}}^\top \mathbf{K}_{\mathbf{Z}\mathbf{Z}}^{-1/2} \left(\mathbf{S}' - \mathbf{I}\right) \mathbf{K}_{\mathbf{Z}\mathbf{Z}}^{-1/2} \mathbf{k}_{\mathbf{Z}\mathbf{x}} && \text{(standard parameters)} \\
&= \mathbf{k}_{\mathbf{Z}\mathbf{x}}^\top \mathbf{K}_{\mathbf{Z}\mathbf{Z}}^{-1/2} \left(\mathbf{H} - \boldsymbol{\eta}\boldsymbol{\eta}^\top - \mathbf{I}\right) \mathbf{K}_{\mathbf{Z}\mathbf{Z}}^{-1/2} \mathbf{k}_{\mathbf{Z}\mathbf{x}} && \text{(expectation parameters)} \\
&= \mathbf{k}_{\mathbf{Z}\mathbf{x}}^\top \mathbf{K}_{\mathbf{Z}\mathbf{Z}}^{-1/2} \left((-2\boldsymbol{\Theta})^{-1} - \mathbf{I}\right) \mathbf{K}_{\mathbf{Z}\mathbf{Z}}^{-1/2} \mathbf{k}_{\mathbf{Z}\mathbf{x}}, && \text{(C.6)} \\
\frac{\partial \text{Var}^*_{\text{aprx}}(\mathbf{x})}{\partial \boldsymbol{\eta}} &= -2 \left(\mathbf{k}_{\mathbf{Z}\mathbf{x}}^\top \mathbf{K}_{\mathbf{Z}\mathbf{Z}}^{-1/2} (-2\boldsymbol{\Theta})^{-1} \boldsymbol{\theta}\right) \mathbf{K}_{\mathbf{Z}\mathbf{Z}}^{-1/2} \mathbf{k}_{\mathbf{Z}\mathbf{x}}, && \text{(C.7)} \\
\frac{\partial \text{Var}^*_{\text{aprx}}(\mathbf{x})}{\partial \mathbf{H}} &= \left(\mathbf{K}_{\mathbf{Z}\mathbf{Z}}^{-1/2} \mathbf{k}_{\mathbf{Z}\mathbf{x}}^\top\right) \left(\mathbf{k}_{\mathbf{Z}\mathbf{x}}^\top \mathbf{K}_{\mathbf{Z}\mathbf{Z}}^{-1/2}\right). && \text{(C.8)}
\end{aligned}$$

In Eqs. (C.4) to (C.8), the only expensive operation involving $\mathbf{K}_{\mathbf{Z}\mathbf{Z}}$ is $\mathbf{K}_{\mathbf{Z}\mathbf{Z}}^{-1/2}\mathbf{k}_{\mathbf{Z}\mathbf{x}}$, which can be computed with CIQ. The only expensive operation involving the variational parameters is $(-2\boldsymbol{\Theta})^{-1}\mathbf{K}_{\mathbf{Z}\mathbf{Z}}^{-1/2}\mathbf{k}_{\mathbf{Z}\mathbf{x}}$, which can be computed with preconditioned conjugate gradients after computing $\mathbf{K}_{\mathbf{Z}\mathbf{Z}}^{-1/2}\mathbf{k}_{\mathbf{Z}\mathbf{x}}$.[2] Those operations only need to be computed once, and then they can be reused across Eqs. (C.4) to (C.8). In total, the entire computation for the expected log likelihood and its derivative is $\mathcal{O}(M^2)$.

C.2 The KL Divergence and its Gradient

We will demonstrate that the KL divergence and its gradient can be computed from $\boldsymbol{\theta}$ and $\boldsymbol{\Theta}$ in $\mathcal{O}(M^2)$ time. All expensive computations involving $\boldsymbol{\theta}$ and $\boldsymbol{\Theta}$ are written in blue.

[2] We typically apply a Jacobi preconditioner to these solves.

The whitened KL divergence from Section 5.4.1 is given by:

$$\begin{aligned}
\mathrm{KL}\left[\, q(\mathbf{u}')\|p(\mathbf{u}')\,\right] &= \frac{1}{2}\left[\mathbf{m}'^{\top}\mathbf{m}' + \mathrm{Tr}\left(\mathbf{S}'\right) - \log|\mathbf{S}'| - M\right] \quad \text{(standard parameters)} \\
&= \frac{1}{2}\left[\mathrm{Tr}\left(\mathbf{H}\right) - \log|\mathbf{H} - \boldsymbol{\eta}\boldsymbol{\eta}^{\top}| - M\right] \quad \text{(expectation parameters)} \\
&= \frac{1}{2}\left[\boldsymbol{\theta}^{\top}(-2\boldsymbol{\Theta})^{-2}\boldsymbol{\theta} + \mathrm{Tr}\left((-2\boldsymbol{\Theta})^{-1}\right) + \log|-2\boldsymbol{\Theta}| - M\right]. \quad \text{(C.9)}
\end{aligned}$$

