# Practical Image Deblurring
## with Synthetic Boundary Conditions, with GPUs, and with Multiple Frames

By

Ying Wai Fan
Doctor of Philosophy

Mathematics

# Practical Image Deblurring
with Synthetic Boundary Conditions, with GPUs, and with Multiple Frames

By

Ying Wai Fan
M.S., Wake Forest University, 2005
M.A., Wake Forest University, 2005
B.Sc., The Chinese University of Hong Kong, 2003

Advisor: James Nagy, Ph.D.

An abstract of
A dissertation submitted to the Faculty of the
James T. Laney School of Graduate Studies of Emory University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in Mathematics
2010

**Abstract**

Practical Image Deblurring
with Synthetic Boundary Conditions, with GPUs, and with Multiple Frames
By Ying Wai Fan

Researchers usually use several assumptions when they tackle the image deblurring problem. In particular, it is usually assumed that the blur is known exactly, and that the true image scene outside the field of view is approximated well by periodic boundary conditions. These assumptions are certainly not true in most realistic situations.

In this thesis we develop a new method to derive adaptive synthetic boundary conditions directly from the blurred images. Compared with classical boundary conditions, our approach gives better deblurring results, especially for motion blurred images. To speed up the deblurring algorithms, we also develop a new regularized DCT preconditioner.

We have written two new software packages to facilitate research in image deblurring. The first one PYRET is a serial CPU implementation in Python. With the object-oriented paradigm, we implement numerical algorithms for the general linear problem, and then specialize them for deblurring problems with a new matrix class. A web user interface for PYRET is also provided.

The second software package PARRET is a parallel implementation on NVIDIA CUDA GPU architecture. GPUs provide an economical way to obtain parallel processing power. On a consumer laptop equipped with a GPU, we can attain order of magnitude speedup with PARRET.

Finally, we consider a blind deconvolution problem in which the involved atmospheric blurs are not known in advance. We first reduce the number of variables using a variable projection technique, then solve the reduced problem by the Gauss-Newton algorithm. With careful mathematical manipulation, the Jacobian matrix is decomposed into a series of diagonal and Fourier matrices for inexpensive multiplication. To further improve the deblurring quality, we use more than one blurred image from the same object. We use a new decoupling approach for the sparsity of the Jacobian matrix in this multi-frame case. Experiments show that the deblurring result improves when more images are used.

# Practical Image Deblurring
with Synthetic Boundary Conditions, with GPUs, and with Multiple Frames

By

Ying Wai Fan
M.S., Wake Forest University, 2005
M.A., Wake Forest University, 2005
B.Sc., The Chinese University of Hong Kong, 2003

Advisor: James Nagy, Ph.D.

A dissertation submitted to the Faculty of the
James T. Laney School of Graduate Studies of Emory University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in Mathematics
2010

## Acknowledgments

I thank God for giving me my life, health and intelligence. He was with me in all the struggles I faced in these five years of PhD studies. I am really thankful for what He has done for me, especially for revealing Himself to me, which touched and encouraged me to become a Christian and to get baptized this year. Thank you Lord.

This dissertation would not be possible without the guidance of my adviser, Prof James Nagy. I treasure the freedom he has given me in working on many different topics. He kept connected with me even with his busy schedule and even when we were on opposite side of the globe. We had meetings through Skype in the months when I was back home in Hong Kong. He is also kind enough to pardon my less than perfect English. He has corrected many English mistakes in my papers and in this dissertation. I am also grateful for his support in attending conferences and workshops, which have broadened my horizons and are essential to my career. And of course, he is a good teacher. I am proud to be a student of this expert in image processing.

My gratitude goes to Dr Esmond Ng from Lawrence Berkeley National Laboratory. He has been like a mentor to me. He was my supervisor for an internship years ago and I have met him in almost every conference I have been to. Besides my adviser, he is the other one who has seen me grow academically, and maybe even physically. I first met him when I was still a young undergraduate student.

I would like to thank all the teachers who have taught me in my every level of education. Their student is finally leaving school and is ready to join the workforce and face greater challenges in the world.

I am also thankful for the staff of the department, and of the whole university in general. They have provided me with a very good learning environment. I also want to thank all the friends I made here in Atlanta, especially those from my fellowship. I thank them for their support in my writing of this dissertation and my job search.

I am forever in debt to my family. I know it was hard for them to let me go abroad to study. I am sorry to keep them worried about me. They should be proud that their little boy has now grown into a mature young man. I thank my brother and sisters for taking care of my parents and grandparents when I am away. I am sorry that I could not be with them when my grandparents passed away.

There are many people I still want to thank. Let me just say thanks to all those who have contributed to my well-being and made my finishing of this degree possible. Thank you very much.

*I dedicate this dissertation to the memory of my grandfather,*
*who passed away during my PhD studies in the US.*

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# List of Notations

| | | |
|---|---|---|
| $\boldsymbol{H}$ | point spread function (PSF) | 6 |
| $\boldsymbol{X}$ | original image | 7 |
| $\boldsymbol{Y}$ | blurred image | 7 |
| $\boldsymbol{x}$ | original image (vectorized) | 7 |
| $\boldsymbol{y}$ | blurred image (vectorized) | 7 |
| $\boldsymbol{A}$ | convolution matrix | 7 |
| $\boldsymbol{T}$ | Toeplitz matrix | 7 |
| $\boldsymbol{B}$ | the augmenting matrix corresponding to the boundary conditions | 7 |
| $\mathcal{D}$ | the image domain over which the blurred image is defined | 18 |
| $\mathcal{B}$ | the border region over which the boundary conditions are defined | 18 |
| $\boldsymbol{C}$ | the discrete cosine transform matrix | 24 |
| $\boldsymbol{\Lambda}$ | the eigenvalue matrix of $\boldsymbol{A}$ | 24 |
| $\boldsymbol{F}$ | the Fourier matrix | 46 |
| $\text{fft}\,(\boldsymbol{X})$ | 2D Fourier transform of the matrix $\boldsymbol{X}$. (When $\boldsymbol{X}$ is a vector, we unvectorize it before and vectorize it after the Fourier transform.) | 46 |
| $\text{vec}(\boldsymbol{X})$ | vectorized $\boldsymbol{X}$, i.e. the vector formed by stacking columns of $\boldsymbol{X}$ | 46 |
| $\text{Diag}\,(\boldsymbol{v})$ | the diagonal matrix with $\boldsymbol{v}$ on the diagonal | 46 |
| $\boldsymbol{x}.*\boldsymbol{y}$ | elementwise multiplication of $\boldsymbol{x}$ and $\boldsymbol{y}$. | 46 |

| | | |
|---|---|---|
| $\boldsymbol{\phi}$ | parameter (vectorized) of the PSF | 76 |
| $\boldsymbol{A}^\dagger$ | (regularized) pseudoinverse of matrix $\boldsymbol{A}$ | 79 |
| $\boldsymbol{p}$ | search direction in iterative methods | 81 |
| $\boldsymbol{r}$ | residual in iterative methods | 81 |
| $\boldsymbol{J}$ | Jacobian matrix of the residual $\boldsymbol{r}$ with respect to the parameter $\boldsymbol{\phi}$ | 81 |
| $\mathrm{Re}\,(X)$ | real part of $\boldsymbol{X}$ | 81 |
| $\hat{\boldsymbol{v}}$ | Fourier transform of $\boldsymbol{v}$ | 82 |
| $\boldsymbol{K}$ | Jacobian matrix of the deblurred image $\boldsymbol{x}$ with respect to the parameter $\boldsymbol{\phi}$ | 90 |
| $\mathrm{diag}\,(\boldsymbol{D})$ | the diagonal of $\boldsymbol{D}$. (When $\boldsymbol{D}$ is a 3-D tensor, $\mathrm{diag}\,(\boldsymbol{D})$ replaces the first two dimensions of $\boldsymbol{D}$ by the diagonal in these two dimensions.) | 90 |
| $\mathrm{ifft}\,(\boldsymbol{X})$ | 2D inverse Fourier transform of the matrix $\boldsymbol{X}$. (When $\boldsymbol{X}$ is a vector, we unvectorize it before and vectorize it after the Fourier transform.) | 93 |
| $\boldsymbol{h}$ | vectorized PSF | 100 |
| $\mathrm{Im}\,(X)$ | imaginary part of $\boldsymbol{X}$ | 101 |

## Remarks

1. We use boldface uppercase for matrices (e.g. $\boldsymbol{A}$), boldface lowercase for vectors (e.g. $\boldsymbol{x}$), and lightface with subscripts for their entries (e.g. $A_{i,j}$ and $x_k$). This makes it easy to distinguish scalars from vectors and matrices in formulae.

2. We use the same notation for both the 1D and 2D problems. For example $\boldsymbol{F}$ the Fourier matrix may represent the 1D or 2D Fourier matrix. The dimension should be clear from the context.

3. In the multi-image case, we use subscript $k$ to denote the variables for the $k$-th image. For example, $\boldsymbol{y}_k$ is the $k$-th blurred image.

# Chapter 1

# Introduction

The use of advanced imaging technologies is an integral part of scientific research, especially in fields such as biology, medicine and astronomy. Imaging is also an important component of modern security systems (e.g., video surveillance and biometric scanning), and is used to inspect machine parts (e.g, jet engine turbine blades) for possible small, but critical, defects.

Although physical limitations of imaging devices, as well as environmental effects, impede the ability to obtain perfect images, the resolution can often be improved through computational postprocessing techniques. In this dissertation we consider the particular, and commonly used, postprocessing technique of image deblurring with a spatially invariant blurring operator (i.e., deconvolution).

## 1.1 Convolution Model of Image Formation

Each pixel in a blurred image can be represented as a weighted average of pixels in the true image scene. The point spread function (PSF) defines these weights. The PSF can sometimes be obtained by calibration of the optical instrument, or expressed by a mathematical formula. If the PSF is assumed to be spatially invariant (as is often the case), then mathematically this image formation process is known as convolution, and can be written as

$$Y_{k,\ell} = \sum_{i,j} H_{i,j} X_{k-i,\ell-j} + N_{k,\ell}\,, \tag{1.1}$$

where $Y_{k,\ell}$ is a pixel of the observed image at the $(k, \ell)$ position, $H_{i,j}$ is the $(i, j)$ entry of the PSF, $X_{k-i,\ell-j}$ is a pixel of the exact original image at the $(k - i, \ell - j)$ position, and $N_{k,\ell}$ is additive noise. The additive noise may come from a combination of background noise, electronic sensor noise, modeling errors, measurement errors, etc. Since the focus of this dissertation is deblurring rather than denoising, without loss of generality, we can usually assume $N_{k,l}$ is zero when describing much of the mathematical and computational approaches proposed in this thesis. However, as we discuss later in this thesis, regularization must be incorporated into the algorithms to provide stability in the presence of noise.

Estimating the original image from the blurred image is called the deblurring problem. We can further classify deblurring problems into two subclasses: when the PSF is known, the deblurring problem is called deconvolution, and when the PSF is unknown, it is called blind deconvolution. We consider the deconvolution problem in Chapters 2 and 3, and the blind deconvolution problem in Chapter 4.

## 1.2 Literature Survey

A vast amount of work has been done on understanding and solving image deblurring problems. In particular, the classic book by Andrews and Hunt [1], published in 1977, provided the first comprehensive linear algebraic description of image deblurring. Since then, the problem is a standard topic in general books on image processing, such as the popular and often cited Gonzalez and Woods [32]. More recently, the book by Chan and Shen [10] provides a mathematical framework based on variational methods, and the book by Hansen, Nagy and O'Leary [37] focuses on regularization, spectral analysis and implementations.

Many different techniques have been developed for the deblurring problem. For example, we have the classical Wiener filtering [38], Bayesian based approaches [7, 43, 58], sparse representation methods [26], wavelet deblurring [8, 21], variational methods [59, 69], as well as general optimization [4] and linear algebra based [35, 48] approaches.

In terms of image processing software, there are both commercial and open-source options. The most popular would probably be the Image Processing Toolbox in Matlab [66]. Recently, Mathematica also included functions for image processing and analysis [72]. For open-source options, there are the multi-dimensional image processing subpackage of SciPy [65], RestoreTools for Matlab [45] and the deconvolution plugin for ImageJ [71].

## 1.3 Overview of this Dissertation

Researchers usually use several assumptions when they tackle the image deblurring problem. In particular, it is usually assumed that the blur (that is, the PSF) is known exactly, and that the true image scene outside the field of view is approximated well by periodic boundary conditions. These assumptions are certainly not true in most realistic situations.

In this thesis we develop a new method to derive boundary conditions directly from the blurred images. Compared with classical boundary conditions, our approach gives better deblurring results, especially for motion blurred images. To speed up the deblurring algorithms, we develop a new regularized preconditioning technique.

In deblurring, we often need to solve systems of linear equations. The matrices involved are usually huge, but with structure. Using object-oriented programming, these matrices do not need to be formed explicitly and efficient algorithms can be used for matrix operations with them. Object-oriented programming also facilitates the development of a web user interface to the deblurring algorithms. In this thesis we describe an object oriented implementation we developed using Python.

Recently, the use of graphical processing units (GPUs) in high performance computing has become popular. GPUs provide an economical way to attain supercomputing power. We describe an implementation that we developed, which extends the capabilities of our Python image deblurring software to effectively exploit modern GPU hardware.

In many situations, we do not know the PSFs in advance of deblurring. This kind of deblurring problem is called blind deconvolution. Using some auxiliary information, like a parametrized formula of the blurs and/or multiple images of the same object, and using numerical optimization methods, we are able to get back clear deblurred images. For this part, we focus on removing atmospheric blurs, which are common in astronomical imaging.

## 1.4 New Contributions

This thesis makes contributions on several aspects of the image deblurring problem, including modeling, algorithms, and software.

New synthetic boundary conditions are devised, including development of an efficient implementation. In addition, a new regularized DCT preconditioner is used for iterative deblurring algorithms when using synthetic boundary conditions. Extensive experiments presented in this thesis illustrate the effectiveness of synthetic boundary conditions and the regularized DCT preconditioner.

To facilitate research in image deblurring, two software packages, PYRET and PARRET, were developed. PYRET (Python RestoreTools), which uses object oriented programming in Python, is a serial implementation on CPUs;

PARRET (Parallel RestoreTools), which makes use of the computing power of GPUs, is a parallel implementation. A Web user interface has also been developed for PYRET. In the course of writing software for these packages, it was necessary to contribute a new complex branch to the open-source software PyCUDA, and to create Python wrappers so that CUBLAS and CUFFT libraries will work with PyCUDA. Benchmark results presented in this thesis show a significant speedup of the GPU implementation (PARRET) over the CPU implementation (PYRET).

For blind deconvolution, the variable projection technique is used to simplify the problem. The formulas involved are carefully derived using the spectral decomposition and two lemmas on conjugate symmetric vectors. Specific details are provided when tackling pupil phase blurs, especially on how to decompose the Jacobian matrix for fast multiplications. In addition, a new approach is proposed to provide a mathematical decoupling of the optimization problem when multiple frames from the same object are used. This approach leads to a block structure of the Jacobian matrix, which allows efficient multiplications. Numerical experiments show the benefits gained by using more than one frame.

# Chapter 2

# Synthetic Boundary Conditions

In this chapter we introduce a new boundary condition that can be used when reconstructing an image from observed blurred and noisy data. We provide an efficient algorithm for implementing the new boundary condition — synthetic boundary conditions, and provide a linear algebraic framework for the approach that puts it in the context of more classical and well known image boundary conditions.

Extensive numerical experiments show that our new synthetic boundary conditions provide a more accurate approximation of the true image scene outside the image boundary, and thus allow for better reconstructions of the unknown, true image scene.

## 2.1 Introduction

The image formation process is modeled as a convolution. Suppose that $\boldsymbol{Y}$ is an $n \times n$ image that is obtained by convolving the $m \times m$ *point spread function* (PSF) $\boldsymbol{H}$ with (an unknown, true) image $\boldsymbol{X}$, where $n \geqslant 2m + 1$. Then the convolution model implies that for each $k, \ell = 0, 1, \ldots, n-1$,

$$Y_{k,\ell} = \sum_{i=-m}^{m} \sum_{j=-m}^{m} H_{i,j} X_{k-i,\ell-j} \,. \tag{2.1}$$

We vectorize $\boldsymbol{X}$ and $\boldsymbol{Y}$ by stacking their columns together to obtain vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ respectively. Equation (2.1) can then be posed as a linear inverse problem,

$$\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}. \tag{2.2}$$

In general there is additive noise, but since the focus in this chapter is on boundary conditions, without loss of generality, we assume the noise is zero. Algorithms to solve (2.2) exploit the structure of $\boldsymbol{A}$, which depends on the PSF $\boldsymbol{H}$ and on the imposed boundary conditions. The convolution matrix $\boldsymbol{A}$ is often severely ill-conditioned, with singular values decaying to zero, without a significant gap to indicate numerical rank. The deblurring problem is, given $\boldsymbol{A}$ and $\boldsymbol{y}$, compute an approximation of $\boldsymbol{x}$.

In general,

$$\boldsymbol{A} = \boldsymbol{T} + \boldsymbol{B} \tag{2.3}$$

where $\boldsymbol{T}$ has a Toeplitz structure and $\boldsymbol{B}$, which is defined by the boundary conditions, is often structured, sparse, and low rank.

In the discrete setting, the matrix $\boldsymbol{X}$ contains pixel values of the true (unknown) image scene on a bounded domain. Boundary conditions make assumptions about how the image behaves outside the field of view, and they are often chosen for algebraic and computational convenience.

For example, periodic boundary conditions result in a matrix $\boldsymbol{A}$ that has a circulant structure, which is diagonalized by the unitary discrete Fourier transform matrix [16]. It is well known that computations with such matrices can be done very efficiently by using fast Fourier transforms (FFT) [15, 67]. Note that periodic boundary conditions assume that the true infinite scene can be represented as a mosaic of a single finite dimensional image, repeated periodically in all directions. Thus, although computationally convenient, for most images it is difficult to provide a physical justification for the use of periodic boundary conditions.

Other boundary conditions can have better physical justification. For example, if the image is assumed to have a black background (such as in the case of astronomical images), then zero boundary conditions may provide a good physical representation for the image scene outside the viewable region. In this case $\boldsymbol{B}$ is zero, and thus $\boldsymbol{A}$ has a Toeplitz structure. Although direct filtering type methods cannot be implemented as efficiently as in the case of circulant structures, it is possible to effectively use iterative methods [42, 44]. However, if there are significant features near the image boundary, then zero boundary conditions may not provide a physically accurate model of the infinite scene.

If there are significant features that overlap the edge of the viewable region, then it may make sense to use reflective boundary conditions, where it is assumed that the scene outside the viewable region is a mirror reflection of the scene inside the viewable region [48]. In this case the matrix $\boldsymbol{A}$ has a Toeplitz-plus-Hankel structure. Iterative methods for such matrices can be implemented efficiently, and, moreover, if the PSF satisfies a strong symmetry condition, $\boldsymbol{A}$ can be diagonalized by the orthogonal discrete cosine transformation matrix, and spectral filtering methods can be implemented very efficiently [37]. With reflective boundary conditions, continuity of the graylevel values of the image is maintained.

More recently, anti-reflective boundary conditions have been proposed, which extend the pixel values across the boundary in such a way that continuity of the image and of the normal derivative are preserved at the boundary [18, 19, 63]. In this case the structure of $\boldsymbol{A}$ is Toeplitz-plus-Hankel, plus an additional structured low rank matrix. As with reflective boundary conditions, iterative methods for such matrices can be implemented efficiently, and, moreover, if the PSF satisfies a strong symmetry condition, spectral filtering methods can be implemented very efficiently (though the details are a bit more complicated); see [2] for more details.

In this chapter we propose a new approach, which we call synthetic boundary conditions. Our goal is to not necessarily continue graylevels at the boundary, but instead to develop a scheme that can continue edge directions and textures of the image inside the viewable region to outside the image boundary. We remark that, although our discussion is for the specific problem of image deblurring (deconvolution), the approach we propose in this chapter can be used in other imaging applications as well.

## 2.2 Image Deblurring and Boundary Conditions

In this section we review some classical boundary conditions that are commonly used in imaging deblurring. We illustrate that in each case the matrix $\boldsymbol{A}$ in equation (2.2) can be put in the form given by equation (2.3).

To simplify the discussion, we begin by describing matrix structures for one dimensional problems. We then extend the discussion to two dimensional problems. Finally we propose a new approach that uses information from the observed image to enforce continuity of image features such as edges and texture across the boundary.

### 2.2.1 One Dimensional Problems

We begin with the one dimensional problem because the matrix descriptions are easier to follow. The two dimensional problem is discussed in the next subsection. We use notation similar to that in [48]. Suppose $\boldsymbol{y}$ is a one dimensional image (i.e., a signal) that is obtained by convolving the PSF $\boldsymbol{h}$

with (an unknown, true) signal $\boldsymbol{x}_{\text{true}}$, where

$$\boldsymbol{y} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix} \quad \text{and} \quad \boldsymbol{h} = \begin{bmatrix} h_{-m} \\ \vdots \\ h_{-1} \\ h_0 \\ h_1 \\ \vdots \\ h_m \end{bmatrix}$$

and $n \geqslant 2m + 1$. Then the convolution model implies that for each $k = 0, 1, \ldots, n-1$,

$$y_k = \sum_{i=-m}^{m} h_i x_{k-i}\,,$$

which can be written in matrix-vector form as

$$\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} h_m & \cdots & h_0 & \cdots & h_{-m} & & & & & & \\ & \ddots & \vdots & \ddots & \vdots & \ddots & & & & & \\ & & h_m & & h_0 & & h_{-m} & & & & \\ & & & \ddots & \vdots & \ddots & \vdots & \ddots & & & \\ & & & & h_m & & h_0 & & h_{-m} & & \\ & & & & & \ddots & \vdots & \ddots & \vdots & \ddots & \\ & & & & & & h_m & \cdots & h_0 & \cdots & h_{-m} \end{bmatrix} \begin{bmatrix} x_{-m} \\ \vdots \\ x_{-1} \\ \hline x_0 \\ \vdots \\ x_{n-1} \\ \hline x_n \\ \vdots \\ x_{n-1+m} \end{bmatrix} \tag{2.4}$$

where we use horizontal lines in $\boldsymbol{x}_{\text{true}}$ to denote the boundaries of the field of view in the true image scene, which correspond to those of the observed signal $\boldsymbol{y}$. Although $m$ is generally small compared to $n$, the problem is underdetermined since values of $\boldsymbol{y}$ near the boundary (such as $y_0$ and $y_{n-1}$) depend on values of $\boldsymbol{x}_{\text{true}}$ outside the field of view.

It will be convenient to rewrite equation (2.4) as

$$\boldsymbol{y} = \left[\begin{array}{c|c|c} \boldsymbol{T}_{-1} & \boldsymbol{T} & \boldsymbol{T}_1 \end{array}\right] \left[\begin{array}{c} \boldsymbol{x}_{-1} \\ \hline \boldsymbol{x} \\ \hline \boldsymbol{x}_1 \end{array}\right]$$

where $\boldsymbol{T}_{-1}$, $\boldsymbol{T}$ and $\boldsymbol{T}_1$ are the following Toeplitz matrices

$$\left[\begin{array}{ccc|ccccccccc|ccc}
\multicolumn{3}{c}{\overbrace{\hphantom{h_m \cdots h_1}}^{\mathbf{T}_{-1}}} & \multicolumn{9}{c}{\overbrace{\hphantom{h_0 \cdots \cdots h_{-m} \quad\quad\quad\quad\quad\quad\quad\quad}}^{\mathbf{T}}} & \multicolumn{3}{c}{\overbrace{\hphantom{h_{-1} \cdots h_{-m}}}^{\mathbf{T}_1}} \\
h_m & \cdots & h_1 & h_0 & \cdots & \cdots & h_{-m} & & & & & & & & \\
 & \ddots & \vdots & \vdots & \ddots & & \vdots & \ddots & & & & & & & \\
 & & h_m & \vdots & & \ddots & \vdots & & h_{-m} & & & & & & \\
 & & & h_m & & & h_0 & & \vdots & \ddots & & & & & \\
 & & & & \ddots & & \vdots & \ddots & \vdots & & \ddots & & & & \\
 & & & & & \ddots & \vdots & & h_0 & & & h_{-m} & & & \\
 & & & & & & h_m & & \vdots & \ddots & & \vdots & h_{-m} & & \\
 & & & & & & & \ddots & \vdots & & \ddots & \vdots & \vdots & \ddots & \\
 & & & & & & & & h_m & \cdots & \cdots & h_0 & h_{-1} & \cdots & h_{-m}
\end{array}\right] \tag{2.5}$$

and

$$\boldsymbol{x}_{-1} = \begin{bmatrix} x_{-m} \\ \vdots \\ x_{-1} \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} x_0 \\ \vdots \\ x_{n-1} \end{bmatrix}, \quad \boldsymbol{x}_1 = \begin{bmatrix} x_n \\ \vdots \\ x_{n-1+m} \end{bmatrix}.$$

Since $\boldsymbol{x}_{-1}$ and $\boldsymbol{x}_1$ are outside the field of view, and are therefore not measurable, boundary conditions replace these with values that can be either set *a priori* or obtained from information within the field of view. Specifically, $\boldsymbol{x}_{-1}$ and $\boldsymbol{x}_1$ are replaced with

$$\hat{\boldsymbol{x}}_{-1} = \boldsymbol{S}_{-1}\boldsymbol{x} \quad \text{and} \quad \hat{\boldsymbol{x}}_1 = \boldsymbol{S}_1\boldsymbol{x}\,,$$

where $\boldsymbol{S}_{-1}$ and $\boldsymbol{S}_1$ are matrices defined by the boundary conditions (specific examples are given below). With this notation, equation (2.4) is approximated by

$$\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}\,, \tag{2.6}$$

where $\boldsymbol{A} = \boldsymbol{T} + \boldsymbol{B}$ and $\boldsymbol{B} = \boldsymbol{T}_{-1}\boldsymbol{S}_{-1} + \boldsymbol{T}_1\boldsymbol{S}_1$. Some well-known examples include:

- For *zero* boundary conditions it is assumed that the signal is always zero outside the field of view; that is, $\hat{\boldsymbol{x}}_{-1} = \hat{\boldsymbol{x}}_1 = \boldsymbol{0}$. In this case, $\boldsymbol{S}_{-1} = \boldsymbol{S}_1 = \boldsymbol{O}$, where $\boldsymbol{O}$ is a matrix of all zeros. Thus $\boldsymbol{B} = \boldsymbol{O}$, and $\boldsymbol{A} = \boldsymbol{T}$.

- For *periodic* boundary conditions we use

$$\hat{\boldsymbol{x}}_{-1} = \begin{bmatrix} x_{n-m} \\ x_{n-m+1} \\ \vdots \\ x_{n-1} \end{bmatrix} \quad \text{and} \quad \hat{\boldsymbol{x}}_1 = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{m-1} \end{bmatrix}.$$

  Thus, $\boldsymbol{S}_{-1} = \begin{bmatrix} \boldsymbol{O} & \boldsymbol{I} \end{bmatrix}$ and $\boldsymbol{S}_1 = \begin{bmatrix} \boldsymbol{I} & \boldsymbol{O} \end{bmatrix}$, where $\boldsymbol{O}$ is a matrix of all zeros, and $\boldsymbol{I}$ is an $m \times m$ identity matrix. In this case, $\boldsymbol{B} = \begin{bmatrix} \boldsymbol{O} & \boldsymbol{T}_{-1} \end{bmatrix} + \begin{bmatrix} \boldsymbol{T}_1 & \boldsymbol{O} \end{bmatrix}$, and it is not difficult to show that $\boldsymbol{A} = \boldsymbol{T} + \boldsymbol{B}$ is a circulant matrix. Note that rank($\boldsymbol{B}$) $= 2m$, which is (often much) less than $n$.

- For *reflective* boundary conditions we use

$$\hat{\boldsymbol{x}}_{-1} = \begin{bmatrix} x_{m-1} \\ \vdots \\ x_1 \\ x_0 \end{bmatrix} \quad \text{and} \quad \hat{\boldsymbol{x}}_1 = \begin{bmatrix} x_{n-1} \\ \vdots \\ x_{n-m+1} \\ x_{n-m} \end{bmatrix}.$$

Thus, $\boldsymbol{S}_{-1} = \begin{bmatrix} \boldsymbol{O} & \boldsymbol{I} \end{bmatrix} \boldsymbol{J}$ and $\boldsymbol{S}_1 = \begin{bmatrix} \boldsymbol{I} & \boldsymbol{O} \end{bmatrix} \boldsymbol{J}$, where $\boldsymbol{J}$ is the "reversal" permutation matrix,

$$\boldsymbol{J} = \begin{bmatrix} & & 1 \\ & \iddots & \\ 1 & & \end{bmatrix}.$$

In this case, $\boldsymbol{B} = \begin{bmatrix} \boldsymbol{O} & \boldsymbol{T}_{-1} \end{bmatrix} \boldsymbol{J} + \begin{bmatrix} \boldsymbol{T}_1 & \boldsymbol{O} \end{bmatrix} \boldsymbol{J}$ is a Hankel matrix, and so $\boldsymbol{A} = \boldsymbol{T} + \boldsymbol{B}$ is a Toeplitz-plus-Hankel matrix. Again we see that the rank$(\boldsymbol{B}) = 2m$.

- For *anti-reflective* boundary conditions, originally proposed by Serra-Capizzano [63], we use

$$\hat{\boldsymbol{x}}_{-1} = \begin{bmatrix} 2x_0 - x_m \\ \vdots \\ 2x_0 - x_2 \\ 2x_0 - x_1 \end{bmatrix} \quad \text{and} \quad \hat{\boldsymbol{x}}_1 = \begin{bmatrix} 2x_{n-1} - x_{n-2} \\ \vdots \\ 2x_{n-1} - x_{n-m} \\ 2x_{n-1} - x_{n-m-1} \end{bmatrix}.$$

  Thus, $\boldsymbol{S}_{-1} = \begin{bmatrix} \boldsymbol{O} & -\boldsymbol{I} & 2\,\mathbf{e} \end{bmatrix} \boldsymbol{J}$ and $\boldsymbol{S}_1 = \begin{bmatrix} 2\,\mathbf{e} & -\boldsymbol{I} & \boldsymbol{O} \end{bmatrix} \boldsymbol{J}$, where $\mathbf{e}$ is a vector of ones. In this case $\boldsymbol{B} = \begin{bmatrix} \boldsymbol{O} & -\boldsymbol{T}_{-1} & 2\boldsymbol{T}_{-1}\mathbf{e} \end{bmatrix} \boldsymbol{J} + \begin{bmatrix} 2\boldsymbol{T}_1\mathbf{e} & -\boldsymbol{T}_1 & \boldsymbol{O} \end{bmatrix} \boldsymbol{J}$ is the sum of a Hankel matrix and a matrix with rank equal to two. The matrix $\boldsymbol{A} = \boldsymbol{T} + \boldsymbol{B}$ is then Toeplitz-plus-Hankel, plus an additional rank-2 matrix. Note that in this case the rank$(\boldsymbol{B}) = 2m + 2$.

Observe that in all of the above examples, the one dimensional deblurring problem can be represented as

$$\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}\,, \quad \boldsymbol{A} = \boldsymbol{T} + \boldsymbol{B}$$

where $\boldsymbol{T}$ is a Toeplitz matrix, and $\boldsymbol{B}$ is a matrix defined by the boundary condition, which is structured, and if $m \ll n$, also sparse and low rank. This linear algebra formulation can be extended to higher dimensions.

### 2.2.2 Two Dimensional Problems

Extending this linear algebraic formulation to two dimensional imaging problems is not so difficult, but the notation can be a bit cumbersome. To facilitate readability, we assume all images are square (e.g., $n \times n$) arrays of pixel values, and that the PSF is separable.

Suppose that $\boldsymbol{Y}$ is an $n \times n$ image that is obtained by convolving the $m \times m$ PSF $\boldsymbol{H}$ with (an unknown, true) image $\boldsymbol{X}_{\text{true}}$, where $n \geqslant 2m + 1$. Then the convolution model implies that for each $k, \ell = 0, 1, \ldots, n-1$,

$$Y_{k,\ell} = \sum_{i=-m}^{m} \sum_{j=-m}^{m} H_{i,j} X_{k-i,\ell-j} \,. \tag{2.7}$$

If the PSF is separable (i.e., the vertical blurring operation is independent of the horizontal blurring operation), then there are vectors

$$\boldsymbol{h}_c = \begin{bmatrix} h^{(c)}_{-m} \\ \vdots \\ h^{(c)}_{-1} \\ h^{(c)}_{0} \\ h^{(c)}_{1} \\ \vdots \\ h^{(c)}_{m} \end{bmatrix} \quad \text{and} \quad \boldsymbol{h}_r = \begin{bmatrix} h^{(r)}_{-m} \\ \vdots \\ h^{(r)}_{-1} \\ h^{(r)}_{0} \\ h^{(r)}_{1} \\ \vdots \\ h^{(r)}_{m} \end{bmatrix}$$

such that

$$\boldsymbol{H} = \boldsymbol{h}_c \boldsymbol{h}_r^T \quad \Leftrightarrow \quad H_{i,j} = h^{(c)}_i h^{(r)}_j \,,$$

where $\boldsymbol{h}_c$ and $\boldsymbol{h}_r$ represent, respectively, the vertical and horizontal components of the PSF [37]. In this case, the convolution equation (2.7) becomes

$$\begin{aligned} Y_{k,\ell} &= \sum_{i=-m}^{m} \sum_{j=-m}^{m} H_{i,j} X_{k-i,\ell-j} \\ &= \sum_{i=-m}^{m} \sum_{j=-m}^{m} h_i^{(c)} h_j^{(r)} X_{k-i,\ell-j} \\ &= \sum_{i=-m}^{m} \left( h_i^{(c)} \sum_{j=-m}^{m} \left( X_{k-i,\ell-j} h_j^{(r)} \right) \right) , \end{aligned}$$

which can be written in matrix-vector form as

$$\boldsymbol{Y} = \begin{bmatrix} \boldsymbol{T}_{c,-1} & \boldsymbol{T}_c & \boldsymbol{T}_{c,1} \end{bmatrix} \begin{bmatrix} \boldsymbol{X}_{-1,-1} & \boldsymbol{X}_{-1,0} & \boldsymbol{X}_{-1,1} \\ \boldsymbol{X}_{0,-1} & \boldsymbol{X} & \boldsymbol{X}_{0,1} \\ \boldsymbol{X}_{1,-1} & \boldsymbol{X}_{1,0} & \boldsymbol{X}_{1,1} \end{bmatrix} \begin{bmatrix} \boldsymbol{T}_{r,-1}^T \\ \boldsymbol{T}_r^T \\ \boldsymbol{T}_{r,1}^T \end{bmatrix} , \qquad (2.8)$$

where $\begin{bmatrix} \boldsymbol{T}_{c,-1} & \boldsymbol{T}_c & \boldsymbol{T}_{c,1} \end{bmatrix}$ and $\begin{bmatrix} \boldsymbol{T}_{r,-1} & \boldsymbol{T}_r & \boldsymbol{T}_{r,1} \end{bmatrix}$ are identical in structure to the matrices given in equation (2.5), $\boldsymbol{X}$ is the $n \times n$ portion of the true image scene within the field of view (defined by $\boldsymbol{Y}$), and $\boldsymbol{X}_{i,j}$ represent sections of the scene that are outside the field of view.

As in the one dimensional model, since $\boldsymbol{X}_{i,j}$ are outside the field of view, we use boundary conditions to replace these with values that are either set *a priori* (e.g., to zero), or with values that can be obtained from information within the field of view. That is, the array representing the true image scene

$$\boldsymbol{X}_{\text{true}} = \begin{bmatrix} \boldsymbol{X}_{-1,-1} & \boldsymbol{X}_{-1,0} & \boldsymbol{X}_{-1,1} \\ \boldsymbol{X}_{0,-1} & \boldsymbol{X} & \boldsymbol{X}_{0,1} \\ \boldsymbol{X}_{1,-1} & \boldsymbol{X}_{1,0} & \boldsymbol{X}_{1,1} \end{bmatrix}$$

is replaced with

$$\begin{bmatrix} \widehat{\boldsymbol{X}}_{-1,-1} & \widehat{\boldsymbol{X}}_{-1,0} & \widehat{\boldsymbol{X}}_{-1,1} \\ \widehat{\boldsymbol{X}}_{0,-1} & \boldsymbol{X} & \widehat{\boldsymbol{X}}_{0,1} \\ \widehat{\boldsymbol{X}}_{1,-1} & \widehat{\boldsymbol{X}}_{1,0} & \widehat{\boldsymbol{X}}_{1,1} \end{bmatrix} = \begin{bmatrix} \boldsymbol{S}_{c,-1}\boldsymbol{X}\boldsymbol{S}_{r,-1}^T & \boldsymbol{S}_{c,-1}\boldsymbol{X} & \boldsymbol{S}_{c,-1}\boldsymbol{X}\boldsymbol{S}_{r,1}^T \\ \boldsymbol{X}\boldsymbol{S}_{r,-1}^T & \boldsymbol{X} & \boldsymbol{X}\boldsymbol{S}_{r,1}^T \\ \boldsymbol{S}_{c,1}\boldsymbol{X}\boldsymbol{S}_{r,-1}^T & \boldsymbol{S}_{c,1}\boldsymbol{X} & \boldsymbol{S}_{c,1}\boldsymbol{X}\boldsymbol{S}_{r,1}^T \end{bmatrix} = \begin{bmatrix} \boldsymbol{S}_{c,-1} \\ \boldsymbol{I} \\ \boldsymbol{S}_{c,1} \end{bmatrix} \boldsymbol{X} \begin{bmatrix} \boldsymbol{S}_{r,-1}^T & \boldsymbol{I} & \boldsymbol{S}_{r,1}^T \end{bmatrix}$$

where $\boldsymbol{S}_{c,-1}$ and $\boldsymbol{S}_{c,1}$ define the vertical boundary conditions (i.e., those imposed at the top and bottom of the image), and $\boldsymbol{S}_{r,-1}$ and $\boldsymbol{S}_{r,1}$ define the horizontal boundary conditions (i.e., those imposed at the left and right of the image).

