Distribution Agreement

Zhiqi Fu April 9, 2020

Fast Gaussian Process Solver

by

Zhiqi Fu

Yuanzhe Xi
Advisor

Department of Mathematics

Yuanzhe Xi
Advisor

Bree Ettinger
Committee Member

Benjamin Miller
Committee Member

Rho Seunghwa
Committee Member

2020

Fast Gaussian Process Solver

by

Zhiqi Fu

Yuanzhe Xi
Advisor

An abstract of
a thesis submitted to the Faculty of Emory College of Arts and Sciences
of Emory University in partial fulfillment
of the requirements of the degree of
Bachelor of Science with Honors

Department of Mathematics

2020

Abstract

Fast Gaussian Process Solver

By Zhiqi Fu

Gaussian Process is a non-parametric, stochastic machine learning technique that has been widely utilized in many fields, such as spatial statistics, geology and computer experiments. However, the application of this powerful technique is limited by its cubic computational complexity. To solve this problem, we studied the intrinsic structures inherent in the kernel matrix and developed linear-complexity solvers to quickly train the Gaussian process model. In this study, we worked on the following aspects:

- Combining stochastic Lanczos Quadrature with density of states, which is a concept developed in quantum physics, to develop a log-determinant solver
- Using the Sherman-Morrison-Woodbury preconditioned conjugate gradient to develop a linear system solver
- Substituting the naive grid search with Bayesian optimization to quickly train the optimal hyper-parameters

After the above steps, we applied it to the synthetic data set and global temperature data to compare our predictions with the true temperature values in order to validate our algorithm.

Fast Gaussian Process Solver

By

Zhiqi Fu

Yuanzhe Xi
Advisor

A thesis submitted to the Faculty of Emory College of Arts and Sciences
of Emory University in partial fulfillment
of the requirements of the degree of
Bachelor of Science with Honors

Department of Mathematics

2020

Acknowledgements

My thanks to Dr. Yuanzhe Xi, my advisor, who offers me this important opportunity to work with him and introducing me to the field of machine learning. Dr. Xi not only guides me patiently through my research process during Emory's REU and honors program, but also provides me precious suggestions which help me to determine my next goal after graduating from college. I would also like to thank Drs. Bree Ettinger, Rho Seunghwa, and Benjamin Miller for taking time to attend my honor thesis defense as committee members. Next, I would like to thank Emory honors program to support me along the way. Finally, I would thank my parents for supporting my studies at Emory University and letting me choose my own career path.

Contents

Chapter 1

Introduction

1.1 The Multivariate Gaussian Distribution

The multivariate Gaussian (normal) distribution has the density function:

$$p(x|\mu,\Sigma) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left((x-\mu)^T\Sigma^{-1}(x-\mu)\right) \tag{1.1}$$

where n is the dimension of x, μ and Σ denote the mean and covariance of x, respectively. In particular, Σ is always symmetric positive definite.

Figure 1.1 illustrates the density function from a two dimensional Gaussian distribution, where the mean and convariance matrix determine the center and the shape of the curve, respectively.

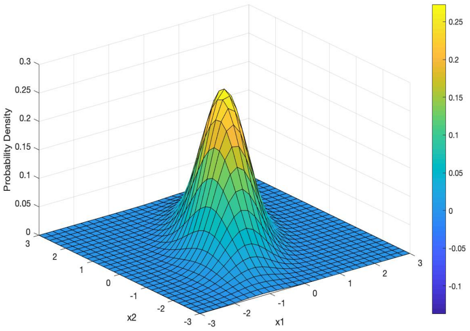

Figure 1.1: Density function of a multivariate Gaussian distribution.

1.2 Gaussian Process

Gaussian Process (GP) is a very powerful non-parametric machine learning technique. It searches the relationships among measured data for making predictions. It is defined as follows:

Definition 1 *A Gaussian Process is a stochastic process (a collection of random variables), such that every finite number of the random variables has a multivariate Gaussian distribution.*

A function f which is considered to be distributed as a GP is usually denoted as

$$f \sim GP(\mu, k),$$

where $\mu(x) : \mathbb{R}^d \to \mathbb{R}$ is a mean field and $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ is a symmetric and positive (semi)-definite covariance kernel. That is, for any set of locations $X = \{x_1, \ldots, x_n\} \subset \mathbb{R}^d$, $f_X \sim N(\mu_X, K_{XX})$, where f_X and μ_X represent the vectors of function values for f and μ evaluated at each $x_i \in X$, and $(K_{XX})_{ij} = k(x_i, x_j)$, respectively [1].

Let us first use a few one-dimensional examples to demonstrate the Gaussian Process. Here, we use the red line to denote the mean field and the grey area to represent the confidence interval of GP. Figure 1.2 illustrates the Gaussian Process Regression when we do not have any training points. In this situation, we will have a wide range of possible and diverse functions shapes on display.

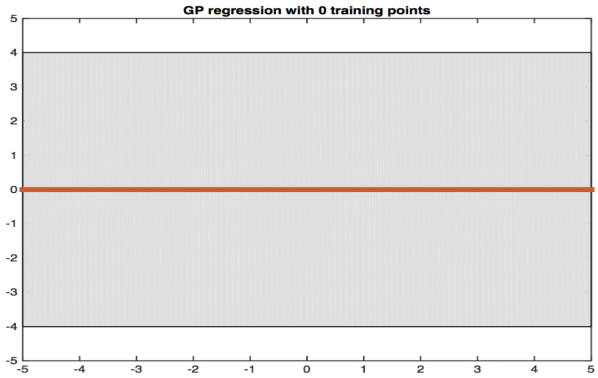

Figure 1.2: Gaussian Process Regression with 0 training points

The mean field and the covariance will be updated once more data points become available. Figure 1.3

shows the Gaussian Process Regression when we have 4 training points. The updated Gaussian process is constrained to the possible functions that fit the training data we introduced.

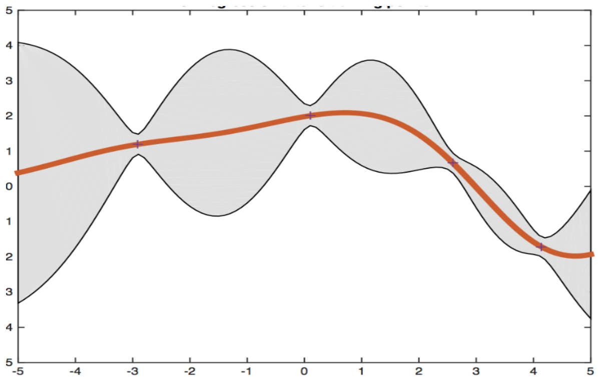

Figure 1.3: Gaussian Process Regression after 4 training points

As we introduce more and more training points, the confidence interval keeps shrinking. We will finally get a certain function with a pretty narrow confidence interval. In general, the mean field of GP works as the datum line. Normally, we will assume the mean function as 0 to simplify the calculation and thus, $y = 0$ is the datum line. Even when the mean is not zero, we could still assume zero mean and add the mean value back after we finished our predictions. Therefore, we will focus on defining the covariance/kernel function which determines the shape of the distribution and captures the relationship between points. After we are certain with the kernel function, we could then make predictions using those relationships.

In our study, and also in many previous works, people will split the data set into two parts, the training points and the testing points. With training points, people could determine the kernel function and with this kernel function, people will use testing points to make predictions.

Therefore, GP could be expressed in the following way:

$$\begin{bmatrix} f \\ f_* \end{bmatrix} \sim N(0, \begin{bmatrix} K(X,X) & K(X,X_*) \\ K(X_*,X) & K(X_*,X_*) \end{bmatrix}),$$

where X is the location of training points, $K(X_*,X)$ is the covariance matrix between the GP evaluated at X_* and X, and f is scalar observations for training points.

1.3 Motivations

GP has been widely used in many fields because it enjoys many advantages over other machine learning techniques:

- Gaussian process is a non-parametric model, which is defined in an infinite dimension Gaussian distribution f. Thus, as the number of data points increases, the amount of information the function f could get will also increase, which makes it more flexible.
- Gaussian process directly captures the model uncertainty and reveals the confidence of the prediction. As an example, in regression, GP directly gives a distribution for the prediction value, rather than simply one value as the prediction.
- When using GP, you can add prior knowledge and specifications about the shape of the model by selecting different kernel functions. For example, we could choose priors that make the function smooth, sparse, change drastically or differentiable.