The KL derivative with respect to $\boldsymbol{\eta}$ and $\mathbf{H}$ is surprisingly simple when re-written in terms of the natural parameters

$$\begin{aligned}
\frac{\partial\,\mathrm{KL}\left[\, q(\mathbf{u}')\|p(\mathbf{u}')\,\right]}{\partial\boldsymbol{\eta}} &= \left(\mathbf{H} - \boldsymbol{\eta}\boldsymbol{\eta}^{\top}\right)^{-1}\boldsymbol{\eta} = \left(\mathbf{S}'\right)^{-1}\boldsymbol{\eta} \\
&= \boldsymbol{\theta} \quad \text{(C.10)} \\
\frac{\partial\,\mathrm{KL}\left[\, q(\mathbf{u}')\|p(\mathbf{u}')\,\right]}{\partial\mathbf{H}} &= \frac{1}{2}\mathbf{I} - \frac{1}{2}\left(\mathbf{H} - \boldsymbol{\eta}\boldsymbol{\eta}^{\top}\right)^{-1} = \frac{1}{2}\mathbf{I} - \frac{1}{2}(\mathbf{S}')^{-1} \\
&= \frac{1}{2}\mathbf{I} + \boldsymbol{\Theta}. \quad \text{(C.11)}
\end{aligned}$$

Thus the derivative of the KL divergence only takes $\mathcal{O}(M^2)$ time to compute. The forward pass can also be computed in $\mathcal{O}(M^2)$ time—using stochastic trace estimation for the trace term [Cutajar et al., 2016, Gardner et al., 2018a], stochastic Lanczos quadrature for the log determinant [Ubaru et al., 2017, Dong et al., 2017], and CG for the solves. However, during training the forward pass can be omitted as only the gradient is needed for NGD steps.

APPENDIX D

FULL GPYTORCH CODE EXAMPLES

The following are code examples for training and evaluating GPyTorch models.[1] We include examples for a standard GP (with no approximations/exploitable structure) and a multitask GP. Both models can be modified to use scalable methods by changing the `covar_module` (i.e. kernel).

D.1 Standard GP Regression

Here we train a standard GP with a `RBFKernel`. As described in Section 3.4, each kernel object outputs a `LazyTensor` object, which defines its own `_matmul(·)` function. If the kernel has exploitable structure—e.g.

- `LinearKernel` for Bayesian linear regression,
- `GridInterpolationKernel` wrapping a `RBFKernel` for KISS-GP,

then the kernel will output the appropriate `LazyTensor` subclass with a structure-exploiting `_matmul(·)` function for use with the mBCG algorithm.

```
import math
import torch
import gpytorch
from matplotlib import pyplot as plt

"""
Training data is 100 points in [0,1] inclusive regularly spaced
True function is sin(2*pi*x) with Gaussian noise
"""
train_x = torch.linspace(0, 1, 100)
```

[1] Tested against GPyTorch v1.1

```
train_y = torch.sin(train_x * (2 * math.pi)) + \
    torch.randn(train_x.size()) * math.sqrt(0.04)

"""
Now we define a class for basic GP models
"""
class ExactGPModel(gpytorch.models.ExactGP):
    def __init__(self, train_x, train_y, likelihood):
        super(ExactGPModel, self).__init__(
            train_x, train_y, likelihood
        )
        self.mean_module = gpytorch.means.ZeroMean()
        # We can implement specialty models by replacing
        # this kernel (e.g. LinearKernel.)
        # Each kernel uses a differen LazyTensor under the hood.
        self.covar_module = gpytorch.kernels.ScaleKernel(
            gpytorch.kernels.RBFKernel()
        )
        # To implement KISS-GP, wrap this kernel inside al
        # gpytorch.kernels.GridInterpolationKernel

    def forward(self, x):
        mean_x = self.mean_module(x)
        # Our kernel module returns a NonLazyTensor object.
        # If we were to replace it with a LinearKernel,
        # the output would be a RootLazyTensor
        covar_x = self.covar_module(x)
        return gpytorch.distributions.MultivariateNormal(
            mean_x, covar_x
        )

"""
Create an instance of our model and likelihood
"""
likelihood = gpytorch.likelihoods.GaussianLikelihood()
model = ExactGPModel(train_x, train_y, likelihood)

"""
A basic training loop.
The GPyTorch objects in this loop use BBMM under the hood.
"""
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

# "Loss" for GPs - the marginal log likelihood
# Calling this funcition uses BBMM to compute the marginal log
# likelihood and its derivative
mll = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood, model)

# Training loop
model.train()
for i in range(100):
```

```
    optimizer.zero_grad()
    loss = -mll(model(train_x), train_y)
    loss.backward()
    optimizer.step()

"""
Making predictions.
We will use LOVE to for fast variances
"""
model.eval()
likelihood.eval()
# The fast_pred_var context manager turns on LOVE variances
with torch.no_grad(), gpytorch.settings.fast_pred_var():
    test_x = torch.linspace(0, 1, 51)
    pred = likelihood(model(test_x))
    # Get mean prediction
    mean = pred.mean
    # Get upper and lower confidence bounds
    lower, upper = pred.confidence_region()

"""
Plotting the model fit
"""
f, ax = plt.subplots(1, 1, figsize=(4, 3))
# Plot training data as black stars
ax.plot(train_x.numpy(), train_y.numpy(), "k*")
# Plot prediction
ax.plot(test_x.numpy(), mean.numpy(), "b")
ax.fill_between(
  test_x.numpy(), lower.numpy(), upper.numpy(), alpha=0.5
)
ax.set(ylim=[-3, 3], xlabel="x",  ylabel="y")
ax.legend(["Observed Data", "Mean", "Confidence"])
f.show()
```

