Distribution Agreement

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

Signature:

_____ _____

Piotr Wendykier Date

High Performance Java Software for Image Processing

By

Piotr Wendykier
Doctor of Philosophy

Mathematics

James G. Nagy
Advisor

Michele Benzi
Committee Member

Alessandro Veneziani
Committee Member

Accepted:

Lisa A. Tedesco, Ph.D.
Dean of the James T. Laney School of Graduate Studies

Date

High Performance Java Software for Image Processing

by

Piotr Wendykier
M.Sc., Adam Mickiewicz University, 2003

Advisor: James G. Nagy, PhD

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 of the degree of
Doctor of Philosophy
in Mathematics
2009

Abstract

High Performance Java Software for Image Processing
By Piotr Wendykier

Parallel computing has been used for scientific computing applications since the 1960s, when the first supercomputers were developed. However, only recently have these programming paradigms become useful for software running on desktop and notebook computers. In this dissertation we demonstrate the advantage of exploiting modern computer architectures in scientific computing with multithreaded programming in Java for applications in image processing. A significant contribution of this work is an open source, multithreaded high performance scientific computing Java library called Parallel Colt. In addition, on top of Parallel Colt, we have implemented six ImageJ plugins for deconvolution, super-resolution, fast Fourier transforms and image cropping. Hence, we are able to provide software to solve important problems in real image processing applications, and which can effectively make use of multi-core CPUs available on affordable desktop and notebook computers.

High Performance Java Software for Image Processing

by

Piotr Wendykier
M.Sc., Adam Mickiewicz University, 2003

Advisor: James G. Nagy, PhD

A dissertation submitted to the Faculty of the
James T. Laney School of Graduate Studies of Emory University
in partial fulfillment of the requirements of the degree of
Doctor of Philosophy
in Mathematics
2009

Acknowledgments

I would like to thank my advisor James Nagy for his wonderful support and guidance. This work would not be the same without his tireless dedication.

I am also thankful to my committee members Michele Benzi and Alessandro Veneziani for their time and insightful comments that allowed me to improve this thesis.

I wish to thank Julianne Chung, Tracy Faber, Sarah Knepper, Nivedita Raghunath and John Votaw, for their collaboration.

Last, but no least, I am very grateful to my wife Miroslawa Wendykier for many sacrifices she made while I was pursuing this work.

To my family

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Image processing is a general term used to refer to computational methods that manipulate, modify, or analyze images. The input to the computational method is usually an image (e.g. photographs or video frames) and the output can either be an image or a set of characteristics or parameters related to the image. Typical image processing operations include: geometric transformations, interpolation, super-resolution, image restoration, image registration, image segmentation and image recognition. In this thesis, we develop algorithms and high-performance computing software for image processing, with a particular focus on geometric transformations, interpolation, super-resolution and image restoration.

Due to recent improvements in personal computer architecture, it is possible to efficiently manipulate large digital images on commodity hardware. Since practically all modern personal computers are equipped with powerful multi-core CPUs and GPUs, there is an urgent need to provide image processing libraries and application programming interfaces (APIs) that are aware of that architecture and are able to fully utilize it.

Until relatively recently, improvements in CPU performance have been achieved by increasing the clock speed, execution optimization, and by maximizing the size of on-chip cache. The *clock race* ended in 2003, when all chip manufacturers reached hard physical limits: increasing heat generation and power consumption, lack of suitable cooling hardware, current leakage problems, and increasing length of wire interconnects. October 2001 marks the beginning of a new era in CPU manufacturing when IBM (Armonk, NY) released the POWER4 microprocessor, the world's first multi-core processor. Since then, all new processors have been designed to consist of two or more independent cores on a single die. Six years later, in February 2007, NVIDIA (Santa Clara, CA) publicly released CUDA SDK [89], a set of development tools to write algorithms for execution on graphic processing units (GPUs). General-Purpose computation on GPUs (GPGPU) became available on virtually all desktop computers. Although software vendors have

started parallelizing their products, the vast majority of existing code is still sequential. In practice this means, for example, that only one-fourth of a quad-core CPU (which is currently standard in a desktop computer) is utilized by a given program.

Parallel computing has been used for scientific computing applications since the 1960s, when the first supercomputers were developed. However, only recently have these programming paradigms become useful for software running on desktop and notebook computers. In this work we demonstrate the advantage of exploiting modern computer architectures in scientific computing with multithreaded programming in Java for applications in image processing. Our aim is to provide software that is robust, efficient, flexible, and easy to use on affordable desktop and notebook computers. This work mainly focuses on image processing operations that require solving large-scale ill-posed inverse problems.

1.2 Inverse Problems

Inverse problems appear in many science and engineering applications and have been extensively studied by many applied mathematicians. The fact that most inverse problems cannot be solved analytically has triggered substantial research

on the development of computational methods that find approximate solutions. In image processing applications, ranging from biology and medicine to physics and astronomy, the goal of solving inverse problems is to reconstruct certain attributes of an image that were obfuscated in the acquisition process. We start the discussion of inverse problems from the formal definition.

Definition 1.1 *Let $\mathcal{H}_1$ and $\mathcal{H}_2$ denote Hilbert spaces and let $K : \mathcal{H}_1 \to \mathcal{H}_2$ be a (possibly nonlinear) operator. An **inverse problem** is to find $f \in \mathcal{H}_1$ such that $g = K(f)$, where K is an operator describing the relationship between data $g \in \mathcal{H}_2$ and model parameters f, and is a representation of the physical system.*

A well known example of an inverse problem is the Fredholm integral equation of the first kind in one space dimension

$$g(t) = \int_a^b k(t,s)f(s)ds, \tag{1.1}$$

The continuous kernel $k(t,s)$ and the function $g(t)$ are given and the goal is to find the function $f(s)$. If the kernel $k(t,s)$ is a function only of the difference of its arguments, $k(t,s) = k(t-s)$, and the limits of integration are $\pm\infty$, then equation (1.1) becomes

$$g(t) = \int_{-\infty}^{\infty} k(t-s)f(s)ds = k * f, \tag{1.2}$$

where $*$ denotes the convolution of the functions k and f. An analytic solution of equation (1.2) is given by

$$f(s) = \mathcal{F}^{-1}\left[\frac{\mathcal{F}[g(t)](\omega)}{\mathcal{F}[k(t)](\omega)}\right] = \int_{-\infty}^{\infty} \frac{\mathcal{F}[g(t)](\omega)}{\mathcal{F}[k(t)](\omega)} e^{2\pi i\omega t} d\omega, \tag{1.3}$$

where $\mathcal{F}$ and $\mathcal{F}^{-1}$ are the forward and inverse Fourier transforms respectively. Many inverse problems suffer from ill-posedness, so in the rest of this chapter we introduce techniques used to solve ill-posed inverse problems.

Definition 1.2 *[115] Let $K : \mathcal{H}_1 \to \mathcal{H}_2$. An operator equation*

$$g = K(f) \tag{1.4}$$

*is said to be **well-posed** provided*

1. *for each $g \in \mathcal{H}_2$ there exists $f \in \mathcal{H}_1$, called a solution, for which (1.4) holds;*

2. *the solution f is unique; and*

3. *the solution is stable with respect to perturbation in g. This means that if $Kf_* = g_*$ and $Kf = g$, then $f \to f_*$ whenever $g \to g_*$*

*A problem that is not well-posed is said to be **ill-posed**.*

Most inverse problems that arise in imaging applications are ill-posed. As a consequence, the matrices in the computational problem (obtained by discretizing the operator equation (1.4)) are very ill-conditioned and an accurate solution cannot be computed with standard linear or nonlinear solvers. To obtain an accurate solution of such problems, one has to use solvers that include some *regularization* techniques. Without regularization, the naïve inverse solution is usually dominated by noise and therefore it is a very poor approximation of the desired true solution.

1.3 Regularization

Regularization involves introducing additional information in order to stabilize an ill-posed inverse problem in the presence of noise. This information is usually in the form of a penalty: restrictions on smoothness of the solution or bounds on the vector space norm. We begin by showing why regularization is needed, and how it can be done through spectral filtering. To simplify the discussion, we assume a linear ill-posed inverse problem of the form

$$\mathbf{g} = \mathbf{K}\mathbf{f}_{\text{true}} + \boldsymbol{\eta}\,. \tag{1.5}$$

This expression is a discrete form of the operator equation (1.4) where, $\mathbf{K}$ is a large, ill-conditioned matrix that models the operator K, $\boldsymbol{\eta}$ is a vector that models additive noise (usually unknown), and $\mathbf{g}$ is a vector representing the data g. The goal is to find an approximation of the vector $\mathbf{f}_{\text{true}}$. In addition, $\mathbf{K}$ is assumed to be an $n \times n$ *normal* matrix [105], meaning that it has a spectral value decomposition (SVD)[1]

$$\mathbf{K} = \mathbf{Q}^* \mathbf{\Lambda} \mathbf{Q}\,, \tag{1.6}$$

where $\mathbf{\Lambda}$ is a diagonal matrix containing the eigenvalues of $\mathbf{K}$, $\mathbf{Q}^*$ is the complex conjugate transpose of $\mathbf{Q}$, and $\mathbf{Q}^*\mathbf{Q} = \mathbf{I}$. We assume further that the eigenvalues are ordered so that $|\lambda_1| \geq |\lambda_2| \geq \cdots \geq |\lambda_n| \geq 0$. Using the spectral decomposition, the inverse solution of (1.5) can be written as

$$\mathbf{f}_{\text{inv}} = \mathbf{K}^{-1}\mathbf{g} = \mathbf{K}^{-1}(\mathbf{K}\mathbf{f}_{\text{true}} + \boldsymbol{\eta}) = \mathbf{f}_{\text{true}} + \mathbf{K}^{-1}\boldsymbol{\eta} = \mathbf{f}_{\text{true}} + \sum_{i=1}^{n} \frac{\widehat{\eta}_i}{\lambda_i}\,\mathbf{q}_i\,, \tag{1.7}$$

where $\widehat{\boldsymbol{\eta}} = \mathbf{Q}\boldsymbol{\eta}$ and $\mathbf{q}_i$ is the i^{th} column of $\mathbf{Q}^*$. That is, the inverse solution is comprised of two terms: the desired true solution and an error term caused by noise in the data. To understand why the error term usually dominates the inverse solution, it is necessary to know the following properties of ill-posed problems:

[1]We realize that "SVD" usually refers to "singular value decomposition". We do not think there should be any confusion because our discussion of filtering can be done using the singular value decomposition in place of the spectral value decomposition.

- Assuming the problem is scaled so that $|\lambda_1| = 1$, the eigenvalues, $|\lambda_i|$, decay to, and cluster at 0, without a significant gap to indicate numerical rank.

- The eigenvectors $\mathbf{q}_i$ corresponding to small $|\lambda_i|$ tend to have more oscillations than the eigenvectors corresponding to large $|\lambda_i|$.

These properties imply that the high frequency components in the error are highly magnified by division of small eigenvalues. The computed inverse solution is dominated by these high frequency components, and is in general a very poor approximation of the true solution, $\mathbf{f}_{\text{true}}$.

In order to compute an accurate approximation of $\mathbf{f}_{\text{true}}$, or at least one that is not horribly corrupted by noise, the solution process must be modified. This process is usually referred to as regularization [56, 115]. One class of regularization methods, called *filtering*, can be formulated as a modification of the inverse solution [56]. Specifically, a filtered solution is defined as

$$\mathbf{f}_{\text{reg}} = \mathbf{K}_r^{\dagger}\mathbf{g} = \mathbf{Q}^*\mathbf{\Phi}\mathbf{\Lambda}^{-1}\mathbf{Q}\mathbf{g}, \tag{1.8}$$

where $\mathbf{\Phi} = \text{diag}\,(\phi_1, \phi_2, \ldots, \phi_n)$ and $\mathbf{\Lambda}^{-1} = \text{diag}\left(\frac{1}{\lambda_1}, \frac{1}{\lambda_2}, \ldots, \frac{1}{\lambda_n}\right)$. The *filter factors*, ϕ_i, satisfy $\phi_i \approx 1$ for large $|\lambda_i|$, and $\phi_i \approx 0$ for small $|\lambda_i|$. That is, the large eigenvalue (low frequency) components of the solution are reconstructed, while the components corresponding to the small eigenvalues (high frequencies)

are filtered out. Different choices of filter factors lead to different methods; popular choices are the truncated SVD (or pseudo-inverse), Tikhonov, and Wiener filters [56, 115, 49]. Some of these methods are discussed in Section 2.1.1. In this work we mainly focus on the efficient implementation of state of the art solvers for ill-posed inverse problems.

1.4 Outline of This Work

In this dissertation we develop new, open source Java software for scientific computing and image processing with an emphasis on solving large-scale inverse problems. The rest of this work is organized as follows. Chapter 2 describes mathematical theory and computational methods for solving two important classes of image processing algorithms: deconvolution and super-resolution. In Chapter 3 we motivate the choice of Java for our imaging applications and describe a high-performance Java library for scientific computing and image processing, which we call Parallel Colt. Chapter 4 discusses the details of the implementation, usage examples and benchmarking of various image processing algorithms implemented on top of Parallel Colt. Conclusions are summarized in Chapter 5. In Appendix A we describe the derivation of a fast Fourier transform algorithm and Appendix

B provides information about software packages that use the libraries and plugins described in this work.

Chapter 2

Deconvolution and Super-resolution

This chapter describes mathematical theory and computational methods for solving two important classes of image processing algorithms: deconvolution and super-resolution.

2.1 Deconvolution

In applications such as astronomy, medicine, physics and biology, scientists use digital images to record and analyze results from experiments. Environmental effects and imperfections in the imaging system can cause the recorded images to be degraded by blurring and noise. Image restoration (sometimes known as deblurring or deconvolution) is the process of reconstructing or estimating the true image from the degraded one. Image deblurring algorithms can be classified into

two types: spectral filtering methods and iterative methods. Another classification divides these algorithms into methods that do not require any information about the blur (also called blind deconvolution algorithms) and methods that need that information. In this work we only discuss the latter ones. Information about the blur is usually given in the form of a point spread function (PSF). A PSF is an image that describes the response of an imaging system to a point object. A theoretical PSF can be obtained based on the optical properties of the imaging system. The main advantage of this approach is that the obtained PSF is noise-free. The experimental technique, on the other hand, relies on taking a picture of a point object, for example in astronomy this can be a distant star.

Mathematically, image deblurring is the process of computing an approximation of a vector $\mathbf{f}_{\text{true}}$ (which represents the true image scene) from the linear inverse problem (1.5). Here, $\mathbf{K}$ is a large, usually ill-conditioned matrix defined by the PSF, and $\mathbf{g}$ is a vector representing the recorded image, which is degraded by blurring and noise. We assume that the PSF, and hence $\mathbf{K}$, is known, but the noise $\boldsymbol{\eta}$ is unknown. Because $\mathbf{K}$ is usually severely ill-conditioned, some form of regularization needs to be incorporated. As was already mentioned in Section 1.3, many regularization methods compute solutions of the form

$$\mathbf{f}_{\text{reg}} = \mathbf{K}_r^{\dagger}\mathbf{g}, \tag{2.1}$$

where $\mathbf{K}_r^{\dagger}$ can be thought of as a regularized pseudo-inverse of $\mathbf{K}$. The precise form of $\mathbf{K}_r^{\dagger}$ depends on many things, including the regularization method, the data $\mathbf{g}$, and the blurring matrix $\mathbf{K}$ [57]. Note that

$$\mathbf{f}_{\text{reg}} = \mathbf{K}_r^{\dagger}\mathbf{g} = \mathbf{K}_r^{\dagger}\mathbf{K}\mathbf{f}_{\text{true}} + \mathbf{K}_r^{\dagger}\boldsymbol{\eta}\,, \tag{2.2}$$

so such regularization methods attempt to balance the desire to have $\mathbf{K}_r^{\dagger}\mathbf{K} \approx \mathbf{I}$ while at the same time keeping $\mathbf{K}_r^{\dagger}\boldsymbol{\eta}$ from becoming too large.

2.1.1 Spectral Deconvolution

Spectral filtering methods exploit structure of the matrix to efficiently compute the singular (or spectral) value decomposition of $\mathbf{K}$, and use this information to construct $\mathbf{K}_r^{\dagger}$. The spectral filtering algorithms include many well known techniques for image deblurring such as the Wiener filter [49] and the pseudo inverse filter. But general approaches, such as truncated spectral decompositions and Tikhonov regularization [57] also belong to this group. Whether or not these techniques work well depends on special structure of the PSF (and hence of $\mathbf{K}$) and on the imposed boundary conditions [57].

The computational efficiency of spectral filtering methods for image deblurring with a *spatially invariant* PSF requires efficient discrete Fourier transform (DFT) and discrete cosine transform (DCT) routines. Although these deconvolution algorithms can be very efficient, and they are fairly easy to implement they have many limitations. First, efficient implementation requires the blur to have a very special structure, and this almost always means spatially invariant. In the case of *spatially variant* blurs, DFT and DCT based methods do not provide the right basis to use in filtering algorithms. It is possible to generalize the filtering ideas, using the singular value decomposition, but generally these approaches are very expensive. One exception is if the space variant blur is separable (i.e., the blurring operation can be separated into components involving a single vertical and a single horizontal blur). In this case, the matrix $\mathbf{K}$ can be represented as a Kronecker product of two smaller matrices. Another limitation of spectral filtering methods is that it is not possible to include additional constraints, such as nonnegativity, in the reconstruction algorithms. Figure 2.1 shows a comparison between the spectral and iterative methods; in practice the reconstruction quality is usually much better when an iterative algorithm is used.

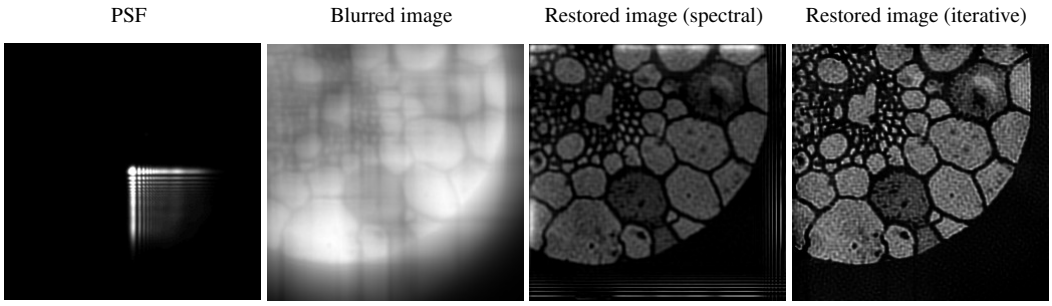

Figure 2.1: Spectral vs. iterative image deblurring.

Boundary Conditions

If the blur is assumed to be spatially invariant, then the PSF is the same regardless of the position of the point source in the image field of view. In this case, if we also enforce periodic boundary conditions, then $\mathbf{K}$ has a circulant matrix structure [57], and the spectral factorization

$$\mathbf{K} = \mathbf{F}^* \mathbf{\Lambda} \mathbf{F} \,, \tag{2.3}$$

where $\mathbf{F}$ is the DFT matrix; a d-dimensional image implies $\mathbf{F}$ is a d-dimensional DFT matrix. In this case, the matrix $\mathbf{F}$ does not need to be constructed explicitly; we simply need a call to a function to evaluate the DFT. Efficient implementations of DFTs are usually referred to as *fast Fourier transforms* (FFT) [33] (see Section 6.1). Computations such as the matrix-vector multiplications $\mathbf{Fb}$ and $\mathbf{F}^*\mathbf{b}$ are

done by functions calls

$$\mathbf{F}\mathbf{b} \quad \Leftrightarrow \quad \texttt{fft}(\mathbf{b}) \quad \text{(forward FFT)}$$

$$\mathbf{F}^{*}\mathbf{b} \quad \Leftrightarrow \quad \texttt{ifft}(\mathbf{b}) \quad \text{(inverse FFT)}$$

The eigenvalues of $\mathbf{K}$ can be obtained by computing an FFT of the first column of $\mathbf{K}$, and the first column of $\mathbf{K}$ can be obtained directly from the PSF [57].

If the image has significant features near the boundary of the field of view, then periodic boundary conditions can cause ringing artifacts in the reconstructed image. In this case, it may be better to use reflexive boundary conditions. But changing the boundary conditions changes the structure of $\mathbf{K}$, and it no longer has the Fourier spectral decomposition given in equation (2.3). However, if the PSF is also symmetric (as in the case of atmospheric turbulence), then $\mathbf{K}$ is a mix of Toeplitz and Hankel structures [57], and has the spectral value decomposition

$$\mathbf{K} = \mathbf{C}^{T}\mathbf{\Lambda}\mathbf{C}\,, \tag{2.4}$$

where $\mathbf{C}$ is the DCT matrix; a d-dimensional image implies $\mathbf{C}$ is a d-dimensional DCT matrix. In this case, the matrix $\mathbf{C}$ does not need to be constructed explicitly; we simply need a call to a function to evaluate the DCT. As with FFTs, there are very efficient algorithms for evaluating DCTs. Furthermore, computations such

as the matrix-vector multiplication $\mathbf{Cb}$ and $\mathbf{C}^T\mathbf{b}$ are done by functions calls

$$\mathbf{Cb} \quad \Leftrightarrow \quad \texttt{dct}(\mathbf{b}) \quad \text{(forward DCT)}$$

$$\mathbf{C}^T\mathbf{b} \quad \Leftrightarrow \quad \texttt{idct}(\mathbf{b}) \quad \text{(inverse DCT)}$$

The eigenvalues of $\mathbf{K}$ can be obtained by computing a DCT of the first column of $\mathbf{K}$, and the first column of $\mathbf{K}$ can be obtained directly from the PSF [57]. Note that in the case of the FFT, $\mathbf{F}$ has complex entries and thus computations necessarily require complex arithmetic. However, in the case of the DCT, $\mathbf{C}$ has real entries, and all computations can be done in real arithmetic.

Truncated SVD

For this method [56] the filter factors have the simple form

$$\phi_i = \begin{cases} 1, & i = 1, \ldots, k \\ \\ 0, & i = k+1, \ldots n \end{cases} \tag{2.5}$$

The truncation parameter k determines the number of SVD components used in the regularized solution

$$\mathbf{f}_k = \mathbf{K}_k^{\dagger}\mathbf{g} = \sum_{i=1}^{k} \frac{\widehat{g}_i}{\lambda_i}\mathbf{q}_i, \tag{2.6}$$

where $\widehat{\mathbf{g}} = \mathbf{Q}\mathbf{g}$, and $\mathbf{q}_i$ is the i^{th} column of $\mathbf{Q}^*$. In other words, the Truncated SVD solution $\mathbf{f}_k$ is obtained by first replacing the ill-conditioned matrix $\mathbf{K}$ by the rank-k matrix $\mathbf{K}_k$ defined as

$$\mathbf{K}_k = \sum_{i=1}^{k} \mathbf{q}_i \lambda_i \mathbf{q}_i^*, \tag{2.7}$$

followed by computing the minimum-norm least squares solution to

$$\min \|\mathbf{f}\|_2^2 \quad \text{subject to} \quad \min \|\mathbf{g} - \mathbf{K}_k \mathbf{f}\|_2^2 \tag{2.8}$$

Tikhonov

For the Tikhonov method [56], the regularization filter factors are of the form

$$\phi_i = \frac{|\lambda_i|^2}{|\lambda_i|^2 + \alpha^2}\,, \tag{2.9}$$

where the scalar α is called a *regularization parameter*, and usually satisfies $|\lambda_n| \leq \alpha \leq |\lambda_1|$. Note that smaller α lead to more ϕ_i approximating 1. This particular choice of the filter factors yields the following minimization problem

$$\min_{\mathbf{f}} \left\{ \|\mathbf{g} - \mathbf{K}\mathbf{f}\|_2^2 + \alpha^2 \|\mathbf{f}\|_2^2 \right\} \tag{2.10}$$

with a solution vector given by

$$\mathbf{f}_{\text{filt}} = \sum_{i=1}^{n} \frac{|\lambda_i|^2}{|\lambda_i|^2 + \alpha^2} \frac{\widehat{g}_i}{\lambda_i} \mathbf{q}_i, \tag{2.11}$$

where $\widehat{\mathbf{g}} = \mathbf{Q}\mathbf{g}$, and $\mathbf{q}_i$ is the i^{th} column of $\mathbf{Q}^*$.

