Distribution Agreement

In presenting this thesis as a partial fulfillment of the requirements for a degree from Emory University, I hereby grant to Emory University and its agents the non-exclusive license to archive, make accessible, and display my thesis in whole or in part in all forms of media, now or hereafter now, including display on the World Wide Web. I understand that I may select some access restrictions as part of the online submission of this thesis. I retain all ownership rights to the copyright of the thesis. I also retain the right to use in future works (such as articles or books) all or part of this thesis.

Qi Luo April 6, 2022

Efficient Preconditioner for Covariance Matrices Using Geometric Information

by

Rocky Luo

Yuanzhe Xi, Ph.D.
Adviser

Department of Mathematics

Yuanzhe Xi, Ph.D.

Adviser

James G. Nagy, Ph.D.

Committee Member

Joyce C. Ho, Ph.D.

Committee Member

2022

Efficient Preconditioner for Covariance Matrices Using Geometric Information

By

Rocky Luo

Yuanzhe Xi, Ph.D.

Adviser

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

2022

Abstract

Efficient Preconditioner for Covariance Matrices Using Geometric Information
By Rocky Luo

The Gaussian process is a non-parametric machine learning tool designed to solve regression and make predictions. While the Gaussian process is very applicable in statistical modeling, in most cases the computation remains expensive because of its dense covariance matrix. To tackle this problem, we examine the geometry of the Gaussian process dataset and propose a method to solve the covariance matrix linear system using sampling techniques and low-rank approximation. We then validate the effectiveness of our method through theoretical analysis and several numerical experiments.

Efficient Preconditioner for Covariance Matrices Using Geometric Information

By

Rocky Luo

Yuanzhe Xi, Ph.D.

Adviser

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

2022

Acknowledgements

I would like to thank my advisor, Dr. Yuanzhe Xi for his consistent guidance. I have learned a lot from the knowledge and experiences he shared with me. I would also like to thank Dr. James G. Nagy and Dr. Joyce C. Ho for the excellent materials they taught and for their support in my work. Lastly, I wish to extend my special thanks to my parents and my friends for their support and encouragement.

Contents

Chapter 1

Introduction

We begin with some relevant background about the Gaussian process. For a more comprehensive review of this topic, please refer to [1].

A random variable $\boldsymbol{x} \in \mathbb{R}^n$ is multivariate Gaussian if it has the following probability density function:

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

where μ is the mean and Σ is the symmetric positive semidefinite covariance matrix. In most cases, the covariance matrix is positive definite.

The Gaussian process is a set of random variables such that the joint distribution of any finite subset is multivariate Gaussian. It extends the multivariate Gaussian distributions to infinite dimension and is thus used to capture the property of infinite-dimensional function:

$$f \sim \mathcal{GP}(\mu, k) \tag{1.2}$$

where $\mu(x) : \mathbb{R}^d \to \mathbb{R}$ is the mean function and $k : \mathbb{R}^d \to \mathbb{R}^d$ is the kernel function.

It is common to assume the mean of a Gaussian process as 0. However, for the kernel function k, it must be the case that the resulting matrix $\boldsymbol{K}$ generated from any subset

of random variables $x_1, x_2, ..., x_m$

$$\boldsymbol{K} = \begin{bmatrix} k(x_1, x_1) & k(x_1, x_2) & \dots & k(x_1, x_m) \\ k(x_2, x_1) & k(x_2, x_2) & \dots & k(x_2, x_m) \\ \vdots & \vdots & \ddots & \vdots \\ k(x_m, x_1) & k(x_m, x_2) & \vdots & k(x_m, x_m) \end{bmatrix}$$

is a valid covariance matrix corresponding to some multivariate Gaussian distribution. To satisfy the positive semidefiniteness requirement for the covariance matrices, the kernel function must also be positive semidefinite such as the squared exponential kernel function $k(x, y) = \sigma^2 \exp(\frac{-\|x-y\|^2}{l^2})$ defined on $\mathbb{R}^n$ or the standard Brownian motion kernel function $k(s, t) = min(s, t)$ defined on $\mathbb{R}^+$.

Gaussian processes are kernel-based probability distributions with shape and smoothness determined by the kernel function k. It is a flexible non-parametric model to make predictions. However, many practical applications of the Gaussian process involve the expensive computation associated with the dense covariance matrix $\boldsymbol{K}$. For instance, $\boldsymbol{K}^{-1}\boldsymbol{y}$ and $\log\det \boldsymbol{K}$ require $\mathcal{O}(N^3)$. The cost becomes not affordable when the matrix size gets large ($N > 10^5$).

In this thesis, we propose an approach that combines the low-rank approximation and sampling techniques to solve the linear system $\boldsymbol{K}\boldsymbol{u} = \boldsymbol{y}$ through iterative methods. In Chapter 2, we review the existing iterative methods and preconditioners we plan to improve on. In Chapter 3, we discuss the low rank approximation idea related to our problem. In Chapter 4, we propose the sampling technique to obtain our preconditioner. In Chapter 5, we provide some theoretical analysis for our method. Numerical experiments and discussions are presented in Chapter 6, and we draw some conclusions in Chapter 7.

Chapter 2

Iterative Methods

An iterative method is a way of solving linear systems by using an initial value to generate a sequence of improving approximations to the true solution. For instance, to solve the equation $\boldsymbol{Ax} = \boldsymbol{b}$, we can start with an initial guess $\boldsymbol{x}^0$ (a common choice of the initial guess is the zero vector). Then, the iterative technique converts the system $\boldsymbol{Ax} = \boldsymbol{b}$ to a linear transformation $\boldsymbol{x}^{k+1} = \boldsymbol{x}^k + \boldsymbol{c}$, where the sequence $\{\boldsymbol{x}_n\}$ will converge to the true solution $\boldsymbol{x}^*$. In practice, the iterative method will stop when some convergence tolerance has been satisfied.

For a very large dimensional system, direct methods such as Gaussian elimination could be prohibitively expensive while iterative methods is usually efficient in terms of running time and space usage. Constructing a preconditioner for iterative methods will further accelerate the computation. In this chapter, we will illustrate an iterative method called conjugate gradient (CG) and some of its existing preconditioners.

2.1 Conjugate Gradient

The conjugate gradient (CG) is a very classic iterative method to solve the linear system $\boldsymbol{Ax} = \boldsymbol{b}$, where $\boldsymbol{A}$ is symmetric and positive definite. This method fits the aim of our problem because the covariance matrix is symmetric and mostly positive definite. It improves the steepest gradient descent by the conjugate search direction. The algorithm is shown as below:

Algorithm 1 Conjugate Gradient

$\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$
if $\boldsymbol{r}_0$ satisfies the convergence tolerance, then return $\boldsymbol{x}_0$ as the result
$\boldsymbol{p}_0 = \boldsymbol{r}_0$ $k = 0$
while $\boldsymbol{r}_{k+1}$ does not satisfy the convergence tolerance **do**
    $\alpha_k = \frac{\boldsymbol{r}_k^{\mathsf{T}} \boldsymbol{r}_k}{\boldsymbol{p}_k^{\mathsf{T}} \boldsymbol{A} \boldsymbol{p}_k}$
    $\boldsymbol{x}_{k+1} := \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k$
    $\boldsymbol{r}_{k+1} := \boldsymbol{r}_k - \alpha_k \boldsymbol{A} \boldsymbol{p}_k$
    $\beta_k := \frac{\boldsymbol{r}_{k+1}^{\mathsf{T}} \boldsymbol{r}_{k+1}}{\boldsymbol{r}_k^{\mathsf{T}} \boldsymbol{r}_k}$
    $\boldsymbol{p}_{k+1} := \boldsymbol{r}_{k+1} + \beta_k \boldsymbol{p}_k$
    $k := k + 1$
end while
return $\boldsymbol{x}_k$ as the result

As the conjugate gradient iteration solves the linear system associated with the positive definite matrix $\boldsymbol{A}$, we have $\boldsymbol{x}^T \boldsymbol{A} \boldsymbol{x} > \boldsymbol{0}$ for every nonzero vector $\boldsymbol{x} \in \mathbb{R}^m$. The function $\| \cdot \|_A$ can then be defined by:

$$\|\boldsymbol{x}\|_A := \sqrt{\boldsymbol{x}^T \boldsymbol{A} \boldsymbol{x}} \tag{2.1}$$

The conjugate gradient method can be interpreted as an optimization problem that minimizes the norm $\|\boldsymbol{e}_k\|_A$, where $\boldsymbol{e}_k = \boldsymbol{x}^* - \boldsymbol{x}^k$. $\|\boldsymbol{e}_k\|_A$ has the following property [2]:

Theorem 1. *Define P_k as the set of polynomials such that $P_k := \{p \in P'_k : p(0) = 1\}$, where P'_k is the set of polynomials of degree less than or equal to k. Then $\|\boldsymbol{e}_k\|_A$ satisfies:*

$$\begin{aligned} \|\boldsymbol{e}_k\|_{\boldsymbol{A}} &= \inf_{p \in P_k} \|p(\boldsymbol{A}) \boldsymbol{e}_0\|_{\boldsymbol{A}} \\ &\leq \inf_{p \in P_k} \max_{\lambda \in \sigma(\boldsymbol{A})} |p(\lambda)| \ \|\boldsymbol{e}_0\|_{\boldsymbol{A}} \\ &\leq 2 \left(\frac{\sqrt{\kappa(\boldsymbol{A})} - 1}{\sqrt{\kappa(\boldsymbol{A})} + 1} \right)^k \|\boldsymbol{e}_0\|_{\boldsymbol{A}} \end{aligned} \tag{2.2}$$

where $\sigma(\boldsymbol{A})$ denotes the spectrum of $\boldsymbol{A}$ and $\kappa(\boldsymbol{A})$ denotes the condition number of $\boldsymbol{A}$.

This theorem illustrates the convergence of the conjugate gradient iteration. Although it only provides a pessimistic upper bound, we would expect that the convergence is faster if $\boldsymbol{A}$ is more well-conditioned.