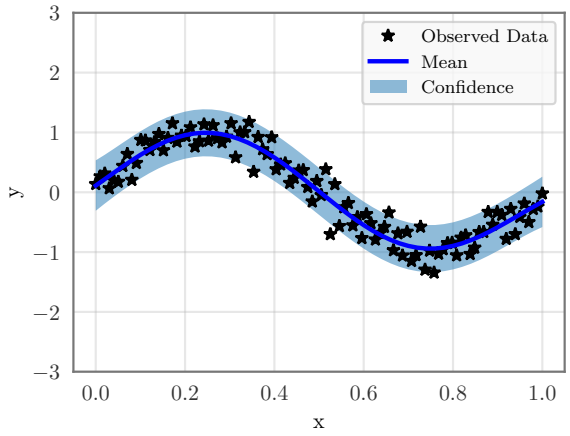

Figure D.1: Output plot from GPyTorch code example for standard GPs.

D.2 Multitask GP Regression

To demonstrate the modularity afforded by MVM methods, we also include a code example of a multitask GP model. What's notable is that this code example is essentially the same as the standard GP code example. The only major difference is the kernel module (`MultitaskKernel`), which uses the (`KroneckerProductLazyTensor`) under the hood for efficient inference.

```
import math
import torch
import gpytorch
from matplotlib import pyplot as plt

"""
Training data is 100 points in [0,1] inclusive regularly spaced
We train two outputs: a sin function and a cos function
    with Gaussian noise
"""
train_x = torch.linspace(0, 1, 100)
train_y = torch.stack([
    torch.sin(train_x * (2 * math.pi)),
    torch.cos(train_x * (2 * math.pi))
], -1)
train_y += torch.randn_like(train_y) * 0.2

"""
Now we define a class for multitask GP models
"""
class MultitaskGPModel(gpytorch.models.ExactGP):
    def __init__(self, train_x, train_y, likelihood):
        super(MultitaskGPModel, self).__init__(
            train_x, train_y, likelihood
        )
        self.mean_module = gpytorch.means.MultitaskMean(
            gpytorch.means.ZeroMean(), num_tasks=2
        )
        self.covar_module = gpytorch.kernels.MultitaskKernel(
            gpytorch.kernels.RBFKernel(), num_tasks=2, rank=1
        )

    def forward(self, x):
        mean_x = self.mean_module(x)
        covar_x = self.covar_module(x)
        return gpytorch.distributions.MultitaskMultivariateNormal(
```

```
            mean_x, covar_x
        )

"""
Create an instance of our model and likelihood
This example uses a MultitaskGaussianLikelihood to have seperate
    observation noise for each task.
"""
likelihood = gpytorch.likelihoods.MultitaskGaussianLikelihood(
    num_tasks=2
)
model = MultitaskGPModel(train_x, train_y, likelihood)


"""
Training with BBMM.
"""
optimizer = torch.optim.Adam(model.parameters(), lr=0.1)
mll = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood, model)

model.train()
for i in range(100):
    optimizer.zero_grad()
    loss = -mll(model(train_x), train_y)
    loss.backward()
    optimizer.step()

"""
Making predictions with LOVE.
"""
model.eval()
likelihood.eval()
with torch.no_grad(), gpytorch.settings.fast_pred_var():
    test_x = torch.linspace(0, 1, 51)
    pred = likelihood(model(test_x))
    mean = pred.mean
    lower, upper = pred.confidence_region()


"""
Plotting the model fit
"""
f, (y1_ax, y2_ax) = plt.subplots(1, 2, figsize=(8, 3))
# Plot training data as black stars
y1_ax.plot(
    train_x.detach().numpy(), train_y[:, 0].detach().numpy(),
    "k*"
)
y2_ax.plot(
    train_x.detach().numpy(), train_y[:, 1].detach().numpy(),
    "k*"
```

```
)
# Plot predictions
y1_ax.plot(test_x.numpy(), mean[:, 0].numpy(), "b")
y2_ax.plot(test_x.numpy(), mean[:, 1].numpy(), "b")
# Shade in confidence
y1_ax.fill_between(
    test_x.numpy(), lower[:, 0].numpy(), upper[:, 0].numpy(),
    alpha=0.5
)
y2_ax.fill_between(
    test_x.numpy(), lower[:, 1].numpy(), upper[:, 1].numpy(),
    alpha=0.5
)
# Legend and axes
y1_ax.legend(["Observed Data", "Mean", "Confidence"])
y1_ax.set(ylim=[-3, 3], xlabel="x",  ylabel="y1")
y2_ax.set(ylim=[-3, 3], xlabel="x",  ylabel="y2")
f.show()
```