On the other hand, the biggest disadvantage of GP is its expensive computational costs. The computational costs of many other methods, such as linear regression and neural network, are only related to the number of parameters. However, as GP is non-parametric, it needs to take into account all the training data each time they make a prediction. That is, as the number of training samples increases, the computational cost will increase cubically.

Before we go into more details and introduce our developed fast GP solver, we first give a brief introduction to the kernel functions associated with GP.

1.4 Kernel Function/Matrix

Recall that in order to set up the distribution in GP, we need to define kernel matrix Σ by evaluating the kernel function. Kernel function $k(x,y)$ describes the relationship between point x and point y. Popular kernels can be categorized into groups: stationary and non-stationary. Stationary kernel functions are used more often because their function values only depend on the distance $||x-y||$ between two points x and y.

By evaluating k at every possible pair of the data points, we get the positive definite *kernel matrix*:

$$K(\mathbf{x},\mathbf{y}) = \begin{bmatrix} k(x_1,y_1) & k(x_1,y_2) & \dots & \dots & k(x_1,y_n) \\ k(x_2,y_1) & k(x_2,y_2) & & & \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ & & & & k(x_{n-1},y_n) \\ k(x_n,y_1) & \dots & \dots & k(x_n,y_{n-1}) & k(x_n,y_n) \end{bmatrix}.$$

There are several common models for stationary kernels such as the Laplace kernel family and Matern kernel family. In this thesis, we will focus on the Gaussian kernel from Matern kernel family and show how to efficiently tune its hyper-parameters. The simplest Gaussian kernel has the form:

$$f(x,y) = se^{-\frac{||x-y||_2^2}{2\theta^2}}, \tag{1.2}$$

where s is the scaling coefficient and θ is the length scale of the kernel. If the kernel function equation (1.2) is used in GP, the kernel matrix is often denoted as:

$$K(\mathbf{x},\mathbf{y},\theta,s) = \begin{bmatrix} k(x_1,y_1,\theta,s) & k(x_1,y_2,\theta,s) & \dots & \dots & k(x_1,y_n,\theta,s) \\ k(x_2,y_1,\theta,s) & k(x_2,y_2,\theta,s) & & & \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ & & & & k(x_{n-1},y_n,\theta,s) \\ k(x_n,y_1,\theta,s) & \dots & \dots & k(x_n,y_{n-1},\theta,s) & k(x_n,y_n,\theta,s) \end{bmatrix}.$$

In order to get a well-fitted kernel function for a given dataset, we need to tune these hyper-parameters, which is the most time consuming part in the GP regression.

1.5 Log Marginal Function

Maximum Likelihood Estimation (MLE) is a popular approach used for parameter estimation in statistics. The principle of this approach is to choose the parameters that fit the data the most. Using this approach, we are able to find the optimal hyper-parameters by maximizing the associated log-likelihood functions. The formula of the log marginal function is:

$$L = \log p(z|X) = -\frac{1}{2}\log\det(K) - \frac{1}{2}(z-\mu)^T K^{-1}(z-\mu) - \frac{n}{2}log(2\pi) \tag{1.3}$$

where K is kernel matrix parameterized by hyper-parameters, z is target value and μ is the mean where we assumed to be zero. We use the term “marginal” to emphasize that we are dealing with a non-parametric model.

1.6 Application of Gaussian Process

In recent years, Gaussian processes have been applied in many domains, both for regression and classification tasks, such as spatio-temporal statistics, solving PDEs and etc,. Here, we will discuss a few representative applications.

1.6.1 Application in Finance/Portfolio

The portfolio is a major part of money management. However, it is extremely hard to choose the best portfolio, since we should not only consider risk and many other factors, but also original assets and profits. Thus, in this circumstance, people could apply the GP to find a minimal risk portfolio [2]. For example, if we know information about a portfolio, including the mean and the variance of the returns of the assets, we could using GP to select a optimal portfolio and help people to gain more profits.

1.6.2 Application in Robotic Techniques

Technological development has made robots more common. Many of those more common robots can walk autonomously because they can detect features of the ground in from of them. Therefore, they can go up and down over terrain and avoid obstructions. GP is a method that can help these autonomous robots model terrain. Figure 1.4 is an example from Wolfram Burgard’s presentation [3] which shows how GP is used to detect the stone block.

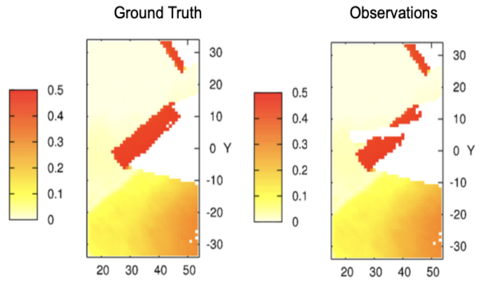

Figure 1.4: Stone Block Detection Using GP [3].

1.6.3 Application in Biosystem

GP is also widely used in biology fields. In K. Azmana, J. Kocijan's paper [4], they experiment a case study about lagoon of Venice. Since the lagoon is shallow, the development of the biomass is crucial, especially the excessive growth of algae. Thus, people want to build a model to predict the algae growth in the lagoon. Due to the lack of data related to the number of selected regressors, data corrupted with noise and measurement errors, and the need for the measure of model prediction confidence, GP Model is the model they decided to choose because they only need to choose covariance function in that case. Figure 1.5 is the resulting GP Regression they obtained.

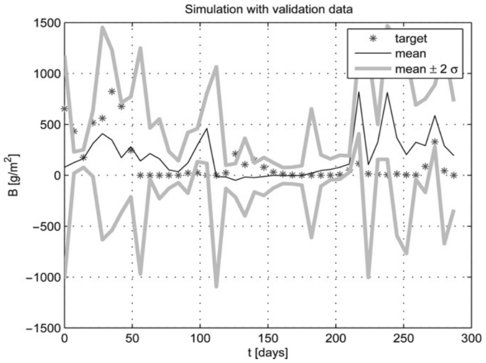

Figure 1.5: Simulation of the GP model with validation data [4].

1.6.4 Application in Weather Forecasts

GP could also help people to make predictions, especially help people to forecast weather. One example is the forecasting of wind power. Wind energy is a crucial resource that has been widely utilized around the world. However, the uncontrollable and variability of wind always make it difficult to forecast. To solve this problem, Mori, H., & Kurata, E applied GP to predict the wind power [5], which could help people solve problems such as maintenance of wind power and reservation of energy.

1.7 Difficulties and Contributions

The kernel function is at the heart of GP. However, the computational cost of GP is often quoted as $O(n^3)$, resulting from the need to compute the log-determinant of the $n \times n$ kernel matrix K and to solve a linear system associated with K.

In the past, several efficient algorithms have been proposed to tackle these computational bottlenecks, including those based on fast matrix-vector multiplication(MVMs) [1, 6], pivoted Choloesky preconditioning [1], hierarchically compositional kernel [7] and stochastic Lanczos Quadrature(SLQ) [8].

In this thesis, we develop an efficient linear complexity matrix-free approach which needs to access the kernel matrix implicitly through matrix-vector multiplications. In Chapter 2, we propose a density of states based algorithm for computing the logarithm of the determinant and bilinear form. In Chapter 3, we present a fast linear system solver based on the Sherman-Morrison-Woodbury preconditioned conjugate gradient algorithm. In Chapter 4, we substitute the naive grid search with Bayesian optimization to quickly train the hyper-parameters. In Chapter 5, we provide some experiments to validate our fast GP solver and draw some conclusions in Chapter 6.

Chapter 2

Log Determinant Solver

In this chapter, we will propose a fast algorithm for computing the logarithm of the determinant term in equation (1.3).

2.1 Matrix Functions

The logarithm of the determinant of a matrix A is defined based on its eigendecomposition. Every symmetric matrix A admits an eigendecomposition

$$A = V \Lambda V^T,$$

with its eigenvalues λ_i along the diagonal of $\Lambda = diag(\lambda_1, \ldots, \lambda_n)$ and the columns of V being the eigenvectors. Figure 2.1 shows the eigendecomposition of a symmetric matrix A.

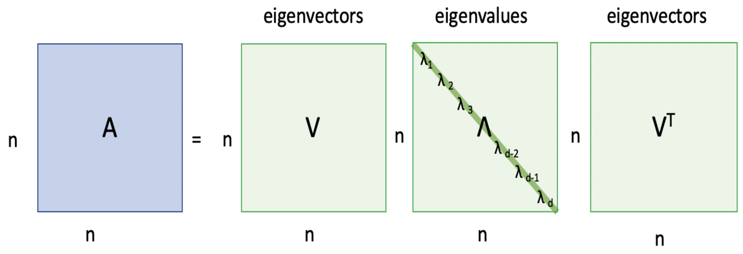

Figure 2.1: Eigendecomposition of a symmetric matrix A.