2.2 Preconditioning

The previous section provides an example that the condition number of the matrix $\boldsymbol{A}$ can affect the convergence of an iterative method. Thus, in many cases, the original linear system can be transformed into another one so that the new matrix is more well-conditioned for faster convergence. This process, which happens before the iteration, is called "preconditioning".

There are three common types of preconditioners: left preconditioners, right preconditioners, and split preconditioners [2].

- In the case of left preconditioners, the linear system becomes:

$$\boldsymbol{M}^{-1}\boldsymbol{A}\boldsymbol{x} = \boldsymbol{M}^{-1}\boldsymbol{b} \tag{2.3}$$

- In the case of right preconditioners, the linear system becomes:

$$\boldsymbol{A}\boldsymbol{M}^{-1}\boldsymbol{y} = \boldsymbol{b} \text{ with } \boldsymbol{x} = \boldsymbol{M}^{-1}\boldsymbol{y} \tag{2.4}$$

- If $\boldsymbol{M}$ can be factorized in the form $\boldsymbol{M} = \boldsymbol{M}_L\boldsymbol{M}_R$, where $\boldsymbol{M}_L$ and $\boldsymbol{M}_R$ are typically lower and upper triangular matrices, respectively, we can construct a split preconditioner as follows:

$$\boldsymbol{M}_L^{-1}\boldsymbol{A}\boldsymbol{M}_R^{-1}\boldsymbol{y} = \boldsymbol{M}_L^{-1}\boldsymbol{b} \text{ with } \boldsymbol{x} = \boldsymbol{M}_R^{-1}\boldsymbol{y} \tag{2.5}$$

A good preconditioner typically has two properties: $\boldsymbol{M}$ or $\boldsymbol{M}_R$ is structured enough, and the new matrix on the left-hand side of the linear system is more well-conditioned. To use conjugate gradient iteration after preconditioning, people will often choose the split preconditioner to preserve symmetry and positive definiteness as long as the matrix $\boldsymbol{M}$ holds these properties.

2.3 Existing Preconditioners for Conjugate Gradient

In order to improve the performance of the conjugate gradient iteration, there are several preconditioners that can make the iteration converge faster.

2.3.1 Jacobi Preconditioner

One of the most straightforward choices of preconditioners is the Jacobi preconditioner: $\boldsymbol{M} = \text{diag}(\boldsymbol{A})$ [2]. The resulting linear system for conjugate gradient iteration will then be:

$$\boldsymbol{M}^{-1/2}\boldsymbol{A}\boldsymbol{M}^{-1/2}\boldsymbol{y} = \boldsymbol{M}^{-1/2}\boldsymbol{b} \tag{2.6}$$

This preconditioner produces $\boldsymbol{A}' = \boldsymbol{M}^{-1/2}\boldsymbol{A}\boldsymbol{M}^{-1/2}$, which is diagonally scaled.

It is also possible to use the block versions of the Jacobi preconditioner. If we have an index set $I = \{1, ..., n\}$, and I is partitioned as $I = \cup_i I_i$ with each subset I_i mutually disjoint, then

$$m_{i,j} = \begin{cases} a_{i,j}, & \text{if } i \text{ and } j \text{ are in the same subset } I_k \\ 0, & \text{otherwise} \end{cases} \tag{2.7}$$

In this case, we will have the preconditioner $\boldsymbol{M}$ as a block diagonal matrix,

The Jacobi preconditioner results in a very computationally inexpensive matrix inverse. However, we will not select this preconditioner as it does not greatly improve the convergence.

2.3.2 Incomplete Cholesky Preconditioner

Consider the preconditioned linear system in Equation 2.5. Even when $\boldsymbol{A}$ and $\boldsymbol{M}$ are favorably sparse, we usually encounter the problem that the lower factor of $\boldsymbol{M}$ is much less sparse than $\boldsymbol{M}$. Under such conditions, the incomplete Cholesky factorization can remedy this issue [2]. The most common incomplete Cholesky preconditioner is constructed by

factorizing the matrix $\boldsymbol{A}$ with zero fill-in, i.e., if $a_{i,j} = 0$, then the corresponding entry in $\boldsymbol{M}_R$ is 0. The algorithm is shown below:

Algorithm 2 Zero Fill-in Incomplete Cholesky Factorization

```
for k = 1 : n − 1 do
    r_{k,k} = √a_{k,k}
    for j = k + 1 : n do
        r_{k,j} = a_{k,j} / r_{k,k}
    end for
    for i = k + 1 : n do
        for j = 1 : n do
            if a_{i,j} ≠ 0 then
                a_{i,j} = a_{i,j} − r_{k,i} r_{k,j}
            end if
        end for
    end for
end for
```

The accuracy of the zero fill-in incomplete Cholesky preconditioner may not result in a good convergence rate. There are several improvements that allow some levels of fill-in. Moreover, instead of dropping out elements during the factorization, there are also methods that compensate for the discarded entries, which is called the diagonal compensation strategy [2]. Unfortunately, the strategies mentioned above only work for the favorable sparse matrices, whereas the covariance matrix in our case is dense. However, the dropping elements idea will be applied to our numerical test. To resolve the traditional incomplete factorization which is blind to numerical values, we can set a tolerance value to help us determine whether to drop an element on the off diagonal, i.e., ignoring small values. More details will be covered in the later chapter.

2.3.3 Spatial Data and Factorized Sparse Approximate Inverse

In one dimensional space, the inverse of the covariance matrix generated from the exponential kernel function is sparse. In higher dimensions, the inverse may not be sparse, but the value of the elements decays rapidly from the diagonal, which makes the Factorized Sparse Approximate Inverse (FSAI) method very effective [3].

This method provides a preconditioner in the form of:

$$\boldsymbol{G}_L^T \boldsymbol{G}_L \approx \boldsymbol{A}^{-1} \tag{2.8}$$

where $\boldsymbol{G}_L$ has some sparsity pattern on its lower triangular part. Suppose $\boldsymbol{A}$ has the exact Cholesky factorization $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^T$. In order to have $\boldsymbol{G}_L \boldsymbol{A} \boldsymbol{G}_L^T$ close to the identity matrix, the FSAI method computes $\boldsymbol{G}_L$ by minimizing the norm:

$$\begin{aligned} \|\boldsymbol{I} - \boldsymbol{G}_L \boldsymbol{L}\|_F^2 &= \mathrm{tr}((\boldsymbol{I} - \boldsymbol{G}_L \boldsymbol{L})(\boldsymbol{I} - \boldsymbol{G}_L \boldsymbol{L})^T) \\ &= \mathrm{tr}(\boldsymbol{I} - \boldsymbol{G}_L \boldsymbol{L} - \boldsymbol{L}^T \boldsymbol{G}_L^T + \boldsymbol{G}_L \boldsymbol{L} \boldsymbol{L}^T \boldsymbol{G}_L^T) \end{aligned} \tag{2.9}$$

The matrix $\boldsymbol{L}$ is not available in this case. However, we assume that the sparsity pattern of $\boldsymbol{G}_L$, denoted as S such that $S = \{(i,j) \mid \boldsymbol{G}_{L(i,j)} \neq 0\}$, is given. By taking the partial derivative of $\mathrm{tr}(\boldsymbol{I} - \boldsymbol{G}_L \boldsymbol{L} - \boldsymbol{L}^T \boldsymbol{G}_L^T + \boldsymbol{G}_L \boldsymbol{L} \boldsymbol{L}^T \boldsymbol{G}_L^T)$ with respect to the entries located in the sparsity pattern of $\boldsymbol{G}_L$, and using the following derivatives of traces formula [4]: $\frac{\partial \mathrm{tr}(\boldsymbol{X}\boldsymbol{A})}{\partial \boldsymbol{X}} = \boldsymbol{A}^T$, $\frac{\partial \mathrm{tr}(\boldsymbol{A}\boldsymbol{X}^T)}{\partial \boldsymbol{X}} = \boldsymbol{A}$ and $\frac{\partial \mathrm{tr}(\boldsymbol{X}\boldsymbol{B}\boldsymbol{X}^T)}{\partial \boldsymbol{X}} = \boldsymbol{X}\boldsymbol{B}^T + \boldsymbol{X}\boldsymbol{B}$, we obtain the following property:

$$(\boldsymbol{G}_L \boldsymbol{A})_{ij} = (\boldsymbol{L}^T)_{ij}, \ \ (i,j) \in S \tag{2.10}$$

Moreover, $(i,j) \in S$ implies that $i \geq j$ since $\boldsymbol{G}_L$ is lower triangular. Since $\boldsymbol{L}^T$ is upper triangular, The Equation 2.10 can be simplified as below:

$$(\boldsymbol{G}_L \boldsymbol{A})_{ij} = \begin{cases} (\boldsymbol{L}^T)_{ij}, & i = j \\ 0, & i \neq j \text{ and } (i,j) \in S \end{cases} \tag{2.11}$$

This property shows that the off-diagonal entries of $\boldsymbol{L}$ do not need to be computed exactly when we try to use $\boldsymbol{G}_L$ to approximate $\boldsymbol{L}^{-1}$. If we cannot obtain the diagonal entries of $\boldsymbol{L}$, we can still construct $\boldsymbol{G}_L$ using the matrix $\boldsymbol{G}'_L$ which shares the same sparsity pattern

and $\boldsymbol{G}'_L$ satisfies:

$$(\boldsymbol{G}'_L \boldsymbol{A})_{ij} = \boldsymbol{I}_{ij} \ \ \forall (i,j) \in S \tag{2.12}$$

Note that $\boldsymbol{G}'_L$ can be constructed relatively cheaply because we can compute each row independently by solving a small system associated with a symmetric positive definite matrix $\boldsymbol{A}_{J,J}$ where J is the index set $\{j \mid (i,j) \in S\}$ [5]. Then choose a diagonal scaling matrix $\boldsymbol{D}$ and let $\boldsymbol{G}_L = \boldsymbol{D}\boldsymbol{G}'_L$ such that:

$$(\boldsymbol{G}_L \boldsymbol{A} \boldsymbol{G}_L^T)_{ij} = 1, \ \ i = j \tag{2.13}$$