Generalized Tikhonov

The above Tikhonov method is a special case of more general approach called *damped least squares* or *generalized Tikhonov* regularization [57]. The minimization problem for this method takes the form

$$\min_{\mathbf{f}} \left\{ \|\mathbf{g} - \mathbf{K}\mathbf{f}\|_2^2 + \alpha^2 \|\mathbf{D}\mathbf{f}\|_2^2 \right\} \tag{2.12}$$

where $\mathbf{D}$ is a regularization matrix, usually an approximation to a derivative operator. One should notice that the damped least squares approach is equivalent to the Tikhonov approach when $\mathbf{D}$ is equal to the identity matrix. To find the regularized solution to problem (2.12), one attempts to balance between the size of two different terms. The first term, $\|\mathbf{g} - \mathbf{K}\mathbf{f}\|_2^2$, measures the *goodness-of-fit* of the solution $\mathbf{f}$. If its value is too large, then the solution does not fit the data $\mathbf{g}$ very well, and if the value is too small, then the solution is probably corrupted by the noise in the data. The second term $\|\mathbf{D}\mathbf{f}\|_2^2$, called the *regularization term*, measures the *smoothing*. The value of this term should be small if $\mathbf{f}$ is a good quality solution

(i.e. the deblurred image matches our expectation) and it should be large when the reconstruction contains a large component of inverted noise. The regularization parameter α allows to keep the balance between the minimization of these two terms. If α is too small, then the solution will be influenced too much by the noise in the data. Conversely, if the value of α is too large, then the solution will be over-smoothed, i.e. the details in the deblurred image will not be visible. In the next section we present three of the most common methods for choosing the optimal value for the regularization parameter.

Parameter Choice Methods

The regularization parameter is problem dependent, and in general it is nontrivial to choose an appropriate value. Various techniques can be used, such as the discrepancy principle, the L-curve, and the generalized cross-validation [56, 115]. There are advantages and disadvantages to each of these approaches [70], especially for large-scale problems.

The discrepancy principle [79] method relies on having a good approximation of the expected value of the error in the data ($\delta = \|\boldsymbol{\eta}\|_2^2$). If this information is given, then the value of α should be chosen so that the norm of the residual ($\|\mathbf{g} - \mathbf{K}\mathbf{f}_{\text{filt}}\|_2$) is approximately equal δ. As the error approaches zero ($\delta \to 0$),

then the filtered solution approaches the exact solution ($\mathbf{f}_{\text{filt}} \rightarrow \mathbf{f}_{\text{true}}$), i.e. the discrepancy principle is convergent as the error norm goes to zero, but it tends to find values of the regularization parameter that over-smooth the solution.

The L-curve criterion [55] is a log-log plot of the norm of the regularized solution ($\|\mathbf{f}_{\text{filt}}\|_2$) versus the corresponding residual norm ($\|\mathbf{g} - \mathbf{K}\mathbf{f}_{\text{filt}}\|_2$) for each of a set of regularization parameter values. This plot is usually in the shape of the letter L and the optimal value of α lies at the corner, or at the location of maximum curvature. This method fails as the error norm approaches zero.

The generalized cross-validation (GCV) [46] criterion relies on the principle that if we remove a data value, then a good choice of the regularization parameter should be able to predict the missing data point well. GCV determines the parameter α as a minimum of the GCV functional

$$G(\alpha) = \frac{\|(\mathbf{I} - \mathbf{K}\mathbf{Q}^*\mathbf{\Phi}\mathbf{\Lambda}^{-1}\mathbf{Q})\mathbf{g}\|_2^2}{(\text{trace}(\mathbf{I} - \mathbf{K}\mathbf{Q}^*\mathbf{\Phi}\mathbf{\Lambda}^{-1}\mathbf{Q}))^2}, \tag{2.13}$$

where $\mathbf{K}\mathbf{Q}^*\mathbf{\Phi}\mathbf{\Lambda}^{-1}\mathbf{Q}$ is the matrix that maps the right hand side $\mathbf{g}$ onto the regularized solution $\mathbf{f}_{\text{filt}}$. For Tikhonov regularization, the above formula can be written in the simplified form

$$G(\alpha) = n \sum_{i=1}^{n} \left(\frac{\alpha^2 \, |\widehat{g}_i|}{|\lambda_i|^2 + \alpha^2} \right)^2 \Bigg/ \left(\sum_{i=1}^{n} \frac{\alpha^2}{|\lambda_i|^2 + \alpha^2} \right)^2, \tag{2.14}$$

where $\widehat{\mathbf{g}} = \mathbf{Q}\mathbf{g}$. Standard optimization routines can be used to minimize $G(\alpha)$. GCV also fails to converge to the true solution as the error norm goes to zero. Moreover, the graph of G can be very flat near its minimum value which makes it hard to determine that value numerically. However, despite the limitations, this parameter choice method usually works best in practice.

From the above discussion it is clear that no parameter choice method is perfect. In this work we use the generalized cross-validation approach.

2.1.2 Iterative Deconvolution

With iterative methods, a sequence of approximations of $\mathbf{f}$ is constructed, where hopefully subsequent approximations provide better reconstructions. Mathematically this is equivalent to solving a particular optimization problem involving $\mathbf{K}$ and $\mathbf{g}$, which could be formulated as something simple like a linear least squares problem, or something more complicated that incorporates (possibly nonlinear) constraints. As with spectral filtering methods, regularization must be incorporated using, for example, *a priori* constraints, or through appropriate convergence criteria, or even a combination of such techniques. All the algorithms considered here have the general form shown in Algorithm 1. The most computationally expensive operations are performed in line 3 of this algorithm and include a

matrix-vector product with $\mathbf{K}$ and a linear system solve involving the preconditioner $\mathbf{P}$. In most cases, both of these operations can be efficiently implemented using trigonometric transforms or with sparse matrix computations. The goal of preconditioning is to speed-up the convergence, but also not too significantly increase the computational cost per iteration.

Algorithm 1 General form of iterative deconvolution algorithms.

1. $\mathbf{f}_0 =$ initial estimate of $\mathbf{f}$
2. for $j = 0,\ 1,\ 2,\ \ldots$
3. $\quad \mathbf{f}_{j+1} =$ computations involving $\mathbf{f}_j$, $\mathbf{K}$, preconditioner $\mathbf{P}$ and other quantities
4. $\quad$ determine if stopping criteria are satisfied
5. end

Well known examples of iterative image reconstruction algorithms include expectation maximization (EM) type approaches (such as the Richardson-Lucy algorithm [101]), conjugate gradient (CG) type methods [111], and many others; see for example [29, 14, 72]. One important advantage of using iterative algorithms is that they can be used on a much wider class of blurring models, including spatially variant blurs. Although iterative methods are generally more expensive than spectral filtering methods for simple spatially invariant blurs, they are much more efficient for difficult spatially variant blurs. Moreover, it is much easier to incorporate constraints (e.g., nonnegativity) in the algorithms. The main disadvantages

of iterative methods are the need to determine how to incorporate regularization (to stabilize the iterative method in the presence of noise), and how to determine an appropriate stopping iteration. Hybrid approaches [90, 15, 31] that combine a conjugate gradient type iterative method with a spectral factorization can be effective in overcoming these disadvantages.

In this work we consider four iterative solvers: hybrid bidiagonalization regularization (HyBR) [31], conjugate gradient for least squares (CGLS) [16], modified residual norm steepest descent (MRNSD) [84], and the nonnegatively constrained Landweber iteration [14].

HyBR is an efficient iterative method that combines an iterative Lanczos bidiagonalization method with a singular value decomposition-based regularization method to stabilize the semiconvergence behavior that is characteristic of many ill-posed problems. In other words, HyBR can automatically choose regularization parameters and stop the iteration process based on the data. Unfortunately, this algorithm does not enforce nonnegativity constraints, and the storage requirements grow as the iterations proceed.

Due to its fast convergence, Krylov subspace methods are attractive for iterative image deblurring, however, they do not find a nonnegative solution. CGLS is a conjugate gradient method applied to the normal equations. The stopping criterion

of CGLS is based only on the value of the relative residual, thus this method is not able to determine an appropriate stopping iteration. An advantage of CGLS is that it is faster than HyBR (CGLS does not need to compute the stopping criterion parameters).

MRNSD is a nonnegatively constrained steepest descent method on the normal equations. Although it produces a nonnegative solution, MRNSD also lacks a sophisticated stopping criterion, such as the one implemented in HyBR. Moreover, it can converge very slowly compared to CGLS and HyBR.

The nonnegatively constrained Landweber iteration is a very simple stationary iterative method with slow convergence rate. Similar to CGLS and MRNSD, Landweber iteration is not able to detect the semiconvergence without a good estimate of the noise level. However, the computational cost per iteration for this method is the lowest out of the four algorithms considered here.

Matrix-Vector Products

In Section 2.1.1 we described how to perform matrix-vector products efficiently when the PSF is spatially invariant and the boundary conditions are assumed to be either periodic or reflexive. There is another approach commonly used in practice - zero boundary conditions. In that case, matrix $\mathbf{K}$ has a Toeplitz structure, which

does not have a spectral decomposition by means of fast trigonometric transforms. However, the Toeplitz matrix can be embedded into a larger circulant matrix, and then the matrix-vector product computations are done by padding the image with an appropriate number of zeros and using FFTs as for periodic boundary conditions.

When a generic spatially variant blur is considered, then every pixel in the image can have its own point spread function. Implementing this approach may not be feasible in practice, but it is often appropriate to assume that for small subregions of the image, the PSF is spatially invariant. Interpolation can then be used to combine the individual PSFs into an approximation of the spatially variant blurring operator. If piecewise constant interpolation is used, then the spatially variant PSF matrix has the following structure

$$\mathbf{K} = \sum_{i=1}^{p} \mathbf{D}_i \mathbf{K}_i, \tag{2.15}$$

where $\mathbf{K}_i$ are Toeplitz matrices, except for those corresponding to the border, where their structure depends on the boundary conditions. $\mathbf{D}_i$ are diagonal matrices with kth diagonal entry of $\mathbf{D}_i$ equal one if the kth point is in region i, and zero otherwise. The matrix-vector products involving spatially variant PSFs are computed by overlap-add and overlap-save methods [82], where the fast algorithms

for spatially invariant blur are used on each subregion of the image.

Preconditioning

Every preconditioner matrix $\mathbf{P}$ for ill-posed inverse problems should satisfy the following general properties:

- $\mathbf{P}$ is relatively inexpensive to construct,
- It is relatively inexpensive to solve linear systems of the form $\mathbf{Px} = \mathbf{y}$,
- The preconditioned system should satisfy $\mathbf{P}^{-1}\mathbf{K}_r \approx \mathbf{I}$, where $\mathbf{K}_r = \mathbf{Q}^*\mathbf{\Phi}\mathbf{\Lambda}\mathbf{Q}$.

The last condition, refers to the rate of convergence, which is faster when more of the large eigenvalues are clustered around 1. The process of image deconvolution requires solving linear systems with highly structured matrices, including circulant, Toeplitz and Hankel. Preconditioners for such systems have been widely studied in the literature [26, 80, 81]. In this work we consider a method, where the preconditioner is constructed to be a circulant approximation of matrix $\mathbf{K}$.

When the PSF is spatially invariant, then there exist very inexpensive methods [27, 28] (based on FFTs) to construct the circulant preconditioner by solving one of the following minimizations problems

$$\min \|\mathbf{K} - \mathbf{P}\|_F \qquad \text{or} \qquad \min \|\mathbf{K} - \mathbf{P}\|_1 \tag{2.16}$$

over all circulant matrices, $\mathbf{P}$. For spatially variant blurs a similar approach is used, however, in that case we construct a preconditioner using a single PSF that is the average of all given PSFs. Although this technique is not optimal, it works well in practice and makes the implementation much easier than creating a separate preconditioner for each PSF [83].

2.2 Motion Correction of PET Brain Images

Deconvolution is also used in medical imaging, where sharper images with more visible details directly translate to better diagnosis. Patient movement during positron emission tomography (PET) scanning introduces motion blur and reduces the resolution of the image. While some patient motion can be tolerable in low-resolution imaging systems, with new PET scanners, even small amount of motion can degrade image quality. The resolution of latest PET scanners approaches 2 mm, however it is only attainable when the subject is motionless. On the other hand, it is unreasonable to expect patients to keep their heads perfectly still, unless the acquisition time is very small. Thus, one can either prevent or

correct for subject motion during the scan. A cooperative patient, with the aid of a head restraint system, can often limit the movement to within 2-4 mm for the duration of a PET study. However, even with that restraint system, translations in the range of 5mm and rotations of 1 degree have been observed [18, 50]. Even more movement may be expected when patients suffer from psychiatric or neurologic diseases.

However, if it is possible to continuously measure the position of the head, this positional information can be used to correct the measured data. Different methods for head motion tracking and correction have been described in the literature. Position monitoring has been implemented using light-emitting diodes (LEDs) [95], magnetic field [50] and infrared [45, 77] sources and targets to track patient head position. A commercial system able to make measurements such as these is the VICRA stereo camera from NDI (Northern Digital, Waterloo, Ontario, Canada). It provides estimates of the position of markers placed on the head at up to 20 Hz. Given that object positioning information is available, there are different ways to use it to correct patient motion.

Motion correction methods that have been reported in the literature fall into three general categories [41]. Sinogram rebinning described by Bloomfield [18], Buhler [22], Menke [77] and Rahmim [98] involves using the known subject movement

to move counts into the position where they would have been detected had the patient not moved. This method requires list mode reconstructions and careful consideration of scanner normalization. A second approach is the multiple acquisition frame (MAF) method described by Picard and Thompson [95] wherein short duration frames are acquired and each is corrected for motion prior to summing to create the final image. However, this method uses only the average head motion within a frame and hence does not correct for large head movements. More recently, known patient motion has been incorporated into a system response function used during maximum likelihood expectation maximization (MLEM) reconstruction of the emission image [99]. Since this method involves system matrix modification, it requires detailed understanding of the geometry of the scanner as well as detector response characteristics and attenuation. In this work we consider only the last approach.

2.2.1 Motion Detection

This section describes a fairly simple procedure for tracking and recording patient movements during a PET scan. See Figure 2.2 for an illustration of the tracking device mounted on the rear of the scanner gantry and the reference tool mounted inside the scanner.

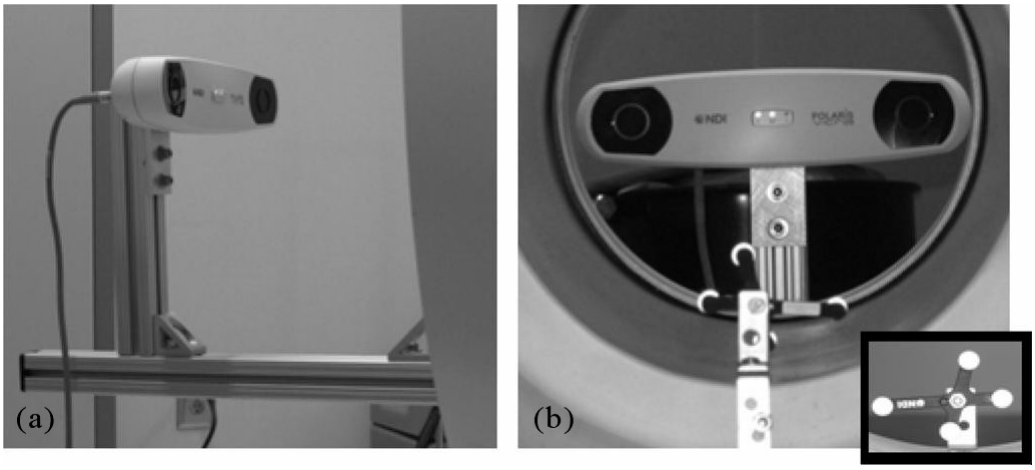

Figure 2.2: (a) The Polaris Vicra mounted on the rear of the scanner gantry using the 80/20 T-slot system. (b) Reference tool mounted inside the scanner (inset – another view of the reference tool) [97].

The mobile, or patient, target is attached to the patient's head using a modified swimming cap. This (hopefully) prevents the target from moving independently of the patient's head, which would cause inaccurate information to be recorded. The patient target is composed of four passive markers that reflect infrared light. The tracker, shown in front of the PET machine in the figure, emits infrared light. The light is reflected off the four markers, and their orientation (as a unit quaternion $\dot{q} = q_0 + iq_x + jq_y + kq_z$) and position (as a vector $[p_x, p_y, p_z]$) are calculated; this provides motion information in six degrees of freedom. These measurements are made multiple times per second and stored, resulting in fairly accurate motion

information.

As the tracker is not attached to the PET machine, it is possible that it may accidentally be moved. In order to account for any unintentional tracker movement, four passive markers are attached to the PET machine itself; this is called a reference target. A computer connected to the tracker records the position and orientation of the reference target in addition to the patient target.

It is important to realize that we now have three different coordinate systems to work with:

- the reference coordinate system,
- the patient target coordinate system, and
- the image coordinate system.

Note that both the reference space and the image space are fixed; the target space is attached to the patient's head and moves as the patient moves.

Following the procedure from [97], let $\mathbf{X}$ be some point that is measured with respect to each of these coordinate systems: $\mathbf{X}_R$ is with respect to the reference space, $\mathbf{X}_T$ is with respect to the target space, and $\mathbf{X}_I$ is with respect to the image

space. We can relate these coordinate systems using transformation matrices as

$$\begin{aligned} \mathbf{X}_I &= \mathbf{C}\mathbf{X}_R \\ \mathbf{X}_R &= \mathbf{Q}\mathbf{X}_T \\ \mathbf{X}_T &= \mathbf{M}_I\mathbf{X}_I \end{aligned} \tag{2.17}$$

Note that, by construction, $\mathbf{M}_I = \mathbf{Q}^{-1}\mathbf{C}^{-1}$; thus, we only need to determine two of the transformation matrices to be able to represent a point in any of these three coordinate systems.

The transformation matrix $\mathbf{C}$ is invariant under patient movement; it is the calibration matrix that relates the reference space and the image space, both of which are fixed. If the reference target does not move once it is attached to the PET machine, this calibration matrix need only be computed once and then used for all future scans. See [97] for a description of how this matrix is computed.

The 4×4 matrix $\mathbf{Q}$ (see Eq. (2.18)) represents the transformation between the reference and target coordinate frames for some orientation and position (which we call the *pose*) of the head:

$$\mathbf{Q} = \begin{bmatrix} q_0^2 + q_x^2 - q_y^2 - q_z^2 & 2(q_x q_y - q_0 q_z) & 2(q_x q_z + q_0 q_y) & p_x \\ 2(q_y q_x + q_0 q_z) & q_0^2 - q_x^2 + q_y^2 - q_z^2 & 2(q_y q_z - q_0 q_x) & p_y \\ 2(q_z q_x - q_0 q_y) & 2(q_z q_y + q_0 q_x) & q_0^2 - q_x^2 - q_y^2 + q_z^2 & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.18}$$

Since the patient target is attached to the patient's head, a particular point in the patient's brain has a constant position in the target space. In other words, $\mathbf{X}_T$ is invariant under patient movement. If we consider some point $\mathbf{X}_T$ before and after a discrete movement, where positions after movement are denoted by $'$, then

$$\begin{aligned} \mathbf{X}'_T &= \mathbf{X}_T \\ \mathbf{M}'_I \mathbf{X}'_I &= \mathbf{M}_I \mathbf{X}_I \\ \mathbf{Q}'^{-1} \mathbf{C}^{-1} \mathbf{X}'_I &= \mathbf{Q}^{-1} \mathbf{C}^{-1} \mathbf{X}_I \\ \mathbf{X}'_I &= \mathbf{C} \mathbf{Q}' \mathbf{Q}^{-1} \mathbf{C}^{-1} \mathbf{X}_I \end{aligned} \tag{2.19}$$

Thus, the transformation from some point before movement to the corresponding point after movement is given by $\mathbf{C}\mathbf{Q}'\mathbf{Q}^{-1}\mathbf{C}^{-1}$.

Since we take all of the motion information recorded during the scan and break it up into several bins, each of which has a fairly distinct pose, we can consider $\mathbf{X}_I$ as coming from the first bin, or the initial pose. Then, every movement is given in relation to the first pose. Thus, we need to compute $\mathbf{Q}^{-1}$ only once and so, for

each distinct pose j after the first one, we compute the transformation from the initial pose to the j^{th} one as $\mathbf{C}\mathbf{Q}_j\mathbf{Q}^{-1}\mathbf{C}^{-1}$, where $\mathbf{Q}_j$ is the matrix $\mathbf{Q}$ computed by using the average quaternion and position vector of the j^{th} pose.

2.2.2 Mathematical Formulation

We now consider how to use the transformation matrices described in the previous section to set up a system of equations. From above, we have

$$\mathbf{X}_i = \mathbf{C}\mathbf{Q}_i\mathbf{Q}^{-1}\mathbf{C}^{-1}\mathbf{X}_1 \tag{2.20}$$

where $\mathbf{X}_i$ is a point in the i^{th} bin, $\mathbf{C}$ is the calibration matrix, $\mathbf{Q}_i$ is determined by Eq. (2.18) using the motion information of the i^{th} bin, $\mathbf{Q}^{-1}$ is the inverse of $\mathbf{Q}_1$, and $\mathbf{X}_1$ is the corresponding point in the first bin. Having these transformation matrices allows us to determine displacements on our regular 3D grid of voxels and by using interpolation we can create displacement matrices $\mathbf{K}_i$. Each $\mathbf{K}_i$ is a large, sparse matrix that contains the interpolation weights for bin i; for trilinear interpolation, each row contains at most eight entries. If nearest neighbor interpolation is used, each row contains at most one entry and its value is always 1. Thus, if $\mathbf{f}$ is the true 3D image stored in vector form, then $\mathbf{K}_i\mathbf{f}$ produces an image that is in the same pose as the patient's head was, on average, during the time frame corresponding to bin i.

If only a single image $\mathbf{g}$ is acquired during the PET scan, then we have

$$\mathbf{Kf} \approx \mathbf{g}, \tag{2.21}$$

where

$$\mathbf{K} = \sum_{i=1}^{n} w_i \mathbf{K}_i. \tag{2.22}$$

n, in the above equation denotes a number of bins and weights w_i are calculated using the segmentation of the motion information described in the next section. Note that, in exact arithmetic, $\mathbf{K}_1$ is just the identity matrix and the equality in equation (2.21) is not possible due to the presence of noise (it is an example of ill-posed inverse problem). To solve the system of equations (2.21) we use one of four iterative solvers: HyBR, CGLS, MRNSD or ordered subsets expectation maximization (OSEM) [65]. Spectral deblurring methods cannot be applied to this problem, because matrix $\mathbf{K}$ does not have any special structure and its eigenvalues cannot be computed by fast trigonometric transforms.

2.2.3 Determining Head Positions

We have developed an algorithm for the segmentation of the motion information that automatically determines distinct head positions (the number of bins). In the previous work [97] this step was done manually by plotting the values of

q_0, q_x, q_y, q_z, p_x, p_y, p_z (against time) and then dividing them into intervals by visual inspection. MATLAB code for this automated method is shown in Algorithm 2. In lines 2 - 9, the variables are initialized. Then, in lines 10 - 18, the weighted average motions `avgMotions` are computed. Finally, in the last part of the algorithm, the segmentation `seg` is calculated from `avgMotions` using the tolerance `tol` (the standard deviation of the weighted average motions (line 19)).