Based on the eigendecomposition, one can define various matrix functions. Suppose f is a scalar function $f : \mathbb{R} \to \mathbb{R}$, applying f on A is equivalent to applying f on each eigenvalue of A in its eigendecomposition:

$$A = V\Lambda V^T \to f(A) = Vf(\Lambda)V^T,$$

with $f(\Lambda) = diag(f(\lambda_1), \ldots, f(\lambda_n))$. See Figure 2.2 for an illustration.

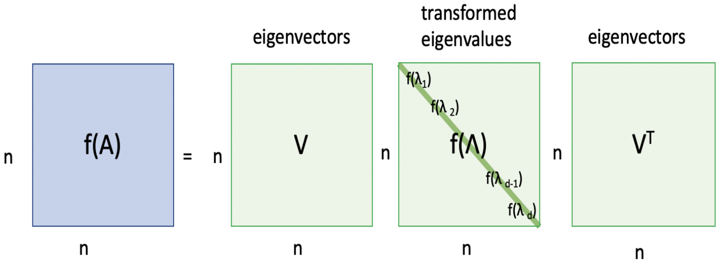

Figure 2.2: Application of f on a symmetric matrix A.

In Gaussian processes, we are interested in the logarithm function $f(x) = \log(x)$:

$$\log(K) = \log(Q\Lambda Q^T) = Q \begin{bmatrix} \log_{\lambda_1} & & & \\ & \log_{\lambda_2} & & \\ & & \ddots & \\ & & & \log_{\lambda_n} \end{bmatrix} Q^T.$$

This is because the logarithm of the determinant of matrix K is equal to the trace of the logarithm of the given matrix:

$$\log\det(K) = \log(\prod_{i=1}^{n} \lambda_i) = tr(\log(K)) = \sum_{i=1}^{n} \log(\lambda_i). \tag{2.1}$$

To estimate the trace of $\log(K)$, we could sum the log of all the eigenvalues of matrix K. Another popular method is based on Cholesky decomposition. Using the Cholesky decomposition $K = LL^T$, where L is a lower triangular matrix, we could obtain the log-determinant of K by $\log\det(K) = 2\sum_{i=1}^{n} \log(L_{ii})$. However, both methods are only applicable when the matrix size is small due to the cubic computational complexity. This computational bottleneck motivates us to study a fast algorithm to approximate the trace of $\log(K)$ for large matrices.

2.2 Stochastic Lanczos Quadrature

Stochastic Lanczos Quadrature is a method that combines the stochastic trace estimator with the Lanczos Quadrature method for estimating the trace of the inverse and the determinant of matrices [8]. This method can approximate the trace of $f(A)$ by only accessing A through matrix-vector multiplications.

2.2.1 Hutchinson's Method

Stochastic trace estimator was first proposed by Hutchinson in [9]:

Lemma 1 *Let A be an n × n symmetric matrix with trace(A)≠ 0. Let x be a random vector whose entries are i.i.d Rademacher random variables (Pr(x_i =± 1) = 1/2). x^TAx is an unbiased estimator of trace(A) i.e.,*

$$E(x^TAx) = trace(A) \tag{2.2}$$

and

$$Var(x^TAx) = 2(\|A\|_F^2 - \sum_{i=1}^{n} A_{ii}^2) \tag{2.3}$$

Later, people show that the method can estimate the $\mathrm{tr}(f(A))$ with random vectors x following either Rademacher distribution ($x_i = \pm 1$ with probability 0.5) or Gaussian distribution ($x_i \sim N(0,1)$) as long as the probe vector x has independent random entries with zero mean and unit variance.

Definition 2 *An Hutchinson trace estimator for a symmetric positive-definite matrix $A \in \mathbb{R}^{n\times n}$ is*

$$H_M = \frac{1}{M}\sum_{i=1}^{M} x_i^T A x_i,$$

where the z_i's are M independent random vectors whose entries are i.i.d Rademacher random variables.

Thus, a Hutchinson trace estimator can be used directly to approximate $\log\det(A)$:

$$\log\det(A) = \mathrm{tr}(\log(A)) = \sum \log(\lambda_i) = \mathbb{E}(x_i^T \log(A) x_i) \approx \frac{1}{n_v}\sum_{i=1}^{n_v} x_i^T \log(A) x_i. \tag{2.4}$$

As a result, the computation of $\log\det(A)$ boils down to computing $x_i^T \log(A)x_i$.

2.2.2 Lanczos Algorithm

Lanczos Quadrature is a method designed for computing matrix bilinear forms. It is based on the famous Lanczos algorithm. For a given real symmetric matrix $A \in \mathbb{R}^{n\times n}$ and a starting vector v_1 of unit 2-norm, the Lanczos Algoirthm generates an orthornormal basis V_m for the Krylov subspace $\mathrm{Span}\{v_1, Av_1, \ldots, Av_1^{m-1}\}$ such that $V_m^T A V_m = T_M$, where T_m is a $m \times m$ tridiagonal matrix and is useful for approximating eigenvalues and eigenvectors of A.

2.2.3 Lanczos Quadrature

Lanczos Quadrature uses the information from the Lanczos decomposition to approximate $x_i^T \log(A)x_i$:

$$\begin{aligned} x_i^T \log(A)x_i &\approx x_i^T V_m \log(T_m) V_m^T x_i \\ &= \beta e_1^T \log(T_m) \beta e_1 \\ &= \beta^2 e_1^T Q \log(\Lambda) Q^T e_1 \\ &= \beta^2 Q(1,:) \log(\Lambda) Q^T(1,:) \\ &= \sum \beta^2 Q(1,i)^2 \log(\lambda_i), \end{aligned}$$

where β is the 2-norm of the Rademacher random vector x_i, the columns of Q are the eigenvectors of matrix T_m and the diagonal entries of Λ are the eigenvalues of matrix T_m. Here we assume $v_1 = x_i/\|x_i\|_2$.

Repeating the above operation for n_v random vectors x_i leads to an approximation to $\log\det(A)$:

$$\log\det(A) \approx \frac{1}{n_v} \sum_{i=1}^{n_v} \sum_{j=1}^{m} \beta^2 Q^{(i)}(1,j)^2 \log(\lambda_j^{(i)}). \tag{2.5}$$

The Stochastic Lanczos Quadrature (SLQ) algorithm for approximating the $\log\det(A)$ is summarized in the following algorithm [8].

Algorithm 1 Trace estimation by SLQ

```
Input: SPD matrix $A \in \mathbb{R}^{n\times n}$, function $f$, degree $m$ and $n_v$.
Output: Approximate trace of $f(A)$.
for l = 1 to $n_v$ do
    Generate a Rademacher random vector $u_l$ and form a unit vector $v_l = u_l/\sqrt{n}$
    $T = Lanczos(A, v_l, m)$; that is, apply $m$ steps of Lanczos algorithm on $A$ with $v_l$ as the starting vector
    $[Y, \Theta] = eig(T)$ and compute $\tau_k = [e_1^T y_k]$ for $k = 1, ..., m$
    $\Gamma \leftarrow \Gamma + \sum_{k=1}^{m} \tau_k^2 f(\theta_k)$ , where $\theta_k$'s are eigenvalues of $T_m$
end for
Output $\Gamma = \frac{n}{n_v}\Gamma$.
```

The approximation error of Algorithm 1 has been studied in the following theorem [8]:

Theorem 1 *Consider a symmetric positive definite matrix $A \in \mathbb{R}^{n\times n}$ with eigenvalues in $[\lambda_{min}, \lambda_{max}]$ and condition number $k = \frac{\lambda_{max}}{\lambda_{min}}$, and let f be a function that is analytic and either positive or negative (does not cross zero) inside this interval, and whose absolute maximum and minimum values in the interval are M_ρ and m_ρ, respectively. Let ε, η be constants in the interval (0,1). Then for SLQ parameters satisfying:*

- $m \leq \frac{\sqrt{k}}{4}\log\frac{K}{\varepsilon}$ *number of Lanczos steps, and*
- $n_v \leq \frac{24}{\varepsilon^2}\log(\frac{2}{\eta})$ *number of starting Rademacher vectors,*

where $K = \frac{(\lambda_{max}-\lambda_{min})(\sqrt{k}-1)^2 M_\rho}{\sqrt{k}m_\rho}$, the output Γ of the Stochastic Lanczos Quadrature method satisfies:

$$Pr[|tr(f(A)) - \Gamma| \leq \varepsilon|tr(f(A))|] \leq 1-\eta. \tag{2.6}$$

Although SLQ can provide very accurate approximations, it usually requires a sufficiently large number of Lanczos steps m. Since the cost of SLQ is $O((nnz(A)m + nm^2)n_v)$, the computational cost of SLQ increases quadratically with respective to m. In the next section, we will show how to use Density Of States (DOS) to alleviate this issue.