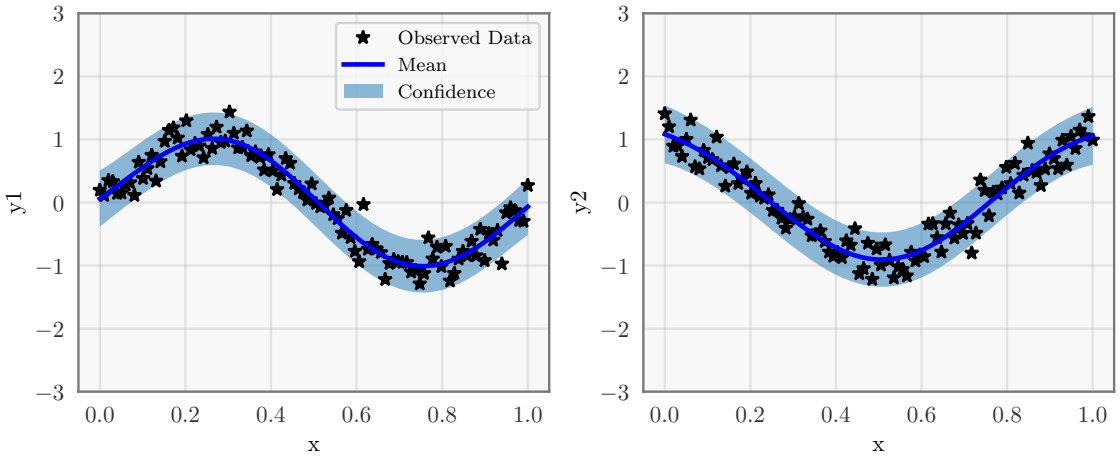

Figure D.2: Output plot from GPyTorch code example for multitask GPs.

BIBLIOGRAPHY

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

Milton Abramowitz and Irene A. Stegun. *Handbook of mathematical functions with formulas, graphs, and mathematical tables*, volume 55. US Government Printing Office, 1948.

Sanjeev Arora, Simon S. Du, Wei Hu, Zhiyuan Li, and Ruosong Wang. Fine-grained analysis of optimization and generalization for overparameterized two-layer neural networks. In *ICML*, 2019.

Arthur Asuncion and David Newman. UCI machine learning repository. `https://archive.ics.uci.edu/ml/`, 2007. Last accessed: 2018-05-18.

Erlend Aune, Jo Eidsvik, and Yvo Pokern. Iterative numerical methods for sampling from high dimensional Gaussian distributions. *Statistics and Computing*, 23(4):501–521, 2013.

Erlend Aune, Daniel P. Simpson, and Jo Eidsvik. Parameter estimation in high dimensional Gaussian distributions. *Statistics and Computing*, 24(2):247–263, 2014.

Haim Avron and Sivan Toledo. Randomized algorithms for estimating the trace of an implicit symmetric positive semi-definite matrix. *Journal of the ACM*, 58 (2):8, 2011.

Francis Bach. Sharp analysis of low-rank kernel matrix approximations. In *COLT*, 2013.

Maximilian Balandat, Brian Karrer, Daniel R. Jiang, Samuel Daulton, Benjamin Letham, Andrew Gordon Wilson, and Eytan Bakshy. BoTorch: Programmable Bayesian optimization in PyTorch. *arXiv preprint arXiv:1910.06403*, 2019.

Matthias Bauer, Mark van der Wilk, and Carl Edward Rasmussen. Understanding probabilistic sparse Gaussian process approximations. In *NeurIPS*, pages 1533–1541, 2016.

Gregory Benton, Wesley J. Maddox, Jayson Salkey, Julio Albinati, and Andrew Gordon Wilson. Function-space distributions over kernels. In *NeurIPS*, pages 14939–14950, 2019.

Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: Deep universal probabilistic programming. *Journal of Machine Learning Research*, 20(1):973–978, 2019.

Christopher M. Bishop. *Pattern recognition and machine learning*. Springer, 2006.

Edwin V. Bonilla, Kian M. Chai, and Christopher Williams. Multi-task Gaussian process prediction. In *NeurIPS*, 2008.

Christopher J. C. Burges. From ranknet to lambdarank to lambdamart: An overview. *Learning*, 11(81):23–581, 2010.

David R. Burt, Carl Edward Rasmussen, and Mark Van Der Wilk. Rates of convergence for sparse variational Gaussian process regression. In *ICML*, 2019.

Roberto Calandra, Jan Peters, Carl Edward Rasmussen, and Marc Deisenroth. Manifold Gaussian processes for regression. In *IJCNN*, pages 3338–3345, 2016.

B. C. Carlson and John Todd. The degenerating behavior of elliptic functions. *Journal on Numerical Analysis*, 20(6):1120–1129, 1983.

Benjamin Charlier, Jean Feydy, Joan Glaunès, François-David Collin, and Ghislain Durif. Kernel operations on the GPU, with autodiff, without memory overflows. 2020.

Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *KDD*, pages 785–794, 2016.

Ching-An Cheng and Byron Boots. Variational inference for Gaussian process models with linear complexity. In *NeurIPS*, pages 5184–5194, 2017.

Sou-Cheng Choi. *Iterative methods for singular linear equations and least-squares problems*. PhD thesis, 2006.

Edmond Chow and Yousef Saad. Preconditioned krylov subspace methods for sampling multivariate Gaussian distributions. *Journal on Scientific Computing*, 36(2):A588–A608, 2014.

John P. Cunningham, Krishna V. Shenoy, and Maneesh Sahani. Fast Gaussian process methods for point process intensity estimation. In *ICML*, 2008.