Algorithm 2 Segmentation of the motion information.

```
1.  function seg = ComputeSegmentation(motions, params)
2.    sr = params.samplingRate;
3.    minSegSize = params.scanDuration / sr;
4.    fs = params.timeOffset * sr;
5.    ls = fs + params.scanDuration * sr - 1;
6.    period = 1.0 / params.samplingRate;
7.    quatsWeight = params.quaternionsPercentage / 100.0;
8.    avgQuats = zeros(ls - fs + 1, 1);
9.    avgTrans = = zeros(ls - fs + 1, 1);
10.   for k=1:4
11.     avgQuats = avgQuats + Normalize(motions(fs:ls,k));
12.   end
13.   avgQuats = avgQuats * quatsWeight;
14.   for k=5:7
15.     avgTrans = avgTrans + Normalize(motions(fs:ls,k));
16.   end
17.   avgTrans = avgTrans * (1.0 - quatsWeight);
18.   avgMotions = Normalize(avgQuats + avgTrans);
19.   tol = std(avgMotions);
20.   seg(1) = 0; i = 2; seg(i) = period;
21.   refValue = avgMotions(1);
22.   for k = 2:length(avgMotions)
23.     if(abs(avgMotions(k) - refValue) >= tol)
24.       if ((seg(i) - seg(i - 1)) < minSegSize)
25.         refValue = avgMotions(k);
26.         seg(i) = seg(i) + period;
27.       else
28.         refValue = avgMotions(k);
29.         i = i + 1; seg(i) = seg(i-1) + period;
30.       end
31.     else
32.       seg(i) = seg(i) + period;
33.     end
34.   end
35.   seg(i) = params.scanDuration;
```

2.3 Super-Resolution

In most image processing applications it is desirable to have images with high spatial resolution. One approach to obtain such images is to build sophisticated devices having intrinsically high-resolution capabilities. However, these techniques, in addition to being costly, suffer from other limitations that are difficult to overcome [94]. Super-resolution [30] is a less expensive alternative that recently has gained popularity in digital imaging and video applications. One very interesting application of this technique is in surveillance cameras, where super-resolution can help for instance in a criminal investigation.

Super-resolution is an image fusion and reconstruction problem, where an improved resolution image is obtained from several geometrically warped, low-resolution images of the same scene. Here we assume that the geometrical warping is limited to affine transformations and each of these images is shifted / rotated by subpixel displacements. The high-resolution image is not only an image that has more pixels (like in the case of interpolation), but it also has more visible details. The subpixel displacements guarantee that each low-resolution image contains different information about the same scene. If the low-resolution images were shifted by an integer multiple of the pixel size, then the process of super-resolution would be equivalent to an interpolation in the sense that the re-

construction would not have more visible details. It should be emphasized that super-resolution is the only nonlinear inverse problem that is considered in this work; all deconvolution methods discussed above were formulated as linear inverse problems.

Suppose we have acquired m low-resolution images $\mathbf{g}^{(1)}, \mathbf{g}^{(2)}, \ldots, \mathbf{g}^{(m)}$ that meet the assumptions described in the previous paragraph.The process of super-resolution can be modeled in the following way

$$\mathbf{g} = \mathbf{K}(\mathbf{y})\mathbf{f} + \boldsymbol{\eta}, \tag{2.23}$$

where

$$\mathbf{K}(\mathbf{y}) = \begin{bmatrix} \mathbf{DS}(\mathbf{y}^{(1)}) \\ \mathbf{DS}(\mathbf{y}^{(2)}) \\ \vdots \\ \mathbf{DS}(\mathbf{y}^{(m)}) \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} \mathbf{y}^{(1)} \\ \mathbf{y}^{(2)} \\ \vdots \\ \mathbf{y}^{(m)} \end{bmatrix}, \quad \mathbf{g} = \begin{bmatrix} \mathbf{g}^{(1)} \\ \mathbf{g}^{(2)} \\ \vdots \\ \mathbf{g}^{(m)} \end{bmatrix}, \quad \boldsymbol{\eta} = \begin{bmatrix} \boldsymbol{\eta}^{(1)} \\ \boldsymbol{\eta}^{(2)} \\ \vdots \\ \boldsymbol{\eta}^{(m)} \end{bmatrix}.$$

In the above equation, $\mathbf{D}$, called a *decimation matrix*, transforms a high-resolution image into a low-resolution image. Matrix $\mathbf{S}$ models a geometric distortion (affine transformation) of the high-resolution image, $\mathbf{f}$ and it is defined by the parameter vector $\mathbf{y}^{(i)}$. This vector contains six values that uniquely define an affine transformation in a 3D space. Finally, a vector $\boldsymbol{\eta}$ models additive nose.

At this point it should be clear that the super-resolution problem requires estimating both the values for the geometric distortions ($\mathbf{y}$) and the unknown high-resolution image ($\mathbf{f}$). In other words, the reconstruction process can be split into two steps: (1) estimating the relative displacement of each point in each image from points in a reference image and (2) solving a linear ill-posed inverse problem to obtain the high-resolution image. In mathematical terms this process can be formulated as a nonlinear least squares problem

$$\min_{\mathbf{f},\mathbf{y}} \phi(\mathbf{f},\mathbf{y}) = \min_{\mathbf{f},\mathbf{y}} \|\mathbf{g} - \mathbf{K}(\mathbf{y})\mathbf{f}\|_2^2 \tag{2.24}$$

which can be solved using a Gauss-Newton approach [88].

Here we assume problem (2.24) is solved by the reduced Gauss-Newton method with HyBR used as a linear solver (see Algorithm 3). $\mathbf{J}_\psi$ in line 5 of Algorithm 3 denotes the Jacobian of the reduced cost functional as described in [30].

Algorithm 3 Reduced Gauss-Newton Algorithm with HyBR.

1. choose initial $\mathbf{y}_0$
2. for $k = 0, 1, 2, \ldots$
3. $\quad \mathbf{f}_k = \text{HyBR}(\mathbf{K}(\mathbf{y}_k), \mathbf{g})$
4. $\quad \mathbf{r}_k = \mathbf{g} - \mathbf{K}(\mathbf{y}_k)\mathbf{f}_k$
5. $\quad \mathbf{d}_k = \arg\min_{\mathbf{d}} \|\mathbf{J}_\psi \mathbf{d} - \mathbf{r}_k\|_2$
6. $\quad \mathbf{y}_{k+1} = \mathbf{y}_k + \mathbf{d}_k$
7. end

Chapter 3

Java in Scientific Computing and Imaging

In this chapter we motivate the choice of Java for our imaging applications and describe Parallel Colt [119], a high performance Java library for scientific computing and image processing.

3.1 Why Java?

Although Java was not designed to be a scientific computing language [25], it has several unique features that are attractive for high-performance scientific computing. Because distributions are available for virtually all computing platforms, Java is an extremely portable programming language. In addition, starting in 2007,

Java has become an open source project, allowing anyone to modify and adapt it to their needs. Java has native support for multithreading, and since version 5.0 [108] it is equipped with concurrency utilities in the `java.util.concurrent` package. Moreover, the performance of the latest version of Java (6.0) is comparable to the performance achieved by programs written in Fortran or C/C++ [7, 52]. Finally, sophisticated imaging functionality is built into Java, allowing for efficient visualization and animation of computational results. This is especially important for our work in image processing, but is also useful in many areas of scientific computation, such as computational fluid dynamics.

However, because of certain design choices, there are also disadvantages to using Java in scientific computing. These include no primitive type for complex numbers, an inability to do operator overloading, and no support for IEEE extended precision floats. In addition, Java arrays were not designed for high-performance computing; a multi-dimensional array is an array of one-dimensional arrays, making it difficult to fully utilize cache memory. Moreover, Java arrays are not resizable, and only 32-bit array indexing is possible. Finally, GPGPU is currently not possible in Java. There are libraries, such as JCuda [67], that provide Java bindings to CUDA, but they are only wrappers to underlying C code.

3.2 Related Work

There are many Java libraries for scientific computing [96], however, in this section we only review the projects that are closely related to our work. MATLAB (The MathWorks, Natick, MA), although not written in Java, is probably the most widely used commercial application in these areas of study. MathWorks introduced multithreading in MATLAB R2007a, but even in the latest version (R2009a) the usage of multiple threads is limited. In particular, most of the linear algebra algorithms, such as matrix decompositions, are still sequential. This situation will probably change in the next release, due to the fact that the package Parallel Linear Algebra for Scalable Multi-core Architectures (PLASMA) [24] is already available. GPU based computations are available in MATLAB through a third-party toolbox called Jacket [5]. Jacket does not introduce a new API, but instead allows programs written in the native M-Language to be automatically wrapped into a GPU compatible form. Currently Jacket supports only NVIDIA graphic cards and, compared to standard MATLAB, its functionality is very limited (in particular, none of the LAPACK [8] routines are supplied).

JScience [34] is an open source package written by Jean-Marie Dautelle with the ultimate goal to "create synergy between all sciences (e.g. math, physics, sociology, biology, astronomy, economics, etc.) by integrating them into a sin-

gle architecture". It supports multithreaded computations through the real-time programming library Javolution. Current features include modules for measures and units, geographic coordinates, mathematical structures (e.g. group, ring, vector space), linear algebra, symbolic computations, numbers of arbitrary precision, physical models (e.g. standard, relativistic, high-energy, etc.) and currency conversions. Nonetheless, JScience provides almost no support for image processing, and its linear algebra module is very limited, containing only the LU factorization. In addition, there is no class that represents a tensor (e.g., a 3D image array), no matrix sub-ranging, and no FFTs.

Matrix Toolkits for Java (MTJ) [60] is a collection of matrices, linear solvers (direct and iterative), preconditioners, least squares methods and matrix decompositions written by Bjørn-Ove Heimsund. This library is based on BLAS [17] and LAPACK [8] for dense and structured sparse computations and on Templates [13] for unstructured sparse computations. By default JLAPACK [37] is used, but MTJ can be configured to use native BLAS and LAPACK libraries (such as ATLAS [126]). Moreover, the library supports distributed computing via an MPI-like interface. However, MTJ does not supply multithreading, tensors, complex matrices, matrix sub-ranging, and FFTs.

OR-Objects [39] is a collection of 500 Java classes developed by DRA Systems.

It contains packages for linear programming, graph algorithms, matrix and linear algebra, numerical integration, probability and statistics, and geometry. Although OR-Objects is a freeware library, the source code is unavailable, which makes it much less attractive from our point of view. Moreover, analogous to JScience and MTJ, OR-Objects does not provide FFTs, tensors, complex matrices and its multithreaded functionality is limited only to BLAS.

UJMP [11] is a new project that aims to provide classes for storing and processing matrices with interfaces to external data sources (such as databases) and matrix libraries (including Parallel Colt). It allows to handle data that does not fit into main memory (the matrix size can be up to 2^{63} rows or columns). In addition, UJMP supports n-dimensional arrays and visualization methods for matrices. However, this package is also not designed for image processing. In particular, UJMP does not provide any parallel algorithms, complex matrices and Fourier transforms.

Mines Java Toolkit (JTK) [53] is a Java package for science and engineering written by Dave Hale. JTK is implemented in 90% pure Java (OpenGL and LAPACK functionality is provided by Java Native Interface (JNI) wrappers). The toolkit provides many algorithms for digital signal processing, including various local and recursive filters and FFTs, a system for 2D graphics, unstructured

meshes of triangles and tetrahedra, and several optimization algorithms. However, their FFTs are not multithreaded and the size of a 1D transform cannot exceed 720720 (due to the prime-factor FFT algorithm of Temperton [110]). Moreover, Mines Java Toolkit does not support tensors, complex matrices and matrix sub-ranging.

Commons-Math [9] is software developed by the Apache Software Foundation with a goal of providing a library of lightweight, self-contained mathematics and statistics components addressing the most common problems not available in the Java programming language. The current version (2.0) provides extensive functionality for statistics, integration, interpolation, random numbers, linear algebra, optimization, ordinary differential equations, genetic algorithms and FFTs. However, all the algorithms are sequential, there are only 1D Fourier transforms available, and there is no support for tensors.

ojAlgo [92] is a Java library developed by Optimatika, Stockholm Sweden. ojAlgo provides algorithms for linear algebra, optimization, finance, and chart visualization. Some algorithms are multithreaded and the library supports n-dimensional arrays. However, image processing functionality is limited; for example, there are no FFTs and no support for sparse matrices.

JAMA [62] is a basic linear algebra package for Java developed by MathWorks

and National Institute of Standards and Technology (NIST). It only supports real dense matrices Cholesky, LU, QR, SVD and eigenvalue decomposition.

Jampack [51] is a collection of Java classes written by G. W. Stewart and NIST. Jampack fully supports complex matrices, but its functionality is as limited as JAMA library. Besides the five factorizations provided by JAMA, Jampack also supports the Hessenberg form and the Schur Decomposition.

ImageJ [100] is an open source image processing program written in Java by Wayne Rasband at the U.S. National Institutes of Health (NIH). Besides having a large number of options for image editing applications, ImageJ is designed with a pluggable architecture that allows developing custom plugins; over 500 user-written plugins are currently available. Due to this unique feature, ImageJ has become a very popular application among a large and knowledgeable worldwide user community. ImageJ is used as a front-end to Parallel Colt, our image processing engine that overcomes many deficiencies of the libraries outlined above.

3.3 Parallel Colt

Parallel Colt is a reimplementation of Colt [64] with the following goals: (1) provide an open source Java library for high performance scientific computing that

utilizes modern hardware architectures, (2) provide an image processing engine with an emphasis on performance and usability.

3.3.1 Colt

Colt [64] is an open source library for high-performance scientific computing in Java written by Wolfgang Hoschek at CERN. It provides efficient and usable data structures and algorithms for data analysis, linear algebra, multi-dimensional arrays, statistics, histogramming, Monte Carlo simulation and concurrent programming. The project is currently inactive; the latest version (1.2.0) was released in September 2004. We have chosen to adapt Colt to fit our purpose of having a powerful computing engine for image processing. Our choice was motivated primarily by the fact that Colt has support for uniform, versatile and efficient multi-dimensional arrays (matrices) [63]. In particular, *views* operations defined on multi-dimensional arrays allow sub-ranging, striding, transposition, slicing, index flipping, cell selection as well as sorting, permuting and partitioning of the elements. This is almost the same range of functionality as provided by MATLAB. In the rest of this section we summarize all the changes and new functionalities that we introduced in Parallel Colt.

3.3.2 Concurrency

Multithreading in Colt 1.2.0 is limited to a subset of BLAS routines: matrix-matrix and matrix-vector multiplications, as well as the generalized matrix scaling/transform. All other algorithms included in the library are sequential. Moreover, Colt uses Doug Lea's `EDU.oswego.cs.dl.util.concurrent` package for concurrency instead of the improved, more efficient and standardized classes (`java.util.concurrent`) which are included in a standard Java distribution since version 5.0. Concurrency in Colt requires setting a maximum number of threads before the first use, as opposed to Parallel Colt, where multithreading is enabled by default (if the number of available CPUs is greater than one). Java utility classes for concurrent programming contain the *cached thread pool* feature that we have found to be very useful. This type of pool creates new threads as needed, and reuses previously constructed threads when they become available, thereby improving the performance of programs that execute many short-lived asynchronous tasks. Because almost all element-by-element operations and BLAS routines can be split into asynchronous tasks, Parallel Colt uses the *cached thread pool* for low-level concurrency.

3.3.3 Multidimensional Arrays

There are many problems in image processing where double precision is unnecessary. This is usually the case when the source image is saved in a grayscale 8-bit format (integers from 0 to 255). From the computational point of view, single precision has two advantages over double precision: arithmetic operations are faster with single precision numbers and they require only half the storage of double precision numbers. All algorithms in Colt 1.2.0 that use floating-point numbers are implemented in double precision, in particular, only double precision multidimensional arrays are available. Therefore, in Parallel Colt we have added single precision equivalents to all double precision based objects.

In Colt, a single contiguous one-dimensional Java array is used to store elements of all dense 2D and 3D matrices. The elements of 2D matrices are addressed in row-major order and the elements of 3D matrices are addressed in (in decreasing order of significance): slice-major, row-major and column-major order. However, there are two problems with this approach. First, matrix decomposition algorithms typically expect input matrices to use column-major order. Second, since Java array indices must be 32-bit integer values, the 1D array cannot contain more than 2^{31} elements, which is a significant limitation for large-scale problems. To overcome these difficulties, Parallel Colt provides additional data structures for

dense matrices: a 2D dense matrix addressed in column-major order as well as 2D and 3D dense matrices where elements are stored in two-dimensional and three-dimensional Java arrays respectively.

The original Colt project supports three types of 2D sparse matrices: row-compressed, tridiagonal, and the general sparse matrix that uses a hashmap to store the nonzero elements. However, it is beneficial for some applications to use alternative sparse storage schemes. Therefore, in Parallel Colt we have added a column-compressed sparse matrix (for fast column access) and a diagonal matrix (for more efficient computations involving only a single diagonal).

Another new and important type of object added to Parallel Colt is a multi-dimensional array of complex numbers. This object is essential for operations involving FFTs. Because there is no primitive type for complex numbers in Java, we decided to store an array of complex numbers as a one-dimensional array of doubles (or floats), interleaving the real and the imaginary parts. This type of storage guarantees much better performance than defining a new object that represents a complex number, and then storing an array of such objects. Currently Parallel Colt does not support linear algebra algorithms (except matrix-matrix and matrix-vector multiplications) for complex matrices.

In addition to matrices holding floating-point elements, Parallel Colt fully sup-

ports matrices holding integer elements (both 32-bit and 64-bit versions). These type of objects are useful for processing RGB images, where the values of red, green and blue channels are packed into single 32-bit integer values.

Colt is equipped with three different sorting algorithms: quicksort, mergesort and binary search, which complement the `java.util.Arrays` class. Moreover, these algorithms are used to sort elements of multidimensional arrays. In Parallel Colt we have implemented a multithreaded version of quicksort that works both on arrays of primitive types and arrays of objects.

Entirely new functionality added to Parallel Colt concerns the input / output (I/O) operations. We have adapted a matrix / vector reader and writer from MTJ [60]. The classes in `cern.colt.matrix.io` package allow performing I/O operations on matrices and vectors stored in Matrix Market Exchange Formats [87].

Finally, Parallel Colt's implementation of multidimensional arrays includes many additional methods, which are summarized in Table 3.1.

Matrix type	Method
All 1D	reshape
All 2D and 3D	vectorize
All real	getMaxLocation, getMinLocation, getNegativeValues, getPositiveValues, normalize
Dense 1D complex	fft, ifft
Dense 1D real	fft, ifft, getFft, getIfft, dht, idht, dct, idct, dst, idst
Dense 2D complex	fft2, ifft2, fftColumns, ifftColumns, fftRows, ifftRows
Dense 2D real	fft2, ifft2, fftColumns, ifftColumns, fftRows, ifftRows, getFft2, getIfft2, getFftColumns, getIfftColumns, getFftRows, getIfftRows, dht2, idht2, dhtColumns, idhtColumns, dhtRows, idhtRows, dct2, idct2, dctColumns, idctColumns, dctRows, idctRows, dst2, idst2, dstColumns, idstColumns, dstRows, idstRows
Dense 3D complex	fft3, ifft3, fft2Slices, ifft2Slices
Dense 3D real	fft3, ifft3, getFft3, getIfft3, getFft2Slices, getIfft2Slices, dht3, idht3, dht2Slices, idht2Slices, dct3, idct3, dct2Slices, idct2Slices, dst3, idst3, dst2Slices, idst2Slices

Table 3.1: Additional methods in Parallel Colt. In the first column, "All" refers to all supported matrix data types, including single precision (complex and real), double precision (complex and real), 32-bit and 64-bit integers, etc.

3.3.4 Iterative Solvers

Once all types of sparse and dense matrices have been implemented, we have added to Parallel Colt a set of iterative solvers and preconditioners. The following solvers and preconditioners have been adapted from MTJ [60].

Solvers [13] [47]:

- BiConjugate Gradients (BiCG)
- BiConjugate Gradients stabilized (BiCGstab)
- Conjugate Gradients (CG)
- Conjugate Gradients squared (CGS)
- Generalized Minimal Residual using restart (GMRES)
- Iterative Refinement (Richardson's method)
- Quasi-Minimal Residual (QMR)
- Chebyshev iteration

Preconditioners [47] [102] [113]:

- Diagonal (uses the inverse of the diagonal as preconditioner)

- Incomplete Cholesky without fill-in (ICC)
- Incomplete LU without fill-in (ILU)
- Incomplete LU with fill-in (ILUT)
- Symmetrical Sucessive Overrelaxation (SSOR)
- Algebraic Multigrid (AMG)

Besides the above solvers and preconditiones, Parallel Colt also supports the following preconditioned and non-preconditioned solvers for ill-posed inverse problems:

- Hybrid Bidiagonalization Regularization (HyBR)
- Modified Residual Norm Steepest Descent (MRNSD)
- Conjugate Gradient for Least Squares (CGLS)

3.3.5 Linear Algebra

Multithreaded dense linear algebra in Parallel Colt is provided by JPlasma [117], which is our Java port of Parallel Linear Algebra for Scalable Multi-core Architectures (PLASMA) [24]. An important matrix factorization for image processing

applications is the singular value decomposition (SVD), but currently PLASMA does not have support for it. Therefore, Parallel Colt implements two sequential SVD algorithms. One is the original Colt version, which is essentially a slightly modified Jama [62] implementation, and the other is a divide-and-conquer routine from JLAPACK (`dgesdd`). Note that our present use of the SVD in image processing is within a Krylov subspace method that enforces regularization on a (small) projected linear system; see [31].

Besides including JPlasma and JLAPACK in Parallel Colt, we have also added the following dense linear algebra operations: Kronecker product of 1D and 2D matrices (complex and real), Euclidean norm of 2D and 3D matrices computed as a norm of a vector obtained by stacking the columns of the matrix on top of one another, and backward and forward substitution algorithms for 2D real, upper and lower triangular matrices.

Finally, we have implemented and included in Parallel Colt a Java version of the Concise Sparse Matrix Package (CSparse) [35], which we call CSparseJ [116]. Although CSparseJ is not multithreaded, it provides a set of matrix factorizations (LU, Cholesky and QR) that are much more efficient on sparse matrices than their dense equivalents. In the previous version of Parallel Colt, we used the same matrix factorization algorithms both for sparse and dense matrices (sparse

matrices were converted to a dense form).

3.3.6 Trigonometric Transforms

Trigonometric transforms, including the Discrete Fourier Transform (DFT) [33], the Discrete Hartley Transform (DHT) [58], the Discrete Cosine Transform (DCT) [6] and the Discrete Sine Transform (DST) [128], are important tools in image processing applications. To provide trigonometric transform functionality to Parallel Colt, we have integrated a library that we developed for this purpose, called JTransforms [125]. We remark that all transforms are implemented as public methods in 1, 2, and 3-dimensional dense matrices (see Table 3.1). In addition, they can be applied to matrix subranges.

JTransforms is the first, open source, multithreaded FFT library written in pure Java. The code was derived from the General Purpose FFT Package by Ooura [91] and from Java FFTPack [129] by Zhang. Ooura's library is a multithreaded C and Fortran implementation of the split-radix FFT algorithm. In order to provide more portability both POSIX threads and Windows threads are used in the implementation. Moreover, the code is highly optimized and in some cases runs faster than FFTW [42]. Even so, the package has several limitations arising from the split-radix algorithm. First, the size of the input data has to be a power-of-two in-

teger. Second, the number of computational threads must also be a power-of-two. Finally, one-dimensional transforms can only use two or four threads. To overcome the power-of-two limitation we have adapted Zhang's Java code which is a straightforward translation of the mixed-radix algorithm from FFTPACK [109]. Since Java FFTPack contains only sequential algorithms for 1D transforms (real and complex), we have implemented multithreaded 2D and 3D transforms. In the case of 1D transforms when the size of the vector is not a power-of-two, JTransforms uses a sequential implementation. However this limitation does not affect the performance of multidimensional transforms because threads are used at higher levels. As a result, the current version of JTransforms can be used for arbitrarily sized data.

There are some important distinctions between our Java code and Ooura's C implementation. First, JTransforms uses a thread pool, while the original package does not. Although thread pooling with POSIX threads is possible, there is no code for this mechanism available in the standard library, and therefore many multithreaded applications written in C do not use thread pools. This has the added problem of causing overhead costs of creating and destroying threads every time they are used. Another difference between our code and Ooura's FFT is the use of *automatic multithreading*. In JTransforms (and in Parallel Colt), threads

are used automatically when computations are done on a machine with multiple CPUs. Conversely, both Ooura's FFT and FFTW require manually setting up the maximum number of computational threads. Lastly, JTransforms' API is much simpler than Ooura's FFT, or even FFTW, since it is only necessary to specify the size of the input data; work arrays are allocated automatically and there is no planning phase.

