**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:

_____ _____

Sarah M. Knepper Date

Large-Scale Inverse Problems in Imaging: Two Case Studies

By

Sarah M. Knepper
Doctor of Philosophy

Mathematics and Computer Science

---

James G. Nagy, Ph.D.
Advisor

---

Michele Benzi, Ph.D.
Committee Member

---

Vaidy Sunderam, Ph.D.
Committee Member

Accepted:

---

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

---

Date

Large-Scale Inverse Problems in Imaging: Two Case Studies

By

Sarah M. Knepper
B.A., College of Saint Benedict, 2006

Advisor: James G. Nagy, Ph.D.

An abstract of
A dissertation submitted to the Faculty of the
James T. Laney School of Graduate Studies of Emory University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in Mathematics and Computer Science
2011

**Abstract**

Large-Scale Inverse Problems in Imaging: Two Case Studies
By Sarah M. Knepper

Solving inverse problems is an important part of scientific computing. As computers become more powerful, solutions to increasingly larger problems are sought, allowing for more accurate representations of real-world applications. We consider solving large-scale inverse problems, ranging from linear to fully nonlinear. We look at aspects common to inverse problems, such as their ill-posedness, and see how regularization can help produce meaningful results. We discuss a number of different methods for solving while providing regularization. One such technique is to solve using an iterative method but stop the iterations early, before convergence is fully achieved.

Iterative solvers are particularly useful for large-scale inverse problems as computations can be done in parallel. Trilinos is a mathematical software library for solving problems coming from many fields of scientific computing. One particular package, Belos, provides both an abstract framework and concrete implementations of various iterative solvers. We have implemented two additional solvers within the Belos framework, LSQR and MRNSD, which can be used to solve linear inverse problems.

We then consider two different case studies, where we wish to solve a large-scale linear inverse problem. In the first study, we want to remove patient motion blur from positron emission tomography (PET) images when motion information is tracked and recorded during the scan. We describe how this problem can be formulated as a linear equation, then we solve it using the solvers we implemented. We also look at a number of results, seeing how the reconstruction improves as more motion information is included in our model.

The second case study comes from the field of adaptive optics. Here we wish to determine the distortion caused by the atmosphere when imaging using ground-based telescopes. Sensors are able to obtain noisy estimates of the gradients of the distortion, resulting in a Kronecker product-structured linear least squares problem. We describe a solving method that employs Tikhonov-type regularization by exploiting properties of the Kronecker product and utilizing the generalized singular value decomposition (GSVD). Our approach includes constructing a preconditioner off-line and then applying a few iterations of preconditioned LSQR.

# Large-Scale Inverse Problems in Imaging: Two Case Studies

By

Sarah M. Knepper
B.A., College of Saint Benedict, 2006

Advisor: James G. Nagy, Ph.D.

A dissertation submitted to the Faculty of the
James T. Laney School of Graduate Studies of Emory University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in Mathematics and Computer Science
2011

**Acknowledgments**

I am indebted to quite a number of people to reach this point in my scholastic journey, but for reasons of brevity, I shall try to limit myself here. Even though a name may not be specifically mentioned, know that I am grateful for your impact on my life.

First, I would be nowhere if it were not for my teachers, so thank you to all of the professors I have had the fortune of meeting, for passing on your knowledge to me and my fellow students.

Prof. Michael Heroux, my undergrad advisor, thank you for allowing me to do research for (and with) you. I am particularly grateful for my first Sandia internship – that is what put me on the road to graduate school – and your suggestion to consider Emory was a great one.

Profs. Michele Benzi and Vaidy Sunderam: thank you both for further research experience, for serving on my committee, and especially for your helpful comments to improve this manuscript.

Thank you to Prof. James Nagy, my advisor, for passing on your love of inverse problems to me. You have given me so many opportunities – writing a book chapter, traveling to international conferences, co-teaching a Computational Methods in Imaging course. Thank you, also, for supporting me and allowing me to complete my last year long distance. I cannot imagine being where I am without you, nor would I want to.

Thank you, also, to my collaborators on various publications; some of that work has helped make up this manuscript. The staff at Emory have been such a great support to me; thank you for all your hard work. To my fellow students – thanks for a great five years!

To one student in particular, Jake, I am so glad our paths crossed at grad school. You have been such a great support, keeping me grounded after I have had my head in the clouds with research all day.

Finally, I would like to acknowledge my family, without which I would not be here today. Thank you for all your support throughout my entire life, always letting me spread my wings and fly, as I find my place in the world.

*S.D.G.*

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Large-scale inverse problems arise in a variety of significant applications in image processing, and efficient regularization methods are needed to compute meaningful solutions. This chapter surveys three common mathematical models including a linear, a separable nonlinear, and a general nonlinear model. Techniques for regularization and large-scale implementations are considered, with particular focus given to algorithms and computations that can exploit structure in the problem. Much progress has been made in the field of large-scale inverse problems, but many challenges still remain for future research.

Powerful imaging technologies, including very large telescopes, synthetic aperture radar, medical imaging scanners, and modern microscopes, typically combine a device that collects electromagnetic energy (e.g., photons) with a computer that assembles the collected data into images that can be viewed by practitioners, such as scientists and doctors. The "assembling" process typically involves solving an *inverse problem*; that is, the image is reconstructed from indirect measurements of the corresponding object. Many inverse problems are also *ill-posed*, meaning that small changes in the measured data can lead to large changes in the solution, and special tools or techniques are needed to deal with this instability. In fact, because real data will not be exact (it will contain at least some small amount of noise or other errors from the data collection device), it is not possible to find the exact solution. Instead, a physically realistic approximation is sought. This is done by formulating an appropriate *regularized* (i.e., stabilized) problem, from which a good approximate solution can be computed.

Inverse problems are ubiquitous in imaging applications, including deconvolution (or, more generally, deblurring) [1, 67], super-resolution (or image fusion) [28, 33], image registration [92], image reconstruction [96, 97], seismic imaging [38], inverse scattering [25], and radar imaging [27]. These problems are referred to as *large-scale*

because they typically require processing a large amount of data (the number of pixels or voxels in the discretized image) and systems with a large (e.g., $10^9$ for a three-dimensional image reconstruction problem) number of equations. Mathematicians began to rigorously study inverse problems in the 1960s, and this interest has continued to grow over the past few decades due to applications in fields such as biomedical, seismic, and radar imaging; see, for example, [22, 35, 62, 64, 128] and the references therein.

This chapter discusses computational approaches to compute approximate solutions of large-scale inverse problems. Mathematical models and some applications are presented in Section 1.1. Three basic models are considered: a general nonlinear model, a linear model, and a mixed linear/nonlinear model. Several regularization approaches are described in Section 1.2. For an extended example problem of each model type, see [29].

We will then use some of these approaches to solve two different inverse problems. First, however, we will consider the large-scale software library Trilinos [69] and describe two iterative solvers we have implemented within the Trilinos framework in Chapter 2. Next, we will consider a problem in Chapter 3 that deals with removing motion blur from positron emission tomography brain scans. The other inverse problem comes from the field of adaptive optics; here we wish to reconstruct a wavefront given noisy gradient information. More details are in Chapter 4.

## 1.1 Background

A mathematical framework for inverse problems is presented in this chapter. The model problems, which range from linear to nonlinear, are fairly general and can be used to describe many other applications. For more complete treatments of inverse problems and regularization, see [22, 35, 62, 64, 66, 128].

### 1.1.1 Model Problems

An inverse problem involves the estimation of certain quantities using information obtained from indirect measurements. A general mathematical model to describe this process is given by

$$\boldsymbol{b}_{\text{exact}} = F(\boldsymbol{x}_{\text{exact}}) \,, \tag{1.1}$$

where $\boldsymbol{x}_{\text{exact}}$ denotes the exact (or ideal) quantities that need to be estimated, and $\boldsymbol{b}_{\text{exact}}$ is used to represent perfectly measured (error-free) data. The function $F$ is

defined by the data collection process and is assumed known. Typically it is assumed that $F$ is defined on Hilbert spaces and that it is continuous and weakly sequentially closed [36].

Unfortunately, in any real application, it is impossible to collect error-free data, so a more realistic model of the data collection process is given by

$$\boldsymbol{b} = F(\boldsymbol{x}_{\text{exact}}) + \boldsymbol{\eta}\,, \tag{1.2}$$

where $\boldsymbol{\eta}$ represents noise and other errors in the measured data. The precise form of $F$ depends on the application; the following three general situations are considered in this chapter:

- For linear problems $F(\boldsymbol{x}) = A\boldsymbol{x}$, where $A$ is a linear operator. In this case the data collection process is modeled as

  $$\boldsymbol{b} = A\boldsymbol{x}_{\text{exact}} + \boldsymbol{\eta}\,,$$

  and the inverse problem is: given $\boldsymbol{b}$ and $A$, compute an approximation of $\boldsymbol{x}_{\text{exact}}$.

- In some cases, $\boldsymbol{x}$ can be separated into two distinct components, $\boldsymbol{x}^{(\ell)}$ and $\boldsymbol{x}^{(n\ell)}$, with $F(\boldsymbol{x}) = F(\boldsymbol{x}^{(\ell)}, \boldsymbol{x}^{(n\ell)}) = A(\boldsymbol{x}^{(n\ell)})\boldsymbol{x}^{(\ell)}$, where $A$ is a linear operator defined by $\boldsymbol{x}^{(n\ell)}$. That is, the data $\boldsymbol{b}$ depends linearly on $\boldsymbol{x}^{(\ell)}$ and nonlinearly on $\boldsymbol{x}^{(n\ell)}$. In this case the data collection process is modeled as

  $$\boldsymbol{b} = A(\boldsymbol{x}^{(n\ell)}_{\text{exact}})\boldsymbol{x}^{(\ell)}_{\text{exact}} + \boldsymbol{\eta}\,,$$

  and the inverse problem is: given $\boldsymbol{b}$ and the parametric form of $A$, compute approximations of $\boldsymbol{x}^{(n\ell)}_{\text{exact}}$ and $\boldsymbol{x}^{(\ell)}_{\text{exact}}$.

- If the problem is not linear or separable, as described above, then the general nonlinear model,

  $$\boldsymbol{b} = F(\boldsymbol{x}_{\text{exact}}) + \boldsymbol{\eta}\,,$$

  will be considered. In this case the inverse problem is: given $\boldsymbol{b}$ and $F$, compute an approximation of $\boldsymbol{x}_{\text{exact}}$.

In most of what follows, it is assumed that the problem has been discretized, so $\boldsymbol{x}$, $\boldsymbol{b}$ and $\boldsymbol{\eta}$ are vectors, and $A$ is a matrix. Depending on the constraints assumed and the complexity of the model used, problems may range from linear to fully nonlinear.

## 1.2 Mathematical Modelling and Analysis

A significant challenge when attempting to compute approximate solutions of inverse problems is that they are typically ill-posed. To be precise, in 1902 Hadamard defined a well-posed problem as one that satisfies the following requirements:

1. The solution is unique;
2. The solution exists for arbitrary data; and
3. The solution depends continuously on the data.

Ill-posed problems, and hence most inverse problems, typically fail to satisfy at least one of these criteria. It is worth mentioning that this definition of an ill-posed problem applies to continuous mathematical models, and not precisely to the discrete approximations used in computational methods. However, the properties of the continuous ill-posed problem are often carried over to the discrete problem in the form of a particular kind of ill-conditioning, making certain (usually-high frequency) components of the solution very sensitive to errors in the measured data. Of course this may depend on the level of discretization; a coarsely discretized problem may not be very ill-conditioned, but it also may not bear much similarity to the underlying continuous problem.

Regularization is a term used to refer to various techniques that modify the inverse problem in an attempt to overcome the instability caused by ill-posedness. Regularization seeks to incorporate *a priori* knowledge into the solution process. Such knowledge may include information about the amount or type of noise, the smoothness or sparsity of the solution, or restrictions on the values the solution may obtain. Each regularization method also requires choosing one or more regularization parameters. A variety of approaches are discussed in this section.

The theory for regularizing linear problems is much more developed than it is for nonlinear problems. This is due, in large part, to the fact that the numerical treatment of nonlinear inverse problems is often highly dependent on the particular application. However, good intuition can be gained by first studying linear inverse problems.

### 1.2.1 Linear Inverse Problems

Consider the linear inverse problem

$$\boldsymbol{b} = A\boldsymbol{x}_{\text{exact}} + \boldsymbol{\eta}\,,$$

where $\boldsymbol{b}$ and $A$ are known, and the aim is to compute an approximation of $\boldsymbol{x}_{\text{exact}}$. The linear problem is a good place to illustrate the challenges that arise when attempting to solve large-scale inverse problems. In addition, some of the regularization methods and iterative algorithms discussed here can be used in, or generalized for, nonlinear inverse problems.

#### 1.2.1.1 SVD Analysis

A useful tool in studying linear inverse problems is the singular value decomposition (SVD). Any $m \times n$ matrix $A$ can be written as

$$A = U\Sigma V^T \tag{1.3}$$

where $U$ is an $m \times m$ orthogonal matrix, $V$ is an $n \times n$ orthogonal matrix, and $\Sigma$ is an $m \times n$ diagonal matrix containing the singular values $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_{\min(m,n)} \geq 0$. If $A$ is nonsingular, then an approximation of $\boldsymbol{x}_{\text{exact}}$ is given by the inverse solution

$$\boldsymbol{x}_{\text{inv}} = A^{-1}\boldsymbol{b} = \sum_{i=1}^{n} \frac{\boldsymbol{u}_i^T\boldsymbol{b}}{\sigma_i}\boldsymbol{v}_i = \underbrace{\sum_{i=1}^{n} \frac{\boldsymbol{u}_i^T\boldsymbol{b}_{\text{exact}}}{\sigma_i}\boldsymbol{v}_i}_{\boldsymbol{x}_{\text{exact}}} + \underbrace{\sum_{i=1}^{n} \frac{\boldsymbol{u}_i^T\boldsymbol{\eta}}{\sigma_i}\boldsymbol{v}_i}_{\text{error}}$$

where $\boldsymbol{u}_i$ and $\boldsymbol{v}_i$ are the singular vectors of $A$ (that is, the columns of $U$ and $V$, respectively). As indicated above, the inverse solution is comprised of two components: $\boldsymbol{x}_{\text{exact}}$ and an error term. Before discussing algorithms to compute approximations of $\boldsymbol{x}_{\text{exact}}$ it is useful to study the error term.

For matrices arising from ill-posed inverse problems, the following properties hold:

P1. The matrix $A$ is severely ill-conditioned, with the singular values $\sigma_i$ decaying to zero without a significant gap to indicate numerical rank.

P2. The singular vectors corresponding to the small singular values tend to oscillate more (i.e., have higher frequency) than singular vectors corresponding to large singular values.

P3. The components $|\boldsymbol{u}_i^T\boldsymbol{b}_{\text{exact}}|$ decay on average faster than the singular values $\sigma_i$. This is referred to as the *discrete Picard condition* [64].

The first two properties imply that the high frequency components of the error term are highly magnified by division of small singular values. The computed inverse solution is dominated by these high frequency components and is in general a very poor approximation of $\boldsymbol{x}_{\text{exact}}$. However, the third property suggests that there is hope

of reconstructing some information about $\boldsymbol{x}_{\text{exact}}$; that is, an approximate solution can be obtained by reconstructing components corresponding to the large singular values, and filtering out components corresponding to small singular values.

#### 1.2.1.2 Regularization by SVD Filtering

The SVD filtering approach to regularization is motivated by observations made in the previous subsection. That is, by filtering out components of the solution corresponding to the small singular values, a reasonable approximation of $\boldsymbol{x}_{\text{exact}}$ can be computed. Specifically, an SVD-filtered solution is given by

$$\boldsymbol{x}_{\text{filt}} = \sum_{i=1}^{n} \phi_i \frac{\boldsymbol{u}_i^T \boldsymbol{b}}{\sigma_i} \boldsymbol{v}_i \,, \tag{1.4}$$

where the *filter factors*, $\phi_i$, satisfy $\phi_i \approx 1$ for large $\sigma_i$, and $\phi_i \approx 0$ for small $\sigma_i$. That is, the large singular value components of the solution are reconstructed, while the components corresponding to the small singular values are filtered out. Different choices of filter factors lead to different methods. Some examples include:

| Truncated SVD Filter | Tikhonov Filter | Exponential Filter |
|---|---|---|
| $\phi_i = \begin{cases} 1 & \text{if } \sigma_i > \alpha \\ 0 & \text{if } \sigma_i \leq \alpha \end{cases}$ | $\phi_i = \dfrac{\sigma_i^2}{\sigma_i^2 + \alpha^2}$ | $\phi_i = 1 - e^{-\sigma_i^2/\alpha^2}$ |

Note that using a Taylor series expansion of the exponential term in the exponential filter, it is not difficult to see that the Tikhonov filter is a truncated approximation of the exponential filter. Moreover, the Tikhonov filter has an equivalent variational form, which is described in Section 1.2.1.3.

Observe that each of the filtering methods has a parameter (in the above examples, $\alpha$) that needs to be chosen to specify how much filtering is done. Appropriate values depend on properties of the matrix $A$ (i.e., on its singular values and singular vectors) as well as on the data, $\boldsymbol{b}$. Some techniques to help guide the choice of the regularization parameter are discussed in Section 1.2.1.6.

Because the SVD can be very expensive to compute for large matrices, this explicit filtering approach is generally not used for large-scale inverse problems. There are some exceptions, though, if $A$ is highly structured. For example, suppose $A$ can be

decomposed as a Kronecker product,

$$A = A_r \otimes A_c = \begin{bmatrix} a_{11}^{(r)} A_c & a_{12}^{(r)} A_c & \cdots & a_{1n}^{(r)} A_c \\ a_{21}^{(r)} A_c & a_{22}^{(r)} A_c & \cdots & a_{2n}^{(r)} A_c \\ \vdots & \vdots & & \vdots \\ a_{n1}^{(r)} A_c & a_{n2}^{(r)} A_c & \cdots & a_{nn}^{(r)} A_c \end{bmatrix}$$

where $A_c$ is an $m \times m$ matrix, and $A_r$ is an $n \times n$ matrix with entries denoted by $a_{ij}^{(r)}$. Then this block structure can be exploited when computing the SVD and when implementing filtering algorithms [67].