2.3 Density of States

Given a $n \times n$ real symmetric matrix A, scientists in various disciplines often want to compute its density of states (DOS), also known as spectral density. DOS is a probability distribution function which describes the probability of finding eigenvalues of a matrix in a given interval near t. Being a distribution, the spectral density of a matrix can be written as a sum of Dirac δ-functions centered at eigenvalues [8]:

$$\phi(t) = \frac{1}{n} \sum \delta(t - \lambda_i), \tag{2.7}$$

where δ is the Dirac δ-function or Dirac distribution and λ_j's are the eigenvalues of A.

There are many applications of density of states. One of the major uses is that we could count the number of eigenvalues in a certain interval $[a,b]$ using DOS [10]:

$$v[a,b] = \int_a^b \sum_j \delta(t - \lambda_j) dt \equiv n \int_a^b \phi(t) dt. \tag{2.8}$$

Similarly, one could approximate log determinant of A as follows:

$$\log\det(A) = \mathrm{tr}(\log(A)) = \sum_{i=1}^{n} \log(\lambda_i) = n \int_{\lambda_{min}}^{\lambda_{max}} \log(t)\phi(t)\, dt. \tag{2.9}$$

To estimate $\phi(t)$, the most straightforward way is to compute all the eigenvalues of matrix A. However, the computation of its entire spectrum is prohibitively expense. Therefore, in our study, we choose to approximate a smoothed version of $\phi(t)$ [11]:

$$\phi(t) \approx \phi_\sigma(t) = \frac{1}{n} \sum_{j=1}^{n} h_\sigma(t - \lambda_j), \tag{2.10}$$

where $h_\sigma(t)$ could be any function that has a peak at zero and has area equal to one below the curve. Here, we choose Gaussian as our function:

$$h_\sigma(t) = \frac{1}{(2\pi\sigma^2)^{1/2}} e^{-\frac{t^2}{2\sigma^2}} \tag{2.11}$$

In practice, we find that $\sigma = \frac{h}{2\sqrt{2\log(\kappa)}}$ works well for many problems, where h is the resolution and κ is a parameter greater than 1.

2.3.1 Limitations of DOS algorithm

A naive application of DOS algorithm does not always yield satisfactory results. This is due to the flexibility of θ used to define kernel functions. In this section, we first use a few numerical experiments to demonstrate this issue.

For example, when the hyper-parameter θ is set to 100, there are relatively few large eigenvalues and most eigenvalues are gathered around 0. As we decrease θ to 10, we could see that the number of large eigenvalues increases. This is a hard case for DOS because the matrix becomes more diagonally dominant and thus, reduce the singularity of the matrix. Figure 2.3 and Figure 2.4 illustrate the eigenvalues of the kernel matrix when the hyper-parameter θ is to 100 and 10, respectively.

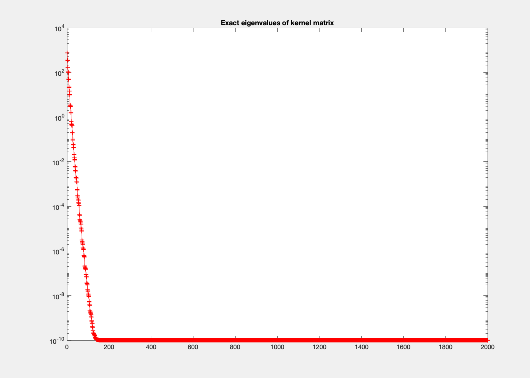

Figure 2.3: Eigenvalues for kernel matrix when hyper-parameter = 100.

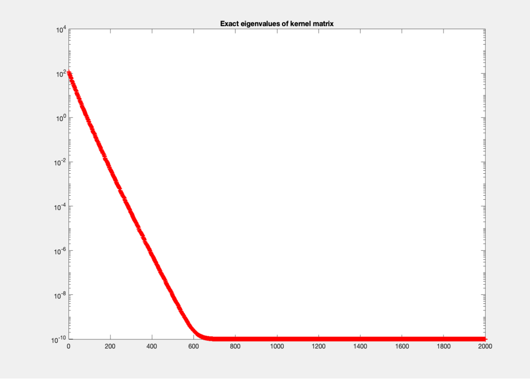

Figure 2.4: Eigenvalues for kernel matrix when hyper-parameter = 10.

In our study, we normally have to tune the hyper-parameter θ ranging from 0 to 100.

Since most of the eigenvalues are extremely small and are clustered around a small value (10^{-10}), even they have subtle difference, after we take the log of those small eigenvalues, the differences will be amplified. Therefore, SLQ/DOS algorithms need to approximate those small eigenvalues accurately enough if an accurate estimation of $\log\det(A)$ is required. Unfortunately, both SLQ and DOS algorithms have difficulties for approximating small eigenvalues.

Let us use a few examples to show this limitation associated with the DOS algorithm. Figure 2.5 plots both the exact density of states and our estimated DOS. The red line is the exact density of states, while the blue line indicates the estimated DOS with 50 Lanczos step.

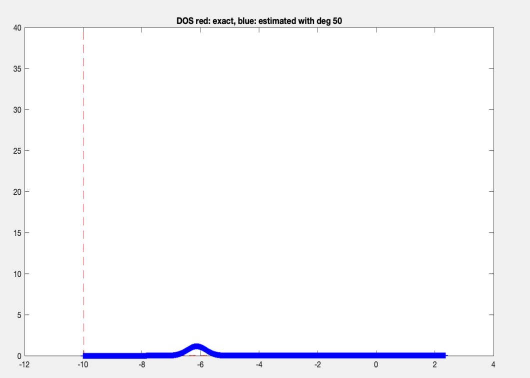

Figure 2.5: Estimated DOS (blue) with 50 Lanczos steps.

It is easy to see that most of the eigenvalues are clustered around 10^{-10}. However, the estimated DOS predicts the cluster of eigenvalues around 10^{-6}. One simple remedy to fix this is to dramatically increase the Lanczos steps. Figure 2.6 shows the estimated DOS when we increase the Lanczos steps to 300.

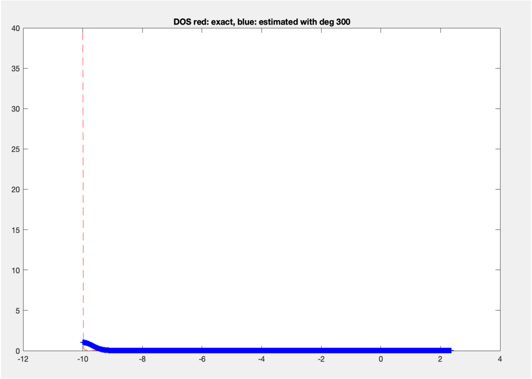

Figure 2.6: Estimated DOS (blue) with 300 Lanczos steps.

After we increase the Lanczos steps, our estimated DOS becomes much more accurate. However, increasing the Lanczos steps significantly increase the computational costs and storage, which is not what we want. In the next section, we will propose a modified DOS algorithm which can achieve high accuracy with relatively small Lanczos steps.

2.4 Modified DOS

The modified DOS algorithm works as follows. First, we divide the eigenvalues into two subintervals: one for large eigenvalues and one for small eigenvalues. Since the DOS estimation of large eigenvalues is quite accurate, we keep this part unchanged and compute the number of eigenvalues located in this interval. For the DOS estimation corresponding to small eigenvalues, we estimate the number of eigenvalues in this interval from the difference between the size of the matrix and the estimated number of large eigenvalues. Then, we assume the logarithm of all the small eigenvalues are centered at one point $\log(\lambda_{min})$ and give a density at this point based on the estimated number of small eigenvalues. This process can be described by the following equation:

$$\begin{aligned} tr(\log(A)) &= n\int_{\lambda_{min}}^{10\lambda_{min}} f(x)\phi(x)dx + n\int_{10\lambda_{min}}^{\lambda_{max}} f(x)\phi(x)dx \\ &\approx (n - n\int_{10\lambda_{min}}^{\lambda_{max}} f(x)\phi(x)dx) \times \log(\lambda_{min}) + n\int_{10\lambda_{min}}^{\lambda_{max}} f(x)\phi(x)dx. \end{aligned}$$

This modified DOS can quickly capture the cluster of small eigenvalues and thus improve the approximation accuracy of the estimated $\log\det(A)$. Figure 2.7 shows the estimated DOS from the standard DOS algorithm where the matrix has the size of $4,000 \times 4,000$, hyper-parameter θ is 20 and Lanczos step is 7. The relative error of the estimated $\log\det(A)$ is 0.725 due to the failure to capture the right cluster around 10^{-6}.