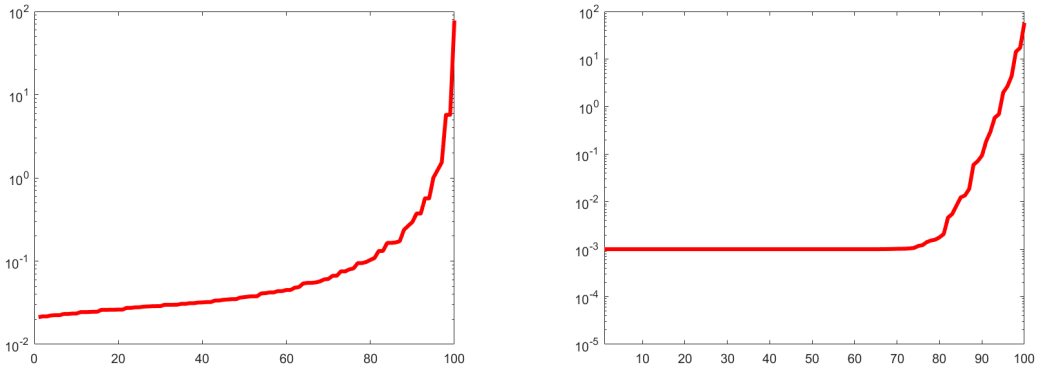
Therefore, $\boldsymbol{G}_L$ can be computed without knowing the Cholesky factor $\boldsymbol{L}$.

FSAI helps us realize the possibility of reducing the cost of constructing a preconditioner for dense and large covariance matrix if we don't use all entries of the covariance matrix during computation. In the next chapter, we will introduce the idea of constructing an efficient Cholesky factor based on the low-rank approximation.

Chapter 3

Another View of the Cholesky Factorization

The eigenvalue distribution of two covariance matrices, as shown in Figure 3.1, presents a low proportion of large eigenvalues. This phenomenon motivates us to construct an efficient preconditioner based on the low-rank approximation idea. In this chapter, we will introduce the pivoted (incomplete) Cholesky factorization using the low-rank approximation, which can reduce the task of constructing an efficient Cholesky factor to selecting rows (columns) from the symmetric and positive definite matrix.



(a) the Absolute Exponential Covariance Matrix

(b) the Squared Exponential Covariance Matrix

Figure 3.1 Eigenvalue Distribution

3.1 Low-Rank Approximation

In this section, we will introduce some background about the low-rank approximation that is related to our method and the comparison experiments in Chapter 6.

A matrix $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ is low-rank if $r = \text{rank}(A) \ll m, n$. Under this condition, $\boldsymbol{A}$ has a nice factorization of the form:

$$\boldsymbol{A} = \boldsymbol{B}\boldsymbol{C}^T \tag{3.1}$$

where $\boldsymbol{B} \in \mathbb{R}^{m \times r}$ and $\boldsymbol{C} \in \mathbb{R}^{n \times r}$.

However, in numerical computations, it is almost impossible to have a low-rank matrix. Therefore, the low-rank approximation aims to find $\boldsymbol{B}$ and $\boldsymbol{C}$ such that $\boldsymbol{A} \approx \boldsymbol{B}\boldsymbol{C}^T$, i.e., $\|\boldsymbol{A} - \boldsymbol{B}\boldsymbol{C}^T\|$ is small. To investigate the low-rank approximation, it is more common to write it in the following form:

$$\boldsymbol{A} \approx \boldsymbol{B}\boldsymbol{H}\boldsymbol{C}^T \tag{3.2}$$

where $\boldsymbol{B} \in \mathbb{R}^{m \times r}$, $\boldsymbol{H} \in \mathbb{R}^{r \times r}$, and $\boldsymbol{C} \in \mathbb{R}^{n \times r}$.

Recall the Singular Value Decomposition (SVD) of $\boldsymbol{A}$ is:

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T \tag{3.3}$$

Then a rank-r approximation of $\boldsymbol{A}$ is given by:

$$\boldsymbol{A} \approx \boldsymbol{B}\boldsymbol{H}\boldsymbol{C}^T = \boldsymbol{U}_r\boldsymbol{\Sigma}_r\boldsymbol{V}_r^T \tag{3.4}$$

where $\boldsymbol{U}_r$ has the first r left singular vectors of $\boldsymbol{A}$ in its column, $\boldsymbol{\Sigma}_r$ is a diagonal matrix with the first r singular value on the diagonal, and $\boldsymbol{V}_r$ has the first r right singular vectors of $\boldsymbol{A}$ in its column.

In fact, the SVD gives the best theoretical low-rank approximation because the sin-

gular values and singular vectors are ordered in terms of "importance"[6]. Nevertheless, the SVD generally runs in cubic time.

The Nyström method provides a low-rank approximation by directly sampling a column index set $J = \{j_1, j_2, ..., j_r\}$ and a row index set $I = \{i_1, i_2, ..., i_r\}$ from the matrix $\boldsymbol{A}$ and constructing $\boldsymbol{B} = A(:, J)$, $\boldsymbol{H}^{-1} = A(I, J)$ and $\boldsymbol{C}^T = A(I, :)$ accordingly [7]. Unfortunately, computing the optimal J and I is an NP-hard problem. The Nyström method based on the determinantal point process (DPP) gives a near-optimal solution while it is still too expensive to be implemented in real cases. There are heuristic algorithms to reduce the computation cost, and the Adaptive Cross Approximation (ACA) is a very popular one among them [8].

The ACA with full pivoting is shown in Algorithm 3. At each step, the ACA selects the maximum element from the matrix $\boldsymbol{A}$. The indices of this entry are put into J and I, respectively. Then, we update $\boldsymbol{A}$ based on the pseudocode in line 8 so that the values in the selected rows and columns were cleared to 0. The algorithm terminated when some stopping criterion fulfilled, such as the updated $\|\boldsymbol{A}\|_F$ is small.

Algorithm 3 Adaptive Cross Approximation with Full Pivoting

1: $\boldsymbol{A}_0 = \boldsymbol{A}$, $I = \{\}$, $J = \{\}$, k = 0
2: **while** the stopping criterion is not satisfied **do**
3: $\quad k = k + 1$
4: $\quad (i^*, j^*) = \arg\max_{i,j} |\boldsymbol{A}_{k-1}(i, j)|$
5: $\quad I \leftarrow I \cup \{i^*\}$, $J \leftarrow J \cup \{j^*\}$
6: $\quad \delta_k = \boldsymbol{A}_{k-1}(i^*, j^*)$
7: $\quad \boldsymbol{u}_k = \boldsymbol{A}_{k-1}(:, j^*)$, $\boldsymbol{v}_k = \boldsymbol{A}_{k-1}(i^*, :)^T / \delta_k$
8: $\quad \boldsymbol{A}_k = \boldsymbol{A}_{k-1} - \boldsymbol{u}_k \boldsymbol{v}_k^T$
9: **end while**
10: **return** I and J as the result

The ACA with full pivoting performs well, but it is still computationally expensive, especially in the process of searching for the maximum element. The ACA with partial pivoting aims to reduce the cost by only searching the maximum in a certain row. The algorithm starts with the 1^{st} row, and at the end of each step it updates the new searching row by selecting the maximum from the indexed column.

When the matrix $\boldsymbol{A}$ becomes symmetric and positive definite, the ACA will always select the entry on the diagonal where the largest element is guaranteed to be. The

Algorithm 4 Adaptive Cross Approximation with Partial Pivoting

$\boldsymbol{A}_0 = \boldsymbol{A}$, $I = \{\}$, $J = \{\}$, $k = 1$, $i^* = 1$
while the stopping criterion is not satisfied **do**
    $j^* = \arg\max_j |\boldsymbol{A}_{k-1}(i^*, j)|$
    $\delta_k = \boldsymbol{A}_{k-1}(i^*, j^*)$
    $\boldsymbol{u}_k = \boldsymbol{A}_{k-1}(:, j^*)$
    if δ_k **then**
        $I \leftarrow I \cup \{i^*\}$, $J \leftarrow J \cup \{j^*\}$
        $i^* = \arg\max_{i \notin I} |\boldsymbol{u}_k(i)|$
    else
        $\boldsymbol{v}_k = \boldsymbol{A}_{k-1}(i^*, :)^T / \delta_k$
        $\boldsymbol{A}_k = \boldsymbol{A}_{k-1} - \boldsymbol{u}_k \boldsymbol{v}_k^T$
        $k = k + 1$
        $I \leftarrow I \cup \{i^*\}$, $J \leftarrow J \cup \{j^*\}$
        $i^* = \arg\max_{i \notin I} |\boldsymbol{u}_k(i)|$
    end if
end while
return I and J as the result

algorithm becomes the same as the pivoted Cholesky factorization, which is the method we plan to modify. In Chapter 6, we will compare the performance of our method with the ACA.

3.2 Pivoted Cholesky Factorization

Consider a symmetric and positive definite matrix $\boldsymbol{A}$ that is partitioned by the first row and the first column:

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & \boldsymbol{a}_1^T \\ \boldsymbol{a}_1 & \boldsymbol{A}_{22} \end{bmatrix} \tag{3.5}$$

where $a_{11} > 0$.