We remark that our approach to defining boundary conditions does not require a separable PSF. However, if the blur is separable, then equation (2.8) can be approximated with

$$\begin{aligned} \boldsymbol{Y} &= \begin{bmatrix} \boldsymbol{T}_{c,-1} & \boldsymbol{T}_c & \boldsymbol{T}_{c,1} \end{bmatrix} \begin{bmatrix} \boldsymbol{S}_{c,-1} \\ \boldsymbol{I} \\ \boldsymbol{S}_{c,1} \end{bmatrix} \boldsymbol{X} \begin{bmatrix} \boldsymbol{S}_{r,-1}^T & \boldsymbol{I} & \boldsymbol{S}_{r,1}^T \end{bmatrix} \begin{bmatrix} \boldsymbol{T}_{r,-1}^T \\ \boldsymbol{T}_r^T \\ \boldsymbol{T}_{r,1}^T \end{bmatrix} \\ &= \left(\boldsymbol{T}_{c,-1}\boldsymbol{S}_{c,-1} + \boldsymbol{T}_c + \boldsymbol{T}_{c,1}\boldsymbol{S}_{c,1}\right) \boldsymbol{X} \left(\boldsymbol{S}_{r,-1}^T\boldsymbol{T}_{r,-1}^T + \boldsymbol{T}_r^T + \boldsymbol{S}_{r,1}^T\boldsymbol{T}_{r,1}^T\right), \end{aligned}$$

or, equivalently, we can write this in matrix-vector form as

$$\begin{aligned} \boldsymbol{y} &= \Big( (\boldsymbol{T}_{r,-1}\boldsymbol{S}_{r,-1} + \boldsymbol{T}_r + \boldsymbol{T}_{r,1}\boldsymbol{S}_{r,1}) \otimes (\boldsymbol{T}_{c,-1}\boldsymbol{S}_{c,-1} + \boldsymbol{T}_c + \boldsymbol{T}_{c,1}\boldsymbol{S}_{c,1}) \Big) \boldsymbol{x} \\ &= \Big( \boldsymbol{T}_r \otimes \boldsymbol{T}_c + \Big[ \boldsymbol{T}_r \otimes (\boldsymbol{T}_{c,-1}\boldsymbol{S}_{c,-1} + \boldsymbol{T}_{c,1}\boldsymbol{S}_{c,1}) \\ &\qquad + (\boldsymbol{T}_{r,-1}\boldsymbol{S}_{r,-1} + \boldsymbol{T}_{r,1}\boldsymbol{S}_{r,1}) \otimes (\boldsymbol{T}_{c,-1}\boldsymbol{S}_{c,-1} + \boldsymbol{T}_c + \boldsymbol{T}_{c,1}\boldsymbol{S}_{c,1}) \Big] \Big) \boldsymbol{x} \end{aligned}$$

where $\otimes$ denotes Kronecker product, and $\boldsymbol{y} = \text{vec}(\boldsymbol{Y})$ and $\boldsymbol{x} = \text{vec}(\boldsymbol{X})$. Again we see that the image deblurring problem with spatially invariant, separable blur, can be represented as

$$\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}\,, \quad \boldsymbol{A} = \boldsymbol{T} + \boldsymbol{B}$$

where $\boldsymbol{T} = \boldsymbol{T}_r \otimes \boldsymbol{T}_c$ is a block Toeplitz matrix with Toeplitz blocks (BTTB), and $\boldsymbol{B} = \boldsymbol{T}_r \otimes (\boldsymbol{T}_{c,-1}\boldsymbol{S}_{c,-1} + \boldsymbol{T}_{c,1}\boldsymbol{S}_{c,1}) + (\boldsymbol{T}_{r,-1}\boldsymbol{S}_{r,-1} + \boldsymbol{T}_{r,1}\boldsymbol{S}_{r,1}) \otimes (\boldsymbol{T}_{c,-1}\boldsymbol{S}_{c,-1} + \boldsymbol{T}_c + \boldsymbol{T}_{c,1}\boldsymbol{S}_{c,1})$ is defined by the boundary conditions. Note that if the blur is not separable, then we do not get neat Kronecker product decompositions of $\boldsymbol{T}$ and $\boldsymbol{B}$, but we still get the basic form where $\boldsymbol{T}$ is BTTB and $\boldsymbol{B}$ is a structured (and typically sparse) matrix.

All of the boundary conditions discussed in the previous subsection for one dimensional problems extend naturally to the two dimensional problem. For example, in the case of periodic boundary conditions, we use

$$\boldsymbol{S}_{c,-1} = \boldsymbol{S}_{r,-1} = \begin{bmatrix} \boldsymbol{O} & \boldsymbol{I} \end{bmatrix} \quad \text{and} \quad \boldsymbol{S}_{c,1} = \boldsymbol{S}_{r,1} = \begin{bmatrix} \boldsymbol{I} & \boldsymbol{O} \end{bmatrix},$$

resulting in a block circulant matrix with circulant blocks (BCCB).

In the next subsection we propose a new boundary condition that is more effective at continuing edges and texture of the image across the boundary than zero, periodic, reflective, and anti-reflective approaches.

### 2.2.3 Synthetic Boundary Conditions

As mentioned in Section 2.1, it is unlikely that a true image scene would be modeled well by periodic boundary conditions, and zero boundary conditions only make sense for scenes with a black background. There may be some rare cases when reflective and anti-reflective boundary conditions provide a good model of the true image scene outside the field of view. In this subsection we develop an approach that provides a more realistic extension of pixels across the boundary. For example, texture and edges should be extended sensibly. The motivation for our approach comes from observing that the problem of defining appropriate boundary conditions is similar to the image recovery problem, in which part of the image is damaged and the aim is to recover missing pixels. In our case, the region we wish to recover corresponds to those pixels outside the boundary. Two common approaches for the image recovery problem are image inpainting [5] and texture synthesis [24]. Image inpainting tries to extend the geometric structure of the image, while texture synthesis extends the texture pattern into the unknown region. In this chapter we use the texture synthesis approach.

With the image recovery idea in mind, we wish to determine a relationship between (unknown) pixel values outside the boundary to those pixel values inside the boundary. Using a basic texture synthesis approach, we can try to find a pixel in the viewable region whose neighborhood (e.g., a rectangular region) is most similar to the corresponding neighborhood of the boundary pixel we wish to fill in. If this idea is applied to a blurred image, it can extend edges across the boundary well, but there is little hope that it can also extend the texture, as texture information is lost in blurring. Hence, instead of copying single pixels, we propose to copy small patches that contain the required texture information. This idea is similar to the generalization of texture synthesis to image quilting [23].

To describe more precisely our approach for synthetic boundary conditions, we need a bit of notation. Let

$$\begin{aligned} \mathcal{D} &= [0, n-1] \times [0, n-1] && \text{(domain)} \\ \mathcal{B} &= ([-m, n+m-1] \times [-m, n+m-1]) \setminus \mathcal{D} && \text{(border)} \end{aligned} \tag{2.9}$$

The algorithm to obtain the synthetic boundary conditions for pixels in $\mathcal{B}$ is given in Algorithm 2.1. This patch-based texture synthesis idea is also illustrated in Figure 2.1. Larger patches can be used, but we have found that $2 \times 2$ patches work well.

An example of padding with synthetic boundary conditions compared to the padding used in other boundary conditions is shown in Figure 2.2 (the

**Algorithm 2.1** Obtaining synthetic boundary conditions.

```
for all $[i, i+1] \times [j, j+1]$ patch $\in \mathcal{B}$ do
    • Find
          $$(k_{\min}, \ell_{\min}) = \arg\min_{k,\ell} \mathrm{SSD}(\mathrm{nbhd}(i,j), \mathrm{nbhd}(k,\ell))$$
      where $\mathrm{nbhd}(i,j)$ is a pixel neighborhood at $(i,j)$, SSD is the
      sum of squared differences between pixels in the two specified
      neighborhoods and the search is over a region near $(i,j)$ in
      $\mathcal{D} \cup \{\text{pixels already processed}\}$.
    • Set the boundary pixels in the $2 \times 2$ patch to be
          $X_{i,j} = X_{k_{\min},\ell_{\min}}$
          $X_{i+1,j} = X_{k_{\min}+1,\ell_{\min}}$
          $X_{i,j+1} = X_{k_{\min},\ell_{\min}+1}$
          $X_{i+1,j+1} = X_{k_{\min}+1,\ell_{\min}+1}$
end for
```

left column shows the full image with padded boundaries, the center column shows a zoom in on the upper left corner, and the right column shows a zoom in on the upper right corner of the image). This figure clearly illustrates that zero and periodic boundary conditions do not preserve continuity of pixel values. Reflective boundary conditions result in continuity of the pixel values across the boundary, but the derivatives of the gray level perpendicular to the image boundary are fixed to be zero. Anti-reflective boundary conditions allow for continuity of the pixel values as well as the derivatives across the boundary. Synthetic boundary conditions do not strive (at least not directly) to maintain continuity, but instead the aim is to match neighborhoods of pixel

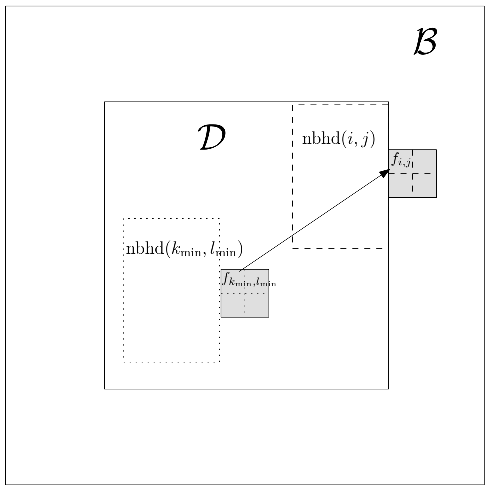


Figure 2.1: Illustration of how the synthetic boundary condition is determined. Specifically, $(k_{\min}, l_{\min}) = \arg\min_{k,l} \mathrm{SSD}(\mathrm{nbhd}\,(i,j)\,, \mathrm{nbhd}\,(k,l))$.

values. Figure 2.2 clearly shows that synthetic boundary conditions are much better at extending edges (e.g. of the books in the zoom of the upper left corner) and texture (e.g., of the chair in the zoom of the upper right corner).

The matrix $\boldsymbol{A}$ for synthetic boundary conditions is similar to the periodic and reflective cases because the pixels in $\mathcal{B}$ are simply copies of pixels in $\mathcal{D}$. Thus, they can be obtained by permutation. To see this, consider again the situation when the blur is separable. Then we can write the matrix-vector model as

$$\boldsymbol{y} = \left( \begin{bmatrix} \boldsymbol{T}_{r,-1} & \boldsymbol{T}_r & \boldsymbol{T}_{r,1} \end{bmatrix} \otimes \begin{bmatrix} \boldsymbol{T}_{c,-1} & \boldsymbol{T}_c & \boldsymbol{T}_{c,1} \end{bmatrix} \right) \boldsymbol{P}\boldsymbol{x}$$

where

$$\boldsymbol{P} = \begin{bmatrix} \boldsymbol{S}_{r,-1} \\ \boldsymbol{I} \\ \boldsymbol{S}_{r,1} \end{bmatrix} \otimes \begin{bmatrix} \boldsymbol{S}_{c,-1} \\ \boldsymbol{I} \\ \boldsymbol{S}_{c,1} \end{bmatrix} .$$

Thus, in the case of periodic and reflective boundary conditions, $\boldsymbol{P}$ is simply a highly structured permutation matrix, which only allows to grab entries

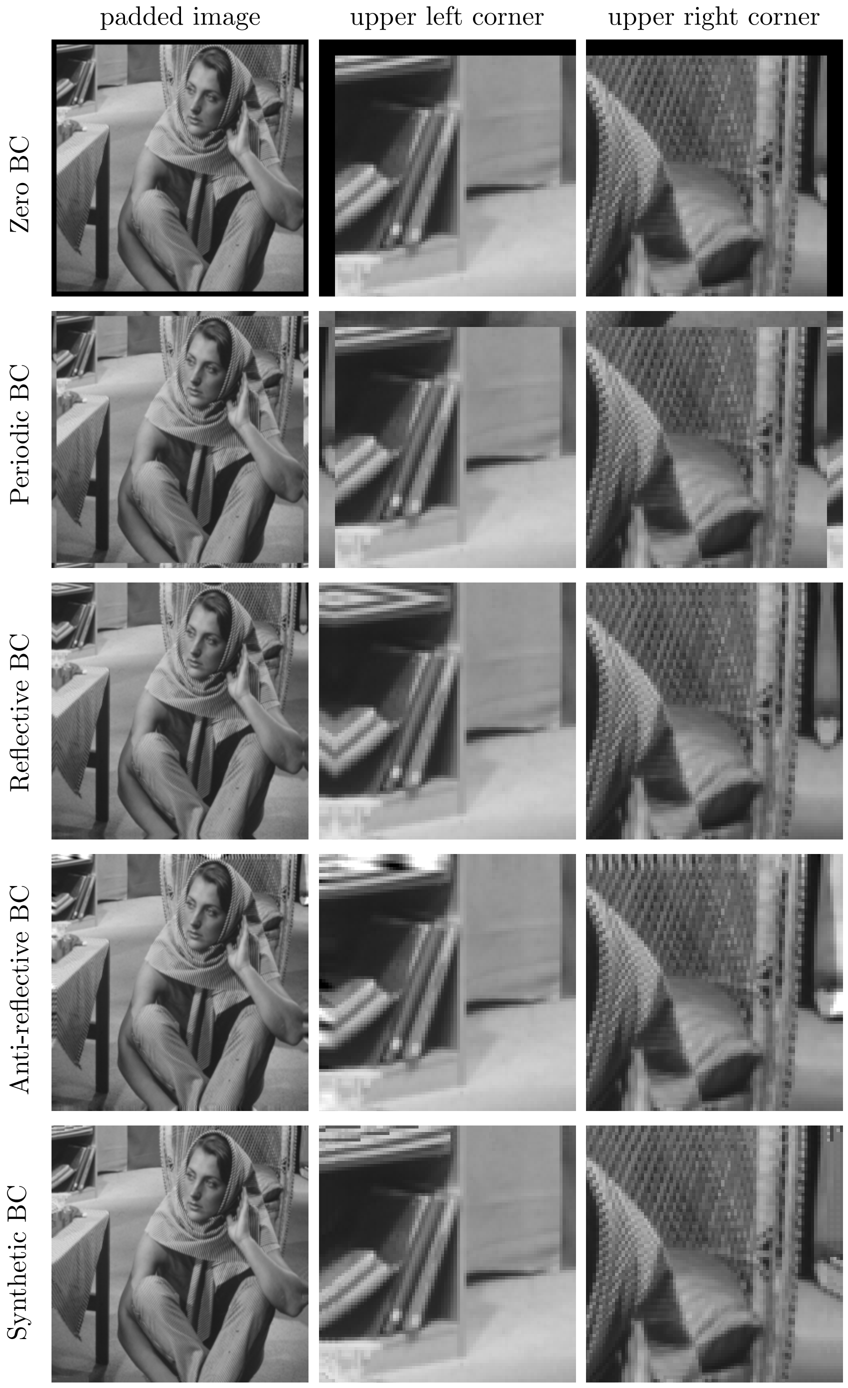


Figure 2.2: Padded results with different boundary conditions

from restricted regions of the viewable region. With the use of synthetic boundary conditions we relax the structure of $\boldsymbol{P}$, and allow the permutation matrix to grab entries more flexibly in the viewable region. The result, as illustrated in Figure 2.2 is a much better representation of the edges and texture of the image across the boundary.

We emphasize that synthetic boundary conditions are image dependent, and are therefore more capable of extending image features. However, an additional step is needed to estimate the boundary conditions. For efficient implementation, the search for $(k_{\min}, l_{\min})$ can be done only over nearby pixels of $(i, j)$, rather than over the whole image. This makes sense since pixels with similar image features (e.g. edge directions, texture) are usually close to each other. For a fixed size for $\mathrm{nbhd}(i, j)$ and $\mathrm{nbhd}(k, l)$ and a fixed size for the search pool, the cost of enforcing synthetic boundary conditions is proportional to the number of pixels to be filled in the border area, i.e. $O(mn)$ for an image with $m \times n$ pixels. Further computational savings in the implementation, similar to that in [23], can be obtained by reusing intermediate values of $\mathrm{SSD}(\mathrm{nbhd}(i, j), \mathrm{nbhd}(k, l))$. We remark that the cost of obtaining the boundary conditions is negligible compared with that of the subsequent iterative methods to deblur the image.

## 2.3 Preconditioners for Synthetic Boundary Conditions

For synthetic boundary conditions, the matrix $\boldsymbol{A}$ does not have the kind of structure that allows efficient implementation of direct filtering type methods. This is similar to the situation when zero boundary conditions are used, or when reflective and anti-reflective boundary conditions are used with a non-symmetric PSF. In these situations it is necessary to use iterative methods, such as a conjugate gradient type approach (e.g., CG, MINRES, or LSQR). We remark that for conjugate gradient type methods, the matrix $\boldsymbol{A}$ need not be formed explicitly, all that is needed is an efficient approach to compute matrix-vector multiplications with $\boldsymbol{A}$. This can be done by exploiting the structure of $\boldsymbol{A} = \boldsymbol{T} + \boldsymbol{B}$; FFTs can be used to multiply $\boldsymbol{T}$ by accessing only the PSF, and $\boldsymbol{B}$ is a sparse matrix. Or, alternatively, FFTs can be used on appropriately padded image arrays. The latter is used in our implementation.

The next issue, then, is to consider preconditioning. Note that for the various boundary conditions considered in this chapter, we have:

$$\begin{aligned} \text{Zero BC:} \quad & \boldsymbol{A}_Z = \boldsymbol{T} + \boldsymbol{B}_Z \\ \text{Periodic BC:} \quad & \boldsymbol{A}_P = \boldsymbol{T} + \boldsymbol{B}_P \\ \text{Reflective BC:} \quad & \boldsymbol{A}_R = \boldsymbol{T} + \boldsymbol{B}_R \\ \text{Anti-reflective BC:} \quad & \boldsymbol{A}_A = \boldsymbol{T} + \boldsymbol{B}_A \\ \text{Synthetic BC:} \quad & \boldsymbol{A}_S = \boldsymbol{T} + \boldsymbol{B}_S \end{aligned}$$

That is, the matrix structures are very similar, and thus we could consider using, for example, $\boldsymbol{A}_P$, or a symmetrized version of $\boldsymbol{A}_R$ as a preconditioner for $\boldsymbol{A}_S$. The important property we need is that it is possible to efficiently compute the spectral decomposition of the preconditioner. Note that $\boldsymbol{A}_P$ is the standard, and well studied, choice for preconditioning $\boldsymbol{A}_Z$; see, for example, [9, 44, 47].

For synthetic boundary conditions, if the PSF is symmetric, or close to being symmetric, then (the symmetrized) $\boldsymbol{A}_R$ is likely to be the most effective preconditioner. If the PSF is far from being symmetric, then $\boldsymbol{A}_P$ may be the best choice. Note that if we use $\boldsymbol{A}_R$ as the preconditioner for $\boldsymbol{A}_S$, then

$$\boldsymbol{A}_S - \boldsymbol{A}_R = \boldsymbol{B}_S - \boldsymbol{B}_R \quad \Rightarrow \quad \boldsymbol{A}_S\boldsymbol{A}_R^{-1} = \boldsymbol{I} + (\boldsymbol{B}_S - \boldsymbol{B}_R)\,\boldsymbol{A}_R^{-1}\,.$$

If the reflective BC is a good approximation of the synthetic BC, then we expect $\boldsymbol{B}_S - \boldsymbol{B}_R$ to have small rank and small norm. Thus $\boldsymbol{A}_R$ would be a good preconditioner for $\boldsymbol{A}_S$.

Since the image deblurring problem is extremely ill-conditioned, some care needs to be taken when incorporating preconditioning so that noise in the observed data is not magnified when we solve systems with the preconditioner,

$$\boldsymbol{A}_R\mathbf{x} = \boldsymbol{A}_R^T\mathbf{x} = \mathbf{y} \tag{2.10}$$

(the first equality is due to the symmetry of $\boldsymbol{A}_R$ for a symmetrized PSF). This equation can be solved efficiently using the discrete cosine transform (DCT) [48]. However, since $\boldsymbol{A}_R$ is usually ill-conditioned, we cannot use it directly as a preconditioner without including regularization [17, 35].

In this chapter we use Tikhonov regularization [25, 33, 36, 68]. Specifically, the spectral decomposition of $\boldsymbol{A}_R$ is

$$\boldsymbol{A}_R = \boldsymbol{C}^T\boldsymbol{\Lambda}\boldsymbol{C},$$

where $\boldsymbol{C}$ is (for $n \times n$ images) the $n^2 \times n^2$ orthogonal DCT matrix and

$$\boldsymbol{\Lambda} = \operatorname{diag}(\lambda_1, \lambda_2, \ldots, \lambda_{n^2}).$$

Under Tikhonov regularization with regularization parameter $\alpha$, $\boldsymbol{A}_R$ is approximated by $\widetilde{\boldsymbol{A}}_R$:

$$\widetilde{\boldsymbol{A}}_R = \boldsymbol{C}^T\widetilde{\boldsymbol{\Lambda}}\boldsymbol{C},$$

where

$$\widetilde{\boldsymbol{\Lambda}} = \operatorname{diag}\left(\tilde{\lambda}_1, \tilde{\lambda}_2, \ldots, \tilde{\lambda}_{n^2}\right) \quad \text{with} \quad \tilde{\lambda}_i = \frac{\lambda_i^2 + \alpha^2}{\lambda_i}.$$

The solution to (2.10) is then computed as

$$\widetilde{\boldsymbol{A}}_R^{-1}\mathbf{y} = \mathcal{C}^{-1}(\widetilde{\boldsymbol{\Lambda}}^{-1} * \mathcal{C}(\boldsymbol{y})), \tag{2.11}$$

where $\mathcal{C}$ and $\mathcal{C}^{-1}$ denotes the DCT and inverse DCT respectively. Using fast DCT algorithms, the cost of computing (2.11) is only $O(n^2 \log n)$. The regularization parameter $\alpha$ can be chosen using a variety of schemes, including discrepancy principle, L-curve, and generalized cross-validation (GCV) [29, 36]. In our work, we use GCV.

The GCV parameter choice method is based on the principle that if a data point is missing, then the remaining data points should predict the missing point well. The regularization parameter $\alpha$ is chosen to be the minimizer of the GCV function. In our case of Tikhonov regularization on $\boldsymbol{A}_R$, the GCV function takes the form of

$$G(\alpha) = \frac{\displaystyle\sum_{i=1}^{n^2} \left( \frac{\hat{y}_i}{\lambda_i^2 + \alpha^2} \right)^2}{\displaystyle\sum_{i=1}^{n^2} \left( \frac{1}{\lambda_i^2 + \alpha^2} \right)^2}, \quad \text{where} \quad \hat{\boldsymbol{y}} = \boldsymbol{C}\boldsymbol{y}. \tag{2.12}$$

The parameter $\alpha$ can be obtained by any optimization algorithm on the above function. Details of the implementation of the GCV parameter choice method for image deblurring can be found in [37].

## 2.4 Numerical Experiments

It is well known that the image deblurring problem requires regularization to stabilize the inversion process when there is noise in $\boldsymbol{y}$ and/or in $\boldsymbol{A}$. Note that even if the data $\boldsymbol{y}$ has no noise (which is highly unlikely in any real problem), because we use only an approximation of the true boundary elements (e.g, with $\boldsymbol{A}_Z$, $\boldsymbol{A}_P$, $\boldsymbol{A}_R$, $\boldsymbol{A}_A$, or $\boldsymbol{A}_S$), there is effectively noise in $\boldsymbol{A}$. For the numerical results reported in this section we use standard Tikhonov regularization [25, 33, 36, 68],

$$\min_{\boldsymbol{x}} \left\{ \|\boldsymbol{y} - \boldsymbol{A}_X \boldsymbol{x}\|_2^2 + \alpha \|\boldsymbol{x}\|_2^2 \right\} ,$$

where $\boldsymbol{A}_X$ is one of $\boldsymbol{A}_Z$, $\boldsymbol{A}_P$, $\boldsymbol{A}_R$, $\boldsymbol{A}_A$, or $\boldsymbol{A}_S$. Our implementation can be obtained from RestoreTools[1] patched with synthetic boundary conditions modification[2] , or Python RestoreTools (PYRET)[3]. The following experiments are done with the function HyBR (hybrid bidiagonalization regularization) [13, 14], which implements a modified version of LSQR [56], in RestoreTools. If the true image is known (as we do in our simulations) HyBR can easily compute Tikhonov solutions with optimal regularization parameters. RestoreTools also facilitates the implementation by providing functions to efficiently implement matrix-vector multiplications.

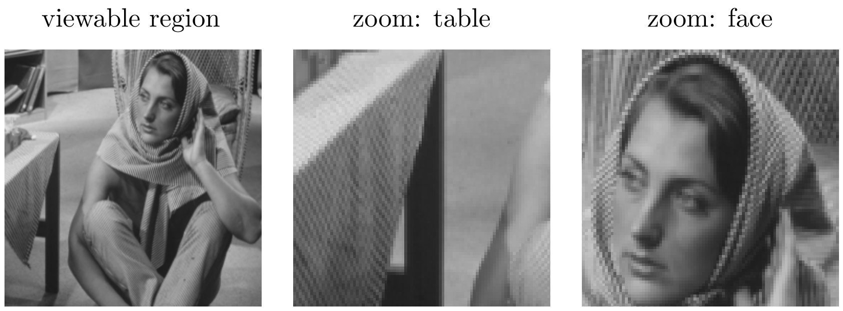


Figure 2.3: “Barbara” image

In our first set of experiments, we use the “Barbara” image (Figure 2.3) as the main test image. The following 4 cases are considered:

- Gaussian blur (Section 2.4.1)
- diagonal motion blur (Section 2.4.2)
- Gaussian blur with additive Gaussian noise (Section 2.4.3)
- diagonal motion blur with additive Gaussian noise (Section 2.4.4)

[1]`http://www.mathcs.emory.edu/~{}nagy/RestoreTools`
[2]`http://www.mathcs.emory.edu/~{}yfan/SyntheticBC/SyntheticBcPatch.tgz`
[3]`http://www.mathcs.emory.edu/~{}yfan/PYRET`

- DCT based preconditioning with $\mathbf{A}_R$ (Section 2.4.5)

Results on other images and additional experimental results are also shown in Section 2.4.6. Note that for display purposes only, pixel values in all of the following figures are clipped to the range [0,255]. We measure the quality of a deblurred image in terms of its peak-signal-to-noise ratio (PSNR), which is defined as

$$\begin{aligned} \mathrm{PSNR}(\boldsymbol{X}, \boldsymbol{X}_{\text{true}}) &= 20 \log_{10} \frac{255}{\mathrm{RMS}(\boldsymbol{X}, \boldsymbol{X}_{\text{true}})} && (2.13) \\ &= 10 \log_{10} \frac{MN255^2}{\sum_{i,j} [(\boldsymbol{X})_{i,j} - (\boldsymbol{X}_{\text{true}})_{i,j}]^2}. && (2.14) \end{aligned}$$

### 2.4.1 Gaussian Blur

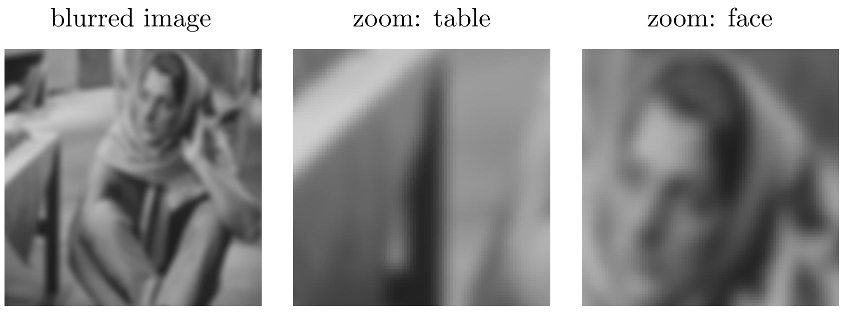
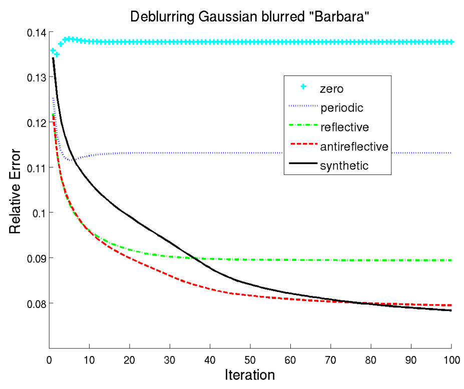


Figure 2.4: Gaussian blurred “Barbara” image

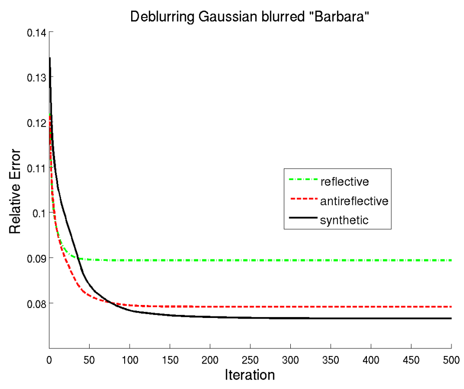
We start with the “Barbara” image, blur it with a Gaussian blur of size 11 with a standard deviation 3 and crop out the central viewable part (Figure 2.4). The formula for this Gaussian blur is given by

$$H_{i,j} = \frac{1}{Z} e^{-\frac{(i-6)^2+(j-6)^2}{18}}, \qquad i, j = 1, 2, \ldots, 11, \tag{2.15}$$

where $Z$ is a constant to ensure that the $H_{i,j}$'s sum up to one.



(a) All boundary conditions



(b) Reflective, anti-reflective and synthetic boundary conditions

Figure 2.5: Relative error vs iteration for deblurring Gaussian blurred "Barbara"

To deblur the image, we use the HyBR function with different boundary conditions. The true image is supplied to HyBR to choose the optimal regularization parameters. We run 100 iterations and select the iterates that yield minimum errors. Since the relative errors for reflective, anti-reflective, and synthetic boundary conditions are still decreasing at the 100th iteration, we continue the iterations for these boundary conditions until 500th iteration. The plot of relative errors against iteration is shown in Figure 2.5. Ideally (when there is no additive noise) with the true boundary conditions, the relative error decreases as the iteration progresses. As can be seen in Figure 2.5, synthetic and anti-reflective boundary conditions are most faithful to the true boundary conditions, with synthetic performing slightly better than anti-reflective. The corresponding peak signal-to-noise ratios (PSNR) of the computed reconstructed images are shown in Table 2.1.

The reconstructed images for the different boundary conditions are shown in Figure 2.6. From the figure, it is obvious that reconstructions with syn-

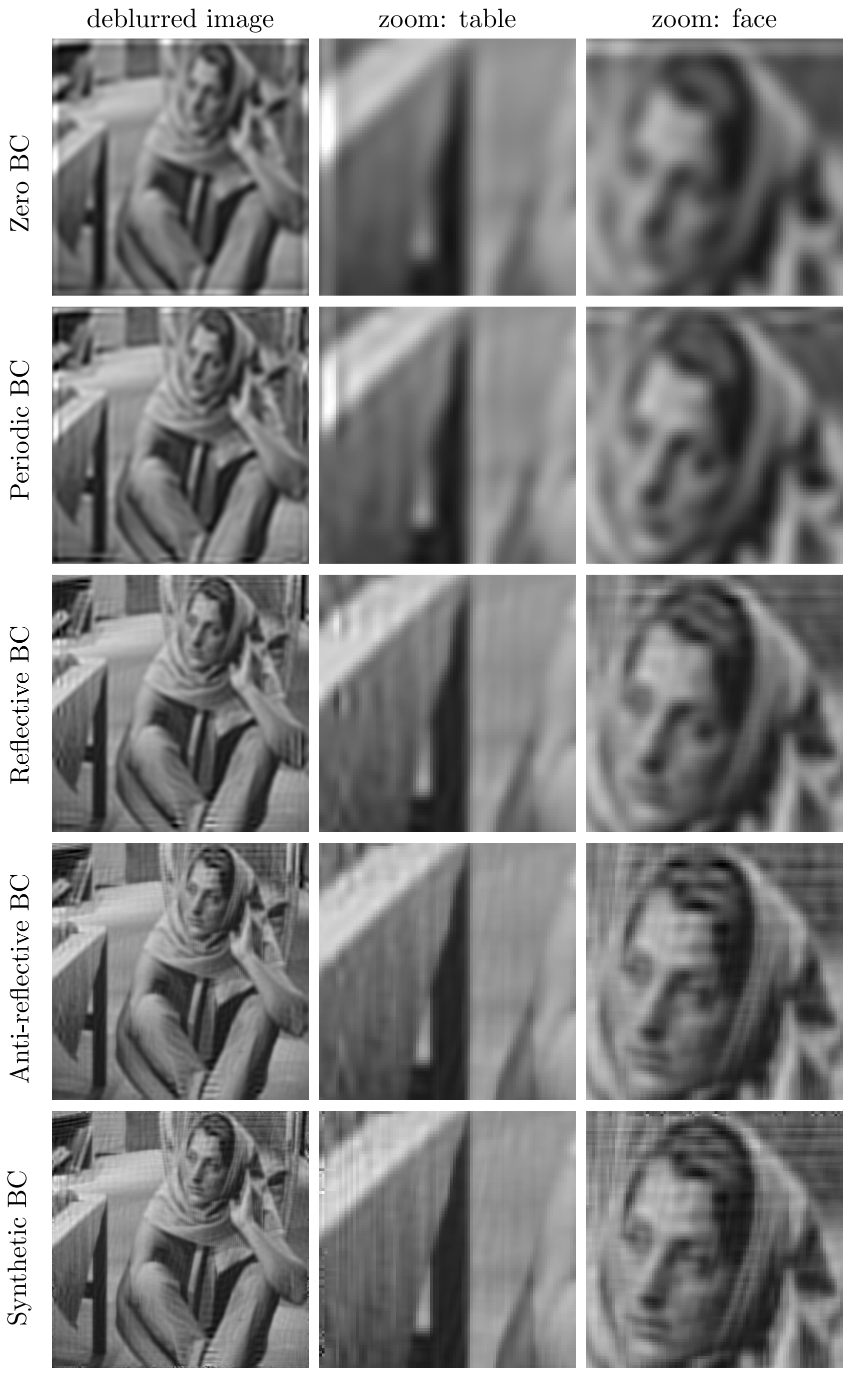


Figure 2.6: Deblurring results on Gaussian blurred "Barbara" with different boundary conditions

Table 2.1: PSNRs of deblurring results on Gaussian blurred "Barbara".

| | Blurred image | Zero | Periodic | Reflective | Anti-ref | Synthetic |
|---|---|---|---|---|---|---|
| PSNR | 24.5646 | 23.8368 | 25.4884 | 27.4083 | 28.4664 | 28.7532 |
| iteration | - | 2 | 5 | 167 | 255 | 500 |

thetic boundary conditions contain the least amount of ringing (oscillation) artifacts. The absence of the oscillation is easily seen at the table cloth on the left and the chair behind the woman. Synthetic boundary conditions also give better facial features.

### 2.4.2 Diagonal Motion Blur

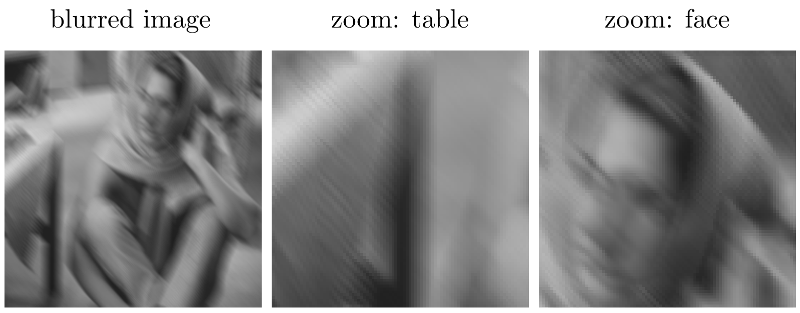


Figure 2.7: Motion blurred "Barbara" image

We repeat the experiment with a diagonal motion blur of size 11; the PSF is given by

$$H_{i,i} = \begin{cases} \frac{1}{11}, & i = j \\ 0, & i \neq j \end{cases}. \tag{2.16}$$

The blurred image is shown in Figure 2.7. A plot of the relative errors is shown in Figure 2.8, which clearly illustrates the effectiveness of synthetic boundary conditions compared to other boundary conditions.

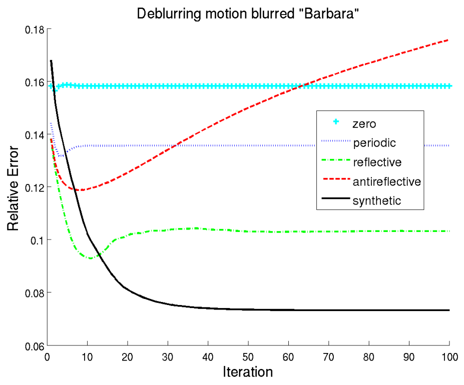


Figure 2.8: Plot of the deblurring errors vs iteration.

Table 2.2: PSNRs of deblurring results on motion blurred "Barbara".

| | Blurred image | Zero | Periodic | Reflective | Anti-ref | Synthetic |
|---|---|---|---|---|---|---|
| PSNR | 22.7484 | 22.5552 | 24.0441 | 27.0792 | 24.9434 | 29.1441 |
| iteration | - | 2 | 3 | 11 | 8 | 77 |

Figure 2.9 shows the computed reconstructions at the point where the iterations reached their smallest error, and the corresponding PSNRs are shown in Table 2.2. Note that with synthetic boundary conditions we are able to recover the texture of the table cloth and the chair very well, while other boundary conditions either return a blur or texture with severe ringing artifacts. Facial features are also well preserved under synthetic boundary conditions. In terms of PSNRs, synthetic boundary conditions give a significantly higher PSNR than other boundary conditions. Thus, for this particular blurring, our synthetic scheme is most faithful to the true boundary conditions.