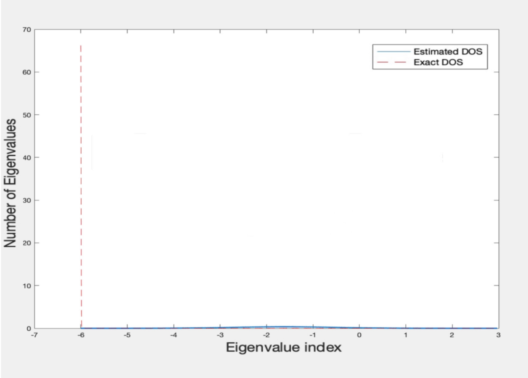

Figure 2.7: DOS, Relative Error:0.725

With the same matrix size, hyper-parameter and Lanczos step, we apply our modified DOS and generate Figure 2.8:

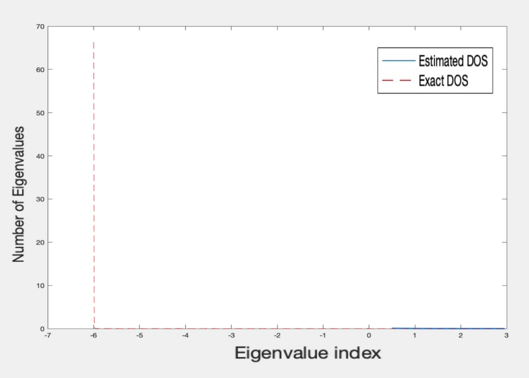

Figure 2.8: Modified DOS, Relative Error:0.007

For the same test problem, the modified DOS could reduce the relative error to 0.007 with only 7 Lanczos steps.

In the above equation and also in this example, we choose $10\lambda_{min}$ as the cut-off point of small eigenvalues and large eigenvalues. However, instead of choosing the cut-off point manually, we can automatically find an optimal cut-off point.

2.4.1 Truncation of DOS

To decide where to truncate DOS, we need to remove the bulk of volume using finite differences to compute where the curve tapers off. We take xdos which records the x-coordinates of the points used to discretize $[\lambda_{min}, \lambda_{max}]$ and ydos which records the function values of the estimated DOS at xdos locations. We first find the peak of the curve via the maximum ydos value. Then, we start from this point and calculate the slope magnitude. Finally, we find our cut-off point which is where the curve tapers off and is to the right of the maximum point.

The truncation procedure is sketched in Algorithm 2.

Algorithm 2 truncateDOS

```
Input: xdos, ydos
Output: index = x-index of truncation point
argmax = max(ydos);
preallocate array to store differences
for the next index after argmax to the last index do
    calculate the slope
end for
Find the index where the slope magnitude is maximal(argmax2)
for each slope index i in the array do
    if the slope at i and i+1 $<$ tolerance value, and i $>$ argmax2  then
        increment index of maximum slope by i
    end if
end for
```

2.4.2 Comparison

To validate the efficiency of the proposed log-determinant solver, we compare its performance with that of stochastic Lanczos Quadrature:

Table 2.1: n = $5,000$, s = 99, sigma = $1e-6$

		modified DOS		SLQ	
k	k_hut	error	time	error	time
50	1	2.475	0.54	37.91	0.51
100	1	1.678	1.23	18.30	1.22
150	1	0.997	1.91	5.504	1.88
200	1	0.897	2.75	0.815	2.73
250	1	0.372	3.83	0.266	3.78
300	1	0.121	4.94	0.158	4.90
350	1	0.027	6.04	0.244	6.23

From this table, we could see that modified DOS and SLQ take roughly the same amount of time because they perform the same number of Lanczos iterations. However, we could say that DOS is more accurate in most cases. On the other hand, if you want both algorithms to reach the same accuracy, SLQ will definitely take a longer time.

Chapter 3

Linear System Solver

3.1 Conjugate Gradient (CG) Algorithm

Conjugate Gradient is an iterative method for solving large linear systems $Ax = b$, when A is symmetric positive definite. Since the covariance matrix in GP is SPD, we decide to develop a fast linear system solver based on some variants of the preconditioned CG method.

3.1.1 Convergence of CG Algorithm

The A-norm of a vector x is defined as:

$$\|x\|_A = (Ax, x)^{1/2}.$$

The following lemma indicates the approximation obtained from the CG Algorithm.

Lemma 2 *Let x_m be the approximate solution obtained from the m-th step CG algorithm, and let $d_m = x_* - x_m$ where x_* is the exact solution. Then, x_m is of the form*

$$x_m = x_0 + q_m(A)r_0$$

where q_m is the polynomial of degree $m-1$ such that

$$\|(I - Aq_m(A))d_0\|_A = \min_{q \in \mathbb{P}_{m-1}} \|(I - Aq(A)d_0\|_A$$

.

From the above lemma, people have proved the convergence rate of CG algorithm.

Theorem 2 *Let x_m be the approximate solution obtained at the m-th step of the Conjugate Gradient algorithm, and x_* be the exact solution. Define*

$$\eta = \frac{\lambda_{min}}{\lambda_{max} - \lambda_{min}}. \tag{3.1}$$

Then,

$$||x_* - x_m||_A \leq \frac{||x_* - x_0||_A}{C_m(1+2\eta)}, \tag{3.2}$$

in which C_m is the Chebyshev polynomial of degree m of the first kind.

Since $C_m(t) \geq \frac{1}{2}(1+2\eta+2\sqrt{\eta(\eta+1)})^m = \frac{\sqrt{\lambda_{max}}+\sqrt{\lambda_{min}}}{\sqrt{\lambda_{max}}-\sqrt{\lambda_{min}}} = \frac{\sqrt{\kappa}+1}{\sqrt{\kappa}-1}$ [12], in which κ is the spectral condition number $\kappa = \lambda_{max}/\lambda_{min}$. Therefore, the bound in (3.2) can be rewritten as

$$||x_* - x_m||_A \leq 2[\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}]^m ||x_* - x_0||_A.$$

This theorem shows that the convergence rate of CG can be extremely slow if A is ill-conditioned.

3.2 Preconditioned Conjugate Gradient

In most cases, preconditioning is crucial to ensure a fast convergence of CG. If a good preconditioner $M \in \mathbb{R}^{n \times n}$ can be found for A, then Preconditioned CG (PCG) can converge much faster. Theoretically, a relative error ε can be obtained within $\frac{1}{2}\sqrt{\kappa(M^{-1}A)}\ln\frac{2}{\varepsilon}+1$ number of PCG iterations, where $\kappa(M^{-1}A)$ is now the condition number of the preconditioned matrix $M^{-1}A$ [13].

With a SPD preconditioner M, there are three ways to implement PCG algorithm [12]:

- The preconditioner can be applied on the left hand side:

$$M^{-1}Ax = M^{-1}b; \tag{3.3}$$

- It could also be applied on the right hand side:

$$AM^{-1}u = b, x \equiv M^{-1}u; \tag{3.4}$$

- When the preconditioner is available in the factored form

$$M = M_L M_R$$

  where, typically M_L and M_R are triangular matrices. In this situation, the preconditioning can be split:

$$M_L^{-1} A M_R^{-1} u = M_L^{-1} b, x \equiv M_R^{-1} u. \tag{3.5}$$

In this thesis, we adopt the first approach to precondition the linear system. In the next section, we will introduce an approximate inverse type preconditioner.

3.3 SMW Preconditioned Conjugate Gradient

The proposed preconditioner is based on the Sherman-Morrison-Woodbury (SMW) formula. This formula provides a way to quickly compute the inverse of a rank k update of a matrix $A \in \mathbb{R}^{n\times n}$.

