Effective Solutions of Cosmic Microwave Background Problem

JiaJin Zhao B.E.

A Dissertation

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science (Future Networked Systems)

Supervisor: Simon Wilson

September 2020

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

JiaJin Zhao

September 3, 2020

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

JiaJin Zhao

September 3, 2020

Acknowledgments

I would like to express my gratitude to my supervisor Professor Simon Wilson for all of his help with this study. Special thanks to Professor Kirk Soodhalter for his advice and guidance in this project.

JiaJin Zhao

University of Dublin, Trinity College
September 2020

Effective Solutions of Cosmic Microwave Background Problem

JiaJin Zhao , Master of Science in Computer Science

University of Dublin, Trinity College, 2020

Supervisor: Simon Wilson

Cosmic Microwave Background (CMB) can reveal information about the early stages of the universe. The aim of the research paper is to provide efficient CMB source separation algorithms with high accuracy.

This research paper explores two algorithms for the CMB source separation problem, the conjugate gradient algorithm and the Sylvester equation algorithm. Experiments based on Planck data have been carried out to test the performance of these algorithms. The conjugate gradient has good performance and high accuracy, the Sylvester equation algorithm has the best performance but lower accuracy.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

The Study Cosmic Microwave Background (CMB) can reveal information about the early stages of the universe, and the source separation algorithm is a classic problem in the study of CMB Observations (1). The CMB is mixture of galactic foregrounds and point sources (2), and the CMB source separation is a process to reconstruct the CMB signal by separating it from other sources such as synchrotron, galactic dust and free-free emission (3). More data is available recently for research due to space missions like COBE, WMAP, and Planck, which provide data of the sky at many frequency channels. However, it remains a problem that how to perform CMB source separation effectively under limited resources because of the large dimension of these data.

1.2 Research Question

In the context of source separation algorithms(3) for CMB, We must solve for x the following linear system: (3)

$$(Q + B^T C B)x = B^T C y \quad \text{with} \quad (Q + B^T C B) \in \mathbb{R}^{nN \times nN} \tag{1.1}$$

where $n \in \{4, 5\}$ and represents the number of sources, and $N = 12 \times N_{side}^2$ where N_{side} is the N_{side} value of fits data. And the value of N_{side} is 512 and 1024 for WMAP and Planck data respectively. So, $N = \mathcal{O}(10^6)$ for WMAP, and $N = \mathcal{O}(10^7)$ for Planck.

The matrix $Q \in \mathbb{R}^{nN \times nN}$ is a block diagonal matrix with each block being of the form $q_i D^T D = q_i D^2$ and can be written as the Kronecker product $Q = P \otimes D^2$ where $P =$

$diag\{p_1, p_2, \dots, p_n\}$ and $D \in \mathbb{R}^{N \times N}$. P is a diagonal matrix and D is a symmetric adjacency matrix that encodes a nearest neighbor coupling of nodes from the discretization of the sky, we can define D_{ij} as

$$D_{ij} = \begin{cases} 1 & \text{if } i \neq j \text{ and } i, j \text{ are neighbors} \\ 0 & \text{if } i \neq j \text{ and } i, j \text{ are not neighbors} \\ -\sum_{\substack{\rho=1 \\ \rho \neq j}}^{N} D_{j\rho} & \text{if i = j} \end{cases} \tag{1.2}$$

The matrix $B \in \mathbb{R}^{mN \times nN}$ with $m = 9$ can be written as the Kronecker product $B = A \otimes I_N$ where $A \in \mathbb{R}^{m \times n}$ and I_N is an $N \times N$ identify matrix, the value of m represents the number of frequency channels. The matrix $C \in \mathbb{R}^{mN \times mN}$ is a diagonal matrix with all positive entries and can be written as the Kronecker product of two diagonal matrices.

$$C = diag\{\tau_1, \tau_2, \dots, \tau_m\} \otimes \{n_1, n_2, \dots, n_N\} = T \otimes N \tag{1.3}$$

The vector y in constructed from Planck or WAP full sky maps, and the vector x is the unknown vector.

In this paper, we proposed two different approaches for CMB source separation. The first approach is an iterative solution based on the conjugate gradient; it has two different variants and can achieve decent accuracy. The second approach is based on the Sylvester equation, it is faster but has a lower accuracy than the conjugate gradient solution. Each will be analyzed in terms of resource usage, performance, and accuracy.

1.3 Dissertation Structure

This dissertation is organized as follows:

- Chapter 1 is the Introduction which gives an introduction to the application problem. It gives a brief background and then describes the research question.
- Chapter 2 is the State of the Art which briefly introduces the existing approaches related to the research question.
- Chapter 3 is the Methodology which describes the approaches to solve the research question. Two different approaches will be discussed.
- Chapter 4 is the Implementation which presents the implementations of these two different approaches, and the performance and resource usage of these approaches will be discussed.

- Chapter 5 is the Results which compares the outcomes of these two approaches. Planck full sky maps are used to construct the input of these two implementations, and their results will be compared in terms of accuracy of resource usage.
- Chapter 6 is the Conclusion and Future Work which summarizes the project and discusses challenges and future works.

Chapter 2

Literature Review

In this chapter, we present and discuss related work in CMB source separation. In section 2.1, several existing source separation approaches will be presented. In section 2.2, we explore some useful mathematical background.

2.1 CMB Source Separation

Cosmic Microwave Background, discovered in 1965 by Penzias and Wilson, is a relic radiation emitted some 13 billion years ago, when the universe was about 370 000 years old (4). Source separation is a process of isolating the emission from all the other components present in the data (1). This technique is used in the cosmology community for the separation of cosmic microwave background. The challenge is how to separate a high-precision CMB map featuring low noise and low foreground contamination.

Several approaches have been proposed. Guillaume et al. present and discuss the application of blind source separation to astrophysical data obtained with the WMAP satellite, Blind source separation permits to identify and isolate a component compatible with the Cosmic Microwave Background, and to measure both its spatial power spectrum and its emission law (5). Wagner-Carena et al. present a hierarchical generalized morphological component analysis, they carry out their experiments on $N_{side} = 256$ simulated sky maps and find equivalent or improved performance when compared to state-of-the-art internal linear combination type algorithms (6).

Wilson et al. developed a fully Bayesian source separation technique that assumes a very flexible model for the sources, namely the Gaussian mixture model with an unknown number of factors, and utilize Markov chain Monte Carlo techniques for model parameter estimation (7). The algorithm incorporates a rich variety of prior information available

in this problem in contrast to most of the previous work which assumes completely blind separation of the sources, it performs well on simulated Planck data. Then, they also introduces a functional approximation to implement Bayesian source separation analysis and it is applied to separation of the Cosmic Microwave Background using WMAP data (3). Bobin et al. introduced a sparsity-based component separation method coined local-generalized morphological component analysis. Their expirments show the high efficiency of the proposed component separation methods for estimating a clean CMB map with a very low foreground contamination (2).

2.2 Mathematical Background

In this section, we discuss some useful mathematical background in this paper.

The equation 1.1 has the form $Ax = b$. And there are several approaches to solve a linear system, one of them is conjugate gradient. Conjugate gradient is an iterative method to solve a linear system. An estimated solution is generated at each iteration, which is an improvement over the one in the preceding iteration, it gives the solution in n steps if no round-off error occurs, n is the number of equations in the linear system (8).

However, the matrix A in the equation 1.1 has large dimension, and it should never be explicitly built in our implementation. Luckily, we can use Kronecker product properties in equation 1.1 to generate an effective way to for fast linear transforms (9). Besides, we can also use Kronecker product properties here to transform equation 1.1 to a Sylvester equation, and there are several approaches to solve a Sylvester equation (10).

Chapter 3

Methodology

Because of the large scale of this problem and its sparse, Kronecker structure properties. Equation 1.1 should not be solved directly by a matrix decomposition method. Instead we propose treating this problem using iterative methods built on efficient implementations of matrix-vector products (matvecs) for the matrices from this problem or as the approximation to the solution of a Sylvester matrix equation. Furthermore, most of the system matrix should never be explicitly constructed. Rather, it should only be represented as a stored procedure implementing matrix-vector product. The only matrice explicitly represented should be A, T, and P.

In this chapter, two different method to solve x will be discussed.

3.1 Efficient matvecs

We describe first how to perform the matrix-vector product for each part of the full system matrix and then we finish by combining them, yielding a routine for $x \to (Q + B^T CB)x$. This will be necessary for using an appropriate iterative method whose core operation is this matvec routine.

3.1.1 Matvec routine for $v \to Dv$

Algorithm 1: Matvec procedure $v \to Dv$

input : vector $v \in \mathbb{R}^N$.

output: Dv

for $i \leftarrow 1$ **to** N **do**

  $v(j) \leftarrow v(j_{left}) + v(j_{right}) + v(j_{above}) + v(j_{below}) - (\sum_{\substack{\rho=1 \\ \rho \neq j}}^{N} D_{j\rho})v(j)$

end

Observe that since D is a sparse matrix encoding the nearest-neighbor coupling for each node, the routine $v \to Dv$ involves updating entry j of v (associated to node j of grid) using its four nearest neighbors and the value at the node itself. Thus a sequential matrix-free routine can be easily formulated. For Algorithm 1, we note that since the matrix D does not change, we can pre-compute and store the rowsums $\sum_{\substack{\rho=1 \\ \rho \neq j}}^{N} D_{j\rho}$. Also, it should be noted that Algorithm 1 is parallelizable, as the grid can be divided up and sent to different processes or threads, and communication between processors or threads only needs to occur for neighboring nodes which are on different processors. The computational cost of a serial version of Algorithm 1 is $\mathcal{O}(N)$, since we do 4 floating point operations per entry of v.