### 2.4.3 Gaussian Blur with Additive Gaussian Noise

Next, we add 1% Gaussian noise to the Gaussian blurred image and deblur it with different boundary conditions. The noisy blurred image is shown

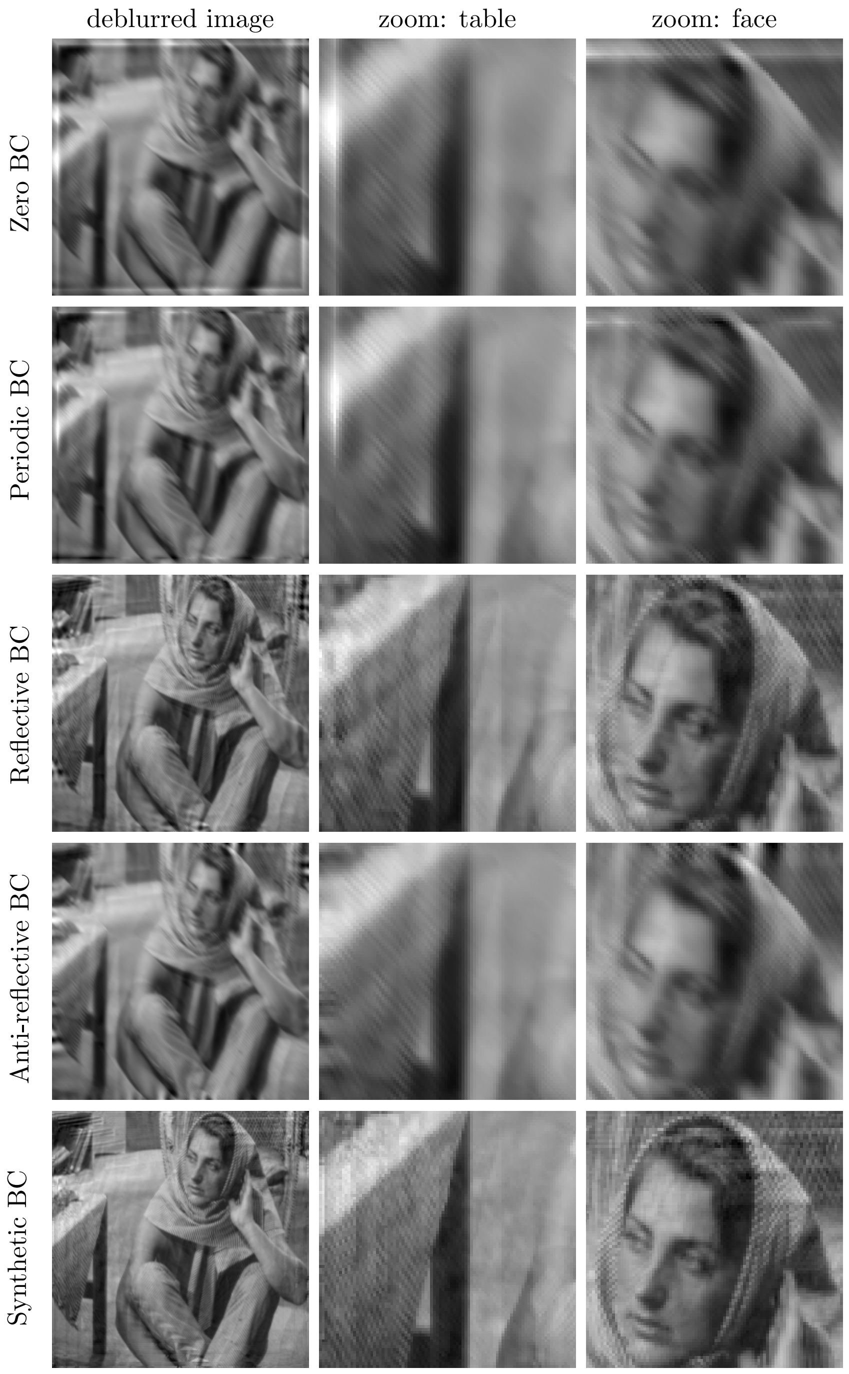


Figure 2.9: Deblurring results on motion blurred "Barbara" with different boundary conditions

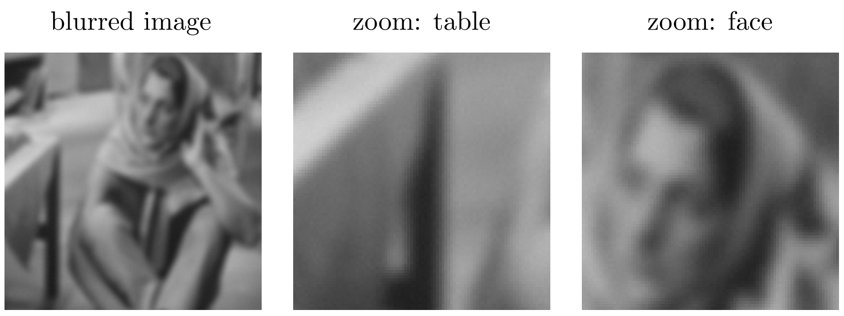


Figure 2.10: Noisy Gaussian blurred "Barbara" image

Table 2.3: PSNRs of deblurring results on noisy Gaussian blurred "Barbara"

| | Blurred image | Zero | Periodic | Reflective | Anti-ref | Synthetic |
|---|---|---|---|---|---|---|
| PSNR | 24.5383 | 23.8341 | 25.4795 | 26.9879 | 27.0608 | 26.3866 |
| iteration | - | 2 | 5 | 23 | 18 | 29 |

in Figure 2.10 and the deblurring results are shown in Figure 2.11. The corresponding PSNRs are shown in Table 2.3, and the relative error plot against iteration is shown in Figure 2.12(a).

In this case, anti-reflective boundary conditions give the best result, reflective boundary conditions the second best, and synthetic boundary conditions a close third. One may suggest that in the process of obtaining synthetic boundary conditions from the noisy image, noise is taken as image feature and incorrect boundary conditions are obtained. However, we believe this is not true; we applied the synthetic boundary conditions, obtained from the *noisy* blurred image, to deblur the corresponding *noise-free* blurred image, and obtained the very good results shown in Figure 2.13, with a PSNR of 28.5262dB. This illustrates that good synthetic boundary conditions can still be obtained from noisy images.

In fact, except for some pixels near the boundary, it is difficult to de-

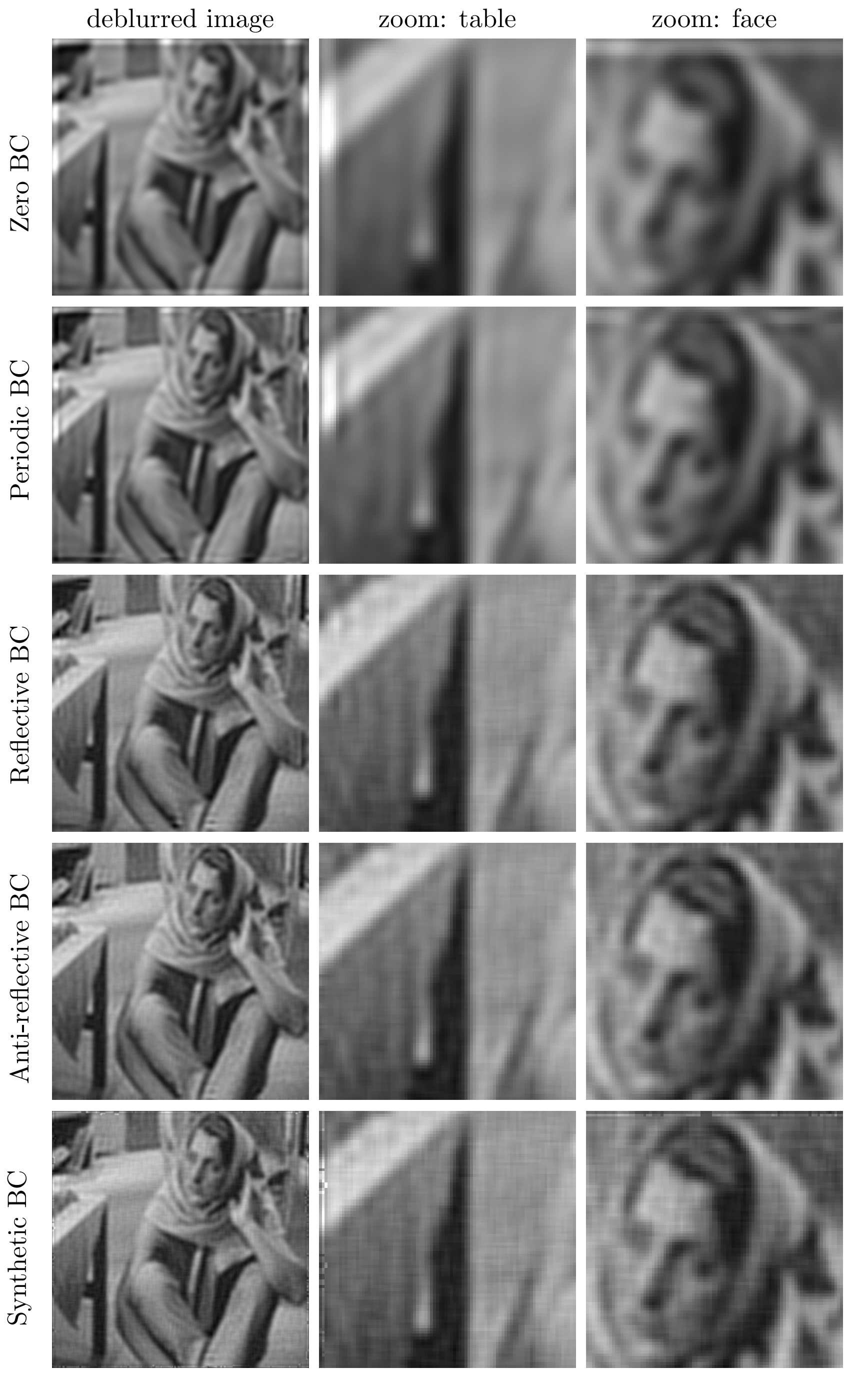
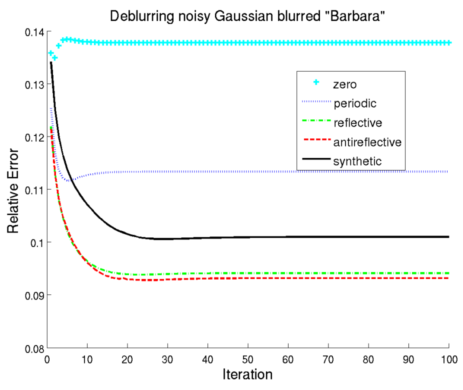
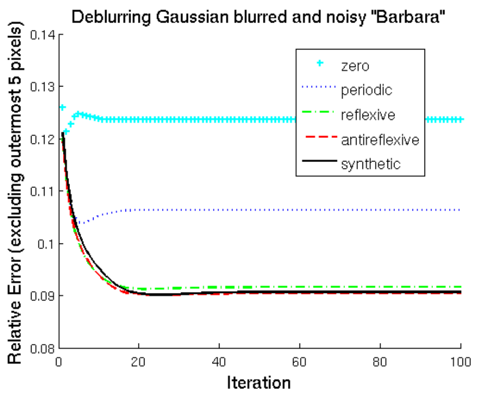


Figure 2.11: Deblurring results on noisy Gaussian blurred “Barbara” with different boundary conditions



(a) whole image



(b) outermost 5 pixels excluded

Figure 2.12: Plot of the deblurring errors vs iteration on noisy Gaussian blurred image.

deblurred image zoom: table zoom: face

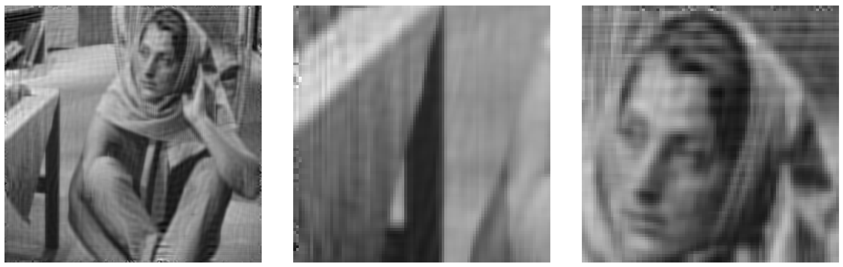


Figure 2.13: Deblurring result of Gaussian blurred “Barbara” with the synthetic boundary conditions obtained from the blurred and noisy counterpart. Its PSNR to the original image is 28.5dB.

termine visually if synthetic boundary conditions really perform worse than reflective and anti-reflective boundary conditions. Note that if we exclude the outermost 5 pixels in the calculation of relative errors and PSNRs, anti-reflective and synthetic boundary conditions give very similar results (cf. Figure 2.12(b) and Table 2.4), with slightly better results being obtained with synthetic boundary conditions.

Table 2.4: PSNRs (excluding outermost 5 pixels) of deblurring results on noisy Gaussian blurred “Barbara”.

| | Blurred image | Zero | Periodic | Reflective | Anti-ref | Synthetic |
|---|---|---|---|---|---|---|
| PSNR | 24.5383 | 24.7184 | 26.0702 | 27.1973 | 27.2696 | 27.3012 |
| iteration | - | 2 | 6 | 22 | 18 | 25 |

### 2.4.4 Diagonal Motion Blur with Additive Gaussian Noise

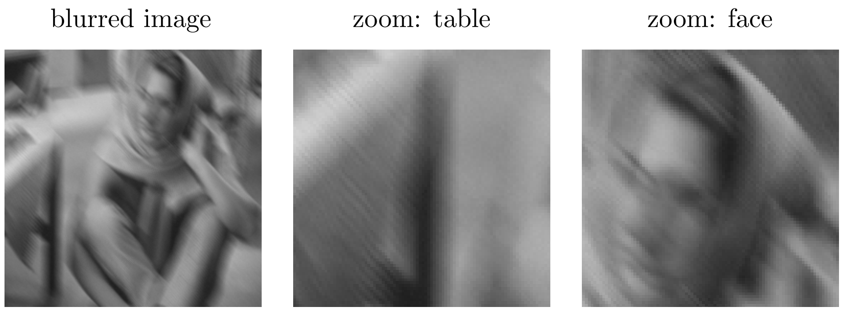


Figure 2.14: Noisy motion blurred “Barbara” image

Next, we add 1% Gaussian noise to the motion blurred image and deblur it with different boundary conditions. The noisy blurred image is shown in Figure 2.14 and the deblurring results are shown in Figure 2.16. The corresponding PSNRs and a plot of the errors at each iteration are shown in Table 2.5 and Figure 2.15 respectively.

Table 2.5: PSNRs of deblurring results on noisy motion blurred “Barbara”

| | Blurred image | Zero | Periodic | Reflective | Anti-ref | Synthetic |
|---|---|---|---|---|---|---|
| PSNR | 22.7309 | 22.5497 | 24.0219 | 26.4248 | 24.8341 | 26.5961 |
| iteration | - | 2 | 3 | 9 | 7 | 20 |

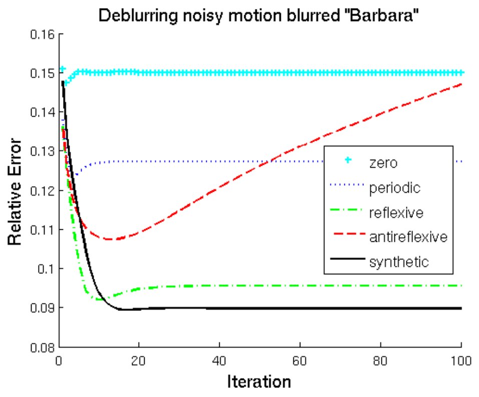


Figure 2.15: Plot of the deblurring errors vs iteration on noisy motion blurred “Barbara”.

We observe similar results as in the noise-free case. With synthetic boundary conditions, the texture of the table cloth and chair are restored quite successfully. The facial features are also restored very well. Overall, there are significantly fewer artifacts in the synthetic boundary conditions results compared to the others. In terms of PSNR, synthetic boundary conditions still give the highest PSNR, but its difference from the next best boundary conditions (reflective) is smaller than in the noise-free case.

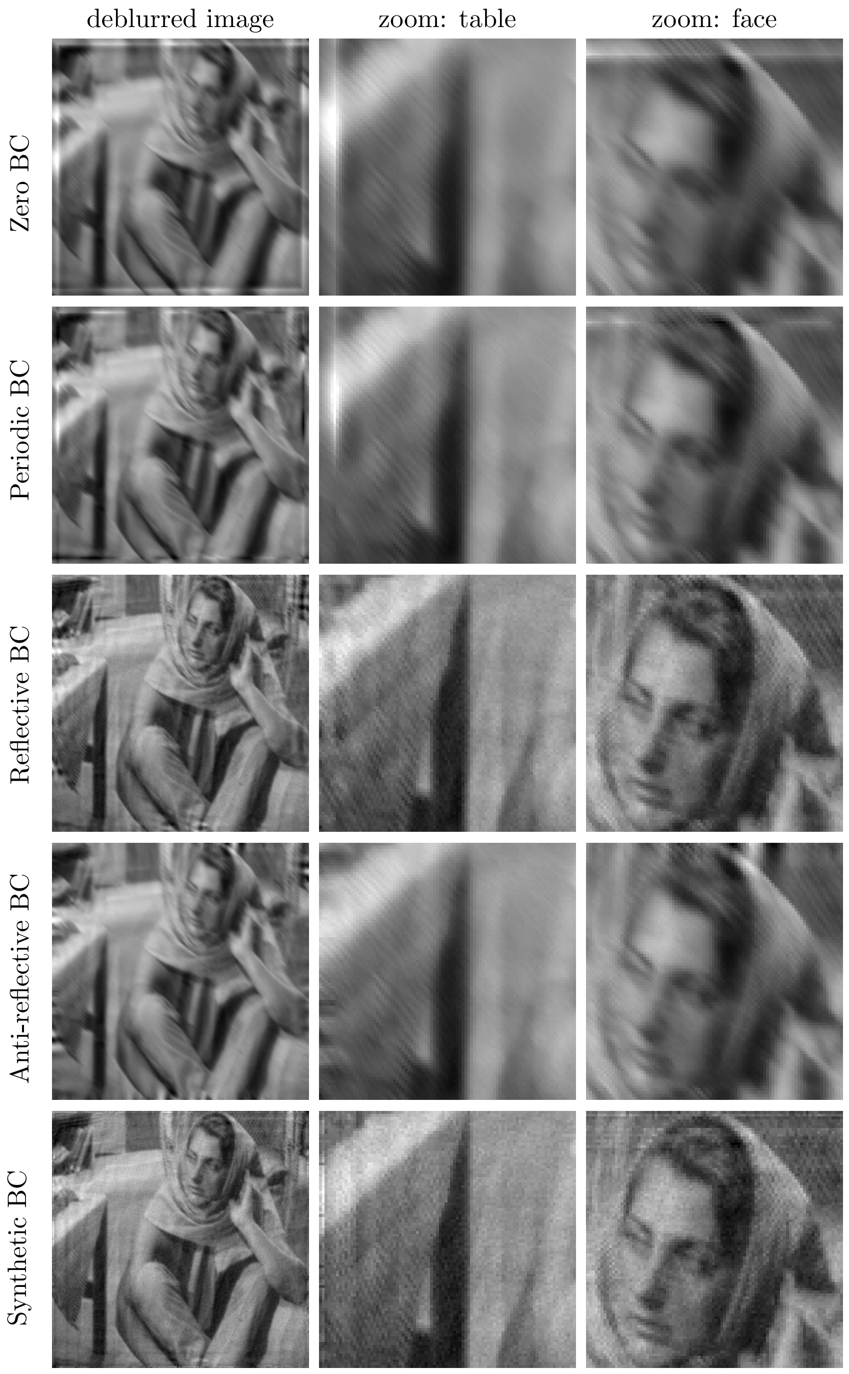


Figure 2.16: Deblurring results on noisy motion blurred "Barbara" with different boundary conditions

### 2.4.5 Preconditioning

Now we illustrate that preconditioning can significantly accelerate convergence of the iterative method. We only show results for the Gaussian blurred image with synthetic boundary conditions; similar results can be obtained with motion blur.

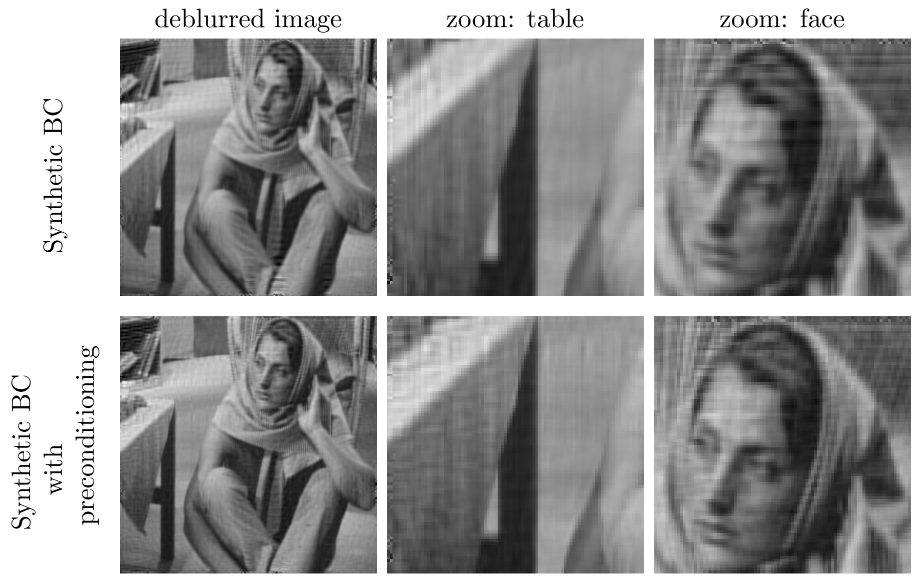


Figure 2.17: Deblurring results on Gaussian blurred "Barbara" with synthetic boundary conditions. The first row is obtained without preconditioning at the 500th iteration; the second row is obtained with preconditioning at the 20th iteration.

Table 2.6: PSNRs of deblurring results with and without preconditioning.

| | Blurred image | Synthetic | Synthetic with preconditioning |
|---|---|---|---|
| PSNR | 24.5681 | 28.7532 | 29.6790 |
| iteration | - | 500 | 20 |

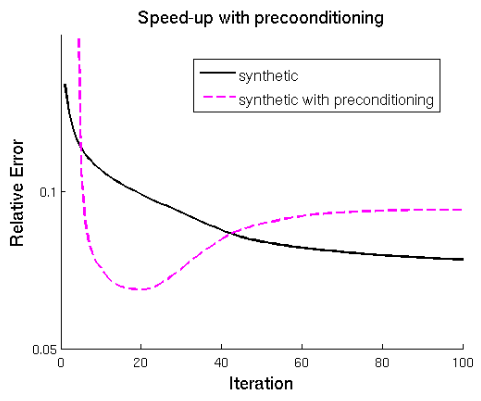


Figure 2.18: Plot of the deblurring errors with and without preconditioning.

The deblurring results with and without preconditioning are shown in Figure 2.17, the corresponding PSNRs are shown in Table 2.6, and the error plots are shown in Figure 2.18. Recall that without preconditioning, the minimum error is not yet attained even at 500th iteration. With preconditioning, the relative error drops very quickly, attaining its minimum at 20th iteration before increasing a little, and then levels off. In addition, we obtain a higher PSNR and recover more details, e.g. the texture of the chair.

### 2.4.6 Other Images and Additional Experiments

We repeated all of the experiments on several other standard test images (see, e.g., the MATLAB Image Processing Toolbox) with reflective, anti-reflective and synthetic boundary conditions respectively. The results are shown in Tables 2.7 and 2.8. Synthetic boundary conditions almost always give the highest PSNRs. Occasionally, synthetic boundary conditions give slightly lower PSNRs than anti-reflective boundary conditions, such as in the case of deblurring the Gaussian blurred “Goldhill” image. But in these

Table 2.7: PSNRs of the blurred images (Blurred), deblurred images with reflective (Ref), anti-reflective (Antiref) and synthetic (Syn) boundary conditions.

| | Gaussian blur | | | | Motion blur | | | |
|---|---|---|---|---|---|---|---|---|
| | Blurred | Ref | Antiref | Syn | Blurred | Ref | Antiref | Syn |
| Barbara | 24.5681 | 27.4083 | 28.4664 | 28.7532 | 22.7484 | 27.0792 | 24.9434 | 29.1441 |
| Baboon | 22.4401 | 24.3756 | 25.0833 | 25.3089 | 22.0289 | 26.0322 | 23.4704 | 27.8405 |
| Peppers | 23.5396 | 26.5654 | 27.7779 | 28.0495 | 21.4316 | 25.4560 | 23.4078 | 27.5207 |
| Goldhill | 24.7255 | 27.6440 | 30.3091 | 29.7711 | 23.2916 | 27.9265 | 25.7929 | 29.9787 |
| Camera-man | 21.2167 | 26.8596 | 27.8042 | 27.8703 | 20.2303 | 26.5291 | 23.8506 | 29.7191 |

Table 2.8: PSNRs of the noisy (1%) blurred images (Blurred), deblurred images with reflective (Ref), anti-reflective (Antiref) and synthetic (Syn) boundary conditions.

| | Gaussian blur + 1% Gaussian noise | | | | Motion blur + 1% Gaussian noise | | | |
|---|---|---|---|---|---|---|---|---|
| | Blurred | Ref | Antiref | Syn | Blurred | Ref | Antiref | Syn |
| Barbara | 24.5383 | 26.9879 | 27.0608 | 26.3866 | 22.7309 | 26.4248 | 24.8341 | 26.5961 |
| Baboon | 22.4176 | 23.5400 | 23.5663 | 23.1965 | 22.0081 | 25.0278 | 23.3752 | 25.2264 |
| Peppers | 23.5193 | 26.3752 | 26.7754 | 26.1722 | 21.4183 | 25.1881 | 23.3622 | 25.7139 |
| Goldhill | 24.6953 | 26.9863 | 27.2287 | 26.1062 | 23.2693 | 26.8872 | 25.5519 | 26.5859 |
| Camera-man | 21.2021 | 23.3848 | 23.3603 | 22.8304 | 20.2189 | 24.9255 | 23.3693 | 24.9763 |

cases synthetic boundary conditions still produce fewer ringing artifacts than anti-reflective boundary conditions; see, for example, Figure 2.19.

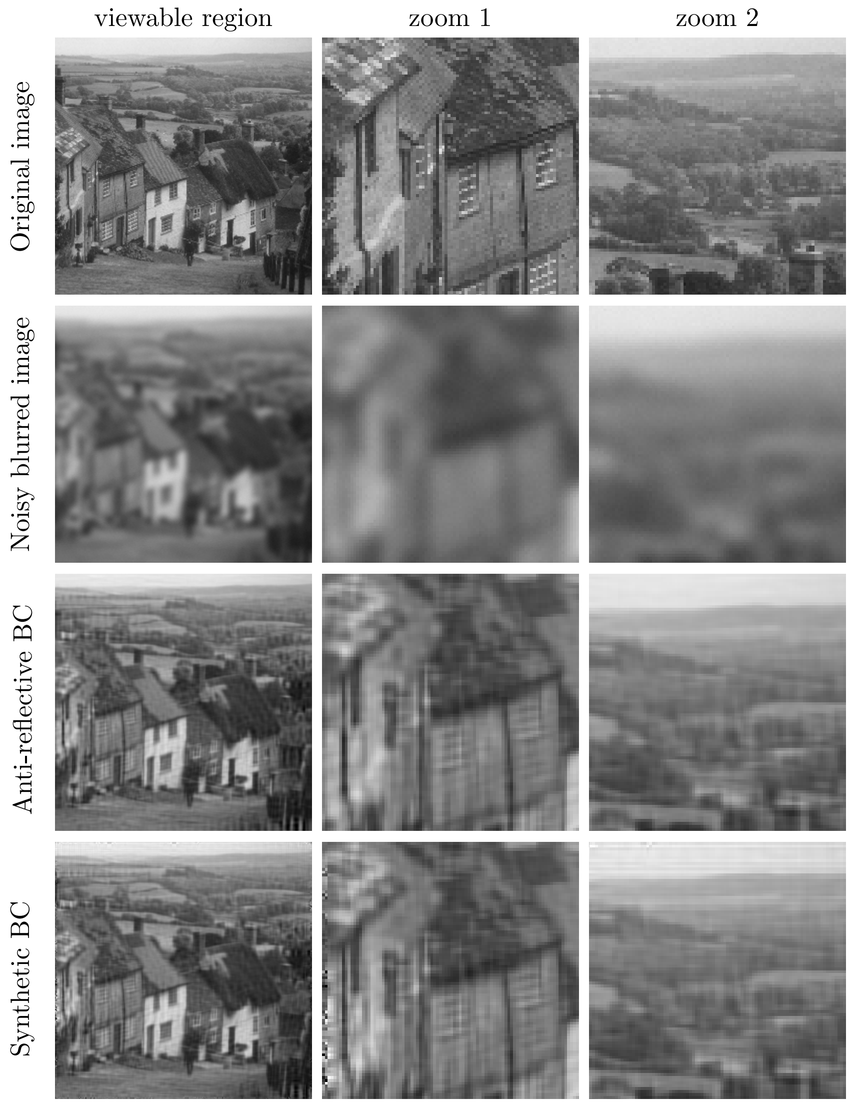


Figure 2.19: Deblurring results on noisy Gaussian blurred “Goldhill” with anti-reflective and synthetic boundary conditions.

In Table 2.9, we show the results when the noise level is only 0.1%. The results are similar to the noise-free case. Synthetic boundary conditions

Table 2.9: PSNRs of the slightly noisy (0.1%) blurred images (Blurred), deblurred images with reflective (Ref), anti-reflective (Antiref) and synthetic (Syn) boundary conditions.

| | Gaussian blur + 0.1% Gaussian noise | | | | Motion blur + 0.1% Gaussian noise | | | |
|---|---|---|---|---|---|---|---|---|
| | Blurred | Ref | Antiref | Syn | Blurred | Ref | Antiref | Syn |
| Barbara | 24.5644 | 27.3972 | 28.3176 | 28.3448 | 22.7321 | 26.4194 | 24.8416 | 26.5773 |
| Baboon | 22.0290 | 26.0191 | 23.4691 | 27.7776 | 22.0291 | 26.0151 | 23.4695 | 27.7875 |
| Peppers | 23.5395 | 26.5659 | 27.7148 | 27.8312 | 21.4312 | 25.4549 | 23.4084 | 27.4904 |
| Goldhill | 24.7251 | 27.6196 | 29.6363 | 28.9025 | 23.2913 | 27.9057 | 25.7940 | 29.8515 |
| Camera-man | 21.2165 | 25.7648 | 25.9675 | 25.6989 | 20.2301 | 26.5009 | 28.8533 | 29.5292 |

give the highest PSNR in all cases except on the noisy Gaussian blurred Goldhill image.

Table 2.10: PSNRs of the slightly blurred (PSF size: $5\times5$) images (Blurred), deblurred images with reflective (Ref), anti-reflective (Antiref) and synthetic (Syn) boundary conditions.

| | Gaussian blur | | | | Motion blur | | | |
|---|---|---|---|---|---|---|---|---|
| | Blurred | Ref | Antiref | Syn | Blurred | Ref | Antiref | Syn |
| Barbara | 28.3808 | 32.6027 | 33.5761 | 32.1339 | 26.5846 | 32.9626 | 30.4176 | 34.1026 |
| Baboon | 24.4705 | 28.2983 | 29.3215 | 28.4046 | 23.9554 | 29.0244 | 26.3301 | 30.3743 |
| Peppers | 28.6461 | 33.5144 | 34.8624 | 32.2711 | 26.3432 | 31.9760 | 30.1423 | 32.1449 |
| Goldhill | 28.5885 | 34.8708 | 36.7407 | 34.4301 | 27.0294 | 33.7078 | 30.8930 | 35.1045 |
| Camera-man | 24.3743 | 32.0713 | 32.5470 | 31.6834 | 23.0170 | 30.2952 | 27.4850 | 33.3327 |

We show in Table 2.10 the results when the PSF size is only $5 \times 5$. With a smaller PSF, the border region $\mathcal{B}$ in Figure 2.1 is narrower and thus the effect of synthetic boundary conditions in continuing edge direction and texture is less significant. This is clearly illustrated for the Gaussian blurred images, where we see that although the reflective and anti-reflective boundary conditions give slightly better results than synthetic boundary conditions, the difference in all cases is small. For motion blurred images, synthetic boundary conditions result in highest PSNRs even with a narrow border. This again demonstrates the strength of synthetic boundary conditions in deblurring motion blurred images.

## 2.5 Conclusions for this Chapter

We have introduced a new approach to choosing boundary conditions for imaging applications. We described the approach, which we call *synthetic boundary conditions*, in the context of image deblurring, and compared its linear algebraic structure, as well as its effectiveness to previously proposed boundary conditions. All four previously proposed boundary conditions (zero, periodic, reflective and anti-reflective) fail to continue important image structures like edge directions and texture outside the viewable region. On the other hand, our synthetic approach can continue these image structures. Extensive numerical experiments illustrated that synthetic boundary conditions typically allow for (sometimes significantly) better image reconstructions than other boundary conditions. In the (rare) situations when other boundary conditions performed better than our synthetic approach, the difference was minimal, and visually one could argue that the reconstructions with synthetic boundary conditions had fewer artifacts. The linear algebraic structure of the new boundary condition allows for efficient implementation of iterative image deblurring algorithms, and construction of effective preconditioners.

# Chapter 3

# Python and GPU Implementation of Deblurring Algorithms

To facilitate deblurring with different boundary conditions, we implemented the deblurring algorithms in the object-oriented programming language Python. We create a matrix class `psfMatrix` to imitate the convolution matrix. With operator overloading, convolution can be done through object multiplication. To use a different boundary condition, we only need to pass a different argument to the class constructor. We collect this Python implementation into a package PYRET[1], which stands for Python RestoreTools. Using the extensibility of Python, we have created a web interface of PYRET.

Several aspects of the deblurring process can be parallelized, including the initialization step for synthetic boundary conditions. This motivated us to develop a parallel implementation. One parallel architecture that is readily available on the market is graphical processing units (GPUs). We choose GPUs as our implementation platform and we call the final software package PARRET, which stands for Parallel RestoreTools. Experiments show an order of magnitude speedup in our GPU implementation.

[1] PYRET is pronounced as "pirate". It can be downloaded at `http://www.mathcs.emory.edu/~yfan/PYRET/doc/`. Its documentation is available at the same website.

## 3.1 Efficient Algorithms for Convolution Matrix Operations

Recall from (2.1), convolution is defined as

$$Y_{k,\ell} = \sum \sum H_{i,j} X_{k-i,\ell-j} \,. \tag{3.1}$$

and its vectorized version is given by

$$\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}. \tag{3.2}$$

When we convolve with periodic boundary conditions, the matrix $\boldsymbol{A}$ has block circulant with circulant blocks (BCCB) structure. Matrices with this structure can be diagonalized by the Fourier matrix $\boldsymbol{F}$,

$$\boldsymbol{A} = \boldsymbol{F}^H \boldsymbol{\Lambda} \boldsymbol{F}. \tag{3.3}$$

Thus, (2.2) can be rewritten as

$$\boldsymbol{y} = \boldsymbol{F}^H \boldsymbol{\Lambda} \boldsymbol{F} \boldsymbol{x}. \tag{3.4}$$

The eigenvalue matrix $\boldsymbol{\Lambda}$ can be obtained as

$$\boldsymbol{\Lambda} = \operatorname{Diag}\left(\operatorname{vec}(\operatorname{fft}(\boldsymbol{H}))\right), \tag{3.5}$$

where fft stands for fast Fourier transform, vec stands for vectorization and Diag forms a diagonal matrix with diagonal entries given by the vector argument. With the fast Fourier transform, convolution can be implemented as

$$\boldsymbol{Y} = \operatorname{ifft}\left(\operatorname{fft}(\boldsymbol{H}) \,.*\, \operatorname{fft}(\boldsymbol{X})\right). \tag{3.6}$$

We use ".*" to denote elementwise multiplication. This implementation decreases the operation cost from $n^4$ to $n^2 \log n$ for images with $n \times n$ pixels.

For boundary conditions other than periodic boundary conditions, we can use a pad and crop approach. We first pad the image to a larger image according to the boundary conditions, then convolve the larger image with periodic boundary conditions, and finally crop out the central part of the convolution:

$$\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x} \quad \Leftrightarrow \quad \boldsymbol{Y} = \text{crop}(\text{conv}(\text{pad}(\boldsymbol{X}))), \tag{3.7}$$

pad: pad with the boundary conditions
conv: convolve with $\boldsymbol{H}$ under periodic boundary conditions
crop: crop out the central part

In many deblurring algorithms, besides multiplication by the convolution matrix $\boldsymbol{A}$, we usually also need to multiply by $\boldsymbol{A}^T$. With periodic boundary conditions, multiplication by $\boldsymbol{A}^T$ is equivalent to a correlation operation, which in turn is equivalent to convolution with the PSF $\boldsymbol{H}$ rotated 180 degrees. With the spectral decomposition (3.3) of $\boldsymbol{A}$,

$$\boldsymbol{A}^T = \boldsymbol{A}^H = \left(\boldsymbol{F}^H \boldsymbol{\Lambda} \boldsymbol{F}\right)^H = \boldsymbol{F}^H \overline{\boldsymbol{\Lambda}} \boldsymbol{F}. \tag{3.8}$$

Hence, we can use almost the same implementation for the multiplication by $\boldsymbol{A}^T$, the only change is the conjugate of the eigenvalue matrix:

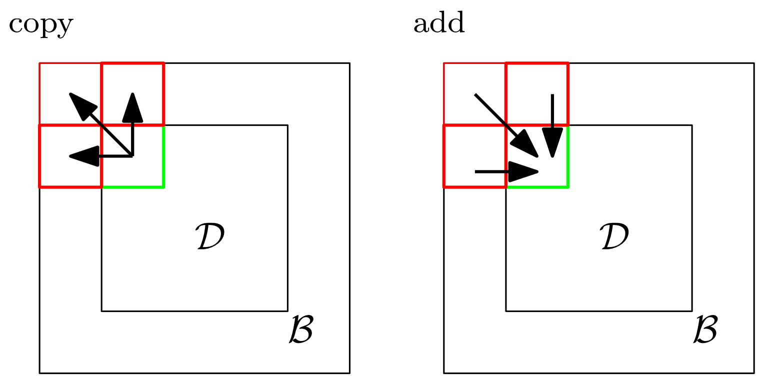
$$\boldsymbol{Y} = \text{ifft}\left(\overline{\text{fft}\,(\boldsymbol{H})}.*\text{fft}\,(X)\right). \tag{3.9}$$

It is a bit complicated for the cases of other boundary conditions. In (3.7), all three operations: pad, conv, crop are linear operations, so each of them can be represented by a matrix. Let us denote these matrices by $\boldsymbol{M}_{\text{pad}}$, $\boldsymbol{M}_{\text{conv}}$ and $\boldsymbol{M}_{\text{crop}}$ respectively, then we have

$$\boldsymbol{A} = \boldsymbol{M}_{\text{crop}} \boldsymbol{M}_{\text{conv}} \boldsymbol{M}_{\text{pad}} \quad \Rightarrow \quad \boldsymbol{A}^T = \boldsymbol{M}_{\text{pad}}^T \boldsymbol{M}_{\text{conv}}^T \boldsymbol{M}_{\text{crop}}^T \tag{3.10}$$

The process of multiplication by $\boldsymbol{A}^T$ should be equivalent to the transpose of these linear operations applied in reverse order. We have already discussed that the transpose of the convolution matrix $\boldsymbol{M}_{\text{conv}}$ with periodic boundary conditions is the correlation matrix with periodic boundary conditions. We denote this matrix by $\boldsymbol{M}_{\text{corr}}$. Next we describe the forms of $\boldsymbol{M}_{\text{pad}}^T$ and $\boldsymbol{M}_{\text{crop}}^T$

During the pad operation, some pixel values in the image domain $\mathcal{D}$ are copied to the border $\mathcal{B}$ (Figure 3.1a). Each pixel in $\mathcal{D}$ corresponds to a column in $\boldsymbol{M}_{\text{pad}}$. If a pixel does not get copied to $\mathcal{B}$, there is only a single nonzero entry with value 1 in the corresponding column; otherwise, there are multiple nonzero entries, each with value 1, for each position to which the pixel gets copied.