It is also sometimes possible to use an alternative factorization. Specifically, suppose that

$$A = Q \Lambda Q^* \,,$$

where $\Lambda$ is a diagonal matrix and $Q^*$ is the complex conjugate transpose of $Q$, with $Q^*Q = I$. This is called a spectral factorization; the columns of $Q$ are eigenvectors and the diagonal elements of $\Lambda$ are the eigenvalues of $A$. Although every matrix has an SVD, only normal matrices (i.e., matrices that satisify $A^*A = AA^*$) have a spectral decomposition. However, if $A$ has a spectral factorization, then it can be used, in place of the SVD, to implement the filtering methods described in this section. The advantage is that it is sometimes more computationally convenient to compute a spectral decomposition than it is an SVD.

#### 1.2.1.3 Variational Regularization and Constraints

Variational regularization methods have the form

$$\min_{\boldsymbol{x}} \left\{ \|\boldsymbol{b} - A\boldsymbol{x}\|_2^2 + \alpha^2 \mathcal{J}(\boldsymbol{x}) \right\} \,, \tag{1.5}$$

where the regularization operator $\mathcal{J}$ and the regularization parameter $\alpha$ must be chosen. The variational form provides a lot of flexibility. For example, one could include additional constraints on the solution, such as nonnegativity, or it may be preferable to replace the least squares criterion with the Poisson log likelihood function [8, 9, 11]. As with filtering, there are many choices for the regularization operator, $\mathcal{J}$, such as Tikhonov, total variation [26, 113, 128], and sparsity constraints [23, 42, 122]:

| Tikhonov | Total Variation | Sparsity |
|---|---|---|
| $\mathcal{J}(\boldsymbol{x}) = \|L\boldsymbol{x}\|_2^2$ | $\mathcal{J}(\boldsymbol{x}) = \left\| \sqrt{(D_h\boldsymbol{x})^2 + (D_v\boldsymbol{x})^2} \right\|_1$ | $\mathcal{J}(\boldsymbol{x}) = \|B\boldsymbol{x}\|_1$ |

Tikhonov regularization, which was first proposed and studied extensively in the early 1960s [91, 105, 117, 118, 119], is perhaps the most well-known approach to regularizing ill-posed problems. $L$ is typically chosen to be the identity matrix or a discrete approximation to a derivative operator, such as the Laplacian. If $L = I$, then it is not difficult to show that the resulting variational form of Tikhonov regularization, namely

$$\min_{\boldsymbol{x}} \left\{ \|\boldsymbol{b} - A\boldsymbol{x}\|_2^2 + \alpha^2 \|\boldsymbol{x}\|_2^2 \right\} , \tag{1.6}$$

can be written in an equivalent filtering framework by replacing $A$ with its SVD [64].

For total variation, $D_h$ and $D_v$ denote discrete approximations of horizontal and vertical derivatives of the 2D image $\boldsymbol{x}$, and the approach extends to 3D images in an obvious way. Efficient and stable implementation of total variation regularization is a nontrivial problem; see [26, 128] and the references therein for further details.

In the case of sparse reconstructions, the matrix $B$ represents a basis in which the image, $\boldsymbol{x}$, is sparse. For example, for astronomical images that contain a few bright objects surrounded by a significant amount of black background, an appropriate choice for $B$ might be the identity matrix. Clearly the choice of $B$ is highly dependent on the structure of the image $\boldsymbol{x}$. The usage of sparsity constraints for regularization is currently a very active field of research, with many open problems.

We also mention that when the majority of the elements in the image $\boldsymbol{x}$ are zero or near zero, as may be the case for astronomical or medical images, it may be wise to enforce nonnegativity constraints on the solution [9, 11, 128]. This requires that each element of the computed solution $\boldsymbol{x}$ is not negative, which is often written as $\boldsymbol{x} \geq \boldsymbol{0}$. Though these constraints add a level of difficulty when solving, they can produce results that are more feasible than when nonnegativity is ignored.

Finally it should be noted that, depending on the structure of matrix $A$, the type of regularization, and the additional constraints to include, a variety of optimization algorithms can be used to solve (1.5). In some cases it is possible to use a very efficient filtering approach, but typically it is necessary to use an iterative method.

#### 1.2.1.4 Iterative Regularization

As mentioned in Section 1.2.1.3, iterative methods are often needed to solve the variational form of the regularized problem. An alternate approach to using variational regularization is to simply apply the iterative method to the least squares problem,

$$\min_{\boldsymbol{x}} \|\boldsymbol{b} - A\boldsymbol{x}\|_2^2 .$$

Note that if an iterative method applied to this unregularized problem is allowed to "converge", it will converge to an inverse solution, $\boldsymbol{x}_{\text{inv}}$, which is corrupted by noise (recall the discussion in Section 1.2.1.1). However, many iterative methods have the property (provided the problem on which it is applied satisfies the discrete Picard condition) that the early iterations reconstruct components of the solution corresponding to large singular values, while components corresponding to small singular values are reconstructed at later iterations. Thus, there is an observed "semi-convergence" behavior in the quality of the reconstruction, whereby the approximate solution improves at early iterations and then degrades at later iterations (a more detailed discussion of this behavior is given in Section 1.2.1.5 in the context of the iterative method LSQR). If the iteration is terminated at an appropriate point, a regularized approximation of the solution is computed. Thus, the iteration index acts as the regularization parameter, and the associated scheme is referred to as an *iterative regularization method*.

Many algorithms can be used as iterative regularization methods, including Landweber [85], steepest descent, and the conjugate gradient method (e.g., for nonsymmetric problems the CGLS implementation [15] or the LSQR implementation [102, 103], and for symmetric indefinite problems, the MR-II implementation [57]). Most iterative regularization methods can be put into a general framework associated with solving the minimization problem

$$\min f(x) = \frac{1}{2}\boldsymbol{x}^T A^T A\boldsymbol{x} - \boldsymbol{x}^T A^T \boldsymbol{b} \tag{1.7}$$

with a general iterative method of the form

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \rho_k M_k \left(A^T\boldsymbol{b} - A^T A\boldsymbol{x}_k\right) = \boldsymbol{x}_k + \rho_k M_k \boldsymbol{r}_k\,, \tag{1.8}$$

where $\boldsymbol{r}_k = A^T\boldsymbol{b} - A^T A\boldsymbol{x}_k$. With specific choices of $\rho_k$ and $M_k$, one can obtain a variety of well-known iterative methods:

- The Landweber method is obtained by taking $\rho_k = \rho$ (that is, $\rho$ remains constant for each iteration), and $M_k = I$ (the identity matrix). Due to its very slow convergence, this classic approach is not often used for linear inverse problems. However, it is very easy to analyze the regularization properties of the Landweber iteration, and it can be useful for certain large-scale nonlinear ill-posed inverse problems.

- The steepest descent method is produced if $M_k = I$ is again fixed as the identity, but now $\rho_k$ is chosen to minimize the residual at each iteration. That is, $\rho_k$ is

chosen as

$$\rho_k = \arg\min_{\rho>0} f(\boldsymbol{x}_k + \rho \boldsymbol{r}_k).$$

Again, this method typically has very slow convergence, but with proper preconditioning it may be competitive with other methods.

- It is also possible to obtain the conjugate gradient method by setting $M_0 = I$ and $M_{k+1} = I - \dfrac{\boldsymbol{s}_k \boldsymbol{y}_k^T}{\boldsymbol{y}_k^T \boldsymbol{s}_k}$, where $\boldsymbol{s}_k = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k$ and $\boldsymbol{y}_k = A^T A\,(\boldsymbol{x}_{k+1} - \boldsymbol{x}_k)$. As with the steepest descent method, $\rho_k$ is chosen to minimize the residual at each iteration. Generally, the conjugate gradient method converges much more quickly than Landweber or steepest descent.

Other iterative algorithms that can be put into this general framework include the Brakhage $\nu$ methods [19] and Barzilai and Borwein's lagged steepest descent scheme [13].

#### 1.2.1.5 Hybrid Iterative-Direct Regularization

One of the main disadvantages of iterative regularization methods is that it can be very difficult to determine appropriate stopping criteria. To address this problem, work has been done to develop hybrid methods that combine variational approaches with iterative methods. That is, an iterative method, such as the LSQR implementation of the conjugate gradient method, is applied to the least squares problem $\min_{\boldsymbol{x}} \|A\boldsymbol{x} - \boldsymbol{b}\|_2^2$, and variational regularization is incorporated within the iteration process. To understand how this can be done, it is necessary to briefly describe how the LSQR iterates are computed.

LSQR is based on the Golub-Kahan (sometimes referred to as Lanczos) bidiagonalization (GKB) process. Given an $m \times n$ matrix $A$ and vector $\boldsymbol{b}$, the $k$-th GKB iteration computes an $m \times (k+1)$ matrix $W_k$, an $n \times k$ matrix $Y_k$, an $n \times 1$ vector $\boldsymbol{y}_{k+1}$, and a $(k+1) \times k$ bidiagonal matrix $B_k$ such that

$$A^T W_k = Y_k B_k^T + \gamma_{k+1} \boldsymbol{y}_{k+1} \boldsymbol{e}_{k+1}^T \tag{1.9}$$

$$A Y_k = W_k B_k, \tag{1.10}$$

where $\boldsymbol{e}_{k+1}$ denotes the $(k+1)$st standard unit vector and $B_k$ has the form

$$B_k = \begin{bmatrix} \gamma_1 & & & \\ \beta_2 & \gamma_2 & & \\ & \ddots & \ddots & \\ & & \beta_k & \gamma_k \\ & & & \beta_{k+1} \end{bmatrix}. \tag{1.11}$$

Matrices $W_k$ and $Y_k$ have orthonormal columns, and the first column of $W_k$ is $\boldsymbol{b}/\|\boldsymbol{b}\|_2$. Given these relations, an approximate solution $\boldsymbol{x}_k$ can be computed from the *projected* least squares problem

$$\min_{\boldsymbol{x}\in R(Y_k)} \|A\boldsymbol{x} - \boldsymbol{b}\|_2^2 = \min_{\hat{\boldsymbol{x}}} \|B_k\hat{\boldsymbol{x}} - \beta\boldsymbol{e}_1\|_2^2 \tag{1.12}$$

where $\beta = \|\boldsymbol{b}\|_2$, and $\boldsymbol{x}_k = Y_k\hat{\boldsymbol{x}}$. An efficient implementation of LSQR does not require storing the matrices $W_k$ and $Y_k$ and uses an efficient updating scheme to compute $\hat{\boldsymbol{x}}$ at each iteration; see [103] for details.

An important property of GKB is that for small values of $k$ the singular values of the matrix $B_k$ approximate very well certain singular values of $A$, with the quality of the approximation depending on the relative spread of the singular values; specifically, the larger the relative spread, the better the approximation [15, 49, 115]. For ill-posed inverse problems the singular values decay to and cluster at zero, such as $\sigma_i = O(i^{-c})$ where $c > 1$, or $\sigma_i = O(c^i)$, where $0 < c < 1$ and $i = 1, 2, \ldots, n$ [124, 125]. Thus the relative gap between large singular values is generally much larger than the relative gap between small singular values. Therefore, if the GKB iteration is applied to a linear system arising from discretization of an ill-posed inverse problem, then the singular values of $B_k$ converge very quickly to the largest singular values of $A$. The following example illustrates this situation.

**Example.** Consider a linear system obtained by discretization of a one-dimensional first kind Fredholm integral equation of the form

$$b(s) = \int_\Omega k(s,t)x(t)dt + \eta(s)\,, \tag{1.13}$$

where the kernel $k(s,t)$ is given by the Green's function for the second derivative and is constructed using `deriv2` in the Matlab package *Regularization Tools* [63]. This is a small-scale canonical ill-posed inverse problem that has properties found in imaging applications. The `deriv2` function constructs an $n \times n$ matrix $A$ from the kernel

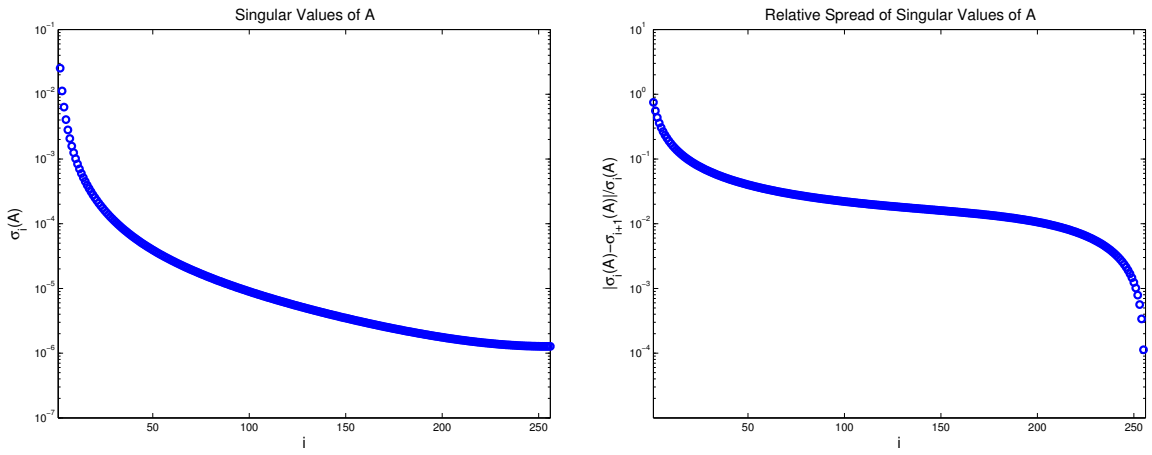
$$k(s,t) = \begin{cases} s\,(t-1) & \text{if } s < t \\ t\,(s-1) & \text{if } s \geq t \end{cases}$$

defined on $[0,1]\times[0,1]$. We use $n = 256$. There are also several choices for constructing vectors $\boldsymbol{x}_{\text{exact}}$ and $\boldsymbol{b}_{\text{exact}}$, (see [63]), but we focus only on the matrix $A$ in this example.

Figure 1.1 shows a plot of the singular values of $A$ and their relative spread; that is,

$$\frac{\sigma_i(A) - \sigma_{i+1}(A)}{\sigma_i(A)},$$

where the notation $\sigma_i(A)$ is used to denote the $i$th largest singular value of $A$. Figure 1.1 clearly illustrates the properties of ill-posed inverse problems; the singular values of $A$ decay to and cluster at 0. Moreover, it can be observed that in general the relative gap of the singular values is larger for large singular values and smaller for the smaller singular values. Thus for small values of $k$, the singular values of $B_k$ converge quickly to the large singular values of $A$. This can be seen in Figure 1.2, which compares the singular values of $A$ with those of the bidiagonal matrix $B_k$ for $k = 10, 20, 50$, also giving the relative difference $\dfrac{|\sigma_i(A) - \sigma_i(B_k)|}{\sigma_i(A)}$. □

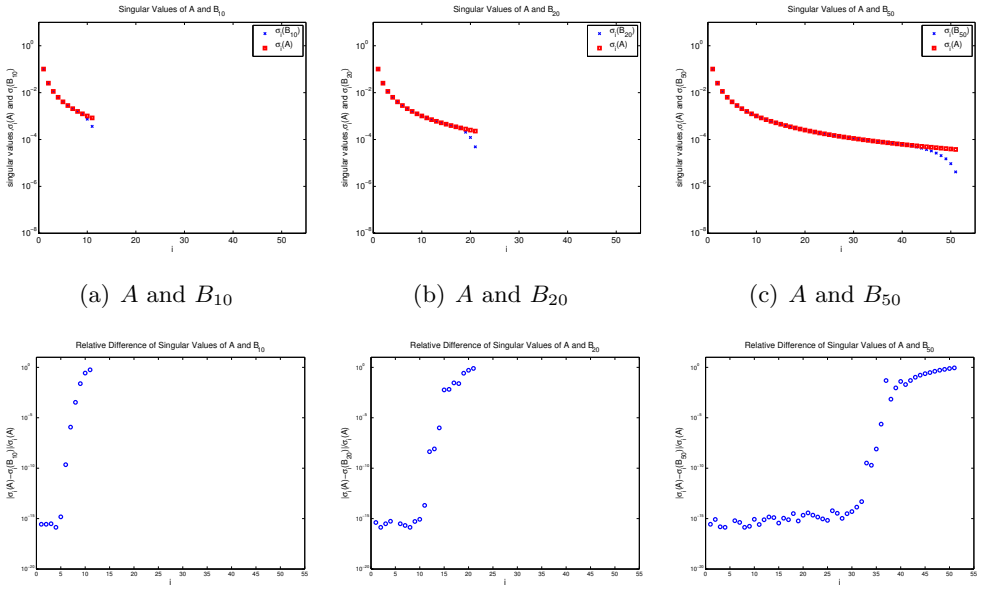



(a) Singular Values of $A$

(b) Relative Spread of Singular Values

Figure 1.1: The singular values of $A$ and their relative spread.