Then the well-defined Schur complement $\boldsymbol{S} := \boldsymbol{A}_{22} - \frac{1}{a_{11}} \boldsymbol{a}_1 \boldsymbol{a}_1^T$ is also symmetric and positive definite shown as below:

$$\boldsymbol{S}^T = \boldsymbol{A}_{22}^T - \frac{1}{a_{11}} (\boldsymbol{a}_1 \boldsymbol{a}_1^T)^T = \boldsymbol{A}_{22} - \frac{1}{a_{11}} \boldsymbol{a}_1 \boldsymbol{a}_1^T = \boldsymbol{S} \tag{3.6}$$

and

$$0 < \begin{bmatrix} x \\ \boldsymbol{y} \end{bmatrix}^T \boldsymbol{A} \begin{bmatrix} x \\ \boldsymbol{y} \end{bmatrix} = \begin{bmatrix} x \\ \boldsymbol{y} \end{bmatrix}^T \begin{bmatrix} a_{11}x + \boldsymbol{a}_1^T\boldsymbol{y} \\ x\boldsymbol{a}_1 + \boldsymbol{A}_{22}\boldsymbol{y} \end{bmatrix} = \begin{bmatrix} x \\ \boldsymbol{y} \end{bmatrix}^T \begin{bmatrix} 0 \\ \boldsymbol{S}\boldsymbol{y} \end{bmatrix} = \boldsymbol{y}^T\boldsymbol{S}\boldsymbol{y} \tag{3.7}$$

where $\boldsymbol{y}$ is an arbitrary nonzero vector in $\mathbb{R}^{n-1}$ and $x := -\frac{\boldsymbol{a}_1^T\boldsymbol{y}}{a_{11}}$.

Therefore, the first step of the Cholesky factorization decomposes $\boldsymbol{A}$ into the form of a rank 1 matrix and an extra matrix $\boldsymbol{E}_1$:

$$\boldsymbol{A} = \frac{1}{a_{11}} \begin{bmatrix} a_{11} \\ \boldsymbol{a}_1 \end{bmatrix} \begin{bmatrix} a_{11} \\ \boldsymbol{a}_1 \end{bmatrix}^T + \boldsymbol{E}_1 \tag{3.8}$$

where $\boldsymbol{E}_1 = \begin{bmatrix} 0 & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{S} \end{bmatrix}$.

In fact, the Cholesky factorization can be viewed as a recursive factorization of the Schur Complement, which implies at the k^{th} step $\boldsymbol{A}$ can be decomposed into k rank 1 matrix and $\boldsymbol{E}_k$. If the rank k factorization already gives a good enough approximation $\boldsymbol{A}$, the Cholesky factorization no longer continues to decompose $\boldsymbol{E}_k$ based on the low-approximation idea .

Figure 3.2 provides a visualization of the approximated Cholesky factor. Notice that starting from the $(k+1)^{th}$ diagonal entries, a small value has been added to preserve the positive definiteness for the approximation.

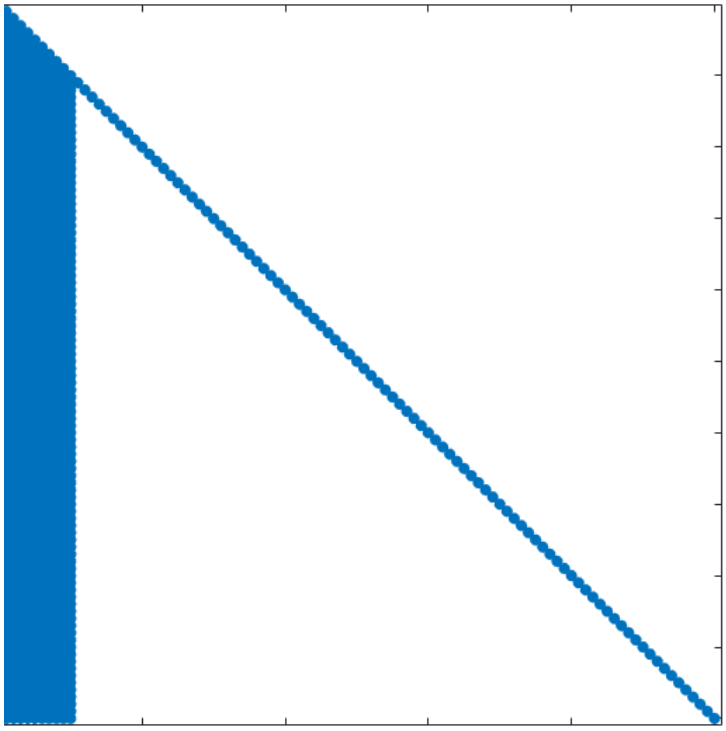


Figure 3.2 Approximated Cholesky Factor Visualization

It is not likely that the first k^{th} columns and rows give a good Cholesky factor. We consider permuting the matrix $\boldsymbol{A}$ so that we can obtain a more accurate approximation provided k is fixed. Under this condition, the original linear system $\boldsymbol{Ax} = \boldsymbol{b}$ is modified to:

$$\boldsymbol{PAPz} = \boldsymbol{Pb} \tag{3.9}$$

where $\boldsymbol{P}$ is an elementary permutation matrix and $\boldsymbol{z} = \boldsymbol{P}^T\boldsymbol{x}$.

As computing $\boldsymbol{Pb}$ and solving $\boldsymbol{z} = \boldsymbol{P}^T\boldsymbol{x}$ only requires $O(n)$ cost, any permutation on $\boldsymbol{A}$ is acceptable. Therefore, we can freely select the set of rows (columns) to be factorized.

In the next chapter, we will propose how to choose the rows (columns) for the covariance matrix generated from the Gaussian process.

Chapter 4

Inducing Points Selection

The low-rank approximation serves as a common strategy to overcome the expensive operation on dense matrices. The examples illustrated in Chapter 3 are all algebraic methods. However, in the case of the covariance matrix $\boldsymbol{K}$ generated from the Gaussian process, the matrix has a nice property that each row and column represents a data point's correlation with all other points in the space. Therefore, to construct the low-rank approximation of $\boldsymbol{K}$ using the Nyström method, we can choose rows (columns) by simply selecting the set of inducing points S. In this chapter, we propose a method to choose S based on the Farthest Point Sampling technique. Some experiments and results to construct the preconditioner using this method will be discussed in Chapter 6.

4.1 Motivation from Spatial Statistics

The covariance matrix $\boldsymbol{K}$ captures the relationship between pairs of data points. Consider a common covariance matrix generated from squared exponential kernel function $k(x, y) = \sigma^2 \exp(\frac{-\|x-y\|^2}{l^2})$. Since the kernel function is Lipschitz continuous, the distance between a pair of data points provides an upper bound for the corresponding kernel function with a constant. Therefore, it would be nice if we could utilize the geometry of the dataset when constructing the low-rank approximation.

Recall the Cholesky factorization can be written as the block version of $\boldsymbol{K} = \boldsymbol{L}\boldsymbol{D}\boldsymbol{L}^T$:

$$\begin{bmatrix} \boldsymbol{K}_{11} & \boldsymbol{K}_{12} \\ \boldsymbol{K}_{21} & \boldsymbol{K}_{22} \end{bmatrix} = \begin{bmatrix} \boldsymbol{I} & \boldsymbol{0} \\ \boldsymbol{K}_{21}(\boldsymbol{K}_{11})^{-1} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{K}_{11} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{K}_{22} - \boldsymbol{K}_{21}(\boldsymbol{K}_{11})^{-1}\boldsymbol{K}_{12} \end{bmatrix} \begin{bmatrix} \boldsymbol{I} & (\boldsymbol{K}_{11})^{-1}\boldsymbol{K}_{12} \\ \boldsymbol{0} & \boldsymbol{I} \end{bmatrix} \tag{4.1}$$

The computation process can be viewed as a recursive factorization of the part $\boldsymbol{K}_{22} - \boldsymbol{K}_{21}(\boldsymbol{K}_{11})^{-1}\boldsymbol{K}_{12}$, which is the Schur complement of the block $\boldsymbol{K}_{11}$.

From the spatial statistics point of view, if we have the joint Gaussian distribution $(X, Y) \sim \mathcal{GP}(0, \boldsymbol{K}^{'})$, the conditional of Y given X is also in the Schur complement form:

$$Cov(Y|X) = \boldsymbol{K}^{'}_{22} - \boldsymbol{K}^{'}_{21}(\boldsymbol{K}^{'}_{11})^{-1}\boldsymbol{K}^{'}_{12} \tag{4.2}$$

The Cholesky factorization represents the iteration of conditioning a Gaussian process, which suggests that the conditional independence is related to the sparsity in the Cholesky factor of $\boldsymbol{K}$ [9].

Many kernel functions under the Matérn kernel family are generated from the finitely smooth Gaussian process. These kernel functions give larger values to pairs of nearby points and smaller values to pairs of distant points. The finite smoothness also indicates that for a data point x_1 its value is nearly independent of a distant point x_2 conditional on a known neighbor point x_3 of x_1 [9]. Therefore, the set of inducing points could be chosen so that any point from the entire dataset X is close enough to an inducing point in S.

The information above guides us to recursively select the most distant point away from the selected set S, which matches an algorithm called the Farthest Point Sampling. To validate this intuitive understanding, in Chapter 5 we will give some theoretical analysis for our low-approximation preconditioner using the Farthest Point Sampling. Before that, we will illustrate this sampling method in detail [10].

4.2 Farthest Point Sampling Method

The Farthest Point Sampling (FPS) is a sampling method widely used in deep learning neural networks such as PointNet++ and VoteNet to process point cloud data or detect 3D objects. In this section, we will illustrate this method, which is applicable to select the set of inducing points S in our problem.

4.2.1 The Method Overview

Suppose there are n data points, and we would like to sample k data points to form the set of inducing points S based on the Farthest Point Sampling. First, create another set called R which represents all the points that will not be selected. Initially, $R = X$, where X is the whole set of data points.

- **Step 1:** Randomly select a data point from R denoted as x_1 and put it to the set S. Now we have 1 point in S and $n - 1$ points in R.

- **Step 2:** Calculate the distance between each point in R to the point in S. Select the point with the maximal distance denoted as x_2. Now we have 2 points in S and $n - 2$ points in R.