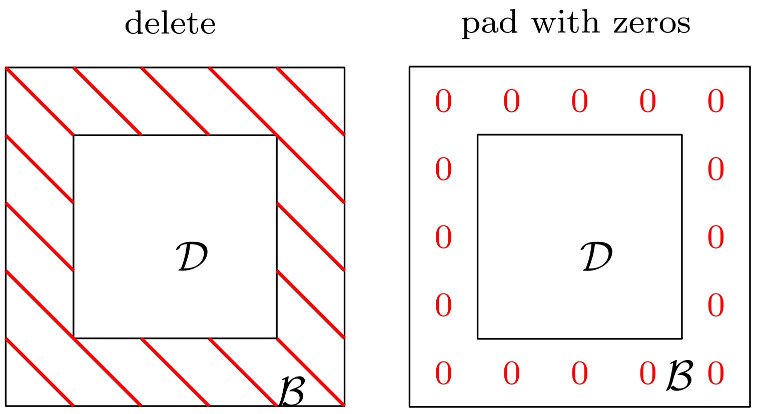


(a) pad operation (b) unpad operation

Figure 3.1: The pad and unpad operations. We only show the operation related to a single pixel in the image domain $\mathcal{D}$ as an example. (a): In the pad operation, some pixel values in the image domain $\mathcal{D}$ are copied to pixels on the border $\mathcal{B}$. (b): In the unpad operation, pixel values on the border $\mathcal{B}$ are added back to the source pixel.

Each column in $\boldsymbol{M}_{\text{pad}}$ becomes a row in $\boldsymbol{M}_{\text{pad}}^T$ during transpose. Thus each row in $\boldsymbol{M}_{\text{pad}}$ contains one or more entries with value 1 and the remaining entries are 0's. Hence in multiplication by $\boldsymbol{M}_{\text{pad}}^T$, source pixels during the pad operation are increased by values of those pixels to which it was copied; other pixels remain unchanged (Figure 3.1b). We call this the unpad operation and the corresponding matrix is denoted by $\boldsymbol{M}_{\text{unpad}}$.

For the crop operation (Figure 3.2a), $\boldsymbol{M}_{\text{crop}}$ is almost like the identity matrix, but with extra zero columns corresponding to those pixels on the border. Thus, $\boldsymbol{M}_{\text{crop}}^T$ is almost like the identity matrix with extra zero rows corresponding to those border pixels. Therefore multiplication by $\boldsymbol{M}_{\text{crop}}^T$ is equivalent to padding with zero (Figure 3.2b). We call this the uncrop operation and the corresponding matrix is denoted by $\boldsymbol{M}_{\text{uncrop}}$.



(a) crop operation (b) uncrop operation

Figure 3.2: The crop and uncrop operations. (a): In the crop operation, pixels on the border $\mathcal{B}$ are deleted. (b): In the uncrop operation, the image is padded with a zero border.

Putting the above together, we have

$$\boldsymbol{A}^T = \boldsymbol{M}_{\text{unpad}}\boldsymbol{M}_{\text{corr}}\boldsymbol{M}_{\text{uncrop}}. \tag{3.11}$$

That is multiplication by $\boldsymbol{A}^T$ is equivalent to an uncrop operation, then a correlation, followed by an unpad operation. The most expensive step is the correlation step. Just like convolution, correlation can be done using the fast Fourier transform and elementwise array multiplication. So the complexity of multiplication by $\boldsymbol{A}^T$ is also $O(n^2 \log n)$.

## 3.2 Deblurring Algorithms

In our Python software package PYRET, we have implemented several deblurring algorithms. We provide a few direct methods, when a spectral decomposition of the convolution matrix can be obtained efficiently. We use iterative methods when an efficient spectral decomposition is not available.

### 3.2.1 Direct Methods

Consider a decomposition of the convolution matrix $\boldsymbol{A}$

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^H, \tag{3.12}$$

where

$$\boldsymbol{U} = \begin{bmatrix} \boldsymbol{u}_1 & \boldsymbol{u}_2 & \cdots & \boldsymbol{u}_N \end{bmatrix} \quad \text{and} \quad \boldsymbol{V} = \begin{bmatrix} \boldsymbol{v}_1 & \boldsymbol{v}_2 & \cdots & \boldsymbol{v}_N \end{bmatrix} \tag{3.13}$$

are unitary matrices ($N$ is the number of pixels in the image) and

$$\boldsymbol{\Sigma} = \operatorname{Diag}(\sigma_1, \sigma_2, \ldots, \sigma_N)\,. \tag{3.14}$$

The classical spectral decomposition and singular value decomposition (SVD) are special cases of (3.12). Note that (3.12) is not necessarily a singular value decomposition, since we allow $\boldsymbol{\Sigma}$ to have negative diagonal entries. In general, we assume that $\boldsymbol{\Sigma}$ has nonzero entries on its diagonal. The solution to (3.2) is then given by

$$\boldsymbol{x} = \boldsymbol{V}\boldsymbol{\Sigma}^{-1}\boldsymbol{U}^H\boldsymbol{y} = \sum_{k=1}^{N} \frac{\boldsymbol{u}_k^H\boldsymbol{y}}{\sigma_k}\boldsymbol{v}_k. \tag{3.15}$$

When $\boldsymbol{U}$ and $\boldsymbol{V}$ correspond to fast transforms, the solution given by (3.15) can be computed efficiently. For example, with periodic boundary conditions, from (3.3), we have $\boldsymbol{U} = \boldsymbol{V} = \boldsymbol{F}^H$. Thus multiplication by $\boldsymbol{U}^H$ and $\boldsymbol{V}$ can be done with fast Fourier transforms, which has computational complexity

$O(n^2 \log n)$, when $\boldsymbol{x}$ and $\boldsymbol{y}$ are vectorized versions of images of size $n \times n$. When reflective boundary conditions are used, $\boldsymbol{U} = \boldsymbol{V} = \boldsymbol{C}^T$, where $\boldsymbol{C}$ is the discrete cosine transform matrix. Multiplication with $\boldsymbol{C}$ and $\boldsymbol{C}^T$ can also be done in $O(n^2 \log n)$. In PYRET, we implemented direct methods of the form (3.15) using Fourier and cosine transforms.

When $\boldsymbol{\Sigma}$ contains some diagonal entries that are small in magnitude, equation (3.15) may lead to magnification of noise in $\boldsymbol{y}$. In this situation, we need to use regularization. We have implemented the two most common regularization methods: truncated SVD (TSVD) and Tikhonov regularization. In TSVD regularization, we discard components corresponds to $\sigma_k$'s that are smaller than a threshold $\tau$,

$$\boldsymbol{x}_{\text{TSVD}} = \sum_{|\sigma_k| \geqslant \tau} \frac{\boldsymbol{u}_k^H \boldsymbol{y}}{\sigma_k} \boldsymbol{v}_k. \tag{3.16}$$

In Tikhonov regularization, we solve (3.2) by the minimization problem

$$\boldsymbol{x}_{\text{Tikhonov}} = \arg\min_{\boldsymbol{x}} \left( \frac{1}{2} \left\| \boldsymbol{y} - \boldsymbol{A}\boldsymbol{x} \right\|_2^2 + \frac{\alpha}{2} \left\| \boldsymbol{x} \right\|_2^2 \right), \tag{3.17}$$

which translates to

$$\boldsymbol{x}_{\text{Tikhonov}} = \left( A^T A + \alpha \boldsymbol{I} \right)^{-1} \boldsymbol{A}^T \boldsymbol{y} = \sum_{k=1}^{n^2} \frac{\overline{\sigma_k} \boldsymbol{u}_k^H \boldsymbol{y}}{\left| \sigma_k \right|^2 + \alpha} \boldsymbol{v}_k. \tag{3.18}$$

There are many methods for choosing the regularization parameters $\tau$ or $\alpha$, like the discrete Picard condition, L-curve, and generalized cross-validation (GCV). We refer the readers to [36] and references therein for more details. In PYRET, we have implemented the GCV method.

### 3.2.2 Iterative Methods

When an efficient decomposition of the convolution matrix is not available, for example when zero boundary conditions are used, we have to resort to iterative methods. We have implemented the following iterative methods in PYRET: CGLS, LSQR, MR2, MRNSD and their preconditioned versions.

Conjugate gradient least squares (CGLS) [6] treats (3.2) as a least squares problem, and is mathematically equivalent to applying the conjugate gradient method to the normal equations. LSQR is another popular method for solving least squares problems. Its use of Lanczos bidiagonalization makes it more stable for ill-conditioned matrices. For details of LSQR, see [56]. MR2 [34] is a modification of the the classical minimum residual method of Paige and Saunders [55] that is more appropriate for ill-posed inverse problems when the matrix is symmetric, and possibly indefinite. MRNSD [46] is a modification of the standard residual norm steepest descent method [61] that enforces a nonnegativity constraint on the solution. Some good references for iterative methods are [12, 20, 61].

**Algorithm 3.1** Conjugate Gradient Least Squares (CGLS)

```
$\boldsymbol{x}_0$ is an initial guess
$\boldsymbol{s}_0 = \boldsymbol{y} - \boldsymbol{A}\boldsymbol{x}_0$; $\boldsymbol{r}_0 = \boldsymbol{A}^T\boldsymbol{s}_0$; $\boldsymbol{p}_0 = \boldsymbol{r}_0$; $\rho_0 = \boldsymbol{r}_0^T\boldsymbol{r}_0$; $\boldsymbol{p}_{-1} = \boldsymbol{0}$; $\beta_{-1} = 0$
for $i = 0, 1, 2, \ldots$ do
    $\boldsymbol{p}_i = \boldsymbol{r}_i + \beta_{i-1}\boldsymbol{p}_{i-1}$
    $\boldsymbol{q}_i = \boldsymbol{A}p_i$
    $\alpha_i = \frac{\rho_i}{q_i^T q_i}$
    $\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \alpha_i\boldsymbol{p}_i$
    $\boldsymbol{s}_{i+1} = \boldsymbol{s}_i - \alpha_i\boldsymbol{q}_i$
    $\boldsymbol{r}_{i+1} = \boldsymbol{A}^T\boldsymbol{s}_{i+1}$
    if $\boldsymbol{x}_{i+1}$ is accurate enough then exit
    $\rho_{i+1} = \boldsymbol{r}_{i+1}^T\boldsymbol{r}_{i+1}$
    $\beta_i = \frac{\rho_{i+1}}{\rho_i}$
end for
```

To illustrate the basic computational costs of these iterative methods, consider the CGLS algorithm shown in Algorithm 3.1. One can see that this algorithm involves three vector additions, one inner product and two matrix-vector multiplications in each iteration. Matrix-vector multiplication is the most expensive operation in the algorithm. Fortunately, as discussed in Section 3.1, there exist efficient methods for these multiplications when the matrix $\boldsymbol{A}$ is a convolution matrix. The computational costs of other iterative methods implemented in PYRET are similar to CGLS.

## 3.3 PYRET: The Implementation in Python

In this section, we describe how we implement the above mentioned deblurring algorithms in Python.

### 3.3.1 Why Python?

We pick Python as our implementation language for many reasons.

- Python is a “higher-level language”. Programs written in Python are very close to pseudocode.
- Python is object-oriented, which facilitates the abstraction of convolution matrices.
- Python uses namespaces, and functions and classes are organized into modules and packages. This encourages good structure of the whole software package.
- Python functions and class methods support default arguments and keyword arguments, making specialization and generalization of functions and classes very robust without breaking other parts of the software.

- Python is dynamically typed. So implementation for one datatype can work with other datatypes as well.
- Python standard library `ctypes` and many other third party packages allow the direct call of functions in C libraries.
- Python comes with a shell and there are some third party shells, such as IPython [57] and Sage [64]. They provide an interactive environment akin to mathematical software like Matlab and Mathematica.
- Python documentation written in the format RestructuredText can be easily converted to HTML and PDF. There are tools to scan Python packages and build documentation websites.
- Mature scientific Python projects are available. These include NumPy and SciPy. There are also many success stories of using Python in the scientific community, such as PyACTS [22] and PyTrilinos [62].

These features of Python make it easier to use than traditional scientific programming languages like Fortran and C, and also easier to use external and legacy libraries than more modern languages like Matlab.

### 3.3.2 Implementation Details of PYRET

Numerical Python (NumPy) is a Python package that provides an array class for matrices and vectors. We build PYRET on top of NumPy. We implement the direct methods and iterative methods described in Section 3.2 in NumPy. These constitute the subpackages `directMethods` and `iterativeMethods` of PYRET. These implementations work with not just image deblurring, but with linear systems in general.

As discussed in Section 3.1, it is very inefficient to form and use the convolution matrix explicitly. Therefore, we create a psfMatrix class to imitate the NumPy array class and to provide the functionalities of a matrix required in deblurring algorithms. In Algorithm 3.1, and in other deblurring algorithms implemented in PYRET, the only operations that involve the convolution matrix are multiplication of this matrix, and its transpose, to a vector.

As mentioned in Section 3.1, the specific implementation for the multiplication depends on the enforced boundary conditions and whether the matrix is transposed. Thus, we design the psfMatrix constructor to take a PSF matrix and a string specifying the boundary conditions as arguments. A psfMatrix object also has a flag to indicate whether the matrix is transposed or not. We overload the multiplication operator (*) by implementing a special method `__mul__` in the psfMatrix class. Since an image can be represented as a 2-D array or vectorized to a 1-D vector, we make the multiplication operator polymorphic to work with either representation. We implement another special method, `transpose`, to toggle the transpose flag of a psfMatrix object. A usage example of the psfMatrix class is shown in Figure 3.3.

In PYRET, we also have a preconditioner module to provide three preconditioner matrix classes: fftPrec (for FFT preconditioning), dctPrec (for DCT preconditioning) and identityPrec (a placeholder when no preconditioning is used). Their implementation details are similar to that of the psfMatrix class. Objects from these preconditioner classes have an `isinverse` flag to indicate whether the preconditioner is approximating the matrix or its inverse, and an `inverse` method to modify the internal data to represent the inverse matrix.

There are several other modules and subpackages in PYRET to provide utility functions helpful in deblurring experiments. For more information on them and other parts of PYRET, please consult the online documentation at `http://www.mathcs.emory.edu/~yfan/PYRET/doc/`.

```
>>> # Suppose we have the following variables
>>> # x : original image
>>> # h : PSF
>>> # y : blurred image (to be computed from x and h)
>>>
>>> # Blur the image using object multiplication
>>> from pyret.psfMatrix import psfMatrix
>>> A = psfMatrix(h, boundary='periodic')
>>> y = A * x
>>>
>>> # Deblur the blurred image using CGLS
>>> from pyret.iterativeMethods import cgls
>>> deblur_result = cgls(A, y)
```

Figure 3.3: Usage example of the psfMatrix class.

### 3.3.3 Web GUI Interface

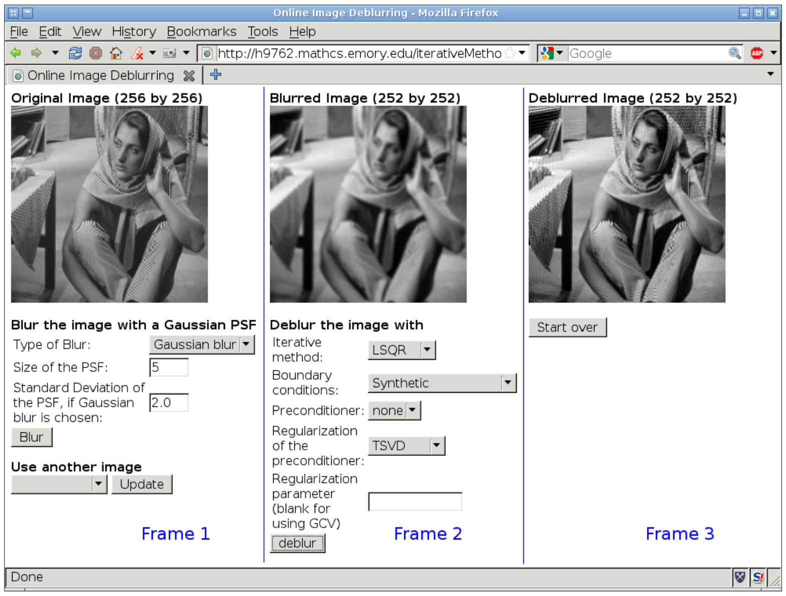
With the great extensibility of Python, we have developed a web interface for users of PYRET. A demo of that interface is available at `http://h9762.mathcs.emory.edu/iterativeMethods/`. Python is one of the top languages for writing web applications nowadays. Many large websites (e.g. YouTube.com[2] [3]) are powered by Python. There are a myriad of Python web frameworks, and the framework we choose is Pylons [28].

Pylons is very lightweight and it comes with everything needed for web application development: a web server, a template engine and a Web Server Gateway Interface (WSGI) controller. Pylons allows the use of the Model-

[2]`http://www.python.org/about/quotes/#youtube-com`

[3]`http://mail.python.org/pipermail/python-dev/2006-December/070323.html`

View-Controller (MVC) design pattern of web applications. It also follows extensively the WSGI standard, thus web applications developed in Pylons can be used with any web server with WSGI capability.


Online Image Deblurring - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://h9762.mathcs.emory.edu/iterativeMetho
Google
Online Image Deblurring
Blur
Update
deblur
Start over
Frame 1
Frame 2
Frame 3
Done

Original Image (256 by 256)

Blurred Image (252 by 252)

Deblurred Image (252 by 252)

Blur the image with a Gaussian PSF

Type of Blur: Gaussian blur

Size of the PSF: 5

Standard Deviation of the PSF, if Gaussian blur is chosen: 2.0

Use another image

Deblur the image with

Iterative method: LSQR

Boundary conditions: Synthetic

Preconditioner: none

Regularization of the preconditioner: TSVD

Regularization parameter (blank for using GCV):

Figure 3.4: Web GUI interface for PYRET.

To keep our web interface simple and compatible to most browsers, we only use basic HTML technology: HTML frames and forms. The processing of inputs are done through WSGI. We divide the webpage into three frames (Figure 3.4). In the first frame, an unblurred image is shown and the user can pick the type and parameters for the blur to apply to the image. The user can also switch to another image from a list of standard test images. After clicking the blur button, the blurred image is shown in the second frame. The user can then choose in the second frame the iterative method, boundary conditions, preconditioner and regularization for deblurring. When the deblur button is clicked, the final deblurring result is shown in the final frame.

## 3.4 PARRET: Parallel Implementation on GPUs

### 3.4.1 Parallelizability of Vector and Matrix Operations in Deblurring Algorithms

To achieve faster deblurring, we consider parallelizing certain computations implemented in PYRET. We note that in iterative methods, such as CGLS (Algorithm 3.1), the operations involved are vector additions/subtractions, inner products and matrix-vector multiplications. All of these operations are parallelizable. When the matrix $\boldsymbol{A}$, in for example Algorithm 3.1, is a convolution matrix, we know from Section 3.1 that the matrix-vector multiplication can be done efficiently with the fast Fourier transform, which is parallelizable. In addition, if synthetic boundary conditions are used, the initial step to find the boundary conditions can be done in parallel.

### 3.4.2 Why GPUs?

There are many parallel architectures, from big clusters to many-core CPUs. We consider approaches that can exploit the architecture of graphical processing units (GPUs). GPUs are massively parallel processors, whose original purpose is for processing 2D or 3D graphics, but they are also good for stream processing. They allow complex floating point operations of large amounts of data. GPUs are readily available from the consumer market and are already installed in many commodity desktop and laptop computers. General-purpose computing on graphics processing units (GPGPU) is an affordable way to attain vast parallel computing power. GPUs have also entered the arena of high performance computing. In June 2010 a supercomputer built with GPUs won second place on the Top500 list[4] and fourth place on the Green500 list[5].

GPU vendors have released application programming interfaces (API) for general purpose programming. Among them, the most mature API is NVIDIA's C for CUDA (Compute Unified Device Architecture). Recently, many companies have joined forces to create a standard called OpenCL (Open Computing Language) for programming heterogeneous platforms (e.g. CPUs, GPUs), but it is still yet to be adopted by many users.

There are many success stories with NVIDIA CUDA architectures in the scientific community. One examples is the SETI@home projects[6]. Many other examples can be found on the CUDA Showcase website[7]. Because of these success stories, we choose NVIDIA GPUs as our parallel programming platform. The vast availability of NVIDIA GPUs on the market provides a large potential audience for our parallel software package.

### 3.4.3 Compute Unified Device Architecture (CUDA)

Compute Unified Device Architecture (CUDA) is a parallel architecture available on NVIDIA GeForce 8 series and later models. NVIDIA has released a C language extension for programming on CUDA (C for CUDA), sometimes abbreviated to just CUDA by some developers.

There are two levels of API (Figure 3.5). The lower level driver API comes with NVIDIA GPU drivers, while the higher level runtime API requires installation of the CUDA Toolkit. This toolkit also includes two CUDA libraries: CUBLAS (CUDA Basic Linear Algebra Subprograms) and CUFFT (CUDA Fast Fourier Transform). The runtime API is easier to program, as

[4] `http://top500.org/lists/2010/06`

[5] `http://www.green500.org/lists/2010/06/top/list.php`

[6] `http://setiathome.berkeley.edu/cuda.php`

[7] `http://www.nvidia.com/object/cuda_showcase_stage.html`

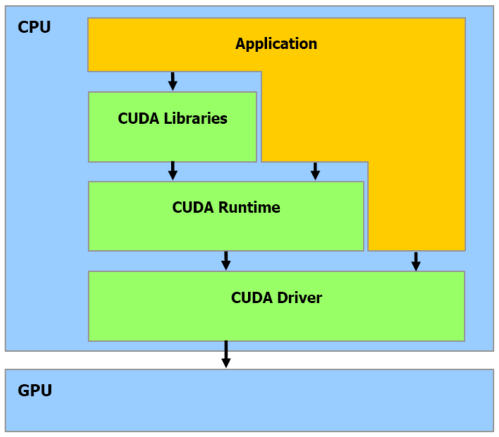


Figure 3.5: CUDA API layout. Image source: NVIDIA CUDA Programming Guide Version 1.1

the user does not need to perform the initialization, packing of kernel arguments and a few other low level operations explicitly. For CUDA version 2.x series, developers are recommended to use just one level of API. Starting from CUDA 3.0, the interoperability of the two levels of API has greatly improved.

For both APIs, a typical workflow of a CUDA application includes the following four steps (Figure 3.6).

1. Data are copied from the main memory to the memory on the GPU.

2. Codes written in C for CUDA are compiled into kernels.

3. Each of the kernels is then executed on the GPU in parallel.

4. After the kernels are run, the results are then copied back to the main memory.

The CUDA programming model is stream processing, which is closely related to the SIMD (single instruction, multiple data) parallel architecture model. CUDA kernels run on GPUs in threads. The threads are grouped

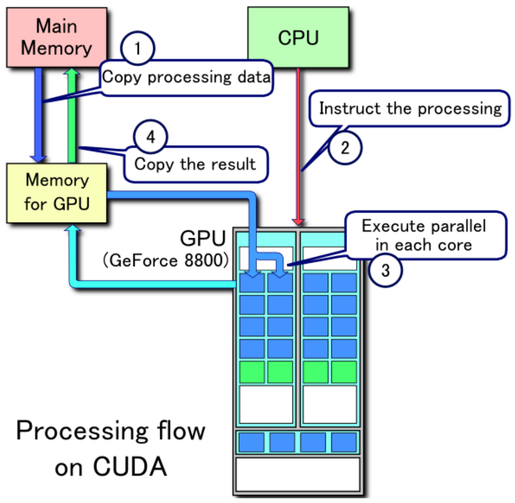


Figure 3.6: Workflow of a CUDA application.
Diagram by Tosaka on Wikimedia Commons and licensed under a Creative Commons Attribution 3.0 Unported license. See `http://en.wikipedia.org/wiki/File:CUDA_processing_flow_(En).PNG`

into thread blocks in a 1D, 2D or 3D organization. The thread blocks are in turn organized into a 1D or 2D grid. Each thread has its own registers and local memory mainly for storage of kernel arguments. Threads within the same thread block can communicate with each other through the shared memory and synchronization. The thread blocks are supposed to be run independently of each other, as their running order cannot be defined by the programmer. The organization of threads and memory on an NVIDIA GPU are illustrated in Figure 3.7.

For more details on programming in C for CUDA and how to attain good performance, we refer the readers to [49–51].

### 3.4.4 Python Wrapper of CUDA (PyCUDA)

PyCUDA [40, 41] is a thin wrapper for the CUDA driver API. It exposes all functionalities of the CUDA driver API in Python. PyCUDA uses a tech-

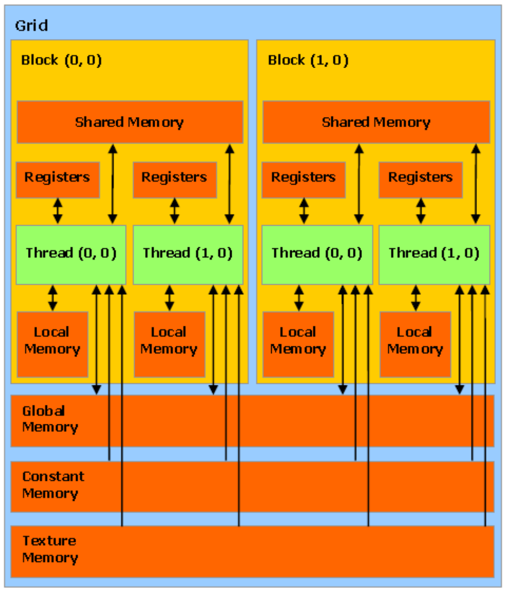


Figure 3.7: Threads on a GPU are organized into thread blocks, which are in turn organized into a grid. Image source: NVIDIA CUDA Programming Guide Version 1.1

nique called runtime code generation (RTCG). It generates CUDA codes at runtime, the codes are then compiled into CUDA binary format and uploaded to the GPUs. Because of the runtime generation, the codes can be catered for different situations, like different data types and sizes and different GPU specifications. In practice, the best configuration (loop slicing, thread block size, grid size, etc) is not obvious and requires extensive experiments to optimize. With the convenience of Python, codes written in PyCUDA can be easily modified to test different configurations.

Another advantage of wrapping in Python is that it allows interactive use of CUDA in the Python shell (either the standard shell, IPython shell, or Sage). The low level configuration and compilation of CUDA codes, and memory allocation and deallocation on the GPUs are hidden from the users.

PyCUDA also implements a gpuarray class to imitate a NumPy array, very much like the psfMatrix class in PYRET. A gpuarray object works like NumPy arrays, but the array data are stored on the GPUs. With operator overloading, matrix and vector operations involving a gpuarray object are done on the GPU in CUDA.

### 3.4.5 Interfacing CUBLAS and CUFFT with PyCUDA

Although PyCUDA has provided a high level interface to C for CUDA, it still lacks interfaces to two widely used CUDA libraries: CUBLAS and CUFFT. CUBLAS provides basic linear operations like vector additions, while CUFFT provides fast Fourier transforms. We make use of the `ctypes` package from the Python Standard Library to wrap CUBLAS and CUFFT in Python. For usage convenience, we provide another level of wrapping to the functions exported by `ctypes`. For example, some CUBLAS functions take lengths of the vectors as arguments, but these are actually redundant as the lengths can be derived from the vector arguments. We remove these redundancies with our wrappings. Many functions from CUFFT take an FFT plan as an argument. After our wrapping, the plan is automatically created when not given. Besides these, we also add in a few error checking functions.

The above steps only expose CUBLAS and CUFFT in Python. Some extra work needs to be done for the wrapped functions to work seamlessly with PyCUDA gpuarray objects. First the wrapped CUBLAS and CUFFT functions expect a pointer for the vector arguments. We provide a `pointer` function to extract from a gpuarray object the pointer to its data on the GPU. The `pointer` function is automatically called when a pointer argument is expected by functions wrapped in `ctypes`.

Another subtle but very important issue is the adjustment for the row and column major orders. The PyCUDA gpuarray class is modeled after the

NumPy [52] array class, which is written in C. Thus, the gpuarray class uses row-major order. Similarly, CUFFT models after the FFTW library [27], which is written in C, so CUFFT follows row-major order too. However, CUBLAS models after the BLAS library, which is written in Fortran, causing CUBLAS to use column-major order.

Two-dimensional complex-to-complex Fourier transform of a matrix is equivalent to two one-dimensional complex-to-complex Fourier transforms along its row and column directions respectively. Thus row-major or column-major order does not matter for these transforms. Nevertheless, in two-dimensional real-to-complex and complex-to-real Fourier transforms, only the non-redundant entries in the last dimension are involved, i.e. the row and column dimensions are treated differently. Therefore row-major or column-major order matters for these kinds of transforms. Fortunately, PyCUDA and CUFFT both assume row-major order, thus gpuarray objects can be used with wrapped CUFFT functions directly.

This is not the case for CUBLAS functions. Switching from row-major order to column-major order has no effect on scalar and vector arguments, but for matrix arguments it is equivalent to a transpose. The influence of switching to column-major order on different levels of BLAS operations is illustrated in Table 3.1. There is no change for level 1 BLAS operations, as they only involve scalars and vectors. For level 2 BLAS operations, the matrix argument needs to be transposed back. For level 3 BLAS operations that involve the product of two matrices, it is necessary to transpose and to switch the order of the matrices. Our wrapping on CUBLAS performs these transposes and order switching for the user.

| BLAS Level | row-major order | column-major order equivalence |
|---|---|---|
| 1 | $\boldsymbol{y} = \alpha\boldsymbol{x} + \boldsymbol{y}$ | $\boldsymbol{y} = \alpha\boldsymbol{x} + \boldsymbol{y}$ |
| 2 | $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x} + \beta\boldsymbol{y}$ | $\boldsymbol{y} = \boldsymbol{A}^T\boldsymbol{x} + \beta\boldsymbol{y}$ |
| 3 | $\boldsymbol{C} = \alpha\boldsymbol{A}\boldsymbol{B} + \beta\boldsymbol{C}$ | $\boldsymbol{C}^T = \alpha\boldsymbol{B}^T\boldsymbol{A}^T + \beta\boldsymbol{C}^T$ |

Table 3.1: Examples of equivalent operations for row-major and column-major order systems.

### 3.4.6 Complex Branch of PyCUDA

When we perform the fast Fourier transform based convolution matrix multiplication (3.9), "$\text{fft}\,(\boldsymbol{H})$" and "$\text{fft}\,(\boldsymbol{X})$" are, in general, complex, and it is therefore necessary to perform computations with complex numbers. At the time of our implementation, CUBLAS was at version 2.3 and PyCUDA was at version 0.93. Both do not support arithmetic with complex numbers. It was therefore necessary to provide this implementation.

Our first attempt used function closure, which does not directly modify PyCUDA. After importing the PyCUDA package into the Python workspace, each imported function that takes complex arguments is substituted by our helper function. A drawback of this approach is that the codes are cluttered with helper functions, and sometimes it is not clear which version of the function is called.

After corresponding about this difficulty with Andreas Klöckner, the author of PyCUDA, we decided to create a new complex branch of PyCUDA to better integrate complex arithmetic. We note that in the C99 standard of the C language, a new complex type has been introduced, but the library implementing this is supposed to run only on CPUs, and thus cannot be linked to CUDA binary codes. There are complex number classes in the C++ Standard Template Library (STL). The GNU Compiler Collection (GCC) implements STL in a library `libstdc++.so`, which again only runs on CPUs. Thus we decided to discard the idea of linking external libraries, and implemented complex arithmetic directly in source code.

We pick C++ instead of C as our implementation language, since C++ templates allow the same codes to work for both float complex type and double complex type. Our prototype was based on STLPort, whose source codes are much simpler than that of GCC. We use the "`extern "C++"`" trick to make the C++ implementation work with the NVIDIA `nvcc` compiler.

We store the real and imaginary parts of complex arrays on GPUs in an interlacing manner, matching the storage scheme of NumPy complex arrays. Using function and operator overloading, the new float and double complex type acts like other C/C++ native data types. This makes the necessary modification in PyCUDA minimal. Basically, the modification is on those functions that map Python/NumPy datatypes to and from C/C++ datatypes.

The approach of injecting complex arithmetic at the C/C++ level rather than at the Python level as in our fist approach, makes maintenance and subsequent development of PyCUDA more robust. We may use the same approach to add to PyCUDA other datatypes that CUDA may support in the future. In the latest release candidate of PyCUDA version 0.94, the complex branch has been merged to the main branch.

### 3.4.7 Features of PARRET

Our implementation of deblurring algorithms on CUDA is called PARRET, which stands for Parallel RestoreTools and is pronounced as "parrot". It includes our wrappers of CUBLAS and CUFFT for use with gpuarray objects. Just like in PYRET, we have a psfMatrix class to represent the convolution matrix. The difference here is that the matrix data are stored in the GPU memory, and objects of this class act on gpuarray objects rather than NumPy arrays. Using the psfMatrix class, we create the GPU counterparts of the `convolve2d` and `correlate2d` functions. We also include code for obtaining the synthetic boundary conditions (see Section 2.2.3) in parallel using GPUs.

Several linear solvers are provided in PARRET. We follow the design of the sparse linear algebra subpackage of SciPy (`scipy.sparse.linalg`) to use a LinearOperator class for the matrices in linear equations. We do this for easy future porting of sparse linear solvers from SciPy into PARRET. In PARRET we implement the CG (conjugate gradient), CGLS (conjugate gradient least squares), and CGNR (conjugate gradient normal residual) solvers with a CPU version and a GPU version. From them, we create specialized versions, by wrapping, for the case when the linear operator is given by a matrix or a PSF. This design gives us the flexibility to include other types of linear operators in the future. For more information on PARRET, please consult its online documentation at
`http://www.mathcs.emory.edu/~yfan/PARRET/doc/index.html`.

### 3.4.8 Speedup of PARRET

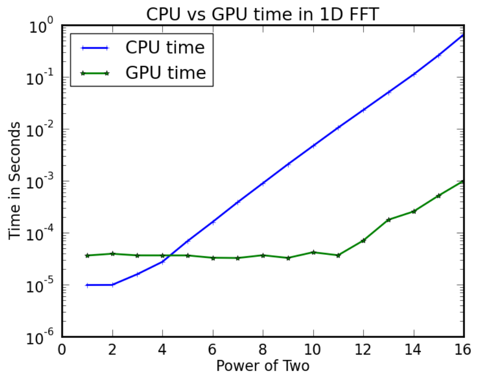
Next, we show some benchmarking of the CPU and GPU implementations. The testbed is a HP Pavilion dv3022tx Entertainment Notebook PC equipped with Intel Core 2 Duo T5550 CPU and NVIDIA GeForce 8400M GS GPU. The specifications of the CPU and GPU are shown in Table 3.2. As shown in the table, the GPU has more cores and memory than the CPU, and thus the GPU should be better at handling parallel computation. Our benchmarking results agree with this expectation.

In Figure 3.8, we show the CPU and GPU times for four different types of fast Fourier transforms on data of different sizes. The four types of transforms are 1D FFT, 2D FFT, 2D real-to-complex FFT[8], and 2D complex-to-real

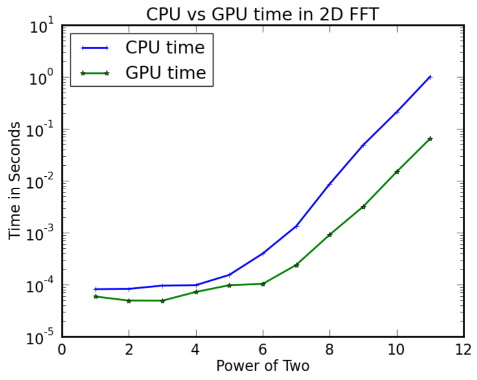
[8] Fourier transform of real data is conjugate symmetric. A real-to-complex FFT only

| | CPU | GPU |
|---|---|---|
| Model | Intel Core 2 Duo CPU T5550 | NVIDIA GeForce 8400M GS |
| Clock Rate | 1.83GHz | 0.8GHz |
| Memory | 2MB (L2 cache) | 256MB |
| No. of Cores | 2 | 16 |

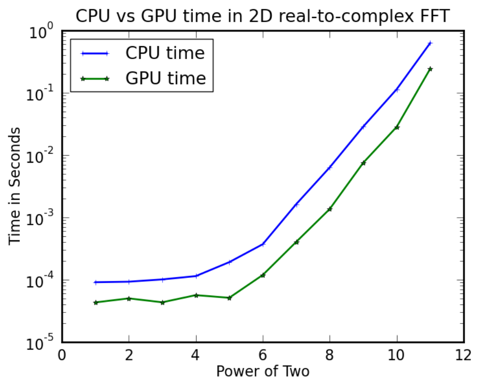
Table 3.2: Specifications of the CPU and GPU used in the benchmarking. Data sources: `http://ark.intel.com/Product.aspx?id=37255` and `http://www.nvidia.com/object/geforce_8400M.html`.