This example implies that if LSQR is applied to the least squares problem $\min_{\boldsymbol{x}} \|A\boldsymbol{x} - \boldsymbol{b}\|_2$, then at early iterations the approximate solutions $\boldsymbol{x}_k$ will be in a subspace that approximates a subspace spanned by the large singular components of $A$. Thus for $k \ll n$, $\boldsymbol{x}_k$ is a regularized solution. However, eventually $\boldsymbol{x}_k$ should converge to the inverse solution, which is corrupted with noise (recall the discussion in Section 1.2.1.4). This means that the iteration index $k$ plays the role of a regularization parameter; if $k$ is too small, then the computed approximation $\boldsymbol{x}_k$ is an over-smoothed solution,




(a) $A$ and $B_{10}$ (b) $A$ and $B_{20}$ (c) $A$ and $B_{50}$

Figure 1.2: Top Row: The singular values of $A$ and $B_k$, for $k = 10, 20, 50$. Bottom Row: The relative difference.

while if $k$ is too large, $\boldsymbol{x}_k$ is corrupted with noise. Again we emphasize that this semi-convergence behavior requires that the problem satisfies the discrete Picard condition. More extensive theoretical arguments of this semi-convergence behavior of conjugate gradient methods can be found elsewhere; see [57] and the references therein.

Instead of early termination of the iteration, hybrid approaches enforce regularization at each iteration of the GKB method. Hybrid methods were first proposed by O'Leary and Simmons in 1981 [100], and later by Björck in 1988 [14]. The basic idea is to regularize the projected least squares problem (1.12) involving $B_k$, which can be done very cheaply because of the smaller size of $B_k$. More specifically, because the singular values of $B_k$ approximate those of $A$, as the GKB iteration proceeds, the matrix $B_k$ becomes more ill-conditioned. The iteration can be stabilized by including Tikhonov regularization in the projected least square problem (1.12), to obtain

$$\min_{\hat{\boldsymbol{x}}} \left\{ \|B_k \hat{\boldsymbol{x}} - \beta \boldsymbol{e}_1\|_2^2 + \alpha^2 \|\hat{\boldsymbol{x}}\|_2^2 \right\} \tag{1.14}$$

where again $\beta = \|\boldsymbol{b}\|_2$ and $\boldsymbol{x}_k = Y_k \hat{\boldsymbol{x}}$. Thus at each iteration it is necessary to solve a regularized least squares problem involving a bidiagonal matrix $B_k$. Notice that since the dimension of $B_k$ is very small compared to $A$, it is much easier to solve for $\hat{\boldsymbol{x}}$ in equation (1.14) than it is to solve for $\boldsymbol{x}$ in the full Tikhonov regularized problem (1.6). More importantly, when solving equation (1.14) one can use sophisticated parameter

choice methods to find a suitable $\alpha$ at each iteration.

To summarize, hybrid methods have the following benefits:

- Powerful regularization parameter choice methods can be implemented efficiently on the projected problem.
- Semi-convergence behavior of the relative errors observed in LSQR is avoided, so an imprecise (over) estimate of the stopping iteration does not have a deleterious effect on the computed solution.

Realizing these benefits in practice, though, is nontrivial. Thus, various authors have considered computational and implementation issues, such as robust approaches to choose regularization parameters and stopping iterations; see for example, [16, 21, 30, 59, 84, 86, 100]. We also remark that our discussion of hybrid methods focused on the case of Tikhonov regularization with $L = I$. Implementation of hybrid methods when $L$ is not the identity matrix, such as a differentiation operator, can be nontrivial; see for example [66, 83].

#### 1.2.1.6 Choosing Regularization Parameters

Each of the regularization methods discussed in this section requires choosing a *regularization parameter*. It is a nontrivial matter to choose "optimal" regularization parameters, but there are methods that can be used as guides. Some require *a priori* information, such as a bound on the noise or a bound on the solution. Others attempt to estimate an appropriate regularization parameter directly from the given data.

To describe some of the more popular parameter choice methods, let $\boldsymbol{x}_{\text{reg}}$ denote a solution computed by a particular regularization method.

- **Discrepancy Principle.** In this approach a solution is sought such that

$$\|\boldsymbol{b} - A\boldsymbol{x}_{\text{reg}}\|_2 = \tau\|\boldsymbol{\eta}\|_2$$

  where $\tau > 1$ is a predetermined number [93]. This is perhaps the easiest of the methods to implement, and there are substantial theoretical results establishing its behavior in the presence of noise. However, it is necessary to have a good estimate for $\|\boldsymbol{\eta}\|_2$.

- **Generalized Cross Validation.** The idea behind generalized cross validation (GCV) is that if one data point is removed from the problem, then a good regularized solution should predict that missing data point well. If $\alpha$ is the

regularization parameter used to obtain $\boldsymbol{x}_{\text{reg}}$, then it can be shown [48] that the GCV method chooses $\alpha$ to minimize the function

$$G(\alpha) = \frac{\|\boldsymbol{b} - A\boldsymbol{x}_{\text{reg}}\|^2}{\left(\text{trace}\left(I - AA^{\dagger}_{\text{reg}}\right)\right)^2},$$

where $A^{\dagger}_{\text{reg}}$ is the matrix such that $\boldsymbol{x}_{\text{reg}} = A^{\dagger}_{\text{reg}}\boldsymbol{b}$. For example, in the case of Tikhonov regularization (1.6),

$$A^{\dagger}_{\text{reg}} = (A^T A + \alpha^2 I)^{-1} A^T.$$

A weighted version of GCV, W-GCV, finds a regularization parameter to minimize

$$G_{\omega}(\alpha) = \frac{\|\boldsymbol{b} - A\boldsymbol{x}_{\text{reg}}\|^2}{\left(\text{trace}\left(I - \omega AA^{\dagger}_{\text{reg}}\right)\right)^2}.$$

W-GCV is sometimes more effective at choosing regularization parameters than the standard GCV function for certain classes of problems. Setting the weight $\omega = 1$ gives the standard GCV method, while $\omega < 1$ produces less smooth solutions and $\omega > 1$ produces smoother solutions. Further details about W-GCV can be found in [30].

- **L-Curve.** This approach attempts to balance the size of the discrepancy (i.e., residual) produced by the regularized solution with the size of the solution. In the context of Tikhonov regularization, this can often be found by a log-log scale plot of $\|\boldsymbol{b} - A\boldsymbol{x}_{\text{reg}}\|_2$ versus $\|\boldsymbol{x}_{\text{reg}}\|_2$ for all possible regularization parameters. This plot often produces an L-shaped curve, and the solution corresponding to the corner of the L indicates a good balance between discrepancy and size of the solution. This observation was first made by Lawson and Hanson [87], and later studied extensively, including efficient numerical schemes to find the corner of the L (i.e., the point of maximum curvature), by Hansen [61, 68]. Although the L-curve tends to work well for many problems, some concerns about its effectiveness have been reported in the literature; see [58, 127].

There exist many other parameter choice methods besides the ones discussed above; for more information, see [35, 64, 128] and the references therein.

A proper choice of the regularization parameter is critical. If the parameter is chosen too small, then too much noise will be introduced in the computed solution. On the other hand, if the parameter is too large, the regularized solution may become over-smoothed and may not contain as much information about the true solution as it

could. However, it is important to keep in mind that no parameter choice method is "fool proof", and it may be necessary to solve the problem with a variety of parameters and use knowledge of the application to help decide which solution is best.

### 1.2.2 Separable Nonlinear Inverse Problems

Separable nonlinear inverse problems,

$$\boldsymbol{b} = A(\boldsymbol{x}^{(n\ell)}_{\text{exact}})\boldsymbol{x}^{(\ell)}_{\text{exact}} + \boldsymbol{\eta}\,, \tag{1.15}$$

arise in many imaging applications, such as blind deconvolution, super-resolution (which is an example of image data fusion) [28, 33, 80, 98], the reconstruction of 3D macromolecular structures from 2D electron microscopy images of cryogenically frozen samples (Cryo-EM) [31, 43, 73, 89, 104, 116], and in seismic imaging applications [55]. One could consider equation (1.15) as a general nonlinear inverse problem and use the approaches discussed in Section 1.2.3 to compute regularized solutions. However, this section considers approaches that exploit the separability of the problem. In particular, some of the regularization methods described in Section 1.2.1, such as variational and iterative regularization, can be adapted to equation (1.15). To illustrate, consider the general Tikhonov regularized least squares problem:

$$\min_{\boldsymbol{x}^{(\ell)},\boldsymbol{x}^{(n\ell)}} \left\{ \|A(\boldsymbol{x}^{(n\ell)})\,\boldsymbol{x}^{(\ell)} - \boldsymbol{b}\|_2^2 + \alpha^2 \|\boldsymbol{x}^{(\ell)}\|_2^2 \right\} = \min_{\boldsymbol{x}^{(\ell)},\boldsymbol{x}^{(n\ell)}} \left\| \begin{bmatrix} A(\boldsymbol{x}^{(n\ell)}) \\ \alpha I \end{bmatrix} \boldsymbol{x}^{(\ell)} - \begin{bmatrix} \boldsymbol{b} \\ \boldsymbol{0} \end{bmatrix} \right\|_2^2 . \tag{1.16}$$

Three approaches to solve this nonlinear least squares problem are outlined in this section.

#### 1.2.2.1 Fully Coupled Problem

The nonlinear least squares problem given in equation (1.16) can be rewritten as

$$\min_{\boldsymbol{x}} \phi(\boldsymbol{x}) = \min_{\boldsymbol{x}} \frac{1}{2} \|\rho(\boldsymbol{x})\|_2^2\,, \tag{1.17}$$

where

$$\rho(\boldsymbol{x}) = \rho(\boldsymbol{x}^{(\ell)}, \boldsymbol{x}^{(n\ell)}) = \begin{bmatrix} A(\boldsymbol{x}^{(n\ell)}) \\ \alpha I \end{bmatrix} \boldsymbol{x}^{(\ell)} - \begin{bmatrix} \boldsymbol{b} \\ \boldsymbol{0} \end{bmatrix}, \quad \text{and} \quad \boldsymbol{x} = \begin{bmatrix} \boldsymbol{x}^{(\ell)} \\ \boldsymbol{x}^{(n\ell)} \end{bmatrix}$$

Nonlinear least squares problems are solved iteratively, with algorithms having the general form:

**General Iterative Algorithm**

choose initial $\boldsymbol{x}_0 = \begin{bmatrix} \boldsymbol{x}_0^{(\ell)} \\ \boldsymbol{x}_0^{(n\ell)} \end{bmatrix}$

for $k = 0, 1, 2, \ldots$

- choose a step direction, $\boldsymbol{d}_k$
- determine step length, $\tau_k$
- update the solution: $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \tau_k \boldsymbol{d}_k$
- stop when a minimum of the objective is obtained

end

Typically $\boldsymbol{d}_k$ is chosen to approximate the Newton direction,

$$\boldsymbol{d}_k = -(\widehat{\phi}''(\boldsymbol{x}_k))^{-1}\phi'(\boldsymbol{x}_k)\,,$$

where $\widehat{\phi}''$ is an approximation of $\phi''$, $\phi' = J_\phi^T \rho$, and $J_\phi$ is the Jacobian matrix

$$J_\phi = \begin{bmatrix} \dfrac{\partial \rho(\boldsymbol{x}^{(\ell)}, \boldsymbol{x}^{(n\ell)})}{\partial \boldsymbol{x}^{(\ell)}} & \dfrac{\partial \rho(\boldsymbol{x}^{(\ell)}, \boldsymbol{x}^{(n\ell)})}{\partial \boldsymbol{x}^{(n\ell)}} \end{bmatrix}.$$

In the case of the Gauss-Newton method, which is often recommended for nonlinear least squares problems, $\widehat{\phi}'' = J_\phi^T J_\phi$.

This general Gauss-Newton approach can work well, but constructing and solving the linear systems required to update $\boldsymbol{d}_k$ can be very expensive. Note that the dimension of the matrix $J_\phi$ corresponds to the number of pixels in the image, $\boldsymbol{x}^{(\ell)}$, plus the number of parameters in $\boldsymbol{x}^{(n\ell)}$, and thus $J_\phi$ may be on the order of $10^6 \times 10^6$. Thus, instead of using Gauss-Newton, it might be preferable to use a low storage scheme such as the (nonlinear) conjugate gradient method. But there is a tradeoff – although the cost per iteration is reduced, the number of iterations needed to attain a minimum can increase significantly.

Relatively little research has been done on understanding and solving the fully coupled problem. For example, methods are needed for choosing regularization parameters. In addition, the rate of convergence of the linear and nonlinear terms may be quite different, and the effect this has on overall convergence rate is not well understood.

#### 1.2.2.2 Decoupled Problem

Probably the simplest idea to solve the nonlinear least squares problem is to decouple it into two problems, one involving $\boldsymbol{x}^{(\ell)}$ and the other involving $\boldsymbol{x}^{(n\ell)}$. Specifically, the approach would have the form:

**Block Coordinate Descent Iterative Algorithm**

choose initial $\boldsymbol{x}_0^{(n\ell)}$

for $k = 0, 1, 2, \ldots$

- choose $\alpha_k$ and solve the linear problem:

$$\boldsymbol{x}_k^{(\ell)} = \arg\min_{\boldsymbol{x}^{(\ell)}} \|A(\boldsymbol{x}_k^{(n\ell)})\boldsymbol{x}^{(\ell)} - \boldsymbol{b}\|_2^2 + \alpha_k^2 \|\boldsymbol{x}^{(\ell)}\|_2^2$$

- solve the nonlinear problem:

$$\boldsymbol{x}_{k+1}^{(n\ell)} = \arg\min_{\boldsymbol{x}^{(n\ell)}} \|A(\boldsymbol{x}^{(n\ell)})\boldsymbol{x}_k^{(\ell)} - \boldsymbol{b}\|_2^2 + \alpha_k^2 \|\boldsymbol{x}_k^{(\ell)}\|_2^2$$

- stop when objectives are minimized

end

The advantage of this approach is that any of the approaches discussed in Section 1.2.1, including methods to determine $\alpha$, can be used for the linear problem. The nonlinear problem involving $\boldsymbol{x}^{(n\ell)}$ requires using another iterative method, such as the Gauss-Newton method. However, there are often significantly fewer parameters than in the fully coupled approach discussed in the previous subsection. Thus, a Gauss-Newton method to update $\boldsymbol{x}_{k+1}^{(n\ell)}$ at each iteration is significantly more computationally tractable. A disadvantage to this approach, which is known in the optimization literature as block coordinate descent, is that it is not clear what are the practical convergence properties of the method. As mentioned in the previous subsection, the rate of convergence of the linear and nonlinear terms may be quite different. Moreover, if the method does converge, it will typically be very slow (linear), especially for problems with tightly coupled variables [99].

#### 1.2.2.3 Variable Projection Method

The variable projection method [50, 51, 82, 101, 114] exploits structure in the nonlinear least squares problem (1.16). The approach exploits the fact that $\phi(\boldsymbol{x}^{(\ell)}, \boldsymbol{x}^{(n\ell)})$ is linear in $\boldsymbol{x}^{(\ell)}$, and that $\boldsymbol{x}^{(n\ell)}$ contains relatively few parameters compared to $\boldsymbol{x}^{(\ell)}$.

However, rather than explicitly separating variables $\boldsymbol{x}^{(\ell)}$ and $\boldsymbol{x}^{(n\ell)}$ as in coordinate descent, variable projection implicitly eliminates the linear parameters $\boldsymbol{x}^{(\ell)}$, obtaining a reduced cost functional that depends only on $\boldsymbol{x}^{(n\ell)}$. Then a Gauss-Newton method is used to solve the optimization problem associated with the reduced cost functional. Specifically, consider

$$\psi(\boldsymbol{x}^{(n\ell)}) \equiv \phi(\boldsymbol{x}^{(\ell)}(\boldsymbol{x}^{(n\ell)}), \boldsymbol{x}^{(\ell)})$$

where $\boldsymbol{x}^{(\ell)}(\boldsymbol{x}^{(n\ell)})$ is a solution of

$$\min_{\boldsymbol{x}^{(\ell)}} \phi(\boldsymbol{x}^{(\ell)}, \boldsymbol{x}^{(n\ell)}) = \min_{\boldsymbol{x}^{(\ell)}} \left\| \begin{bmatrix} A(\boldsymbol{x}^{(n\ell)}) \\ \alpha I \end{bmatrix} \boldsymbol{x}^{(\ell)} - \begin{bmatrix} \boldsymbol{b} \\ \boldsymbol{0} \end{bmatrix} \right\|_2^2 . \tag{1.18}$$

To use the Gauss-Newton algorithm to minimize the reduced cost functional $\psi(\boldsymbol{x}^{(n\ell)})$, it is necessary to compute $\psi'(\boldsymbol{x}^{(n\ell)})$. Note that because $\boldsymbol{x}^{(\ell)}$ solves (1.18), it follows that $\dfrac{\partial \phi}{\partial \boldsymbol{x}^{(\ell)}} = 0$, and thus

$$\psi'(\boldsymbol{y}) = \frac{d\boldsymbol{x}}{d\boldsymbol{y}} \frac{\partial \phi}{\partial \boldsymbol{x}^{(\ell)}} + \frac{\partial \phi}{\partial \boldsymbol{x}^{(n\ell)}} = \frac{\partial \phi}{\partial \boldsymbol{x}^{(n\ell)}} = J_\psi^T \boldsymbol{\rho} \,,$$

where the Jacobian of the reduced cost functional is given by

$$J_\psi = \frac{\partial \left( A(\boldsymbol{x}^{(n\ell)}) \boldsymbol{x}^{(\ell)} \right)}{\partial \boldsymbol{x}^{(n\ell)}} \,.$$

Thus, a Gauss-Newton method applied to the reduced cost functional has the basic form:

**Variable Projection Gauss-Newton Algorithm**

choose initial $\boldsymbol{x}_0^{(n\ell)}$

for $k = 0, 1, 2, \ldots$

- choose $\alpha_k$
- $\boldsymbol{x}_k^{(\ell)} = \arg\min_{\boldsymbol{x}^{(\ell)}} \left\| \begin{bmatrix} A(\boldsymbol{x}_k^{(n\ell)}) \\ \alpha_k I \end{bmatrix} \boldsymbol{x}^{(\ell)} - \begin{bmatrix} \boldsymbol{b} \\ \boldsymbol{0} \end{bmatrix} \right\|_2$
- $\boldsymbol{r}_k = \boldsymbol{b} - A(\boldsymbol{x}_k^{(n\ell)})\, \boldsymbol{x}_k^{(\ell)}$
- $\boldsymbol{d}_k = \arg\min_{\boldsymbol{d}} \|J_\psi \boldsymbol{d} - \boldsymbol{r}_k\|_2$
- determine step length $\tau_k$
- $\boldsymbol{x}_{k+1}^{(n\ell)} = \boldsymbol{x}_k^{(n\ell)} + \tau_k \boldsymbol{d}_k$

end

Although computing $J_\psi$ is nontrivial, it is often much more tractable than constructing $J_\phi$. In addition, the problem of variable convergence rates for the two sets of parameters, $\boldsymbol{x}^{(\ell)}$ and $\boldsymbol{x}^{(n\ell)}$, has been eliminated. Another big advantage of the variable projection method for large-scale inverse problems is that standard approaches, such as those discussed in Section 1.2.1, can be used to solve the linear regularized least squares problem at each iteration, including the schemes for estimating regularization parameters.