3.3.7 Accuracy

There are two aspects about the accuracy of floating-point arithmetic in Java. The first is related to the internal design and implementation of Java's floating-point arithmetic. There are several flaws in this implementation [69]. First of all, Java does not completely conform to the IEEE 754 standard, since it does not support the flags for IEEE 754 exceptions: Invalid Operation, Overflow, Division-by-Zero, Underflow, Inexact Result. In other words, no event occurs when the value of a floating-point number becomes either Infinity or NaN. Moreover, Java does not provide capability to work with the IEEE extended precision, even though over 95% of today's computers have hardware that can support these types of numbers. Finally, of two traditional policies for mixed precision evaluation, Java chose the worse. However, our experience shows that Java's floating-point arith-

metic is good enough for applications in image processing. This is supported by the fact that usually the pixels of an image are stored as integers (byte and short) or as a single-precision floats, thus the double (or even single) precision arithmetic provides a sufficient amount of accuracy.

Another aspect of the accuracy is related to the stability of an algorithm and round-off errors. In the previous release of Parallel Colt we observed inaccurate results for trigonometric transforms when the size of the input data was an integer with a large prime factor. The inaccurate results were caused by the mixed-radix FFT algorithm. When encountering a large prime factor, a slow, $\mathcal{O}(n^2)$, discrete Fourier transform algorithm was used. It is known [103], however, that in these situations the root mean square error is $\mathcal{O}(\sqrt{n})$, where n is the size the input data. The original FFTPACK library is also burdened with this error. In the current version of Parallel Colt (and JTransforms) we have fixed all the accuracy issues by implementing Bluestein's FFT algorithm [19]. Figures 3.1 and 3.2 show that both single and double precision FFTs in Parallel Colt are as accurate as FFTs in MATLAB. Jacket's FFTs, on the other hand, are much less accurate when the size of the data is a prime number. Since the source code of CUFFT library (used by Jacket) is not available, we can only speculate that the Jacket's accuracy problems are also caused by the mixed-radix FFT algorithm. The length of the vertical error

bars in these figures is equal to two standard deviation units.

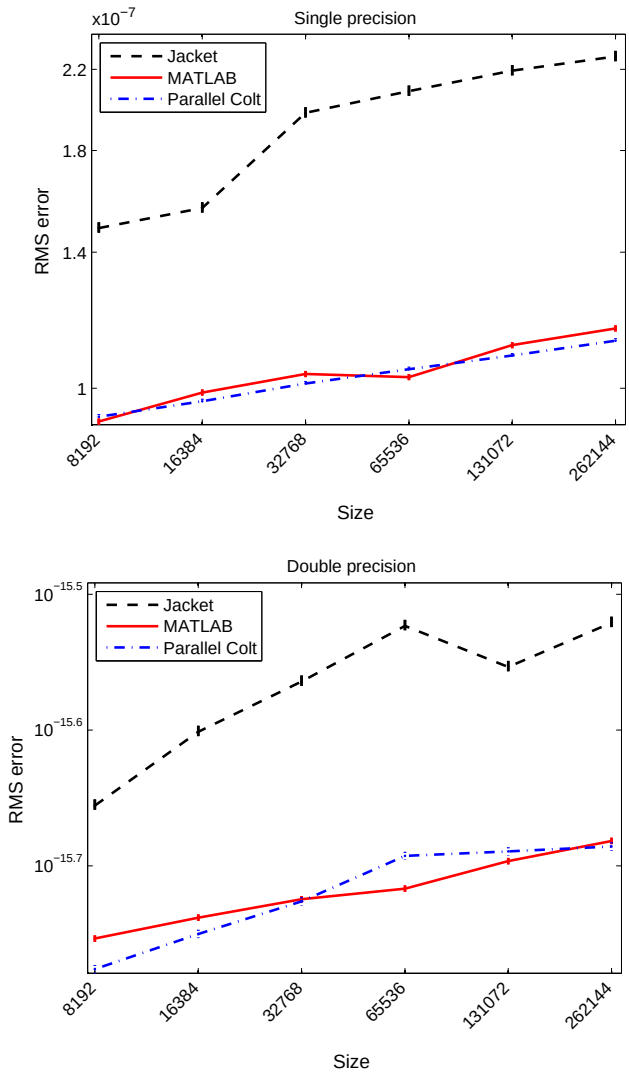

Figure 3.1: Accuracy of complex, 1D FFT (power of two sizes). The vertical axis is the root mean square error, $\frac{\|\mathbf{x} - \text{ifft}(\text{fft}(\mathbf{x}))\|_2}{\sqrt{n}}$, where $\mathbf{x}$ is a vector whose size n is shown on the horizontal axis.

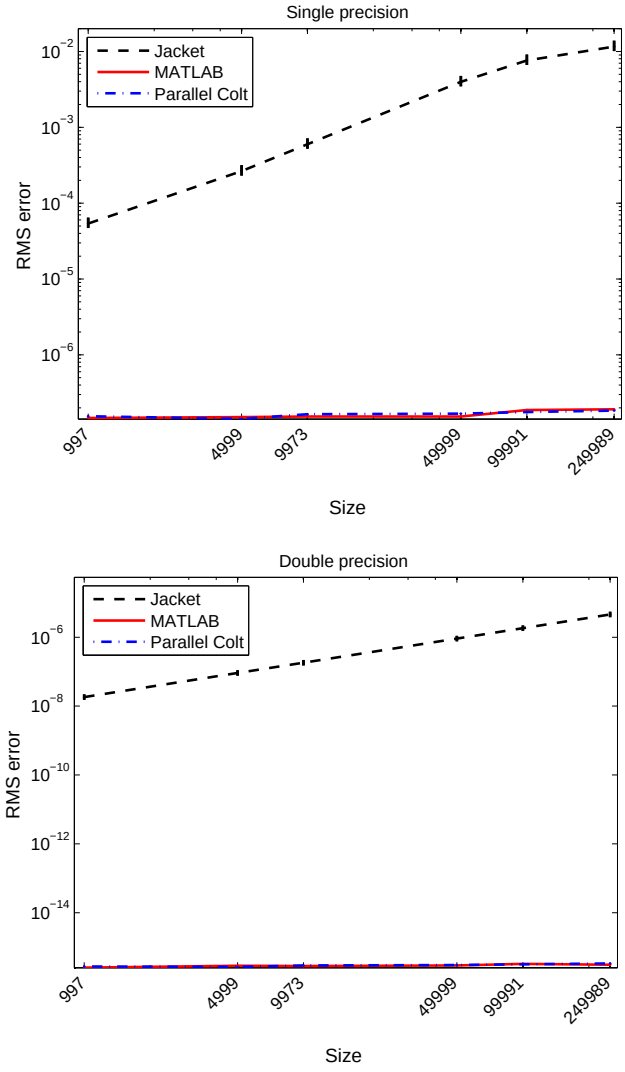

Figure 3.2: Accuracy of complex, 1D FFT (prime sizes). The vertical axis is the root mean square error, $\frac{\|\mathbf{x} - \text{ifft}(\text{fft}(\mathbf{x}))\|_2}{\sqrt{n}}$, where $\mathbf{x}$ is a vector whose size n is shown on the horizontal axis.

3.3.8 Other Additions

To support new kinds of matrices described in Section 3.3.3 we had to implement some new data structures. In particular, we have added new hashmaps holding (key,value) associations of type (long -> double), (double -> long), (long -> float), (float -> long), (long -> long), (int -> long) and (long -> int). In addition, since the HyBR solver requires the 1-dimensional minimization routine (fmin), we have included the nonlinear optimization package [114] into our library.

Unit testing is an important part of every library, but it is absolutely crucial for mathematical software. Therefore, Parallel Colt contains a unit test framework based on JUnit [3]. The framework allows to write and run new test cases in a very intuitive way. We currently have available 6552 tests to check all the functionalities provided by sparse and dense matrices and iterative solvers.

Finally, a highly configurable benchmark framework for Parallel Colt has been developed. At this time we provide benchmarks for dense matrices (holding complex and real numbers) as well as for iterative solvers. In the configuration files, the user can define such properties as the number of threads, the size of the matrix, the number of repeats (to compute average time), and the path to the matrix file (stored in Matrix Market Exchange Formats). The timings computed by these benchmarks are automatically saved in text files.

3.3.9 Examples of Usage

Table 3.2 shows eight examples of different operations in MATLAB and in Parallel Colt. Since Java is a statically typed language, all variable names (along with their types) must be explicitly declared. MATLAB, on the other hand, is a dynamically typed language so there is no need to declare anything. An assignment statement binds a name to an object of any type and later the same name may be assigned to an object of a different type. This feature makes MATLAB expressions generally much more concise than the corresponding expressions in Java. Another essential difference between MATLAB and Parallel Colt arises from the inability to do operator overloading in Java (compare the matrix times vector expressions). Aside from these two differences, the expressions in Table 3.2 show that the same level of abstraction is used in MATLAB and Parallel Colt.

Description	MATLAB	Parallel Colt
Copy of A	B = A;	DoubleMatrix2D B = A.copy();
Transpose of A	B = A';	DoubleMatrix2D B = A.viewDice();
Matrix times vector	B = A*x;	DoubleMatrix2D B = A.zMult(x);
2D FFT of A	B = fft2(A);	DComplexMatrix2D B = A.getFft2();
FFT along columns of A	B = fft(A,2);	DComplexMatrix2D B = A.getFftColumns();
Cosine of A (in-place)	A = cos(A);	A.assign(DoubleFunctions.cos);
Sum all entries of A	s = sum(A(:));	double s = A.zSum();
Location of max of A	[i, j] = find(A == max(A(:)));	double[] max = A.getMaxLocation();

Table 3.2: Comparison of MATLAB and Parallel Colt expressions for a sample set of matrix operations.

3.3.10 Benchmarks

In Section 3.3.8 we remarked that Parallel Colt is equipped with framework for performing benchmarks. Here we present benchmark results of two important computational kernels used in deconvolution and super-resolution algorithms: FFT and sparse matrix-vector product.

FFT

For many image deblurring problems, matrix-vector multiplications are done most efficiently with FFTs. Thus, FFTs can be considered a key computational kernel in spectral and iterative image deblurring algorithms. In this section we benchmark the performance of 2D real FFTs. As a testbed for our benchmarks we used a machine equipped with two Quad-Core Intel Xeon E5472 processors operating at 3.0 GHz, 32GB RAM, and an NVIDIA Tesla C1060. The theoretical peak performance of these processors is equal to 12 Gflops (4 floating-point operations per cycle due to SSE extensions) per core or 96 Gflops for the whole machine. The system was running Ubuntu Linux 9.04 (64-bit), NVIDIA CUDA 2.3, MATLAB R2009b, AccelerEyes Jacket 1.2, Sun Java 1.6.0_16 (64-Bit Server VM) and ImageJ 1.43h. The following Java options were used: `-d64 -server -Xms10g -Xmx10g -XX:+UseParallelGC -XX:ParallelGCThreads=1`.

We benchmarked single and double precision, real input 2D FFTs in native MATLAB, MATLAB with Jacket, and Parallel Colt. The benchmarking methodology was adapted from FFTW [43]. First, we run the *warm up* phase (the first two calls require more time) which is not incorporated into the results. Then, we measured the FFT performance by performing repeated FFTs (100 times) of the same zero-initialized array. To compute the performance in Gflops we used the

following formula

$$\text{Gflops} = 2.5 \cdot N \cdot \log_2(N)/(\text{wall-clock time in nanoseconds}),$$

where N is number of data points (the product of the FFT dimensions). This formula does not count the flops accurately; it is however a convenient scaling, based on the fact that the radix-2 Cooley-Tukey algorithm asymptotically requires $2.5 \cdot N \cdot \log_2(N)$ floating-point operations. Depending on the size of the input data, different FFT algorithms (with different computational complexity) are used both in MATLAB and Parallel Colt. Therefore, especially in MATLAB , it is hard to predict which algorithm will be used on the given machine and with the given input data. Benchmark results for Jacket include the time required for data transfer to and from the GPU memory. The maximum 8 threads were used in MATLAB and Parallel Colt. In addition, for Parallel Colt, one thread was used for the garbage collector, by specifying the flag `-XX:ParallelGCThreads=1`. It should be noted that the amount of GPU memory is a serious limitation for large-scale problems. On the hardware available for the tests reported here, the largest matrix size that fit into the GPU memory was 8192×8192 (4096×4096 for double precision). The benchmark results are reported in Tables 3.3 and 3.4. The reader should only compare the difference in performance between MATLAB , Jacket and Parallel Colt and not among the different sizes for the same library

(we are aware of the fact that the formula used for computing Gflops undercounts the FFT work when N is not a power-of-two). The performance of all three libraries is comparable, with native MATLAB having some advantages for smaller data sizes. It should be emphasized that multithreaded FFTs were introduced in MATLAB R2009a; the performance of these routines in the previous versions was significantly lower. The other important conclusion that can be drawn from these results is that although the GPU-based single precision FFTs outperform both native MATLAB and Parallel Colt algorithms in most cases, they are much less accurate when the matrix dimensions are not a power-of-two numbers (see Figure 3.2).

Size	**MATLAB**	**Jacket**	**Parallel Colt**
2000×2000	4.50	3.43	1.89
2048×2048	1.82	3.64	2.22
4000×4000	4.73	3.21	3.10
4096×4096	1.85	3.38	2.66
8000×8000	2.21	3.26	2.59
8192×8192	1.92	3.53	2.55
16000×16000	1.73	-	1.75
16384×16384	1.85	-	2.50

Table 3.3: Performance in Gflops for single precision, real input 2D FFT.

Size	MATLAB	Jacket	Parallel Colt
2000×2000	2.76	1.40	1.61
2048×2048	1.12	1.72	1.88
4000×4000	2.18	1.39	2.05
4096×4096	1.15	1.87	2.08
8000×8000	1.24	-	1.34
8192×8192	1.13	-	1.81
16000×16000	1.16	-	1.24
16384×16384	1.10	-	1.64

Table 3.4: Performance in Gflops for double precision, real input 2D FFT.

Sparse Matrix-Vector Product

Sparse matrix-vector product is a key operation for all iterative solvers. Here we present the performance comparison of this operation in MATLAB and Parallel Colt. MATLAB uses compressed-column format for storing sparse matrices. In Parallel Colt both compressed-column and compressed-row formats are available, but in this benchmark we only tested the former one. It should be emphasized that the latest release of MATLAB, R2009a, does not support single precision sparse matrices and its sparse matrix-vector product implementation is sequential.

Parallel Colt, on the other hand, supports multithreaded sparse matrix-vector operations. However, there are two restrictions. When $\mathbf{A}$ is in compressed-column format, at most two threads are used to compute $\mathbf{y} = \mathbf{Ax}$, and when $\mathbf{A}$ is in compressed-row format, at most two threads are used to compute $\mathbf{y} = \mathbf{A}^T\mathbf{x}$. These limitations arise from the fact that it is not possible to split the job for these particular computations (with the associated storage format) into asynchronous tasks; all threads have to operate on the whole vector $\mathbf{y}$. Therefore, in our implementation, if one of these situations occur and two threads are being used, then the first thread operates on the output vector $\mathbf{y}$ and the second thread works on its local copy of vector $\mathbf{y}$. A reducing addition is performed at the end of the multiplication. Our experiments have shown that using more than two threads for these cases slows down the performance.

The machine described in the previous section (with the same Java options) was also used for this benchmark. To compute the performance in Mflops we used the following formula

$$\text{Mflops} = 2 \cdot nnz/(\text{wall-clock time in microseconds}),$$

where nnz denotes the number of nonzero elements in a matrix. Table 3.5 shows the performance of two operations $\mathbf{y} = \mathbf{Ax}$ and $\mathbf{y} = \mathbf{A}^T\mathbf{x}$, where $\mathbf{A}$ is a sparse matrix and $\mathbf{x}$ is a dense vector. Four different matrices were used in this test.

A random matrix (the first row in Table 3.5) was generated using the MATLAB command `A=sprand(1e6,1e6,1e-7)`, and the other three matrices come from the University of Florida Sparse Matrix Collection [36]. It can be seen that Parallel Colt outperforms MATLAB for all tested matrices. This difference is much more significant for the transpose operation, where Parallel Colt uses multiple threads.

Matrix	**Nonzeroes**	**MATLAB**	**Parallel Colt (double)**	**Parallel Colt (single)**
Random	100,000	14.1 (17.8)	14.7 (30.4)	20.6 (46.3)
Rajat31	20,316,253	264.3 (300.0)	356.6 (669.5)	451.5 (1048.0)
Nlpkkt120	95,117,792	349.4 (366.7)	719.0 (1061.3)	1247.9 (1668.8)
S3dkq4m2	2,455,670	402.9 (417.7)	668.8 (1094.3)	1200.9 (1711.9)

Table 3.5: Performance in Mflops for the sparse matrix-vector multiplications $\mathbf{y} = \mathbf{A}\mathbf{x}$ and $\mathbf{y} = \mathbf{A}^T\mathbf{x}$ (numbers in brackets show the performance of $\mathbf{y} = \mathbf{A}^T\mathbf{x}$).

Chapter 4

Implementation

This chapter describes the implementation of various image processing algorithms as plugins for ImageJ. Parallel Colt is used as a computational engine for all implemented algorithms and all the plugins support a batch mode, so in particular, they can be called from an ImageJ macro.

4.1 Parallel Spectral Deconvolution

Parallel Spectral Deconvolution [123, 125] is an ImageJ plugin for spectral image deblurring. The code is based on methods described in [57]. The graphical user interface (GUI) for the current version (1.11) is illustrated in Figure 4.1.

4.1.1 Description and Usage

Parallel Spectral Deconvolution implements Tikhonov- and TSVD-based image deblurring assuming either periodic of reflexive boundary conditions. Although the plugin can handle arbitrary-sized 2- and 3-dimensional images, its usage is limited to grayscale images. To deconvolve a color image, the user would have to split the channels and deblur each channel separately.

There are seven drop-down lists (combo-boxes) available in the plugin's GUI. From the *Image* list, the user can choose a blurred image. *PSF* list is for selection of a point spread function image. The content of these two lists depends on what is currently open in ImageJ - if no image windows are displayed, then both lists are empty. The next two lists (*Method* and *Stencil*) allow the user to choose an algorithm used for deconvolution (*Generalized Tikhonov*, *Tikhonov* or *TSVD*) and a stencil (for *Generalized Tikhonov* only). The stencil is used for creating a regularization matrix (an approximation of a derivative operator) and by default the *Laplacian* matrix is used. The *Resizing* combo-box is used to choose whether or not to resize (pad) the blurred image to the next power-of-two size before processing. This feature is available mainly for performance reasons (FFTs are computed faster when the size of the data is a power-of-two number). Note that if the size of each dimension of a blurred image is already a power-of-two number, then the

image will not be padded even if the *Next power of two* option is selected. To display a padded image, the *Show padded image* check-box needs to be selected. The *Output* list allows to specify the type of the reconstructed image (i.e. number of bits per pixel). Finally, in the *Precision* combo-box the user can choose a floating-point precision used in computations. Practice shows that single precision is sufficient for most problems.

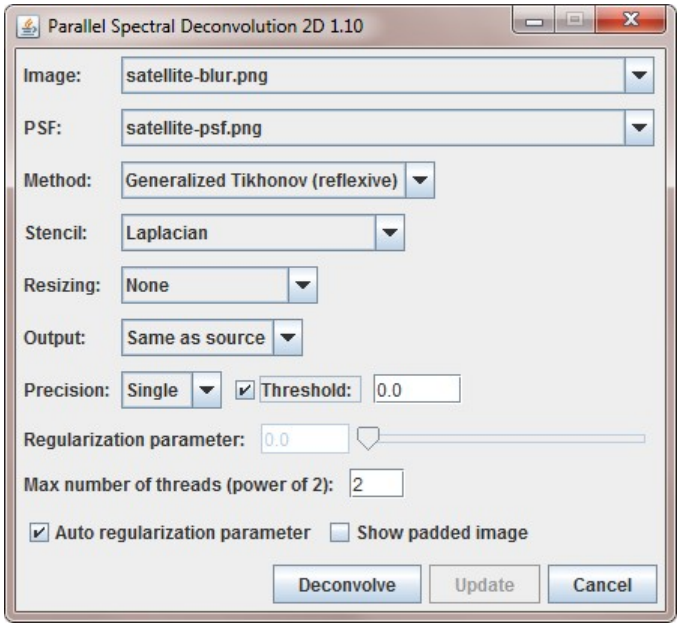

Figure 4.1: Parallel Spectral Deconvolution GUI.

There are a few other important options in the GUI that require some explanation. The *Threshold* check-box and text field are used to remove negative values from the reconstructed image. Since it is not possible to impose nonnegativity

constraints in the spectral algorithms, the threshold option is the only way to get a nonnegative solution. If the threshold option is enabled, then all values in the reconstructed image that are less than the value specified in the threshold text field are replaced by zero. In the *Max number of threads (power of 2)* text field the user can specify how many computational threads will be used. By default this value is equal to the number of CPUs available on the given machine.

Parallel Spectral Deconvolution has an option to automatically compute the value of a regularization parameter (*Auto regularization parameter* check-box). If this box is selected, then the generalized cross-validation (GCV) algorithm is used. Since GCV may fail to find an optimal value of a regularization parameter, it is possible to manually adjust the automatically computed value (*Regularization parameter* text field and slider). Once the initial deconvolution is finished, the *Regularization parameter* text field and slider, as well as the *Update* button are enabled. At this point, the user can change the value of a regularization parameter either by using the slider or by entering a new value in the text field. The *Update* button is used to recompute the solution with the new value of a regularization parameter. This functionality allows to save computational time, because most of the objects that are already in memory do not need to be reevaluated (only the new filter factors and an inverse FFT have to be computed).

4.1.2 Benchmark

We have compared the performance and the quality of reconstruction of Parallel Spectral Deconvolution with another ImageJ plugin called DeconvolutionJ. DeconvolutionJ [75] is a plugin written by Nick Linnenbrügger that implements spectral deconvolution based on the Regularized Wiener Filter [49]. The plugin has a number of limitations. It can handle arbitrary-sized 2- and 3-dimensional images, although it requires the PSF image to be the same size as the blurred image, and it must be centered in the field of view. In addition, the regularization parameter of the Wiener filter must be specified manually and there is no update option to efficiently deblur the same image with different values of the regularization parameter. Last, but not least, DeconvolutionJ is a serial implementation, and therefore cannot take advantage of modern multi-core processors.

To benchmark the plugins we have generated two test images and then we have run the software on the machine described in Section 3.3.10. Figure 4.2 shows the true 2D image (the picture of Ed White performing the first U.S. spacewalk in 1965 [85]) as well as the blurred and reconstructed data. The true image has 4096×4096 pixels. The blurred image was generated by reflexive padding of the true data to size 6144×6144, convolving it with Gaussian blur PSF (standard deviation = 20), adding 1% white noise and then cropping the resulting image

to the size of 4096×4096 pixels. To better illustrate the quality of deblurring, we display a small region of the blurred and reconstructed images. In Parallel Spectral Deconvolution (denoted by PSD in Figure 4.2), we used Tikhonov regularization with reflexive boundary conditions and regularization parameter equal 0.004. Similarly, in DeconvolutionJ, we used no resizing (the image size was already a power-of-two), double precision for complex numbers and the same value for the regularization parameter.

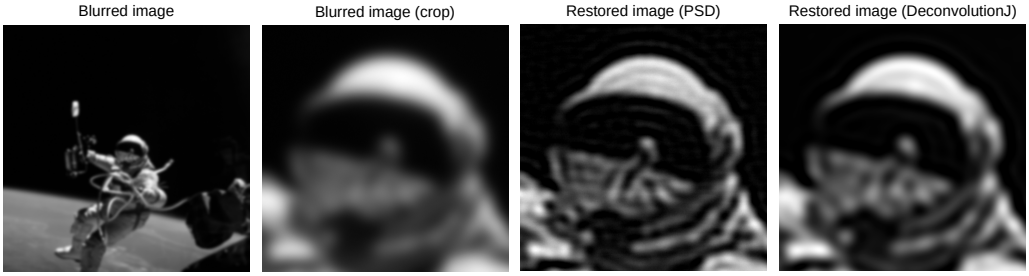

Figure 4.2: Astronaut image: blurred and restored data.