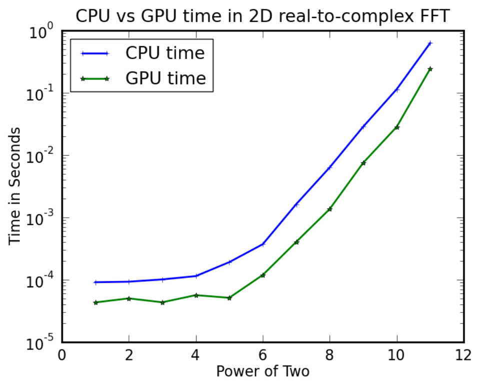
(a) 1D FFT



(b) 2D FFT



(c) 2D real-to-complex FFT



(d) 2D complex-to-real FFT

Figure 3.8: Computation time of fast Fourier transforms with CPU and GPU implementations.

| $\log_2$(Size) | Size | CPU Time | GPU Time | Speedup |
|---:|---:|---:|---:|---:|
| 1 | 2 | 9.8292e-06 | 3.6342e-05 | 0.2705 |
| 2 | 4 | 9.8678e-06 | 3.9292e-05 | 0.2511 |
| 3 | 8 | 1.5825e-05 | 3.6555e-05 | 0.4329 |
| 4 | 16 | 2.7404e-05 | 3.6520e-05 | 0.7504 |
| 5 | 32 | 6.8549e-05 | 3.6483e-05 | 1.8789 |
| 6 | 64 | 1.6025e-04 | 3.2893e-05 | 4.8718 |
| 7 | 128 | 3.8541e-04 | 3.2541e-05 | 11.8438 |
| 8 | 256 | 8.9296e-04 | 3.6775e-05 | 24.2818 |
| 9 | 512 | 2.0721e-03 | 3.2685e-05 | 63.3974 |
| 10 | 1024 | 4.6704e-03 | 4.1923e-05 | 111.4044 |
| 11 | 2048 | 1.0511e-02 | 3.6759e-05 | 285.9552 |
| 12 | 4096 | 2.2953e-02 | 7.0362e-05 | 326.2083 |
| 13 | 8192 | 5.0333e-02 | 1.7722e-04 | 284.0205 |
| 14 | 16384 | 1.1100e-01 | 2.5508e-04 | 435.1616 |
| 15 | 32768 | 2.6045e-01 | 5.1627e-04 | 504.4888 |
| 16 | 65536 | 6.5235e-01 | 9.9759e-04 | 653.9286 |

Table 3.3: Computation time of 1D fast Fourier transform on CPU vs GPU.

| $\log 2$(Size) | Size | CPU Time | GPU Time | Speedup |
|---:|---:|---:|---:|---:|
| 1 | 2 | 8.1623e-05 | 5.8891e-05 | 1.3860 |
| 2 | 4 | 8.2869e-05 | 4.9318e-05 | 1.6803 |
| 3 | 8 | 9.5595e-05 | 4.9008e-05 | 1.9506 |
| 4 | 16 | 9.7627e-05 | 7.2129e-05 | 1.3535 |
| 5 | 32 | 1.5366e-04 | 9.7412e-05 | 1.5774 |
| 6 | 64 | 3.9668e-04 | 1.0316e-04 | 3.8452 |
| 7 | 128 | 1.3328e-03 | 2.3928e-04 | 5.5698 |
| 8 | 256 | 8.6852e-03 | 9.1548e-04 | 9.4871 |
| 9 | 512 | 4.8673e-02 | 3.1805e-03 | 15.3035 |
| 10 | 1024 | 2.1038e-01 | 1.4876e-02 | 14.1426 |
| 11 | 2048 | 1.0108e+00 | 6.4937e-02 | 15.5658 |

Table 3.4: Computation time of 2D fast Fourier transform on CPU vs GPU.

| log 2(Size) | Size | CPU Time | GPU Time | Speedup |
|---:|---:|---:|---:|---:|
| 1 | 2 | 9.1242e-05 | 4.3251e-05 | 2.1096 |
| 2 | 4 | 9.2973e-05 | 5.0182e-05 | 1.8527 |
| 3 | 8 | 1.0116e-04 | 4.3444e-05 | 2.3285 |
| 4 | 16 | 1.1425e-04 | 5.6569e-05 | 2.0197 |
| 5 | 32 | 1.9137e-04 | 5.1307e-05 | 3.7300 |
| 6 | 64 | 3.7088e-04 | 1.1903e-04 | 3.1159 |
| 7 | 128 | 1.6324e-03 | 4.0469e-04 | 4.0337 |
| 8 | 256 | 6.3456e-03 | 1.3623e-03 | 4.6582 |
| 9 | 512 | 2.8563e-02 | 7.5162e-03 | 3.8002 |
| 10 | 1024 | 1.1391e-01 | 2.8350e-02 | 4.0179 |
| 11 | 2048 | 6.2770e-01 | 2.4214e-01 | 2.5923 |

Table 3.5: Computation time of 2D real-to-complex fast Fourier transform on CPU vs GPU.

| log 2(Size) | Size | CPU Time | GPU Time | Speedup |
|---:|---:|---:|---:|---:|
| 1 | 2 | 1.2220e-04 | 4.4764e-05 | 2.7299 |
| 2 | 4 | 1.4426e-04 | 4.4870e-05 | 3.2151 |
| 3 | 8 | 1.4919e-04 | 4.4843e-05 | 3.3268 |
| 4 | 16 | 2.6912e-04 | 4.4649e-05 | 6.0276 |
| 5 | 32 | 4.5526e-04 | 8.8517e-05 | 5.1432 |
| 6 | 64 | 4.7522e-04 | 1.3205e-04 | 3.5988 |
| 7 | 128 | 1.0721e-02 | 4.3613e-04 | 24.5832 |
| 8 | 256 | 1.5637e-02 | 1.3713e-03 | 11.4038 |
| 9 | 512 | 4.7771e-02 | 7.2078e-03 | 6.6276 |
| 10 | 1024 | 1.9013e-01 | 2.9989e-02 | 6.3400 |
| 11 | 2048 | 8.6536e-01 | 2.4508e-01 | 3.5310 |

Table 3.6: Computation time of 2D complex-to-real fast Fourier transform on CPU vs GPU.

FFT[9]. The data sizes used are the powers of two. The measured times are plotted on a log-log scale, with the base-two logarithm of the size as the horizontal axis, and the base-ten logarithm of the time as the vertical axis. The timings are also shown in Tables 3.3, 3.4, 3.5 and 3.6. The speedup displayed in this table is computed as CPU time divided by GPU time.

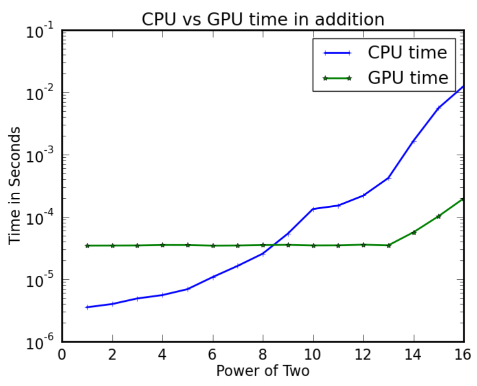
For 1D fast Fourier transform, the running time on the GPU is faster than the time running on the CPU in most cases. The speedup increases rapidly as we use longer vectors. When the vector size is 1024, the GPU FFT is more than 100 times faster than the CPU FFT. The speedup exceeds 600 when the vector size is 65536. For 2D fast Fourier transforms, the GPU implementation is faster than the CPU implementation in all cases. When the data size is more than or equal to 512, we get more than ten times speedup. We speculate that the speedups for 2D FFTs are smaller than those of 1D FFTs because more data movement is needed for 2D FFTs, thus diminishing the performance gained in parallelism. In the cases of 2D real-to-complex and complex-to-real Fourier transforms, the GPU times are a few times faster than the CPU times.

Next, we compare the speeds of CPU and GPU implementations for computing vector additions and dot products. Again, we use vector sizes that are powers of two, and the log-log scale is used for the plots. The computation times for the two operations are shown in Tables 3.7 and 3.8, and plotted in Figures 3.9a and 3.9b. For vectors of sizes smaller or equal to 256, the CPU computations are faster than those on the GPU. However for larger vectors, the GPU outperforms the CPU and the speedup quickly increases to over 50 for vectors of length 65536.

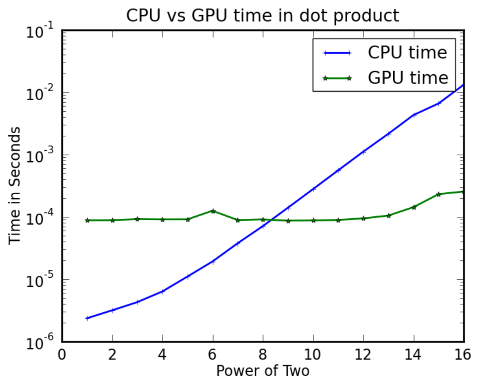
Finally, we measure time of using CGLS to solve the image deblurring

returns the non-redundant entries.

[9] The input of complex-to-real FFT only includes the non-redundant entries, the rest of the data are deduced through conjugate symmetry.



(a) vector addition



(b) vector addition

Figure 3.9: Computation time of vector addition and dot product with CPU and GPU implementations.

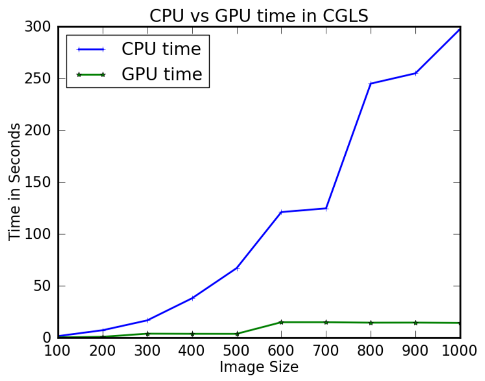


Figure 3.10: Computation time of 100 CGLS iterations in image deblurring with CPU and GPU implementations.

| log 2(Size) | Size | CPU Time | GPU Time | Speedup |
|---:|---:|---:|---:|---:|
| 1 | 2 | 3.5498e-06 | 3.4664e-05 | 0.1024 |
| 2 | 4 | 3.9965e-06 | 3.4725e-05 | 0.1151 |
| 3 | 8 | 4.9097e-06 | 3.4826e-05 | 0.1410 |
| 4 | 16 | 5.5590e-06 | 3.5440e-05 | 0.1569 |
| 5 | 32 | 6.9150e-06 | 3.5381e-05 | 0.1954 |
| 6 | 64 | 1.0794e-05 | 3.4601e-05 | 0.3120 |
| 7 | 128 | 1.6362e-05 | 3.4780e-05 | 0.4704 |
| 8 | 256 | 2.5730e-05 | 3.5355e-05 | 0.7278 |
| 9 | 512 | 5.3905e-05 | 3.5606e-05 | 1.5139 |
| 10 | 1024 | 1.3408e-04 | 3.4856e-05 | 3.8466 |
| 11 | 2048 | 1.5189e-04 | 3.4948e-05 | 4.3460 |
| 12 | 4096 | 2.1932e-04 | 3.5809e-05 | 6.1247 |
| 13 | 8192 | 4.2075e-04 | 3.4970e-05 | 12.0318 |
| 14 | 16384 | 1.6597e-03 | 5.6664e-05 | 29.2909 |
| 15 | 32768 | 5.5902e-03 | 1.0256e-04 | 54.5096 |
| 16 | 65536 | 1.2549e-02 | 1.9783e-04 | 63.4318 |

Table 3.7: Computation time of vector addition on CPU vs GPU.

problem. An image of a particular size is first blurred by a Gaussian PSF of size $11 \times 11$, then we run 100 iterations of CGLS to remove the blur. The timing is listed in Table 3.9. It is also plotted in Figure 3.10. The horizontal axis shows the length of one side of the image, while the vertical axis shows the computation time in seconds. As the size of the image increases, the CPU computation time increases quadratically, but the GPU computation time remains almost constant with two small jumps when the size increases from 200 to 300 and from 500 to 600. We speculate that these jumps are due to the saturation of communication and computation on the GPUs. From

| $\log 2$(Size) | Size | CPU Time | GPU Time | Speedup |
|---:|---:|---:|---:|---:|
| 1 | 2 | 2.3637e-06 | 8.8247e-05 | 0.0268 |
| 2 | 4 | 3.1647e-06 | 8.8618e-05 | 0.0357 |
| 3 | 8 | 4.2835e-06 | 9.2052e-05 | 0.0465 |
| 4 | 16 | 6.3518e-06 | 9.1285e-05 | 0.0696 |
| 5 | 32 | 1.1022e-05 | 9.1363e-05 | 0.1206 |
| 6 | 64 | 1.9264e-05 | 1.2547e-04 | 0.1535 |
| 7 | 128 | 3.7726e-05 | 8.9039e-05 | 0.4237 |
| 8 | 256 | 7.0906e-05 | 9.1228e-05 | 0.7772 |
| 9 | 512 | 1.4012e-04 | 8.7117e-05 | 1.6084 |
| 10 | 1024 | 2.7859e-04 | 8.7849e-05 | 3.1713 |
| 11 | 2048 | 5.5765e-04 | 8.9324e-05 | 6.2431 |
| 12 | 4096 | 1.1106e-03 | 9.4431e-05 | 11.7614 |
| 13 | 8192 | 2.1565e-03 | 1.0507e-04 | 20.5240 |
| 14 | 16384 | 4.3082e-03 | 1.4318e-04 | 30.0900 |
| 15 | 32768 | 6.6053e-03 | 2.3167e-04 | 28.5113 |
| 16 | 65536 | 1.3192e-02 | 2.5576e-04 | 51.5784 |

Table 3.8: Computation time of dot product on CPU vs GPU.

the timing, we see about an order of magnitude speedup in most cases. The greatest speedup of 20 is achieved when the image is $1000 \times 1000$.

## 3.5 Conclusions for this Chapter

In this chapter, we discussed the implementation details of deblurring algorithms. We began with a theoretical overview of convolution matrix operations, and direct and iterative deblurring methods. Then we explained our choice of using Python for our CPU implementation PYRET. We use an

| Image Size | CPU Time | GPU Time | Speedup |
|---|---|---|---|
| 100 × 100 | 1.4245e+00 | 2.6947e-01 | 5.2862 |
| 200 × 200 | 7.0978e+00 | 7.6932e-01 | 9.2260 |
| 300 × 300 | 1.6612e+01 | 3.7942e+00 | 4.3782 |
| 400 × 400 | 3.7870e+01 | 3.6744e+00 | 10.3065 |
| 500 × 500 | 6.6984e+01 | 3.6351e+00 | 18.4269 |
| 600 × 600 | 1.2101e+02 | 1.4832e+01 | 8.1589 |
| 700 × 700 | 1.2459e+02 | 1.4829e+01 | 8.4017 |
| 800 × 800 | 2.4485e+02 | 1.4407e+01 | 16.9951 |
| 900 × 900 | 2.5476e+02 | 1.4471e+01 | 17.6051 |
| 1000 × 1000 | 2.9747e+02 | 1.4202e+01 | 20.9454 |

Table 3.9: Computation time of CGLS on CPU vs GPU.

object-oriented approach to represent the convolution matrix and preconditioner matrix. This saves a lot of storage and time, and allows the reuse of codes written for other classes. We also supply a web GUI interface to PYRET using the web framework Pylons.

The rest of the chapter discussed another package, PARRET, which is a parallel implementation on NVIDIA Compute Unified Device Architecture (CUDA). We provided motivation for parallelizing deblurring algorithms and using GPUs as the programming platform. After an introduction of CUDA and its Python wrapper PyCUDA, we presented our work in extending PyCUDA to support CUBLAS and CUFFT libraries and complex arithmetic. In PARRET, we include wrappers for CUBLAS and CUFFT, a psfMatrix class for the convolution matrix, codes to obtain synthetic boundary conditions and several linear solvers to run on CPUs or GPUs. Finally, our experimental results showed an order of magnitude speedup of GPU computation over CPU computation on a consumer-grade laptop.

# Chapter 4

# Multi-frame Pupil Phase Blind Deconvolution Problem

## 4.1 Overview of Blind Deconvolution

In many situations the PSF is not known, and it is necessary to use deblurring algorithms that can jointly estimate the PSF and the unknown true image scene. This is referred to as blind deconvolution. If we know a parametrized formula of the PSF, we can formulate the blind deconvolution problem as follows:

$$\boldsymbol{y} = \boldsymbol{A}(\boldsymbol{\phi})\boldsymbol{x}, \tag{4.1}$$

or equivalently,

$$\boldsymbol{Y} = \boldsymbol{H}(\boldsymbol{\phi}) * \boldsymbol{X}, \tag{4.2}$$

where $\boldsymbol{\phi}$ is a vector of some unknown parameters. For example, if we know the PSF is represented by a Gaussian function, with unknown mean $(\mu_1, \mu_2)$ and standard deviation $\sigma$, we can take $\boldsymbol{\phi} = (\mu_1, \mu_2, \sigma)$ and

$$H(\boldsymbol{\phi})_{i,j} = H(\mu_1, \mu_2, \sigma)_{i,j} = \frac{1}{2\pi\sigma^2} e^{-\frac{(i-\mu_1)^2+(j-\mu_2)^2}{2\sigma^2}}. \tag{4.3}$$

Usually, there are far fewer parameters than values in the PSF, thus parametrization helps reduce the number of unknowns of the problem. However, sometimes, as we discuss in this chapter, there are almost as many parameters as values in the PSF, and although parametrization may not substantially reduce the number of unknowns, it still serves as a strong constraint on the structure of the PSF.

To solve the blind deconvolution problem, one can repose (4.1) as a minimization problem:

$$\min_{\boldsymbol{\phi},\boldsymbol{x}} \Big( f(\boldsymbol{\phi},\boldsymbol{x}) := \|\boldsymbol{y} - \boldsymbol{A}(\boldsymbol{\phi})\boldsymbol{x}\|_2^2 \Big). \tag{4.4}$$

One way to solve the resulting minimization problem is the alternating minimization algorithm [11].

**Algorithm 4.1** Alternating minimization to minimize $f(\boldsymbol{\phi},\boldsymbol{x})$

```
while not converged do
    $\boldsymbol{\phi} \Leftarrow \arg\min_{\boldsymbol{\phi}} f(\boldsymbol{\phi},\boldsymbol{x})$
    $\boldsymbol{x} \Leftarrow \arg\min_{\boldsymbol{x}} f(\boldsymbol{\phi},\boldsymbol{x})$
end while
```

This alternating minimization method can sometimes take many iterations without making much progress. For solving blind deconvolution problems, we use an alternative approach called the variable projection method.

## 4.2 Variable Projection Method

In a minimization problem, when the objective function depends on two variables, and the subproblem of minimizing with respect to just one of the two variables has an easy solution, we can use the variable projection method to reduce the number of variables.

Consider the objective function $f(\boldsymbol{\phi}, \boldsymbol{x})$ for each fixed $\boldsymbol{\phi}$, and define

$$\tilde{f}(\boldsymbol{\phi}) = \min_{\boldsymbol{x}} f(\boldsymbol{\phi}, \boldsymbol{x}), \tag{4.5}$$

and thus

$$\min_{\boldsymbol{\phi},\boldsymbol{x}} f(\boldsymbol{\phi}, \boldsymbol{x}) = \min_{\boldsymbol{\phi}} \min_{\boldsymbol{x}} f(\boldsymbol{\phi}, \boldsymbol{x}) = \min_{\boldsymbol{\phi}} \tilde{f}(\boldsymbol{\phi}). \tag{4.6}$$

Thus, minimizing $f(\boldsymbol{\phi}, \boldsymbol{x})$ is equivalent to minimizing $\tilde{f}(\boldsymbol{\phi})$. Note that $\boldsymbol{x}$ in $f(\boldsymbol{\phi}, \boldsymbol{x})$ is eliminated in (4.5) and the projected function $\tilde{f}(\boldsymbol{\phi})$ only depends on $\boldsymbol{\phi}$. If there is an efficient algorithm or even closed form formula for the projection, working on the projected function can be much cheaper than the original function.

The variable projection method is used in many nonlinear optimization problems with separable variables [30, 31, 39, 53, 54, 60]. However there is a big difference between problems in these references and our applications of interest. Problems in these references have only a few (usually less than five) parameters left after the projection, while we are interested in applications where tens of thousands of parameters may still remain. For example, in an astronomical imaging test problem described in Section 4.5, we still have over 60,000 parameters left after the projection. Thus we still have a very difficult minimization problem even after the projection.

## 4.3 Applying Variable Projection Method to Blind Deconvolution Problems

In the blind deconvolution problem, the matrix $\boldsymbol{A}$ is a convolution matrix. For simplicity, we assume periodic boundary conditions. This assumption makes sense for our application of interest, which deals with astronomical images.

The function we minimize is given by

$$f(\boldsymbol{\phi}, \boldsymbol{x}) = \|\boldsymbol{y} - \boldsymbol{A}(\boldsymbol{\phi})\boldsymbol{x}\|_2^2 . \tag{4.7}$$

One can note that $f$ depends on $\boldsymbol{\phi}$ nonlinearly and on $\boldsymbol{x}$ linearly. We apply the variable projection method to eliminate the linear variable $\boldsymbol{x}$. The resulting projected function $\tilde{f}(\boldsymbol{\phi})$ is obtained by the following subproblem.

$$\tilde{f}(\boldsymbol{\phi}) = \min_{\boldsymbol{x}} f(\boldsymbol{\phi}, \boldsymbol{x}) = \min_{\boldsymbol{x}} \|\boldsymbol{y} - \boldsymbol{A}(\boldsymbol{\phi})\boldsymbol{x}\|_2^2 . \tag{4.8}$$

The subproblem in (4.8) is a linear least squares problem. By simple numerical linear algebra, we know that the minimum of the subproblem is attained at

$$\hat{x} = \boldsymbol{A}(\boldsymbol{\phi})^\dagger \boldsymbol{y}, \tag{4.9}$$

where $\boldsymbol{A}(\boldsymbol{\phi})^\dagger$ is the pseudoinverse of $\boldsymbol{A}(\boldsymbol{\phi})$. Thus

$$\tilde{f}(\boldsymbol{\phi}) = \left\|\boldsymbol{y} - \boldsymbol{A}(\boldsymbol{\phi})\boldsymbol{A}(\boldsymbol{\phi})^\dagger \boldsymbol{y}\right\|_2^2 = \left\|(\boldsymbol{I} - \boldsymbol{A}(\boldsymbol{\phi})\boldsymbol{A}(\boldsymbol{\phi})^\dagger)\boldsymbol{y}\right\|_2^2 . \tag{4.10}$$

Next, we derive a spectral decomposition of $\tilde{f}(\boldsymbol{\phi})$. With the assumption of periodic boundary conditions, the convolution matrix $\boldsymbol{A}(\boldsymbol{\phi})$ has the following spectral decomposition:

$$\boldsymbol{A}(\boldsymbol{\phi}) = \boldsymbol{F}^H \boldsymbol{\Lambda}(\boldsymbol{\phi}) \boldsymbol{F}, \tag{4.11}$$

where $\boldsymbol{F}$ is the Fourier matrix and $\boldsymbol{\Lambda}(\boldsymbol{\phi})$ is a diagonal matrix. If $\boldsymbol{H}(\boldsymbol{\phi})$ of size $n \times n$ is the PSF corresponding to $\boldsymbol{A}(\boldsymbol{\phi})$, then

$$\boldsymbol{\Lambda}(\boldsymbol{\phi}) = \operatorname{Diag}\left(\operatorname{vec}(n \operatorname{fft}(\boldsymbol{H}(\boldsymbol{\phi})))\right). \tag{4.12}$$

The matrix $\boldsymbol{A}(\boldsymbol{\phi})$ is usually ill-conditioned, thus instead of taking $\boldsymbol{A}(\boldsymbol{\phi})^\dagger$ as the Moore-Penrose pseudoinverse, we use a regularized pseudoinverse:

$$\boldsymbol{A}(\boldsymbol{\phi})^\dagger = (\boldsymbol{A}(\boldsymbol{\phi})^H \boldsymbol{A}(\boldsymbol{\phi}) + \alpha^2)^{-1} \boldsymbol{A}(\boldsymbol{\phi})^H = \boldsymbol{F}^H \frac{\overline{\boldsymbol{\Lambda}(\boldsymbol{\phi})}}{|\boldsymbol{\Lambda}(\boldsymbol{\phi})|^2 + \alpha^2} \boldsymbol{F}, \tag{4.13}$$

where $\alpha$ is a regularization parameter. Here we use the shorthand notation $\alpha^2$ in place of $\alpha^2\boldsymbol{I}$, and arithmetic operations between diagonal matrices to mean elementwise operations on the diagonal entries.

For notational convenience, we drop "$(\boldsymbol{\phi})$" in equations henceforth. We also use the following notation:

$$\boldsymbol{P} = \boldsymbol{A}\boldsymbol{A}^{\dagger} = \boldsymbol{F}^H \frac{|\boldsymbol{\Lambda}|^2}{|\boldsymbol{\Lambda}|^2 + \alpha^2} \boldsymbol{F} \tag{4.14}$$

and

$$\boldsymbol{P}^{\perp} = \boldsymbol{I} - \boldsymbol{P} = \boldsymbol{I} - \boldsymbol{A}\boldsymbol{A}^{\dagger} = \boldsymbol{F}^H \frac{\alpha^2}{|\boldsymbol{\Lambda}|^2 + \alpha^2} \boldsymbol{F}. \tag{4.15}$$

With this notation and (4.10), the minimization problem can be reposed as

$$\min_{\boldsymbol{\phi}} \left( \tilde{f}(\boldsymbol{\phi}) := \left\| \boldsymbol{P}^{\perp}(\boldsymbol{\phi}) \boldsymbol{y} \right\|_2^2 \right). \tag{4.16}$$

Equation (4.16) is a nonlinear least squares problem. We use the Gauss-Newton algorithm with conjugate gradient for the inner iterations (Algorithm 4.2), to solve this problem.

**Algorithm 4.2** Solving the blind deconvolution problem by Gauss-Newton algorithm with conjugate gradient as the inner solver

```
while not converged do
    Solve the normal equations
        $\boldsymbol{J}(\boldsymbol{\phi})^H \boldsymbol{J}(\boldsymbol{\phi})\boldsymbol{p} = -\boldsymbol{J}(\boldsymbol{\phi})^H \boldsymbol{r},$    (4.17)
      where
        $\boldsymbol{r} = \boldsymbol{y} - \boldsymbol{A}(\boldsymbol{\phi})\boldsymbol{x} = \boldsymbol{P}^{\perp}(\boldsymbol{\phi})\boldsymbol{y},$    (4.18)
        $\boldsymbol{J} = \nabla(\boldsymbol{P}^{\perp}(\boldsymbol{\phi})\boldsymbol{y}) = \nabla\boldsymbol{P}^{\perp}(\boldsymbol{\phi})\boldsymbol{y},$    (4.19)
    for the search direction $\boldsymbol{p}$ by conjugate gradient method.
    Set $\boldsymbol{\phi} \Leftarrow \boldsymbol{\phi} + \alpha\boldsymbol{p}$, where $\alpha$ is chosen using a line search on minimizing $\|\boldsymbol{r}\|_2$.
end while
Set $\boldsymbol{x} \Leftarrow \boldsymbol{A}(\boldsymbol{\phi})^{\dagger}\boldsymbol{y}$.
```

The $\nabla$ in (4.19) denotes differentiation with respect to $\boldsymbol{\phi}$. Thus $\nabla\boldsymbol{P}^{\perp}$ is a three-dimensional tensor. Care must be taken when computing the multiplication of $\nabla\boldsymbol{P}^{\perp}\boldsymbol{y}$: the inner product is done along the second dimension of $\nabla\boldsymbol{P}^{\perp}$.

In general, multiplication of a three-dimensional tensor to a vector takes $O(N^3)$ operations, where $N = n^2$ is the number of pixels in the image. Using the spectral decomposition of $\nabla\boldsymbol{P}^{\perp}$ and the special property of our test problem (to be discussed in Section 4.5), we can reduce the complexity down to just $O(N \log N)$. We prove in Appendix 6.3 that

$$\nabla\left(\frac{1}{|\boldsymbol{\Lambda}|^2 + \alpha^2}\right) = -2\frac{\operatorname{Re}\left(\overline{\boldsymbol{\Lambda}}\nabla\boldsymbol{\Lambda}\right)}{(|\boldsymbol{\Lambda}|^2 + \alpha^2)^2}, \tag{4.20}$$

where $\operatorname{Re}(\cdot)$ returns the real part of a complex matrix or tensor. Then from (4.15),

$$\nabla \boldsymbol{P}^{\perp} = \alpha^2 \boldsymbol{F}^H \nabla \left( \frac{1}{|\boldsymbol{\Lambda}|^2 + \alpha^2} \right) \boldsymbol{F} = -2\alpha^2 \boldsymbol{F}^H \frac{\mathrm{Re}\left(\overline{\boldsymbol{\Lambda}} \nabla \boldsymbol{\Lambda}\right)}{(|\boldsymbol{\Lambda}|^2 + \alpha^2)^2} \boldsymbol{F}. \tag{4.21}$$

Therefore, the Jacobian matrix $\boldsymbol{J}$ has the spectral decomposition:

$$\begin{aligned} \boldsymbol{J} &= \nabla \boldsymbol{P}^{\perp} \boldsymbol{y} && (4.22) \\ &= -2\alpha^2 \boldsymbol{F}^H \frac{\mathrm{Re}\left(\overline{\boldsymbol{\Lambda}} \nabla \boldsymbol{\Lambda}\right)}{(|\boldsymbol{\Lambda}|^2 + \alpha^2)^2} \boldsymbol{F} \boldsymbol{y} && (4.23) \\ &= -2\alpha^2 \boldsymbol{F}^H \frac{\mathrm{Re}\left(\overline{\boldsymbol{\Lambda}} \nabla \boldsymbol{\Lambda}\right)}{(|\boldsymbol{\Lambda}|^2 + \alpha^2)^2} \hat{\boldsymbol{y}}. && (4.24) \end{aligned}$$

Here we use $\hat{\boldsymbol{y}}$ to denote the Fourier transform of $\boldsymbol{y}$. Again, the tensor-vector products in (4.23) are (4.24) are done along the second dimension of the tensor $\mathrm{Re}\left(\overline{\boldsymbol{\Lambda}} \nabla \boldsymbol{\Lambda}\right)$. Since $\boldsymbol{\Lambda}$ is diagonal, the tensor $\mathrm{Re}\left(\overline{\boldsymbol{\Lambda}} \nabla \boldsymbol{\Lambda}\right)$ essentially has only two dimensions.

We note that (4.24) only involves the Fourier matrix $\boldsymbol{F}$, a diagonal matrix $\boldsymbol{\Lambda}$ and a "diagonal" tensor $\nabla\boldsymbol{\Lambda}$. Multiplication by $\boldsymbol{F}$, $\boldsymbol{\Lambda}$ and $\nabla\boldsymbol{\Lambda}$ can be done respectively in $O(N \log N)$ (by fast Fourier transform [15, 67]), $O(N)$, and $O(N \log N)$ (to be shown in Subsection 4.5.1).

The left hand side of equation (4.17) is thus given by

$$\begin{aligned} \boldsymbol{J}^H \boldsymbol{J} \boldsymbol{p} &= \left[ -2\alpha^2 \boldsymbol{F}^H \frac{\mathrm{Re}\left(\overline{\boldsymbol{\Lambda}} \nabla \boldsymbol{\Lambda}\right)}{(|\boldsymbol{\Lambda}|^2 + \alpha^2)^2} \hat{\boldsymbol{y}} \right]^H \left[ -2\alpha^2 \boldsymbol{F}^H \frac{\mathrm{Re}\left(\overline{\boldsymbol{\Lambda}} \nabla \boldsymbol{\Lambda}\right)}{(|\boldsymbol{\Lambda}|^2 + \alpha^2)^2} \hat{\boldsymbol{y}} \right] \boldsymbol{p} && (4.25) \\ &= 4\alpha^4 \left[ \hat{\boldsymbol{y}}^H \frac{\mathrm{Re}\left( (\nabla \boldsymbol{\Lambda})^T \overline{\boldsymbol{\Lambda}} \right)}{(|\boldsymbol{\Lambda}|^2 + \alpha^2)^2} \boldsymbol{F} \boldsymbol{F}^H \frac{\mathrm{Re}\left(\overline{\boldsymbol{\Lambda}} \nabla \boldsymbol{\Lambda}\right)}{(|\boldsymbol{\Lambda}|^2 + \alpha^2)^2} \hat{\boldsymbol{y}} \right] \boldsymbol{p} && (4.26) \\ &= 4\alpha^4 \left[ \hat{\boldsymbol{y}}^H \frac{\mathrm{Re}\left( (\nabla \boldsymbol{\Lambda})^T \overline{\boldsymbol{\Lambda}} \right) \mathrm{Re}\left(\overline{\boldsymbol{\Lambda}} \nabla \boldsymbol{\Lambda}\right)}{(|\boldsymbol{\Lambda}|^2 + \alpha^2)^4} \hat{\boldsymbol{y}} \right] \boldsymbol{p}, && (4.27) \end{aligned}$$

while the right hand side of equation (4.17) is given by

$$\begin{aligned}
-\boldsymbol{J}^H \boldsymbol{r} &= -\boldsymbol{J}^H \boldsymbol{P}^{\perp} \boldsymbol{y} && (4.28)\\
&= -\left[ -2\alpha^2 \boldsymbol{F}^H \frac{\operatorname{Re}\left(\overline{\boldsymbol{\Lambda}} \nabla \boldsymbol{\Lambda}\right)}{(|\boldsymbol{\Lambda}|^2 + \alpha^2)^2} \hat{\boldsymbol{y}} \right]^H \boldsymbol{F}^H \frac{\alpha^2}{|\boldsymbol{\Lambda}|^2 + \alpha^2} \boldsymbol{F} \boldsymbol{y} && (4.29)\\
&= 2\alpha^4 \hat{\boldsymbol{y}}^H \frac{\operatorname{Re}\left((\nabla \boldsymbol{\Lambda})^T \overline{\boldsymbol{\Lambda}}\right)}{(|\boldsymbol{\Lambda}|^2 + \alpha^2)^2} \boldsymbol{F} \boldsymbol{F}^H \frac{1}{|\boldsymbol{\Lambda}|^2 + \alpha^2} \hat{\boldsymbol{y}} && (4.30)\\
&= 2\alpha^4 \hat{\boldsymbol{y}}^H \frac{\operatorname{Re}\left((\nabla \boldsymbol{\Lambda})^T \overline{\boldsymbol{\Lambda}}\right)}{(|\boldsymbol{\Lambda}|^2 + \alpha^2)^3} \hat{\boldsymbol{y}}. && (4.31)
\end{aligned}$$

Hence equation (4.17) is reduced to

$$4\alpha^4 \left[ \hat{\boldsymbol{y}}^H \frac{\operatorname{Re}\left((\nabla \boldsymbol{\Lambda})^T \overline{\boldsymbol{\Lambda}}\right) \operatorname{Re}\left(\overline{\boldsymbol{\Lambda}} \nabla \boldsymbol{\Lambda}\right)}{(|\boldsymbol{\Lambda}|^2 + \alpha^2)^4} \hat{\boldsymbol{y}} \right] \boldsymbol{p} = 2\alpha^4 \hat{\boldsymbol{y}}^H \frac{\operatorname{Re}\left((\nabla \boldsymbol{\Lambda})^T \overline{\boldsymbol{\Lambda}}\right)}{(|\boldsymbol{\Lambda}|^2 + \alpha^2)^3} \hat{\boldsymbol{y}} \qquad (4.32)$$

$$2 \left[ \hat{\boldsymbol{y}}^H \frac{\operatorname{Re}\left((\nabla \boldsymbol{\Lambda})^T \overline{\boldsymbol{\Lambda}}\right) \operatorname{Re}\left(\overline{\boldsymbol{\Lambda}} \nabla \boldsymbol{\Lambda}\right)}{(|\boldsymbol{\Lambda}|^2 + \alpha^2)^4} \hat{\boldsymbol{y}} \right] \boldsymbol{p} = \hat{\boldsymbol{y}}^H \frac{\operatorname{Re}\left((\nabla \boldsymbol{\Lambda})^T \overline{\boldsymbol{\Lambda}}\right)}{(|\boldsymbol{\Lambda}|^2 + \alpha^2)^3} \hat{\boldsymbol{y}}. \qquad (4.33)$$

To make this equation clearer, we use $\nabla_k \boldsymbol{\Lambda}$ to denote $\frac{d\boldsymbol{\Lambda}}{d\phi_k}$ and $p_j$ to denote the $j$-th component of $\boldsymbol{p}$.[1] The $i$-th components of both sides of (4.33) are then given by

$$\sum_j \left[ 2\hat{\boldsymbol{y}}^H \frac{\operatorname{Re}\left(\nabla_i \boldsymbol{\Lambda}^T \overline{\boldsymbol{\Lambda}}\right) \operatorname{Re}\left(\overline{\boldsymbol{\Lambda}} \nabla_j \boldsymbol{\Lambda}\right)}{(|\boldsymbol{\Lambda}|^2 + \alpha^2)^4} \hat{\boldsymbol{y}} \right] p_j = \hat{\boldsymbol{y}}^H \frac{\operatorname{Re}\left(\nabla_i \boldsymbol{\Lambda} \overline{\boldsymbol{\Lambda}}\right)}{(|\boldsymbol{\Lambda}|^2 + \alpha^2)^3} \hat{\boldsymbol{y}} \qquad (4.34)$$

$$2\hat{\boldsymbol{y}}^H \frac{\operatorname{Re}\left(\nabla_i \boldsymbol{\Lambda}^T \overline{\boldsymbol{\Lambda}}\right)}{(|\boldsymbol{\Lambda}|^2 + \alpha^2)^2} \sum_j \left[ \frac{\operatorname{Re}\left(\overline{\boldsymbol{\Lambda}} \nabla_j \boldsymbol{\Lambda}\right)}{(|\boldsymbol{\Lambda}|^2 + \alpha^2)^2} \hat{\boldsymbol{y}} \right] p_j = \hat{\boldsymbol{y}}^H \frac{\operatorname{Re}\left(\nabla_i \boldsymbol{\Lambda} \overline{\boldsymbol{\Lambda}}\right)}{(|\boldsymbol{\Lambda}|^2 + \alpha^2)^3} \hat{\boldsymbol{y}}. \qquad (4.35)$$

If we use the notation

$$\begin{aligned}
\tilde{\boldsymbol{y}}_i &= \frac{\operatorname{Re}\left(\overline{\boldsymbol{\Lambda}} \nabla_i \boldsymbol{\Lambda}\right)}{(|\boldsymbol{\Lambda}|^2 + \alpha^2)^2} \hat{\boldsymbol{y}} && (4.36)\\
\check{\boldsymbol{y}} &= \frac{1}{|\boldsymbol{\Lambda}|^2 + \alpha^2} \hat{\boldsymbol{y}}, && (4.37)
\end{aligned}$$

[1] We use lightface for scalars (even when they are vector or matrix components), boldface lowercase for vectors, and boldface uppercase for matrices.

equation (4.34) can be further reduced to

$$2\tilde{\boldsymbol{y}}_i^H \sum_j \tilde{\boldsymbol{y}}_j p_j = \tilde{\boldsymbol{y}}_i^H \check{\boldsymbol{y}}. \tag{4.38}$$

The Gauss-Newton algorithm with the simplified formula is shown in Algorithm 4.3.