- **Step 3:** Now, it brings up a question about how to define the distance between each point in R to the points in S because there is more than one element in the latter set. Suppose x_r is a data point in R, then its valid distance to S is defined as:

$$\min(d(x_1, x_r), d(x_2, x_r))$$

Pick the maximum from $n - 2$ valid distances and put the corresponding data point to S.

...

- **Step k:** Pick the maximum from $n-k+1$ valid distances and put the corresponding data point to S. Now, we finalize our sampling with k points in S.

The core of the FPS method can be summarized as an iteration of the following process: for each data point in R, calculate its nearest neighbor in S and record the distance. Then select the point in R that has the maximal distance to its nearest neighbor and move the point from R to S.

Figure 4.1 and Figure 4.2 illustrate 100 sampling vertices from 1859 vertices of the Simplified Stanford Bunny model [11] based on uniform sampling and the FPS, respectively. As we can see, the FPS method generates a sample that spreads more evenly over the model.

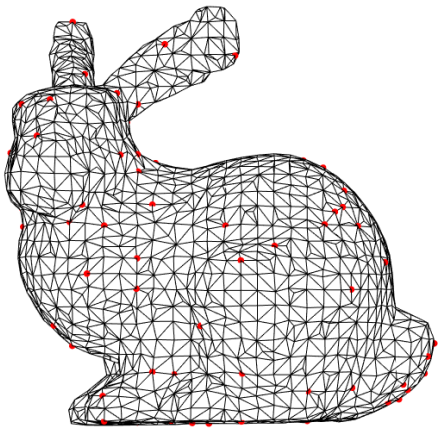


Figure 4.1 Stanford Bunny and Uniform Sampling Vertices

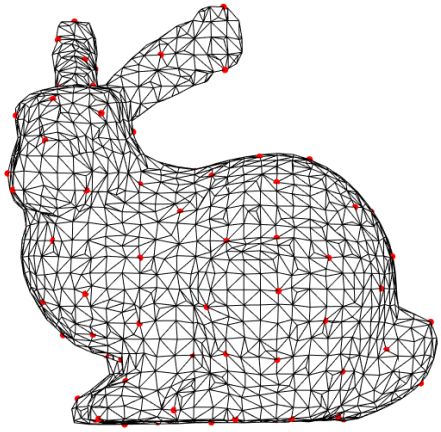


Figure 4.2 Stanford Bunny and the FPS Vertices

4.2.2 The First Point of the FPS

At **Step 1**, it brings up a question of whether any arbitrary point is good enough to be the first selected point. Figure 3.3 presents an extreme case that the Farthest Point Sampling may not capture comprehensive geometric information if the first selected point is bad and the number of selected points is limited.

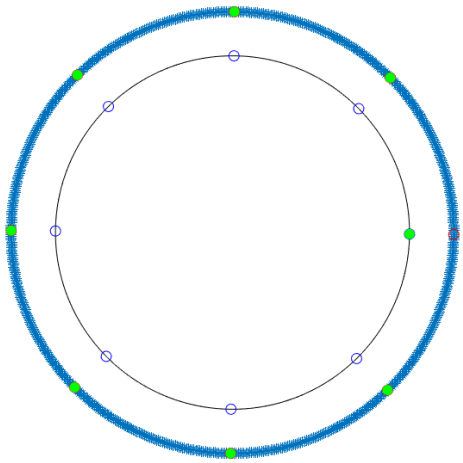


Figure 4.3 Farthest Point Sampling from Data Points on Two Loops

Figure 4.3 displays 16 data points, which are evenly distributed on two circular rings, where the radius ratio is 5 : 4. If we would like to obtain a set of only 8 selected points

S, then there is a 50% chance that we will select all points on the boundary (outer ring) without jumping into the interior region (inner ring). This issue tends to be more severe if the data points are in a very high dimension [12]. Therefore, although computing the center of dataset costs more time, it could be considered in advance to ensure performance.

4.2.3 Reducing the Repetitive Computation

The method in **Section 4.2.1** includes a large amount of repetitive computation. At the start of **Step m**, where $m \geq 3$, the set S has $m-1$ data points, and R has $n-m+1$ data points. For every point S, we need to calculate its distance to every point in R in order to obtain the valid maximum distance. The time complexity for this single step will be $O(nm - m^2)$ if the brute force approach is implemented. However, for an arbitrary data point x_r in R, at **Step (m-1)** we have already calculated its distance to $m-2$ points in S, denoted as $d(x_1, x_r), d(x_2, x_r), ..., d(x_{m-2}, x_r)$. Moreover, we also obtained the valid distance for x_r, denoted as d_r^{m-2} such that:

$$d_r^{m-2} = \min(d(x_1, x_r), d(x_2, x_r), ..., d(x_{m-2}, x_r)) \tag{4.3}$$

Therefore, if we store the value d_r^{m-2}, we only need to compute $d(x_{m-1}, x_r)$ at **Step m** based on the following property:

$$\min(d(x_1, x_r), d(x_2, x_r), ..., d(x_{m-1}, x_r)) = \min(d_r^{m-2}, d(x_{m-1}, x_r)) \tag{4.4}$$

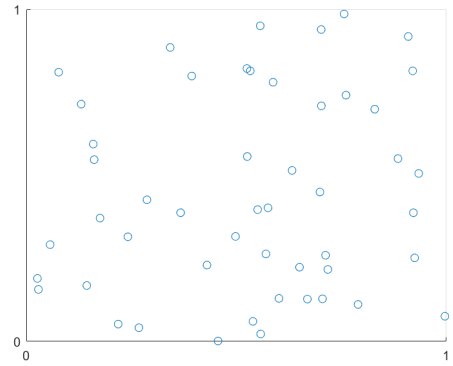
We can create an array of size n to store the valid distance between every point in R to the set S and thus reduce each step's implementation to linear time complexity.

4.2.4 Applying Domain Decomposition

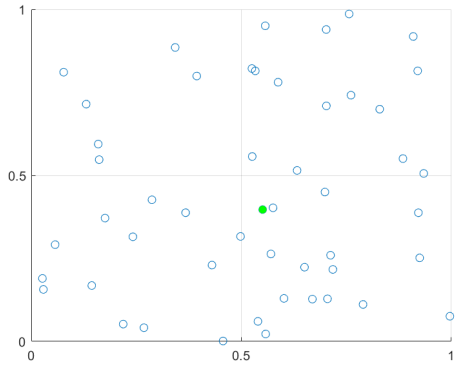
Domain decomposition is the partitioning of a certain region or space, which splits a problem on a large domain into smaller problems on subdomains. The subproblems are often times independent, which makes them suitable for parallel computing. Domain decompo-

sition methods have been widely used for solving PDEs and constructing preconditioners for Krylov iterative methods.

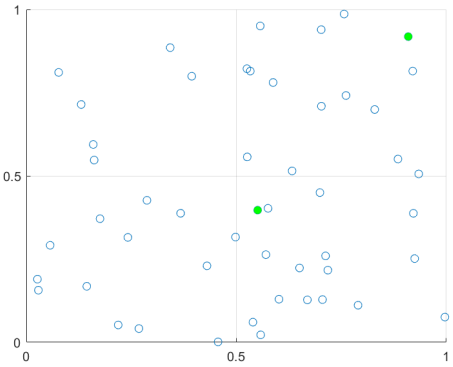
To further reduce the number of candidate points in the set R at each step, we can apply the non-overlapping domain decomposition technique to obtain an approximate solution to the optimal FPS. For example, in Figure 4.4, we generate 50 random data points on the domain $[0,1] \times [0,1] \in \mathbb{R}^2$ to demonstrate the first few steps of the heuristic FPS. First, partition the domain into four equal squares, and select the first point using the original FPS method, which is located in the lower-right square. Now we only search the data points in the upper-left square because those points in the upper-left corner tends to have the maximal distance to the set S. In the next two steps, we only search the data points in the remaining two squares and the remaining one square. Now, partition each square into four small squares and iterative this process on small empty squares. Figure 4.4 illustrates a descent result because the FPS using domain decomposition actually provides the same set of selected points as the optimal FPS. It is not a common case as the discrepancy between two methods in a certain step may affect the following valid distances. Nevertheless, overall we could expect good performance for this heuristic FPS because in each step it will select one of the top candidates.