Lemma 3 *(Sherman-Morrison-Woodbury formula (SMW-formula)) Let $A \in \mathbb{R}^{n\times n}$, and $U, V \in \mathbb{R}^{n\times k}$ be two matrices such that both A and $C^{-1} + VA^{-1}U$ are nonsingular. Then $A + UCV$ is nonsingular and it holds that:*

$$M^{-1} = (A+UCV)^{-1} = A^{-1} - A^{-1}U(C^{-1} + VA^{-1}U)^{-1}VA^{-1}. \tag{3.6}$$

Recall that the Lanczos algorithm returns a tridiagonal matrix T_m and a matrix V_m which has orthonormal columns such that $T_m = V_m^T A V_m$. By calculating the eigenpairs of matrix T_m, we have two matrices Q and Λ where the columns of Q are eigenvectors of T_m and entries of Λ are the eigenvalues of T_m.

In order to apply Lemma 3 to develop our preconditioner, we treat A as a diagonal matrix σI, U as Q, C as Λ and V as Q^T. The application of M^{-1} on a vector x is described in Algorithm 3:

Algorithm 3 Application of SMW preconditioner on a vector x

```
Input: $V$, $\Lambda$, $\sigma$, x
Output: y
$M = inv(\Lambda^{-1} + \frac{1}{\sigma}I)$
tmp1 = $A^{-1}x = (\sigma I)^{-1}x = \frac{x}{\sigma}$
tmp2 = $Q^T \times$ tmp1
tmp3 = $M \times$ tmp2
tmp4 = $Q \times$ tmp3
$y = \text{tmp1} - \frac{1}{\sigma}\text{tmp2}$
```

With the propsed SMW-preconditioner, we can quickly compute the second term in equation (1.3) $\frac{1}{2}z^T K^{-1} z$.

3.4 Comparison

To validate the performance of SMW PCG algorithm, we compare its performance with two other precontitioners: naive inverse pcg, which directly using the inv() function in Matlab to form the preconditioner, and unpreconditioned conjugate gradient, which does not utilize any preconditioner.

Table 3.1 and Table 3.2 illustrate running time, number of iterations used until convergence and relative error for CG algorithm with three preconditioners when the matrix size is $4,000 \times 4,000$ and $10,000 \times 10,000$.

Table 3.1: n = 4,000, s = 100, Mdeg = 100

	SMW PCG			Naive Inverse PCG			Unpreconditioned CG		
sigma	time	iter	res	time	iter	res	time	iter	res
1e-6	0.755	136	9.2e-5	5.379	300+	0.002	1.392	223	6.9e-5
1e-5	0.374	67	8.8e-5	46.36	257	9.7e-5	1.111	223	6.9e-5
1e-4	0.223	41	9.2e-5	34.79	132	8.5e-5	1.075	223	6.9e-5
0.001	0.239	41	9.2e-5	28.74	69	8.1e-5	1.094	223	6.9e-5
0.01	0.239	41	9.2e-5	29.75	41	9.2e-5	1.166	223	6.9e-5
0.1	0.238	41	9.2e-5	26.29	43	6.5e-5	1.148	223	6.9e-5

Table 3.2: n = 10,000, s = 100, Mdeg = 100

	SMW PCG			Naive Inverse PCG			Unpreconditioned CG		
sigma	time	iter	res	time	iter	res	time	iter	res
1e-6	5.051	174	8.4e-5	49.76	300+	0.0127	4.600	156	8.7e-5
1e-5	3.209	87	7.8e-5	46.36	295	0.0001	4.510	156	8.7e-5
1e-4	1.947	44	9.7e-5	34.79	163	9.0e-5	4.367	156	8.7e-5
0.001	2.042	44	8.9e-5	28.74	86	9.0e-5	4.464	156	8.7e-5
0.01	1.591	44	8.9e-5	29.75	44	8.9e-5	4.448	156	8.7e-5
0.1	1.458	44	8.9e-5	26.29	38	9.5e-5	4.393	156	8.7e-5
1	1.419	44	8.9e-5	26.66	32	9.3e-5	4.615	156	8.7e-5
5	1.430	44	8.9e-5	35.19	43	9.6e-5	4.459	156	8.7e-5

It is easy see that SMW PCG performs the best. Taking Table 3.2 as an example. When sigma is $1e-4$, SMW PCG converges within only 44 iterations and takes 1.95 seconds to converge. However, the naive inverse PCG takes 163 iterations to converge, which takes 37.79 seconds. For unpreconditioned conjugate gradient, although it converges much faster than naive inverse PCG, it is still slower than SMW PCG and takes many more iterations to converge.

Even when the naive inverse PCG takes the same number of iterations to converge as the SMW PCG takes, it takes much longer time to converge since its computational cost is much higher at each iteration.

Chapter 4

Hyper-parameter Training

In previous chapters, we introduce a fast GP solver which could help us calculate the log-likelihood function value. However, this only improves the efficiency of calculating the log-likelihood. In order to train the optimal hyper-parameters, we need to try with multiple hyper-parameters. To simplify the discussion in this section, we fix the parameter s and only show how to train the parameter θ.

4.1 Naive Grid Search

When we first start our study, we choose to use the naive grid search. We provide a boundary for θ and try with a bunch of numbers inside the interval. This could be seen in the following synthetic data experiment.

4.1.1 Synthetic Data Experiment

After we develop the log determinant solver and the linear system solver, we tried to apply our solvers on the synthetic data. We set a 150×150 grid size and $\theta = 40$ as the true hyper-parameter. We also provide a search range for the hyper-parameter which is $[10, 100]$. Our goal is to tune the optimal hyper-parameter θ.

To have a direct visualization, we scale the value to generate figures for this example. Figure 4.1 is the Gaussian Process random field we generated:

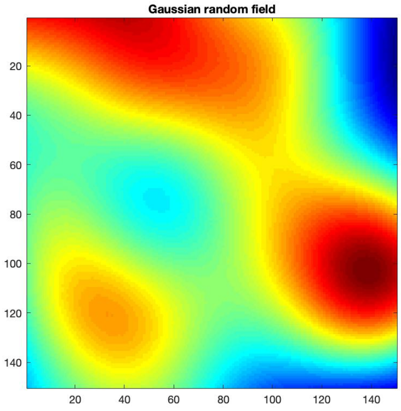

Figure 4.1: Synthetic dataset.

After generating the GP sample on the grid, we randomly choose 20 percent of data points as our training data. Figure 4.2 illustrates the training data, which are non-white pixels, for hyper-parameter estimation.

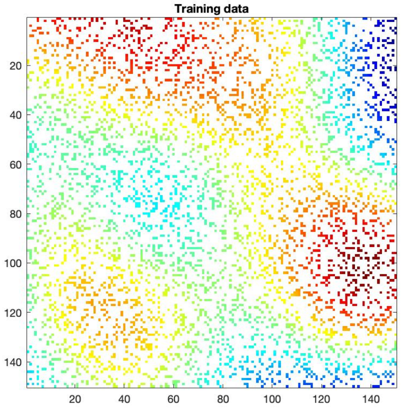

Figure 4.2: 20 percent data points from synthetic data (training data).

Using those 20 percent training data, we try with different hyper-parameters and compute the log-likelihood function. By plotting the log-likelihood function value at each point, we generate a log-likelihood curve which is shown in Figure 4.3. The horizontal axis denotes the length-scale parameter θ.

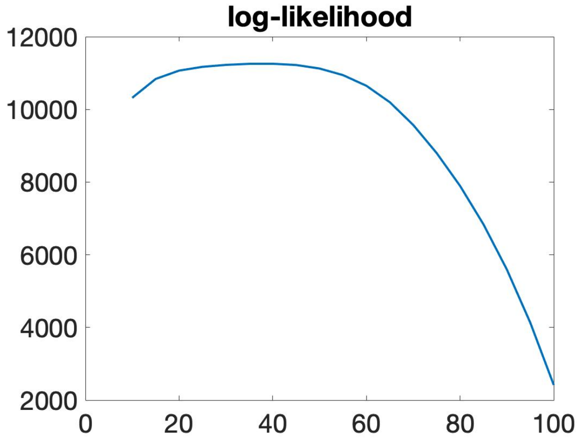

Figure 4.3: Computed log-likelihood function values where the true hyper-parameter $\theta = 40$.

By selecting the one which leads to the maximum log-likelihood, we obtain the optimal hyper-parameter which is 40. We could also see it from graph: when the parameter value is 40, the log-likelihood function reaches the maximum point. Finally, we use the estimated parameters to perform predictions and fill the missing data(test points). Figure 4.4 illustrates the prediction by using the estimated hyper-parameter.