**Algorithm 4.3** Solving the blind deconvolution problem by Gauss-Newton algorithm with conjugate gradient as the inner solver using the simplified formula

```
while not converged do
    Solve the normal equations
        $$2\tilde{\boldsymbol{y}}_i^H \sum_j \tilde{\boldsymbol{y}}_j p_j = \tilde{\boldsymbol{y}}_i^H \check{\boldsymbol{y}}. \tag{4.39}$$
    where
        $$\tilde{\boldsymbol{y}}_i = \frac{\mathrm{Re}\left(\overline{\boldsymbol{\Lambda}(\boldsymbol{\phi})}\nabla_i \boldsymbol{\Lambda}(\boldsymbol{\phi})\right)}{(|\boldsymbol{\Lambda}(\boldsymbol{\phi})|^2 + \alpha^2)^2}\hat{\boldsymbol{y}} \tag{4.40}$$
        $$\check{\boldsymbol{y}} = \frac{1}{|\boldsymbol{\Lambda}(\boldsymbol{\phi})|^2 + \alpha^2}\hat{\boldsymbol{y}}, \tag{4.41}$$
        $$\tag{4.42}$$
    by conjugate gradient method for the search direction $\boldsymbol{p}$.
    Set $\boldsymbol{\phi} \Leftarrow \boldsymbol{\phi} + \alpha \boldsymbol{p}$, where $\alpha$ is chosen using a line search on minimizing $\|\boldsymbol{r}\|_2$.
end while
Set $\boldsymbol{x} \Leftarrow \boldsymbol{A}(\boldsymbol{\phi})^\dagger \boldsymbol{y}$.
```

When solving the normal equations in (4.39) by the conjugate gradient method, we need to do the multiplications $\sum_j \tilde{\boldsymbol{y}}_j p_j$ and $[\tilde{\boldsymbol{y}}_i^H \check{\boldsymbol{y}}]_{i=1}^n$. There are efficient algorithms for these multiplications. These algorithms make use of the following definition and its two associated lemmas.

**Definition 4.1.** *A vector $\boldsymbol{u}$ of length $n$ is called conjugate symmetric if*

$$\begin{cases} \boldsymbol{u}_1 \text{ is real} \\ \boldsymbol{u}_k = \overline{\boldsymbol{u}_{n-k+2}} \quad \text{for } k = 2, \dots n \end{cases} \tag{4.43}$$

It is a well-known fact that Fourier transforms of real vectors are conjugate symmetric, which is the case for many vectors in this chapter. We have two lemmas on conjugate symmetric vectors.

**Lemma 4.2.** *If $\boldsymbol{u}$ and $\boldsymbol{v}$ are two conjugate symmetric vectors, then*

$$\overline{\boldsymbol{u}}^T \boldsymbol{v} = \boldsymbol{u}^T \overline{\boldsymbol{v}}. \tag{4.44}$$

**Lemma 4.3.** *If $\boldsymbol{u}$ and $\boldsymbol{v}$ are two conjugate symmetric vectors, then*

$$Re\,(\boldsymbol{u})^T \boldsymbol{v} = \boldsymbol{u}^T \, Re\,(\boldsymbol{v})\,. \tag{4.45}$$

We only need Lemma 4.3 for this section, but Lemma 4.2 will be needed in subsequent sections. Proofs of these two Lemma are given in the Appendix.

In the following, we use the assumptions that $\boldsymbol{p}$ is a real vector and $\check{\boldsymbol{y}}$ is a conjugate symmetric vector, which are true in our algorithm.

First, we note that

$$\sum_j \tilde{\boldsymbol{y}}_j p_j = \sum_j \frac{\text{Re}\left(\overline{\boldsymbol{\Lambda}} \nabla_i \boldsymbol{\Lambda}\right)}{(|\boldsymbol{\Lambda}|^2 + \alpha^2)^2} \hat{\boldsymbol{y}} p_j \tag{4.46}$$

$$= \left( \frac{\text{Re}\left(\overline{\boldsymbol{\Lambda}} \nabla \boldsymbol{\Lambda}\right)}{(|\boldsymbol{\Lambda}|^2 + \alpha^2)^2} \hat{\boldsymbol{y}} \right) \boldsymbol{p} \tag{4.47}$$

$$= \frac{\text{Diag}\,(\hat{\boldsymbol{y}})}{(|\boldsymbol{\Lambda}|^2 + \alpha^2)^2} \text{Re}\left(\overline{\boldsymbol{\Lambda}} \nabla \text{diag}\,(\boldsymbol{\Lambda})\right) \boldsymbol{p} \tag{4.48}$$

$$= \frac{\text{Diag}\,(\hat{\boldsymbol{y}})}{(|\boldsymbol{\Lambda}|^2 + \alpha^2)^2} \text{Re}\left(\overline{\boldsymbol{\Lambda}} \nabla \text{diag}\,(\boldsymbol{\Lambda})\, \boldsymbol{p}\right) \tag{4.49}$$

Equation (4.49) is due to the assumption that $\boldsymbol{p}$ is real. Computing $\sum_j \tilde{\boldsymbol{y}}_j p_j$ as in (4.49) has the advantage that operations can be done from right to left: first compute $\nabla\mathrm{diag}\left(\boldsymbol{\Lambda}\right)\boldsymbol{p}$, then $\overline{\boldsymbol{\Lambda}}\nabla\mathrm{diag}\left(\boldsymbol{\Lambda}\right)\boldsymbol{p}$ and so on. Each intermediate step returns a vector of the same size, thus the need for extra temporary memory is minimized. Also the matrices involved are diagonal matrices except $\nabla\mathrm{diag}\left(\boldsymbol{\Lambda}\right)$, so each step can be done very cheaply, except possibly the step with $\nabla\mathrm{diag}\left(\boldsymbol{\Lambda}\right)$, which is application dependent.

Then for $[\tilde{\boldsymbol{y}}_i^H \boldsymbol{p}]_{i=1}^n$, we consider

$$
\begin{aligned}
[\tilde{\boldsymbol{y}}_i^H \check{\boldsymbol{y}}]_{i=1}^n &= \left[\left(\frac{\mathrm{Re}\left(\overline{\boldsymbol{\Lambda}}\nabla_i\boldsymbol{\Lambda}\right)}{(|\boldsymbol{\Lambda}|^2+\alpha^2)^2}\hat{\boldsymbol{y}}\right)^H \check{\boldsymbol{y}}\right]_{i=1}^n && (4.50)\\
&= \left(\frac{\mathrm{Re}\left(\overline{\boldsymbol{\Lambda}}\nabla\boldsymbol{\Lambda}\right)}{(|\boldsymbol{\Lambda}|^2+\alpha^2)^2}\hat{\boldsymbol{y}}\right)^H \check{\boldsymbol{y}} && (4.51)\\
&= \left(\frac{\mathrm{Diag}\left(\hat{\boldsymbol{y}}\right)}{(|\boldsymbol{\Lambda}|^2+\alpha^2)^2}\mathrm{Re}\left(\overline{\boldsymbol{\Lambda}}\nabla\mathrm{diag}\left(\boldsymbol{\Lambda}\right)\right)\right)^H \check{\boldsymbol{y}} && (4.52)\\
&= \mathrm{Re}\left(\nabla\mathrm{diag}\left(\boldsymbol{\Lambda}\right)\overline{\boldsymbol{\Lambda}}\right)\frac{\mathrm{Diag}\left(\overline{\hat{\boldsymbol{y}}}\right)}{(|\boldsymbol{\Lambda}|^2+\alpha^2)^2}\check{\boldsymbol{y}} && (4.53)\\
&= \nabla\mathrm{diag}\left(\boldsymbol{\Lambda}\right)\overline{\boldsymbol{\Lambda}}\,\mathrm{Re}\left(\frac{\mathrm{Diag}\left(\overline{\hat{\boldsymbol{y}}}\right)}{(|\boldsymbol{\Lambda}|^2+\alpha^2)^2}\check{\boldsymbol{y}}\right) && (4.54)
\end{aligned}
$$

Equation (4.54) is due to the assumption that $\check{\boldsymbol{y}}$ is conjugate symmetric and Lemma 4.3. Again, computing $[\tilde{\boldsymbol{y}}_i^H \check{\boldsymbol{y}}]_{i=1}^n$ as in (4.54) has the advantage that operations can be done from right to left, and the matrices involved are diagonal matrices except $\nabla\mathrm{diag}\left(\boldsymbol{\Lambda}\right)$.

In Subsection 4.5.1, we discuss how to compute multiplications with $\nabla\mathrm{diag}\left(\boldsymbol{\Lambda}\right)$ and its transpose efficiently for our our application of interest.

## 4.4 Deblurring Using More than One Image

We can improve the deblurring result by using more than one blurred image from the same original image. This new problem is called multi-frame blind deconvolution. We are given $m$ blurred images, in which the $i$-th blurred image $\boldsymbol{y}_i$ comes from the original image $\boldsymbol{x}$ with the $i$-th convolution matrix $\boldsymbol{A}(\boldsymbol{\phi}_i)$. Mathematically we can write this as

$$\begin{cases} \boldsymbol{A}(\boldsymbol{\phi}_1)\boldsymbol{x} = \boldsymbol{y}_1 \\ \boldsymbol{A}(\boldsymbol{\phi}_2)\boldsymbol{x} = \boldsymbol{y}_2 \\ \qquad \vdots \\ \boldsymbol{A}(\boldsymbol{\phi}_m)\boldsymbol{x} = \boldsymbol{y}_m \end{cases}, \tag{4.55}$$

or equivalently

$$\begin{bmatrix} \boldsymbol{A}(\boldsymbol{\phi}_1) \\ \boldsymbol{A}(\boldsymbol{\phi}_2) \\ \vdots \\ \boldsymbol{A}(\boldsymbol{\phi}_m) \end{bmatrix} \boldsymbol{x} = \begin{bmatrix} \boldsymbol{y}_1 \\ \boldsymbol{y}_2 \\ \vdots \\ \boldsymbol{y}_m \end{bmatrix}. \tag{4.56}$$

One way to solve this multi-frame blind deconvolution problem is the following minimization problem:

$$\min_{\boldsymbol{\phi},\boldsymbol{x}} \left\| \begin{bmatrix} \boldsymbol{y}_1 \\ \boldsymbol{y}_2 \\ \vdots \\ \boldsymbol{y}_m \end{bmatrix} - \begin{bmatrix} \boldsymbol{A}(\boldsymbol{\phi}_1) \\ \boldsymbol{A}(\boldsymbol{\phi}_2) \\ \vdots \\ \boldsymbol{A}(\boldsymbol{\phi}_m) \end{bmatrix} \boldsymbol{x} \right\|_2^2. \tag{4.57}$$

Using variable projection, we eliminate $\boldsymbol{x}$ by substituting

$$\boldsymbol{x} = \begin{bmatrix} \boldsymbol{A}(\boldsymbol{\phi}_1) \\ \boldsymbol{A}(\boldsymbol{\phi}_2) \\ \vdots \\ \boldsymbol{A}(\boldsymbol{\phi}_m) \end{bmatrix}^{\dagger} \begin{bmatrix} \boldsymbol{y}_1 \\ \boldsymbol{y}_2 \\ \vdots \\ \boldsymbol{y}_m \end{bmatrix} \tag{4.58}$$

to obtain

$$\min_{\boldsymbol{\phi}} \left\| \left( \boldsymbol{I} - \begin{bmatrix} \boldsymbol{A}(\boldsymbol{\phi}_1) \\ \boldsymbol{A}(\boldsymbol{\phi}_2) \\ \vdots \\ \boldsymbol{A}(\boldsymbol{\phi}_m) \end{bmatrix} \begin{bmatrix} \boldsymbol{A}(\boldsymbol{\phi}_1) \\ \boldsymbol{A}(\boldsymbol{\phi}_2) \\ \vdots \\ \boldsymbol{A}(\boldsymbol{\phi}_m) \end{bmatrix}^{\dagger} \right) \begin{bmatrix} \boldsymbol{y}_1 \\ \boldsymbol{y}_2 \\ \vdots \\ \boldsymbol{y}_m \end{bmatrix} \right\|_2^2 . \tag{4.59}$$

We can then proceed as in Section 4.3 with the Gauss-Newton algorithm. This, however, has several drawbacks. The formula for the pseudoinverse in (4.59) is complicated. Because of the coupling of $\boldsymbol{A}(\boldsymbol{\phi}_i)$'s in the pseudoinverse, the Jacobian matrix has an even more complicated formula and it is dense.

To get a simpler formula and for more efficient implementation, we reformulate the minimization problem (4.55) through decoupling. In particular, we solve each of the individual blind deconvolution problems, allowing reconstruction of different objects $\boldsymbol{x}_i$. However, since each $\boldsymbol{x}_i$ should actually be identical, we include additional constraints that minimize the difference between $\boldsymbol{x}_i$ and $\boldsymbol{x}_{i+1}$. Specifically, we solve the minimization problem

$$\begin{aligned} \min \|\boldsymbol{y}_1 - A(\boldsymbol{\phi}_1)\boldsymbol{x}_1\|_2^2 + \|\boldsymbol{y}_2 - A(\boldsymbol{\phi}_2)\boldsymbol{x}_2\|_2^2 + \cdots + \|\boldsymbol{y}_m - A(\boldsymbol{\phi}_m)\boldsymbol{x}_m\|_2^2 \\ + \|\boldsymbol{x}_1 - \boldsymbol{x}_2\|_2^2 + \|\boldsymbol{x}_2 - \boldsymbol{x}_3\|_2^2 + \cdots + \|\boldsymbol{x}_{m-1} - \boldsymbol{x}_m\|_2^2 + \|\boldsymbol{x}_m - \boldsymbol{x}_1\|_2^2 , \end{aligned} \tag{4.60}$$

where

$$\boldsymbol{x}_i = \boldsymbol{A}(\boldsymbol{\phi}_i)^{\dagger}\boldsymbol{y}_i. \tag{4.61}$$

Equation (4.60) can be rewritten as

$$\min \|\boldsymbol{r}\|_2^2 , \tag{4.62}$$

where

$$\boldsymbol{r} = \begin{bmatrix} \boldsymbol{y}_1 - \boldsymbol{A}(\boldsymbol{\phi}_1)\boldsymbol{x}_1 \\ \boldsymbol{y}_2 - \boldsymbol{A}(\boldsymbol{\phi}_2)\boldsymbol{x}_2 \\ \vdots \\ \boldsymbol{y}_m - \boldsymbol{A}(\boldsymbol{\phi}_m)\boldsymbol{x}_m \\ \boldsymbol{x}_1 - \boldsymbol{x}_2 \\ \boldsymbol{x}_2 - \boldsymbol{x}_3 \\ \vdots \\ \boldsymbol{x}_{m-1} - \boldsymbol{x}_m \\ \boldsymbol{x}_m - \boldsymbol{x}_1 \end{bmatrix}. \tag{4.63}$$

This decoupling idea is similar to an approach used in [70] to solve the deblurring and denoising problem with total variation regularization.

We use the following notations.

$$\begin{aligned} \boldsymbol{J}_i &= \nabla_{\boldsymbol{\phi}_i}\Big(\boldsymbol{y}_i - \boldsymbol{A}(\boldsymbol{\phi}_i)\boldsymbol{x}_i\Big) && (4.64) \\ &= \nabla_{\boldsymbol{\phi}_i}\Big(\boldsymbol{P}^{\perp}(\boldsymbol{\phi}_i)\boldsymbol{y}_i\Big) && (4.65) \\ &= -2\alpha^2 \boldsymbol{F}^H \frac{\mathrm{Re}\left(\overline{\boldsymbol{\Lambda}_i}\nabla\boldsymbol{\Lambda}_i\right)}{(|\boldsymbol{\Lambda}_i|^2 + \alpha^2)^2}\hat{\boldsymbol{y}}_i && (4.66) \\ &= -2\alpha^2 \boldsymbol{F}^H \frac{\mathrm{Diag}\left(\hat{\boldsymbol{y}}_i\right)}{(|\boldsymbol{\Lambda}_i|^2 + \alpha^2)^2}\,\mathrm{Re}\left(\overline{\boldsymbol{\Lambda}_i}\nabla\mathrm{diag}\left(\boldsymbol{\Lambda}_i\right)\right), && (4.67) \end{aligned}$$

and

$$\begin{aligned}
\boldsymbol{K}_i &= \nabla_{\boldsymbol{\phi}_i} \boldsymbol{x}_i && (4.68)\\
&= \nabla_{\boldsymbol{\phi}_i} \left( \boldsymbol{A}(\boldsymbol{\phi}_i)^{\dagger} \boldsymbol{y}_i \right) && (4.69)\\
&= \nabla_{\boldsymbol{\phi}_i} \left( \boldsymbol{F}^H \frac{\overline{\boldsymbol{\Lambda}_i}}{|\boldsymbol{\Lambda}_i|^2 + \alpha^2} \boldsymbol{F} \boldsymbol{y}_i \right) && (4.70)\\
&= \boldsymbol{F}^H \nabla_{\boldsymbol{\phi}_i} \left( \frac{\overline{\boldsymbol{\Lambda}_i}}{|\boldsymbol{\Lambda}_i|^2 + \alpha^2} \right) \boldsymbol{F} \boldsymbol{y}_i && (4.71)\\
&= \boldsymbol{F}^H \frac{\alpha^2 \overline{\nabla \boldsymbol{\Lambda}_i} - \overline{\boldsymbol{\Lambda}_i}^2 \nabla \boldsymbol{\Lambda}_i}{(|\boldsymbol{\Lambda}_i|^2 + \alpha^2)^2} \hat{\boldsymbol{y}}_i && (4.72)\\
&= \boldsymbol{F}^H \frac{\mathrm{Diag}\,(\hat{\boldsymbol{y}}_i)}{(|\boldsymbol{\Lambda}_i|^2 + \alpha^2)^2} (\alpha^2 \overline{\nabla \mathrm{diag}\,(\boldsymbol{\Lambda}_i)} - \overline{\boldsymbol{\Lambda}_i}^2 \nabla \mathrm{diag}\,(\boldsymbol{\Lambda}_i)), && (4.73)
\end{aligned}$$

where $\boldsymbol{A}(\boldsymbol{\phi}_i) = \boldsymbol{F}^H \boldsymbol{\Lambda}_i \boldsymbol{F}$, $\mathrm{diag}\,(\boldsymbol{D})$ extracts the diagonal of matrix $\boldsymbol{D}$ as a column vector, and $\mathrm{Diag}\,(\boldsymbol{v})$ forms a diagonal matrix with diagonal $\boldsymbol{v}$. In (4.72), we use the identity

$$\nabla \left( \frac{\overline{\boldsymbol{\Lambda}}}{|\boldsymbol{\Lambda}|^2 + \alpha^2} \right) = \frac{\alpha^2 \overline{\nabla \boldsymbol{\Lambda}} - \overline{\boldsymbol{\Lambda}}^2 \nabla \boldsymbol{\Lambda}}{(|\boldsymbol{\Lambda}|^2 + \alpha^2)^2}, \tag{4.74}$$

which is proved in the Appendix 6.4.

The Jacobian matrix of (4.62) has the form of

$$\boldsymbol{J} = \begin{bmatrix}
\boldsymbol{J}_1 & & & & \\
& \boldsymbol{J}_2 & & & \\
& & \ddots & & \\
& & & \ddots & \\
& & & & \boldsymbol{J}_m \\
\boldsymbol{K}_1 & -\boldsymbol{K}_2 & & & \\
& \boldsymbol{K}_2 & -\boldsymbol{K}_3 & & \\
& & \ddots & \ddots & \\
& & & \boldsymbol{K}_{m-1} & -\boldsymbol{K}_m \\
-\boldsymbol{K}_1 & & & & \boldsymbol{K}_m
\end{bmatrix}. \tag{4.75}$$

With the decoupling formulation, the multi-frame blind deconvolution problem can be solved by the Gauss-Newton algorithm (Algorithm 4.4).

**Algorithm 4.4** Solving the multi-frame blind deconvolution problem by Gauss-Newton algorithm with conjugate gradient as the inner solver

```
while not converged do
    Solve the normal equations
        $$\boldsymbol{J}^H \boldsymbol{J} \boldsymbol{p} = -\boldsymbol{J}^H \boldsymbol{r}, \qquad (4.76)$$
    where $\boldsymbol{r}$ is given by (4.63) and $\boldsymbol{J}$ is given by (4.67), (4.73) and (4.75), for the search direction $\boldsymbol{p}$ by conjugate gradient method.
    Set $\begin{bmatrix} \boldsymbol{\phi}_1 \\ \boldsymbol{\phi}_2 \\ \vdots \\ \boldsymbol{\phi}_m \end{bmatrix} \Leftarrow \begin{bmatrix} \boldsymbol{\phi}_1 \\ \boldsymbol{\phi}_2 \\ \vdots \\ \boldsymbol{\phi}_m \end{bmatrix} + \alpha \boldsymbol{p}$, where $\alpha$ is chosen using a line search on minimizing $\|\boldsymbol{r}\|_2$.
end while
Set $\boldsymbol{x} = \begin{bmatrix} \boldsymbol{A}(\boldsymbol{\phi}_1) \\ \boldsymbol{A}(\boldsymbol{\phi}_2) \\ \vdots \\ \boldsymbol{A}(\boldsymbol{\phi}_m) \end{bmatrix}^{\dagger} \begin{bmatrix} \boldsymbol{y}_1 \\ \boldsymbol{y}_2 \\ \vdots \\ \boldsymbol{y}_m \end{bmatrix}$.
```

When solving the normal equations (4.76) in Algorithm 4.4 with the conjugate gradient method, we need to multiply $\boldsymbol{J}$ and $\boldsymbol{J}^H$ to vectors. From (4.75), we see that $\boldsymbol{J}$ has a block diagonal structure, and these operations can be done very efficiently. We describe these algorithms in the rest of this section.

For the matrix $\boldsymbol{J}$,

$$\boldsymbol{J}\boldsymbol{p} = \boldsymbol{J}\begin{bmatrix}\boldsymbol{p}_1\\ \boldsymbol{p}_2\\ \vdots\\ \boldsymbol{p}_m\end{bmatrix} = \begin{bmatrix}\boldsymbol{J}_1\boldsymbol{p}_1\\ \boldsymbol{J}_2\boldsymbol{p}_2\\ \vdots\\ \boldsymbol{J}_m\boldsymbol{p}_m\\ \boldsymbol{K}_1\boldsymbol{p}_1 - \boldsymbol{K}_2\boldsymbol{p}_2\\ \boldsymbol{K}_2\boldsymbol{p}_2 - \boldsymbol{K}_3\boldsymbol{p}_3\\ \vdots\\ \boldsymbol{K}_{m-1}\boldsymbol{p}_{m-1} - \boldsymbol{K}_m\boldsymbol{p}_m\\ \boldsymbol{K}_m\boldsymbol{p}_m - \boldsymbol{K}_1\boldsymbol{p}_1\end{bmatrix} \tag{4.77}$$

To save operations, $\boldsymbol{J}_i\boldsymbol{p}_i$ and $\boldsymbol{K}_i\boldsymbol{p}_i$ can be computed together. Assuming $\boldsymbol{p}_i$'s to be real vectors, we note that

$$\boldsymbol{J}_i\boldsymbol{p}_i = -2\alpha^2\boldsymbol{F}^H\frac{\mathrm{Diag}\left(\hat{\boldsymbol{y}}_i\right)}{(\left|\boldsymbol{\Lambda}_i\right|^2+\alpha^2)^2}\,\mathrm{Re}\left(\overline{\boldsymbol{\Lambda}_i}\nabla\mathrm{diag}\left(\boldsymbol{\Lambda}_i\right)\right)\boldsymbol{p}_i \tag{4.78}$$

$$= -2\alpha^2\boldsymbol{F}^H\frac{\mathrm{Diag}\left(\hat{\boldsymbol{y}}_i\right)}{(\left|\boldsymbol{\Lambda}_i\right|^2+\alpha^2)^2}\,\mathrm{Re}\left(\overline{\boldsymbol{\Lambda}_i}\nabla\mathrm{diag}\left(\boldsymbol{\Lambda}_i\right)\boldsymbol{p}_i\right) \tag{4.79}$$

$$\boldsymbol{K}_i\boldsymbol{p}_i = \boldsymbol{F}^H\frac{\mathrm{Diag}\left(\hat{\boldsymbol{y}}_i\right)}{(\left|\boldsymbol{\Lambda}_i\right|^2+\alpha^2)^2}(\alpha^2\overline{\nabla\mathrm{diag}\left(\boldsymbol{\Lambda}_i\right)} - \overline{\boldsymbol{\Lambda}_i}^2\nabla\mathrm{diag}\left(\boldsymbol{\Lambda}_i\right))\boldsymbol{p}_i \tag{4.80}$$

$$= \boldsymbol{F}^H\frac{\mathrm{Diag}\left(\hat{\boldsymbol{y}}_i\right)}{(\left|\boldsymbol{\Lambda}_i\right|^2+\alpha^2)^2}(\alpha^2\overline{\nabla\mathrm{diag}\left(\boldsymbol{\Lambda}_i\right)\boldsymbol{p}_i} - \overline{\boldsymbol{\Lambda}_i}^2\nabla\mathrm{diag}\left(\boldsymbol{\Lambda}_i\right)\boldsymbol{p}_i). \tag{4.81}$$

The algorithm to compute $\boldsymbol{J}\boldsymbol{p}$ as in (4.77) is shown in Algorithm 4.5.

**Algorithm 4.5** Computing the matrix-vector product with the multi-frame Jacobian matrix.

```
**for** $i = 1, 2, \ldots, n$ **do**

$$\tilde{\boldsymbol{p}}_i = \nabla \mathrm{diag}\left(\boldsymbol{\Lambda}_i\right) \boldsymbol{p}_i \tag{4.82}$$
$$\boldsymbol{D}_i = \frac{\mathrm{Diag}\left(\hat{\boldsymbol{y}}_i\right)}{(\left|\boldsymbol{\Lambda}_i\right|^2 + \alpha^2)^2}. \tag{4.83}$$
$$\boldsymbol{J}_i \boldsymbol{p}_i = -2\alpha^2 \,\mathrm{ifft}\left(D_i \,\mathrm{Re}\left(\overline{\boldsymbol{\Lambda}_i} \tilde{\boldsymbol{p}}_i\right)\right) \tag{4.84}$$
$$\boldsymbol{K}_i \boldsymbol{p}_i = \mathrm{ifft}\left(D_i(\alpha^2 * \overline{\tilde{\boldsymbol{p}}_i} - \overline{\boldsymbol{\Lambda}_i}^2 \tilde{\boldsymbol{p}}_i)\right) \tag{4.85}$$

**end for**
**return**
```

$$\boldsymbol{J}\boldsymbol{p} = \boldsymbol{J} \begin{bmatrix} \boldsymbol{p}_1 \\ \boldsymbol{p}_2 \\ \vdots \\ \boldsymbol{p}_m \end{bmatrix} = \begin{bmatrix} \boldsymbol{J}_1 \boldsymbol{p}_1 \\ \boldsymbol{J}_2 \boldsymbol{p}_2 \\ \vdots \\ \boldsymbol{J}_m \boldsymbol{p}_m \\ \boldsymbol{K}_1 \boldsymbol{p}_1 - \boldsymbol{K}_2 \boldsymbol{p}_2 \\ \boldsymbol{K}_2 \boldsymbol{p}_2 - \boldsymbol{K}_3 \boldsymbol{p}_3 \\ \vdots \\ \boldsymbol{K}_{m-1} \boldsymbol{p}_{m-1} - \boldsymbol{K}_m \boldsymbol{p}_m \\ \boldsymbol{K}_m \boldsymbol{p}_m - \boldsymbol{K}_1 \boldsymbol{p}_1 \end{bmatrix} \tag{4.86}$$

Now we consider computing $\boldsymbol{J}^T\boldsymbol{r}$,

$$\boldsymbol{J}^T\boldsymbol{r} = \boldsymbol{J}^T \begin{bmatrix} \boldsymbol{r}_1 \\ \boldsymbol{r}_2 \\ \vdots \\ \boldsymbol{r}_m \\ \boldsymbol{s}_1 \\ \boldsymbol{s}_2 \\ \vdots \\ \boldsymbol{s}_m \end{bmatrix} = \begin{bmatrix} \boldsymbol{J}_1^T\boldsymbol{r}_1 + \boldsymbol{K}_1^T(\boldsymbol{s}_1 - \boldsymbol{s}_m) \\ \boldsymbol{J}_2^T\boldsymbol{r}_2 + \boldsymbol{K}_2^T(\boldsymbol{s}_2 - \boldsymbol{s}_1) \\ \vdots \\ \boldsymbol{J}_m^T\boldsymbol{r}_m + \boldsymbol{K}_m^T(\boldsymbol{s}_m - \boldsymbol{s}_{m-1}) \end{bmatrix}. \tag{4.87}$$

Now for each $i$,

$$\begin{aligned} &\boldsymbol{J}_i^T\boldsymbol{r}_i \\ &= -2\alpha^2 \operatorname{Re}\left(\nabla\mathrm{diag}\left(\boldsymbol{\Lambda}_i\right)^T \overline{\boldsymbol{\Lambda}_i}\right) \frac{\mathrm{Diag}\left(\hat{\boldsymbol{y}}_i\right)}{(\left|\boldsymbol{\Lambda}_i\right|^2 + \alpha^2)^2} \boldsymbol{F}^H\boldsymbol{r}_i && (4.88) \\ &= -2\alpha^2 \nabla\mathrm{diag}\left(\boldsymbol{\Lambda}_i\right)^T \overline{\boldsymbol{\Lambda}_i}\ \operatorname{Re}\left(\frac{\mathrm{Diag}\left(\hat{\boldsymbol{y}}_i\right)}{(\left|\boldsymbol{\Lambda}_i\right|^2 + \alpha^2)^2} \boldsymbol{F}^H\boldsymbol{r}_i\right), && (4.89) \end{aligned}$$

Equation (4.89) is due to Lemma 4.3. We prefer (4.89) over (4.88) because operations in (4.89) can be done from right to left: first compute $\boldsymbol{F}^H\boldsymbol{r}_i$, then $\frac{\mathrm{Diag}(\hat{\boldsymbol{y}}_i)}{(|\boldsymbol{\Lambda}_i|^2+\alpha^2)^2}\boldsymbol{F}^H\boldsymbol{r}_i$ and so on, while for (4.88) we need an intermediate product for $\operatorname{Re}\left(\nabla\mathrm{diag}\left(\boldsymbol{\Lambda}_i\right)^T \overline{\boldsymbol{\Lambda}_i}\right)$. In Subsection 4.5.1, we will describe efficient methods for multiplication with $\nabla\mathrm{diag}\left(\boldsymbol{\Lambda}_i\right)$ and $\nabla\mathrm{diag}\left(\boldsymbol{\Lambda}_i\right)^T$.

Now consider, for each $i$,

$$\begin{aligned} &\boldsymbol{K}_i^T\boldsymbol{s}_i \\ &= (\alpha^2 \overline{\nabla\mathrm{diag}\left(\boldsymbol{\Lambda}_i\right)}^T - \nabla\mathrm{diag}\left(\boldsymbol{\Lambda}_i\right)^T \overline{\boldsymbol{\Lambda}_i}^2) \frac{\mathrm{Diag}\left(\hat{\boldsymbol{y}}_i\right)}{(\left|\boldsymbol{\Lambda}_i\right|^2 + \alpha^2)^2} \boldsymbol{F}^H\boldsymbol{s}_i && (4.90) \\ &= \alpha^2 \overline{\nabla\mathrm{diag}\left(\boldsymbol{\Lambda}_i\right)}^T \frac{\mathrm{Diag}\left(\hat{\boldsymbol{y}}_i\right)}{(\left|\boldsymbol{\Lambda}_i\right|^2 + \alpha^2)^2} \boldsymbol{F}^H s_i - \nabla\mathrm{diag}\left(\boldsymbol{\Lambda}_i\right)^T \overline{\boldsymbol{\Lambda}_i}^2 \frac{\mathrm{Diag}\left(\hat{\boldsymbol{y}}_i\right)}{(\left|\boldsymbol{\Lambda}_i\right|^2 + \alpha^2)^2} \boldsymbol{F}^H s_i && (4.91) \\ &= \alpha^2 \nabla\mathrm{diag}\left(\boldsymbol{\Lambda}_i\right)^T \overline{\frac{\mathrm{Diag}\left(\hat{\boldsymbol{y}}_i\right)}{(\left|\boldsymbol{\Lambda}_i\right|^2 + \alpha^2)^2} \boldsymbol{F}^H s_i} - \nabla\mathrm{diag}\left(\boldsymbol{\Lambda}_i\right)^T \overline{\boldsymbol{\Lambda}_i}^2 \frac{\mathrm{Diag}\left(\hat{\boldsymbol{y}}_i\right)}{(\left|\boldsymbol{\Lambda}_i\right|^2 + \alpha^2)^2} \boldsymbol{F}^H s_i && (4.92) \\ &= \nabla\mathrm{diag}\left(\boldsymbol{\Lambda}_i\right)^T \left[\alpha^2 \overline{\frac{\mathrm{Diag}\left(\hat{\boldsymbol{y}}_i\right)}{(\left|\boldsymbol{\Lambda}_i\right|^2 + \alpha^2)^2} \boldsymbol{F}^H s_i} - \overline{\boldsymbol{\Lambda}_i}^2 \frac{\mathrm{Diag}\left(\hat{\boldsymbol{y}}_i\right)}{(\left|\boldsymbol{\Lambda}_i\right|^2 + \alpha^2)^2} \boldsymbol{F}^H s_i\right]. && (4.93) \end{aligned}$$

Equation (4.92) is due to Lemma 4.2.

Adding $\boldsymbol{J}_i^T \boldsymbol{r}_i$ and $\boldsymbol{K}_i^T \boldsymbol{s}_i$ together, we have

$$\begin{aligned}
&\boldsymbol{J}_i^T \boldsymbol{r}_i + \boldsymbol{K}_i^T \boldsymbol{s}_i \\
&= \nabla \mathrm{diag}\,(\boldsymbol{\Lambda}_i)^T \Bigg[ -2\alpha^2 \overline{\boldsymbol{\Lambda}_i}\, \mathrm{Re}\left( \frac{\mathrm{Diag}\,(\hat{\boldsymbol{y}}_i)}{(|\boldsymbol{\Lambda}_i|^2 + \alpha^2)^2} \boldsymbol{F}^H \boldsymbol{r}_i \right) \\
&\qquad + \alpha^2 \overline{\frac{\mathrm{Diag}\,(\hat{\boldsymbol{y}}_i)}{(|\boldsymbol{\Lambda}_i|^2 + \alpha^2)^2} \boldsymbol{F}^H \boldsymbol{s}_i} - \overline{\boldsymbol{\Lambda}_i}^2 \frac{\mathrm{Diag}\,(\hat{\boldsymbol{y}}_i)}{(|\boldsymbol{\Lambda}_i|^2 + \alpha^2)^2} \boldsymbol{F}^H \boldsymbol{s}_i \Bigg]. \qquad (4.94)
\end{aligned}$$

The algorithm to compute $\boldsymbol{J}^T \boldsymbol{r}$ is shown in Algorithm 4.6.