3.1.2 Matvec routine for $u \to Qu$

As already discussed, Q is a block diagonal matrix with blocks $q_i D^2$. Thus, it can be represented as a Kronecker product, namely

$$Q = \begin{bmatrix} q_1 & & & \\ & q_2 & & \\ & & \ddots & \\ & & & q_n \end{bmatrix} \otimes D^2 = P \otimes D^2 \tag{3.1}$$

Thus, for $u = w \otimes v$ with $w \in \mathbb{R}^n$, we can build a simple matvec algorithm on top of Algorithm 1 to calculate $u \to Qu$, which shown as Algorithm 2. Note that we simply store w and v. There is no reason to compute the Kronecker product explicitly. The computational cost of Algorithm 2 is also $\mathcal{O}(N)$ since the dominant cost is two applications of Algorithm 1.

Algorithm 2: Matvec procedure $u \to Qu$ Kronecker style

input : vector $u = w \otimes v \in \mathbb{R}^{nN}$ with $w \in \mathbb{R}^n$ and $v \in \mathbb{R}^N$

output: Qu

1 $w \leftarrow Pw$

2 $v \leftarrow Dv$; $v \leftarrow Dv$

Observe that we made an assumption that we have a Kronecker representation of $u = w \otimes v$, which we may not know nor wish to compute. However, if we do not know this representation, we can perform this matrix-vector product by reshaping u.

Definition 3.1.1. Let $u \in \mathbb{R}^{nN}$. We use the Matlab-style notation $\mathit{reshape}(u, N, n) \in \mathbb{R}^{N \times n}$ to denote the matrix obtained from u by arranging every N entries of u as a column

of the resulting matrix; i.e.,

$$\text{if } u = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix} \text{ then } \mathit{reshape}(u, N, n) = \begin{bmatrix} u_1 & u_2 & \dots & u_n \end{bmatrix}$$

Let $U = \mathit{reshape}(u, N, n)$ for ease of notation. Then the matvec Qu can also be represented by reshaping u and computing D^2UP. The result can then be vectorized, which is simply the inverse of the reshaping operation; i.e., u = vectorize(U). We present this matvec procedure as Algorithm 3.

Algorithm 3: Matvec procedure $u \to Qu$ using reshaping

input : vector $u \in \mathbb{R}^{nN}$, matrix $P \in \mathbb{R}^{n \times n}$ and $D \in \mathbb{R}^{N \times N}$

output: Qu

1 Set $U = \mathit{reshape}(u, N, n)$

2 $U \leftarrow D^2UP$

3 $u \leftarrow \mathit{vectorize}(U)$

If P or D were full matrices, this would not be efficient to carry out, but since P is diagonal and D is sparse, it is worth considering.Furthermore, we will return this idea when presenting our second method of solving the linear system in section 3.3.

3.1.3 Matvec routine for $x \to B^T CBx$

we can take advantage of the Kronecker structures of B and C. We can write

$$B^T CBx = (A^T \otimes I_N)(T \otimes N)(A \otimes I_N)x \tag{3.2}$$

If x has the known Kronecker structure $x = w_x \otimes v_x$, with $w_x \in \mathbb{R}^n$ and $v_x \in \mathbb{R}^N$, then we compute equation 3.2 using Algorithm 4

Algorithm 4: Matvec procedure $x \to B^T CBx$ Kronecker style

input : vector $x = w_x \otimes v_x \in \mathbb{R}^{nN}$ with $w_x \in \mathbb{R}^n$, $v_x \in \mathbb{R}^N$

output: $B^T CBx$

1 $w_x \leftarrow Aw_x$

2 $w_x \leftarrow Tw_x$; $v_x \leftarrow Nv_x$

3 $w_x \leftarrow A^T w_x$

4 $x \leftarrow w_x \otimes v_x$

If we set X = reshape(x, N, n), then we can use the same trick as before. Then Algorithm 5 can be used to efficiently compute the matvec.

Algorithm 5: Matvec procedure $x \to B^T CBx$ using reshaping

input : vector $x \in \mathbb{R}^{nN}$, matrix $A \in \mathbb{R}^{m \times n}$, $N \in \mathbb{R}^{N \times N}$ and $T \in \mathbb{R}^{m \times m}$

output: $B^T CBx$

1 Set $X = reshape(x, N, n)$

2 $X \leftarrow XA^T$

3 $X \leftarrow NXT$

4 $X \leftarrow XA$

5 $x \leftarrow vectorize(X)$

3.2 Conjugate Gradient Method

The first strategy we propose entails considering the linear system 1.1 as it is presented with the coefficient matrix being built from and applied using the Kronecker product formulation. We propose to use a so-called "matrix-free" method which only requires a stored procedure to perform $x \to (Q + B^T CB)x$, meaning we need no explicit representation of the coefficient matrix. We describe with brevity how such a method works.

Given a general linear system

$$Gx = b \text{ with } G \in \mathbb{R}^{M \times M} \text{ and } b \in \mathbb{R}^M \tag{3.3}$$

where $M \gg 0$ is large and G is sparse, a general strategy of discretization can be used to exchange the solution of this large problem for the solution of a much smaller, easier-to-handle linear system. Given a basis of vectors $\{v_1, v_2, \dots, v_j\}$ spanning a subspace $\mathcal{V}$, we approximate $x \approx x_j = \sum_{i=1}^{j} z_j v_j \in \mathcal{V}$, and we determine x_j by enforcing that it satisfy

$$v_i^T Gx_j = v_i^T b \text{ for all } 1 \leq i \leq j$$

If we let $H \in \mathbb{R}^{j \times j}$ be the matrix whose $(i,k)^{th}$ is $v_i G v_k$, $z_j \in \mathbb{R}^j$ be the vector whose entries are the unknown coefficients z_i , and $s \in \mathbb{R}^j$ be the vector whose entries are $v_i^T b$, then solving for z_j is equivalent to solving the small $j \times j$ linear system

$$Hz = s$$

Thus, we have exchanged solving a large $M \times M$ problem for solving a small $j \times j$ problem, and we must never explicitly store G in memory (we simply need to apply it to every

basis vector v_i). One of the most common and effective choices for $\mathcal{V}$ is the Krylov subspace, whereby the construction of the basis entails one matvec per iteration which also means that H is generated automatically during basis construction. We briefly review standard Krylov subspace methods and introduce some terminology. For a more detailed description, see, e.g.(11, 12) and the references therein. Consider solving

$$G(x_0 + t) = b \tag{3.4}$$

where $G \in \mathbb{C}^{n\times n}$, $x_0 \in \mathbb{C}^{n\times n}$ is an initial approximation, and t is the unknown correction. Let $r_0 = b - Gx_0$ be the initial residual. In iteration j, a Krylov method selects a correction t_j from the jth Krylov subspace generated by G and r_0,

$$t_j \in \mathcal{K}_j(G, r_0) = span\{r_0, Gr_0, G^2r_0, \dots, G^{j-1}r_0\} \tag{3.5}$$

i.e., from the space of polynomials of degree less than j in G acting on r_0 . The constraint space defines the Krylov subspace method up to implementation choices and must also be compatible with the correction space such that the method being implicitly defined can be efficiently implemented.

For coefficient matrices G that are symmetric positive definite (such as that in 1.1), the celebrated, efficient, and stable method of Conjugate Gradients is the Krylov subspace method of choice. Implicitly, at iteration j, the method has generated the subspace $\mathcal{K}_j(G, r_0)$ and computes $t_j \in \mathcal{K}_j(G, r_0)$ such that the approximation $x_j = x_0 + t_j$ minimizes

$$\|b - Gx_j\|_G = (b - Gx_j)^T G(b - Gx_j) \tag{3.6}$$

The actual derivation of the method can be found in, e.g., (11). We present it for the

case $G = (Q + B^T CB)$ and right-hand side $b = B^T Cy$ as Algorithm 6.

Algorithm 6: Method of Conjugate Gradients

input : vector y, matrix B,C, and Q defined as above , and x_0 as the initial value of the solution; $\epsilon > 0$ a convergence tolerance.

output: An approximation solution of vector x.