Kurt Cutajar, Michael Osborne, John Cunningham, and Maurizio Filippone. Preconditioning kernel matrices. In *ICML*, 2016.

Marco Cuturi. Sinkhorn distances: Lightspeed computation of optimal transport. In *NeurIPS*, pages 2292–2300, 2013.

Andreas Damianou and Neil Lawrence. Deep Gaussian processes. In *AISTATS*, pages 207–215, 2013.

Biswa Nath Datta and Youcef Saad. Arnoldi methods for large Sylvester-like observer matrix equations, and an associated algorithm for partial spectrum assignment. *Linear Algebra and its Applications*, 154-156:225 – 244, 1991. ISSN 0024-3795.

Philip I. Davies and Nicholas J. Higham. Computing $f(\mathbf{a})\mathbf{b}$ for matrix functions f. In *QCD and Numerical Analysis III*, pages 15–24. Springer, 2005.

Marc Deisenroth and Jun Wei Ng. Distributed Gaussian processes. In *ICML*, pages 1481–1490, 2015.

Marc Deisenroth and Carl Edward Rasmussen. PILCO: A model-based and data-efficient approach to policy search. In *ICML*, pages 465–472, 2011.

Marc Deisenroth, Dieter Fox, and Carl Edward Rasmussen. Gaussian processes for data-efficient learning in robotics and control. *Pattern Analysis and Machine Intelligence*, 37(2):408–423, 2015.

James W. Demmel. *Applied numerical linear algebra*, volume 56. SIAM, 1997.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL*, 2019.

Kun Dong, David Eriksson, Hannes Nickisch, David Bindel, and Andrew Gordon Wilson. Scalable log determinants for Gaussian process kernel learning. In *NeurIPS*, 2017.

David Duvenaud, James Robert Lloyd, Roger Grosse, Joshua B. Tenenbaum, and Zoubin Ghahramani. Structure discovery in nonparametric regression through compositional kernel search. In *ICML*, 2013.

David Eriksson, Michael Pearce, Jacob Gardner, Ryan D. Turner, and Matthias Poloczek. Scalable global optimization via local Bayesian optimization. In *NeurIPS*, pages 5497–5508, 2019.

Trefor W. Evans and Prasanth B. Nair. Scalable Gaussian processes with grid-structured eigenfunctions (GP-GRIEF). In *ICML*, 2018.

Jack Fitzsimons, Michael Osborne, Stephen J. Roberts, and Joseph F. Fitzsimons. Improved stochastic trace estimation using mutually unbiased bases. In *UAI*, 2018.

Peter Frazier, Warren Powell, and Savas Dayanik. The knowledge-gradient policy for correlated normal beliefs. *Journal on Computing*, 21(4):599–613, 2009.

Roland Freund. On conjugate gradient type methods and polynomial preconditioners for a class of complex non-Hermitian matrices. *Numerische Mathematik*, 57(1):285–312, 1990.

Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997.

Jerome H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of Statistics*, pages 1189–1232, 2001.

Jerome H. Friedman. Stochastic gradient boosting. *Computational Statistics and Data Analysis*, 38(4):367–378, 2002.

Joseph Futoma, Sanjay Hariharan, and Katherine Heller. Learning to detect sepsis with a multitask Gaussian process RNN classifier. In *ICML*, pages 1174–1182, 2017.

Jacob R. Gardner, Chuan Guo, Kilian Q. Weinberger, Roman Garnett, and Roger Grosse. Discovering and exploiting additive structure for Bayesian optimization. In *AISTATS*, pages 1311–1319, 2017.

Jacob R. Gardner, Geoff Pleiss, Kilian Q. Weinberger, David Bindel, and Andrew Gordon Wilson. GPyTorch: Blackbox matrix-matrix Gaussian process inference with GPU acceleration. In *NeurIPS*, pages 7576–7586, 2018a.

Jacob R. Gardner, Geoff Pleiss, Ruihan Wu, Kilian Q. Weinberger, and Andrew Gordon Wilson. Product kernel interpolation for scalable Gaussian processes. In *AISTATS*, 2018b.

Gene H. Golub and Charles F. Van Loan. *Matrix computations*, volume 3. JHU Press, 2012.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

GPy. GPy: A Gaussian process framework in Python. `http://github.com/SheffieldML/GPy`, 2012.

Anne Greenbaum. *Iterative Methods for Solving Linear Systems*. SIAM, 1997.

Nicholas Hale, Nicholas J. Higham, and Lloyd N. Trefethen. Computing A^α, $\log(A)$, and related matrix functions by contour integrals. *Journal on Numerical Analysis*, 46(5):2505–2523, 2008.

Helmut Harbrecht, Michael Peters, and Reinhold Schneider. On the low-rank approximation by the pivoted cholesky decomposition. *Applied Mumerical Mathematics*, 62(4):428–440, 2012.

Marton Havasi, José Miguel Hernández-Lobato, and Juan José Murillo-Fuentes. Inference in deep Gaussian processes using stochastic gradient Hamiltonian Monte Carlo. In *NeurIPS*, pages 7506–7516, 2018.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

James Hensman, Magnus Rattray, and Neil D. Lawrence. Fast variational inference in the conjugate exponential family. In *NeurIPS*, pages 2888–2896, 2012.