**Algorithm 4.6** Computing the matrix-vector product with the transpose of the multi-frame Jacobian matrix.

```
for $i = 1, 2, \ldots, n$ do
    $\boldsymbol{D}_i = \frac{\mathrm{Diag}\,(\hat{\boldsymbol{y}}_i)}{(|\boldsymbol{\Lambda}_i|^2 + \alpha^2)^2}$    (4.95)
    $\tilde{\boldsymbol{r}}_i = \boldsymbol{D}_i\, \mathrm{ifft}\,(\boldsymbol{r}_i)$    (4.96)
    $\tilde{\boldsymbol{s}}_i = \boldsymbol{D}_i\, \mathrm{ifft}\,(\boldsymbol{s}_i - \boldsymbol{s}_{i-1})$    (4.97)
    $\boldsymbol{t}_i = -2\alpha^2 \overline{\boldsymbol{\Lambda}_i}\ \mathrm{Re}\,(\tilde{\boldsymbol{r}}_i) + \alpha^2 \overline{\tilde{\boldsymbol{s}}_i} - \overline{\boldsymbol{\Lambda}_i}^2 \tilde{\boldsymbol{s}}_i$    (4.98)
    $\boldsymbol{J}_i^T \boldsymbol{r}_i + \boldsymbol{K}_i^T (\boldsymbol{s}_i - \boldsymbol{s}_{i-1}) = \nabla \mathrm{diag}\,(\boldsymbol{\Lambda}_i)^T \boldsymbol{t}_i.$    (4.99)
end for
return
    $\boldsymbol{J}^T \boldsymbol{r} = \boldsymbol{J}^T \begin{bmatrix} \boldsymbol{r}_1 \\ \vdots \\ \boldsymbol{r}_m \\ \boldsymbol{s}_1 \\ \vdots \\ \boldsymbol{s}_m \end{bmatrix} = \begin{bmatrix} \boldsymbol{J}_1^T \boldsymbol{r}_1 + \boldsymbol{K}_1^T (\boldsymbol{s}_1 - \boldsymbol{s}_m) \\ \boldsymbol{J}_2^T \boldsymbol{r}_2 + \boldsymbol{K}_2^T (\boldsymbol{s}_2 - \boldsymbol{s}_1) \\ \vdots \\ \boldsymbol{J}_m^T \boldsymbol{r}_m + \boldsymbol{K}_m^T (\boldsymbol{s}_m - \boldsymbol{s}_{m-1}) \end{bmatrix}$    (4.100)
```

Finally we consider computations with the complex conjugate transpose,

$\boldsymbol{J}^H$. Since $\boldsymbol{J}$ is real, $\boldsymbol{J}^H = \boldsymbol{J}^T$, but the implementation is slightly different and we can save some operations in computing $\boldsymbol{J}^H\boldsymbol{J}\boldsymbol{p}$. First notice that

$$\boldsymbol{J}^H\boldsymbol{r} = \boldsymbol{J}^H \begin{bmatrix} \boldsymbol{r}_1 \\ \boldsymbol{r}_2 \\ \vdots \\ \boldsymbol{r}_m \\ \boldsymbol{s}_1 \\ \boldsymbol{s}_2 \\ \vdots \\ \boldsymbol{s}_m \end{bmatrix} = \begin{bmatrix} \boldsymbol{J}_1^H\boldsymbol{r}_1 + \boldsymbol{K}_1^H(\boldsymbol{s}_1 - \boldsymbol{s}_m) \\ \boldsymbol{J}_2^H\boldsymbol{r}_2 + \boldsymbol{K}_2^H(\boldsymbol{s}_2 - \boldsymbol{s}_1) \\ \vdots \\ \boldsymbol{J}_m^H\boldsymbol{r}_m + \boldsymbol{K}_m^H(\boldsymbol{s}_m - \boldsymbol{s}_{m-1}) \end{bmatrix} \tag{4.101}$$

Now for each $i$,

$$\begin{aligned} &\boldsymbol{J}_i^H\boldsymbol{r}_i \\ &= -2\alpha^2 \operatorname{Re}\left(\nabla\mathrm{diag}\,(\boldsymbol{\Lambda}_i)^T\,\overline{\boldsymbol{\Lambda}_i}\right)\frac{\mathrm{Diag}\,\left(\overline{\hat{\boldsymbol{y}}_i}\right)}{(|\boldsymbol{\Lambda}_i|^2+\alpha^2)^2}\boldsymbol{F}\boldsymbol{r}_i & (4.102) \\ &= -2\alpha^2\nabla\mathrm{diag}\,(\boldsymbol{\Lambda}_i)^T\,\overline{\boldsymbol{\Lambda}_i}\ \operatorname{Re}\left(\frac{\mathrm{Diag}\,\left(\overline{\hat{\boldsymbol{y}}_i}\right)}{(|\boldsymbol{\Lambda}_i|^2+\alpha^2)^2}\boldsymbol{F}\boldsymbol{r}_i\right), & (4.103) \end{aligned}$$

and

$$\begin{aligned} &\boldsymbol{K}_i^H\boldsymbol{s}_i \\ &= (\alpha^2\nabla\mathrm{diag}\,(\boldsymbol{\Lambda}_i)^T - \overline{\nabla\mathrm{diag}\,(\boldsymbol{\Lambda}_i)}^T\boldsymbol{\Lambda}_i^2)\frac{\mathrm{Diag}\,\left(\overline{\hat{\boldsymbol{y}}_i}\right)}{(|\boldsymbol{\Lambda}_i|^2+\alpha^2)^2}\boldsymbol{F}\boldsymbol{s}_i & (4.104) \\ &= \alpha^2\nabla\mathrm{diag}\,(\boldsymbol{\Lambda}_i)^T\,\frac{\mathrm{Diag}\,\left(\overline{\hat{\boldsymbol{y}}_i}\right)}{(|\boldsymbol{\Lambda}_i|^2+\alpha^2)^2}\boldsymbol{F}s_i - \overline{\nabla\mathrm{diag}\,(\boldsymbol{\Lambda}_i)}^T\boldsymbol{\Lambda}_i^2\frac{\mathrm{Diag}\,\left(\overline{\hat{\boldsymbol{y}}_i}\right)}{(|\boldsymbol{\Lambda}_i|^2+\alpha^2)^2}\boldsymbol{F}s_i & (4.105) \\ &= \alpha^2\nabla\mathrm{diag}\,(\boldsymbol{\Lambda}_i)^T\,\frac{\mathrm{Diag}\,\left(\overline{\hat{\boldsymbol{y}}_i}\right)}{(|\boldsymbol{\Lambda}_i|^2+\alpha^2)^2}\boldsymbol{F}s_i - \nabla\mathrm{diag}\,(\boldsymbol{\Lambda}_i)^T\,\overline{\boldsymbol{\Lambda}_i}^2\,\overline{\frac{\mathrm{Diag}\,\left(\overline{\hat{\boldsymbol{y}}_i}\right)}{(|\boldsymbol{\Lambda}_i|^2+\alpha^2)^2}\boldsymbol{F}s_i} & (4.106) \\ &= \nabla\mathrm{diag}\,(\boldsymbol{\Lambda}_i)^T\left[\alpha^2\frac{\mathrm{Diag}\,\left(\overline{\hat{\boldsymbol{y}}_i}\right)}{(|\boldsymbol{\Lambda}_i|^2+\alpha^2)^2}\boldsymbol{F}s_i - \overline{\boldsymbol{\Lambda}_i}^2\,\overline{\frac{\mathrm{Diag}\,\left(\overline{\hat{\boldsymbol{y}}_i}\right)}{(|\boldsymbol{\Lambda}_i|^2+\alpha^2)^2}\boldsymbol{F}s_i}\right]. & (4.107) \end{aligned}$$

Equations (4.103) and (4.106) are due to Lemmas 4.3 and 4.2 respectively.

Adding $\boldsymbol{J}_i^H \boldsymbol{r}_i$ and $\boldsymbol{K}_i^H \boldsymbol{s}_i$ together, we have

$$\begin{aligned}
&\boldsymbol{J}_i^H \boldsymbol{r}_i + \boldsymbol{K}_i^H \boldsymbol{s}_i \\
&\quad = \nabla \mathrm{diag}\left(\boldsymbol{\Lambda}_i\right)^T \left[ -2\alpha^2 \overline{\boldsymbol{\Lambda}_i} \,\mathrm{Re}\left( \frac{\mathrm{Diag}\left(\overline{\hat{\boldsymbol{y}}_i}\right)}{(|\boldsymbol{\Lambda}_i|^2 + \alpha^2)^2} \boldsymbol{F}\boldsymbol{r}_i \right) \right. \\
&\qquad\qquad \left. + \alpha^2 \frac{\mathrm{Diag}\left(\overline{\hat{\boldsymbol{y}}_i}\right)}{(|\boldsymbol{\Lambda}_i|^2 + \alpha^2)^2} \boldsymbol{F}\boldsymbol{s}_i - \overline{\boldsymbol{\Lambda}_i}^2 \overline{\frac{\mathrm{Diag}\left(\overline{\hat{\boldsymbol{y}}_i}\right)}{(|\boldsymbol{\Lambda}_i|^2 + \alpha^2)^2} \boldsymbol{F}\boldsymbol{s}_i} \right].
\end{aligned} \tag{4.108}$$

The algorithm to compute $\boldsymbol{J}^H \boldsymbol{r}$ is shown in Algorithm 4.7.

**Algorithm 4.7** Computing the matrix-vector product with conjugate transpose of the multi-frame Jacobian matrix.

```
for $i = 1, 2, \ldots, n$ do
    $\overline{\boldsymbol{D}_i} = \frac{\mathrm{Diag}\left(\overline{\hat{\boldsymbol{y}}_i}\right)}{(|\boldsymbol{\Lambda}_i|^2 + \alpha^2)^2}$    (4.109)
    $\tilde{\boldsymbol{r}}_i = \overline{\boldsymbol{D}_i}\,\mathrm{fft}\left(\boldsymbol{r}_i\right)$    (4.110)
    $\tilde{\boldsymbol{s}}_i = \overline{\boldsymbol{D}_i}\,\mathrm{fft}\left(\boldsymbol{s}_i - \boldsymbol{s}_{i-1}\right)$    (4.111)
    $\boldsymbol{t}_i = -2\alpha^2 \overline{\boldsymbol{\Lambda}_i}\ \mathrm{Re}\left(\tilde{\boldsymbol{r}}_i\right) + \alpha^2 \tilde{\boldsymbol{s}}_i - \overline{\boldsymbol{\Lambda}_i}^2 \overline{\tilde{\boldsymbol{s}}_i}$    (4.112)
    $\boldsymbol{J}_i^H \boldsymbol{r}_i + \boldsymbol{K}_i^H (\boldsymbol{s}_i - \boldsymbol{s}_{i-1}) = \nabla \mathrm{diag}\left(\boldsymbol{\Lambda}_i\right)^T \boldsymbol{t}_i.$    (4.113)
end for
return
    $\boldsymbol{J}^H \boldsymbol{r} = \boldsymbol{J}^H \begin{bmatrix} \boldsymbol{r}_1 \\ \vdots \\ \boldsymbol{r}_m \\ \boldsymbol{s}_1 \\ \vdots \\ \boldsymbol{s}_m \end{bmatrix} = \begin{bmatrix} \boldsymbol{J}_1^H \boldsymbol{r}_1 + \boldsymbol{K}_1^H (\boldsymbol{s}_1 - \boldsymbol{s}_m) \\ \boldsymbol{J}_2^H \boldsymbol{r}_2 + \boldsymbol{K}_2^H (\boldsymbol{s}_2 - \boldsymbol{s}_1) \\ \vdots \\ \boldsymbol{J}_m^H \boldsymbol{r}_m + \boldsymbol{K}_m^H (\boldsymbol{s}_m - \boldsymbol{s}_{m-1}) \end{bmatrix}$    (4.114)
```

Some operations can be saved when we compute $\boldsymbol{J}^H \boldsymbol{J}\boldsymbol{p} = \boldsymbol{J}^H(\boldsymbol{J}\boldsymbol{p})$. The last step in the multiplication by $\boldsymbol{J}$ is an inverse Fourier transform, while the first step in the multiplication by $\boldsymbol{J}^H$ is a Fourier transform. These two steps can be skipped in computing $\boldsymbol{J}^H\boldsymbol{J}\boldsymbol{p} = \boldsymbol{J}^H(\boldsymbol{J}\boldsymbol{p})$ as they cancel each other's effect.

## 4.5 Pupil Phase Parametrization of Atmospheric Blurs in Astronomical Imaging

In this section, we describe our target application — removing atmospheric blur in astronomical imaging. Given the pupil phase function $\boldsymbol{\Phi}$, which describes the wavefront at the pupil of a telescope, the PSF is defined by

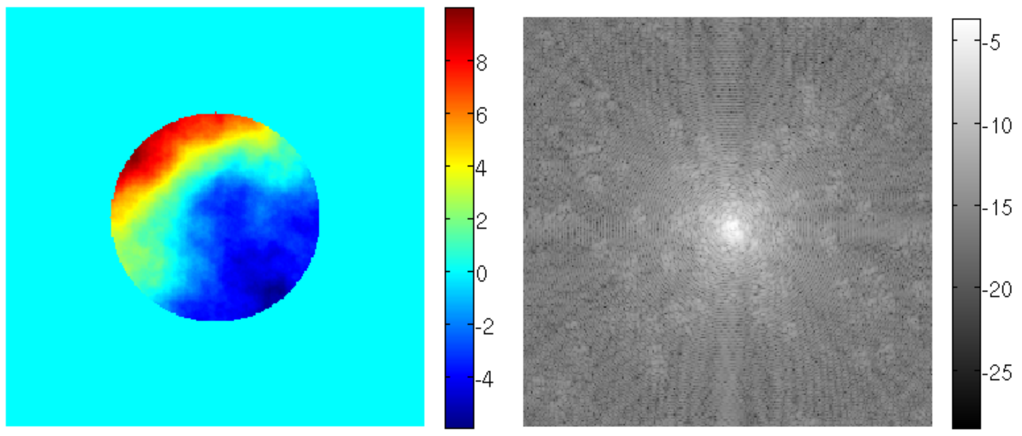
$$\boldsymbol{H} = \left|\operatorname{ifft}\left(e^{\imath\boldsymbol{\Phi}}\right)\right|^2. \tag{4.115}$$

We use $\imath$ to denote $\sqrt{-1}$, $\operatorname{ifft}(\boldsymbol{X})$ to denote the inverse 2D FFT of $\boldsymbol{X}$ and the exponential function in (4.115) is done elementwise. An example of a pupil phase function and its corresponding point spread function is shown in Figure 4.1. In this and subsequent figures of PSFs, the logarithms of the PSFs are shown instead of PSFs themselves for better contrast. Values of the pupil phase function $\boldsymbol{\Phi}$ are zero outside the pupil, hence we only need to consider those values of $\boldsymbol{\Phi}$ inside the pupil.

In the test problem used in this section, $\boldsymbol{\Phi}$ is of size $256 \times 256$, with a total of 65536 elements. After discarding elements outsides the pupil, we still have 12851 elements. Recall from Section 4.4, in blind deconvolution we are minimizing

$$f(\boldsymbol{\Phi}, \boldsymbol{x}) = \min_{\boldsymbol{\Phi},\boldsymbol{x}} \|\boldsymbol{y} - \boldsymbol{A}(\boldsymbol{\phi})\boldsymbol{x}\|_2^2. \tag{4.116}$$

In our test problem, the original image $\boldsymbol{x}$ and the blurred image $\boldsymbol{y}$ each contains 65536 elements. The joint minimization is over space of dimension



(a) pupil phase function (b) point spread function

Figure 4.1: A pupil phase function and its corresponding point spread function.

65536 + 12851 = 78387 using 65536 data values. Using variable projection, we instead are minimizing

$$\tilde{f}(\boldsymbol{\Phi}) = \left\|(\boldsymbol{I} - \boldsymbol{A}(\boldsymbol{\phi})\boldsymbol{A}(\boldsymbol{\phi})^{\dagger})\boldsymbol{y}\right\|_2^2 . \qquad (4.117)$$

The dimension of the search space drops to 12851, but this still is a very large number compared to other problems in the literature that use the variable projection method, in which only a few parameters (e.g., 3) remain after projection.

## 4.5.1 Efficient Computations with $\nabla\mathrm{diag}\,(\boldsymbol{\Lambda})$

From Sections 4.3 and 4.4, we know that efficient application of Gauss-Newton algorithms depends on an efficient way to do multiplication with $\nabla\mathrm{diag}\,(\boldsymbol{\Lambda})$. We now derive the formula for $\nabla\mathrm{diag}\,(\boldsymbol{\Lambda})$ for the specific case when the PSF has the form given in (4.115).

In the following derivation, we vectorize $\mathbf{\Phi}$ by stacking together all entries inside the pupil. We represent this vectorized pupil phase function by $\boldsymbol{\phi}$. Similarly, we vectorize $\boldsymbol{H}$ by stacking its columns together and denote it by $\boldsymbol{h}$. We use $\boldsymbol{F}$ to denote the 2D unitary FFT matrix acting on vectorized matrices, and $\boldsymbol{e}_k$ to denote the unit vector with 1 at $k$–th position and 0 at other positions. We use ".$*$" to denote elementwise multiplication.

With these notations, the convolution matrix $\boldsymbol{A}$ corresponding to the PSF $\boldsymbol{h}$ is given by

$$\boldsymbol{A} = \boldsymbol{F}^H \mathbf{\Lambda} \boldsymbol{F}, \tag{4.118}$$

with

$$\mathbf{\Lambda} = \text{Diag}\left(\sqrt{N}\boldsymbol{F}\boldsymbol{h}\right), \tag{4.119}$$

where $N$ is number of elements in $\boldsymbol{h}$.

We let

$$\boldsymbol{\varphi} = e^{\imath \boldsymbol{\phi}}. \tag{4.120}$$

Then

$$\frac{d\boldsymbol{\varphi}}{d\phi_k} = \imath \varphi_k \boldsymbol{e}_k. \tag{4.121}$$

The formula (4.115) for $\boldsymbol{h}$ can be rewritten as

$$\boldsymbol{h} = \overline{\boldsymbol{F}^H \boldsymbol{\varphi}} .* \boldsymbol{F}^H \boldsymbol{\varphi}. \tag{4.122}$$

Differentiating $\boldsymbol{h}$ with respect to an entry $\phi_k$ of $\boldsymbol{\phi}$, we have

$$\frac{d\boldsymbol{h}}{d\phi_k} = \overline{\boldsymbol{F}^H \frac{d\boldsymbol{\varphi}}{d\phi_k}} . * \boldsymbol{F}^H \boldsymbol{\varphi} + \overline{\boldsymbol{F}^H \boldsymbol{\varphi}} . * \boldsymbol{F}^H \frac{d\boldsymbol{\varphi}}{d\phi_k} \tag{4.123}$$

$$= 2\,\mathrm{Re}\left(\boldsymbol{F}^H \boldsymbol{\varphi} . * \overline{\boldsymbol{F}^H \frac{d\boldsymbol{\varphi}}{d\phi_k}}\right) \tag{4.124}$$

$$= 2\,\mathrm{Re}\left(\boldsymbol{F}^H \boldsymbol{\varphi} . * \overline{\boldsymbol{F}^H \imath \varphi_k \boldsymbol{e}_k}\right) \tag{4.125}$$

$$= 2\,\mathrm{Re}\left(-\imath \boldsymbol{F}^H \boldsymbol{\varphi} . * \overline{\boldsymbol{F}^H \varphi_k \boldsymbol{e}_k}\right) \tag{4.126}$$

$$= 2\,\mathrm{Im}\left(\boldsymbol{F}^H \boldsymbol{\varphi} . * \overline{\boldsymbol{F}^H \varphi_k \boldsymbol{e}_k}\right), \tag{4.127}$$

where $\mathrm{Im}(\cdot)$ returns the imaginary part of a complex matrix. It follows from (4.127) that

$$\frac{d\boldsymbol{h}}{d\boldsymbol{\phi}} = 2\,\mathrm{Im}\left(\boldsymbol{F}^H \boldsymbol{\varphi} . * \overline{\boldsymbol{F}^H \,\mathrm{Diag}\,(\boldsymbol{\varphi})}\right) \tag{4.128}$$

$$= 2\,\mathrm{Im}\left(\boldsymbol{F}^H \boldsymbol{\varphi} . * \boldsymbol{F}\,\mathrm{Diag}\,(\overline{\boldsymbol{\varphi}})\right). \tag{4.129}$$

From (4.119) and (4.129),

$$\nabla \mathrm{diag}\,(\boldsymbol{\Lambda}) = \nabla(\sqrt{N}\boldsymbol{F}\boldsymbol{h}) \tag{4.130}$$

$$= \sqrt{N}\boldsymbol{F}\frac{d\boldsymbol{h}}{d\boldsymbol{\phi}} \tag{4.131}$$

$$= 2\sqrt{N}\boldsymbol{F}\,\mathrm{Im}\left(\boldsymbol{F}^H \boldsymbol{\varphi} . * \boldsymbol{F}\,\mathrm{Diag}\,(\overline{\boldsymbol{\varphi}})\right) \tag{4.132}$$

$$= 2\sqrt{N}\boldsymbol{F}\,\mathrm{Im}\left(\mathrm{Diag}\left(\boldsymbol{F}^H \boldsymbol{\varphi}\right) \boldsymbol{F}\,\mathrm{Diag}\,(\overline{\boldsymbol{\varphi}})\right) \tag{4.133}$$

Now we multiply $\nabla \mathrm{diag}\,(\boldsymbol{\Lambda})$ to a real vector $\boldsymbol{p}$.

$$\nabla \mathrm{diag}\,(\boldsymbol{\Lambda})\,\boldsymbol{p} = 2\sqrt{N}\boldsymbol{F}\,\mathrm{Im}\left(\mathrm{Diag}\left(\boldsymbol{F}^H \boldsymbol{\varphi}\right) \boldsymbol{F}\,\mathrm{Diag}\,(\overline{\boldsymbol{\varphi}})\right)\boldsymbol{p} \tag{4.134}$$

$$= 2\sqrt{N}\boldsymbol{F}\,\mathrm{Im}\left(\mathrm{Diag}\left(\boldsymbol{F}^H \boldsymbol{\varphi}\right) \boldsymbol{F}\,\mathrm{Diag}\,(\overline{\boldsymbol{\varphi}})\,\boldsymbol{p}\right). \tag{4.135}$$

The above multiplication (4.135) can then be done from right to left: first compute $\mathrm{Diag}\,(\overline{\boldsymbol{\varphi}})\,\boldsymbol{p}$, then $\boldsymbol{F}\,\mathrm{Diag}\,(\overline{\boldsymbol{\varphi}})\,\boldsymbol{p}$ and so on. Each intermediate step returns a vector of the same size, thus the need for extra temporary memory is minimized. Also each intermediate step involves only a diagonal or Fourier matrix, so each step can be done very cheaply.

Now we multiply $(\nabla \text{diag}\,(\boldsymbol{\Lambda}))^T$ to a conjugate symmetric vector $\boldsymbol{p}$.

$$(\nabla \text{diag}\,(\boldsymbol{\Lambda}))^T \boldsymbol{p} = 2\sqrt{N}\,\text{Im}\left(\text{Diag}\,(\overline{\boldsymbol{\varphi}})\,\boldsymbol{F}\,\text{Diag}\left(\boldsymbol{F}^H \boldsymbol{\varphi}\right)\right)\boldsymbol{F}\boldsymbol{p} \tag{4.136}$$

$$= 2\sqrt{N}\,\text{Im}\left(\text{Diag}\,(\overline{\boldsymbol{\varphi}})\,\boldsymbol{F}\,\text{Diag}\left(\boldsymbol{F}^H \boldsymbol{\varphi}\right)\boldsymbol{F}\boldsymbol{p}\right) \tag{4.137}$$

Equation (4.137) uses the fact that $\boldsymbol{F}\boldsymbol{p}$ is real. Again, (4.137) can be done from right to left, with each intermediate step involving only a diagonal or Fourier matrix, and the result is a vector of the same size.

We have now finished the algorithms behind each individual step of the Gauss-Newton method. In the next section we show some experimental results on multi-frame blind deconvolution using algorithms described so far.

## 4.6 Experimental Results

We test the Gauss-Newton algorithm with variable projection on a satellite image (Figure 4.2). First we blur the satellite image by the PSFs of three pupil phase functions, and then deblur using different number of blurred images. We try two sets of pupil phase functions, one set gives only mild blurs, while the other gives more severe blurs. These test data were provided to us by Stuart Jefferies from the Institute of Astronomy, University of Hawaii. Our experiments show that using more blurred images can improve the deblurring result.

### 4.6.1 Removing Mild Blurs

In Figure 4.3, we show three pupil phase functions and their corresponding PSFs for our first test case. We blur the satellite image with the three PSFs to obtain three blurred images (left hand side of Figure 4.4). They

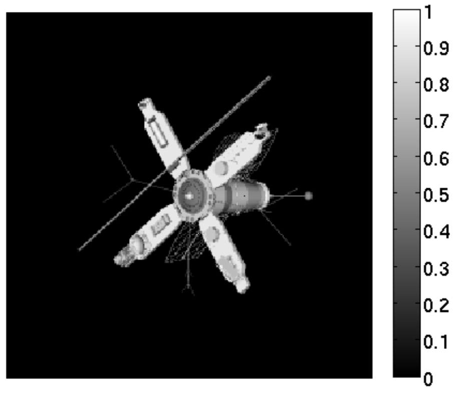


Figure 4.2: The original unblurred satellite image.

do not cause very severe blurs, but since we do not assume the blurs to be known in advance, the deblurring is a hard problem. Recalling our discussion in Section 4.5, we are minimizing over a space of dimension 12851 after variable projection.

The convergence of Newton methods depends on the proximity of the initial guess to the true solution. We generate the initial guess of the pupil phase functions by adding 50 % random noise to the exact true pupil phase functions. Methods to determine a good initial guess of the pupil phase functions are application dependent. One approach being investigated uses coarse gradient measurements of the phase, which can be obtained by many modern adaptive optics telescopes [3]. The deblurred images using these initial guesses are shown on the right side of Figure 4.4. One can see that the images are sharper but with pixel values lying in the wrong range.

We first deblur using only the first image. Algorithm 4.3 is used for the single image case. The result is shown in Figure 4.5. There are some artifacts around the edges and the range of pixel intensity is not correct. The original satellite has pixel values between 0 and 1, but the deblurred image has pixel values between about -0.2156 and 1.5638. This explains the

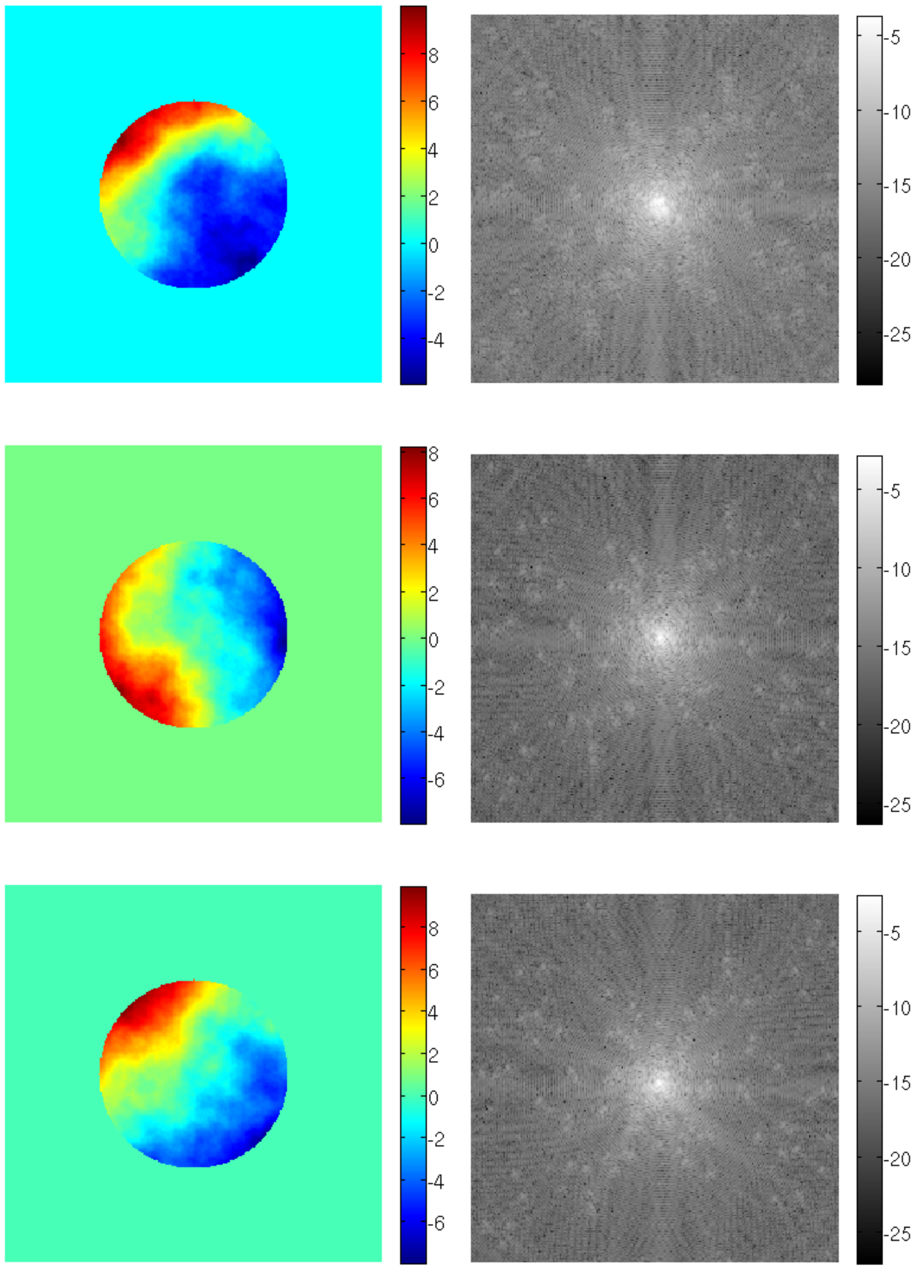


Figure 4.3: The pupil phase functions and their corresponding point spread functions giving mild blurs.

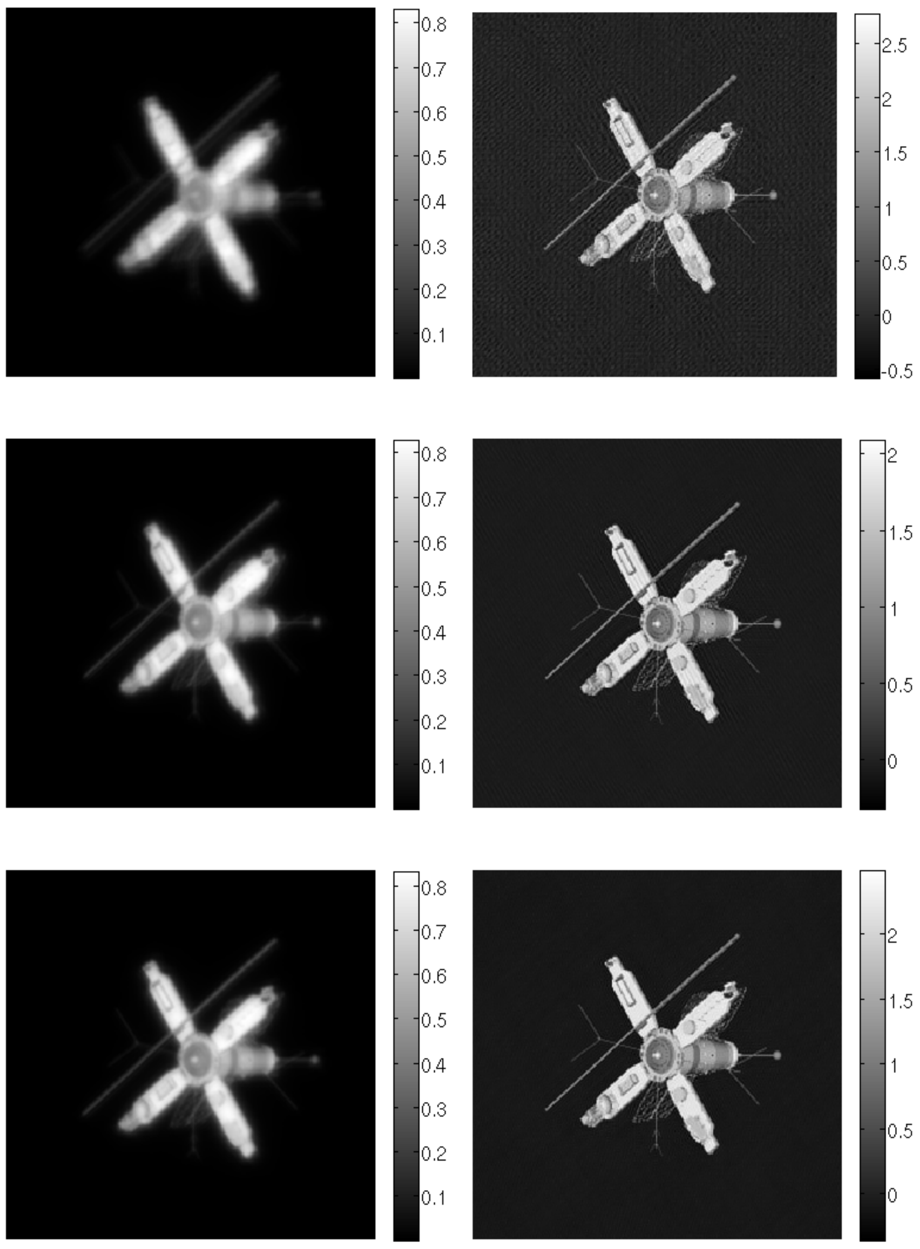


Figure 4.4: Left: Images obtained by blurring with the three point spread functions in Figure 4.3. Right: The deblurring results using the initial guess of the pupil phase functions.

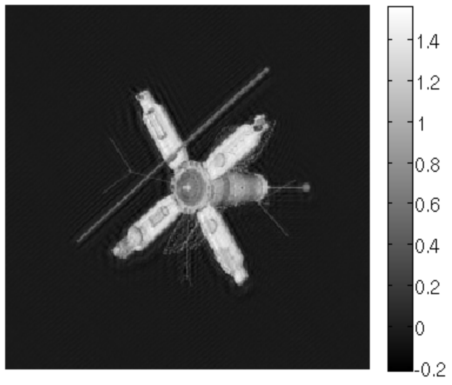


Figure 4.5: Deblurring result using only one image. Relative error=0.4760

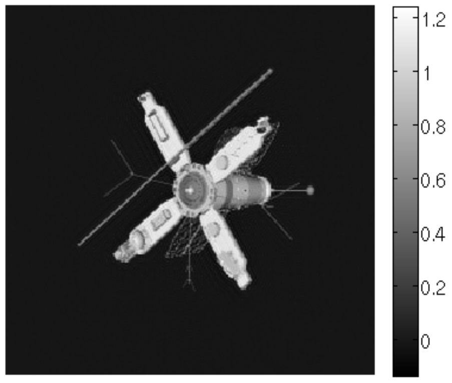


Figure 4.6: Deblurring result using two images. Relative error=0.2190

high relative error: 0.4760.

We get better result using two images. For this and the three-frame case, we utilize Algorithm 4.4. The deblurring result with the first two images is shown in Figure 4.6. The artifacts that appear in the single frame reconstruction are removed and the pixel values are in the right range, from about -0.1342 to about 1.2427. The resulting relative error is 0.2190.

With all three images, we obtain a further improved deblurred image (Figure 4.7). There are no visible artifacts and the pixel values fall in the

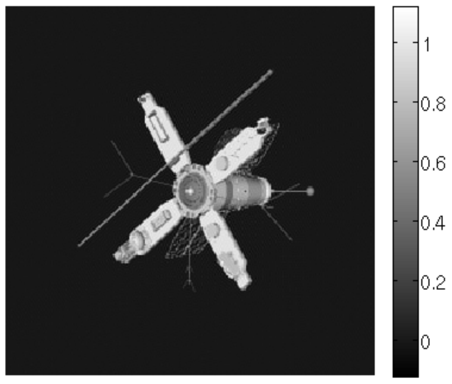


Figure 4.7: Deblurring result using three images. Relative error=0.1162

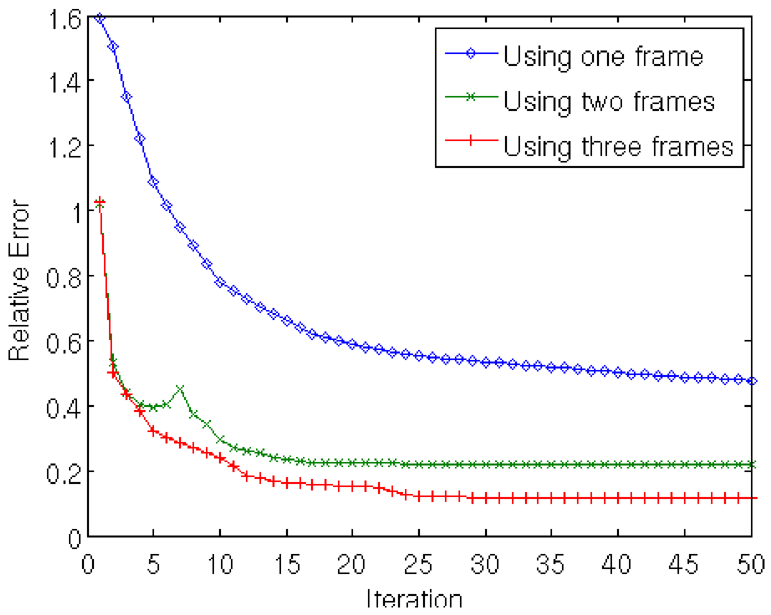


Figure 4.8: Plot of relative errors at each iteration using one, two and three frames for the blind deconvolution problem of the mild blur case.

correct range. The relative error, 0.1162, is smaller than those from using just one or two images.

In Figure 4.8, we plot the relative errors at each iteration when we deblur with one, two and three images. The plot clearly indicates that the relative error converges faster and to a lower value when more images are used.

### 4.6.2 Removing Severe Blurs

In this subsection, we perform similar tests with more severe blurs. The three pupil phase functions and their corresponding PSFs used in this experiment are shown in Figure 4.9. We see that the PSFs are much flatter than those in the previous experiment. After blurring the satellite image with the three PSFs, we have three blurred images (left hand side of Figure 4.10). We now try to deblur them. Our initial guess of pupil phase functions are obtained by adding 10 % random noise to the true pupil phase functions. The right column in Figure 4.10 shows the deblurred images using the initial guess. The initial deblurred images are sharper than the starting blurred images, but artifacts spread throughout the whole image and the pixel values lie in the wrong range.

Next, we deblur the blurred images using one, two and three frames and compare the results. With just one frame, we do not get back a clear image (Figure 4.11), although most artifacts are gone and pixel values are in the correct range. The relative error for this case is 0.4645. If we use two frames, a clearer image (Figure 4.12) is obtained, but we still have some artifacts. The relative error has improved by a little to 0.3104. Just like the previous experiment, further improvement is observed when we use three frames. The deblurred image (Figure 4.13) is very sharp and the relative error is now just 0.1222.

In Figure 4.14 we plot the relative errors at each iteration when using one, two and three frames. When only one frame is used, we start at a high (over 2) relative error; while when two or three frames are used, we obtain much smaller relative errors after the first iteration. The two frame

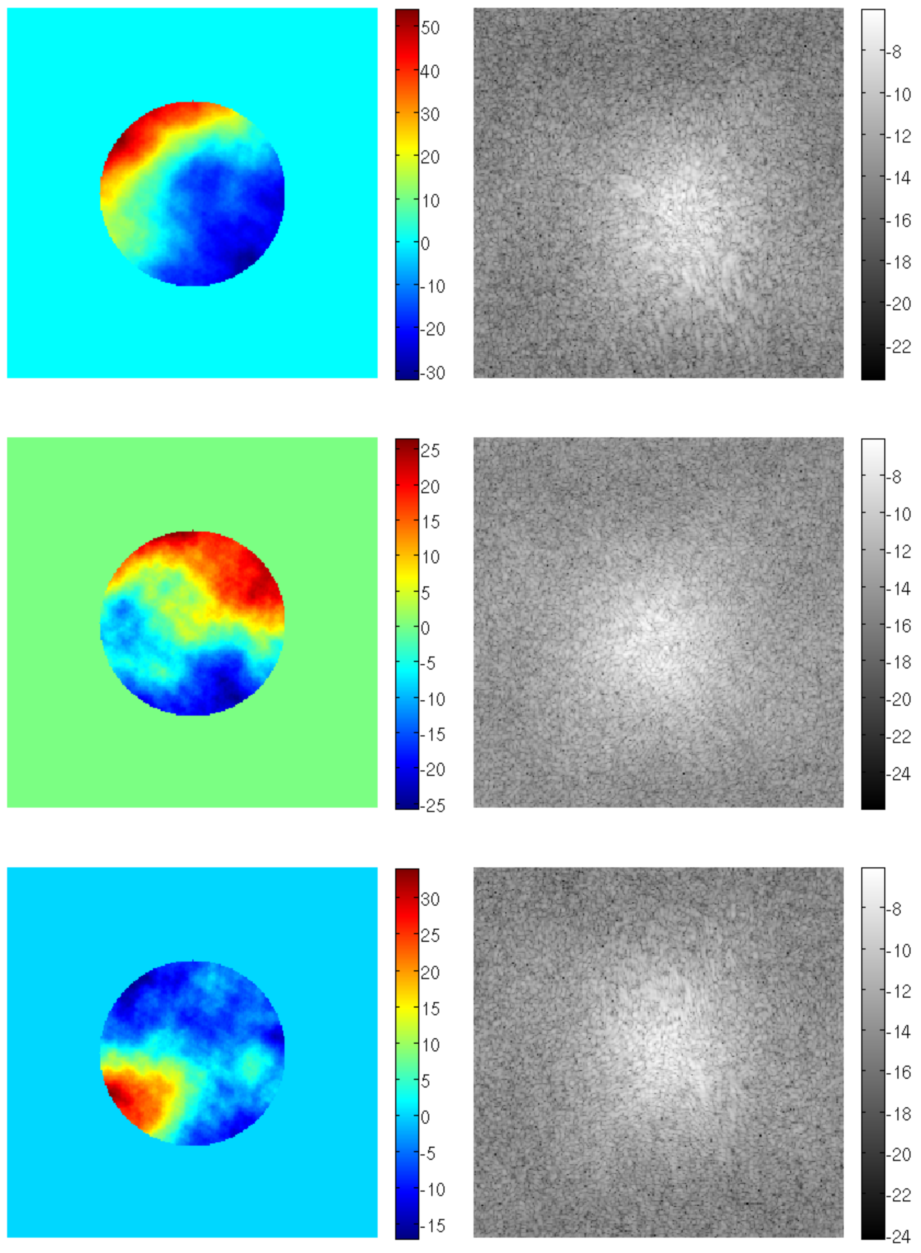


Figure 4.9: The pupil phase functions and their corresponding point spread functions giving severe blurs.