$x = x_0$; $r = B^T Cy - (Q + B^T CB)x$; $p = r_0$

while $\|r\| > \epsilon\|r_0\|$ **do**

  $r_{old} \leftarrow r$

  $q \leftarrow (Q + B^T CB)p$

  $\alpha \leftarrow \frac{r^T r}{p^T q}$

  $x \leftarrow x + \alpha p$

  $r \leftarrow r - \alpha q$

  $\beta \leftarrow \frac{r^T r}{r_{old}^T r_{old}}$

  $p \leftarrow r + \beta p$

end

3.2.1 Advantages of Conjugate Gradients method

- Coefficient matrix of equation 1.1 does not need to be constructed (and there is no expensive decomposition).
- Convergence speed of Conjugate Gradients is determined by the eigenvalues of the coefficient matrix.
- If convergence speed is too slow, preconditioning techniques can be used to implicitly produce a system with better eigenvalue distribution to accelerate convergence.
- Amenable to parallelization by distributing sky domain grid onto different processors and parallelizing matvec procedures.

3.2.2 Disadvantages of Conjugate Gradients method

- We still must deal with vectors of size nN.
- We cannot take advantage of any structure or compressibility which may be present in y or rather in $reshape(y, N, n)$.
- Applying an iterative method directly to the tensor-structured problem means the performance of the method will be hampered by the large condition number of the full-size problem.

3.3 Sylvester Equation Method

Using the $\mathit{reshape}(\cdot,\cdot,\cdot)$ operation, we can reinterpret equation 1.1 as a Sylvester matrix equation, which will allow us to take advantage of additional structure in y that may be available. Reshaping the solution ($M = \mathit{reshape}(x, N, n)$) and ($Y = \mathit{reshape}(y, N, m)$). With these reshapings, we can rewrite equation 1.1 as the generalize Sylvester equation

$$D^2MP + NMA^TTA = NYTA$$

As N and P are diagonal and invertible, we can easily multiply on the left by N^{-1} and on the right by P^{-1}, yielding the Sylvester equation

$$N^{-1}D^2M + MA^TTAP^{-1} = YTAP^{-1} \tag{3.7}$$

A survey by Simoncini covering a variety of solution techniques was recently published (13). Since $A^TYAP^{-1} \in \mathbb{R}^{n\times n}$, this falls into a category of Sylvester equations with one large coefficient matrix ($N^{-1}D^2$) and one small one (A^TYAP^{-1}). Such problems are amenable to solution by generating a block Krylov subspace associated to the large coefficient matrix, i.e., $N^{-1}D^2$. This space at iteration j has the form

$$\begin{aligned}\mathbb{K}_j(N^{-1}D^2, YTAP^{-1}) :=& \mathcal{K}_j(N^{-1}D^2, YTAe_1) \\ &+ \mathcal{K}_j(N^{-1}D^2, YTAe_2) + ... \\ &+ \mathcal{K}_j(N^{-1}D^2, YTAe_m)\end{aligned}$$

where e_i is the ith Cartesian basis vector. The Sylvester equation is projected onto this space (through a process of discretization, as before), and a smaller Sylvester problem is solved at each iteration. The matrix $N^{-1}D^2$ is no longer symmetric positive definite in the standard inner product, but it is in a different inner product.

Any symmetric positive-definite matrix can be used to define a new inner product. The standard inner product $(x, y) = y^Tx$ can be thought of as being induced by the identity matrix; i.e., $y^Tx = y^TIx$. Any other symmetric, positive-definite matrix can take the place of the identity matrix to create a new inner product. The matrix N is a diagonal matrix with only positive entries on its diagonal; therefore, it is a positive-definite matrix and can be used to induce the inner product $(x, y)_N = y^TNx$. Formally, symmetry of a matrix is defined with respect to an inner product. We have that a matrix G satisfies $G^T = G$ if and only if $(Gx, y) = (x, Gy)$. In other words $y^TGx = x^TGy$ for all $x, y \in \mathbb{R}^n$. Similarly, a matrix G is symmetric with respect to $(\cdot,\cdot)_N$ if for all $x, y \in \mathbb{R}^n$ $(Gx, y)_N = (x, Gy)_N$; i.e., $y^TNGx = y^TG^TNx$.

What is important here is that we exploit that

Lemma 3.3.1. *The coefficient matrix $N^{-1}D^2$ is symmetric with respect to $(\cdot,\cdot)_N$.*

Proof. This is straightforward to show by direct computation. For all $x, y \in \mathbb{R}^n$, we have

$$(N^{-1}D^2x.y)_N = y^T NN^{-1}D^2x = y^T D^2x = y^T D^2N^{-1}Nx = (x, N^{-1}D^2y)_N$$

Thus the symmetry with respect to $(\cdot,\cdot)_N$ is proven. □

Iterative methods designed for symmetric matrices can be converted to methods for matrices symmetric with respect to any inner product simply by using that inner product in the algorithm. We can implement a Conjugate Gradient-like iteration which uses a fixed amount of memory, is amenable to parallelization, and can be implemented to have good data movement characteristics.

3.3.1 The Lanczos process

We mentioned in Section 3.2 that CG is a Krylov subspace method. However, we avoided describing any of the details about how this subspace is used to derive CG for brevity. We elaborate here on some components of Krylov subspaces for symmetric coefficient matrices to derive an iterative method for Sylvester equations whose larger coefficient matrix (here $N^{-1}D2$) is symmetric with respect to an inner product.

Let us denote $\hat{Y} = YTAP^{-1}$ and $\hat{S} = A^T TAP^{-1}$. The symmetric Lanczos process is an iterative procedure to generate an orthonormal basis for $\mathbb{K}_j(N^{-1}D^2, \hat{Y})$ block-by-block. At step 1, we get a basis for $\mathbb{K}_2(N^{-1}D2, \hat{Y})$. At step 2, we add a new vector to get $\mathbb{K}_3(N^{-1}D2, \hat{Y})$, and in general at step i, we add to the previous basis to get a basis for $\mathbb{K}_{i+1}(N^{-1}D^2, \hat{Y})$. What has been shown is that if the matrix is symmetric with respect to the inner product used for orthogonalization, at step i, the $(i+1)$st block of vectors generated at that step are already orthogonal with respect to all but blocks i and $i-1$, meaning the orthogonalization procedure requires we only store the two most recently generated blocks. At step i, we have generated the orthonormal basis $\{V_1, V_2, \ldots, V_i\}$ with $V_\rho \in \mathbb{R}^{N\times n}$, $V_\rho^T V_k = 0$ for $\rho \neq k$, and $V_\rho^T V_\rho = I_n$, the $n \times n$ identity matrix. To obtain the next basis vector, we compute $W_{i+1} = (N^{-1}D^2)V_i$. Normally, we would then need to orthogonalize these vectors against all previous, but since $N^{-1}D^2$ is symmetric with respect to $(\cdot,\cdot)_N$, it has been proven that $V_\ell^T NW_{i+1} = 0 for \ell < i-1$. The orthogonalization coefficients are stored in the matrix $\underline{T}_j \in \mathbb{R}^{(j+1)n,jn}$ whose first jn

rows are the symmetric matrix $T_m \in \mathbb{R}^{jn \times jn}$. These have the structure

$$\underline{T}_j = \begin{bmatrix} H_1 & B_2 & & & \\ B_2 & H_2 & B_3 & & \\ & B_3 & \ddots & \ddots & \\ & & \ddots & H_{j-1} & B_j \\ & & & B_j & H_j \\ & & & & B_{j+1} \end{bmatrix} \text{ and } T_j = \begin{bmatrix} H_1 & B_2 & & & \\ B_2 & H_2 & B_3 & & \\ & B_3 & \ddots & \ddots & \\ & & \ddots & H_{j-1} & B_j \\ & & & B_j & H_j \end{bmatrix},$$

where each block represents an $n \times n$ matrix. The diagonal and super-diagonal blocks represent coefficients are related to orthogonalization and the subdiagonal blocks represent blocks related to normalization. The symmetry tells us that the normalization block B_{i+1} , generated when normalizing V_{i+1} , can be reused as an orthogonalization coefficient at step $i + 1$, where it can be shown that

$$V_{i+1}B_{i+1} = N^{-1}D^2V_i - V_iH_i - V_{i-1}B_i \tag{3.8}$$

In terms of the matirx T_j , we can use 3.8 to derive the Lanczos relation, relating $N^{-1}D^2$, $\mathcal{V}_j$, and $\underline{T}_j$,

$$N^{-1}D^2\mathcal{V}_j = \mathcal{V}_{j+1}\underline{T}_j \tag{3.9}$$

The construction of $\underline{T}_j$ is shown as a part of Algorithm 7. If we take these block basis vectors and put them as columns of a matrix $\mathcal{V}_j = [V1, V2, \dots, Vj] \in \mathbb{R}^{N \times jn}$, then we have the symmetric Lanczos relation $N^{-1}D^2\mathcal{V}_j = \mathcal{V}_{j+1}\underline{T}_j$. Furthermore, since the columns of $\mathcal{V}_j$ form an N orthonormal basis for $\mathbb{K}_j(N^{-1}D2, \hat{Y})$, we have that $\mathcal{V}_j^T N\mathcal{V}_{j+1} = [I_j 0]$, and one can then show that,

$$\begin{aligned} N^{-1}D^2\mathcal{V}_j &= \mathcal{V}_{j+1}\underline{T}_j \\ \mathcal{V}_j^T NN^{-1}D^2\mathcal{V}_j &= \mathcal{V}_j^T N\mathcal{V}_{j+1}\underline{T}_j \\ \mathcal{V}_j^T D^2\mathcal{V}_j &= \begin{bmatrix} I_j & 0 \end{bmatrix} \underline{T}_j \\ \mathcal{V}_j^T D^2\mathcal{V}_j &= T_j \end{aligned} \tag{3.10}$$

which means that T_j is symmetric.