James Hensman, Nicolo Fusi, and Neil D. Lawrence. Gaussian processes for big data. In *UAI*, 2013.

James Hensman, Alexander G. de G. Matthews, Maurizio Filippone, and Zoubin Ghahramani. Mcmc for variationally sparse Gaussian processes. In *NeurIPS*, pages 1648–1656, 2015a.

James Hensman, Alexander G. de G. Matthews, and Zoubin Ghahramani. Scalable variational Gaussian process classification. In *ICML*, 2015b.

James Hensman, Nicolas Durrande, and Arno Solin. Variational Fourier features for Gaussian processes. *Journal of Machine Learning Research*, 18(1):5537–5588, 2017.

José Miguel Hernández-Lobato, Matthew W. Hoffman, and Zoubin Ghahramani. Predictive entropy search for efficient global optimization of black-box functions. In *NeurIPS*, pages 918–926, 2014.

José Miguel Hernández-Lobato, James Requeima, Edward O. Pyzer-Knapp, and Alán Aspuru-Guzik. Parallel and distributed Thompson sampling for large-scale accelerated exploration of chemical space. In *ICML*, pages 1470–1479, 2017.

Magnus R. Hestenes et al. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436, 1952.

Nicholas J. Higham. *Functions of matrices: theory and computation*, volume 104. SIAM, 2008.

Matthew D. Hoffman, David Blei, Chong Wang, and John Paisley. Stochastic variational inference. *Journal of Machine Learning Research*, 14(1):1303–1347, 2013.

Jeremy Howard. Training Imagenet in 3 hours for $25; and CIFAR10 for $0.26, Apr 2018. URL `https://www.fast.ai/2018/04/30/dawnbench-fastai/`.

Gao Huang, Zhuang Liu, Kilian Q. Weinberger, and Laurens van der Maaten. Densely connected convolutional networks. In *CVPR*, 2017.

Gao Huang, Zhuang Liu, Geoff Pleiss, Laurens Van Der Maaten, and Kilian Q. Weinberger. Convolutional networks with dense connectivity. *Pattern Analysis and Machine Intelligence*, 2019.

Michael F. Hutchinson. A stochastic estimator of the trace of the influence matrix for laplacian smoothing splines. *Communications in Statistics-Simulation and Computation*, 19(2):433–450, 1990.

Rob J. Hyndman. Time series data library. `http://www-personal.buseco.monash.edu.au/~hyndman/TSDL/`, 2005. Last accessed: 2018-05-18.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015.

Pavel Izmailov, Alexander Novikov, and Dmitry Kropotov. Scalable Gaussian processes with billions of inducing inputs via tensor train decomposition. In *AISTATS*, pages 726–735, 2018a.

Pavel Izmailov, Dmitrii Podoprikhin, Timur Garipov, Dmitry Vetrov, and Andrew Gordon Wilson. Averaging weights leads to wider optima and better generalization. In *UAI*, 2018b.

Arthur Jacot, Franck Gabriel, and Clément Hongler. Neural tangent kernel: Convergence and generalization in neural networks. In *NeurIPS*, pages 8571–8580, 2018.

Martin Jankowiak, Geoff Pleiss, and Jacob R. Gardner. Deep sigma point processes. In *UAI*, 2020a.

Martin Jankowiak, Geoff Pleiss, and Jacob R. Gardner. Parametric Gaussian process regressors. In *ICML*, 2020b.

Beat Jegerlehner. Krylov space solvers for shifted linear systems. *arXiv preprint hep-lat/9612014*, 1996.

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *ACMMM*, pages 675–678, 2014.

Kaggle. Kaggle's state of data science and machine learning 2019. `https://www.kaggle.com/c/kaggle-survey-2019`, 2019.

Kirthevasan Kandasamy, Jeff Schneider, and Barnabás Póczos. High dimensional Bayesian optimisation and bandits via additive models. In *ICML*, pages 295–304, 2015.

Kirthevasan Kandasamy, Akshay Krishnamurthy, Jeff Schneider, and Barnabás Póczos. Parallelised Bayesian optimisation via Thompson sampling. In *AISTATS*, pages 133–142, 2018.

Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *NeurIPS*, pages 3146–3154, 2017.

Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. In *ICLR*, 2017.

Robert Keys. Cubic convolution interpolation for digital image processing. *Acoustics, Speech, and Signal Processing*, 29(6):1153–1160, 1981.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015.

Diederik P. Kingma and Max Welling. Auto-encoding variational Bayes. In *ICLR*, 2014.

Jeremias Knoblauch, Jack Jewson, and Theodoros Damoulas. Generalized variational inference. *arXiv preprint arXiv:1904.02063*, 2019.

Pang Wei Koh and Percy Liang. Understanding black-box predictions via influence functions. In *ICML*, pages 1885–1894, 2017.

Malte Kuss and Carl Edward Rasmussen. Assessing approximate inference for binary Gaussian process classification. *Journal of Machine Learning Research*, 6 (Oct):1679–1704, 2005.

Cornelius Lanczos. *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators*. United States Governm. Press Office Los Angeles, CA, 1950.

Quoc Le, Tamás Sarlós, and Alex Smola. Fastfood: approximating kernel expansions in loglinear time. In *ICML*, 2013.