### 1.2.3 Nonlinear Inverse Problems

Developing regularization approaches for general nonlinear inverse problems can be significantly more challenging than the linear and separable nonlinear case. Theoretical tools such as the SVD that are used to analyze ill-posedness in the linear case are not available here, and previous efforts to extend these tools to the nonlinear case do not always apply. For example, a spectral analysis of the linearization of a nonlinear problem does not necessarily determine the degree of ill-posedness for the nonlinear problem [37]. Furthermore, convergence properties for nonlinear optimization require very strict assumptions that are often not realizable in real applications [35, 36]. Nevertheless, nonlinear inverse problems arise in many important applications, motivating research on regularization schemes and general computational approaches. This section discusses some of this work.

One approach for nonlinear problems of the form

$$F(\boldsymbol{x}) = \boldsymbol{b} \tag{1.19}$$

is to reformulate the problem to find a zero of $F(\boldsymbol{x}) - \boldsymbol{b} = 0$. Then a Newton-like method, where the nonlinear function is repeatedly linearized around the current estimate, can be written as

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \rho_k \boldsymbol{p}_k \tag{1.20}$$

where $\boldsymbol{p}_k$ solves the Jacobian system

$$J(\boldsymbol{x}_k)\boldsymbol{p} = \boldsymbol{b} - F(\boldsymbol{x}_k)\,. \tag{1.21}$$

Though generally not symmetric, matrix and matrix-transpose multiplication with the Jacobian, whose elements are the first derivatives of $F(\boldsymbol{x})$, are typically computable. However, the main disadvantages of using this approach are that the existence and uniqueness of a solution are not guaranteed and the sensitivity of solutions

depends on the conditioning of the Jacobian. Furthermore, there is no natural merit function that can be monitored to help select the step length, $\rho_k$.

Another approach to solve (1.19) is to incorporate prior assumptions regarding the statistical distribution of the model and maximize the corresponding likelihood function. For example, an additive Gaussian noise model assumption under certain conditions corresponds to solving the following nonlinear least squares problem:

$$\min_{\boldsymbol{x}} \frac{1}{2}\|\boldsymbol{b} - F(\boldsymbol{x})\|_2^2. \tag{1.22}$$

Since this is a standard nonlinear optimization problem, any optimization algorithm such as a gradient descent or Newton approach can be used here. For problem (1.22), the gradient vector can be written as $g(\boldsymbol{x}) = J(\boldsymbol{x})^T(F(\boldsymbol{x}) - \boldsymbol{b})$ and Hessian matrix can be written as $H(\boldsymbol{x}) = J(\boldsymbol{x})^T J(\boldsymbol{x}) + Z(\boldsymbol{x})$, where $Z(\boldsymbol{x})$ includes second derivatives of $F(\boldsymbol{x})$. The main advantage of this approach is that a variety of line search methods can be used. However, the potential disadvantages of this approach are that the derivatives may be too difficult to compute or that negative eigenvalues introduced in $Z(\boldsymbol{x})$ may cause problems in optimization algorithms.

Some algorithms for solving nonlinear optimization problems are direct extensions of the iterative methods described in Section 1.2.1.4. The nonlinear Landweber iteration can be written as

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + J(\boldsymbol{x}_k)^T(\boldsymbol{b} - F(\boldsymbol{x}_k))\,, \tag{1.23}$$

which reduces to the standard Landweber iteration if $F(\boldsymbol{x})$ is linear, and it can be easily extended to other gradient descent methods such as the steepest descent approach. Newton and Newton-type methods are also viable options for nonlinear optimization, resulting in iterates (1.20) where $\boldsymbol{p}_k$ solves

$$H(\boldsymbol{x}_k)\boldsymbol{p} = -g(\boldsymbol{x}_k)\,. \tag{1.24}$$

Oftentimes, a quasi-Newton approach is used to approximate the Hessian. For example, the Gauss-Newton algorithm, which takes $H \approx J(\boldsymbol{x}_k)^T J(\boldsymbol{x}_k)$, is a preferred choice for large-scale problems because it ensures positive semi-definiteness, but is not advisable for large residual problems or highly nonlinear problems [55]. Additionally, quasi-Newton methods such as LBFGS, nonlinear Conjugate Gradient, and Truncated-Newton (Newton-CG) can be good alternatives if storage is a concern. It is important to remark that finding a global minimizer for a nonlinear optimization problem is in general very difficult, especially since convexity of the objective function

is typically not guaranteed, as in the linear case. Thus, it is very likely that a descent algorithm may get stuck in one of many local minima solutions.

When dealing with ill-posed problems, the general approach to incorporate regularization is to couple an iterative approach with a stopping criteria such as the discrepancy principle to produce reasonable solutions. In addition, for Newton-type methods it is common to incorporate additional regularization for the inner system since the Jacobian or Hessian may become ill-conditioned. For example, including linear Tikhonov regularization in (1.21) would result in

$$(J(\boldsymbol{x}_k)^T J(\boldsymbol{x}_k) + \alpha^2 I)\boldsymbol{p} = J(\boldsymbol{x}_k)^T(\boldsymbol{b} - F(\boldsymbol{x}_k)),$$

which is equivalent to a Levenberg-Marquardt iterate, where the update, $\boldsymbol{p}_k$, is the solution of a particular Tikhonov minimization problem:

$$\min_{\boldsymbol{p}} \|F(\boldsymbol{x}_k) + J(\boldsymbol{x}_k)\boldsymbol{p} - \boldsymbol{b}\|_2^2 + \alpha^2\|\boldsymbol{p}\|_2^2\,,$$

where $F(\boldsymbol{x})$ has been linearized around $\boldsymbol{x}_k$. Other variations for regularizing the update can be found in [35] and the references therein. Regularization for the inner system can also be achieved by solving the inner system inexactly using an iterative method and terminating the iterations early. These are called *inexact Newton* methods, and the early termination of the inner iterations is a good way to not only make this approach practical for large-scale problems but also to enforce regularization on the inner system.

The variational approaches discussed in Section 1.2.1.3 can be extended for the second class of algorithms where a likelihood function results in an nonlinear optimization problem. For example, after selecting a regularization operator $\mathcal{J}(\boldsymbol{x})$ and regularization parameter $\alpha$ for (1.22), the goal would be to solve a nonlinear optimization problem of the form

$$\min_{\boldsymbol{x}} \left\{ \|\boldsymbol{b} - F(\boldsymbol{x})\|_2^2 + \alpha^2 \mathcal{J}(\boldsymbol{x}) \right\} . \tag{1.25}$$

The flexibility in the choice of the regularization operator is nice, but selecting a good regularization parameter *a priori* can be a computationally demanding task, especially for large-scale problems. Some work on estimating the regularization parameter within a constrained optimization framework has been done [55, 56], but the most common approach for regularization of nonlinear ill-posed inverse problems is to use standard iterative methods to solve (1.22), where regularization is obtained via early termination of the iterations. It cannot be stressed enough that when using any iterative method to solve a nonlinear inverse problem where the regularization is

not already incorporated, a good stopping iteration for the outer iteration that serves as a regularization parameter is imperative. See also [2, 35, 36, 39, 72, 120, 126] for additional references on nonlinear inverse problems.

## 1.3 Outline of Work

In the rest of this work, we will consider linear inverse problems as described in Section 1.2.1. The majority of our work will be focused on iterative methods, as these are ideal approaches for solving large-scale inverse problems. Chapter 2 will describe two iterative solvers that we have implemented within the Trilinos framework [69]. We will then consider an application whereby we wish to remove patient motion blur from positron emission tomography brain scans. Chapter 3 will describe the problem more fully, including our formulation of the problem and results. Next we will look at a problem from adaptive optics. As described in Chapter 4, we wish to reconstruct a wavefront given noisy gradient information. We will consider two approaches to solving this problem, one a direct approach based on the truncated singular value decomposition, the other an iterative approach based on Tikhonov regularization. Finally, we will draw some conclusions in Chapter 5.

## 1.4 Contributions

This dissertation describes a number of contributions we have made, as detailed below.

- We have implemented two iterative methods, preconditioned LSQR and MRNSD, within the Belos framework of Trilinos. Our non-preconditioned LSQR implementation is being included in the Trilinos source code.
- We have also implemented a Belos status test for returning the iteration with least relative error, when the true solution is known.
- We have developed a new approach to solving the motion deblurring problem for PET brain images, when motion information is tracked and recorded during the scan. Our approach involves constructing a large, though sparse, matrix and solving a linear inverse problem using iterative methods.
- We have implemented our linear approach in both Matlab and C++, utilizing the Trilinos framework.

- We have used our implementations to test the effect of a number of variables, including scalar precision, number of intervals into which the motion information is divided, relative size of intervals, and patient motion.
- We have exploited properties of the Kronecker product and the generalized singular value decomposition to develop an efficient approach to determine a distorted wavefront given gradient information in an adaptive optics application.
- We have developed a technique to solve this problem that applies truncated SVD-type regularization in a direct fashion.
- We have also determined a preconditioner, allowing us to use Tikhonov regularization to solve. We have analyzed the approximation quality of our preconditioner.
- We have implemented our efficient approach for this adaptive optics problem in both Matlab and C++, allowing us to use our preconditioned LSQR code. Matrix-matrix multiplications and Hadamard (that is, element-wise) multiplications are used.
- We have verified the effectiveness of our approach by looking at the number of iterations required for different $\alpha$ values using Tikhonov regularization.
- We have also applied our square-aperture preconditioner to a problem with a circular or annular aperture with good results.

# Chapter 2

# Large-Scale Software for Inverse Problems

## 2.1 Motivation

For important scientific applications, one routinely has need of solving large-scale inverse problems. Due to their size and often lack of structure, iterative methods are commonly used to solve and provide regularization. The basic elements – distributed matrices and vectors, efficient matrix-vector operations, vector-update operations – are the same, regardless of which iterative algorithm is used to solve. Additionally, flexibility on the type of computer architecture is desirable, as different users have different environments available to them. A number of software packages have been developed to enable users to solve large-scale problems. Some of these are given in Section 2.2. The one we will focus on, Trilinos, is described in Section 2.3. In addition to providing a multitude of solvers for inverse problems, it is able to run on a variety of architectures, from serial devices to many-core parallel computers. Using the linear iterative framework provided by one of the packages of Trilinos, we have implemented two solvers: one for LSQR and the other for MRNSD. See Sections 2.4 and 2.5, respectively. Finally, for pedagogical reasons, the true solution may be known and we may wish to automatically stop iterations once the solution with least error is found; this procedure and its implementation are detailed in Section 2.6.

## 2.2 Previous Work

In the past fifteen years or so, a number of software libraries have been created to enable users to solve large-scale problems in parallel. We will discuss but a few here.

These include ScaLAPACK [17], PETSc [3, 4, 5], and Aztec [123]. Each has its own advantages and disadvantages, compared to Trilinos. ScaLAPACK is designed to be a scalable linear algebra package (hence the name). Support for general sparse matrices is not provided in ScaLAPACK, though support for banded and dense matrices is given. The least-squares solvers depend on either a QR or LQ factorization. PETSc is more similar to Trilinos as they both can solve sparse and dense distributed systems, and in fact, its matrix and vector libraries could be used within Trilinos. However, there are a number of differences, including that Trilinos is written mainly in C++ with an emphasis on packages. The parallel iterative library Aztec grew out of a solver for a specific application. Trilinos can access this library via an object-oriented interface called AztecOO; in fact, AztecOO even provides additional functionality not available in the original Aztec solver library.

It is important to note that other software packages that are not designed for parallel use but rather are designed to solve inverse problems utilizing regularization also exist. The ones we wish to highlight are written in Matlab and include Regularization Tools [65], RestoreTools [94], and MOORe Tools [78, 79]. Regularization Tools provides for both direct and iterative regularization, including a number of regularization parameter choice methods. However, it was designed mainly for small-scale problems, with many of the routines dependent on the SVD of the system matrix. The RestoreTools package is designed for image restoration; in particular, support for spatially variant blurs is included. An object-oriented design is employed, allowing for efficient storage of and solving methods for separable blurs and point spread function (PSF) objects. MOORe Tools also has an object-oriented design; its name is an acronym for Modular Object Oriented Regularization Tools.

## 2.3 Overview of Trilinos

The Trilinos project [69, 70, 71] grew out of a desire to assist in the design of algorithms and, at the same time, provide robust solvers for a variety of scientific applications. Currently, several dozen self-contained, publicly-available packages make up this solver library. Roughly translated from Greek, Trilinos means “a string of pearls”, referring to the fact that each individual package is a gem in its own right, but when combined, they create a mathematical software library worth more than the sum of its parts. It is worth noting that though there is a high level of interoperability, the interdependency of packages is intentionally low. The majority of Trilinos is written in C++, taking advantage of its object-oriented features, including

the ability to have abstract interfaces. These allow users to extend parts of Trilinos, if needed, to create solvers for specific problems, though concrete classes are provided that are scalable and quite robust. Let us now focus on just three aspects of Trilinos: the Petra object model, the all-purpose toolkit package of Teuchos, and the iterative linear solver package Belos.

### 2.3.1 Petra Model

The Petra object model [70] is the foundation for Trilinos, describing the basic objects of matrices, vectors, and graphs for use in a parallel, distributed memory environment. In the abstract, Petra was designed with the ability to do parallel data redistribution efficiently and easily. To that end, an important object is the map object (referred to in [70] as an ElementSpace object); this describes the layout of an object (be it a matrix, vector, or graph) across a parallel machine. The same map object can be, and often is, shared by multiple distributed objects. Map objects work by detailing which processor "manages" each element, each of which is identified by a unique global ID (GID). The GIDs may be in any order on any processor and have a multiplicity greater than one across all the processors (meaning that more than one processor owns a given GID). To fully define a distributed matrix or graph, four map objects are needed:

- *RowMap*: On each processor, this lists the GIDs that are managed by that processor for each row. Typically, this means that part or all of the data on that row is owned by this processor.
- *ColumnMap*: This is the same as RowMap, but regarding the distribution of columns instead of rows.
- *DomainMap*: On each processor, this lists the GIDs that are associated with a vector in the domain of the matrix; here, each GID must be uniquely associated with a single processor.
- *RangeMap*: This is the same as DomainMap, but regarding the distribution of vectors in the range of the matrix instead of in the domain. Again, the GIDs must have a multiplicity of exactly one.

For a square matrix of size $n \times n$ distributed in a typical linear style across $p$ processors where processor 0 owns the first $n/p$ rows, processor 1 owns the next $n/p$ rows, and so forth, the RowMap, DomainMap, and RangeMap may all be the same

object, while the ColumnMap would likely assert that each processor owns every column (that is, no column is uniquely owned by a single processor). However, as will be shown in Chapter 3, this need not be the case.

The map objects allow easy redistribution of data facilitating, for instance, the ability for a single processor to perform all the necessary I/O. The head processor may read all of the requisite data for a vector then send the needed values to each processor. Alternatively, one processor may collect all of the values from a solution vector and then write them to a file. These tasks, and others, can be effortlessly accomplished using map objects, without the user having to worry about the details.

Though the Petra object model is abstract, there are three concrete implementations in Trilinos: Epetra, Jpetra, and Tpetra. Epetra is restricted to using real-valued, double-precision data with integer GIDs. However, this is often sufficient for many users and was particularly useful before compilers that could handle all aspects of C++ were available. Jpetra is a pure Java implementation, with byte-code portability. Tpetra takes advantage of the templated aspects of C++, allowing real- or complex-valued, single- or double-precision data, or even a user-defined type such as a $2 \times 3$ matrix.

### 2.3.2 Teuchos Toolkit

For completeness, we must briefly mention the Teuchos package. This "toolkit"-type package contains utility classes that are useful to a variety of other packages in Trilinos. For our purposes, the `ParameterList` object in Teuchos allows us to easily pass solver options to our linear solvers. This affords us much flexibility in our code, for if certain options are present, the code may behave in one way; otherwise, perhaps, other behavior may be executed.

Another useful class is the `Teuchos::SerialDenseMatrix` class. This class allows us to construct templated rectangular matrices on a processor. The primary reason we need such matrices is the inclusion of a matrix-matrix multiplication routine. Currently Tpetra does not include such capabilities.

### 2.3.3 Belos Framework

The Belos package provides a powerful framework for iterative linear solvers. In addition, several implementations are given, including different variants of CG, Gram-Schmidt, and GMRES. At the heart of Belos is a linear problem class that contains information about the problem to be solved: an operator, optional preconditioners,

and left- and right-hand sides, among others. Note that the presence of an operator allows much freedom for the user; a matrix is a natural choice, but anything that is able to produce the effect of applying the linear operator to a vector is permissible.