Algorithm 7: Symmetric Lanczos process for iteratively generating a basis for $\mathbb{K}_{j+1}(N^{-1}D^2, \hat{Y})$

input : starting block of vectors $\hat{Y} \in \mathbb{R}^{N\times n}$

output: $(\cdot,\cdot)_N$-orthonormal basis $\{V_1, V_2, \dots, V_{j+1}\}$

Orthogonalize starting block in $(\cdot,\cdot)_N$ producing QR-factorization:
$\hat{Y} = \underbrace{V_1}_{\in \mathbb{R}^{N\times n}} \underbrace{B_0}_{\in \mathbb{R}^{n\times n}}$ with $V_1^T N V_1 = I_m$

for $i = 1, 2, \dots, j$ **do**
  $W \leftarrow N^{-1}D^2 V_i$
  if $i > 1$ **then**
    $W \leftarrow W - V_{i-1}B_i$
  end
  $H_i = V_i^T NW$
  $W \leftarrow W - V_i H_i$
  Orthogonalize W in $(\cdot,\cdot)_N$ producing QR-factorization: $W = V_{i+1}B_{i+1}$
end

Building the basis in blocks with the computational kernal being a matrix-times-matrix product and also block orthogonalization produces an algorithm with computationally attractive properties in terms of computational intensity (i.e., the amount of calculations done per unit of data moved into and out of cache).

3.3.2 Projection methods for Sylvesters equations

We now adapt the work in (10), Section 5.2 to develop a practical method for treating equation 3.7. Similar to methods for solving linear systems, we can discretize the Sylvester equations by taking approximations from a subspace and solving a constrainted set of equations to determine the specific approximation. In this case, the unknown from 3.7 is $M \in \mathbb{R}^{N\times n}$. At step j, we make an approximation of the form

$$M \approx M_j = \mathcal{V}_j Z_j = \sum_{i=1}^{j} V_i G_i \text{ where } G_i \in \mathbb{R}^{n\times n} \text{ and } Z_j = \begin{bmatrix} G_1 \\ G_2 \\ \vdots \\ G_j \end{bmatrix} \in \mathbb{R}^{jn\times n} \tag{3.11}$$

Substituting into 3.7 yields

$$N^{-1}D^2\mathcal{V}_j Z_j + \mathcal{V}_j Z_j \hat{S} \approx \hat{Y}$$

We determine Z_j to be the solution satisfying the equation

$$\mathcal{V}_j^T(N^{-1}D^2\mathcal{V}_jZ_j + \mathcal{V}_jZ_j\hat{S}) = \mathcal{V}_j^T\hat{Y} \tag{3.12}$$

Let $E_1 \in \mathbb{R}^{jn\times n}$ be the matrix with the identity matrix in the first $n \times n$ block and zeros elsewhere. Observing that $\hat{Y} = V_1B_0 = \mathcal{V}_jE_1B_0$ and using 3.10, we can simplify 3.12 as

$$T_jZ_j + Z_j\hat{S} = E_1B_0 \tag{3.13}$$

The task has now become to solve a small, fairly dense set of Sylvester equations. We can use standard methods to treat this problem.

We approach 3.13 by decomposing the matrices T_j and $\hat{S}$. The matrix T_j is symmetric, tridiagonal. Thus, we can efficiently compute the eigendecomposition $T_j = Q_j\Phi_jQ_j^T$ where $Q_j^TQ_j = I_j$ and $\Phi_j = diag\{\phi_1, \phi_2, \dots, \phi_j\}$ is the diagonal matrix of eigenvalues of T_j. The matrix $\hat{S}$ is not symmetric with respect to the standard Euclidean inner product. However, we see that

Lemma 3.3.2. *The matrix $\hat{S} = A^TTAP^{-1}$ is symmetric with respect to the inner product induced by P^{-1}.*

Proof. The matrix P is a diagonal matrix with positive entries on the diagonal. Thus it and its inverse are symmetric positive-definite, and P^{-1} induces the inner product $(\cdot,\cdot)_{P^{-1}}$, with $(x, y)_{P^{-1}} = y^TP^{-1}x$. We can show by direct calculation that $\hat{S}$ is symmetric with respect to $(\cdot,\cdot)_{P^{-1}}$,

$$(A^TTAP^{-1}x, y)_{P^{-1}} = y^TP^{-1}A^TTAP^{-1}x = (x, A^TTAP^{-1}y)_{P^{-1}}$$

□

Since $\hat{S}$ is symmetric with respect to $(\cdot,\cdot)_{P^{-1}}$, it is diagonalizable and its eigenvectors form an orthonormal basis with respect to $(\cdot,\cdot)_{P^{-1}}$. Thus, we can write $\hat{S} = URU^T$ with $R = diag\{\rho_1, \rho_2, \dots, \rho_n\}$ being a diagonal matrix of eigenvalues, and $U^TP^{-1}U = I_n$. This decomposition can be computed once at the beginning of execution.

In [(10), Section 5.2], it is shown that one can directly compute the solution

$$Z_j = -Q_jF_jU \text{ where } (F_j)_{ij} = \frac{e_i^TQ_j^TE_1B_0Ue_j}{\phi_i + \rho_j}$$

If n were quite large, we could explore doing a data compression by finding a low-rank

approximation of F_j ; however, as n is quite small, this is not needed for this problem.

Remark. Although T_j contains T_{j-1} as its upper-left $(j-1)n \times (j-1)n$ sub-matrix, the eigendecomposition changes completely and must be recomputed at every iteration.

Computing the solution and checking the residual error at each iteration would be costly. Fortunately, we can compute the residual norm at each iteration without constructing the solution. Thus, we can run the Lanczos iteration to build T_j , use this and other quantities to check the residual norm. Once the convergence criteria has been satisfied, we stop, construct Z_j , and rerun the Lanczos process to get the vectors again for constructing the solution. In [(10), Section 5.2], it is shown how to calculate an expresson for the residual norm without forming the solution explicitely. This done via accumulation, and in [(10), Algorithm 5], we see that this in principle, one needs to calculate the eigendecomposition of T_i at each iteration i. However, the authors discuss in [(10), Section 3.2] how one can calculate only the needed entries of the eigenvectors to calculate the residual norm. In this version of the algorithm, we ignore this trick and simply compute the full eigendecomposition. We incorporate this calculation into our description of the procedure, Algorithm 8.

Remark. We note that from 3.11, we can construct the approximation M_j block-for-block meaning when we rerun the Lanczos iteration, we can use the blocks as they are constructed and then discard them, retaining the memory efficiency of Algorithm 8.

Algorithm 8: Symmetric Sparse Sylvester Solver

input : starting block of vectors $\hat{Y} \in \mathbb{R}^{N\times n}$; coefficient matrix $N^{-1}D^2 \in \mathbb{R}^{N\times N}$; a small coefficient matrix $\hat{S} \in \mathbb{R}^{n\times n}$; $\epsilon > 0$ a convergence tolerance

output: An approximation $M_j \in \mathbb{R}^{N\times n}$ to the solution of the Sylvester equations 3.7

Compute eigendecomposition $\hat{S} = URU^T$

Orthogonalize starting block in $(\cdot,\cdot)_N$ producing QR-factorization: $\hat{Y} = VB_0$

$V_{old} = 0$; $i = 0$

while $\sqrt{\gamma} > \epsilon\|B_0\|_F$ **do**

  $i \leftarrow i + 1$

  `/* Lanczos step ****************************************** */`

  $W \leftarrow N^{-1}D^2V$

  if $i > 1$ **then**

    $W \leftarrow W - V_{old}B_i$

  end

  $H_i = V^TNW$

  $W \leftarrow W - VH_i$

  $V_{old} = V$

  Orthogonalize W in $(\cdot,\cdot)_N$ producing QR-factorization: $W = VB_{i+1}$

  `/* computing residual norm ******************************** */`

  $\gamma \leftarrow 0$

  Compute eigendecomposition $Ti = Q_i\Phi_iQ_i^T$

  $S \leftarrow (U^TB_0^T)(E_1^TQ_i)$

  $J \leftarrow (Q_iE_n)B_{i+1}^T$

  for $j = 1, 2, \ldots, n$ **do**

    $K \leftarrow \rho_jI + \Phi_i$

    $\gamma \leftarrow \gamma + \|(e_j^TS)K^{-1}J\|_2^2$

  end

end

`/* constructing approximation coefficients entry-wise ****** */`

$F = 0 \in \mathbb{R}^{in\times n}$

for $k = 1, 2, \ldots, in$ **do**

  for $\ell = 1, 2, \ldots, n$ **do**

    $F_{k\ell} = \frac{e_k^TQ_\ell^TE_1B_0Ue_\ell}{\phi_k+\rho_\ell}$

  end

end

$Z = Q_iFU^T$

`/* rerun Lanczos to assemble solution ********************** */`

$M = 0$

$V \leftarrow \hat{Y}B_0^{-1}$

$V_{old} = 0$

for $j = 1, 2, \ldots, i$ **do**

  $M \leftarrow M + VG_j$

  $W \leftarrow N^{-1}D^2V$

  if $j > 1$ **then**

    $W \leftarrow W - V_{old}B_j$

  end

  $W \leftarrow W - VH_j$

  $V_{old} \leftarrow V$

  $V \leftarrow WB_{j+1}^{-1}$

end

Chapter 4

Implementation

This chapter describes the implementation of the conjugate gradient method and the Sylvester equation method, and these methods were implemented using Golang since it has easy-to-use support for concurrency programming. A standard method which builds matrices $Q + B^T CB$ and $B^T Cy$ explicitly was also implemented as the baseline of these two methods. The implementations can be found in the appendix.