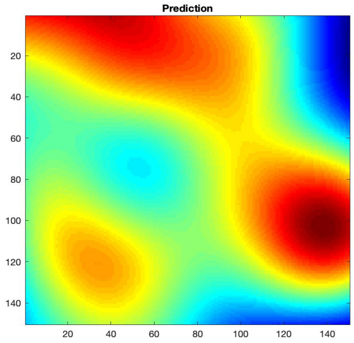

Figure 4.4: Recovered GP sample (training point+prediction).

We basically completely recover the original graph. Also, by calculating the relative error between the predicted target value and the true target value, we validate that our algorithm could achieve 3 digits of accuracy, which is around 0.0035.

Then, we try with different true hyper-parameter values and also get very precise results.

Table 4.1: n = $22,500$, k_{lan} = 100, sigma = $1e-3$

Setting		Result	
True parameter	Initial Guess (interval)	Estimated parameter	Relative Error
20	[10,100]	20	0.0050
40	[10,100]	40	0.0035
60	[10,100]	60	0.0028

In this experiment, we choose to try 10 grid points between 10 to 100 and find the optimal hyper-parameter successfully. In real cases, however, it is almost impossible for us to choose an interval that includes the optimal hyper-parameter. We could indeed set a wide range and the more numbers we try, the more accurate our result will be in general. Nevertheless, it is extremely expensive to search at each point, especially when we need to train two hyper-parameters jointly. Therefore, we want to exploit Bayesian Optimization to help us tune the hyper-parameters.

4.2 Bayesian Optimization

Bayesian Optimization is an approach that could help us find the maximum point of a continuous objective function f(x). Normally, we will choose to use Bayesian Optimization to approximate the maximum point when the input x is in $\mathbb{R}^d$ where $d \leq 20$ and the evaluations of f are expensive or maybe perturbed.

In order to use Bayesian statistical model, we first have to choose some prior measure over the space of possible objective function values. Then we need to combine prior and the likelihood to get a posterior measure over the objective given some observations. Then, according to the acquisition function, we need to decide where to take the next evaluation. To pick the hyperparameters of the next experiment, one can optimize the expected improvement(EI) over the current best result or the Gaussian process upper confidence bound (UCB).

In our study, we first construct log-likelihood function with two hyper-parameters as our objective function f. We also set an array called hypercube which contains the boundaries for two hyper-parameters. Then, we construct a few sample points based on the objective function f and returned a 2 by *num_rand* (the number of random points) cell array filled with random points uniformly sampled from hypercube. The BO algorithm is illustrated in the following algorithm.

Algorithm 4 Bayesian Optimization

```
Input: f, hypercube, num_rand, ell, sigma, num_iter, num_search
Output: minimum and minimizer of objective function f
Constructed a few sample points based on the objective function f
Create a discrete grid in hypercube
Find the cur_min and arg_min of samples points
running covariance matrix of samples points to get statistical model K
for i = 1, ..., number of iterations do
    calculate mean and covariance of all points
    for j = 1, ..., length of all points do
        calculate mean and covariance based on Gaussian process
    end for
    set Upper Confidence Bound(UCB) Acquisition Function: mean - $\kappa \times$ covariance
    Optimize the acquisition function
    Objective function evaluation: obj = f(arg_min)
    if obj $<$ cur_min then
        updates arg_cur_min and cur_min
    end if
    Augment data(prev_inputs, prev_labels) and update statistical model (k)
end for
```

4.3 Comparison

To validate Bayesian optimization is better than the naive grid search, we utilize both methods on the synthetic dataset. We change the grid size and record their running times of two methods. From Table 4.2, we notice that when the number of data points is not that large, there is no huge difference between those two methods. However, when the grid size becomes larger and we get many more data points in our analysis, the differences become larger and larger. When the grid size is 750×750, the running time of Bayesian

Optimization is only half of the running time of Naive Grid Search.

Table 4.2: Running time of two hyper-parameter tuning methods

	Grid Size					
	250×250	350×350	450×450	550×550	650×650	750×750
Naive Grid Search (sec)	28.08	58.28	98.21	150.7	210.6	278.9
Bayesian Optimization (sec)	12.33	25.05	42.60	64.33	89.57	120.3

We could also see the trend directly from Figure 4.5:

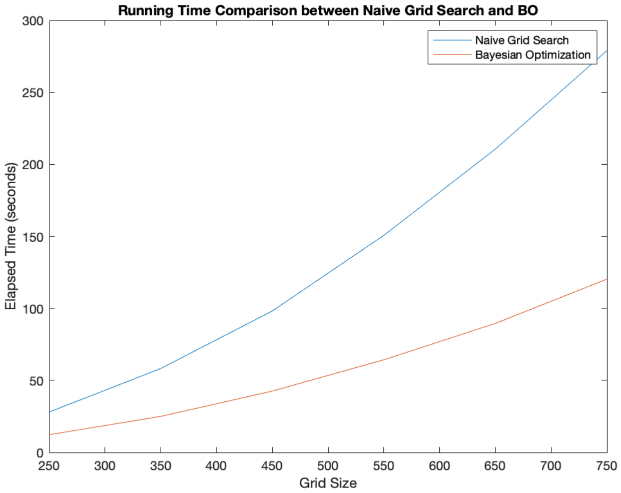

Figure 4.5: Running Time Comparison between Naive Grid Search and BO

From the above comparison, we could see that when the grid size is large, BO performs much better than the naive grid search. What's more, the above experiment utilizes a synthetic dataset which means we know the true hyper-parameters, so we set fewer search steps and could still get the true hyper-parameter. However, when we apply the algorithm to the real dataset and do not have any information about the value of optimal hyper-parameter, the naive grid search will take many more steps to find the hyper-parameter than Bayesian optimization and thus, have much higher computational cost.

Chapter 5

Experiment Outline

After we complete our fast Gaussian process solver, we apply it to the real dataset to validate it on some real-world problems.

5.1 Data Description

We apply our approach to the National Centers for Environmental Prediction Climate Forecast System Reanalysis (CFSR) data released by the National Oceanic and Atmospheric Administration (NOAA) [7]. We choose the hourly and monthly mean temperature data, over a period of eight years, from 2011 to 2019, but only include July at 550 mbar isobaric surface. The grid is 0.5-deg $\times$ 0.5-deg from 0E to 359.5E and 90N to 90S and has 720 different values for longitude, 361 different values for latitude. Therefore, the dataset is of size $720 \times 361 \times 1 \times 8$.

5.2 Dataset Resize

For this study, we compress the dimension to two by calculating the average temperature for each grid points over 8 years, which results in $259,920$ observations with two variables: longitude and latitude (720×361 Longitude/Latitude).

Also, instead of directly working on the temperature, we focus on the demeaned temperature which helps us to see the global temperature changes trend better. In our study, we choose 2011 as the year we want to work on. Therefore, we subtract the average temperature of all eight years from the temperature in 2011 at each grid point and worked on the demeaned temperature in the rest of our analysis.

5.3 Dataset Snapshot

To have a more direct view of the global temperature data, we create the dataset snapshot. Figure 5.1 shows the global temperature at 550 mbar in July, 2011.

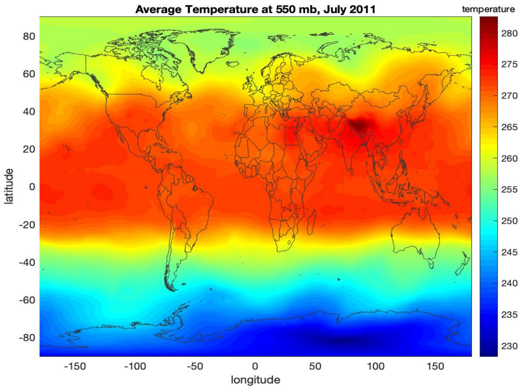

Figure 5.1: Global Temperature at 550 mbar, July 2011

We could see an extremely usual phenomenon that the area near the equator has the highest temperature, while as it gets closer and closer to the Arctic and Antarctic, the temperature becomes colder and colder. Figure 5.2 shows the resulting data after the subtraction of pixelwise mean for July over 8 years at 550 mbar in July, 2011.

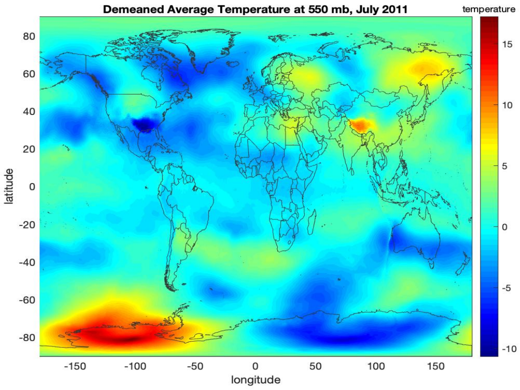

Figure 5.2: Demeaned Global Temperature at 550 mbar, July 2011

From the above graph, we could see an obvious global warming trend. For most areas, the temperature in 2011 is much lower than the average temperature from 2011-2019. It is consistent with common sense that the global temperature has been gradually increased. However, we could also see that there are some areas that have a higher temperature in 2011. From the graph, we could conjure it is Tibet in China and the southern part of the Pacific ocean.