Other classes that make up the Belos package include an abstract solver manager class, which details the functionality required for solvers; an iterator class, which can be extended to perform the specific iterations for a given solver algorithm; and an abstract status test class, which is used to determine if convergence has been reached. Concrete implementations of these classes, including a clever status test that combines the results of other status tests, are provided in Belos.

A small subset of the classes found in Belos and Tpetra as well as some interactions between them is modeled in Figure 2.1; here, italics signify abstractness. As can be seen, the `SolverManager` class in Belos is purely virtual, though any class that extends it would likely require a `Teuchos::ParameterList` object and a `Belos::LinearProblem` object as class members. This shows the separation between solver parameters, which are provided in the parameter list, and the linear problem, which is given in the `LinearProblem` class. The `LinearProblem` itself is composed of an operator object as well as two multivector objects. A common choice for the virtual operator object is a `Tpetra::CrsMatrix`; similarly, a `Tpetra::MultiVector` would work for the virtual multivector objects. Both of these objects require one or more `Tpetra::Map` objects to describe their parallel distribution. The `Belos::Iteration` object describes the steps to be performed during each iteration of the solver algorithm, while the `Belos::StatusTest` class is used to test for convergence.

## 2.4 LSQR

LSQR, described in Section 1.2.1.5, is a well-known iterative solver when the transpose of the (possibly rectangular) operator is available. See [102, 103] for more details.

A right-preconditioned version of the LSQR algorithm is found in [81]. Here the right preconditioner $M$ is based on an incomplete LU factorization of the inverse of the normal matrix $A^TA$. However, other choices for the preconditioner also produce good results. Both the original, unpreconditioned LSQR algorithm and the preconditioned algorithm from are given. The necessary changes for the preconditioned algorithm are highlighted in red.

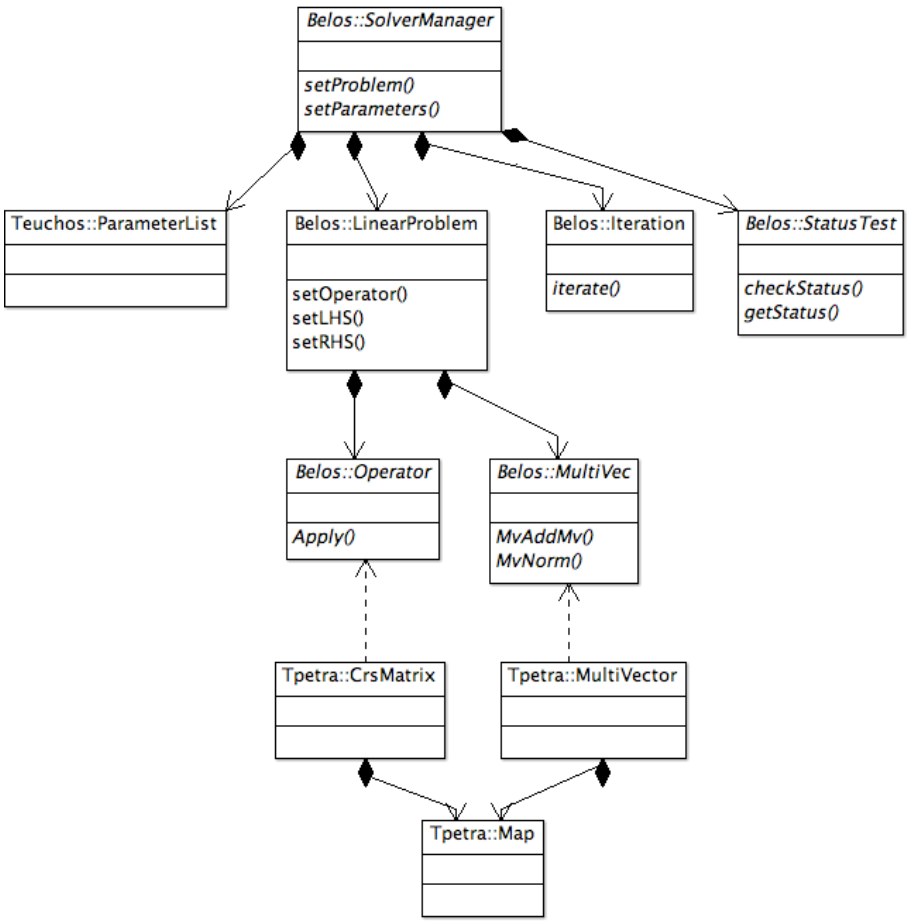



Figure 2.1: Brief diagram of key classes from Belos and Tpetra.

**Original LSQR Algorithm**

set $\boldsymbol{x}_0 = 0$

compute $\beta_0 = \|\boldsymbol{b}\|_2, \boldsymbol{u}_0 = \boldsymbol{b}/\beta_0, \alpha_0 = \|A^T\boldsymbol{u}_0\|_2, \boldsymbol{v}_0 = A^T\boldsymbol{u}_0/\alpha_0$

set $\boldsymbol{w}_0 = \boldsymbol{v}_0, \bar{\phi}_0 = \beta_0, \bar{\rho}_0 = \alpha_0$

for $k = 0, 1, 2, \ldots$

- $\beta_{k+1} = \|A\boldsymbol{v}_k - \alpha_k\boldsymbol{u}_k\|_2$
- $\boldsymbol{u}_{k+1} = (A\boldsymbol{v}_k - \alpha_k\boldsymbol{u}_k)/\beta_{k+1}$
- $\boldsymbol{z}_k = A^T\boldsymbol{u}_{k+1} - \beta_{k+1}\boldsymbol{v}_k$
- $\alpha_{k+1} = \|\boldsymbol{z}_k\|_2$
- $\boldsymbol{v}_{k+1} = \boldsymbol{z}_k/\alpha_{k+1}$
- $\rho_k = \left(\bar{\rho}_k^2 + \beta_{k+1}^2\right)^{1/2}$
- $c_k = \bar{\rho}_k/\rho_k$
- $s_k = \beta_{k+1}/\rho_k$
- $\theta_{k+1} = s_k\alpha_{k+1}$
- $\bar{\rho}_{k+1} = -c_k\alpha_{k+1}$
- $\phi_k = c_k\bar{\phi}_k$
- $\bar{\phi}_{k+1} = s_k\bar{\phi}_k$
- $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + (\phi_k/\rho_k)\,\boldsymbol{w}_k$
- $\boldsymbol{w}_{k+1} = \boldsymbol{v}_{k+1} - (\theta_{k+1}/\rho_k)\,\boldsymbol{w}_k$
- test for convergence

end

**LSQR Algorithm with Right Preconditioning**

compute approximate inverse factor $M$

set $\boldsymbol{y}_0 = 0$

compute $\beta_0 = \|\boldsymbol{b}\|_2, \boldsymbol{u}_0 = \boldsymbol{b}/\beta_0$, $\boldsymbol{q}_0 = A^T\boldsymbol{u}_0, \alpha_0 = \|M^T\boldsymbol{q}_0\|_2, \boldsymbol{v}_0 = M^T\boldsymbol{q}_0/\alpha_0$

set $\boldsymbol{w}_0 = \boldsymbol{v}_0, \bar{\phi}_0 = \beta_0, \bar{\rho}_0 = \alpha_0$

for $k = 0, 1, 2, \ldots$

- $\boldsymbol{p}_k = M\boldsymbol{v}_k$
- $\beta_{k+1} = \|A\boldsymbol{p}_k - \alpha_k\boldsymbol{u}_k\|_2$
- $\boldsymbol{u}_{k+1} = (A\boldsymbol{p}_k - \alpha_k\boldsymbol{u}_k)/\beta_{k+1}$
- $\boldsymbol{q}_{k+1} = A^T\boldsymbol{u}_{k+1}$
- $\boldsymbol{z}_k = M^T\boldsymbol{q}_{k+1} - \beta_{k+1}\boldsymbol{v}_k$
- $\alpha_{k+1} = \|\boldsymbol{z}_k\|_2$
- $\boldsymbol{v}_{k+1} = \boldsymbol{z}_k/\alpha_{k+1}$
- $\rho_k = \left(\bar{\rho}_k^2 + \beta_{k+1}^2\right)^{1/2}$
- $c_k = \bar{\rho}_k/\rho_k$
- $s_k = \beta_{k+1}/\rho_k$
- $\theta_{k+1} = s_k\alpha_{k+1}$
- $\bar{\rho}_{k+1} = -c_k\alpha_{k+1}$
- $\phi_k = c_k\bar{\phi}_k$
- $\bar{\phi}_{k+1} = s_k\bar{\phi}_k$
- $\boldsymbol{y}_{k+1} = \boldsymbol{y}_k + (\phi_k/\rho_k)\,\boldsymbol{w}_k$
- $\boldsymbol{w}_{k+1} = \boldsymbol{v}_{k+1} - (\theta_{k+1}/\rho_k)\,\boldsymbol{w}_k$
- if $\left|\bar{\phi}_{k+1}\right|$ is small enough then compute $\boldsymbol{x}_{k+1} = M\boldsymbol{y}_{k+1}$

test for convergence

end

### 2.4.1 Implementation Details of LSQR

As written, the `Belos::LinearProblem` class in Trilinos is designed to work only with square matrices. However, by changing just a few lines in the source code and recompiling Trilinos, it now works with rectangular matrices. In particular, we needed to change which vector was cloned when constructing the residual vector. The cloning determines the size of the vector. For square matrices, the left- and right-side vectors will be the same size. For rectangular matrices, though, the residual vector should be the same size as the right-side vector.

Our LSQR implementation is based on the C++ implementation by John Tomlin [121], which in turn is based on the C version by James Howse [74].

## 2.5 MRNSD

MRNSD stands for Modified Residual Norm Steepest Descent. It is an iterative scheme that enforces nonnegativity at each iteration, making it useful when it is known that the true solution has only positive values or zero, as is often the case for imaging problems [60, 95]. In particular, when the majority of pixel values are at or near zero, reconstructions are often significantly better when nonnegativity is enforced.

Following the procedure given in [95], let us derive an algorithm for MRNSD. We wish to minimize

$$\Phi(\boldsymbol{x}) = \frac{1}{2}\|\boldsymbol{b} - A\boldsymbol{x}\|^2 \tag{2.1}$$

subject to the constraint $\boldsymbol{x} \geq 0$. We use the parameterization $\boldsymbol{x} = e^{\boldsymbol{z}}$, defined element-wise, then compute the gradient taking advantage of the chain rule. Defining $X = \operatorname{diag}(\boldsymbol{x})$ for convenience, we have

$$\operatorname{grad}_{\boldsymbol{z}}\Phi(\boldsymbol{x}) = X \operatorname{grad}_{\boldsymbol{x}}\Phi(\boldsymbol{x}) = XA^T(A\boldsymbol{x} - \boldsymbol{b}).$$

We obtain the KKT conditions by setting the gradient to be zero; that is, $\operatorname{grad}_{\boldsymbol{z}}\Phi(\boldsymbol{x}) = 0$.

To write out the *modified* RNSD iterative method of the form

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \tau_k \boldsymbol{d}_k,$$

where $\boldsymbol{x}_k$ represents the solution at the $k$th iteration, we set $\boldsymbol{d}_k$ to be $X_k A^T (A\boldsymbol{x}_k - \boldsymbol{b})$. This is the direction of the negative gradient. We use a line search to find $\tau_k$ to minimize the residual norm, but it must be bounded to ensure nonnegativity. Thus,

we can write the MRNSD algorithm as follows; here $\boldsymbol{x}_k[i]$ refers to the $i$th element of $\boldsymbol{x}_k$ and similarly for $\boldsymbol{d}_k$.

**MRNSD Algorithm**

choose initial nonnegative $\boldsymbol{x}_0$

$\boldsymbol{g}_0 = A^T(A\boldsymbol{x}_0 - \boldsymbol{b})$

$X_0 = \text{ diag}(\boldsymbol{x}_0)$

$\gamma_0 = \boldsymbol{g}_0^T X_0 \boldsymbol{g}_0$

for $k = 0, 1, 2, \ldots$

- $\boldsymbol{d}_k = -X_k \boldsymbol{g}_k$
- $\boldsymbol{u}_k = A\boldsymbol{d}_k$
- $\tau_k = \min\left(\gamma_k / \boldsymbol{u}_k^T \boldsymbol{u}_k, \min_{\boldsymbol{d}_k[i]<0}\left(-\boldsymbol{x}_k[i]/\boldsymbol{d}_k[i]\right)\right)$
- $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \tau_k \boldsymbol{d}_k$
- $X_{k+1} = \text{ diag}(\boldsymbol{x}_{k+1})$
- $\boldsymbol{z}_k = A^T \boldsymbol{u}_k$
- $\boldsymbol{g}_{k+1} = \boldsymbol{g}_k + \tau_k \boldsymbol{z}_k$
- $\gamma_{k+1} = \boldsymbol{g}_{k+1}^T X_{k+1} \boldsymbol{g}_{k+1}$

end

### 2.5.1 Implementation Details of MRNSD

The implementation of MRNSD into the Trilinos framework is fairly straightforward, though several changes to the given algorithm can be made for the sake of efficiency. For instance, the number of intermediary vectors can be reduced. Additionally, since the only computations involving $X$ are matrix-vector products, these can be done as component-wise (Hadamard) products with the vector $\boldsymbol{x}$.

One key part of the implementation of MRNSD into Trilinos is the choice of the initial guess, if not provided by the user. By default, a vector is initialized to all zeros in Trilinos. If the user opts not to change this, then an all-zero vector would be passed into MRNSD as the initial guess. However, due to the nature of MRNSD, when a component of the solution $\boldsymbol{x}$ becomes zero, it stays zero. Therefore this method cannot progress if an all-zero vector is passed as the initial guess. An initial vector with norm zero is detected and replaced with a vector (usually) containing the

mean value of the given right-hand side. Additionally, if the initial guess contains any negative components, we compensate for them by adding a positive scalar to every entry.

Initially we tried to use the square root of machine epsilon for the given floating-point data type as the value to insert in the initial guess (in the case of an all-zero vector). However, this may cause a variety of problems during the execution of the iterative method due to its closeness to zero. For example, rounding errors in finite-precision arithmetic could cause some values to be small numbers (negative or positive) rather than zero. This may cause an incorrect $\tau$ value to be computed on occasion, which in turn may cause more iterations to be run than necessary. However, if the mean value of the right-hand side is smaller than the square root of machine epsilon, the latter value is used instead in the case of an all-zero initial guess.

## 2.6 Least Error Convergence Test

One aspect of the Belos iterative solver framework is the separation of the linear solver from the convergence testing. Thus the number of iterations, which is a common test for many different solvers, can be used by any solver. Additionally any other tests can also be used to check for convergence. LSQR and MRNSD both have their tests, but if the true solution is known, then a least error convergence test may also be used.

The least error convergence test works by computing the error in the initial guess. It then makes its own copy of the initial solution and keeps track of the error. At each iteration it computes the new error. If the current iteration produces a solution with worse error, then a counter is incremented. Once this counter meets a given window size, convergence is declared. On the other hand, if the error at this iteration is less than the current "best" error, it replaces the previous "best" solution with the current iterate, updates the saved error information, and resets the counter.

Some iterative methods, including LSQR, produce a convergence curve that steadily decreases to a minimum, then increases as more iterations are performed and the impact of small singular values on the noise becomes apparent. In this situation, a small window size would be sufficient. Other iterative methods, such as MRNSD when begun with a poor initial guess, produce a bumpier convergence curve with multiple local minima. A large window size that would likely be able to find the true global minimum is better in these cases.

Regardless of the window size, once the least error convergence test has determined convergence, it must make some updates to the `Belos::LinearProblem` ob-

ject. Specifically, it must change the values in the left-hand side vector to reflect those that yield a solution with minimal error; additionally, the number of iterations must be changed to reflect the true iteration number of the solution. It must be noted, therefore, that using the least error convergence test will cause several more iterations to be performed past the number returned when the linear problem object is queried post-solve – how many iterations more is dependent upon the window size.

## 2.7 Remarks and Future Directions

Trilinos provides a wonderful framework for the development of algorithms by providing both abstract interfaces and efficient concrete implementations. The Belos package provides a variety of iterative solvers, while allowing users to add their own solvers. Additionally, new features are being added continually. For these reasons, it may be desirable to implement a hybrid iterative method, such as HyBR [30], into this framework in the future. As the solvers are not dependent upon a matrix, it may be useful to implement point spread function objects and similar structures as is done in RestoreTools [94]. It may also be beneficial to have some methods to estimate regularization parameters, as discussed in Section 1.2.1.6.

# Chapter 3

# Case Study 1: Positron Emission Tomography Application

During a positron emission tomography (PET) scan, a patient may move. These movements degrade the reconstructed image, especially as the resolution of the scanner increases. When imaging a rigid object, such as the brain, these movements may be tracked and recorded with fairly high precision. Then, this information may be used in the deconvolution process to produce a better, sharper image.

We will consider the motivation behind removing motion blur from PET brain images in Section 3.1 as well as some previous approaches in Section 3.2. We will next describe our approach in Section 3.3, giving specific implementation details in Section 3.4. Finally, we will show some results on simulated data in Section 3.5, with concluding remarks in Section 3.6. For additional results using actual clinical images, see [130].

## 3.1 Motivation

When positron emission tomography is used for brain imaging, movement of the patient's head during the scanning process introduces motion blur, and thus reduces the resolution of the reconstructed image. While some patient motion can be tolerable in low-resolution imaging systems, with new PET scanners, even a small amount of motion can degrade image quality. The resolution of the latest PET scanners approaches 2 mm, but this 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. 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, 54]. 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) [106], magnetic field [54] and infrared [45, 90] 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.