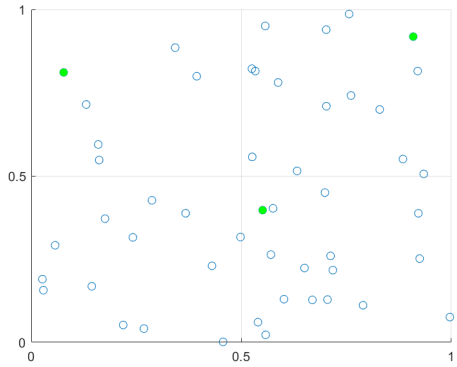
(a) Initial Stage



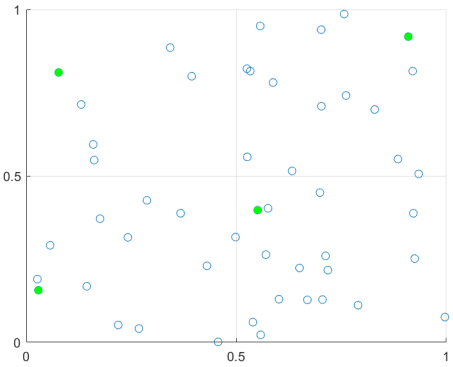
(b) Step 1



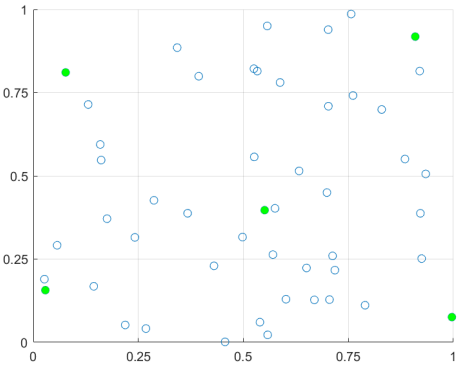
(c) Snapshot 2



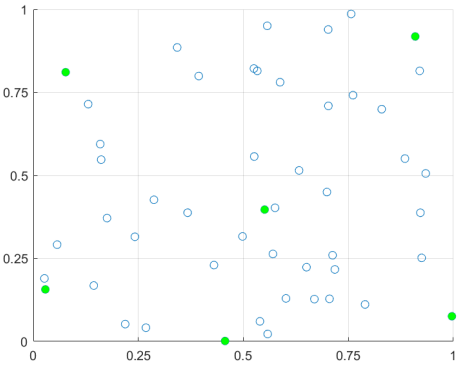
(d) Snapshot 3



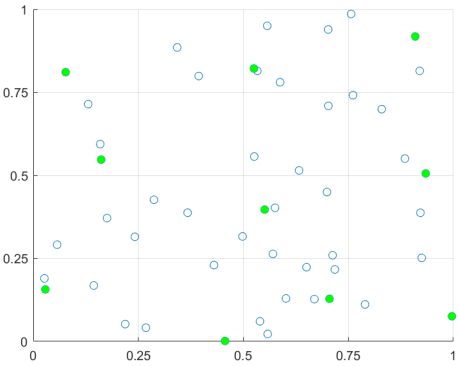
(e) Snapshot 4



(f) Snapshot 5



(g) Snapshot 6



(h) Snapshot 10

Figure 4.4 FPS using Domain Decomposition

Chapter 5

Theoretical Analysis

In this chapter, we provide some theoretical analysis of our FPS Cholesky factorization.

Recall we can write the rank k factorization of a covariance matrix $\boldsymbol{K} \in \mathbb{R}^{n \times n}$ as the following form:

$$\boldsymbol{K}_k = \begin{bmatrix} \boldsymbol{K}_{11} \\ \boldsymbol{K}_{12}^T \end{bmatrix} \boldsymbol{K}_{11}^{-1} \begin{bmatrix} \boldsymbol{K}_{11} & \boldsymbol{K}_{12} \end{bmatrix} \tag{5.1}$$

where is $\boldsymbol{K}_{11}$ is a $k \times k$ submatrix of $\boldsymbol{K}$.

For the FPS method, suppose we have sampled i data points from the dataset X, and the set of selected points is denoted as X_i. Therefore, the next sample point x_{i+1} is defined as:

$$x_{i+1} := \underset{x \in X \setminus X_i}{\arg\max}\, d(x, X_i) \tag{5.2}$$

Now, we define the fill distance between X and X_i as:

$$h_i = \max_{x \in X \setminus X_i} d(x, X_i) \tag{5.3}$$

According to [13], the following error bound has been proved for a rank k approxima-

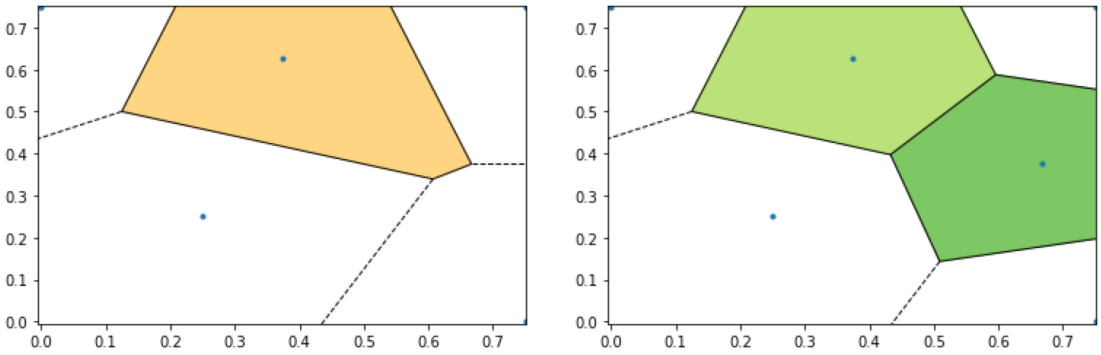
tion $\boldsymbol{K}_k$ to the Gaussian Kernel matrix $\boldsymbol{K}$:

$$\|\boldsymbol{K} - \boldsymbol{K}_k\|_2 < C^{'} \|\boldsymbol{K}\|_2 \, e^{-C/h_k} \tag{5.4}$$

where C and $C^{'}$ are positive constants.

As we can see from Equation 5.4, a smaller h_k leads to better low-rank approximation. This result matches what we stated before that X_k needs to be spread over the entire dataset X so that any non-selected data point is not far away from X_k. Unfortunately, at the $(i+1)^{th}$ step, the FPS can only select the point which obtains the fill distance h_i, but it does not guarantee to minimize h_i as it is already fixed. However, we can provide an upper bound for the fill distance h_k obtained by the desired k^{th} step of the FPS. The argument is based on the setting of continuous points in the Voronoi Diagram [14], and we suppose $r \leq k$, where k is the desired rank we want for our low-rank approximation. Figure 5.1 shows an example of the Voronoi diagram with inducing points sampled through the FPS.

Definition 1. *The Voronoi diagram of a set of points X_r, denoted as $VD(X_r)$, is defined to be the collection of Voronoi cells of all the points in the set, where each Voronoi cell $Vc(x_i)$ of x_i is defined to be all the points which are closer to x_i than other selected points x_j. The bounded Voronoi diagram $BVD(X_r)$ restricts $VD(X_r)$ to have the domain X (the entire dataset in our case). The vertices of edges of $BVD(X_r)$ are denoted as V^r and E^r. Define d_M^r as the maximum distance between the vertex and the set of sampling points X_r. d_m^r as the minimum distance between the vertex and the set of sampling points X_r.*



(a) Voronoi Diagram - Before Adding $(\frac{2}{3}, 0.375)$

(b) Voronoi Diagram - After Adding $(\frac{2}{3}, 0.375)$

Figure 5.1 Voronoi Diagram with Inducing Points Sampled through the FPS: the domain X is $[0, 0.75] \times [0, 0.75] \subset \mathbb{R}^2$, and we continuously select the data points $(0.25, 0.25)$, $(0.75, 0.75)$, $(0, 0.75)$, $(0.75, 0)$, $(0.375, 0.625)$ via the FPS. The next point selected via the FPS is $(\frac{2}{3}, 0.375)$.

Theorem 2. *In [10], the following properties have been proved if the set X_k is sampled via the FPS:*

- *The next sampling point x_{r+1} lies on the vertex V^k.*
- *$d(x_i, x_j) \geq d_M^k$ for all the data points $x_i, x_j \in X_k$*
- *$d_M^k \leq 2d_m^k$.*
- *$d(x_i, x_j) \leq d_M^k$ for all the data points $x_i, x_j \in X_k$ such that x_i and x_j share a common edge.*

Let $q_k := \min_{x_i, x_j \in X_k} d(x_i, x_j)$. We can obtain these two results: $q_k \leq 2h_k$ and $h_k \leq q_k$ [15]. Based on Equation 5.4, the second result guarantees the low-rank approximation accuracy because the fill distance h_k via the FPS method is bounded above by the minimum distance between two points in X_k. From the numerical stability perspective, if two data points in the set of selected points X_k are too close to each other, then the $r \times r$ covariance matrix $\boldsymbol{K}_{11}$ could be ill-conditioned, which leads to a huge gap between the theoretical approximation error and the numerical error. However, the first result guarantees that the next point sampled via the FPS will not be too close to any of the selected points. Therefore, the FPS Cholesky factorization is validated to have good approximation accuracy and numerical stability.

Chapter 6

Numerical Experiments

In this chapter, we conduct several experiments to examine the effectiveness of the preconditioners using the FPS Cholesky factorization.

6.1 Number of Large Elements for the Cholesky Factor

To verify that the FPS method facilitates the Cholesky factorization to become "sparser," we compare the number of large values in the Cholesky factors that were generated without permutation with those generated with permutation based the FPS. Our experiments incorporate the following variables: the threshold to determine whether a value is large (1e$-$2, 1e$-$3, and 1e$-$4), the dataset dimension ($\mathbb{R}^2$, $\mathbb{R}^3$, and $\mathbb{R}^6$), the distribution of data points (points on a grid and random coordinates), the kernel function type (the squared exponential kernel function and the absolute exponential kernel function), the covariance matrix size (1000×1000, 2500×2500, 6000×6000), and the Cholesky factor type (no permutation, partial permutation, and full permutation). The results are displayed in Table 6.1. To make each subtable more concise, we only name them using the keywords for the variables. For instance, "2500×2500, $\mathbb{R}^3$, Squared Exponential Kernel Function, Random Coordinates" means that this subtable displays the experiments conducted on 2500×2500 squared exponential covariance matrices generated from data points in $\mathbb{R}^3$.