Table 4.1 presents average execution times among 10 calls of each method. All timings are given in seconds and the numbers in brackets include the computation of the regularization parameter. One should notice a significant speedup, especially from 1 to 2 threads. The last row in Table 4.1 shows the execution time for DeconvolutionJ, which is over 7 times greater than the worst case of Parallel Spectral Deconvolution (Generalized Tikhonov, 1 thread) and over 30 times

greater than the best case of Parallel Spectral Deconvolution (TSVD, 8 threads).

Method	**1 thread**	**2 threads**	**4 threads**	**8 threads**
TSVD	6.2 (17.0)	3.9 (11.3)	2.8 (9.4)	2.0 (8.0)
Tikhonov	6.7 (19.1)	4.3 (12.0)	3.1 (9.3)	2.1 (8.5)
Generalized Tikhonov	8.3 (18.0)	5.1 (12.1)	3.7 (9.5)	2.8 (6.9)
DeconvolutionJ	61.01	-	-	-

Table 4.1: Average execution times (in seconds) for 2D spectral deblurring (numbers in brackets include the computation of the regularization parameter).

For 3D deblurring, the test image (see Figure 4.3) was a T1 weighted MRI image of Jeff Orchard's head [93]. The size of this image was equal $128 \times 256 \times 256$ pixels. The blurred image was generated by zero padding of the true data to size $128 \times 512 \times 512$, convolving it with a Gaussian blur PSF (standard deviation = 1), adding 1% white noise and then cropping the resulting image to the size of $128 \times 256 \times 256$ pixels. Figure 4.3 shows the 63^{rd} slice of the blurred and restored data. In Parallel Spectral Deconvolution, we used the reflexive boundary conditions and regularization parameter equal 0.02. In DeconvolutionJ, we used exactly the same parameters as for the 2D benchmark and 0.01 for the regularization parameter.

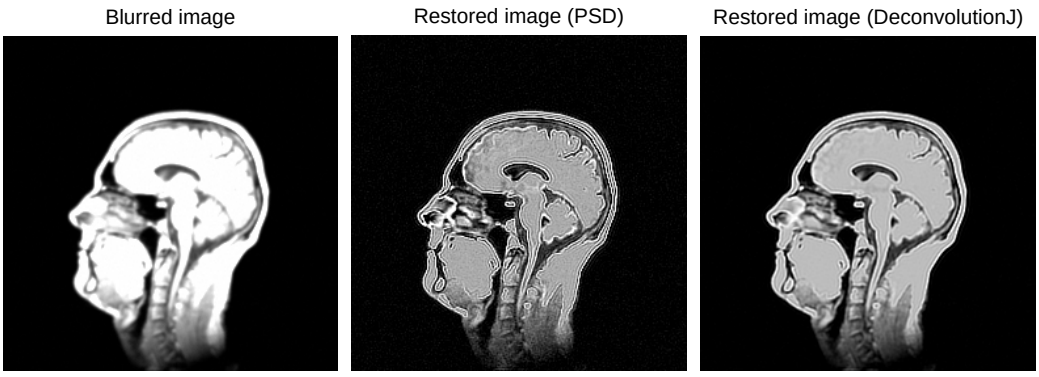

Figure 4.3: Head image (63^{rd} slice): blurred and restored data.

In Table 4.2, we have collected all timings. Once again, the execution time for DeconvolutionJ is almost 6 times greater than the worst case of Parallel Spectral Deconvolution (Generalized Tikhonov, 1 thread) and over 27 times greater than the best case of Parallel Spectral Deconvolution (TSVD, 8 threads).

Method	**1 thread**	**2 threads**	**4 threads**	**8 threads**
TSVD	3.2 (6.6)	1.9 (4.5)	1.2 (3.1)	0.9 (2.6)
Tikhonov	3.8 (12.2)	2.3 (8.4)	1.5 (5.3)	1.1 (3.8)
Generalized Tikhonov	4.0 (15.9)	2.6 (9.1)	1.6 (5.9)	1.2 (4.7)
DeconvolutionJ	23.45	-	-	-

Table 4.2: Average execution times (in seconds) for 3D spectral deblurring (numbers in brackets include the computation of the regularization parameter).

4.2 Parallel Iterative Deconvolution

Parallel Iterative Deconvolution [122] is an ImageJ plugin we developed for iterative image deblurring. The code is based on the RestoreTools MATLAB toolbox and on Iterative Deconvolve 3D plugin. The GUI for the current version (1.11) is illustrated in Figure 4.4.

4.2.1 Description and Usage

We start the discussion about the implementation of iterative deconvolution by reviewing two packages related to our work.

MATLAB's Image Processing Toolbox contains some methods for image restoration, but these have several limitations. For example, they cannot be used with spatially variant blurs. The RestoreTools [83] package contains several additional, modern algorithms which have been studied in the inverse problems and numerical analysis literature. In addition, the toolbox can be easily used for 2D and 3D images and its object oriented design allows users to incorporate efficient computational kernels in their own algorithms. The package includes iterative methods for unsymmetric (CGLS [16], HyBR [31]) and symmetric blurs (MR2 [54]), as well as an algorithm that enforces nonnegativity constraints (MRNSD [84]). To

accelerate convergence of iterative methods, preconditioners are provided (with automatic choice of certain tolerances) based on FFTs, DCTs and the SVD. All the algorithms work for both spatially invariant as well as spatially variant blurs. Moreover, three types of boundary conditions (zero, periodic and reflexive) can be used in a reconstruction. There are two limitations in the current release: no graphical user interface and no support for color images.

Iterative Deconvolve 3D [38] is an ImageJ plugin written by Robert Dougherty, OptiNav Inc (the GUI is shown in Figure 4.5). Whereas RestoreTools, and our Parallel Iterative Deconvolution package provide a variety of tools and algorithms for image deblurring, Iterative Deconvolve 3D contains only a single method: a nonnegatively constrained Landweber iteration [14], where a regularized Wiener filter is used as a preconditioner. Besides the fact that the code is sequential, this plugin has two limitations. First of all, it requires a PSF image to be centered in the field of view. Moreover, it uses a Discrete Hartley Transform (DHT) that works only when the size of the data is a power-of-two number. This means that a blurred image and a PSF need to be padded to the next power-of-two size before processing. When the FFT or DHT are used for image deblurring, padding is almost always necessary (to avoid ringing artifacts), but it is enough to pad each side of a blurred image with an amount that is only half of the size of the PSF

image. This property is not exploited in Iterative Deconvolve 3D. Instead, the size of the PSF is disregarded and the blurred image is always padded to the next power-of-two size that is at least 1.5 times larger than the original image. Since usually a blurred image is much larger than the PSF image, this type of padding results not only in very poor performance but it also requires much more memory.

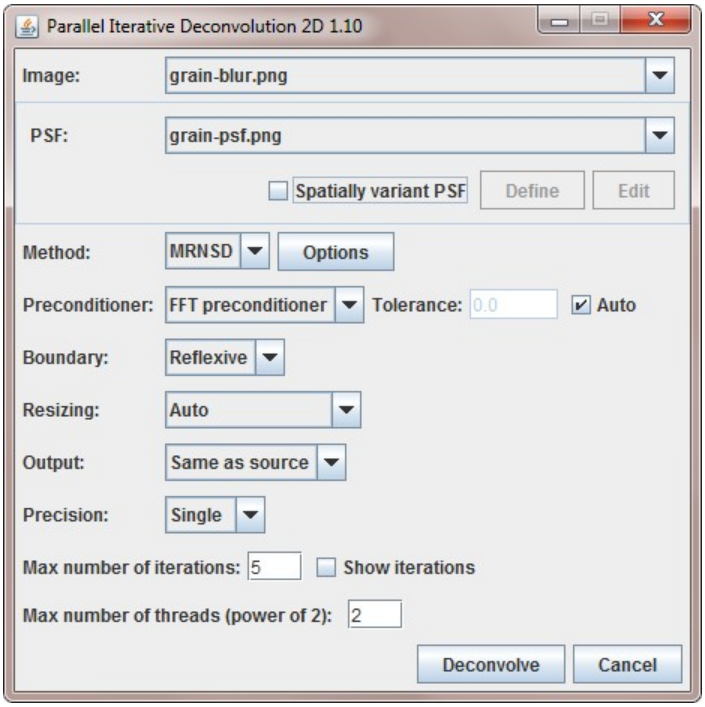

Figure 4.4: Parallel Iterative Deconvolution GUI.

Our Parallel Iterative Deconvolution plugin implements several methods that can be used for image deblurring, including MRNSD, CGLS, HyBR and Landweber algorithms. The first three methods are derived from RestoreTools, and the

Landweber algorithm is a parallel version of Iterative Deconvolve 3D with some enhancements. In particular, we have fixed the two aforementioned limitations of the original plugin. Similarly to Parallel Spectral Deconvolution the usage of the plugin is limited to grayscale images.

There are eight drop-down lists (combo-boxes) available in the plugins's GUI. The first two lists (*Blurred image* and *PSF*) have exactly the same functions as in the spectral deconvolution plugin. The next two lists (*Method* and *Preconditioner*) allow to select an algorithm used for deconvolution (*MRNSD*, *WPL*, *CGLS*, *HyBR*) and a preconditioner. Currently only the *FFT-preconditioner* is available (*WPL* uses the Wiener Filter as a preconditioner). The tolerance for the preconditioner is computed automatically (via Generalized Cross Validation) by default (*Auto* check-box), but it is also possible to specify the value manually. In the *Boundary* combo-box the user can choose from three types of boundary conditions: *Reflexive*, *Periodic* or *Zero*. The first type, reflexive, is usually the best choice. The *Resizing* combo-box allows to specify how the blurred image will be padded before processing. *Minimal* resizing means that the pixel data in each dimension of a blurred image are padded by the size of the corresponding dimension of a PSF image. If the *Next power of two* option is selected, then the pixel data in each dimension of a blurred image are padded to the next power-of-two size that is

greater or equal to the size of an image obtained by minimal padding. Finally, the *Auto* option chooses between the two other options to maximize the performance. The *Output* and *Precision* combo-boxes have again the same functionality as in the case of Parallel Spectral Deconvolution.

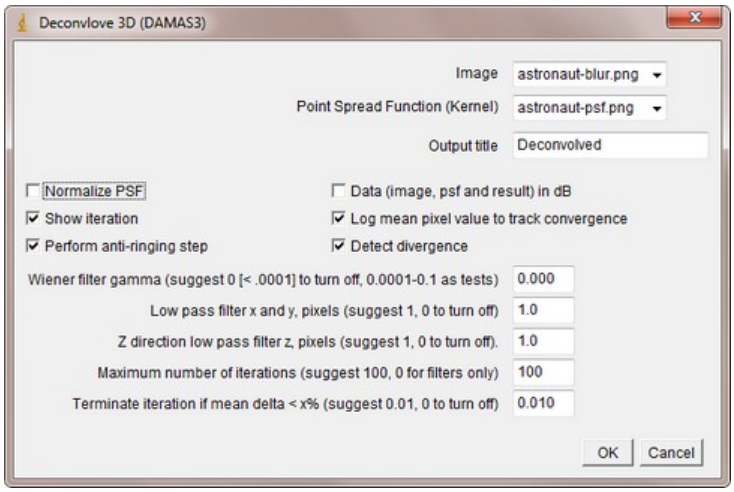

Figure 4.5: Iterative Deconvolve 3D GUI.

There are a few other elements available in the GUI. The *Options* button (next to the *Method* combo-box) is used to display a dialog window with advanced preferences for each algorithm (all of these options are described in the following subsections). In a typical usage scenario, there is no need to change the advanced preferences, since the default values are usually optimal. In contrast, Dougherty's Iterative Deconvolve 3D GUI (Figure 4.5) shows all available options in one win-

dow which may discourage less experienced users. The *Max number of iterations* text field is used to specify how many iterations a given method should perform. It is a maximal value, which means that the process of reconstruction may stop earlier (when the stopping criterion is met). If the *Show iterations* check-box is selected, then the reconstructed image will be displayed after each iteration. Finally, in the *Max number of threads (power of 2)* text field the user can enter how many computational threads will be used. We now describe the available advanced features provided by the *Options* button.

MRNSD and CGLS Options

MRNSD has only three advanced properties (see Figure 4.6). The *Stopping tolerance* text field allows to manually specify the value that will be used as a stopping criterion. By default that value is computed automatically. When the *Threshold* option is enabled, then all values in the reconstructed image that are less than the value specified in the threshold text field are replaced by zero. However, since MRNSD is a nonnegatively constrained algorithm, this option is not very useful and is disabled by default. Finally, selecting *Log convergence* has the effect of displaying the convergence progress in a separate Log window.

The CGLS options panel looks exactly the same as the MRNSD options panel.

The only difference is that the *Threshold* option is enabled by default, since it is not a nonnegativity constrained method.

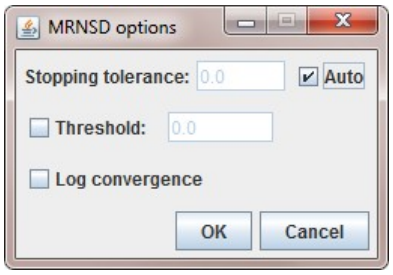

Figure 4.6: MRNSD options panel.

WPL Options

WPL (see Figure 4.7), similarly to MRNSD, is a nonnegatively constrained algorithm, therefore the *Threshold* option is disabled by default. Moreover, the *Log mean pixel value to track convergence* has the same functionality as in the case of MRNSD - the convergence history is displayed in the separate Log window. If *Normalize PSF* is selected, then the point spread function is normalized before processing. To reduce artifacts from features near the boundary of the imaging volume the user should use the *Perform anti-ringing step* option. The *Detect divergence* property stops the iteration if the changes appear to be increasing. For WPL, inputs in decibels are permitted (*Data (image, psf and result) in dB*). This is uncommon in optical image processing, but is the norm in acoustics. The *Wiener*

filter gamma defines the tolerance for the preconditioner. Setting this parameter to zero turns off the preconditioner. The *Low pass filter x and y* settings, in pixels, provide a way to smooth the results and accelerate convergence. Zero should be chosen to disable this function. Finally, the *Terminate iteration if mean delta less than x%* is used as a stopping criterion.

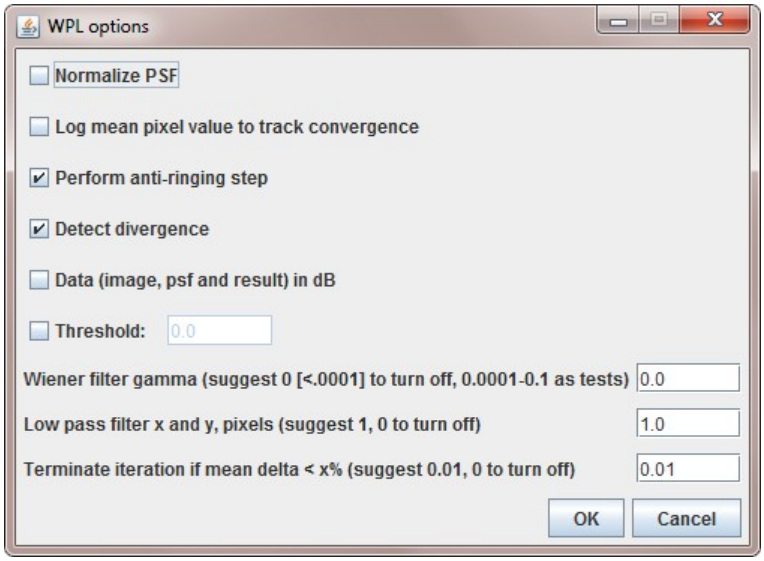

Figure 4.7: WPL options panel.

HyBR Options

In the HyBR options panel (see Figure 4.8) the properties relevant to regularization are grouped in the box called *Regularization options*. The *Method* combo-box allows to decide how the regularization parameter will be computed. If the user

selects *None*, then the value of the parameter has to entered in the *Parameter* text field. When WGCV (Weighted Generalized Cross-Validation) is chosen, then the user has to specify the weight (*Omega*) manually. In the *Begin regularization after this iteration* text field the user can decide after which iteration the regularization will begin. Before that iteration the QR factorization is used to solve the least squares problem on the projected subspace at each iteration.

In addition to regularization properties, it is possible to adjust five other options. In the *Inner solver* combo-box the user can choose the solver that will be used at each iteration. Currently it can be either *Tikhonov* or *None*. If *None* is selected as an inner solver, then the QR factorization is used to solve the projected least squares problem. In the *Stopping tolerance* text field the user can enter a value that will be used to detect flatness in the GCV curve as a stopping criterion. If *Reorthogonalize Lanczos subspaces* is selected, then during the process of Lanczos bidiagonalization, the subspaces will be reorthogonalized. This process requires more memory and usually does not improve the quality of reconstruction, so by default it is disabled. The *Threshold* check-box should be selected, because HyBR does not compute a nonnegative solution. The *Log Convergence* has the same functionality as for all other methods.

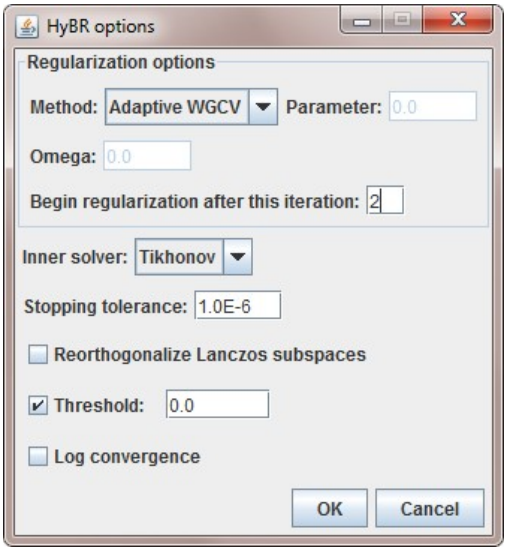

Figure 4.8: HyBR options panel.

Spatially Variant PSF

There are three elements in the Parallel Iterative Deconvolution's GUI that have not been described above, namely: *Spatially variant PSF* check-box, *Define* and *Edit* buttons. These elements allow to work with spatially variant PSFs (i.e. if there are multiple PSF images associated with a single blurred image). Figure 4.9 shows two dialog windows that appear when either the *Define* or *Edit* button are clicked. In the *Create Spatially Variant PSF* panel the user can specify the number of PSFs in the form of 2D (or 3D) matrices. Then, after clicking the *OK* button, the *Edit Spatially Variant PSF* panel will appear. This dialog contains a grid of

buttons that are used to enter paths to the PSF files.

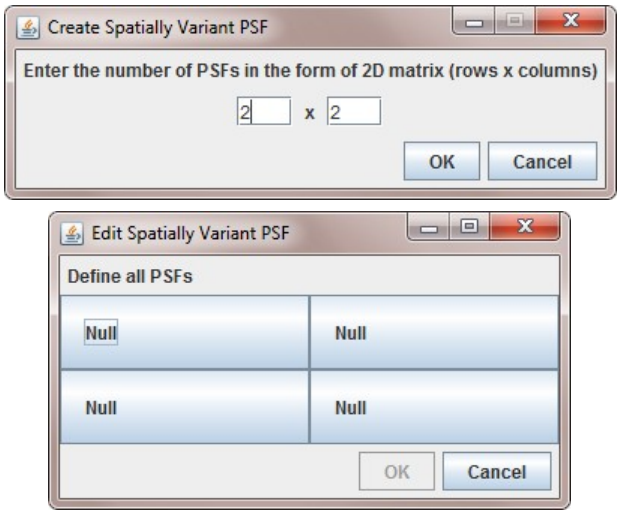

Figure 4.9: Spatially Variant PSF panels.

A 2D image reconstructed with Parallel Iterative Deconvolution is shown in Figure 4.10. It is a picture of a star cluster, courtesy of the Space Telescope Science Institute [4]. The blurred image was generated using a spatially variant blur that simulates images from the original Hubble Telescope. Moreover, some amount of read-out and Poisson noise has been added (consult the README file [104] for details). 25 point spread functions from different regions (see middle of Figure 4.10) of the field of view are provided to test image deblurring algorithms. The reconstructed image was obtained after running 39 iterations of MRNSD with an FFT-preconditioner and reflexive boundary conditions.

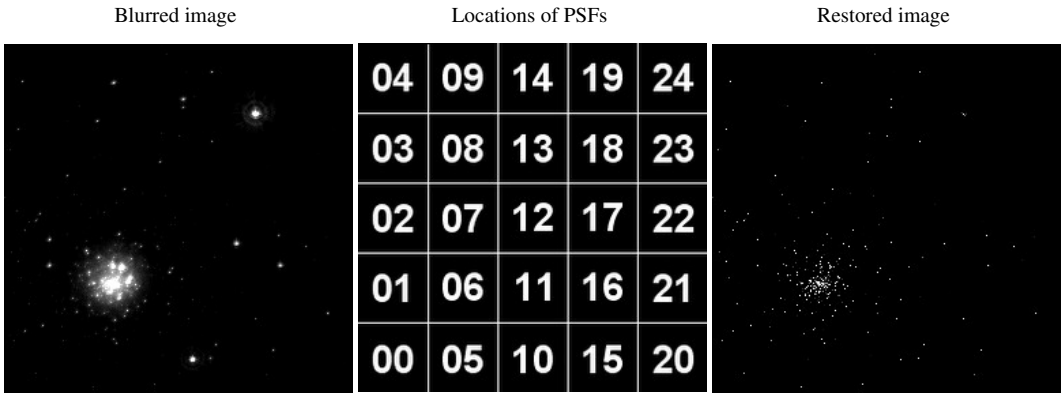

Figure 4.10: Start cluster image: blurred and restored images.

4.2.2 Benchmark

To benchmark Parallel Iterative Deconvolution and Iterative Deconvolve 3D, we used the same test images as for the spectral deconvolution case. Tables 4.3 and 4.4 present average execution times (among ten calls) required to perform five iterations of the preconditioned algorithms in single precision. In the case of Parallel Iterative Deconvolution, one should observe a significant speedup, especially from 1 to 2 threads. In addition, our implementation of the Landweber algorithm outperforms Iterative Deconvolve 3D (denoted by ID3D in the tables) by over 52 times for 2D problems and for almost 44 times for 3D problems (8 threads).

Method	1 thread	2 threads	4 threads	8 threads
CGLS	169.6	110.0	67.2	48.9
MRNSD	179.0	116.8	72.2	52.3
HyBR	183.5	125.6	80.7	58.9
WPL	38.7	21.4	12.3	8.8
ID3D	460.2	-	-	-

Table 4.3: Average execution times (in seconds) for 2D iterative deblurring.

Method	1 thread	2 threads	4 threads	8 threads
CGLS	85.6	54.3	31.2	21.7
MRNSD	91.5	58.7	33.5	23.1
HyBR	95.5	61.9	36.5	26.1
WPL	26.4	13.9	8.12	5.7
ID3D	250.1	-	-	-

Table 4.4: Average execution times (in seconds) for 3D iterative deblurring.

4.3 Parallel HRRT Deconvolution

Parallel HRRT Deconvolution [121] is an ImageJ plugin we developed for motion correction of PET brain images. The goal of Parallel HRRT Deconvolution is to provide an efficient software that is easy to use by PET technicians. Figure 4.11 shows a GUI for the plugin.

4.3.1 Description and Usage

To our knowledge, no other Java software for motion correction of PET brain images currently exists. There is a software package written by Raghunath et al [97], however it requires a deep knowledge of the IDL (ITT Visual Information Solutions, Boulder, CO) programming language and its performance is too low for clinical usage. That package implements a modified version of the ordered subsets expectation maximization (OSEM) algorithm [65] (the subsets are defined in image space rather than in projection space as is normally done). The time required to deconvolve a typical image with Raghunath's software ranges from an average of 6-15 min for 5 and 20 movements, respectively, and uses only two subsets. The time taken to perform the deconvolution increases with the number of subsets used. Moreover, even with the ordered subsets technique the storage requirements are still too high (at least 8GB of RAM memory is needed).

Parallel HRRT Deconvolution implements four iterative solvers (MRNSD, CGLS, HyBR and OSEM) and has superior performance compared to previously used IDL code. A typical usage scenario of the plugin involves the following steps:

1. Start ImageJ.

2. Open a blurred image either by using *File>Open or File>Import* (ImageJ's

menu).