4.1 Preparation

4.1.1 Input Matrix Generation

To solve equation 1.1, several input parameters (m,n, and N) and matrices (A, T and y) is needed.

The value of these parameters and matrices is determined by the input dataset. The value of n represents the number of sources, and the value of m represents the number of frequencies. The value of N is determined by the N_{side} value of fits file $N = N_{side} \times N_{side} * 12$ as there are 12 different patches in each fits file. In the case of Planck data, n is 4, and m is 9, the N_{side} value of Planck full sky maps is 1024 or 2048. Input matrices A and T can be found in the source code, the matrix $y \in \mathbb{R}^{nN}$ is constructed using the following

method,

Algorithm 9: Building the input vector y

input : N_{side} of fits file; 9 Planck full sky maps downloaded from here

output: y

```
1 y = 0
2 for frequency = 1, 2, ... , 9 do
3 |   curmap = frequencymaps[frequency] for patch = 1, 2, ... , 12 do
4 |   |   tmpvector ← extract(curmap, patch)
5 |   |   y ← append(y, tmpvector)
6 |   end
7 end
```

Algorithm 9 shows the process of building the vector y. In the fourth step, we extract each base from a Planck full sky map using cfitsio library. Planck All Sky Maps are in HEALPix format with minimum N_{side} 1024, and each map contains 12 different patches, and we can extract a vector of length 12,582,912 from each map. Then, these 9 vectors were concatenated together to create vector $y \in \mathbb{R}^{113246208}$. So the maximum length of vector y is 113,246,208.

In the implementations, we assume matrices N and P are identity matrix for simplicity.

4.1.2 Adjacency Matrix D

The adjacency matrix D encodes a nearest neighbor coupling of nodes from the discretization of the sky. When N_{side} is 1, D is an adjacency matrix of the following matrix

$$L = \begin{bmatrix} 0 & 3 & 6 & 9 \\ 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \end{bmatrix}$$

Matrix L has $3 \times N_{side}$ rows and $4 \times N_{side}$ columns, the values in L represents the indexes of matrix D. So, we have the following adjacency D when N_{side} is 1 according to 1.2.

$$D = \begin{pmatrix} -2 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -3 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -2 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -3 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & -3 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & -3 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -3 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -3 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -2 \end{pmatrix} \text{ with } N_{side} = 1$$

Due to the size of adjacency matrix $D \in \mathbb{R}^{N \times N}$, it would not be explicitly built in Conjugate Gradient method and Sylvester Equation method. Instead, we implemented an

API to retrieve any row from D easily, Algorithm 10 describes how this API works,

Algorithm 10: RowView of matrix D

input : rowIdx as the row index; N_{side} of fits file.

output: neighbours as a index list of all the neighbour nodes

$row = 3 \times N_{side}$

$col = 4 \times N_{side}$

$neighbours \leftarrow set(rowIdx - row, rowIdx + row, rowIdx + 1, rowIdx - 1)$

`/* rowIdx is in the first row ********** */`

if $rowIdx \mod row == 0$ **then**

  $neighbours.remove(rowIdx - 1)$

end

`/* rowIdx is in the last row ********** */`

if $rowIdx \mod row == row - 1$ **then**

  $neighbours.remove(rowIdx + 1)$

end

`/* rowIdx is in the first column ********** */`

if $rowIdx < row$ **then**

  $neighbours.remove(rowIdx - row)$

end

`/* rowIdx is in the last column ********** */`

if $rowIdx > row \times col - 1 - row$ **then**

  $neighbours.remove(rowIdx + row)$

end

Algorithm 10 returns a neighbour index list of all a given node, we can create a multi-threading function to compute Dv by combining it with Algorithm 1. Algorithm 11 describes this function. It is worth noting that a little trick was used here to reduce the overhead of $reshape(\cdot, \cdot, \cdot)$ operation. In Algorithm 3, we need to perform $U = reshape(u, N, n)$, this reshape operation is only performed implicitly, we create an API based on the vector u, and the matrix multiplication is performed based on this API, and this can improve performance and memory usage as no extra memory or reshape operation is needed in

this way.

Algorithm 11: Parallel Function for Qv

input : N_{side} of fits file; vector $v \in \mathbb{R}^{nN}$; ThreadNum for the number of threads
output: result of Dv

```
1  result = 0
2  for threadID = 0, 1, ... , ThreadNum do
3      do in parallel
4          for i ← threadID to N by ThreadNum do
               /* Use 10 to find all the neighbor indexes                */
5              neighborIDs = findNeighbors(i, N_side)
6              for j ← 0to n by 1 do
7                  val ← 0
8                  for nid in neighborIDs do
9                      curv ← v[N × j + i]
10                     val ← val + curv
11                 end
12                 result[N * j + i] = val
13             end
14         end
15     end
16 end
```

There are 4 loops in Algorithm 11, the first loop will create multiple threads (goroutine in Golang) to compute the result, the time complexity for the second loop is $\mathcal{O}(N/ThreadNum)$, the time complexity for the third loop is $\mathcal{O}(n)$, and the time complexity for the fourth loop is $\mathcal{O}(1)$ since there are at most four neighbors for each node. So, the time complexity for Algorithm 11 is $\mathcal{O}(nN/ThreadNum)$.

4.2 Standard Method using built-in API

A standard method is created as a baseline. This method using the *VecDense.solveVec* API in the mat library. This method is very low efficient in terms of memory and CPU usage as it needs to build matrices with $nN \times nN$ dimension. For example, when N_{side} is 1024 and n is 4, each element is a 64-bit float number, a matrix of size $nN \times nN$ will take 2,359,296 GiB of memory.

4.3 Conjugate Gradient Method

The Conjugate Gradient Method is an implementation of Algorithm 6 with minor changes.

4.3.1 Initialization

In the initialization step, parameters (m,n, and N) and matrices (A, T and y)were initialized. The vector $b = B^T Cy$ were computed using the Algorithm 12. The time complexity for this step is $\mathcal{O}(Nmn + Nn^2)$.

Algorithm 12: Matvec procedure $b \to B^T Cy$ using reshaping

input : vector $y \in \mathbb{R}^{mN}$; matrix $B = A \otimes I_N$ with $A \in \mathbb{R}^{m \times n}$; matrix $C = T \otimes I_N$ with $T \in \mathbb{R}^{m \times m}$

output: Qu

1 $Y = reshape(y, N, m)$

2 $Y \leftarrow YTA$

3 $b = vectorize(Y)$

$A^T TA$ was also computed at the beginning, it will be used by Algorithm 5 during the iteration step.

4.3.2 Iteration

This section describes the detailed iteration steps in Algorithm 6, the time complexity for each step will be discussed.

Algorithm 13: Conjugate Gradient Implementation

input : vector $y \in \mathbb{R}^{mN}$; matrix $B = A \otimes I_N$ with $A \in \mathbb{R}^{m \times n}$; matrix $C = T \otimes I_N$ with $T \in \mathbb{R}^{m \times m}$

output: solution x to equation 1.1

1 $x = 0$

/* Qx is computed using multiple threads */

/* $B^T CBx$ is computed using reshaping */

2 $r = b - Qx - B^T CBx \quad \leftarrow \mathcal{O}(nN/ThreadNum + n^2N + nN)$

3 $p = r$

4 $rNorm = r^T r \quad \leftarrow \mathcal{O}(nN)$

5 **for** $i \leftarrow 1, 2, \ldots, MaxIterNum$ **do**

6 $\quad q \leftarrow Qp + B^T CBp \quad \leftarrow \mathcal{O}(nN/ThreadNum + n^2N + nN)$

7 $\quad \alpha \leftarrow rNorm/p^T q \quad \leftarrow \mathcal{O}(nN)$

8 $\quad x \leftarrow x + \alpha p \quad \leftarrow \mathcal{O}(nN)$

$\quad$ /* re-computed residual ****** */

9 $\quad r = b - Qx - B^T CBx \quad \leftarrow \mathcal{O}(nN/ThreadNum + n^2N + nN)$

10 $\quad newNorm \leftarrow r^T r \quad \leftarrow \mathcal{O}(nN)$

11 $\quad \beta \leftarrow newNorm/rNorm \quad \leftarrow \mathcal{O}(1)$

12 $\quad rNorm \leftarrow newNorm$

13 $\quad$ **if** $newNorm < \epsilon$ **then**

$\quad\quad$ /* residual is small enough, return result vector x */

14 $\quad\quad$ break

15 $\quad$ **end**

16 $\quad p \leftarrow r + \beta p \quad \leftarrow \mathcal{O}(nN)$

17 **end**

In Algorithm 6, the residual is computed using $r \leftarrow r - \alpha q$, and the time complexity is $\mathcal{O}(nN)$. However, the residual becomes inaccurate due to the round-off error introduced in each iteration. So, two different approaches are implemented. In the first approach, the residual is computed using $r \leftarrow r - \alpha q$. In the second approach, the residual is re-computed using $B^T Cy - (Q + B^T CB)x$ at each iteration, it could help improve accuracy, but it also increases time complexity for residual computation, Algorithm 13 describes this approach. And the time complexity of residual computation for both approaches is $\mathcal{O}(nN)$ and $\mathcal{O}(nN/ThreadNum + n^2N + nN)$ respectively. A $MaxIterNum$ variable also used here to make sure that this method will stop if it fails to converge. We choose 10^-5

as the ϵ value in the implementation, a smaller ϵ value will increase accuracy, but it also increases the number of iterations and the overall time cost.