Table 6.1: **Number of Large Values in the Covariance Matrix**

(a) 1000×1000, $\mathbb{R}^2$, the Squared Exponential Kernel Function, Random Coordinates

	Threshold Value		
	1e−2	1e−3	1e−4
Chol without permutation	31,242	90,428	265,345
Chol full permutation	26,798	80,480	251,806
Chol partial permutation (k = 120)	26,777	80,758	254,591

(b) 2500×2500, $\mathbb{R}^2$, the Squared Exponential Kernel Function, Random Coordinates

	Threshold Value		
	1e−2	1e−3	1e−4
Chol without permutation	88,078	304,837	1,008,935
Chol full permutation	79,348	262,268	923,169
Chol partial permutation (k = 120)	79,336	269,067	953,357

(c) 6000×6000, $\mathbb{R}^2$, the Squared Exponential Kernel Function, Random Coordinates

	Threshold Value		
	1e−2	1e−3	1e−4
Chol without permutation	246,049	910,241	3,262,113
Chol full permutation	213,310	757,634	2,935,150
Chol partial permutation (k = 120)	217,297	826,045	3,164,918

(d) 2500×2500, $\mathbb{R}^3$, the Squared Exponential Kernel Function, Random Coordinates

	Threshold Value		
	1e−2	1e−3	1e−4
Chol without permutation	191,069	742,091	1,762,144
Chol full permutation	175,567	693,307	1,615,506
Chol partial permutation (k = 120)	190,296	746,083	1,758,407
Chol partial permutation (k = 250)	178,476	733,955	1,744,043

(e) 2500×2500, $\mathbb{R}^6$, the Squared Exponential Kernel Function, Random Coordinates

	Threshold Value		
	1e−2	1e−3	1e−4
Chol without permutation	78,233	329,682	953,791
Chol full permutation	74,906	318,963	919,325
Chol partial permutation (k = 120)	78,070	326,080	937,319
Chol partial permutation (k = 800)	76,463	321,801	920,283

To be Continued

(f) 2500×2500, $\mathbb{R}^2$, the Squared Exponential Kernel Function, Points on a Grid

	Threshold Value		
	1e−2	1e−3	1e−4
Chol without permutation	1,317,230	2,519,366	3,099,515
Chol full permutation	41,292	174,221	973,346
Chol partial permutation (k = 120)	41,290	171,137	1,438,313

(g) 2500×2500, $\mathbb{R}^2$, the Absolute Exponential Kernel Function, Random Coordinates

	Threshold Value		
	1e−2	1e−3	1e−4
Chol without permutation	72,785	186,399	346,231
Chol full permutation	57,514	140,989	254,992
Chol partial permutation (k = 120)	60,513	153,332	284,262

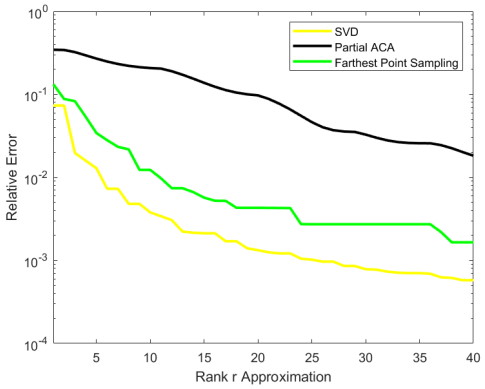
As we can see from Table 6.1, selecting the rows (columns) based on the FPS will decrease the number of large elements in the Cholesky factor, which intuitively suggests that this method can improve the approximation of the pivoted Cholesky factorization. Moreover, if the threshold value is set to be 1e−2 or 1e−3, the partial permutation also performs well. The result improves the feasibility of the FPS method because the cost to reorder all data points is prohibitive.

There are several additional findings of partial permutation worth mentioning. First, from subtables (a) to (c), we see that the performance of the partial permutation remains as good as the full permutation in a fixed number of steps even if the size of the dataset increases. This result implies that in a low dimension, we could avoid cubic computational cost by not selecting the inducing points proportional to the size of the dataset. Second, both partial permutation and full permutation lead to a huge decrease in the number of large values if the data points are on a grid. The grid domain is more regular and well-shaped for the FPS to capture the geometric information.

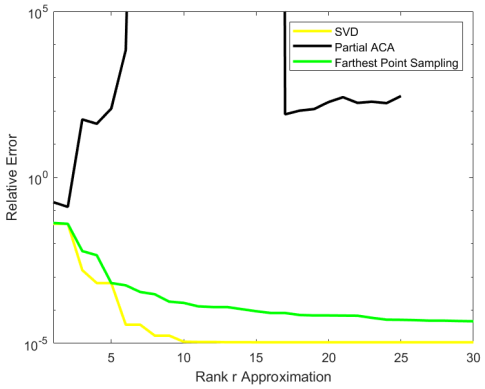
Table 6.1 also presents a limitation of the FPS method. As the dimension of the dataset increases, the full sampling maintains a good performance while the effect of the partial sampling weakens. This finding corresponds to what we stated in Chapter 4: in higher dimensions, it is harder for the FPS to capture the geometric information because the method tends to first select a large number of data points on the boundary [12]. Therefore, we may need to expand the set of selected rows to improve the performance.

6.2 Low-rank Approximation Comparison

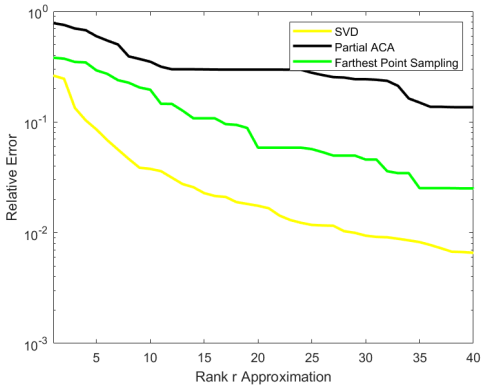
We then compare the performance of the low-rank approximation between our method, the ACA with partial pivoting, and the SVD. In this test, we generate the absolute exponential covariance matrix and the squared exponential covariance matrix from 100 grid points and 100 random coordinates, respectively. From Figure 6.1, we find that the FPS Cholesky factorization outperforms the ACA with partial pivoting in terms of both accuracy and numerical stability. A potential reason is that the FPS Cholesky factorization utilizes the geometric information from the dataset while the ACA algorithm is purely algebraic.



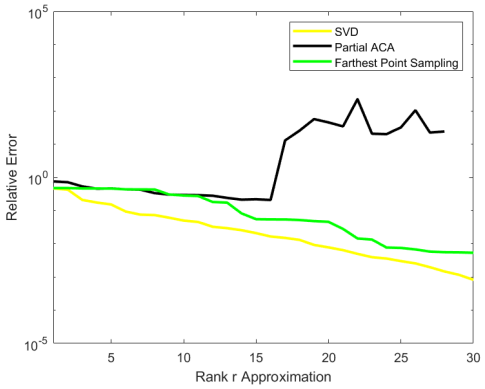
(a) Absolute Exponential - Points on a Grid



(b) Squared Exponential - Points on a Grid



(c) Absolute Exponential - Random Coordinates



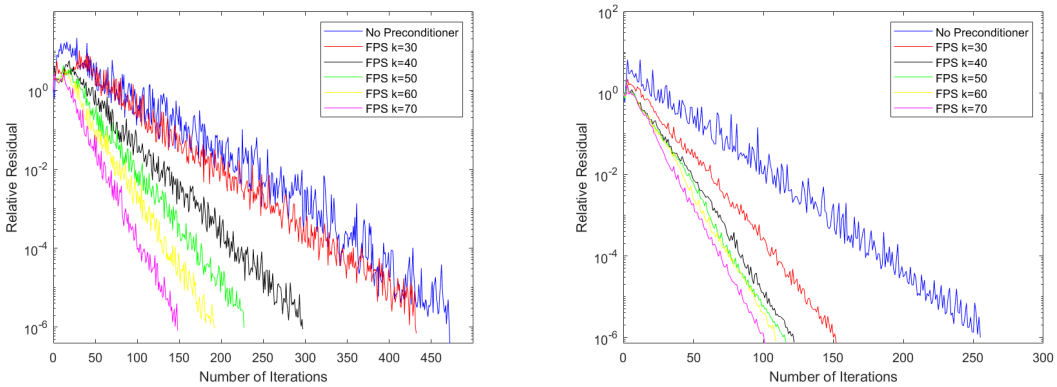
(d) Squared Exponential - Random Coordinates

Figure 6.1 Comparison of Low-rank Approximation

6.3 Solving Linear Systems and Related Applications

Now we apply our method to solve some linear systems associated with the squared exponential covariance matrix and the absolute exponential covariance matrix. First, we generate a 2500×2500 covariance matrix from data points on the grid $[0, 50] \times [0, 50]$. Based on the number of inducing points k, we construct the preconditioners using the FPS Cholesky factorization. Then, we compare the number of CG iterations with preconditioners applied with the number of iterations without preconditioners applied. The tolerance for the relative residual is set to be $1e{-}6$, and the initial guess for the solution is the zero vector. From Figure 6.2, we see that for both covariance matrices, our preconditioners decrease the number of CG iterations. For the squared exponential matrix, selecting 70 out of 2500 data points will lead to one-third number of iterations. For the absolute exponential covariance matrix, 40 inducing points help us achieve half the number of iterations.

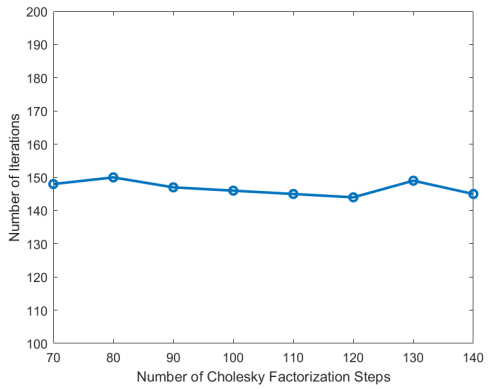
We also investigate the number of CG iterations when the number of pivoted Cholesky steps exceeds the number of inducing points. In Figure 6.3, as we do not see any further improvement in the convergence rate, we conclude that the main contribution of the faster convergence in CG is made by the FPS method.