3. Run the plugin by going to *Plugins>Parallel HRRT Deconvolution* (ImageJ's menu).

4. Select a file with a calibration matrix (plugin's GUI).

5. Select a file with motion information (plugin's GUI).

6. Enter the values for sampling rate, scan duration and time offset.

7. Click *Deconvolve* button (plugin's GUI).

8. Adjust the segmentation (if necessary) (visual segmentation editor).

9. Click *Continue* button (visual segmentation editor).

If the reconstruction is not satisfactory, the user can change the default settings in the *Options panel*. The current version of the plugin supports two types of interpolation schemes (*Nearest neighbor* and *Trilinear*) to construct the displacement matrices, four different types of the output image (*Same as source*, *Byte (8-bit)*, *Short (16-bit)* and *Float(32-bit)*) and two precision choices (*Double* and *Single*) used by the algorithms. Moreover, the maximal number of iterations, the maximal number of threads and show iterations options can be adjusted in the plugin's GUI. The *Solve* button allows to use a different solver and/or the maximal number

of iterations for the current data stored in memory. This option saves a lot of computational time, since the preprocessing work needed to prepare input data for a solver is already generated, and does not need to be recomputed.

The algorithm described in Section 2.2.3 does not always generate the optimal segmentation, therefore we developed an easy way to manually adjust the segments. The integral part of the plugin constitutes the visual segmentation editor illustrated in Figure 4.12. This tool allows for editing the automatically generated segmentation in a very intuitive way. The segmentation graph displays normalized motion information (which are called *Data series*) against the scan time (in seconds). The vertical lines plotted on top of the segmentation graph are the segmentation markers. The numerical values (horizontal axis) corresponding to each marker line are displayed in the segmentation table dialog. To remove an unwanted marker line, the user needs to right-click on the appropriate value in the segmentation table and choose *Delete* from the pop-up menu. A unique color is assigned to each marker line and the corresponding row in the table simplifies finding the appropriate value. Adding a new marker line is even simpler; the user just left-clicks on the segmentation graph. Finally, the segmentation can be saved in a file (using the *Save* button in the segmentation table dialog) and used later (when *Auto segmentation* in the plugin's GUI is not selected).

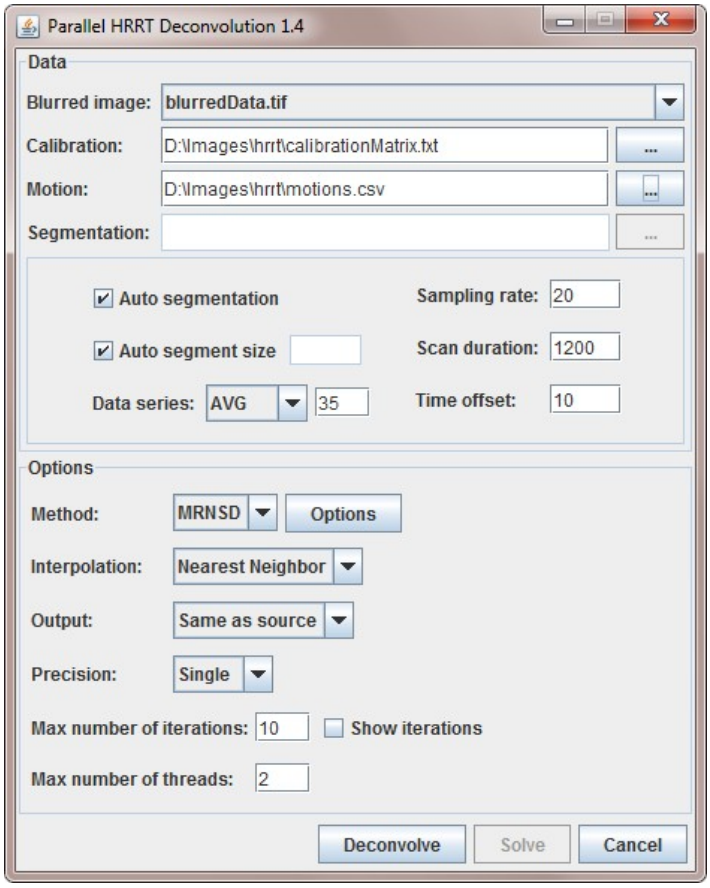

Figure 4.11: Parallel HRRT Deconvolution GUI

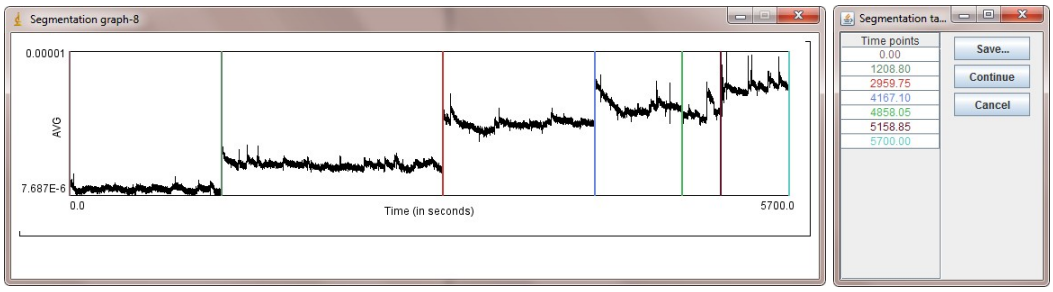

Figure 4.12: Visual segmentation editor.

4.3.2 Benchmark

To show how the plugin works, we simulated motion blur of a software-generated Hoffman phantom image with $256 \times 256 \times 95$ voxels. The motion blurred image was generated using random motion information, trilinear interpolation and 10% of normally distributed noise. The reconstruction algorithms were run by using two different segmentations of the motion data (see Figure 4.13 and 4.14). In the first case, the motion information was divided into seven segments and in the second case into 14 segments. Tables 4.6 and 4.5 report the results (number of iterations, relative error and execution time (in seconds)) for nearest neighbor and trilinear interpolation. OSEM was run with two subsets. For segmentation I we report the number of iterations required to obtain the best solution (smallest relative error), the value of the smallest relative error and the time required to compute the reconstruction (in seconds). For segmentation II, we report relative errors and timings both for the number of iterations required to obtain best solution and for the number of iterations from segmentation I. The numbers in brackets indicate results when HyBR automatically stopped the reconstruction.

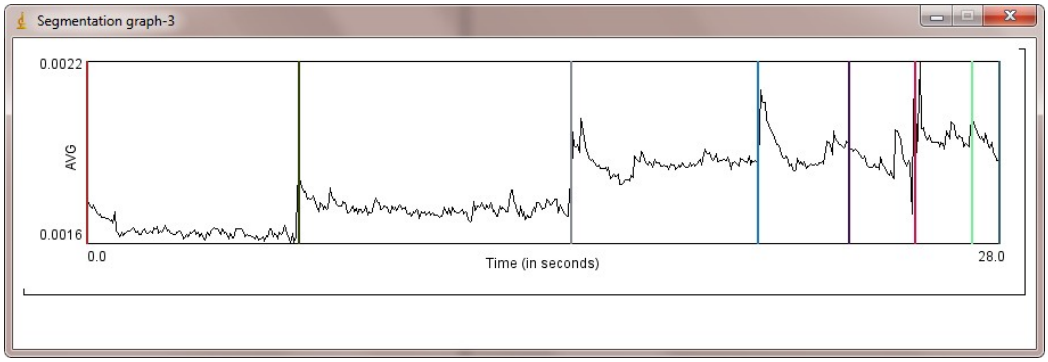

Figure 4.13: Segmentation I of the motion data used with Hoffman phantom data.

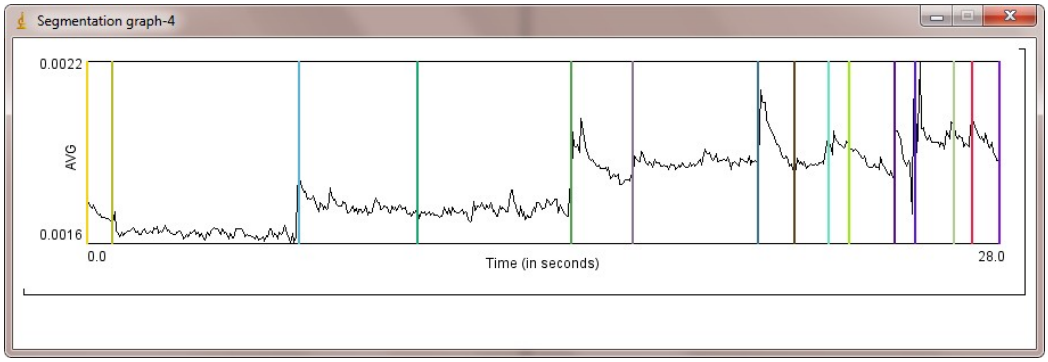

Figure 4.14: Segmentation II of the motion data used with Hoffman phantom data.

The following conclusions can be derived from these results. First of all, trilinear interpolation is not only computationally much more expensive than nearest neighbor interpolation, but also requires up to eight times more memory. More-

over, trilinear interpolation does not improve the reconstruction significantly (see Figure 4.16), thus, we recommend using nearest neighbor interpolation for most problems. From the three methods that we have tested here (see Table 4.6), HyBR was the fastest and OSEM was the slowest. However, in terms of the quality of reconstruction (i.e. relative error and visual inspection), MRNSD outperformed the other two methods (see Figure 4.15). Figure 4.17 shows that the solution obtained from MRNSD, when using segmentation II, is visually better (but not significantly) than the reconstruction generated from segmentation I. Finally, Figure 4.18 illustrates that double precision does not improve the reconstruction. Overall, MRNSD with segmentation II, nearest neighbor interpolation and single precision was the best choice for the software generated Hoffman phantom data.

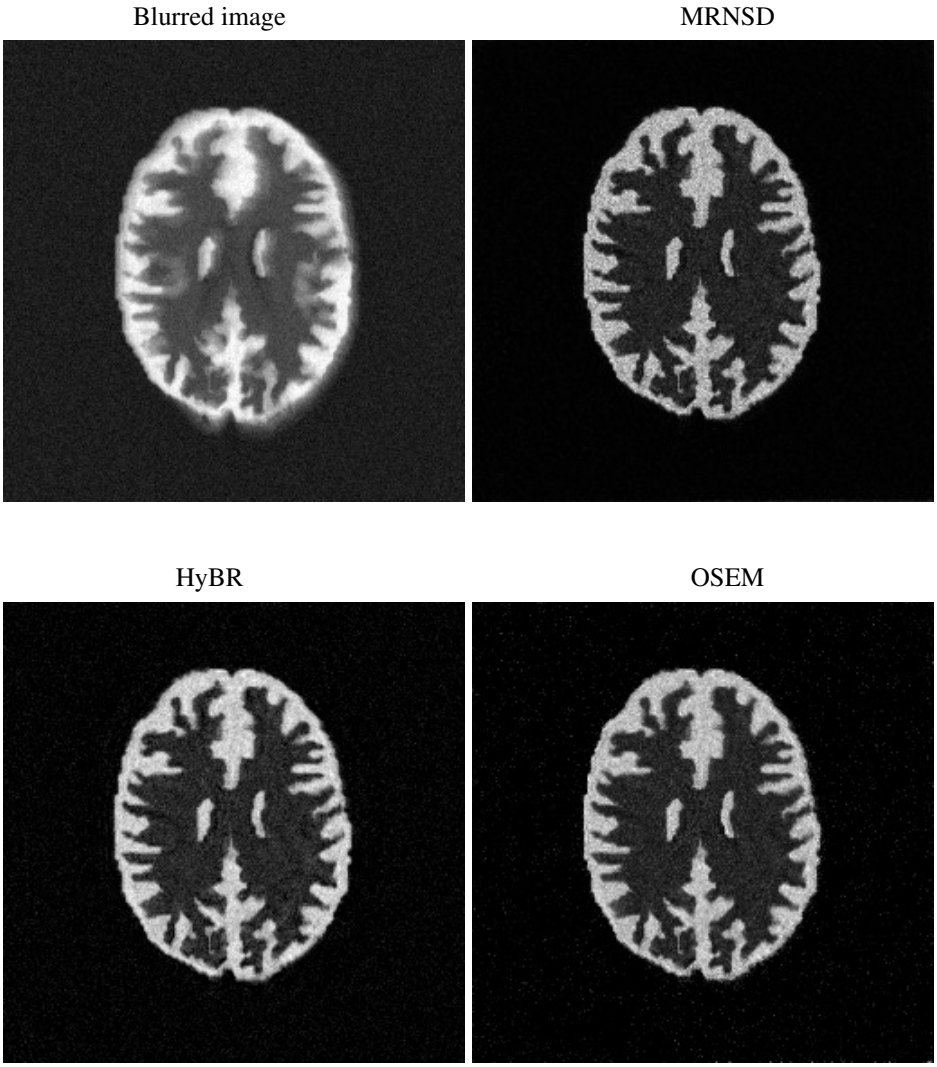

Figure 4.15: Comparison of MRNSD, HyBR and OSEM for Hoffman phantom data (segmentation II, nearest neighbor interpolation, single precision).

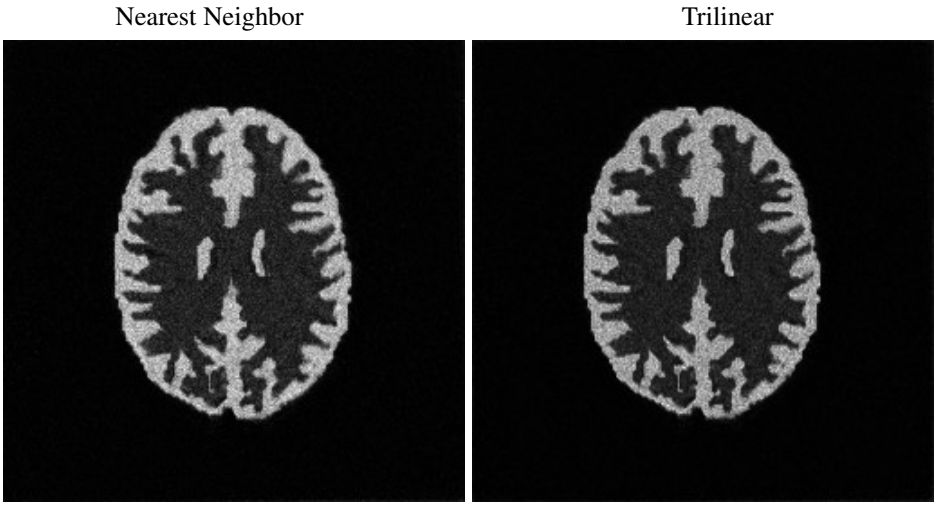

Figure 4.16: Comparison of trilinear and nearest neighbor interpolation for Hoffman phantom data (MRNSD, segmentation II, single precision).

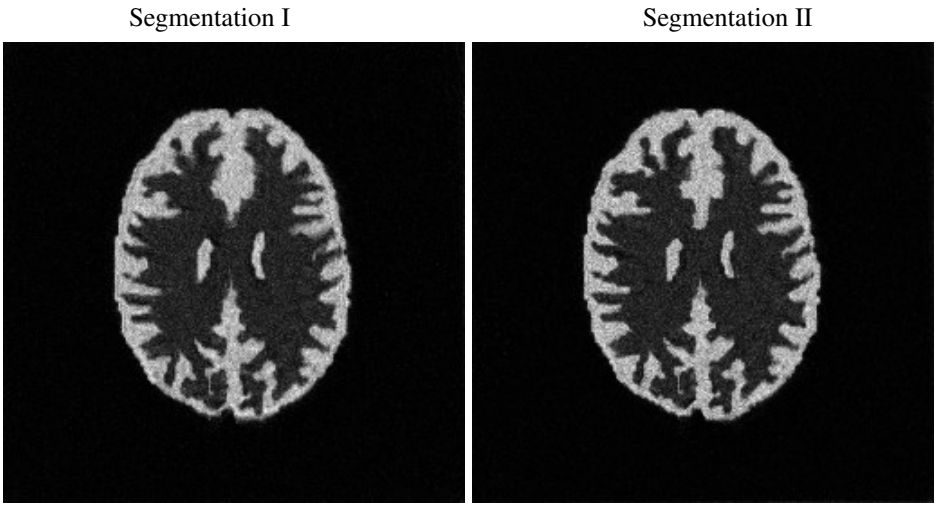

Figure 4.17: Comparison of segmentation I and segmentation II for Hoffman phantom data (MRNSD, nearest neighbor interpolation, single precision).

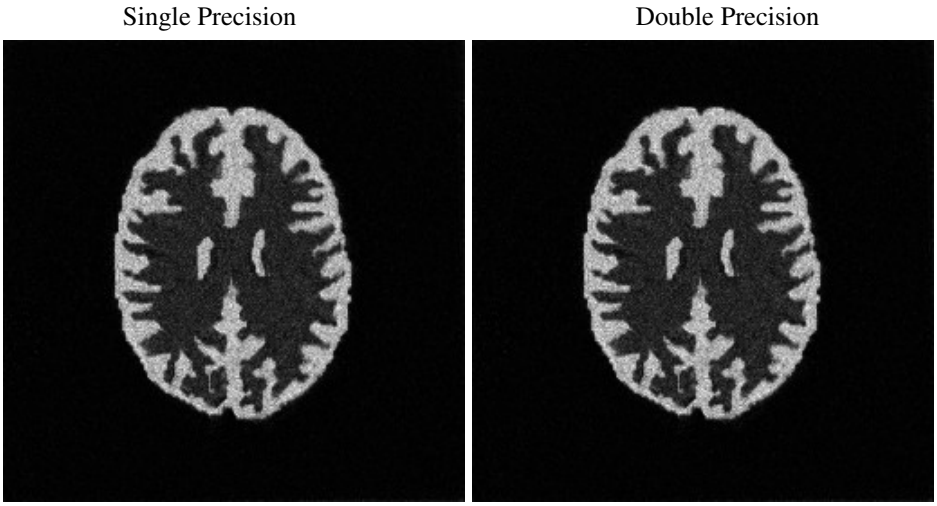

Figure 4.18: Comparison of single and double precision for Hoffman phantom data (MRNSD, segmentation II, nearest neighbor interpolation).

Method	**Segm.**	**Iterations**	**Relative error**	**Time**
MRNSD	I	15	0.3060	96.8
MRNSD	II	15	0.2196	234.2
OSEM	I	7	0.3041	106.3
OSEM	II	7	0.2460	241.3
OSEM	II	14	0.2379	292.1
HyBR	I	2 (10)	0.3182 (0.4086)	77.3 (85.9)
HyBR	II	2	0.2544	196.1
HyBR	II	3 (10)	0.2434 (0.2910)	199.8 (221.3)

Table 4.5: Comparison of timings (in seconds), iterations, and relative errors for Hoffman phantom data (trilinear interpolation, single precision).

Method	Segm.	Iterations	Relative error	Time
MRNSD	I	13	0.3098	15.5
MRNSD	II	13	0.2344	31.5
MRNSD	II	15	0.2339	32.7
OSEM	I	7	0.3094	21.4
OSEM	II	7	0.2522	53.8
OSEM	II	11	0.2490	63.2
HyBR	I	2 (9)	0.3277 (0.3636)	9.4 (14.1)
HyBR	II	2	0.2678	22.15
HyBR	II	3 (9)	0.2676 (0.3166)	22.9 (28.5)

Table 4.6: Comparison of timings (in seconds), iterations, and relative errors for Hoffman phantom data (nearest neighbor interpolation, single precision).

4.4 Parallel Super-Resolution

Parallel Super-Resolution [124] is an ImageJ plugin for super-resolution. The code is based on Julianne Chung's MATLAB scripts. Figure 4.19 shows a graphical user interface for the plugin.

4.4.1 Description and Usage

To our knowledge, no other Java implementation of the super-resolution problem currently exist. The plugin implements the solution scheme based on the Reduced

Gauss-Newton method described in Section 2.3 where HyBR is used to solve a regularized least squares problem in each iteration. Analogous to other plugins described in this chapter, Parallel Super-Resolution can handle arbitrary-sized 2- and 3-dimensional images, but its usage is limited to grayscale images. In addition, the geometrical warping of low-resolution images is limited to the linear affine transformations.

There are six drop-down lists available in the plugin's GUI. From the *Images* list, the user can choose a set of low-resolution images. For 2D images this must be a 3D grayscale stack and for 3D images, the input must be in the form of a 4D grayscale hyperstack. The next list, called *Reference*, allows to choose a reference image. It contains the indices of all 2D input images in a 3D stack (or the indices of 3D input stacks in a 4D hyperstack). On the right of the *Reference* list, there are two text fields that are used to specify scaling factors (the same factors are used for x and y dimension and possibly a different factor for the z dimension). The next two combo-boxes (*Solver* and *Method*) allow to choose a non-linear and a linear solver. However, in the current version of the plugin, both of these lists contain only a single item (*Gauss-Newton* as a non-linear solver and *HyBR* as a linear solver). The *Output* and *Precision* lists provide the same functionalities as for previously described ImageJ plugins.

The Number of iterations text field is used to specify how many iterations of Gauss-Newton should be performed (2 iterations by default). To specify the maximal number of iterations for HyBR (10 iterations by default) or change any other HyBR parameters, the user has to click on the *Options* button. The HyBR options dialog is illustrated in Figure 4.20. The *Show iterations* check-box, if selected, allows to display the reconstructed image after each Gauss-Newton iteration. Finally, in the *Max number of threads* text field the user can specify how many computational threads will be used.

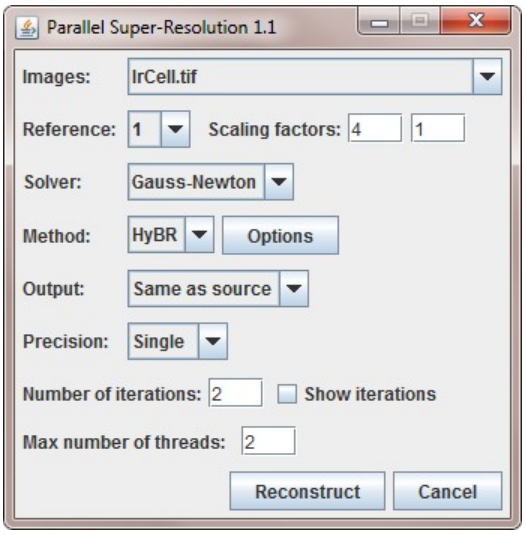

Figure 4.19: Parallel Super-Resolution GUI

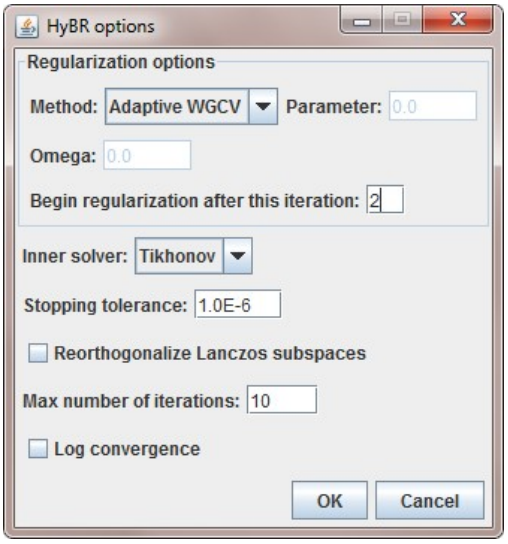

Figure 4.20: HyBR options panel.

Figures 4.21 and 4.22 show reconstructions of 2D and 3D data together with the low-resolution reference and interpolated (using quintic B-spline) images. The low-resolution images were created by rotations and translations of the reference images. Moreover, 1% of normally distributed noise has been added to this data. The image in Figure 4.21 is a cancer cell from a rat's prostate, courtesy of Alan W. Partin, M.D., Ph.D., Johns Hopkins University School of Medicine. The input data comprised of 33 2D images of size 32×32 and the reconstructed image was of the size 256×256. For 3D test data (see Figure 4.22) we used an MRI image from MATLAB's Image Processing Toolbox. In that case, there are 23 3D low-resolution images of size $32 \times 32 \times 7$ and the high-resolution image was of the

size $128 \times 128 \times 7$. As can be seen from these figures, the high-resolution images obtained from the super-resolution process contain significantly more details than the corresponding interpolated images.