The overall time complexity for our implementation of the conjugate gradient method is $\mathcal{O}(in^2N + mnN)$ where i is the number of iterations. For space complexity, the vector $y \in \mathbb{R}^{mN}$ is needed at the initialization step, several vectors of length nN is also stored during the iteration, so the overall space complexity is $\mathcal{O}(nN + mN)$.

4.4 Sylvester Equation Method

The Sylvester Equation Method is an implementation of Algorithm 8.

In step 4 of Algorithm 8, a ϵ value is used as the stop condition of the while loop. However, we took a different approach in the implementation. We save the γ value to another variable γ_{old} before γ is recomputed at each loop, and the while loop will be stopped if $\gamma_{old} > \gamma$.

Next, we discuss the most time-consuming steps in the algorithm.

In step 2 and 13 of Algorithm 8, we need to perform QR-factorization to a matrix of size $\mathbb{R}^{N\times n}$. A full QR-factorization $A = QR$ with $Q \in \mathbb{R}^{N\times N}$ and $R \in \mathbb{R}^{N\times n}$ would increase time and space complexity. Instead, a reduced QR factorization using the Gram-Schmidt method is used here to reduce time and space complexity. We implemented a reduced QR-factorization function here for reusability. This function take a matrix as a parameter and returns a matrix $Q \in \mathbb{R}^{N\times n}$ and an upper triangular matrix $R \in \mathbb{R}^{n\times n}$. The time complexity for this matrix is $\mathcal{O}(n^2N)$, and space complexity is $\mathcal{O}(nN)$.

In step 6, matrix product operation $N^{-1}D^2V$ needs to be performed in step 6 and step 35 of Algorithm 8, because we treat N as an identity matrix in the implementation, so $N^{-1}D^2V = D^2V$, and the function of 11 can be reused here, and we will take one column from V each time and pass it to function 11, and V has n columns, so the time complexity for step 6 and step 35 is $\mathcal{O}(nN/ThreadNum)$.

In some steps such as step 8, step 11, and step 34, matrix multiplication needs to be performed, and the time complexity for these steps is $\mathcal{O}(n^2N)$. So, the overall time complexity for Algorithm 8 is $\mathcal{O}(n^2N)$, and the space complexity is $\mathcal{O}(nN)$.

Chapter 5

Results

In the results chapter, we run all three implementations using Planck full sky maps (conjugate gradient solution has two variants), and results will be compared in terms of time cost, memory usage, and accuracy.

The experiment is designed as follows. The standard solution, conjugate gradient solution, and Sylvester equation solution will be run multiple times with different N_{side} value. Each time, these three methods receive the same input parameters:$m = 9$,$n = 4$,$A =$
$$\begin{bmatrix} 1 & 24.31408176 & 0.1808804 & 13.15799486 \\ 1 & 8.81745815 & 0.3154685 & 5.80101879 \\ 1 & 2.58070139 & 0.6121030 & 2.15148219 \\ 1 & 1.00587794 & 1.0058779 & 1.00587794 \\ 1 & 0.39224839 & 1.6297015 & 0.47074157 \\ 1 & 0.13192930 & 2.7832105 & 0.19585548 \\ 1 & 0.03801049 & 4.9311851 & 0.07232168 \\ 1 & 0.01327148 & 7.7040185 & 0.03151240 \\ 1 & 0.00509560 & 11.3366157 & 0.01524107 \end{bmatrix}$$
, and T = diag{629881.6, 694444.4, 783146.7, 12755102.0, 30864197.5, 30864197.5, 30864197.5, 30864197.5, 30864197.5}. The input vector $y \in \mathbb{R}^{mN}$ is generated for each N_{side} value before the testing using the Planck full sky maps. 100 goroutines will be used when concurrency programming is applicable.

5.1 Environment Setup

An EC2 instance in AWS(Amazon Web Services) is created for experiments. Table 5.1 describes the details for the EC2 instance.

Table 5.1: AWS EC2 instance details

Instance Type	t2.2xlarge
ECUs	Variable
vCPUs	8
Architecture	x86_64
Threads per core	1
Sustained clock speed (GHz)	2.3
Memory (MiB)	32768
Instance Storage (GB)	EBS only
Volume Type	gp2(General Purpose SSD)
IOPS	100 / 3000
Operating System	Ubuntu 18.04.4 LTS
Kernel Version	5.3.0-1032-aws

5.2 Time Cost

Table 5.2: Time Cost (milliseconds) of Three Solutions.

N_{side}	Standard	Conjugate Gradient	Conjugate Gradient With Auto-Correction	Sylvester Equation
2	7	2	4	3
4	91	3	7	4
8	3108	7	19	13
16	176817	20	54	22
32	-	73	195	35
64	-	269	707	287
128	-	1157	3299	721
256	-	6333	17256	2949
512	-	27255	79194	14082
1024	-	117757	310429	77564

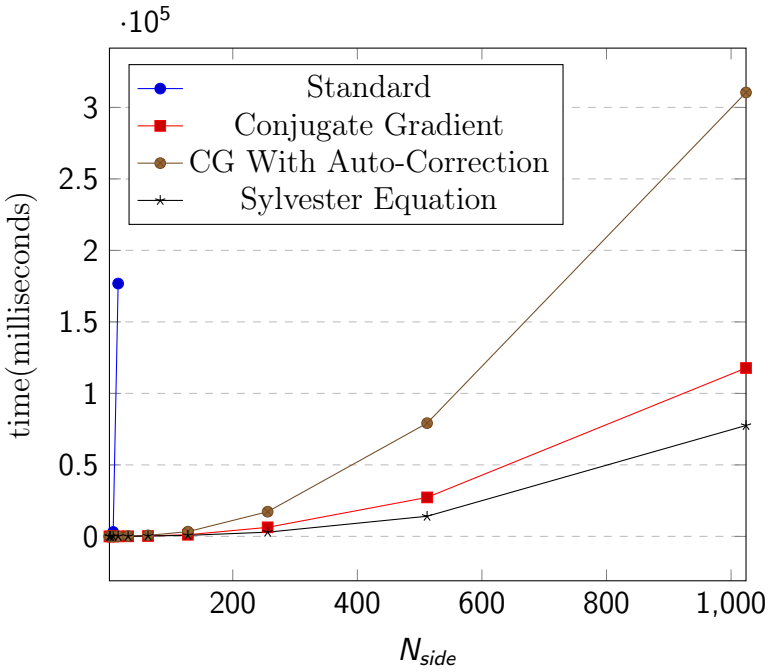

Figure 5.1: Time Cost with different N_{side}

Table 5.2 and figure 5.1 describe the time cost for three implementations with different N_{side} values. The time cost for the standard method is not available when N_{side} is bigger than 16 because of out-of-memory error. And Conjugate Gradient With Auto-Correction represents the conjugate gradient method where the residual being recomputed using $r \leftarrow b - Ax$ at every iteration.

As shown in table 5.2 and figure 5.1 , the standard method has better performance only when N_{side} is 2, then, it encounters out-of-memory after N_{side} is 16 because it needs to build matrices of size $\mathbb{R}^{nN \times nN}$ in memory. The Sylvester Equation solution provides the best performance for all N_{side} values. The Conjugate Gradient implementations are faster than the standard solution but slower than the Sylvester Equation solution. The conjugate gradient implementation is faster than the conjugate gradient with auto-correction as it does not need to recomputed the residual at each iteration.

5.3 Accuracy

Table 5.3 and figure 5.2 describe the residual of different solutions. The residual is computed using the following method,

1. Computing $b = B^T Cy$ using Algorithm 12.

2. Computing Qx using Algorithm 3 and Algorithm 11.

3. Computing $B^T CBx$ using Algorithm 5.

4. Computing $r = b - Qx - B^T CBx$.

5. $residual = r^T r$.

The time cost for computing residual is not included in table 5.2 and figure 5.1.

Table 5.3: Residual of Three Solutions.