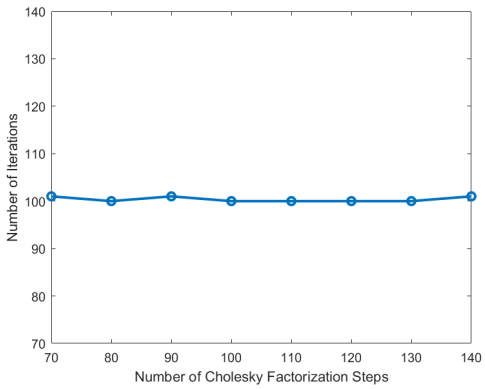
(a) Squared Exponential Covariance Matrix

(b) Absolute Exponential Covariance Matrix

Figure 6.2 Comparison of Number of Iterations



(a) Squared Exponential Covariance Matrix



(b) Absolute Covariance Matrix with smooth

Figure 6.3 Convergence Rate When the Number of Cholesky Steps Exceeds the Number of Inducing Points ($k = 70$)

Finally, we apply our methods to the Gaussian process predictions, which include a step of solving linear systems. First, we generate a Gaussian process sample on a regular $[0, 100] \times [0, 100]$ grid using the squared exponential kernel $k(x, y) = \sigma^2 \exp(\frac{-\|x-y\|^2}{l^2})$ where σ^2 is the signal covariance parameter and l is the length parameter. We fix the signal variance $\sigma^2 = 1$ and attempt to tune the length parameter l with the true hyper-parameter $l = 7$. The dataset is split into a training part and a testing part. Our goal is to utilize the training dataset to tune the optimal length parameter and perform predictions for the testing dataset based on that length parameter. To select the length parameter that fits the data well, we need to calculate the log-likelihood functions for each potential parameter and choose the maximum among them. This process is also called the Maximum Likelihood Estimation, and it requires solving several linear systems. The formula of the log-likelihood function is displayed below:

$$L = -\frac{1}{2}(\mathbf{y_{trn}} - \mu)^{\mathbf{T}}\mathbf{K^{-1}}(\mathbf{y_{trn}} - \mu) - \frac{\mathbf{1}}{\mathbf{2}}\log\det(\mathbf{K}) - \frac{\mathbf{n}}{\mathbf{2}}\log \mathbf{2}\pi \tag{6.1}$$

where $\boldsymbol{K}$ is the covariance matrix parameterized by a potential length parameter l, $\mathbf{y_{trn}}$ is the training sample, and μ is the mean which we assumed to be $\mathbf{0}$.

Since directly calculating $\boldsymbol{K}^{-1}$ is very unstable, we would like to obtain the first part of Equation 6.1 by solving $\boldsymbol{K}\boldsymbol{x} = \mathbf{y_{trn}}$. We will make the prediction about testing values

using the length parameter l we choose:

$$\mathbf{y_{pred}} = \mathbf{K}(\mathbf{X}_*, \mathbf{X}, \mathbf{l})\mathbf{K}(\mathbf{X}, \mathbf{X}, \mathbf{l})^{-1}\mathbf{y_{trn}} \tag{6.2}$$

where $\boldsymbol{K}(X_*, X, l)$ measures the covariance between testing dataset X_* and training dataset X given our calculated l and $\boldsymbol{K}(X, X, l)$ is the covariance matrix of training data points given our calculated l.

Figure 6.4 provides the visualization of our Gaussian process random field. We randomly select 30% of the data points as our training dataset displayed in Figure 6.5.

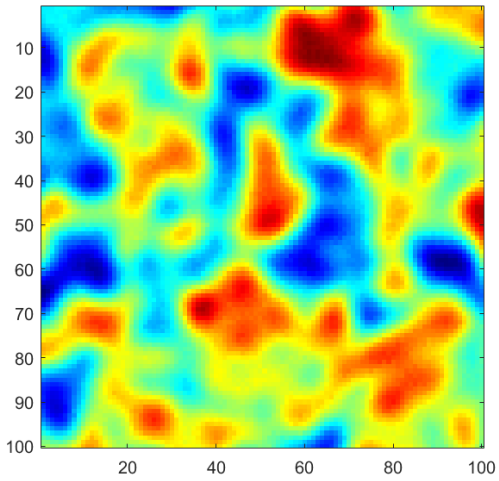


Figure 6.4 Gaussian Random Field

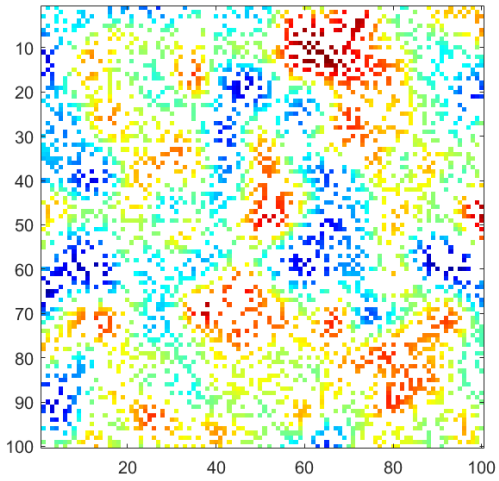


Figure 6.5 Training Data

Assume we know the interval for tuning l is $[5, 9]$ and we pick nine evenly-spaced points inside the interval. Then we have to solve nine linear systems associated with 3000×3000 covariance matrices. In this experiment, we select 50 inducing data points out of 3000 using the FPS and constructed the preconditioners by the pivoted Cholesky factorization. Notice that we only need to conduct the selection process once because this set of inducing points works for all nine linear systems.

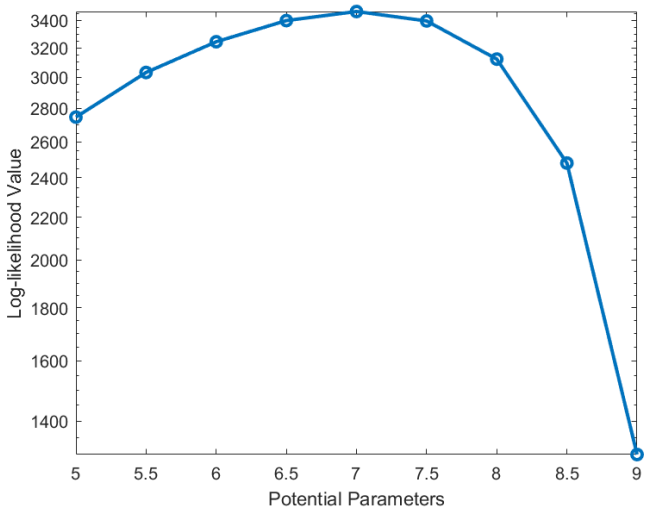


Figure 6.6 Log-likelihood Function

Figure 6.6 shows the log-likelihood plot after we compute the log-likelihood for nine potential length parameters. We obtain the true parameter $l = 7$ as the log-likelihood function reaches its maximal value at this point. Lastly, we use the estimated length parameters to make predictions for the testing data points and generate the Gaussian random field shown in Figure 6.7. The relative error is 0.031. As we can see, the random field is well recovered by Gaussian process predictions.

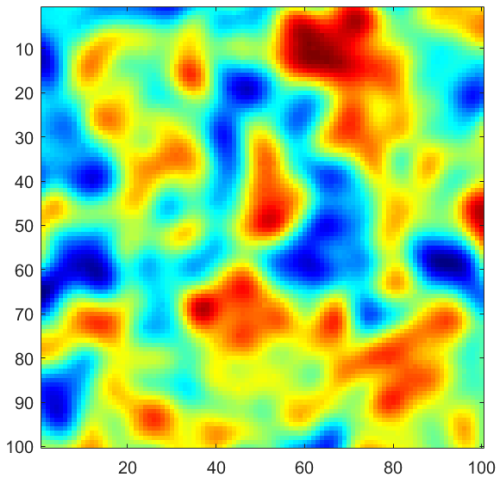


Figure 6.7 Prediction

Chapter 7

Conclusions

In this thesis, we propose an efficient method to solve the linear system associated with the dense covariance matrix. Motivated by the geometric information of the data points from the Gaussian process and the low-rank approximation idea, we use the Farthest Point Sampling Cholesky factorization to construct the preconditioner for the conjugate gradient iterative method. We then validate the effectiveness of our method through both theoretical analysis and numerical experiments, including comparing with the ACA method, analyzing the number of iterations when solving linear systems, and applying it to make predictions on a synthetic Gaussian process dataset. In the future, we will work on deriving the low-approximation error bound and the conditional number bound for our method when the data points from the Gaussian process are in discrete settings. Moreover, we hope to propose an efficient implementation of Farthest Point Sampling Cholesky factorization in high dimensions.

References

(1) Rasmussen, C. E.; Williams, C. K. I., *Gaussian Processes for Machine Learning*; The MIT Press: 2005.

(2) Saad, Y., *Iterative methods for sparse linear systems*; SIAM: 2003.

(3) Kolotilina, L. Y.; Yeremin, A. Y. Factorized Sparse Approximate Inverse Preconditionings I. Theory, 1993.

(4) Petersen, K. B.; Pedersen, M. S. The Matrix Cookbook, Version 20121115, 2012.

(5) Chow, E.; Saad, Y. Preconditioned Krylov Subspace Methods for Sampling Multivariate Gaussian Distributions, 2014.

(6) Roughgarden, T.; Valiant, G. CS168: The Modern Algorithmic Toolbox Lecture #9: The Singular Value Decomposition (SVD) and Low-Rank Matrix Approximations, en.

(7) Williams, C.; Seeger, M. In *Advances in Neural Information Processing Systems*, ed. by Leen, T.; Dietterich, T.; Tresp, V., MIT Press: 2000; Vol. 13.

(8) Börm, S.; Grasedyck, L.; Hackbusch, W. Hierarchical Matrices, 2003.

(9) Schäfer, F.; Sullivan, T. J.; Owhadi, H. Compression, inversion, and approximate PCA of dense kernel matrices at near-linear computational complexity, 2017.

(10) Eldar, Y.; Lindenbaum, M.; Porat, M.; Zeevi, Y. The farthest point strategy for progressive image sampling, Conference Name: IEEE Transactions on Image Processing, 1997.

(11) The Stanford 3D Scanning Repository, 1994.

(12) Cai, D.; Nagy, J.; Xi, Y. Fast deterministic approximation of symmetric indefinite kernel matrices with high dimensional datasets, 2021.

(13) Wendland, H., *Scattered Data Approximation*; Cambridge University Press: 2004.

(14) Aurenhammer, F. Voronoi diagrams–a survey of a fundamental geometric data structure, 1991.

(15) Müller, S. Komplexität und Stabilität von kernbasierten Rekonstruktionsmethoden, ger, Accepted: 2009-03-19T15:27:22Z, 2009.