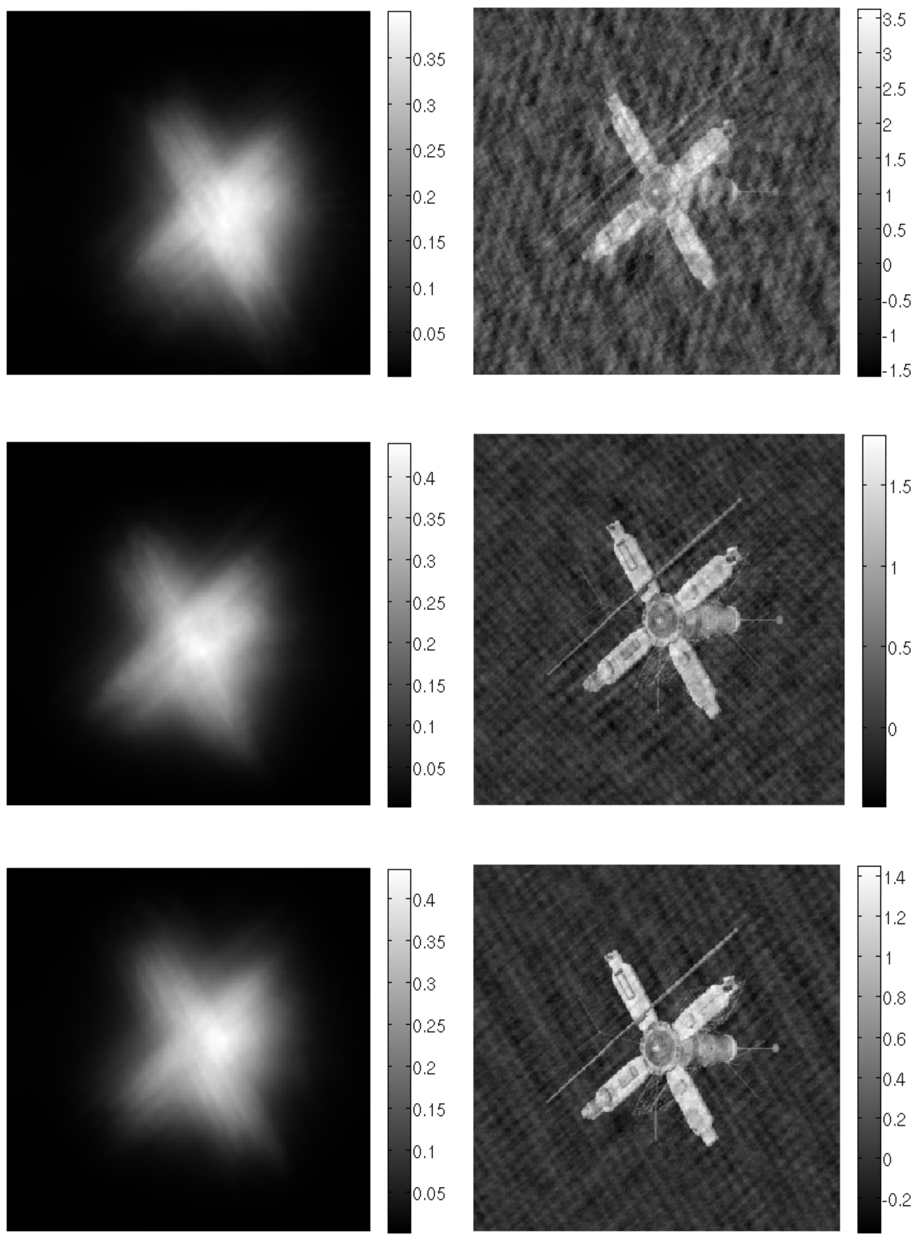


Figure 4.10: Left: Images obtained by blurring with the three point spread functions in Figure 4.9. Right: The deblurring results using the initial guess of the pupil phase functions.

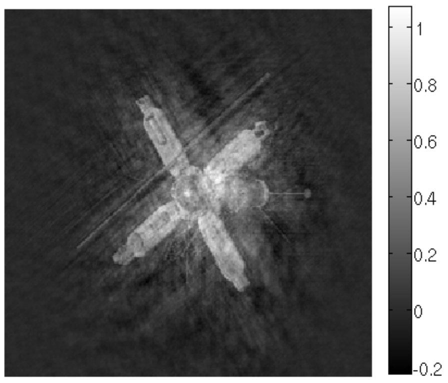


Figure 4.11: Deblurring result using only one image. Relative error=0.4645

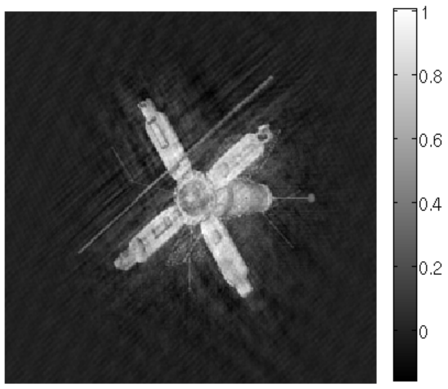


Figure 4.12: Deblurring result using two images. Relative error=0.3104

case relative error levels off to just over 0.3 but the three frame case relative error decreases to close to 0.1.

Our experimental results have illustrated the success of the variable projection method in reducing the number of variables, and the effectiveness of the Gauss-Newton method in minimizing the projected objective function. The results also show that using multiple frames can significantly improve the blind deconvolution quality.

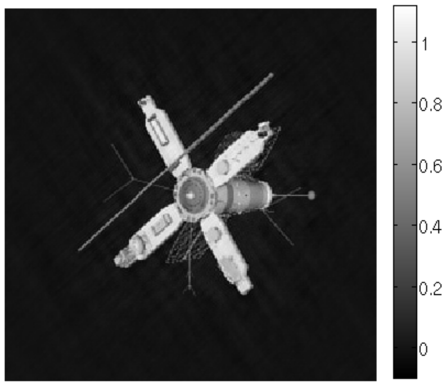


Figure 4.13: Deblurring result using three images. Relative error=0.1222

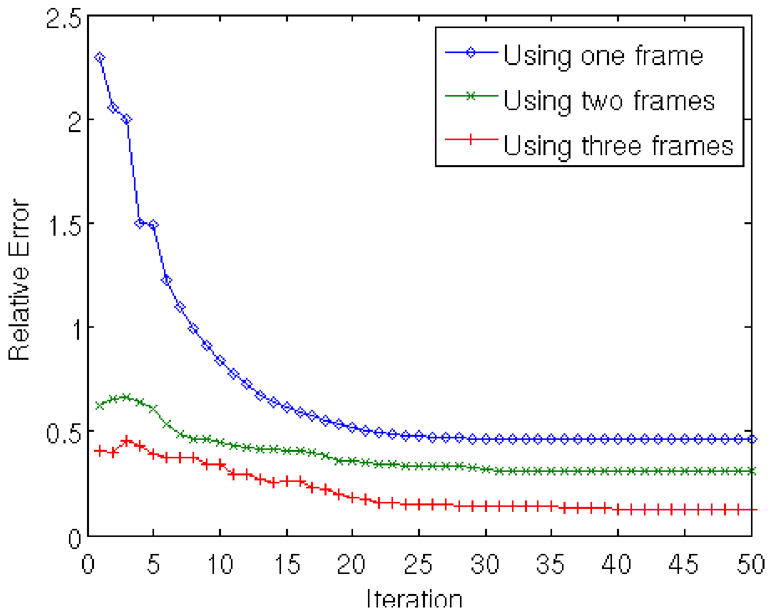


Figure 4.14: Plot of relative errors at each iteration using one, two and three frames for blind deconvolution problem of the severe blur case.

## 4.7 Conclusions for this Chapter

In this chapter, we investigated the blind deconvolution problem for images affected by pupil phase atmospheric blurs common in astronomical imaging. In many blind deconvolution problems, the PSFs are parametrized to reduce the number of variables. But unlike common blurs like Gaussian, motion and out-of-focus blurs, the number of parameters of an atmospheric blur described by a pupil phase function is of the order of 10000 for images of size $256 \times 256$. Together with the linear variables (the unknown image), the total number of variables is more than 70000. The variable projection approach eliminates the linear terms, but the reduced cost functional still requires optimizing over the nonlinear terms defined by the pupil phase function. Compared with many problems in the literature solved by variable projection, our problem has significantly more variables, even after the projection.

We posed blind deconvolution as a nonlinear minimization problem, which we solved using the Gauss-Newton method with a conjugate gradient inner solver. With the assumption of periodic boundary conditions, which is valid for astronomical imaging, the convolution matrix has an efficient spectral decomposition. Together with the formula of the pupil phase atmospheric blur and two lemmas on conjugate symmetric vectors, it was shown that multiplication with the Jacobian matrix can be done through a series of simple diagonal and Fourier matrix multiplications. For images with $N$ pixels, this significantly reduces the operation cost from $O(N^3)$ (since a 3D tensor is involved) down to $O(N \log N)$. Therefore we use the full Jacobian matrix, instead of its approximation as in many other problems.

The multiframe case, which uses more than one observed image of the same object, often produces better deblurring results for blind deconvolution. This was illustrated in our numerical experiments. By decoupling the frames into separate deblurring problems, with the constraint that the deblurred images should be close to each other, we obtained a sparse, block structured Jacobian matrix. The block structure significantly reduces the storage requirements, and allows for parallel implementation of the Jacobian matrix-vector multiplications.

# Chapter 5

# Conclusions

This dissertation focused on several practical issues in image deblurring. After a literature survey on image deblurring research, we pointed out the drawbacks in many deblurring methods in that they make some unrealistic assumptions on the problem. One common assumption is that the digital image extends out of the boundary in a wrap-around fashion. This assumption makes sense when the background is mostly dark like in astronomical imaging, but for other imaging situations, this is far from reality. Another common assumption is that precise knowledge of the blur is available. In some cases, the blur can be measured through instrument calibration or from images of point source objects. However for imaging environments that are constantly changing, for example blurs due to atmospheric air movement in astronomical imaging, it is very hard to obtain the blur exactly. Finally, research in image deblurring typically covers only theoretical issues, models and algorithms, and rarely provides efficient and robust software. This dissertation has addressed all of these issues.

In Chapter 2.2, we explained the importance of boundary conditions to the deblurring problem. Expressing the deblurring problem as a linear deconvolution problem, we compared the different structure of the convolution matrix when different boundary conditions are used. We also showed an example to display the different kind of padding with different boundary conditions. This example clearly shows that the classical zero, periodic, reflective and anti-reflective boundary conditions fail to give a realistic extension of the image. They result in jumps in intensity and sharp turns of edge directions across the image boundary. These problems do not appear when we use our new synthetic boundary conditions.

Synthetic boundary, as its name implies, synthesize the boundary conditions from the given image. Hence the boundary conditions are tailored to the particular image we are working on, unlike the other boundary conditions which do not adapt to the image in hand. We have given an algorithm to obtain the synthetic boundary conditions from an image.

Except for periodic and reflective boundary conditions, the convolution matrices for the other boundary conditions do not have a simple and efficient spectral decomposition. Thus, direct methods cannot be used and we have to resort to iterative methods. To speed up the convergence of the iterative methods when synthetic boundary conditions are used, we have designed a new preconditioner. We notice the similarity between synthetic and reflective boundary conditions, so we base our preconditioner on the DCT preconditioner. But since the convolution matrix is ill-conditioned, we incorporate Tikhonov regularization to obtain a regularized DCT preconditioner

We gave extensive experimental results to show the effectiveness of synthetic boundary conditions, especially in removing motion blurs. The experiments also showed that the regularized DCT preconditioning gives a significant improvement to the deblurring results and reduces the number of iterations.

The next part of the dissertation is about two new software packages, PYRET and PARRET for image deblurring. We started by introducing the two main types of deblurring algorithms: direct methods and iterative methods. Direct methods are used when a spectral decomposition of the convolution matrix can be acquired with low computational cost, otherwise, we need to use iterative methods. Since in many iterative methods the most expensive steps are the matrix-vector multiplications with the convolution matrix and its transpose, it is essential to find an efficient way to perform these multiplications. We decompose the multiplication process into three smaller steps, each of which can be done quickly. Combining the corresponding transposes of these small steps, we get the multiplication by the transpose of the convolution matrix.

Then we moved on to the implementation details. We have explained the reasons for choosing Python as our implementation language. Using an object-oriented paradigm we created a Python package PYRET. We abstract out the convolution matrix by building a special psfMatrix class, so we can perform matrix operations with efficient algorithms. With operator overloading, psfMatrix objects can be used in functions originally implemented for ordinary NumPy array objects. This facilitates the reuse of program codes. Similarly, we have a matrix class for the preconditioner matrix. We also have some other helper functions useful for deblurring experiments.

To make it easy for users to try out PYRET, we have supplied a web interface with the popular Python web framework Pylons. This ease of extension is a strength in Python over other commercial mathematical languages.

We noticed the potential of faster deblurring if we use parallel computing, so we decided on a parallel implementation and the end product is another package we developed, called PARRET. In the supercomputing community, GPU programming is getting the attention of many scientists, and it is an economical way to obtain parallel computing power, so we pick GPUs as our implementation platform. More specifically, we use NVIDIA CUDA architecture, since it comes with the most mature GPU programming API. The workflow of a CUDA program consists of the steps of moving data to the GPU memory, processing the data in parallel by the many cores on the GPU, and moving the results back to the main memory.

The Python wrapper PyCUDA makes all these steps much easier for programmers. PyCUDA exposes the CUDA API in Python and allows interactive experiments in various Python shells. Since PyCUDA did not provide all the functionalities required for deblurring algorithms, we had to make a few improvements to it. First we provided a wrapper for the CUBLAS and CUFFT libraries to work with PyCUDA objects. Secondly we added the capability of complex arithmetic to PyCUDA.

Just like in PYRET, we create a psfMatrix class to represent the convolution matrix. And for the iterative linear solvers, we follow the design of SciPy sparse linear algebra package for easy future extension. Benchmarking results show that this GPU implementation gives orders of magnitude speedup over CPU implementation.

The final part of this dissertation deals with the case when the blurs are not known in advance. For this part, we work on the removal of pupil phase atmospheric blurs common in astronomical imaging. As discussed above, for astronomical imaging, we can assume periodic boundary conditions and thus easy spectral decomposition of the convolution matrix is available. But since the blurs are not known, and thus the convolution matrix is not known, we cannot use direct methods as in the non-blind case.

We formulate the blind deconvolution problem as a nonlinear minimization problem. The objective function depends nonlinearly on the pupil phase function, which determines the blur, and linearly on the unknown true image. We eliminate the linear variable of the objective function using a technique called variable projection. The reduced function only depends on the pupil phase functions. Unlike other problems treated by variable projection, the reduced function still depends on more than 10000 variables, thus every step in the optimization process is expensive, since large matrices and tensors are involved. With careful mathematical manipulation, we decompose the involved Jacobian matrix into a series of diagonal and Fourier matrices, hence inexpensive multiplication with the Jacobian matrix and its transpose is possible.

To further improve the deblurring quality, we use more than one blurred image from the same object. This is called multi-frame blind deconvolution. With a decoupling approach, the large Jacobian matrix is sparse with block structure, simplifying the implementation and with the potential for parallelization. We test our decoupled multi-frame blind deconvolution approach on a mild blur case and a severe blur case. Both cases show decreases in relative errors when more images are used. The improvement is especially prominent in the severe blur case.

To conclude, this dissertation has investigated many levels of practical image deblurring. On the theoretical level, we have devised new synthetic boundary conditions, a new regularized preconditioner and a new formulation for the blind deconvolution problem. On the algorithmic level, we show efficient algorithms for multiplication with the convolution matrices, preconditioner matrices and Jacobian matrices. On the implementation level, we discuss the choice of programming language and parallel architecture, design of the software packages, incorporation of different libraries and the development of a web user interface. We hope this dissertation can provide useful resources for future research in image deblurring.

# Appendix

## 6.1 Proof of Lemma 4.2 in Section 4.4

**Lemma 4.2.** *If $\boldsymbol{u}$ and $\boldsymbol{v}$ are two conjugate symmetric vectors, then*

$$\overline{\boldsymbol{u}}^T \boldsymbol{v} = \boldsymbol{u}^T \overline{\boldsymbol{v}}. \tag{6.1}$$

*Proof.* Let $n$ be the length of $\boldsymbol{u}$ and $\boldsymbol{v}$.

$$\begin{aligned}
\overline{\boldsymbol{u}}^T \boldsymbol{v} &= \sum_{k=1}^{n} \overline{\boldsymbol{u}_k} \boldsymbol{v}_k && (6.2)\\
&= \overline{\boldsymbol{u}_1} \boldsymbol{v}_1 + \sum_{k=2}^{n} \overline{\boldsymbol{u}_k} \boldsymbol{v}_k && (6.3)\\
&= \boldsymbol{u}_1 \boldsymbol{v}_1 + \sum_{k=2}^{n} \boldsymbol{u}_{n-k+2} \overline{\boldsymbol{v}_{n-k+2}} && (6.4)\\
&= \boldsymbol{u}_1 \overline{\boldsymbol{v}_1} + \sum_{l=2}^{n} \boldsymbol{u}_l \overline{\boldsymbol{v}_l} \qquad \text{(letting } l = n-k+2\text{)} && (6.5)\\
&= \sum_{l=1}^{n} \boldsymbol{u}_l \overline{\boldsymbol{v}_l} && (6.6)\\
&= \boldsymbol{u}^T \overline{\boldsymbol{v}}. && (6.7)
\end{aligned}$$

□

## 6.2 Proof of Lemma 4.3 in Section 4.4

**Lemma 4.3.** *If $\boldsymbol{u}$ and $\boldsymbol{v}$ are two conjugate symmetric vectors, then*

$$Re\,(\boldsymbol{u})^T \boldsymbol{v} = \boldsymbol{u}^T \, Re\,(\boldsymbol{v})\,. \tag{6.8}$$

*Proof.* Let $n$ be the length of $\boldsymbol{u}$ and $\boldsymbol{v}$.

$$\begin{aligned}
\operatorname{Re}(\boldsymbol{u})^T \boldsymbol{v} &= \frac{1}{2}(\boldsymbol{u} + \overline{\boldsymbol{u}})^T \boldsymbol{v} && (6.9)\\
&= \frac{1}{2}\left(\boldsymbol{u}^T\boldsymbol{v} + \overline{\boldsymbol{u}}^T\boldsymbol{v}\right) && (6.10)\\
&= \frac{1}{2}\left(\boldsymbol{u}^T\boldsymbol{v} + \boldsymbol{u}^T\overline{\boldsymbol{v}}\right) \quad \text{(by Lemma 4.2)} && (6.11)\\
&= \boldsymbol{u}^T\left(\frac{1}{2}(\boldsymbol{v} + \overline{\boldsymbol{v}})\right) && (6.12)\\
&= \boldsymbol{u}^T \operatorname{Re}(\boldsymbol{v}). && (6.13)
\end{aligned}$$

□

## 6.3 Derivation of (4.20) in Section 4.3

First we determine a formula for $\nabla|\boldsymbol{\Lambda}|^2$.

$$\begin{aligned}
\nabla|\boldsymbol{\Lambda}|^2 &= \nabla(\overline{\boldsymbol{\Lambda}}\boldsymbol{\Lambda}) && (6.14)\\
&= \nabla\overline{\boldsymbol{\Lambda}}\boldsymbol{\Lambda} + \overline{\boldsymbol{\Lambda}}\nabla\boldsymbol{\Lambda} && (6.15)\\
&= \overline{\nabla\boldsymbol{\Lambda}}\boldsymbol{\Lambda} + \overline{\boldsymbol{\Lambda}}\nabla\boldsymbol{\Lambda} && (6.16)\\
&= \boldsymbol{\Lambda}\overline{\nabla\boldsymbol{\Lambda}} + \overline{\boldsymbol{\Lambda}}\nabla\boldsymbol{\Lambda} && (6.17)\\
&= 2\operatorname{Re}\left(\overline{\boldsymbol{\Lambda}}\nabla\boldsymbol{\Lambda}\right). && (6.18)
\end{aligned}$$

We also need to use the following identity on derivatives of matrix inverses. Assume that $\boldsymbol{A}$ is a matrix in which each entry is a function of a scalar $x$. Then

$$\frac{d\boldsymbol{A}^{-1}}{dx} = -\boldsymbol{A}^{-1}\frac{d\boldsymbol{A}}{dx}\boldsymbol{A}^{-1}. \tag{6.19}$$

Taking derivatives with respect to an entry $\phi_k$ of $\boldsymbol{\phi}$, we get

$$\frac{d}{d\phi_k}\left(\frac{1}{|\mathbf{\Lambda}|^2+\alpha^2}\right) = -\frac{d}{d\phi_k}(|\mathbf{\Lambda}|^2+\alpha^2)^{-1} \tag{6.20}$$

$$= -(|\mathbf{\Lambda}|^2+\alpha^2)^{-1}\frac{d}{d\phi_k}\left(|\mathbf{\Lambda}|^2+\alpha^2\right)(|\mathbf{\Lambda}|^2+\alpha^2)^{-1} \tag{6.21}$$

$$= -(|\mathbf{\Lambda}|^2+\alpha^2)^{-2}\frac{d}{d\phi_k}\left(|\mathbf{\Lambda}|^2+\alpha^2\right) \tag{6.22}$$

$$= -(|\mathbf{\Lambda}|^2+\alpha^2)^{-2}\frac{d}{d\phi_k}(|\mathbf{\Lambda}|^2) \tag{6.23}$$

In (6.22), we use the fact that diagonal matrices commute. Finally,

$$\nabla\left(\frac{1}{|\mathbf{\Lambda}|^2+\alpha^2}\right) = -(|\mathbf{\Lambda}|^2+\alpha^2)^{-2}\nabla(|\mathbf{\Lambda}|^2) \tag{6.24}$$

$$= -(|\mathbf{\Lambda}|^2+\alpha^2)^{-2}\left(2\operatorname{Re}\left(\overline{\mathbf{\Lambda}}\nabla\mathbf{\Lambda}\right)\right) \tag{6.25}$$

$$= -2\frac{\operatorname{Re}\left(\overline{\mathbf{\Lambda}}\nabla\mathbf{\Lambda}\right)}{\left(|\mathbf{\Lambda}|^2+\alpha^2\right)^2}. \tag{6.26}$$

## 6.4 Derivation of (4.74) in Section 4.4

Using the product rule and (6.25), we have

$$\nabla\left(\frac{\overline{\mathbf{\Lambda}}}{|\mathbf{\Lambda}|^2+\alpha^2}\right) \tag{6.27}$$

$$= \nabla\left(\frac{1}{|\mathbf{\Lambda}|^2+\alpha^2}\overline{\mathbf{\Lambda}}\right) \tag{6.28}$$

$$= \nabla\left(\frac{1}{|\mathbf{\Lambda}|^2+\alpha^2}\right)\overline{\mathbf{\Lambda}} + \frac{1}{|\mathbf{\Lambda}|^2+\alpha^2}\overline{\nabla\mathbf{\Lambda}} \tag{6.29}$$

$$= -\left(|\mathbf{\Lambda}|^2+\alpha^2\right)^{-2}\left(2\,\mathrm{Re}\left(\overline{\mathbf{\Lambda}}\nabla\mathbf{\Lambda}\right)\right)\overline{\mathbf{\Lambda}} + \left(|\mathbf{\Lambda}|^2+\alpha^2\right)^{-1}\overline{\nabla\mathbf{\Lambda}} \tag{6.30}$$

$$= \left(|\mathbf{\Lambda}|^2+\alpha^2\right)^{-2}\left(-(\overline{\mathbf{\Lambda}}\nabla\mathbf{\Lambda}+\mathbf{\Lambda}\overline{\nabla\mathbf{\Lambda}})\overline{\mathbf{\Lambda}} + \left(|\mathbf{\Lambda}|^2+\alpha^2\right)\overline{\nabla\mathbf{\Lambda}}\right) \tag{6.31}$$

$$= \left(|\mathbf{\Lambda}|^2+\alpha^2\right)^{-2}\left(-\overline{\mathbf{\Lambda}}^2\nabla\mathbf{\Lambda} - |\mathbf{\Lambda}|^2\,\overline{\nabla\mathbf{\Lambda}} + |\mathbf{\Lambda}|^2\,\overline{\nabla\mathbf{\Lambda}} + \alpha^2\overline{\nabla\mathbf{\Lambda}}\right) \tag{6.32}$$

$$= \left(|\mathbf{\Lambda}|^2+\alpha^2\right)^{-2}\left(-\overline{\mathbf{\Lambda}}^2\nabla\mathbf{\Lambda} + \alpha^2\overline{\nabla\mathbf{\Lambda}}\right) \tag{6.33}$$

$$= \frac{\alpha^2\overline{\nabla\mathbf{\Lambda}} - \overline{\mathbf{\Lambda}}^2\nabla\mathbf{\Lambda}}{(|\mathbf{\Lambda}|^2+\alpha^2)^2}. \tag{6.34}$$

# Bibliography

[1] H. Andrews and B. Hunt.
*Digital Image Restoration.*
Prentice Hall, 1977.

[2] A. Aricò, M. Donatelli, and S. Serra-Capizzano.
Spectral analysis of the anti-reflective algebra.
*Linear Algebra and its Applications*, vol. 428, pp. 657–675, 2008.

[3] J. Bardsley, S. Knepper, and J. Nagy.
Structured linear algebra problems in adaptive optics imaging.
*to appear in Advances in Computational Mathematics*, 2010.

[4] J. M. Bardsley and C. R. Vogel.
A nonnegatively constrained convex programming method for image reconstruction.
*SIAM Journal on Scientific Computing*, vol. 25(4), pp. 1326–1343, 2003.

[5] M. Bertalmio, G. Sapiro, V. Caselles, and C. Ballester.
Image inpainting.
In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 417–424. ACM Press/Addison-Wesley Publishing Co. New York, NY, USA, 2000.

[6] Å. Björck and T. Elfving.

Accelerated projection methods for computing pseudoinverse solutions of systems of linear equations.
*BIT Numerical Mathematics*, vol. 19(2), pp. 145–163, 1979.

[7] D. Calvetti and E. Somersalo.
Bayesian image deblurring and boundary effects.
In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, vol. 5910, pp. 281–289. 2005.

[8] R. Chan, T. Chan, L. Shen, and Z. Shen.
Wavelet deblurring algorithms for spatially varying blur from high-resolution image reconstruction.
*Linear Algebra and its Applications*, vol. 366, pp. 139–155, 2003.

[9] R. H. Chan and X. Q. Jin.
*An Introduction to Iterative Toeplitz Solvers.*
SIAM, Philadelphia, 2007.

[10] T. Chan and J. Shen.
*Image Processing and Analysis: Variational, PDE, Wavelet, and Stochastic Methods.*
SIAM, Philadelphia, PA, USA, 2005.

[11] T. Chan and C. Wong.
Total variation blind deconvolution.
*IEEE Transactions on Image Processing*, vol. 7(3), pp. 370–375, 1998.

[12] K. Chen.
*Matrix Preconditioning Techniques and Applications.*
Cambridge Univ Pr, 2005.

[13] J. Chung.

*Numerical approaches for large-scale ill-posed inverse problems.*
Ph.D. thesis, Emory University, 2009.

[14] J. Chung, J. Nagy, and D. O'Leary.
A weighted-GCV method for Lanczos-hybrid regularization.
*Electronic Transactions on Numerical Analysis*, vol. 28, pp. 149–167, 2008.

[15] J. Cooley and J. Tukey.
An algorithm for the machine calculation of complex Fourier series.
*Mathematics of Computation*, vol. 19(90), pp. 297–301, 1965.

[16] P. J. Davis.
*Circulant Matrices.*
Wiley, New York, 1979.

[17] F. Di Benedetto, C. Estatico, and S. Serra-Capizzano.
Superoptimal preconditioned conjugate gradient iteration for image deblurring.
*SIAM Journal on Scientific Computing*, vol. 26(3), pp. 1012–1035, 2005.

[18] M. Donatelli, C. Estatico, A. Martinelli, and S. Serra-Capizzano.
Improved image deblurring with anti-reflective boundary conditions and re-blurring.
*Inverse Problems*, vol. 22, pp. 2035–2053, 2006.

[19] M. Donatelli, C. Estatico, J. Nagy, L. Perrone, and S. Serra-Capizzano.
Anti-reflective boundary conditions and fast 2D deblurring models.
In *Proceeding to SPIEs 48th Annual Meeting, San Diego, CA USA, F. Luk Ed*, vol. 5205, pp. 380–389. 2003.

[20] J. Dongarra, I. Duff, H. van der Vorst, and D. Sorensen.

*Numerical Linear Algebra for High-performance Computers.*
SIAM, 1998.

[21] D. Donoho.
Nonlinear solution of linear inverse problems by wavelet–vaguelette decomposition.
*Applied and Computational Harmonic Analysis*, vol. 2(2), pp. 101–126, 1995.

[22] L. A. Drummond, V. Galiano, V. Migallón, and J. Penadés.
PyACTS: a high-level framework for fast development of high performance applications.
In *Proceedings from Seventh International Meeting on High Performance Computing for Computational Science - VECPAR'06*, pp. 373–378. Rio de Janeiro, Brazil, 2006.

[23] A. A. Efros and W. T. Freeman.
Image quilting for texture synthesis and transfer.
In *Proceedings of SIGGRAPH 2001*, pp. 341–346. Los Angeles, CA, 2001.

[24] A. A. Efros and T. K. Leung.
Texture synthesis by non-parametric sampling.
In *International Conference on Computer Vision*, pp. 1033–1038. 1999.

[25] H. W. Engl, M. Hanke, and A. Neubauer.
*Regularization of Inverse Problems.*
Kluwer Academic Publishers, Dordrecht, 2000.

[26] M. Fadili and J. Starck.
Sparse representation-based image deconvolution by iterative threshold-

ing.
In *Astronomical Data Analysis ADA06, Marseille, France*. 2006.

[27] M. Frigo and S. Johnson.
FFTW: an adaptive software architecture for the FFT.
In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, 1998*, pp. 1381–1384. 1998.

[28] J. Gardner.
*The Definitive Guide to Pylons*.
Springer, 2008.

[29] G. Golub, M. Heath, and G. Wahba.
Generalized cross-validation as a method for choosing a good ridge parameter.
*Technometrics*, vol. 21(2), pp. 215–223, 1979.

[30] G. Golub and V. Pereyra.
The differentiation of pseudo-inverses and nonlinear least squares problems whose variables separate.
*SIAM Journal on Numerical Analysis*, vol. 10(2), pp. 413–432, 1973.

[31] G. Golub and V. Pereyra.
Separable nonlinear least squares: the variable projection method and its applications.
*Inverse Problems*, vol. 19, pp. R1–R26, 2003.

[32] R. Gonzalez and R. Woods.
*Digital Image Processing*.
Prentice-Hall, Englewood Cliffs, NJ, 2002.

[33] C. W. Groetsch.
*The Theory of Tikhonov Regularization for Fredholm Integral Equations*

*of the First Kind.*
Pitman, Boston, 1984.

[34] M. Hanke.
*Conjugate Gradient Type Methods for Ill-posed Problems.*
Chapman & Hall/CRC, 1995.

[35] M. Hanke, J. Nagy, and R. Plemmons.
Preconditioned iterative regularization methods for ill-posed problems.
In R. V. L. Reichel, A. Ruttan, editor, *Numerical Linear Algebra and Scientific Computing*, pp. 141–163. de Gruyter, Berlin, Germany, 1993.

[36] P. C. Hansen.
*Rank-Deficient and Discrete Ill-Posed Problems: Numerical Aspects of Linear Inversion.*
SIAM, Philadelphia, 1998.

[37] P. C. Hansen, J. G. Nagy, and D. P. O'Leary.
*Deblurring Images: Matrices, Spectra, and Filtering.*
SIAM, Philadelphia, 2006.

[38] S. Haykin.
*Adaptive Filter Theory.*
Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1986.

[39] L. Kaufman.
A variable projection method for solving separable nonlinear least squares problems.
*BIT Numerical Mathematics*, vol. 15(1), pp. 49–57, 1975.

[40] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih.
PyCUDA: GPU Run-Time Code Generation for High-Performance

Computing.
*Arxiv preprint arXiv:0911.3456*, 2009.

[41] A. Klöeckner.
PyCUDA, 2009.
URL `http://mathema.tician.de/software/pycuda`

[42] R. L. Lagendijk and J. Biemond.
*Iterative Identification and Restoration of Images.*
Kluwer Academic Publishers, Boston/Dordrecht/London, 1991.

[43] L. B. Lucy.
An iterative technique for the rectification of observed distributions.
*The Astronomical Journal*, vol. 79(6), pp. 745–754, 1974.

[44] J. Nagy, K. Palmer, and L. Perrone.
Iterative methods for image deblurring: a Matlab object-oriented approach.
*Numerical Algorithms*, vol. 36(1), pp. 73–93, 2004.

[45] J. G. Nagy, K. Palmer, and L. Perrone.
RestoreTools: an object oriented Matlab package for image restoration.
URL `http://www.mathcs.emory.edu/~nagy`

[46] J. G. Nagy and Z. Strakoš.
Enforcing nonnegativity in image reconstruction algorithms.
*Mathematical Modeling, Estimation, and Imaging*, vol. 4121, pp. 182–190, 2000.

[47] M. K. Ng.
*Iterative Methods for Toeplitz Systems.*
Oxford University Press, Oxford, UK, 2004.

[48] M. K. Ng, R. H. Chan, and W. C. Tang.
A fast algorithm for deblurring models with Neumann boundary conditions.
*SIAM Journal on Scientific Computing*, vol. 21, pp. 851–866, 1999.

[49] NVIDIA.
*NVIDIA CUDA C Programming Best Practices Guide CUDA Toolkit 2.3*, 2009.

[50] NVIDIA.
*NVIDIA CUDA Programming Guide Version 2.3*, 2009.

[51] NVIDIA.
*NVIDIA CUDA Reference Manual Version 2.3*, 2009.

[52] T. E. Oliphant.
Guide to NumPy, 2006.

[53] M. R. Osborne.
Some special nonlinear least squares problems.
*SIAM Journal on Numerical Analysis*, vol. 12(4), pp. 571–592, 1975.

[54] M. R. Osborne.
Separable least squares, variable projection, and the Gauss-Newton algorithm.
*Electronic Transactions on Numerical Analysis*, vol. 28, pp. 1–15, 2007.

[55] C. C. Paige and M. A. Saunders.
Solution of sparse indefinite systems of linear equations.
*SIAM Journal on Numerical Analysis*, vol. 12(4), pp. 617–629, 1975.

[56] C. C. Paige and M. A. Saunders.
LSQR: an algorithm for sparse linear equations and sparse least squares.

*ACM Transactions on Mathematical Software (TOMS)*, vol. 8(1), pp. 43–71, 1982.

[57] F. Pérez and B. E. Granger.
IPython: a system for interactive scientific computing.
*Comput. Sci. Eng.*, vol. 9(3), pp. 21–29, 2007.
URL `http://ipython.scipy.org`

[58] W. H. Richardson.
Bayesian-based iterative method of image restoration.
*Journal of the Optical Society of America*, vol. 62, pp. 55–59, 1972.

[59] L. Rudin and S. Osher.
Total variation based image restoration with free local constraints.
In *IEEE International Conference Image Processing, 1994. Proceedings. ICIP-94.*, vol. 1. 1994.

[60] A. Ruhe and P. A. Wedin.
Algorithms for separable nonlinear least squares problems.
*SIAM Review*, vol. 22(3), pp. 318–337, 1980.

[61] Y. Saad.
*Iterative Methods for Sparse Linear Systems.*
SIAM, 2003.

[62] M. Sala, W. F. Spotz, and M. A. Heroux.
PyTrilinos: High-performance distributed-memory solvers for Python.
*ACM Transactions on Mathematical Software*, vol. 34(2), pp. 7:1–7:33, 2008.

[63] S. Serra-Capizzano.
A note on antireflective boundary conditions and fast deblurring models.

*SIAM J. Sci. Comput.*, vol. 25(4), pp. 1307–1325, 2003. doi:http://dx.doi.org/10.1137/S1064827502410244.

[64] W. A. Stein et al.
*Sage Mathematics Software (Version 4.2.1).*
The Sage Development Team, 2009.
URL `http://www.sagemath.org`

[65] The SciPy Community.
Multi-dimensional image processing (scipy.ndimage) — SciPy v0.8.dev Reference Guide.
URL `http://docs.scipy.org/doc/scipy/reference/ndimage.html`

[66] C. Thompson and L. Shure.
Matlab image processing toolbox users guide.
*The MathWorks, Inc.*, 1995.

[67] C. F. Van Loan.
*Computational Frameworks for the Fast Fourier Transform.*
SIAM, Philadelphia, 1992.

[68] C. R. Vogel.
*Computational Methods for Inverse Problems.*
SIAM, Philadelphia, 2002.

[69] M. Welk, D. Theis, and J. Weickert.
Variational deblurring of images with uncertain and spatially variant blurs.
*Pattern Recognition*, pp. 485–492, 2005.

[70] Y.-W. Wen, M. K. Ng, and W.-K. Ching.
Iterative algorithms based on decoupling of deblurring and denoising for

image restoration.
*SIAM Journal on Scientific Computing*, vol. 30(5), pp. 2655–2674, 2008.

[71] P. Wendykier.
*High Performance Java Software for Image Processing.*
Ph.D. thesis, Emory University, 2009.

[72] Wolfram Research.
Image Processing and Analysis — Wolfram Mathematica 7 Documentation.
URL `http://reference.wolfram.com/mathematica/guide/ImageProcessing.html`