N_{side}	Standard	Conjugate Gradient	Conjugate Gradient With Auto-Correction	Sylvester Equation
2	1.159225e-07	4.626469e-07	4.602519e-06	2.349828e-03
4	2.461592e-07	5.766567e-07	6.915607e-06	3.277893e-03
8	5.614913e-07	1.588664e-06	7.934169e-06	4.892446e-04
16	1.032700e-06	3.176654e-06	1.008632e-06	2.319466e-04
32	-	4.304763e-06	1.374675e-06	5.817023e-03
64	-	5.535777e-06	1.743090e-06	2.186816e-04
128	-	6.702022e-06	2.154607e-06	2.051122e-04
256	-	7.604039e-06	2.353452e-06	1.624415e-04
512	-	8.252590e-06	2.557356e-06	6.294534e-05
1024	-	2.119601e-05	8.388576e-06	3.567721e-02

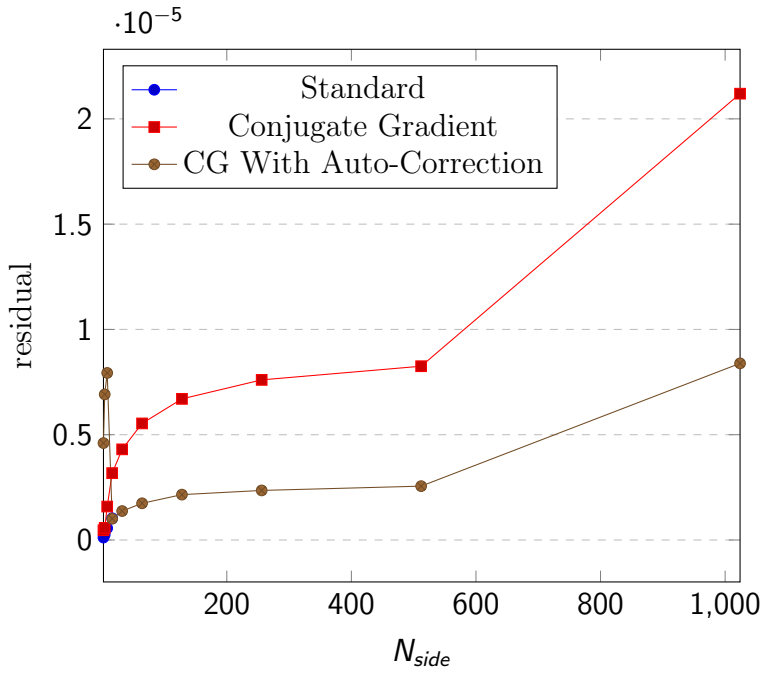

Figure 5.2: Residual with different N_{side}

As shown in table 5.3, the standard solution gives the best accuracy, and it encounters out-of-memory error when N_{side} is 32. The conjugate Gradient solution with auto-correction is

the next best solution in terms of accuracy, it is worth noting that the stop loop condition is $residual < 10^{-5}$, the residual may decrease if we change the stop condition, but the time cost could also increase because more iterations are needed to achieve better accuracy. The conjugate gradient solution provides decent accuracy, but it is not as accurate as the conjugate gradient with auto-correction, and we could see how the round-off error affects the accuracy from figure 5.2. Finally, the Sylvester Equation implementation has lower accuracy, and it is not plotted in figure 5.2.

Table 5.4: Residual of Sylvester Equation Solution.

N_{side}	Iteration Residual	Final Residual
2	2.457549e+08	2.349828e-03
4	1.721217e+08	3.277893e-03
8	4.918023e+07	4.892446e-04
16	2.346382e+07	2.319466e-04
32	1.441014e+08	5.817023e-03
64	6.251446e+07	2.186816e-04
128	6.008143e+07	2.051122e-04
256	1.711665e+08	1.624415e-04
512	1.365440e+08	6.294534e-05
1024	3.638237e+09	3.567721e-02

Table 5.4 describes the residual during iteration and the final residual. According to Algorithm 8, the residual that Sylvester Equation solution computes within iteration is not the actual residual, a different residual representation is used because it is easier to compute. After the solution x is found, the residual is computed using the method 5.3.

Table 5.5: The Error between the Standard solution and other solutions

N_{side}	Conjugate Gradient	Conjugate Gradient With Auto-Correction	Sylvester Equation
2	3.0180889776024903e-21	4.2905896490884486e-21	1.4985910770697552e-15
4	8.859910588909837e-21	1.0834453453914871e-20	2.09007356359636e-15
8	2.17794839011864e-20	1.8716053445427665e-20	3.090258469421773e-16
16	4.212746208331008e-20	3.757696375911772e-20	1.4569865476906593e-16

Table 5.5, 5.6, 5.7, and 5.8 describe differences of different solutions, the error is measured using $\|x_1 - x_2\|_2^2$. The results in these tables are consistent with the results in table 5.3. The standard solution, the conjugate gradient solution, and the conjugate gradient solution with auto-correction are in the same level of accuracy, they have similar residuals and errors. As shown in table 5.5, 5.6, and 5.7, results of the Standard method, the Conjugate

Table 5.6: The Error between the Conjugate Gradient solution and other solutions

N_{side}	Standard	Conjugate Gradient With Auto-Correction	Sylvester Equation
2	3.0180889776024903e-21	3.7667853516894236e-21	1.498364533232887e-15
4	8.859910588909837e-21	7.963321715133915e-21	2.0915953240318653e-15
8	2.17794839011864e-20	9.65210028520249e-21	3.106570590280657e-16
16	4.212746208331008e-20	1.679910211233754e-20	1.4589451358002164e-16
32	-	2.845103856694158e-20	3.702287849388512e-15
64	-	4.1124088585501615e-20	1.2496524250543757e-16
128	-	4.944374950088266e-20	1.2229585296470213e-16
256	-	5.477400230053128e-20	9.919744223655086e-17
512	-	6.183014084279126e-20	3.535262581317476e-17
1024	-	1.4478246538147941e-19	2.2707199637015396e-14

Table 5.7: The Error between the Conjugate Gradient With Auto-Correction solution and other solutions

N_{side}	Standard	Conjugate Gradient	Sylvester Equation
2	4.2905896490884486e-21	3.7667853516894236e-21	1.4985629938289051e-15
4	1.0834453453914871e-20	7.963321715133915e-21	2.0908257938289905e-15
8	1.8716053445427665e-20	9.65210028520249e-21	3.1048538801260387e-16
16	3.757696375911772e-20	1.679910211233754e-20	1.4600360044051388e-16
32	-	2.845103856694158e-20	3.703337743689728e-15
64	-	4.1124088585501615e-20	1.251820078717131e-16
128	-	4.944374950088266e-20	1.220658131349531e-16
256	-	5.477400230053128e-20	9.923869835052757e-17
512	-	6.183014084279126e-20	3.546528314450234e-17
1024	-	1.4478246538147941e-19	2.2709370743112487e-14

Gradient method, and the Conjugate Gradient With Auto-Correction method are very close. As shown in table 5.3 and 5.8, the Sylvester solution has the worst accuracy, it has the biggest residuals and error. However, as shown in table 5.8, these errors are much smaller than the residuals, it is possible that errors between different solutions are actually small but they are stretched and amplified by matrices when they are mapped into the residual space.

5.4 Convergence

The conjugate gradient and Sylvester equation solution are both iterative methods. In this section, we discuss how many iterations it takes for each algorithm to reach convergence and the residual for each iteration. For the following experiment results in this

Table 5.8: The Error between the Sylvester Equation solution and other solutions

N_{side}	Standard	Conjugate Gradient	Conjugate Gradient With Auto-Correction
2	1.4985910770697552e-15	1.498364533232887e-15	1.4985629938289051e-15
4	2.09007356359636e-15	2.0915953240318653e-15	2.0908257938289905e-15
8	3.090258469421773e-16	3.106570590280657e-16	3.1048538801260387e-16
16	1.4569865476906593e-16	1.4589451358002164e-16	1.4600360044051388e-16
32	-	3.702287849388512e-15	3.703337743689728e-15
64	-	1.2496524250543757e-16	1.251820078717131e-16
128	-	1.2229585296470213e-16	1.220658131349531e-16
256	-	9.919744223655086e-17	9.923869835052757e-17
512	-	3.535262581317476e-17	3.546528314450234e-17
1024	-	2.2707199637015396e-14	2.2709370743112487e-14

section, the value of N_{side} is 1024.

Table 5.9: Residual vs Iteration.