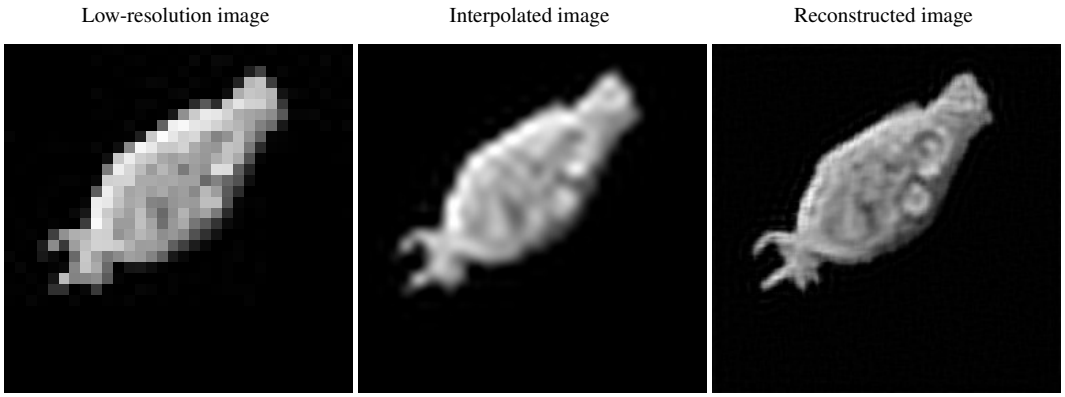

Figure 4.21: Cancer cell from a rat's prostate: low-resolution, interpolated and high-resolution images.

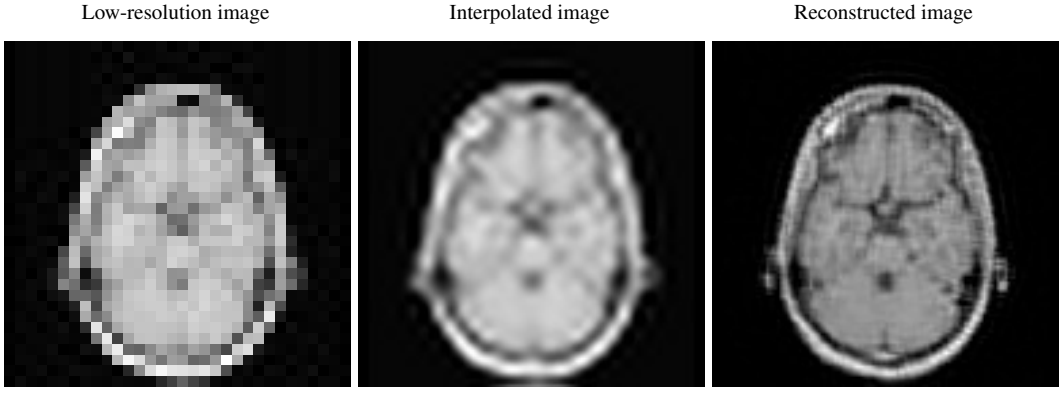

Figure 4.22: MRI: low-resolution, interpolated and high-resolution images (only a single slice is shown).

4.4.2 Benchmark

To benchmark the Parallel Super-Resolution plugin we used the data shown in Figures 4.21 and 4.22. Table 4.7 contains the results of this benchmark as average execution times (in seconds) for different numbers of threads. One can easily notice that the scalability of the plugin is very poor. The main reason of this behavior lies in the data structures and algorithms used in the implementation. The most computationally intensive operations involve sparse matrix-matrix products and sparse matrix-vector products. Only the second operation is currently multithreaded, but its performance depends strongly on the structure of the sparse matrix. Nonetheless, the overall performance of the plugin is satisfactory.

Size	**1 thread**	**2 threads**	**4 threads**	**8 threads**
$(33) \times 32 \times 32$	8.46	8.34	8.17	8.45
$(23) \times 32 \times 32 \times 7$	54.37	54.09	52.66	53.58

Table 4.7: Average execution times (in seconds) for Parallel Super-Resolution.

4.5 Parallel FFTJ

Parallel FFTJ [120] is an FFT plugin for ImageJ. The code is derived from FFTJ (version 2.0) [75] written by Nick Linnenbrügger. Figure 4.23 shows the graphical

user interface for the plugin. Although ImageJ has a built-in FFT functionality, these operations require images of power-of-two size; FFTJ plugin does not have this limitation.

4.5.1 Description and Usage

Parallel FFTJ has the same functionality as FFTJ, but its performance is much better. There is one significant difference between these two plugins: Parallel FFTJ applies scaling when the inverse FFT is chosen, while FFTJ scales only the forward FFT. Contrary to all other software described in this chapter, the code for the GUI of Parallel FFTJ was copied from FFTJ 2.0. The plugin can handle arbitrary-sized 3D volumes as well as single 2D images. Besides defining the input images (both real and imaginary parts), the user can choose the complex number precision (*Single* or *Double*), FFT direction (*Forward* or *Inverse*) and the number of computational threads (must be a power-of-two number). For the output of the transformation (the other dialog window in Figure 4.23), it is possible to choose one of the two Fourier domain origins (*At point (0,0,0)* or *At Volume-Center*). Once this decision is made, the following images can be displayed: real Part, imaginary part, Fourier frequency spectrum, logarithmic Fourier frequency spectrum, Fourier phase spectrum, Fourier power spectrum, and logarithmic Fourier

power spectrum.

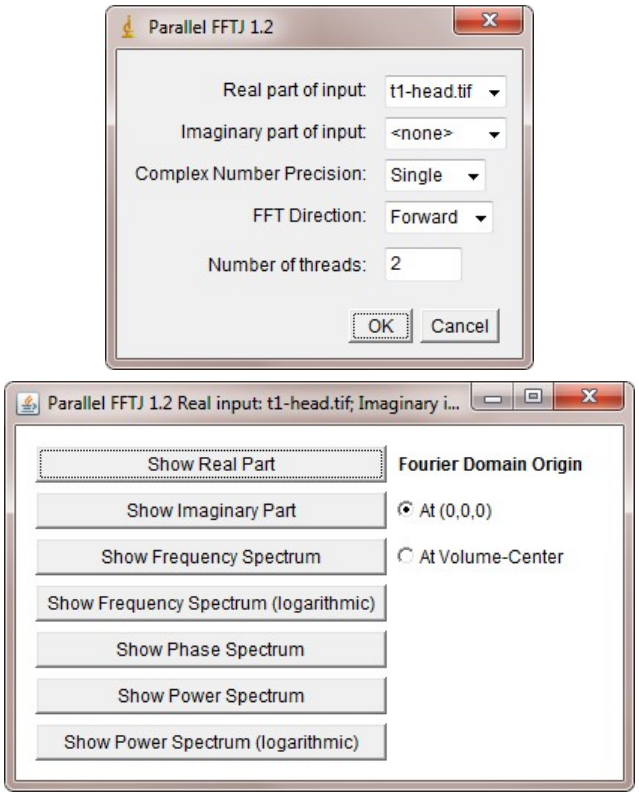

Figure 4.23: Parallel FFTJ GUI.

Figure 4.24 shows the satellite image (real input data) and the corresponding frequency ($\ln(1+|\text{fft2(image)}|)$) and phase ($\arctan(\text{im(fft2(image)), re(fft2(image))})$) spectrum. The images were computed using a froward Fourier transform with the origin at volume center.

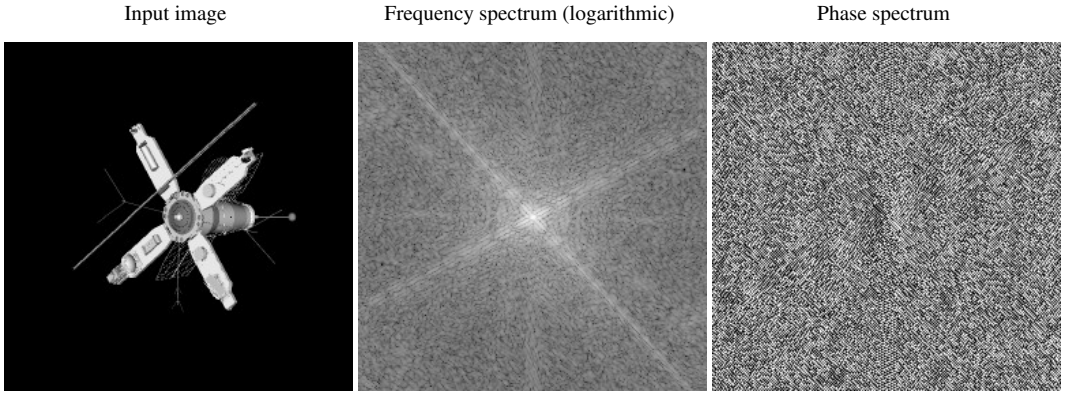

Figure 4.24: Satellite image: input image, frequency spectrum (logarithmic), and phase spectrum.

4.5.2 Benchmark

The performance of Parallel FFTJ was compared to its predecessor - FFTJ 2.0. Table 4.8 contains the timings for computing 2D and 3D single precision, real forward Fourier transforms. For the power-of-two sizes Parallel FFTJ outperforms FFTJ significantly (over 23 times speedup for 4096 x 4096 image (1 thread)). However, the whole power of Parallel FFTJ is revealed for data sizes which are not power-of-two (over 14832 times speedup for 4000 x 4000 image (1 thread)). These results can be explained by the fact that FFTJ 2.0 uses a very naïve implementation of the discrete Fourier transform (DFT) algorithm when the size of the data is not power-of-two number (see Algorithm 4).

Size	FFTJ	Parallel FFTJ (1 thread)	Parallel FFTJ (8 threads)
4000 x 4000	24770.01	1.67	0.89
4096 x 4096	28.62	1.23	0.62
100 x 200 x 200	373.25	0.43	0.17
128 x 256 x 256	5.38	0.70	0.38

Table 4.8: Average execution times (in seconds) for 2D and 3D single precision, real forward Fourier transforms.

Algorithm 4 FFTJ 2.0 implementation of 1D DFT algorithm.

```
1.  void DFT1D(Direction dir, ComplexNum[] src, int len, ComplexNum[] buf) {
2.    for (int i = 0; i < length; i++) {
3.        buf[i].setValue(0, 0);
4.        double arg = -2d * Math.PI * i / len;
5.        if (direction == INVERSE)
6.            arg = -arg;
7.        for (int k = 0; k < len; k++) {
8.            double cosarg = Math.cos(k * arg);
9.            double sinarg = Math.sin(k * arg);
10.           buf[i].addValue(src[k].getRealValue() * cosarg -
11.                           src[k].getImagValue() * sinarg,
12.                           src[k].getRealValue() * sinarg +
13.                           src[k].getImagValue() * cosarg);
14.       }
15.   }
16.   // scaling for forward transformation
17.   if (dir == FORWARD)
18.        for (int i = 0; i < len; i++)
19.            buf[i].divideByValue(len);
20.   // Copy the data back
21.   for (int i = 0; i < len; i++)
22.        src[i].setValue(buf[i]);
23. }
```

4.6 Lincoln Papers

Lincoln Papers [118] in an ImageJ plugin for automatic classification and cropping of scanned paper documents. The graphical user interface for the plugin is illustrated in Figure 4.27.

4.6.1 Description and Usage

The main goal of the *Papers of Abraham Lincoln* project [66] at the University of Illinois' National Center for Supercomputing Applications (NCSA) is to promote learning about Abraham Lincoln via a web-based interface. The architecture diagram of the project is shown in Figure 4.25. The website (client-side) consists of an HTML file with Google Map loaded, a search form, predefined data sets, and a JavaScript script. The server-side consists of a PHP file and MySQL database. The server returns the result as an XML response to the AJAX engine. The user functionalities include searching the database, viewing the results, viewing individual documents, editing or transcribing them, deleting existing postings or adding new ones.

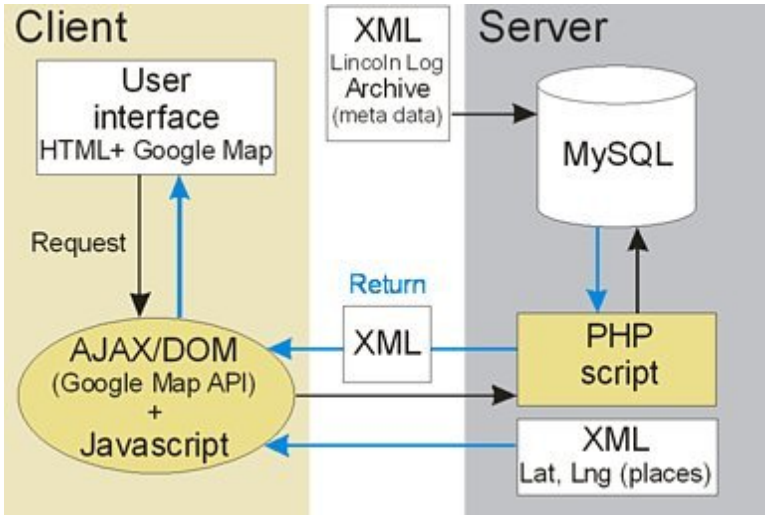

Figure 4.25: Architecture diagram of the Papers of Abraham Lincoln project [66].

From the computational point of view, the project requires automatic classification and cropping of very large collections of scanned paper documents. These documents represent the incoming and outgoing correspondence of Abraham Lincoln. Some documents are scanned together with a color scale bar in order to preserve the color scale. The average image size is about 150 MB. Currently, there are about 23,000 scanned document images (3.45 TB), but the expected amount is 200,000 or 300,000 images (45 TB). The collection (illustrated in Figure 4.26) is characterized by a large variability of paper and ink colors in image scans and the density of writing (text to background ratio). Finally, position of the color scale bar differs among subsets of the documents. Manual cropping would require

about 5 years for a single full-time person with 2 minutes allocated for opening, cropping and saving an image.

To process this large amount of data, the algorithms for automatic classification of images containing the documents with or without additional background patterns have been designed by the Image Spatial Data Analysis Group at NCSA [66]. Then, the images with background are automatically analyzed to identify the crop region containing only the text portion of the documents. Finally, the algorithms are applied to a large volume of images that consists of 1,000 to 100,000 pages. The whole workflow is divided into the following stages:

1. convert the image from RGB to HSB color space
2. create a grid of tiles from each image
3. select the training tile (auto or semi-auto methods)
4. compute the histogram of hue for the training tile
5. select the similarity threshold for comparing histograms (auto or semi-auto methods)
6. compute the histogram of every tile

7. compare the histogram of every tile with the histogram of a training tile and assign labels "with or without color bar"

8. extract the crop area based on tile labels

9. crop the image.

The sequential algorithms for all the stages have been implemented by Melvin Casares and Peter Bajcsy. In order to run efficiently the whole process on multiple multi-core architectures, it was necessary to parallelize the most computationally intensive stages that include: image color space transformation, image tiling, image histogram, and image classification. The contribution of this work is an ImageJ plugin that provides a robust and scalable Java implementation of the whole process.

Figure 4.26: Collection of Lincoln papers [66].

Two combo-boxes available in the plugin's GUI (*Input images extension* and *Cropped images extension*) allow to choose a format for input and output images. Currently six types of images are supported: TIF, JPG, PNG, PGM, BMP, ZIP. The *Number of threads* text field allows the user to define a maximal number of computational threads. The remaining thirteen text fields are used to define the paths to the input and output files and directories.

Since the process requires a lot of time-consuming input/output (I/O) operations, a parallel I/O library has been developed. The code for this library is derived from ImageJ and allows to speed up reading and writing images when a parallel file system is available. The multithreaded implementation is based on the approach discussed in [20]. In particular, we divide the indices of the pixel array among multiple threads and each thread uses its own RandomAccessFile (from `java.io` package) object to set the file-pointer offset and then read / write the corresponding part of the image. Currently the parallel I/O library supports reading and writing of uncompressed TIF files and reading of LZW-compressed TIF files. Sequential algorithms are used for the other file formats, however the parallelization approach is general enough to be applied to any image coding format.

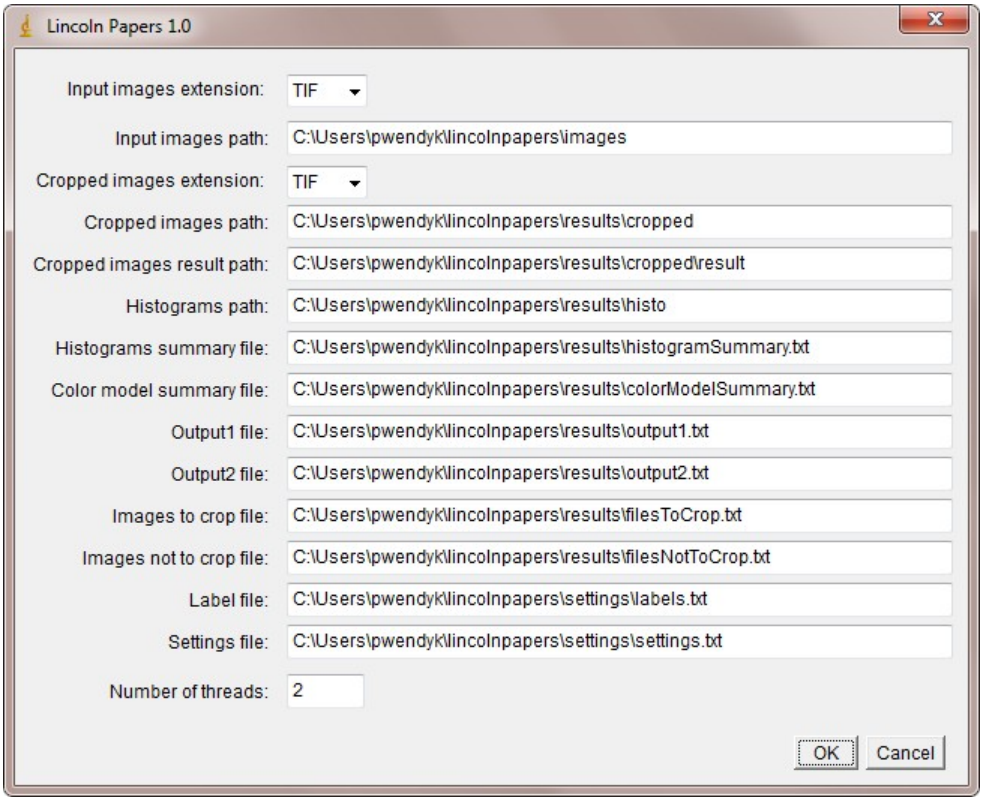

Figure 4.27: Lincoln Papers GUI.

4.6.2 Benchmark

The Lincoln Papers plugin has been benchmarked on SGI Altix 3700 Bx2 supercomputer (Cobalt) [86] equipped with 512 Intel Itanium 2 Processors (1.60 GHz, 9MB L3 Cache), 3TB of RAM memory, Linux kernel: 2.6.16.54-0.2.5 and BEA JRockit R27.6 Java distribution. The test data consisted of 1133 image scans with average image size equal 93.8 MB. For the parallel file installed on Cobalt, we

were able to achieve best performance for four I/O threads. Table 4.9 shows a total execution time of the workflow for the sequence of 1133 image scans as well as for a single image. The number of I/O threads varied according to the formula: number of I/O threads = min(4, number of computational threads).

It can be concluded from these data that the code is scalable up to 16 threads, which means that the whole collection (300,000 images) can be cropped within 217 hours using only 16 CPUs and 4GB RAM. In addition, if one can run 31 separate Java processes on Cobalt, where each process uses 16 threads, then the whole collection can be cropped within 7 hours (vs. 5 years of manual cropping).

Scans	**1 thread**	**2 threads**	**4 threads**	**8 threads**	**16 threads**	**32 threads**
1133	21306.1	12470.2	8420.3	5143.8	2947.7	3467.3
1	18.8	11.0	7.4	4.5	2.6	3.1

Table 4.9: Total execution time (in seconds) of the workflow for the whole sequence (1133 image scans) and a single image scan.

Chapter 5

Conclusions

In this work we have demonstrated the advantage of exploiting available hardware on modern computer architectures in scientific computing and image processing with multithreaded programming in Java. A significant contribution of our work is an open source, multithreaded Java library for high performance scientific computing, Parallel Colt. The current features of Parallel Colt, combined with the front-end functionality of ImageJ, constitute a powerful, portable and user-friendly solution for large-scale image processing.

We have also implemented six ImageJ plugins that utilize many functionalities available in Parallel Colt. Parallel Spectral Deconvolution and Parallel Iterative Deconvolution are the first open source software packages that provide a complete solution to PSF-based image deconvolution. Parallel HRRT Deconvolution, currently used in the Department of Radiology, Emory University School

of Medicine, allows to improve the quality of PET brain images. Parallel Super-Resolution is the first ImageJ plugin for reconstructing of a high-resolution image from a set of low-resolution data. Parallel FFTJ is a reimplementation of fast Fourier transform plugin for ImageJ that, contrary to its predecessor, allows for real-time visualization of Fourier space characteristics. Finally, Lincoln Papers provides an efficient implementation for the problem of automatic classification and cropping of scanned paper documents with color scale bars.

Thus, we are able to provide Java software to solve important problems in real image processing applications, and which can effectively make use of multi-core CPUs available on affordable desktop and notebook computers.

Appendix A

6.1 Fast Fourier Transform

A fast Fourier transform (FFT) algorithm is the most efficient method to compute the discrete Fourier transform (DFT), with a complexity of $\Theta(N \log(N))$ to compute a DFT of a d-dimensional array containing N components. An FFT algorithm was first proposed by Gauss in 1805 [59], but it was the 1965 work by Cooley and Tukey [33] that is generally credited for popularizing its use. The most common variant of the algorithm, called radix-2, uses a divide-and-conquer approach to recursively split the DFT of size N into two parts of size N/2. Other splittings can be used as well, including mixed-radix and split-radix algorithms [112].

The split-radix algorithm requires the lowest arithmetic operation count to compute a DFT when N is a power-of-two [68]. The algorithm was first described in 1968 by Yavne [127] and then rediscovered in 1984 by Duhamel and Hollmann [40]. The idea here is to recursively divide a DFT of size N into one DFT of length

N/2 and two DFTs of length N/4. As a result, the split-radix algorithm can only be applied when N is a multiple of 4. Further details about split-radix algorithm can be found in [112]; here we only present the decomposition. The DFT of vector $\mathbf{x}$ of length N is defined as

$$y_k = \sum_{n=0}^{N-1} x_n \omega_N^{nk} \tag{6.1}$$

where $k = 0, \cdots, N-1$ and $\omega_N = \exp(-2\pi i/N)$. We want to express summation (6.1) in the form of three summations of size N/2, N/4 and N/4 respectively. Let n_m denote an index variable such that $n_m = 0, \cdots, N/m - 1$. Then, we can define the elements with even indices by x_{2n_2}. Similarly, the elements with odd indices can be defined by x_{4n_4+1} (when the index $= 1 \mod 4$) and x_{4n_4+3} (when the index $= 3 \mod 4$). This allows us to write equation (6.1) in the following way

$$y_k = \sum_{n_2=0}^{N/2-1} x_{2n_2} \omega_{N/2}^{n_2 k} + \omega_N^k \sum_{n_4=0}^{N/4-1} x_{4n_4+1} \omega_{N/4}^{n_4 k} + \omega_N^{3k} \sum_{n_4=0}^{N/4-1} x_{4n_4+3} \omega_{N/4}^{n_4 k} \tag{6.2}$$

where $\omega_N^{mnk} = \omega_{N/m}^{nk}$. One should notice that the three sums in equation (6.2) correspond to one DFT of size N/2 and two DFTs of size N/4. All these subtransforms can be now computed by using the same splitting recursively. Thus we can write the split-radix decomposition as

$$y_k = u_k + \omega_N^k z_k + \omega_N^{3k} z_k^{'} \tag{6.3}$$

where u_k ($k = 0, \cdots, N/2 - 1$) is the result of the DFT of length N/2, and z_k and

$z_k^{'}$ ($k = 0, \cdots, N/4 - 1$) are results of two DFTs of length N/4. It can be shown that by using scheme (6.3), many unnecessary calculations are performed. This is due to the fact that the so-called *twiddle factors* ω_N^k and ω_N^{3k} are related

$$\begin{aligned} \omega_N^{k+N/4} &= -i\omega_N^k \\ \omega_N^{3(k+N/4)} &= i\omega_N^{3k} \end{aligned} \tag{6.4}$$

After applying this relation to equation (3), we get the final form of the split-radix decomposition

$$\begin{aligned} x_k &= u_k + \left(\omega_N^k z_k + \omega_N^{3k} z_k^{'}\right) \\ x_{k+N/2} &= u_k - \left(\omega_N^k z_k + \omega_N^{3k} z_k^{'}\right) \\ x_{k+N/4} &= u_{k+N/4} - i\left(\omega_N^k z_k - \omega_N^{3k} z_k^{'}\right) \\ x_{k+3N/4} &= u_{k+N/4} + i\left(\omega_N^k z_k - \omega_N^{3k} z_k^{'}\right) \end{aligned} \tag{6.5}$$

for $k = 0, \cdots, N/4 - 1$.