## 3.2 Previous Work

Motion correction methods that have been reported fall into three general categories [41]. Sinogram rebinning described by Bloomfield [18], Buhler [20], Menke [90] and Rahmim [108] uses 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 [106] 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, the known patient motion has been incorporated into a system response function used during maximum likelihood expectation maximization (MLEM) reconstruction of the emission image [109]. Since this method involves system matrix modification, it requires a detailed understanding of the geometry of the scanner as well as detector response characteristics and attenuation.

In [41], an MLEM-based deconvolution algorithm was implemented that worked directly on the reconstructed image and hence no additional information specific to the scanner was required. Head position was detected using the VICRA optical tracking system and a system matrix was computed using the head motion data. This matrix was used to deconvolve the motion-corrupted reconstruction. In software simulations and physical phantom experiments, significant improvement in contrast

and accuracy with these deconvolution methods was shown. Improvement depended upon the noise level and the amount of motion. However, direct implementation of the EM deconvolution algorithm has distinct disadvantages. The system matrix is very large ($n^2 \times n^2$, where $n$ is the total number of voxels in the volume) and generally cannot be stored in the memory of most standard PCs. In addition, the number of operations is large and consists mainly of matrix multiplications. These problems were somewhat addressed by using a modification of the ordered subset technique [76, 107]. The subsets were defined in image space rather than in projection space as is normally done. In addition, IDL's (ITT Visual Information Solutions, Boulder, CO) sparse matrix capability for the matrix operations was used. Even with these modifications, the time taken for deconvolution was still large. This was in part because each matrix subset still had to be separately calculated and written out to the hard disk and then read back at each iteration during the deconvolution step. In addition, the time taken for deconvolution increased with the number of head movements and the number of subsets used.

## 3.3 Methodology of Our Approach

The deconvolution used in our work requires solving a large-scale inverse problem of the form

$$\boldsymbol{b} = A\boldsymbol{x} + \boldsymbol{\eta} \tag{3.1}$$

where $\boldsymbol{b}$ is a vector representing the motion-blurred reconstructed image, $\boldsymbol{x}$ is a vector that represents the true object, and $\boldsymbol{\eta}$ is additive noise. The matrix $A$ models the motion blur, which is highly spatially variant. Thus standard methods based on the fast Fourier transform (FFT), such as the Wiener filter, cannot be used for the deconvolution. Instead, it is necessary to use iterative methods to compute an approximation of $\boldsymbol{x}$. Quality of the reconstruction depends on how well the motion information can be estimated, which in turn provides necessary information to construct the matrix $A$. Computational efficiency is obtained by exploiting modern sparse matrix techniques.

### 3.3.1 Motion Detection

Following the procedure from [107], a set of targets, which is composed of four passive markers that reflect infrared light, is attached to the patient's head using a modified swimming cap. A motion tracker emits infrared light, which is reflected off the four markers, and their orientation (as a unit quaternion) and position (as a vector of

length three) are calculated. This provides motion information in six degrees of freedom, which can be equivalently written in terms of an affine transformation. These measurements are made multiple times per second and stored, resulting in fairly accurate motion information. In particular, this information represents the transformation between the reference and target coordinate frames for some orientation and position of the head.

### 3.3.2 Construction of the Matrix

In this section we describe an approach to model the motion blur that allows for efficient construction of the large and sparse matrix $A$. To simplify the discussion we describe the process for two-dimensional images; extension to three-dimensional images is straightforward. The basic idea is to assume the motion-blurred image is the (normalized) sum of images at incremental times during acquisition. Each of the individual images represents a snapshot of the object in a fixed position. To obtain a mathematical model, let $x(s,t)$ be a continuous function representing the object, and let $X$ be a discrete image, whose $(i,j)$ entry is given by

$$X(i,j) = x(s_i, t_j), \quad i = 1, 2, \ldots n, \quad j = 1, 2, \ldots n.$$

Now suppose $X_1$ is a discrete image obtained from the object $x$ after a rigid movement. Then there is an affine transformation $\mathcal{A} \in \mathcal{R}^{3\times 3}$ such that

$$\begin{bmatrix} \hat{s}_i \\ \hat{t}_j \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_i \\ t_j \\ 1 \end{bmatrix}$$

and

$$X_1(i,j) = x(\hat{s}_i, \hat{t}_j).$$

Note that because the continuous image $x$ is not known at every point $(s,t)$ – all that is known is the discrete image $X$ – it may not be possible to evaluate $x(\hat{s}_i, \hat{t}_j)$, unless $\hat{s}_i = s_{\hat{i}}$ and $\hat{t}_j = t_{\hat{j}}$ for some $1 \le \hat{i} \le n$ and $1 \le \hat{j} \le n$. However, an approximation of $x(\hat{s}_i, \hat{t}_j)$ can be computed by interpolating known values of $x$ near $x(\hat{s}_i, \hat{t}_j)$. Suppose as illustrated in Figure 3.1 that $x(s_{\hat{i}}, t_{\hat{j}})$, $x(s_{\hat{i}+1}, t_{\hat{j}})$, $x(s_{\hat{i}}, t_{\hat{j}+1})$ and $x(s_{\hat{i}+1}, t_{\hat{j}+1})$ are four known pixel values surrounding the unknown value $x(\hat{s}_i, \hat{t}_j)$. Nearest neighbor interpolation uses the known pixel value closest to $x(\hat{s}_i, \hat{t}_j)$; for example, in the illustration in Figure 3.1 we have

$$X_1(i,j) = x(\hat{s}_i, \hat{t}_j) \approx x(s_{\hat{i}}, t_{\hat{j}+1}).$$

In the case of bilinear interpolation, a weighted average of the four pixels surrounding $x(\hat{s}_i, \hat{t}_j)$ is used for the approximation:

$$\begin{aligned} X_1(i,j) &= x(\hat{s}_i, \hat{t}_j) \\ &\approx (1-\Delta s_i)(1-\Delta t_j)x(s_{\hat{i}}, t_{\hat{j}}) \\ &\quad + (1-\Delta s_i)\Delta t_j x(s_{\hat{i}}, t_{\hat{j}+1}) \\ &\quad + \Delta s_i(1-\Delta t_j)x(s_{\hat{i}+1}, t_{\hat{j}}) \\ &\quad + \Delta s_i \Delta t_j x(s_{\hat{i}+1}, t_{\hat{j}+1}), \end{aligned}$$

where $\Delta s_i = \hat{s}_i - s_{\hat{i}}$ and $\Delta t_j = \hat{t}_j - t_{\hat{j}}$.

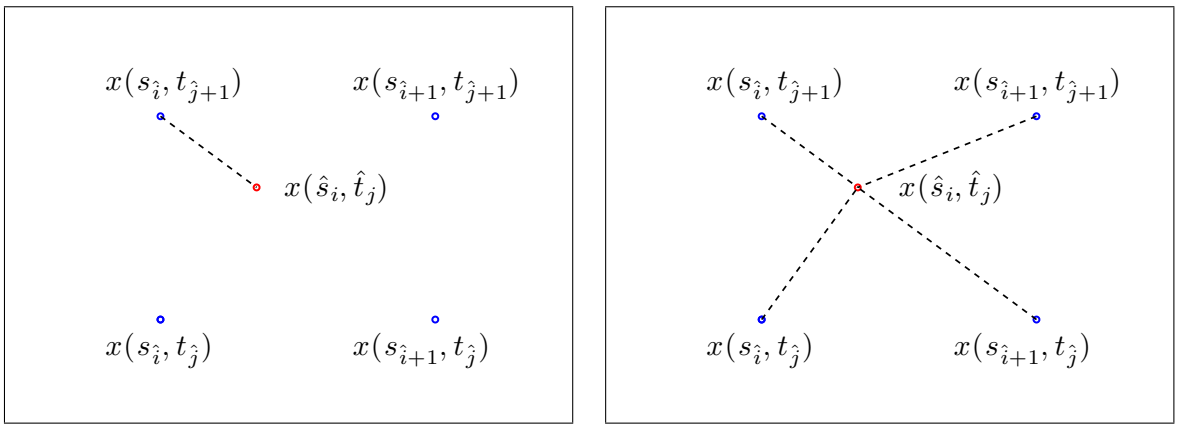



Figure 3.1: Illustration of two interpolation schemes to approximate the value of $x(\hat{s}_i, \hat{t}_j)$.

If we define vectors $\boldsymbol{x} = \mathrm{vec}(X)$ and $\boldsymbol{x}_1 = \mathrm{vec}(X_1)$ from the discrete image arrays (e.g., through lexicographical ordering), we can write

$$\boldsymbol{x}_1 = A_1 \boldsymbol{x}$$

where $A_1$ is a sparse matrix that contains the interpolation weights. Specifically, the $k$th row of $A_1$ contains the weights for the pixel in the $k$th entry of $\boldsymbol{x}_1$. For example, in the case of bilinear interpolation, there are at most four nonzero entries per row, given by

$$(1-\Delta s_i)(1-\Delta t_j),\ (1-\Delta s_i)\Delta t_j,\ \Delta s_i(1-\Delta t_j),\ \Delta s_i \Delta t_j.$$

In the case of nearest neighbor interpolation, there is just one nonzero entry in each row. We emphasize that by using a sparse data format (e.g., compressed row [32]) to represent $A$, we need only keep track of the nonzero entries and their locations in the matrix $A$. Moreover, this discussion assumes the affine transformation $\mathcal{A}$ is

known, because this provides the necessary information to construct the interpolation weights.

As previously discussed, we assume the observed motion-blurred image is the (normalized) sum of images at incremental times during the acquisition. That is, we assume

$$\boldsymbol{b} = \sum_{\ell=1}^{m} \omega_\ell \boldsymbol{x}_\ell + \boldsymbol{\eta}$$

where $\boldsymbol{x}_\ell = A_\ell \boldsymbol{x}$ is a vector representing the discrete image at time $t_\ell$, $\omega_\ell$ is the normalization weight for the $\ell$th image (for example, we could simply use $\omega_\ell = \frac{1}{m}$), and $\boldsymbol{\eta}$ is additive noise. Furthermore, we assume that the position of the object at time $t_\ell$ is known, and thus we can construct the sparse matrix $A_\ell$. Thus, we obtain the linear inverse problem given in equation (3.1), where the matrix modeling the motion blur is

$$A = \sum_{\ell=1}^{m} \omega_\ell A_\ell. \tag{3.2}$$

Note that the motion detection system used in our work provides the position information needed to construct the matrices $A_\ell$. Since each $A_\ell$ has a different sparsity pattern, the overall sparseness of $A$ decreases (that is, $A$ becomes more dense) as more motion information is used. Thus there is a significant tradeoff between accurately modeling the motion blur and computational cost. To overcome this, we segment the position information into intervals where the position of the object is essentially fixed and compute an average position for each interval. Thus, although the position tracking device may record, say, one thousand distinct head positions, in practice there may be only a few (e.g., ten) significantly different positions. Thus, the integer $m$ in equation (3.2) denotes the number of intervals, and the normalization weights $\omega_\ell$ are determined from how the position information was segmented.

### 3.3.3 Iterative Deblurring

The linear system given in equation (3.1) is an example of an ill-posed inverse problem [64]. Regularization is typically needed to suppress noise amplification in the reconstructed image, and because $A$ is a very large, sparse matrix, it is essential to use iterative methods. Here we consider two methods: LSQR [103] and Modified Residual Norm Steepest Descent (MRNSD) [60], both discussed in Chapter 2. MRNSD is nonnegatively constrained, though it can converge slower than LSQR.

## 3.4 Implementation Details

Our solution to this problem was first implemented in Matlab, next in multithreaded Java by Piotr Wendykier [129], and then in C++ using the Trilinos software framework.

### 3.4.1 Memory Requirements

As the PET problem is large scale in the sense of large amounts of data, let us consider the storage required to generate and solve one such problem on multiple processors. For convenience, $n$ will be the size of the problem (the number of voxels in $\boldsymbol{x}$), $p$ will be the number of processors used, and $m$ will be the number of intervals into which the motion data is divided. First we will consider storage requirements for generating the matrix, then the maps, and finally the vectors, including intermediary ones for solving. We will then look at the total storage and per processor storage required for various $p$ and $m$ values with a given realistic value for $n$.

#### 3.4.1.1 Matrix Memory Requirements

The matrix is stored using a `CrsMatrix` object from the Tpetra class, which uses a compressed row storage scheme as opposed to, say, $i-j-k$ storage. In this storage scheme, the values are stored in one array. Another array contains the column indices, ordered by row, with a third array providing pointers into these arrays for each row.

As a reminder, the $A$ matrix is the sum of $m$ interpolation matrices; that is, $A = \sum_{\ell=1}^{m} A_\ell$. In the simplest form, each of these interpolation matrices is formed using nearest neighbor interpolation, so each $A_\ell$ matrix contains at most one nonzero entry per row. Each processor constructs and stores approximately $m/p$ of these $A_\ell$ matrices and sums them together. However, the $A$ matrix is never explicitly formed. Thus, entries from multiple $A_\ell$ matrices on one processor may overlap and will be summed together (saving storage), but the same location may be replicated across multiple processors. For instance, more than one processor may have a nonzero at location $(0,0)$, but if more than one $A_\ell$ matrix produces a nonzero at that location on one processor, those values will be summed together locally.

This storage scheme, where each processor "owns" all rows, is different than the typical distributed scheme, where processor 0 contains the first $n/p$ rows, processor 1 contains the next $n/p$ rows, and so forth. However, due to the nature of this problem (interpolation matrices), it would be difficult and time consuming to use the

typical scheme for $A$. It would likely require each processor to, in essence, form the interpolation matrix $A_\ell$ and check each row to see if it owns that row and needs to store the nonzero there.

Since each processor forms approximately $m/p$ interpolation matrices, each of which may contain at most one nonzero per row using nearest neighbor interpolation, in the worst case (where no overlapping occurs on a processor), each processor will require storage for $(m/p)*n$ scalars. As there are $p$ processors, this sums to storage for $m * n$ scalars. If trilinear interpolation is used instead, where at most eight nonzeros are stored per row of each interpolation matrix, then each processor may require storage for $(8m/p) * n$ scalars. Over all $p$ processors, this would equate to storage for $8m * n$ scalars. Since Tpetra takes advantage of the templated features of C++, the size of the scalar is not set. A `double`, `float`, or other data type may be used.

As stated before, a `CrsMatrix` also stores the location of each column, plus pointers to each row. Thus, in the worst case, storage for another $m * n$ values, this time of ordinal type, is required, as well as $p * n$ more ordinals. This is because each nonzero on a processor requires storage of its column, and each processor will need a pointer to each of the $n$ rows in the worst case. Thus, a total of $m * n$ scalars (or $8m * n$ scalars for trilinear interpolation) plus $(m + p) * n$ ordinals are needed for a `CrsMatrix`. Additionally, during the creation and storage optimization phases of the matrix, temporary storage is needed; this storage is not included in our analysis.

#### 3.4.1.2 Map Memory Requirements

In Trilinos, as discussed in Section 2.3.1, maps are used to describe the distribution of elements across multiple processors. Each `Tpetra::CrsMatrix` needs four maps to describe it thoroughly, though some of these may be the same `Tpetra::Map` object, saving storage. The four maps are a row map, a column map, a domain map, and a range map. The row map describes which rows a processor owns or has an interest in, similarly for the column map. Suppose we wish to multiply $A\boldsymbol{v} = \boldsymbol{y}$. Then the domain map describes the distribution of $\boldsymbol{v}$, while the range map describes the distribution of $\boldsymbol{y}$.

In our situation, the row and column maps are the same, with each processor owning all the rows and columns. Thus, the storage for these maps is $p * n$ ordinals. The domain and range maps are the same, with processor 0 owning the first $n/p$ rows, and so forth, requiring $(n/p) * p = n$ storage of ordinals. Thus, the total storage costs for maps is $(p + 1) * n$ ordinal values.

#### 3.4.1.3 Vector Memory Requirements

As described in the previous section, the domain and range maps have the typical uniform contiguous distribution, where each row is owned by exactly one processor. Thus, our left- and right-hand side vectors together require storage of $2 * n$ scalars.

Additionally, since the row map and domain map of $A$ differ, import and export vectors are required. Together, these need about $2 * n$ scalars.

The solvers and convergence testers may require additional vectors, each of which needs storage for $n$ scalars. The LSQR solver, when least-error convergence testing is enabled, will require about five such vectors and MRNSD needs about the same number as well. However, the vectors needed by LSQR may be released before the MRNSD solver is created, so the space needed can be re-used rather than required all at once.

#### 3.4.1.4 Total Memory Requirements

Thus for nearest neighbor interpolation, storage for approximately $m*n+2*n+2*n+5*n = (m+9)*n$ scalars and $(m+p)*n+p*n+n = (m+2*p+1)*n$ ordinals is needed; each processor will require storage for $(m+9) * n/p$ scalars and $(m+1) * n/p + 2 * n$ ordinals. Trilinear interpolation will require storage for approximately $7m * n$ more scalars total. Let us set $n$ to be 256*256*95, which is 6,225,920; that is, the image consists of $256 \times 256 \times 95$ voxels. Table 3.1 gives the approximate total and per processor memory used in gigabytes when the data type `double` is used for scalars (assuming 8 bytes each) and `int` for ordinals (assuming 4 bytes each), for various $m$ and $p$ values; Table 3.2 provides similar results for the `float` datatype (assuming 4 bytes each) for scalars and `int` for ordinals. Both of these tables assume nearest neighbor interpolation.