Iteration	Conjugate Gradient	Conjugate Gradient With Auto-Correction	Sylvester Equation
0	9.609525807492873e+25	9.609525807492873e+25	9.80281888412712e+12
1	1.259418859319246e+23	1.259418859319246e+23	1.0719467776878633e+10
2	2.3870612202524674e+23	2.3870612202524644e+23	1.8455672681064317e+09
3	1.0574369640294e+21	1.0574369640294167e+21	3.798235797307401e+08
4	4.911828490873264e+20	4.911828244770185e+20	
5	6.834080278989695e+12	6.83408034618299e+12	
6	1.8299976007865277e+11	1.8299975970160046e+11	
7	4.999243517546015e+09	4.999241243639853e+09	
8	7.425974042107423e+09	7.425979093104691e+09	
9	40220.07766953336	40259.65550802165	
10	48.08397442770766	2605.246727349994	
11	10.214055009836672	8.448136801219533	
12	0.18223545970466598	0.27235044921784624	
13	0.0009248160851663429	0.5508047514686585	
14	0.001908637364531107	21.623935700077862	
15	1.24791182380328e-09	22.033337865469395	
16		0.008319263132760272	
17		0.00015894137193445015	
18		0.003924241135929268	
19		0.00014430056657147418	
20		9.559741174228862e-05	
21		0.00010990319798219979	
22		7.609952200856389e-06	

Table 5.9 describes the relationship between iteration and residual of each solution when N_{side} value is 1024. The residual here is the residual during iteration, it might not be the real residual for some solutions e.g. Conjugate Gradient and Sylvester equation. The residual of iteration 0 means the starting residual for each solution.

For the conjugate gradient method, it takes 15 iterations to find a solution, the residual here is computed using $r \leftarrow r - \alpha q$, so it might be larger than the real residual. For the conjugate gradient with auto-correction solution, the residual is re-corrected using $b - Ax$ at each iteration, so this method takes more iterations and longer time cost than the conjugate gradient method. The Sylvester equation method only takes 4 iterations, which is one of the reasons that it is the fastest solution. The residual here is a different quantity from the real residual, and the iteration stops when the residual value increases.

5.5 Memory Usage

Table 5.10: Memory Usage (MB) of Three Solutions.

N_{side}	Standard	Conjugate Gradient	Conjugate Gradient With Auto-Correction	Sylvester Equation
2	689	688	689	690
4	757	689	690	689
8	1748	688	691	691
16	17075	691	691	691
32	-	691	691	691
64	-	691	691	691
128	-	824	758	824
256	-	1289	1223	1157
512	-	3216	3348	3084
1024	-	9935	10331	10067

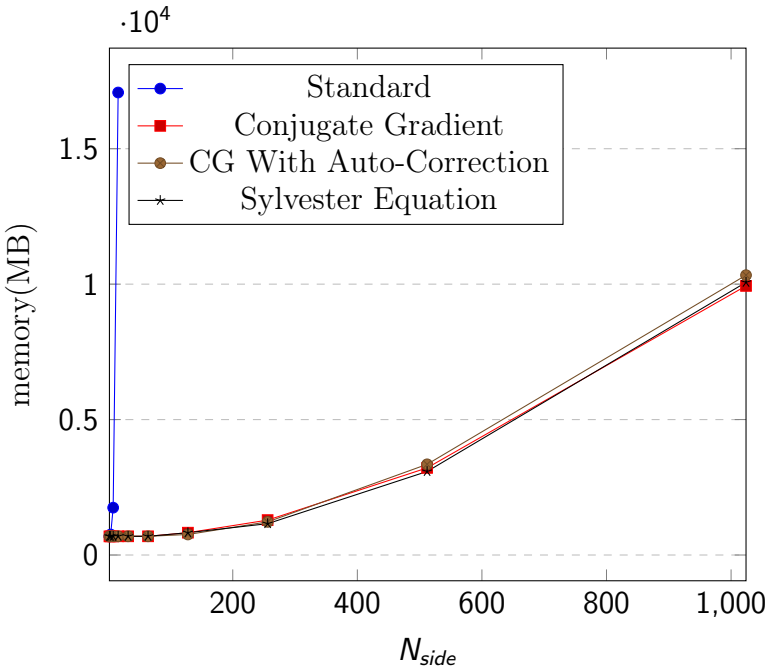

Figure 5.3: Memory Usage with different N_{side}

Table 5.10 and figure 5.3 describe the memory usage of three solutions. The memory usage is recorded using pmap. After the process is started, a shell script will be invoked, and it will use pmap to record the memory usage of the process every 10 milliseconds. We will take the maximum number during process runtime as the memory usage of this experiment.

As shown in table 5.10 and figure 5.3, the conjugate gradient, the conjugate gradient with auto-correction, and Sylvester equation solutions have similar memory usage, since they all need to store vectors of size nN in memory. The memory usage of standard solution increase dramatically until it is killed because of out-of-memory error when N_{side} is 32, these results meet our expectations because it needs to save matrics of size $nN \times nN$ in memory.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this paper, we proposed and discussed two approaches for the CMB source separation problem, and experiments have been carried out with Planck data. These experiments show that both algorithms have good performance, but the accuracy of these two algorithms varies.

The conjugate gradient method can solve source separation with good accuracy in a reasonable time for the data of the scale of Planck, it has linear time and space complexity. Trade-offs can be made between accuracy and time cost by re-computing the residual or altering the ϵ value in the stop condition. The residual can be re-computed at each iteration to minimize the effect of round-off error introduced by iterations to increase accuracy. Another approach to increase accuracy is by choosing a smaller ϵ value for the stop condition, it increases the number of iterations, which increases the time cost. In our experiment, the conjugate gradient solution find a solution to our CMB source separation problem under 2 minutes when N_{side} is 1024 and $N = \mathcal{O}(10^7)$.

The Sylvester equation method works by re-writing the equation 1.1 as a Sylvester equation, and it also has linear space and time complexity. This solution is faster than the conjugate gradient solution. In our experiment, this method can solve source separation for Planck data in 77 seconds. However, this method also has lower accuracy.

6.2 Future Work

There are still many methods to improve the performance and accuracy of these two algorithms. We leave these improvements for the future.

One possible improvement is that parallel programming can be applied to more processes. In our implementation, the only step that utilizes parallel programming is the production of the adjacency matrix and a vector $u \leftarrow Du$, it is feasible to apply parallel programming to more steps such as matrix production to reduce the time cost of both solutions even further.

It might be helpful to do a systematic numerical analysis about how the round-off error affects the convergence and accuracy. The Sylvester equation solution does not perform very well in terms of accuracy, the research should be helpful to find a better convergence point and a smaller residual. Also, we take some measures in the conjugate gradient method to reduce the effect of the round-off error, but these measures also affect the performance of this method, and it remains unclear how the round-off error affects the Sylvester equation method. The numerical analysis research about round-off error might help us to improve the performance and accuracy of these two algorithms.

Bibliography

[1] J. Delabrouille and J. F. Cardoso, *Diffuse source separation in CMB observations*, pp. 159–205. Springer, 2008.

[2] J. Bobin, J. L. Starck, F. Sureau, and S. Basak, "Sparse component separation for accurate cosmic microwave background estimation," *Astronomy & Astrophysics*, vol. 550, p. A73, 2013.

[3] S. P. Wilson and J. Yoon, "Bayesian ica-based source separation of cosmic microwave background by a discrete functional approximation," *arXiv preprint arXiv:1011.4018*, 2010.

[4] Y. Moudden, J. F. Cardoso, J. L. Starck, and J. Delabrouille, "Blind component separation in wavelet space: Application to cmb analysis," *EURASIP Journal on Advances in Signal Processing*, vol. 2005, no. 15, p. 484606, 2005.

[5] G. Patanchon, J. Delabrouille, and J.-F. Cardoso, "Source separation on astrophysical data sets from the wmap satellite," pp. 1221–1228, Springer.

[6] S. Wagner-Carena, M. Hopkins, A. Diaz Rivero, and C. Dvorkin, "A novel cmb component separation method: hierarchical generalized morphological component analysis," *Monthly Notices of the Royal Astronomical Society*, vol. 494, no. 1, pp. 1507–1529, 2020.

[7] S. P. Wilson, E. E. Kuruoglu, and E. Salerno, "Fully bayesian source separation of astrophysical images modelled by mixture of gaussians," *IEEE Journal of selected topics in signal processing*, vol. 2, no. 5, pp. 685–696, 2008.

[8] M. R. Hestenes and E. Stiefel, "Methods of conjugate gradients for solving linear systems," *Journal of research of the National Bureau of Standards*, vol. 49, no. 6, pp. 409–436, 1952.

[9] C. F. Van Loan, "The ubiquitous kronecker product," *Journal of computational and applied mathematics*, vol. 123, no. 1-2, pp. 85–100, 2000.

[10] D. Palitta and V. Simoncini, "Computationally enhanced projection methods for symmetric sylvester and lyapunov matrix equations," *Journal of Computational and Applied Mathematics*, vol. 330, pp. 648–659, 2018.

[11] Y. Saad, *Iterative methods for sparse linear systems*. SIAM, 2003.

[12] V. Simoncini and D. B. Szyld, "Recent computational developments in krylov subspace methods for linear systems," *Numerical Linear Algebra with Applications*, vol. 14, no. 1, pp. 1–59, 2007.

[13] V. Simoncini, "Computational methods for linear matrix equations," vol. 58, no. 3, pp. 377–441, 2016.

Appendix A

Source Code

Description:

The source code is uploaded to a GitHub repository, the readme file contains the instructions to run the program. The input files of vector y are uploaded to Google Drive, the link is also included in the readme file.

Link:

https://github.com/ZhaoJiaJin/dissertation/tree/master/golang