Appendix B

7.1 Popularity of Parallel Colt

Parallel Colt has been used in the following software projects.

- **NPAIRS/PLS-J** [106] is a Java program developed at Rotman Research Institute (Canada), capable of performing both nonparametric, prediction, activation, influence, reproducibility, re-sampling (NPAIRS) [107] and partial least squares (PLS) [76] analysis.

- **TomoJ** [78] is an ImageJ plugin that provides a user friendly interface for alignment, reconstruction, and combination of multiple tomographic volumes and includes the most recent algorithms for volume reconstructions used in three-dimensional electron microscopy (the algebraic reconstruction technique and simultaneous iterative reconstruction technique) as well as the commonly used approach of weighted back-projection.

- **Incanter** [74] is a Clojure-based, statistical computing and graphics environment for the Java Virtual Machine. Clojure was chosen because its seamless integration with Java and ability to use the large number of existing Java libraries for data access, data processing, and presentation.

- **JQuantLib** [48] is a comprehensive framework for quantitative finance, written in Java. It provides a wide range of mathematical and statistical tools for the valuation of shares, options, futures, swaps, and other financial instruments. It also supports tools related to risk and money management.

- **Endrov** [61] is a multi-purpose image analysis program similar to ImageJ, but with some additional functionalities. Current features include 2D and 3D visualization of image data and annotation, visualization of 3D isosurfaces, support for large data sets (50GB+), data compression, infinite number of channels, batch processing, integration with external tools such as MATLAB image filters and analysis tools, non-destructive real-time application of image operations, and 5D regions of interests.

7.2 Popularity of JTransforms

JTransforms has been used in the following software projects.

- **MusicReader** [73] is a commercial software for musicians. It offers solutions for many problems musicians have with traditional sheet music, both individually and in orchestras and ensembles.

- **Spectro-Edit** [44] is a software that reads PCM audio files and shows the audio visually in a time vs. frequency plot. It is possible to *paint out* any part of the visualization and play back the audio subject to the modifications.

- **Yaprnn** (Yet another pattern recognizing neural network) [21] is a project that uses a neural network trained by the backpropagation algorithm to recognize patterns (audio and visual data).

- **Jkis** [2] is a speech enhancement application written in Java.

- **3dsearch** [1] is a 3D shape search engine written in Java. The system uses the spheric harmonics transform (SHT) to form a shape descriptor.

- **ExpertEyes** [23] is an open source eyetracking application built in Java.

- **METgames** [71] is a software for collaborative online activities for acoustics education and psychoacoustic data collection.

7.3 Popularity of ImageJ Plugins

ImageJ plugins developed as a part of this work have been used in the following.

- **Fiji** [10] is an image processing package based on ImageJ. Fiji enhances ImageJ by shipping with a set of plugins in a coherent menu structure, and comprehensive documentation. The aim of the project is to simplify the installation of ImageJ, the usage of ImageJ, the usage of specific, powerful ImageJ plugins and the development of plugins using ImageJ. Currently Parallel Spectral Deconvolution and Parallel Iterative Deconvolution plugins are available in Fiji.

- **MBF ImageJ** [32] is a customized distribution of ImageJ bundled with plugins useful for microscopy. A detailed manual for each plugin is also provided. Parallel Spectral Deconvolution is a part of this package.

- Parallel Spectral Deconvolution has been used in the article by R. Barbier et al [12].

7.4 Lines of Code

The following list shows the total number of lines of code for each project.

- Parallel Colt 0.9.1: 202,595
- JTransforms 2.3: 48,230
- JPlasma 1.0: 5,171
- CSparseJ 1.0: 5,728
- Parallel Spectral Deconvolution 1.11: 10,245
- Parallel Iterative Deconvolution 1.11: 20,143
- Parallel HRRT Deconvolution 1.5: 8,863
- Parallel Super-Resolution 1.2: 5,202
- Parallel FFTJ 1.3: 1,339
- Lincoln Papers 1.0: 7,220
- TOTAL: 314,736

Bibliography

[1] 3dsearch Project, 2009. http://code.google.com/p/3dsearch/.

[2] Jkis Project, 2009. http://code.google.com/p/jkis/.

[3] JUnit Project, 2009. http://www.junit.org/.

[4] Space Telescope Science Institute, 2009. http://www.stsci.edu/resources/.

[5] AccelerEyes. Jacket, 2009. http://accelereyes.com/.

[6] N. Ahmed, T. Natarajan, and K. R. Rao. Discrete cosine transform. *Transactions on Computers*, C-23(1):90–93, 1974.

[7] B. Amerdo, V. Bodnartchouk, D. Caromel, C. Delbé, F. Huet, and G. L. Taboada. Current state of Java for HPC. Technical Report inria-00312039, INRIA, 2008.

[8] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (third ed.)*. SIAM, Philadelphia, PA, 1999.

[9] Apache Software Foundation. Commons-Math Project, 2009. http://commons.apache.org/math/.

[10] I. Arganda-Carreras, A. Cardona, E. Frise, G. Jefferis, V. Kaynig, G. Landini, M. Longair, S. Preibisch, S. Saalfeld, J. Schindelin, B. Schmid, C. Sicker, J. Tinevez, D. White, and P. Tomancak. Fiji Project, 2009. http://pacific.mpi-cbg.de/wiki/index.php/Main_Page.

[11] H. Arndt, M. Bundschus, and A. Nägele. Towards a next-generation matrix library for Java. In *33rd Annual IEEE International Computer Software and Applications Conference*, 2009.

[12] R. Barbier, J. Baudot, E. Chabanat, P. Depasse, W. Dulinski, N. Estre, C. T. Kaiser, N. Laurent, and M. Winter. Performance study of a megapixel single photon position sensitive photodetector EBCMOS. In *5th International Conference on New Developments In Photodetection*, 2008.

[13] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[14] M. Bertero and P. Boccacci. *Introduction to Inverse Problems in Imaging*. IOP Publishing Ltd., London, 1998.

[15] Å. Björck. A bidiagonalization algorithm for solving large and sparse ill-posed systems of linear equations. *BIT*, 28(3):659–670, 1988.

[16] Å. Björck. *Numerical Methods for Least Squares Problems*. SIAM, Philadelphia, PA, 1996.

[17] L. S. Blackford, J. Demmel, J. J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, June 2002.

[18] P. M. Bloomfield, J. J. Spinks, J. Reed, L. Schnorr, A. M. Westrip, L. Livieratos, R. Fulton, and T. Jones. The design and implementation of a motion

correction scheme for neurological PET. *Phys Med Biol*, 48:959–978, Apr 2003.

[19] L. I. Bluestein. A linear filtering approach to the computation of the discrete Fourier transform. *Northeast Electronics Research and Engineering Meeting Record*, (10):218–219, 1968.

[20] D. Bonachea, P. M. Dickens, and R. Thakur. High-performance file I/O in Java: Existing approaches and bulk I/O extensions. *Concurrency and Computation: Practice and Experience*, 13(8-9):713–736, 2001.

[21] L. Bronstein, S. Sauerstein, F. Rodemund, O. Boroda, F. Schwabe, T. Kwanka, and A. Gruner. Yaprnn Project, 2009. http://code.google.com/p/yaprnn/.

[22] P. Bühler, U. Just, E. Will, J. Kotzerke, and J. van den Hoff. An accurate method for correction of head movement in PET. *IEEE Trans Med Imaging*, 23:1176–1185, Sep 2004.

[23] T. Busey and R. Akavipat. ExpertEyes Project, 2009. http://code.google.com/p/experteyes/.

[24] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. Technical report, Innovative Computing Laboratory, 2007.

[25] J. Byous. Java technology: the early years, 2003. http://java.sun.com/features/1998/05/birthday.html.

[26] R. H. Chan and M. K. Ng. Conjugate gradient methods for Toeplitz systems. *SIAM Review*, 38:427–482, 1996.

[27] T. F. Chan. An optimal circulant preconditioner for Toeplitz systems. *SIAM J. Sci. Stat. Comp.*, 9:766–771, 1988.

[28] T. F. Chan and J. A. Olkin. Preconditioners for Toeplitz-block matrices. *Numer. Algo.*, 6:89–101, 1993.

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

[30] J. Chung and J. G. Nagy. Nonlinear least squares and super resolution. *J. Phys.: Conf. Ser.*, 124(1):P–012019, 2008.

[31] J. Chung, J. G. Nagy, and D. P. O'Leary. A weighted GCV method for Lanczos hybrid regularization. *Elec. Trans. Numer. Anal.*, 28:149–167, 2008.

[32] T. Collins. MBF ImageJ Project, 2009. http://www.macbiophotonics.ca/imagej/.

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

[34] J. Dautelle. JScience Project, 2007. http://jscience.org/.

[35] T. A. Davis. *Direct Methods for Sparse Linear Systems*. SIAM, Philadelphia, PA, 2006.

[36] T. A. Davis. The University of Florida Sparse Matrix Collection, 2009. http://www.cise.ufl.edu/research/sparse/matrices/.

[37] D. M. Doolin, J. J. Dongarra, and K. Seymour. JLAPACK - compiling LAPACK Fortran to Java. *Sci. Program.*, 7(2):111–138, 1999.

[38] R. Dougherty. Extensions of DAMAS and benefits and limitations of deconvolution in beamforming. In *11th AIAA/CEAS Aeroacoustics Conference*, 2005.

[39] DRA Systems. OR-Objects, 2000. http://opsresearch.com/OR-Objects/.

[40] P. Duhamel and H. Hollmann. Split radix FFT algorithms. *Electronic Letters*, 20:14–16, 1984.

[41] T. L. Faber, N. Raghunath, D. Tudorascu, and J. R. Votaw. Motion correction of PET brain images through deconvolution: I. Theoretical development and analysis in software simulations. *Phys Med Biol*, 54(3):797–811, Feb 2009.

[42] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[43] M. Frigo and S. G. Johnson. FFT benchmark methodology, 2009. http://www.fftw.org/speed/method.html.

[44] J. Fuerth. Spectro-Edit Project, 2009. http://code.google.com/p/spectro-edit/.

[45] R. R. Fulton, S. R. Meikle, S. Eberl, J. Pfeiffer, C. J. Constable, and M. J. Fulham. Correction for head movements in PET using an optical motion-tracking system. *IEEE Trans. Nucl. Sci*, 49:116–123, 2002.

[46] G. H. Golub, M. Heath, and G. Wahba. Generalized Cross-Validation as a method for choosing a good ridge parameter. *Technometrics*, 21(2):215–223, 1979.

[47] G. H. Golub and C. F. Van Loan. *Matrix Computations (Johns Hopkins Studies in Mathematical Sciences)*. The Johns Hopkins University Press, October 1996.

[48] R. Gomez. JQuantLib Project, 2009. http://www.jquantlib.org/index.php/Main_Page.

[49] R. C. Gonzalez and P. Wintz. *Digital Image Processing*, chapter 5. Addison-Wesley Pub. Co, 1977.

[50] M. V. Green, J. Seidel, S. D. Stein, T. E. Tedder, K. M. Kempner, C. Kertzman, and T. A. Zeffiro. Head movement in normal subjects during simulated PET brain imaging with and without head restraint. *J. Nucl. Med.*, 35:1538–1546, 1994.

[51] G.W. Stewart. Jampack project, 2009. ftp://math.nist.gov/pub/Jampack/Jampack/AboutJampack.html.

[52] D. Hale. The Java and C++ platforms for scientific computing. Technical Report CWP-547, Center for Wave Phenomena, Department of Geophysics, Colorado School of Mines, 2006.

[53] D. Hale. Mines Java Toolkit (JTK), 2009. http://inside.mines.edu/%7Edhale/jtk/index.html.

[54] M. Hanke. *Conjugate gradient type methods for ill-posed problems*. Pitman Research Notes in Mathematics, Longman Scientific & Technical, Harlow, Essex, 1995.

[55] P. C. Hansen. Analysis of discrete ill-posed problems by means of the L-curve. *SIAM Rev.*, 34(4):561–580, 1992.

[56] P. C. Hansen. *Rank-Deficient and Discrete Ill-Posed Problems*. SIAM, Philadelphia, PA, 1997.

[57] P. C. Hansen, J. G. Nagy, and D. P. O'Leary. *Deblurring Images Matrices, Spectra and Fitering*. SIAM, Philadelphia, PA, 2006.

[58] R. V. L. Hartley. A more symmetrical Fourier analysis applied to transmission problems. In *Proceedings of IRE*, 1942.

[59] M. Heideman, D. Johnson, and C. Burrus. Gauss and the history of the fast Fourier transform. *ASSP Magazine, IEEE [see also IEEE Signal Processing Magazine]*, 1(4):14–21, 1984.

[60] B. Heimsund. Matrix toolkits for Java, 2007. http://ressim.berlios.de/.

[61] J. Henriksson. Endrov Project, 2009. http://www.endrov.net/index.php/Main_Page.

[62] J. Hicklin, C. Moler, P. Webb, R. F. Boisvert, B. Miller, R. Pozo, and K. Remington. JAMA : A Java Matrix Package, 2005. http://math.nist.gov/javanumerics/jama/.

[63] W. Hoschek. Uniform, versatile and efficient dense and sparse multidimensional arrays, 2000. http://acs.lbl.gov/%7Ehoschek/publications/ACMJava2000.pdf.

[64] W. Hoschek. Colt Project, 2004. http://dsd.lbl.gov/%7Ehoschek/colt/index.html.

[65] M. Hudson and R. Larkin. Accelerated image reconstruction using ordered subsets of projection data. *IEEE Trans. Med. Imag*, 13:601–609, 1994.

[66] Image Spatial Data Analysis Group at NCSA. The Papers of Abraham Lincoln Project, 2009. http://isda.ncsa.uiuc.edu/lpapers/index.html.

[67] JCuda. JCuda Project, 2009. http://www.jcuda.org/jcuda/JCuda.html.

[68] S. G. Johnson and M. Frigo. A modified split-radix FFT with fewer arithmetic operations. *IEEE Trans. Signal Processing*, 55(1):111–119, 2007.

[69] W. Kahan and J. D. Darcy. How Java's floating-point hurts everyone everywhere, 1998. http://www.cs.berkeley.edu/%7Ewkahan/JAVAhurt.pdf.

[70] M. E. Kilmer and D. P. O'Leary. Choosing regularization parameters in iterative methods for ill-posed problems. *SIAM J. Matrix Anal. Appl.*, 22:1204–1221, 2001.

[71] Y. E. Kim, T. M. Doll, and R. Migneco. Collaborative online activities for acoustics education and psychoacoustic data collection. *IEEE Transactions on Learning Technologies*, 99(2), 2009.

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

[73] Leonè MusicReader. MusicReader, 2009. http://www.musicreader.net/.

[74] D. Liebke. Incanter Project, 2009. http://wiki.github.com/liebke/incanter.

[75] N. Linnenbrügger. FFTJ and DeconvolutionJ, 2002. http://rsb.info.nih.gov/ij/plugins/fftj.html.

[76] A. R. Mcintosh, F. L. Bookstein, J. V. Haxby, and C. L. Grady. Spatial pattern analysis of functional brain images using partial least squares. *Neuroimage*, 3:143–157, 1996.

[77] M. Menke, M. Atkins, and K. Buckley. Compensation methods for head motion detected during PET imaging. *IEEE Trans. Nucl. Sci*, 43:310–317, 1996.

[78] C. Messaoudi, T. Boudier, C. Sorzano, and S. Marco. TomoJ: tomography software for three-dimensional reconstruction in transmission electron microscopy. *BMC Bioinformatics*, 8(1):288, 2007.

[79] V. A. Morozov. On the solution of functional equations by the method of regularization. *Soviet Math. Dokl.*, 7:414–417, 1966.

[80] J. G. Nagy and D. P. O'Leary. Preconditioned iterative regularization for ill-posed problems. In *Numerical Linear Algebra and Scientific Computing*, number 3162, pages 141–163, 1993.

[81] J. G. Nagy and D. P. O'Leary. Restoring images degraded by spatially-variant blur. *SIAM J. Sci. Comput*, 19:1063–1082, 1996.

[82] J. G. Nagy and D. P. O'Leary. Fast iterative image restoration with a space-varying PSF. In *Advanced Signal Processing Algorithms, Architectures, and Implementations IV*, number 3162, pages 388–399, 1997.

[83] J. G. Nagy, K. Palmer, and L. Perrone. Iterative methods for image deblurring: A MATLAB object-oriented approach. *Numerical Algorithms*, 36(1):73–93, May 2004.

[84] J. G. Nagy and Z. Strakoš. Enforcing nonnegativity in image reconstruction algorithms. In D.C. Wilson et. al., editor, *Mathematical Modeling, Estimation and Imaging*, volume 4121, pages 182–190, 2000.

[85] NASA. Great images in NASA. Ed White performs first U.S. spacewalk., 1965. http://grin.hq.nasa.gov/ABSTRACTS/GPN-2006-000025.html.

[86] NCSA. SGI Altix 3700 Bx2 (Cobalt), 2009. http://www.ncsa.illinois.edu/UserInfo/Resources/Hardware/SGIAltix/.

[87] NIST. Matrix Market Exchange Formats, 2009. http://math.nist.gov/MatrixMarket/formats.html#MMformat.

[88] J. Nocedal and S. Wright. *Numerical Optimization*. New York: Springer, 1999.

[89] NVIDIA Corporation. CUDA Zone, 2009. http://www.nvidia.com/object/cuda_home.html.

[90] D. P. O'Leary and J. A. Simmons. A bidiagonalization - regularization procedure for large scale discretizations of ill-posed problems. *SIAM J. Sci. Stat. Comp.*, 2:474–489, 1981.

[91] T. Ooura. General Purpose FFT (Fast Fourier/Cosine/Sine Transform) Package, 2006. http://www.kurims.kyoto-u.ac.jp/%7Eooura/fft.html.

[92] Optimatika. ojAlgo Project, 2009. http://ojalgo.org/index.html.

[93] J. Orchard. His brain, 2007. http://www.cs.uwaterloo.ca/%7Ejorchard/mri/.

[94] S. C. Park, M. K. Park, and M. G. Kang. Super-resolution image reconstruction: a technical overview. *Signal Processing Magazine, IEEE*, 20(3):21–36, 2003.

[95] Y. Picard and C. J. Thompson. Motion correction of PET images using multiple acquisition frames. *IEEE Transactions on Medical Imaging*, 16:137–144, 1997.

[96] R. Pozo and R. Boisvert. Java numerics, 2009. http://math.nist.gov/javanumerics/.

[97] N. Raghunath, T. L. Faber, S. Suryanarayanan, and J. Votaw. Motion correction of PET brain images through deconvolution: II. Practical implementation and algorithm optimization. *Phys Med Biol*, 54(3):813–829, Feb 2009.

[98] A. Rahmim, P. Bloomfield, S. Houle, M. Lenox, C. Michel, K. R. Buckley, T. J. Ruth, and V. Sossi. Motion compensation in histogram-mode and list-mode EM reconstructions: beyond the event-driven approach. *IEEE Trans. Nucl. Sci*, 51:2588–2596, 2004.

[99] A. Rahmim, J. C. Cheng, K. Dinelle, M. Shilov, W. P. Segars, O. G. Rousset, B. M. Tsui, D. F. Wong, and V. Sossi. System matrix modelling of externally tracked motion. *Nucl. Med. Commun.*, 29:574–581, 2008.

[100] W. S. Rasband. ImageJ, U. S. National Institutes of Health, Bethesda, Maryland, USA, 2009. http://rsb.info.nih.gov/ij/.

[101] H. W. Richardson. Bayesian-based iterative method of image restoration. *Journal of the Optical Society of America*, 62(1):55–59, January 1972.

[102] Y. Saad. ILUT: a dual threshold incomplete ILU factorization. *Numerical Linear Algebra with Applications*, 1:387–402, 1994.

[103] J. C. Schatzman. Accuracy of the discrete Fourier transform and the fast Fourier transform. *SIAM Journal on Scientific Computing*, 17(5):1150–1166, 1996.

[104] Space Telescope Science Institute. Star cluster readme, 2009. ftp://ftp.stsci.edu/software/stsdas/testdata/restore/sims/star_cluster/README.

[105] G. W. Stewart. *Matrix Algorithms, Volume 1: Basic Decompositions*. SIAM, Philadelphia, PA, 1998.

[106] S. Strother, A. R. McIntosh, I. Somji, A. Oder, N. Spreng, J. Waller, F. Wong, D. Wright, G. Yourganov, and R. Zhao. PLS/NPAIRS-J Project, 2009. http://code.google.com/p/plsnpairs/.

[107] S. C. Strother, J. Anderson, and L. K. Hansen. The quantitative evaluation of functional neuroimaging experiments: The NPAIRS. *NeuroImage*, 15(4):747–771, 2002.

[108] Sun Microsystems. New features and enhancements J2SE 5.0, 2004. http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html.

[109] P. N. Swarztrauber. FFTPACK Project, 2004. http://www.cisl.ucar.edu/css/software/fftpack5/.

[110] C. Temperton. Implementation of a self-sorting in-place prime factor FFT algorithm. *Journal of Computational Physics*, 58:283–299, 1985.

[111] H. A. van der Vorst. *Iterative Krylov Methods for Large Linear Systems*. Cambridge University Press, Cambridge UK, 2003.

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

[113] P. Vanek, J. Mandel, and M. Brezina. Algebraic multigrid based on smoothed aggregation for second and fourth order problems. *Computing*, 56:179–196, 1996.

[114] S. Verrill. Nonlinear Optimization Java Package, 2009. http://www1.fpl.fs.fed.us/optimization.html.

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

[116] P. Wendykier. CSparseJ Project, 2009. http://sites.google.com/site/piotrwendykier/software/csparsej.

[117] P. Wendykier. JPlasma Project, 2009. http://sites.google.com/site/piotrwendykier/software/jplasma.

[118] P. Wendykier. Lincoln Papers ImageJ plugin, 2009. http://code.google.com/p/google-summer-of-code-2008-ncsa/.

[119] P. Wendykier. Parallel Colt Project, 2009. http://sites.google.com/site/piotrwendykier/software/parallelcolt.

[120] P. Wendykier. Parallel FFTJ Project, 2009. http://sites.google.com/site/piotrwendykier/software/parallelfftj.

[121] P. Wendykier. Parallel HRRT Deconvolution Project, 2009. http://sites.google.com/site/piotrwendykier/software/deconvolution/parallelhrrtdeconvolution.

[122] P. Wendykier. Parallel Iterative Deconvolution Project, 2009. http://sites.google.com/site/piotrwendykier/software/deconvolution/paralleliterativedeconvolution.

[123] P. Wendykier. Parallel Spectral Deconvolution Project, 2009. http://sites.google.com/site/piotrwendykier/software/deconvolution/parallelspectraldeconvolution.

[124] P. Wendykier. Parallel Super-Resolution Project, 2009. http://sites.google.com/site/piotrwendykier/software/parallelsuperresolution.

[125] P. Wendykier and J. G. Nagy. Large-scale image deblurring in Java. In *Computational Science - ICCS 2008, 8th International Conference, Kraków, Poland, June 23-25, 2008, Proceedings, Part I*, pages 721–730, 2008.

[126] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software ATLAS. In *Proceedings of Supercomputing 1998*, 1998.

[127] R. Yavne. An economical method for calculating the discrete Fourier transform. In *AFIPS Fall Joint Computer Conference*, pages 115–125, 1968.

[128] P. Yip and K. R. Rao. A fast computational algorithm for the discrete sine transform. *IEEE Trans. Commun.*, 28(2):304 – 307, 1980.

[129] B. Zhang. Java FFTPack Project, 2005. http://jfftpack.sourceforge.net/.