Table 3.1: Approximate storage requirements in gigabytes for various numbers of intervals and processors with `double` and `int` datatypes, using nearest neighbor interpolation. The storage per processor is given in parentheses following the total storage requirements.

| $p \setminus m$ | $m = 1$ | $m = 20$ | $m = 100$ | $m = 560$ |
|---|---|---|---|---|
| $p = 1$ | 0.56 (0.56) | 1.88 (1.88) | 7.45 (7.45) | 39.45 (39.45) |
| $p = 2$ | 0.60 (0.30) | 1.93 (0.96) | 7.49 (3.75) | 39.50 (19.75) |
| $p = 5$ | 0.74 (0.15) | 2.06 (0.41) | 7.63 (1.53) | 39.64 (7.93) |
| $p = 10$ | 0.97 (0.10) | 2.30 (0.23) | 7.86 (0.79) | 39.87 (3.99) |

Table 3.2: Approximate storage requirements in gigabytes for various numbers of intervals and processors with `float` and `int` datatypes, using nearest neighbor interpolation. The storage per processor is given in parentheses following the total storage requirements.

| $p \setminus m$ | $m = 1$ | $m = 20$ | $m = 100$ | $m = 560$ |
|---|---|---|---|---|
| $p = 1$ | 0.32 (0.32) | 1.21 (1.21) | 4.92 (4.92) | 26.25 (26.25) |
| $p = 2$ | 0.37 (0.19) | 1.25 (0.63) | 4.96 (2.48) | 26.30 (13.15) |
| $p = 5$ | 0.51 (0.10) | 1.39 (0.28) | 5.10 (1.02) | 26.44 (5.29) |
| $p = 10$ | 0.74 (0.07) | 1.62 (0.16) | 5.33 (0.53) | 26.67 (2.67) |

### 3.4.2 Scalability Analysis

When using multiple nodes, it is important to ensure the code scales well. Otherwise, it may not be worth the effort of parallel programming if scalability becomes an issue. We checked the scalability of our distributed matrix $A$ in two different ways, for two different sizes. First, we constructed a matrix using 560 intervals with nearest neighbor interpolation, then performed from one thousand to ten thousand matrix-vector multiplications (in increments of one thousand), on one, two, four, eight, and sixteen processors. The results from this test are given in Tables 3.3 and 3.4, for problem sizes $32 \times 32 \times 12$ and $64 \times 64 \times 24$ respectively. This test shows that, for a given matrix, the time required to perform matrix-vector multiplications scales nearly linearly when the number of processors is kept constant. Also, when the number of matrix-vector multiplications is kept constant, the time required initially decreases as the number of processors used increases, until communication costs overtake the computational costs. Figure 3.2 gives a visual representation of these results.

Once we verified that, for a given matrix, the matrix-vector multiplications scale, we considered how the timings scale when the number of intervals used changes. We constructed matrices with nearest neighbor interpolation containing 16, 32, 64, 128, 256, and 512 intervals and performed 2500 matrix-vector multiplications on each, recording the time used by one, two, four, eight, and sixteen processors. These results are given in Tables 3.5 and 3.6, for problems with 12,288 voxels and 98,304 voxels respectively. We see that, as expected, the amount of time to perform matrix-vector multiplications increases as the number of nonzeros in the matrix increases, but that doubling the number of intervals does not double the time. Again, we also see speedup with increasing number of processors but just to the point where communication costs

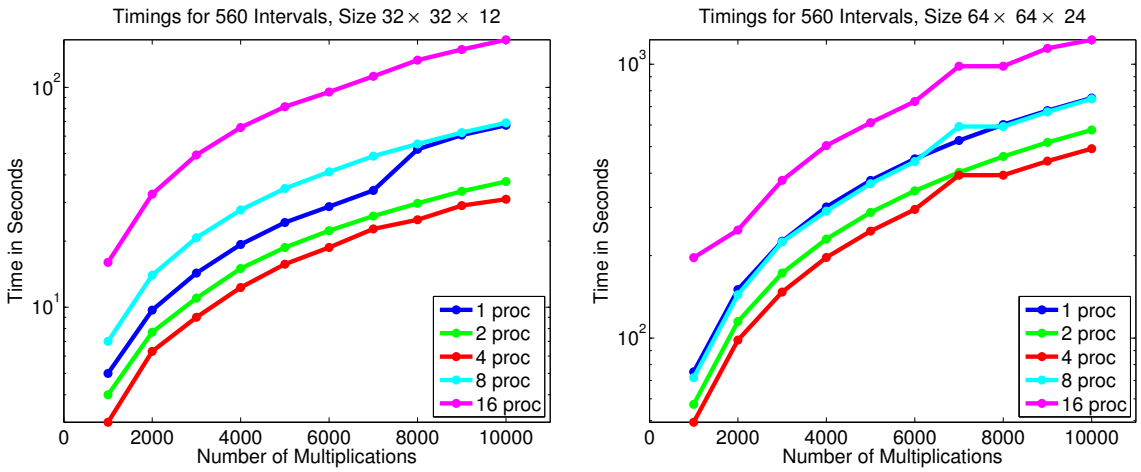
prevail. For a visualization of the results from this test, see Figure 3.3.

Table 3.3: Time (in seconds) for varying numbers of matrix-vector multiplications to be performed for problem size $32 \times 32 \times 12$ when number of intervals is fixed.

| # Mults | 1 proc | 2 proc | 4 proc | 8 proc | 16 proc |
|---|---|---|---|---|---|
| 1000 | 5.0 | 4.0 | 3.0 | 7.0 | 16.7 |
| 2000 | 9.7 | 7.7 | 6.3 | 14.0 | 32.7 |
| 3000 | 14.3 | 11.0 | 9.0 | 20.7 | 49.3 |
| 4000 | 19.3 | 15.0 | 12.3 | 27.7 | 65.7 |
| 5000 | 24.3 | 18.7 | 15.7 | 34.7 | 81.7 |
| 6000 | 28.7 | 22.3 | 18.7 | 41.3 | 95.3 |
| 7000 | 34.0 | 26.0 | 22.7 | 48.7 | 112.3 |
| 8000 | 52.3 | 29.7 | 25.0 | 55.3 | 133.0 |
| 9000 | 60.7 | 33.7 | 29.0 | 62.3 | 148.7 |
| 10000 | 67.3 | 37.3 | 31.0 | 69.0 | 164.7 |

Table 3.4: Time (in seconds) for varying numbers of matrix-vector multiplications to be performed for problem size $64 \times 64 \times 24$ when number of intervals is fixed.

| # Mults | 1 proc | 2 proc | 4 proc | 8 proc | 16 proc |
|---|---|---|---|---|---|
| 1000 | 75.3 | 57.3 | 49.3 | 71.7 | 196.7 |
| 2000 | 150.3 | 115.0 | 98.3 | 144.0 | 248.0 |
| 3000 | 225.7 | 172.7 | 147.3 | 224.3 | 376.7 |
| 4000 | 301.0 | 230.0 | 197.0 | 290.7 | 504.3 |
| 5000 | 376.3 | 287.7 | 245.7 | 366.0 | 611.3 |
| 6000 | 451.3 | 345.0 | 295.0 | 441.0 | 731.3 |
| 7000 | 526.3 | 402.7 | 344.3 | 508.3 | 855.3 |
| 8000 | 601.7 | 460.3 | 393.3 | 592.3 | 982.7 |
| 9000 | 676.7 | 518.0 | 442.7 | 669.7 | 1142.3 |
| 10000 | 752.3 | 575.3 | 491.7 | 746.7 | 1228.0 |




(a) Size $32 \times 32 \times 12$ (b) Size $64 \times 64 \times 24$

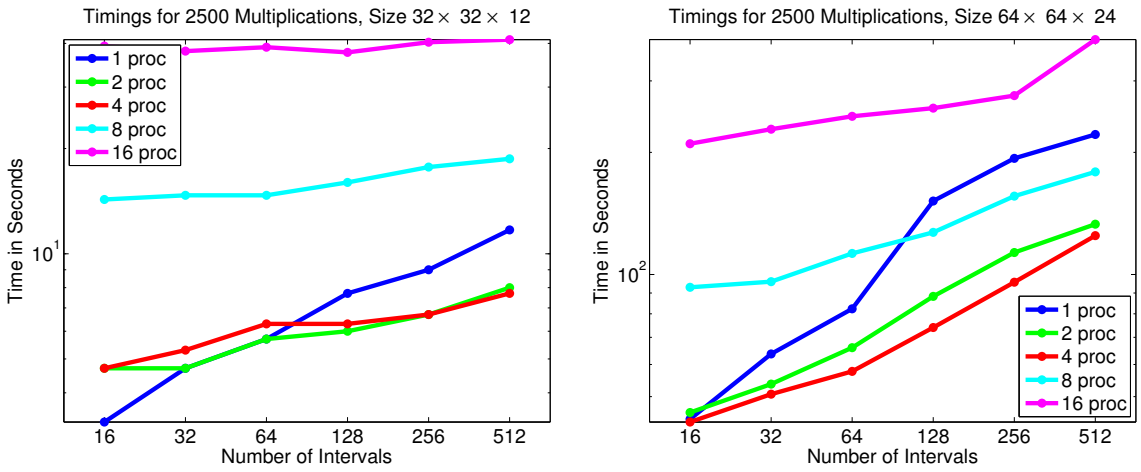
Figure 3.2: Timings for varying numbers of matrix-vector multiplications on varying numbers of processors when the matrix is composed from 560 intervals.

Table 3.5: Time (in seconds) for 2500 matrix-vector multiplications to be performed for problem size $32 \times 32 \times 12$ when number of intervals varies.

| # Intervals | 1 proc | 2 proc | 4 proc | 8 proc | 16 proc |
|---|---|---|---|---|---|
| 16 | 3.3 | 4.7 | 4.7 | 14.3 | 39.3 |
| 32 | 4.7 | 4.7 | 5.3 | 14.7 | 38.0 |
| 64 | 5.7 | 5.7 | 6.3 | 14.7 | 39.0 |
| 128 | 7.7 | 6.0 | 6.3 | 16.0 | 37.7 |
| 256 | 9.0 | 6.7 | 6.7 | 17.7 | 40.3 |
| 512 | 11.7 | 8.0 | 7.7 | 18.7 | 41.0 |

Table 3.6: Time (in seconds) for 2500 matrix-vector multiplications to be performed for problem size $64 \times 64 \times 24$ when number of intervals varies.

| # Intervals | 1 proc | 2 proc | 4 proc | 8 proc | 16 proc |
|---|---|---|---|---|---|
| 16 | 44.0 | 45.7 | 43.3 | 93.0 | 210.0 |
| 32 | 63.7 | 53.7 | 50.7 | 96.0 | 228.0 |
| 64 | 82.3 | 66.0 | 57.7 | 112.7 | 245.3 |
| 128 | 151.7 | 88.3 | 74.0 | 127.0 | 257.0 |
| 256 | 193.3 | 113.3 | 95.7 | 156.0 | 276.0 |
| 512 | 221.3 | 133.0 | 124.7 | 179.0 | 379.3 |




(a) Size $32 \times 32 \times 12$ (b) Size $64 \times 64 \times 24$

Figure 3.3: Timings for varying numbers of intervals on varying numbers of processors when 2500 matrix-vector multiplications are performed.

### 3.4.3 Testbed

These timings and the following results were performed on the "puma" cluster at Emory University, which is a high performance cluster with thirty-two nodes and 128 processor cores. Each node has two dual core AMD 2214 2.2 GHz Opteron CPUs, four gigabytes of RAM and an eighty gigabyte drive. The nodes are connected via a High Performance InfiniBand network and also Gigabit Ethernet. These timings were done while other users also had access and jobs running. However, the overall trend of the results would likely be the same had these tests been run in isolation.

## 3.5 Results

We will consider three different simulations. First, we will consider the effect of precision on the results; namely, how the results compare when single or double precision is used. Next, we will compare results when equally-sized intervals are used compared to unequally-sized intervals. Finally, we will consider the effect of motion on the results by considering three levels of patient motion: low, mid, and high.

Numerical simulations in this section were performed on a software-generated Hoffman phantom object. The original size of the object was $256 \times 256 \times 95$ voxels, but due to memory limitations when using a large number of intervals we had to resize the image in Matlab. We used bilinear interpolation on each slice to ensure nonnegativity, then selected every other slice to create a "true" image of size $128 \times 128 \times 48$

voxels. The motion-blurred image was generated using real patient motion information, trilinear interpolation, and 10% normally distributed noise; see Figure 3.4 for the middle slice of an example image.

Figure 3.4: Motion-blurred phantom image.

### 3.5.1 Effect of Scalar Precision

We first consider the effects of scalar precision on the solution when motion from a fairly cooperative patient is used. We generated blurred data for an image of size of $128 \times 128 \times 48$ and added ten percent noise (see Figure 3.4). We then solved this using both single and double precision with LSQR and MRNSD solvers for an increasing number of intervals, from two to one hundred. Both nearest neighbor (NN) and trilinear (Tri) interpolation are considered. We ran out of memory when using double-precision scalars with trilinear interpolation around eighty intervals, so we stopped at the same point when using nearest neighbor interpolation.

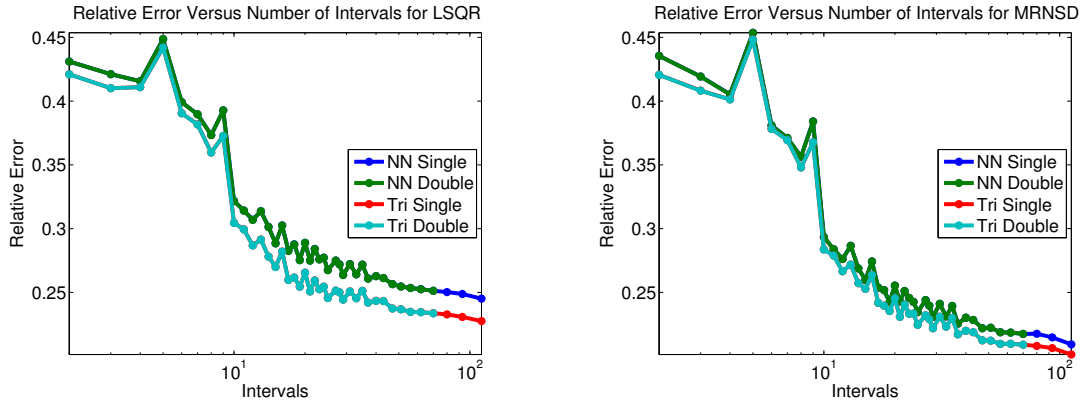
The results in Figure 3.5 seem to show that the choice of precision has very little impact on the relative error (the results are nearly indistinguishable). The iteration count depends much more on the choice of solver and interpolation type than data precision, as shown in Figure 3.6. Finally, in Figure 3.7, a visual comparison of the solutions for both single and double precision is given. This figure shows the middle slice of the reconstructed solutions for both solvers, when single or double precision scalars are used. Trilinear interpolation is used, though the results are similar for nearest neighbor. Notice the presence of ringing in the reconstructions using LSQR;

this is because there are no nonnegativity constraints. However, the reconstruction when single precision is used is virtually identical to the reconstruction when double precision is used, for the same solver.

The metric used in these results in relative error, defined as

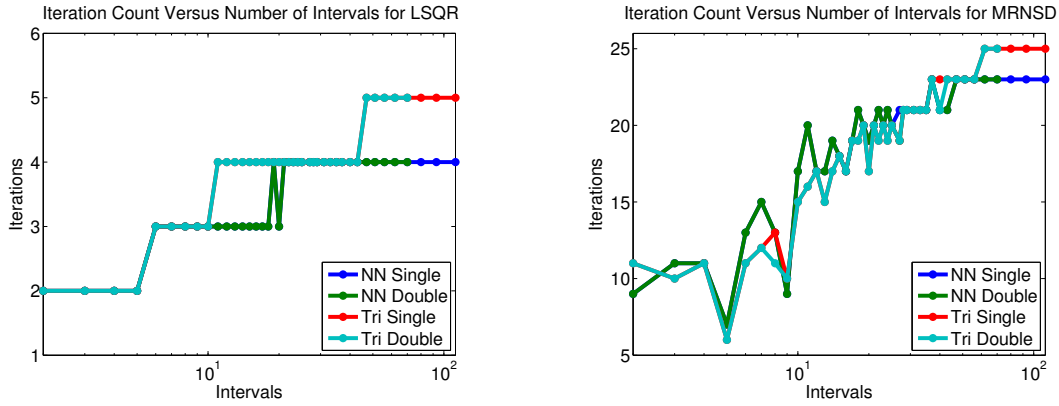
$$\frac{\|\boldsymbol{x}_{\text{recon}} - \boldsymbol{x}_{\text{exact}}\|_2}{\|\boldsymbol{x}_{\text{exact}}\|_2},$$

where $\boldsymbol{x}_{\text{recon}}$ is the reconstructed solution and $\boldsymbol{x}_{\text{exact}}$ is the exact solution, both in vector form. We use the least error status test, described in Section 2.6, to stop at the iteration with least relative error for a given number of intervals.




(a) LSQR (b) MRNSD

Figure 3.5: Comparison of relative error to number of intervals used when scalar precision varies.




(a) LSQR (b) MRNSD

Figure 3.6: Comparison of iterations to number of intervals used when scalar precision varies.