Yingzhen Li, José Miguel Hernández-Lobato, and Richard E. Turner. Stochastic expectation propagation. In *NeurIPS*, pages 2323–2331, 2015.

Dong C. Liu and Jorge Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45(1-3):503–528, 1989.

Haitao Liu, Yew-Soon Ong, Xiaobo Shen, and Jianfei Cai. When Gaussian process meets big data: A review of scalable GPs. *Neural Networks and Learning Systems*, 2020.

Alexander G. de G. Matthews. *Scalable Gaussian process inference using variational methods*. PhD thesis, University of Cambridge, 2017.

Alexander G. de G. Matthews, James Hensman, Richard Turner, and Zoubin Ghahramani. On sparse variational methods and the kullback-leibler divergence between stochastic processes. In *AISTATS*, pages 231–239, 2016.

Alexander G. de G. Matthews, Mark van der Wilk, Tom Nickson, Keisuke Fujii, Alexis Boukouvalas, Pablo León-Villagrá, Zoubin Ghahramani, and James Hensman. GPflow: A Gaussian process library using TensorFlow. *Journal of Machine Learning Research*, 18(40):1–6, 2017.

Karl Meerbergen. The solution of parametrized symmetric linear systems. *Journal on Matrix Analysis and Applications*, 24(4):1038–1059, 2003.

Kenneth R. Meyer. Jacobi elliptic functions from a dynamical systems point of view. *The American Mathematical Monthly*, 108(8):729–737, 2001.

Charles A. Micchelli, Yuesheng Xu, and Haizhang Zhang. Universal kernels. *Journal of Machine Learning Research*, 7(Dec):2651–2667, 2006.

Thomas P. Minka. *A family of algorithms for approximate Bayesian inference*. PhD thesis, Massachusetts Institute of Technology, 2001.

Iain Murray. Gaussian processes and fast matrix-vector multiplies. In *ICML Workshop on Numerical Mathematics in Machine Learning*, 2009.

Iain Murray, Ryan Adams, and David MacKay. Elliptical slice sampling. In *AISTATS*, pages 541–548, 2010.

Mojmir Mutny and Andreas Krause. Efficient high dimensional Bayesian optimization with additivity and quadrature Fourier features. In *NeurIPS*, pages 9005–9016, 2018.

Duc-Trung Nguyen, Maurizio Filippone, and Pietro Michiardi. Exact Gaussian process regression with distributed computations. In *SAC*, pages 1286–1295, 2019.

Christopher C. Paige. Practical use of the symmetric Lanczos process with reorthogonalization. *BIT Numerical Mathematics*, 10(2):183–195, 1970.

Christopher C. Paige and Michael A. Saunders. Solution of sparse indefinite systems of linear equations. *Journal on Numerical Analysis*, 12(4):617–629, 1975.

Beresford N. Parlett. A new look at the Lanczos algorithm for solving symmetric systems of linear equations. *Linear Algebra and its Applications*, 29:323–346, 1980.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NeurIPS Autodiff Workshop*, 2017.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, pages 8024–8035, 2019.

S. L. Qiu, Mavina Krishna Vamanamurthy, and Matti Vuorinen. Some inequalities for the growth of elliptic integrals. *Journal on Mathematical Analysis*, 29(5): 1224–1237, 1998.

Joaquin Quiñonero-Candela and Carl Edward Rasmussen. A unifying view of sparse approximate Gaussian process regression. *Journal of Machine Learning Research*, 6(Dec):1939–1959, 2005.

Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. In *NeurIPS*, pages 1177–1184, 2008.

Carl Edward Rasmussen and Zoubin Ghahramani. Occam's razor. In *NeurIPS*, pages 294–300, 2001.

Carl Edward Rasmussen and Christopher Williams. *Gaussian processes for machine learning*, volume 1. MIT Press, 2006.

Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic backpropagation and approximate inference in deep generative models. In *ICML*, pages 1278–1286, 2014.

Matthew Richardson, Ewa Dominowska, and Robert Ragno. Predicting clicks: estimating the click-through rate for new ads. In *WWW*, pages 521–530, 2007.

Farbod Roosta-Khorasani and Uri Ascher. Improved bounds on sample size for implicit matrix trace estimators. *Foundations of Computational Mathematics*, 15 (5):1187–1212, 2015.

Youcef Saad. On the Lanczos method for solving symmetric linear systems with several right-hand sides. *Mathematics of Computation*, 48(178):651–662, 1987.

Yousef Saad. *Iterative methods for sparse linear systems*, volume 82. SIAM, 2003.

Yunus Saatçi. *Scalable inference for structured Gaussian process models*. PhD thesis, University of Cambridge, 2012.

Hugh Salimbeni and Marc Deisenroth. Doubly stochastic variational inference for deep Gaussian processes. In *NeurIPS*, pages 4588–4599, 2017.

Hugh Salimbeni, Ching-An Cheng, Byron Boots, and Marc Deisenroth. Orthogonally decoupled variational Gaussian processes. In *NeurIPS*, pages 8711–8720, 2018a.

Hugh Salimbeni, Stefanos Eleftheriadis, and James Hensman. Natural gradients in practice: Non-conjugate variational inference in Gaussian process models. In *AISTATS*, pages 689–697, 2018b.