However, in most cases, it is hard to collect all the temperature data at each location, or there will have some situations when the data is destroyed. At that time, we will use our developed GP solver to fill the missing fields. For example, in this experiment, we only take 70 percent of data points as our training data. Figure 5.3 illustrates the training points, which are non-white pixels, for our hyper-parameter tuning.

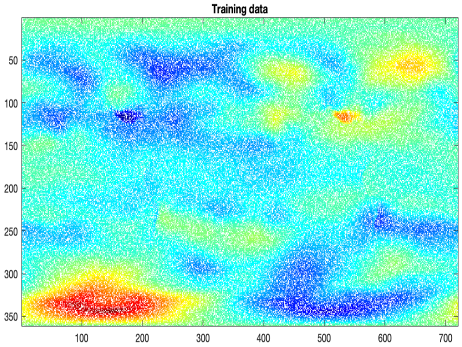

Figure 5.3: 70 percent data points(Training Data)

5.4 Result

Recall that the Gaussian kernel function has two hyper-parameters θ and s. Table 5.1 illustrates the hyper-parameters we tuned and also the log-likelihood value we obtained using those two hyper-parameters. After we use the estimated hyper-parameters to perform prediction, we compare the predicted value with the real value and get the prediction errors which are also included in the table.

Table 5.1: Prediction Result

	Hyper-parameters		Prediction Error		
Log-Likelihood	θ	s	Relative Error	MSE	Root MSE
-82377.8	3	0.04	0.0192	0.0008	0.0278

Also, using the predicted temperature value, we combine it with the training data to recover the figures. Figure 5.4 shows the predicted demeaned temperature at 550 mb in July, 2011 and Figure 5.5 illustrates the resulting predicted average temperature after adding the pixel mean.

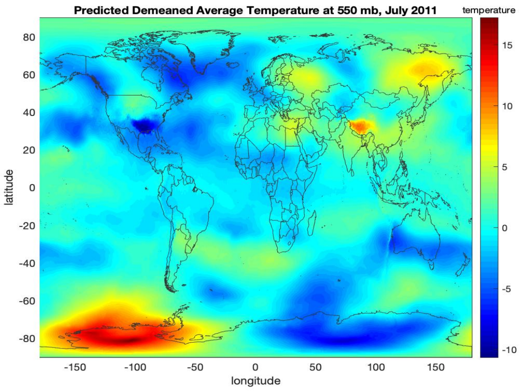

Figure 5.4: Predicted Demeaned Average Temperature at 550mbar, July 2011

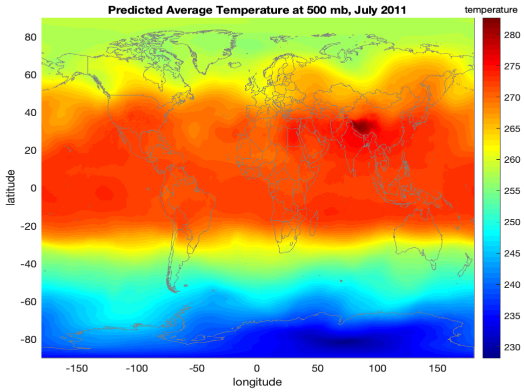

Figure 5.5: Predicted Average Temperature at 550mbar, July 2011

Thus, even when we do not have enough data points, like Figure 5.3 shows, we could still make predictions on those missing values and recover the whole data sets like Figure 5.4 shows.

5.5 Comparison

Although we compared Bayesian Optimization with Naive grid search in the previous section, we want to test the performance of these two methods on the real dataset. However, the dataset we used in the previous section is so large that it will run about five hours if we use the grid search. Therefore, we decide to use a smaller dataset to compare two methods.

This dataset is also provided by NOAA. It is hourly and monthly mean temperature data, over 32 years, from 1979 to 2011 and only include July at 500 mbar isobaric surface. However, we choose a smaller grid size which is 144×73 meaning it has 144 different values for longitude and 73 different values for latitude. In this experiment, we will have $144 \times 73 = 10,512$ data points.

We do the same dataset resize procedure and tune the hyper-parameters using both Bayesian Optimization and naive grid search. Table 5.2 compares the results of these two methods:

Table 5.2: Prediction Result Comparison

	Estimation Result			Prediction Result	
	θ	s	Elapsed Time(s)	Relative Error	MSE
Bayesian Optimization	3.5	0.02	72.12	0.0063	0.0003
Naive Grid Search	1.72	0.04	792.9	0.0156	0.0018

We could see that the naive grid search takes far more time to tune the optimal hyper-parameters compared to Bayesian Optimization. Also, the result is even worse than the result we obtain using Bayesian Optimization. Therefore, using the real temperature dataset to compare these two methods, we could confirm that Bayesian Optimization works more efficiently.

Chapter 6

Conclusions

In this thesis, we develop a fast Gaussian process solver. Since the heart of the Gaussian process, the kernel function, is determined by the hyper-parameters, we need to calculate the log-likelihood function of different hyper-parameters to tune the optimal ones. Since the computational cost of log-likelihood is extremely high due to the log determinant of the kernel matrix and the inverse of it, we have developed some novel methods to quickly evaluate those terms. After knowing the value of those two terms, we could then calculate the log-likelihood function, which helps us to tune the optimal hyper-parameters.

For the log determinant of the kernel matrix, we combine the Stochastic Lanczos Quadrature with the Density of States to approximate it. For the inverse of the kernel matrix, which in turn is equivalent to solving a linear system, we apply the SMW-preconditioned conjugate gradient to quickly solve it. Then, to quickly tune the hyper-parameters which lead to the maximum log-likelihood function, we substitute the naive grid search with Bayesian Optimization.

Finally, to validate our developed algorithm, we test it on a synthetic dataset and a real global temperature dataset. We find that the Bayesian optimization can find optimal hyper-parameters much faster.

Bibliography

[1] David Eriksson, Kun Dong, Eric Lee, David Bindel, and Andrew G Wilson. Scaling gaussian process regression with derivatives. In *Advances in Neural Information Processing Systems*, pages 6867–6877, 2018.

[2] Rajbir S Nirwan and Nils Bertschinger. Applications of gaussian processes in finance. 2018.

[3] Wolfram Burgard, Cyrill Stachniss, Kai Arras, and Maren Bennewitz. Gaussian processes in robotics.

[4] Kristjan Ažman and Juš Kocijan. Application of gaussian processes for black-box modelling of biosystems. *ISA transactions*, 46(4):443–457, 2007.

[5] Hiroyuki Mori and Eitaro Kurata. Application of gaussian process to wind speed forecasting for wind power generation. In *2008 IEEE International Conference on Sustainable Energy Technologies*, pages 956–959. IEEE, 2008.

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

[7] Jie Chen, Haim Avron, and Vikas Sindhwani. Hierarchically compositional kernels for scalable nonparametric learning. *The Journal of Machine Learning Research*, 18(1):2214–2255, 2017.

[8] Shashanka Ubaru, Jie Chen, and Yousef Saad. Fast estimation of tr(f(a)) via stochastic lanczos quadrature. *SIAM Journal on Matrix Analysis and Applications*, 38(4):1075–1099, 2017.

[9] Haim Avron and Sivan Toledo. Randomized algorithms for estimating the trace of an implicit symmetric positive semi-definite matrix. *Journal of the ACM (JACM)*, 58(2):1–34, 2011.

[10] Lin Lin, Yousef Saad, and Chao Yang. Approximating spectral densities of large matrices. *SIAM review*, 58(1):34–65, 2016.

[11] Yousef Saad. Density of states and eigenvalue counts via approximation theory methods, 2014.

[12] Yousef Saad. *Iterative methods for sparse linear systems*, volume 82. siam, 2003.

[13] Zhong-zhi Bai, Gui-qing Li, and Lin-zhang Lu. Combinative preconditioners of modified incomplete cholesky factorization and sherman-morrison-woodbury update for self-adjoint elliptic dirichlet-periodic boundary value problems. *Journal of Computational Mathematics*, pages 833–856, 2004.