Robert E. Schapire and Yoav Freund. Boosting: Foundations and algorithms. *Kybernetes*, 2013.

Michael K. Schneider and Alan S. Willsky. Krylov subspace estimation. *Journal on Scientific Computing*, 22(5):1840–1864, 2001.

Peter Schulam and Suchi Saria. A framework for individualizing predictions of disease trajectories by exploiting multi-resolution structure. In *NeurIPS*, pages 748–756, 2015.

Peter Schulam and Suchi Saria. What-if reasoning with counterfactual Gaussian processes. In *NeurIPS*, 2017.

Rishit Sheth and Roni Khardon. Excess risk bounds for the bayes risk using variational inference in latent Gaussian models. In *NeurIPS*, pages 5151–5161, 2017.

Jonathan Richard Shewchuk. An introduction to the conjugate gradient method without the agonizing pain, 1994.

Jiaxin Shi, Michalis K. Titsias, and Andriy Mnih. Sparse orthogonal variational inference for Gaussian processes. In *AISTATS*, 2020.

Bernhard W. Silverman. Some aspects of the spline smoothing approach to non-parametric regression curve fitting. *Journal of the Royal Statistical Society: Series B (Methodological)*, 47(1):1–21, 1985.

Horst D. Simon. The Lanczos algorithm with partial reorthogonalization. *Mathematics of Computation*, 42(165):115–142, 1984.

Alex J. Smola and Peter L. Bartlett. Sparse greedy Gaussian process regression. In *NeurIPS*, pages 619–625, 2001.

Edward Snelson and Zoubin Ghahramani. Sparse Gaussian processes using pseudo-inputs. In *NeurIPS*, 2006.

Jasper Snoek, Hugo Larochelle, and Ryan Adams. Practical Bayesian optimization of machine learning algorithms. In *NeurIPS*, pages 2951–2959, 2012.

Michael L Stein. *Interpolation of spatial data: some theory for kriging*. Springer Science & Business Media, 2012.

William R. Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294, 1933.

Michalis K. Titsias. Variational learning of inducing variables in sparse Gaussian processes. In *AISTATS*, pages 567–574, 2009.

Lloyd N. Trefethen and David Bau III. *Numerical linear algebra*, volume 50. SIAM, 1997.

R. E. Turner and M. Sahani. Two problems with variational expectation maximisation for time-series models. In D. Barber, T. Cemgil, and S. Chiappa, editors, *Bayesian Time series models*, chapter 5, pages 109–130. Cambridge University Press, 2011.

Stephen Tyree, Kilian Q. Weinberger, Kunal Agrawal, and Jennifer Paykin. Parallel boosted regression trees for web search ranking. In *WWW*, pages 387–396, 2011.

Shashanka Ubaru, Jie Chen, and Yousef Saad. Fast estimation of $\mathrm{Tr}(f(\mathbf{A}))$ via stochastic Lanczos quadrature. *Journal on Matrix Analysis and Applications*, 38 (4):1075–1099, 2017.

Henk A. Van der Vorst. *Iterative Krylov methods for large linear systems*, volume 13. Cambridge University Press, 2003.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NeurIPS*, pages 5998–6008, 2017.

Zi Wang and Stefanie Jegelka. Max-value entropy search for efficient Bayesian optimization. In *ICML*, 2017.

Zi Wang, Chengtao Li, Stefanie Jegelka, and Pushmeet Kohli. Batched high-dimensional Bayesian optimization via structural kernel learning. In *ICML*, 2017.

Andrew Gordon Wilson. *Covariance kernels for fast automatic pattern discovery and extrapolation with Gaussian processes*. PhD thesis, University of Cambridge, 2014.

Andrew Gordon Wilson and Ryan Adams. Gaussian process kernels for pattern discovery and extrapolation. In *ICML*, pages 1067–1075, 2013.

Andrew Gordon Wilson and Hannes Nickisch. Kernel interpolation for scalable structured Gaussian processes (KISS-GP). In *ICML*, 2015.

Andrew Gordon Wilson, David A. Knowles, and Zoubin Ghahramani. Gaussian process regression networks. In *ICML*, 2012.

Andrew Gordon Wilson, Christoph Dann, and Hannes Nickisch. Thoughts on massively scalable Gaussian processes. *arXiv preprint arXiv:1511.01870*, 2015.

Andrew Gordon Wilson, Zhiting Hu, Ruslan Salakhutdinov, and Eric P. Xing. Deep kernel learning. In *AISTATS*, 2016a.

Andrew Gordon Wilson, Zhiting Hu, Ruslan R. Salakhutdinov, and Eric P. Xing. Stochastic variational deep kernel learning. In *NeurIPS*, 2016b.

James T. Wilson, Viacheslav Borovitskiy, Alexander Terenin, Peter Mostowsky, and Marc Deisenroth. Efficiently sampling functions from Gaussian process posteriors. In *ICML*, 2020.

Zhen-Hang Yang and Jing-Feng Tian. Convexity and monotonicity for elliptic integrals of the first kind and applications. *Applicable Analysis and Discrete Mathematics*, 13(1):240–260, 2019.

Zichao Yang, Andrew Wilson, Alex Smola, and Le Song. A la carte–learning fast kernels. In *AISTATS*, pages 1098–1106, 2015.