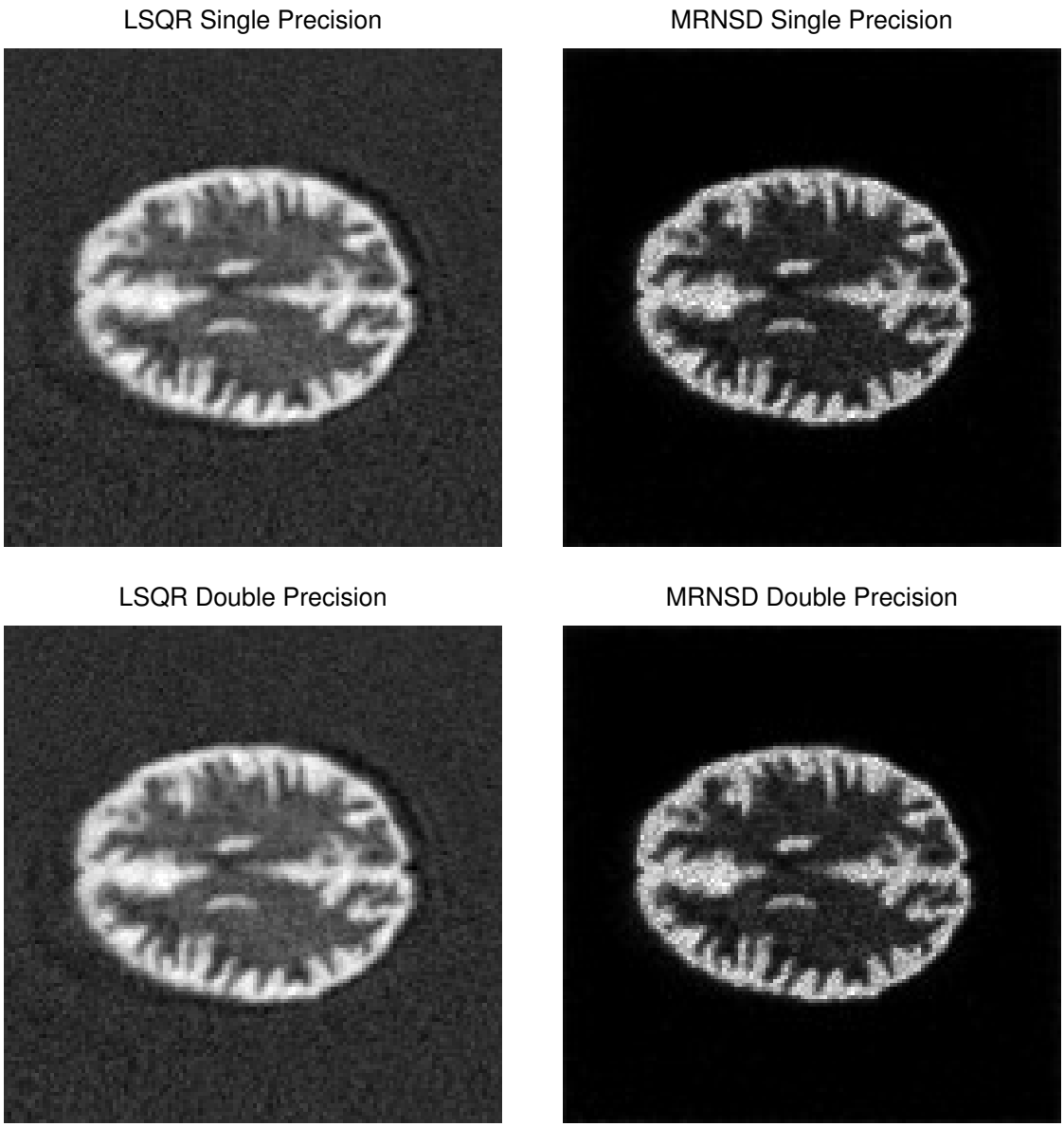
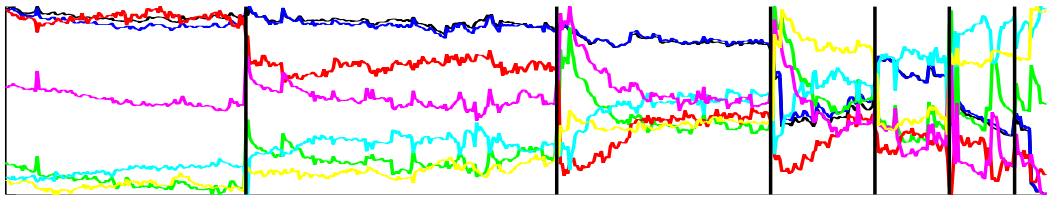


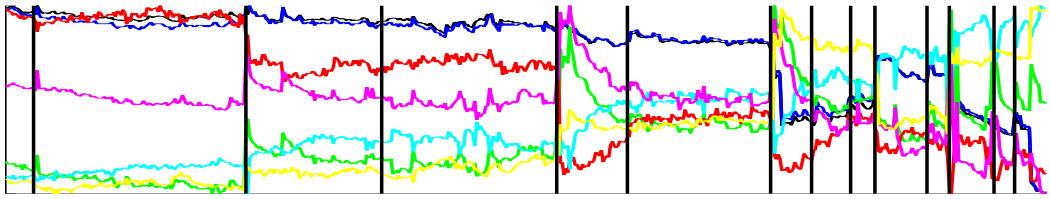

Figure 3.7: Comparison of reconstructions when scalar precision and solver type varies.

### 3.5.2 Effect of Interval Choice

Using the same setup as in the previous simulation, we now consider the effect of interval choice. Namely, we wish to compare intervals of unequal size to ones of equal size. We performed a manual segmentation of the motion into both seven and fourteen intervals of differing lengths. This was done by plotting the motion and visually determining where the best locations for intervals may be. As such, this is somewhat subjective, but no robust software for automating this task is currently available. However, a Java program that attempts to determine the best locations with some user guidance is described in [130]. See Figure 3.8 to compare the two manual segmentations, plotted atop a scaled version of the motion information.



(a) 7 Unequal Intervals

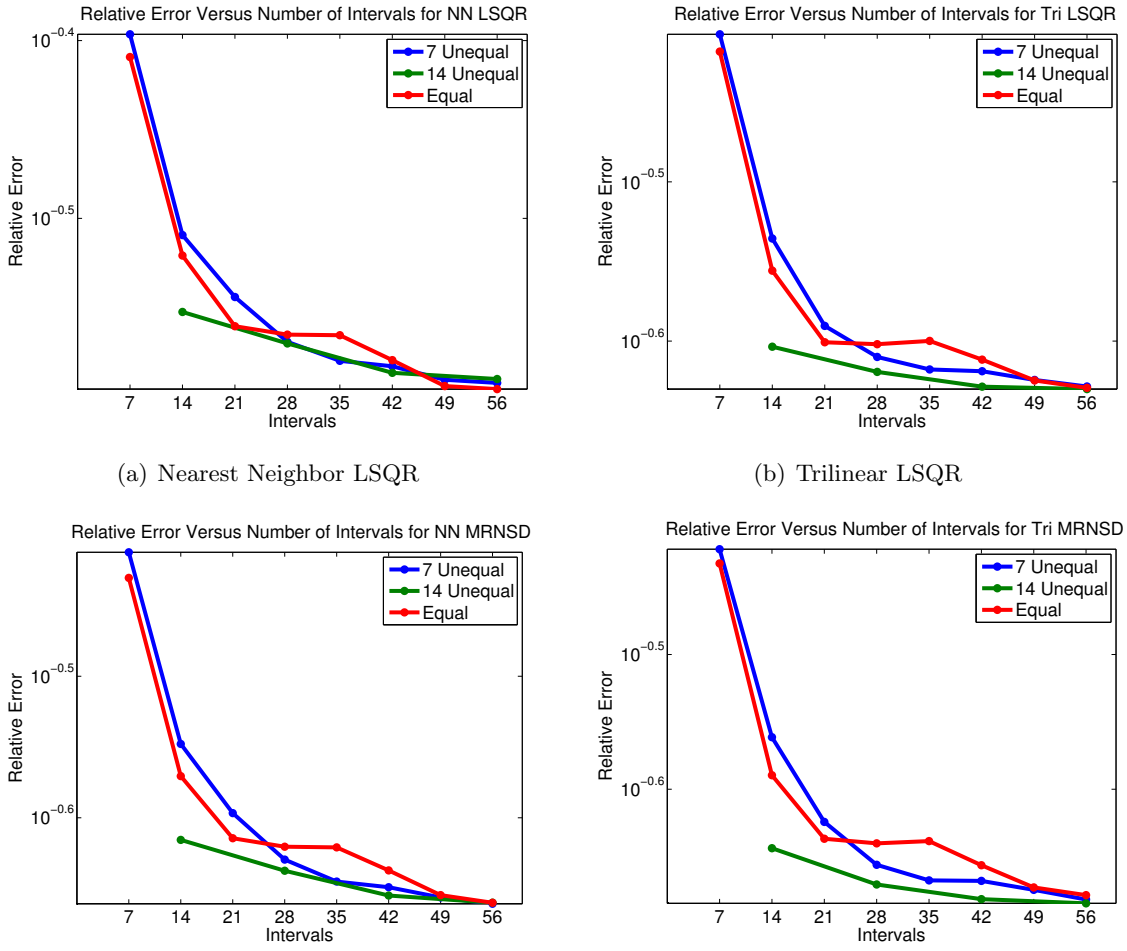


(b) 14 Unequal Intervals

Figure 3.8: Comparison of two segmentations of patient motion.

After dividing the motion into either seven or fourteen intervals, we solved the problem, stopping at the iteration with least relative error. We then subdivided the intervals equally (or as equally as possible), and solved again. We continued in this manner until we had at most 56 different intervals, formed by subdividing the original seven intervals eight times or the fourteen intervals four times. It is to be noted that only the original seven or fourteen intervals were manually chosen; after that, the subdivisions were made equally within the larger intervals.

Figure 3.9 compares the relative error to the number of intervals for various interpolation scheme and solver combinations. For these and the following results, single precision scalars were used. As can be seen, when solving with seven ideally-chosen intervals, the results were not quite as good as when seven equally-spaced intervals were used. However, as the intervals were further subdivided, the results improved.




(a) Nearest Neighbor LSQR

(b) Trilinear LSQR

(c) Nearest Neighber MRNSD

(d) Trilinear MRNSD

Figure 3.9: Comparison of relative error to number of intervals used when segmentation process varies.

After a certain point, though, when the intervals were all sufficiently small, it seemed to make little difference if ideally-chosen or equally-chosen intervals were originally used.

On the other hand, when fourteen ideally-chosen intervals were used, the results were significantly better than those obtained with fourteen equally-sized intervals. But again, after the intervals had been subdivided sufficiently, there was little difference in the results.

This seems to suggest that if the intervals are made sufficiently small, then the simple method of using intervals of equal size will work nearly as well as a more complicated method of choosing interval size. Additionally, when few intervals are used, then the choice becomes much more important. For well-chosen intervals of a correct number, as was the case of fourteen initial intervals, the results can be dramatically better than when using a few intervals of equal size. However, when the wrong number of intervals is used, as was the case with seven intervals, then using equally-sized intervals may prove to be better. This is likely because the intervals at the beginning of the scan were much longer than those at the end of the scan when seven manually-chosen intervals were used – the patient experienced much more motion near the end of the scan. However, this unequal weighting can have a negative effect on the results. When a larger number of intervals were manually chosen, the results were significantly better.

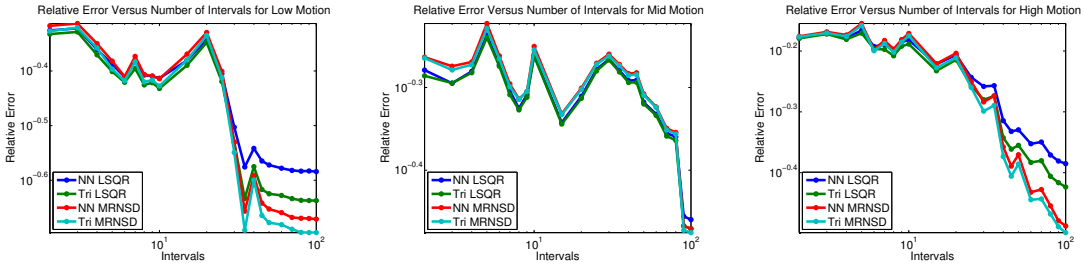
As it would be incredibly costly and not a realistic solution to try every single combination of number of intervals and relative size, it appears that using intervals of equal size, and as many as computationally feasible, is a viable option.

### 3.5.3 Effect of Patient Motion

We next consider the effect of patient motion on the reconstructed solutions. We consider three levels of motion: low motion from a cooperative patient, mid motion from a semi-cooperative patient, and high motion from an uncooperative patient. From each motion file, we generated a blurred image as described above, adding 10% noise.

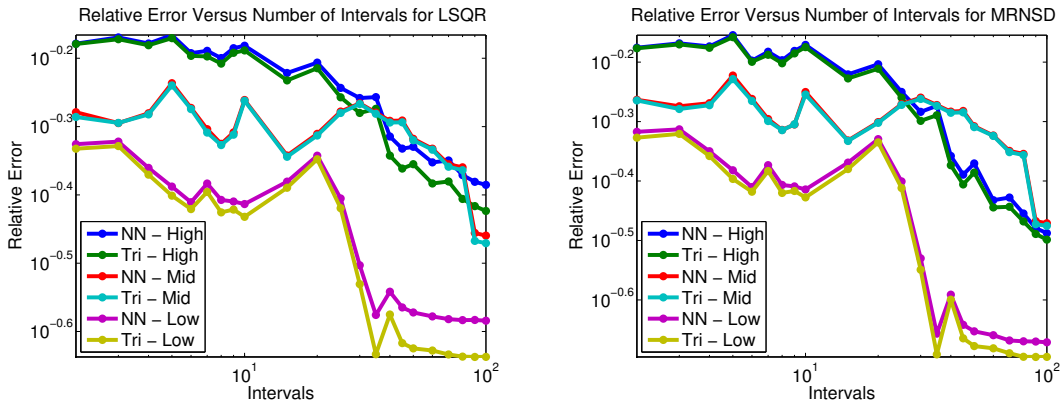
The following results are computed using single precision, as we have shown previously that the choice of single or double precision has little effect on the relative error. Each reconstruction was formed using from two to one hundred equal-sized intervals, using nearest neighbor (NN) or trilinear (Tri) interpolation, and using LSQR or MRNSD as the solver. Since the true solution was known, we stopped at the

iteration giving the least relative error. Figure 3.10 displays the results comparing the relative error to the number of intervals for each level of motion. Figure 3.11 compares the results for all levels of motion using either LSQR or MRNSD.




(a) Low Motion (b) Mid Motion (c) High Motion

Figure 3.10: Comparison of relative error to number of intervals used when level of patient motion varies.



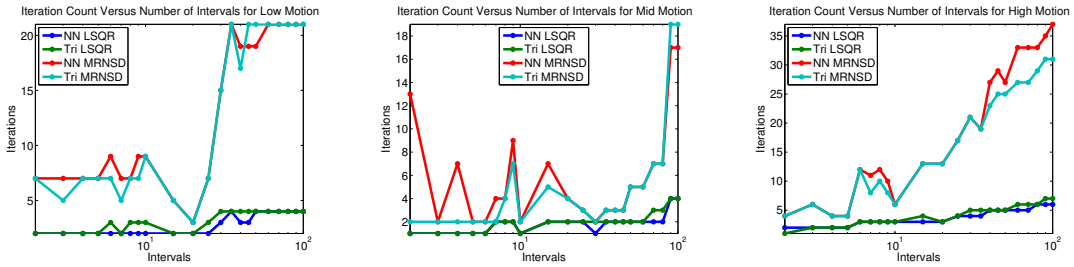

(a) LSQR (b) MRNSD

Figure 3.11: Comparison of relative error to number of intervals used when level of patient motion varies, per solver type.

This next set of figures (Figures 3.12 and 3.13) compares the number of iterations required to the number of intervals, both for each level of motion and for each solver type.

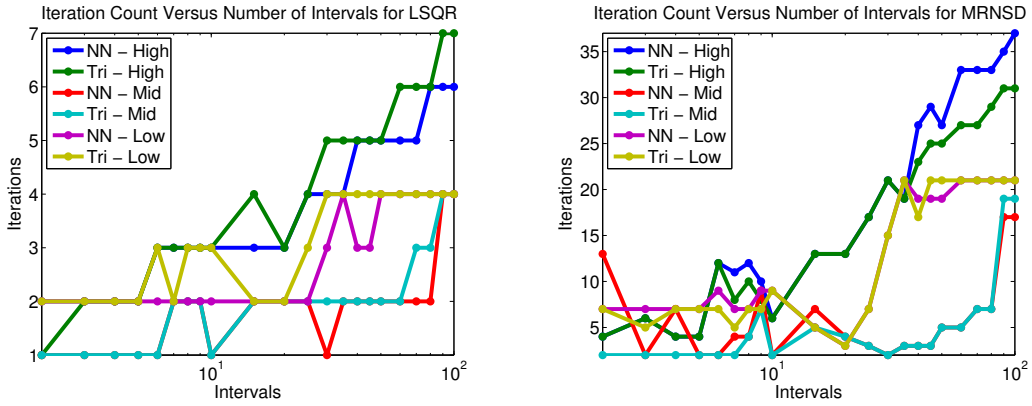
The last set of figures we present displays the reduction in error compared to the number of intervals for each motion level. The error reduction was computed as follows. First, the initial relative error for each motion level was computed by using the blurred, noisy data compared to the true solution. Table 3.7 displays the initial relative error for each motion level. Notice how the images with less motion have a smaller initial relative error than those with more motion.

Next we scale the relative error given in the previous figures by the initial relative




(a) Low Motion (b) Mid Motion (c) High Motion

Figure 3.12: Comparison of iterations to number of intervals used when level of patient motion varies.




(a) LSQR (b) MRNSD

Figure 3.13: Comparison of iterations to number of intervals used when level of patient motion varies, per solver type.

Table 3.7: Initial relative error for each motion level.

| Motion Level | Initial Relative Error |
|---|---|
| Low | 0.4016 |
| Mid | 0.5705 |
| High | 0.6528 |

error simply by dividing. Finally, we subtract the scaled relative error from 1. Thus, a reduction in relative error of 0.1 should signify an approximate 10% improvement in the reconstructed image compared to the blurred, noisy image. This allows us to compensate for the differing initial relative errors present in Figure 3.11.

This metric may not be the most appropriate choice as it is possible for the scaled relative error to be larger than 1; that is, that a reconstructed solution using some number of intervals produces a relative error that is worse than the relative error of the blurred, noisy image. However, it appears that this metric shows the long-term trend in error reduction as the number of intervals increases, taking the motion level into account.

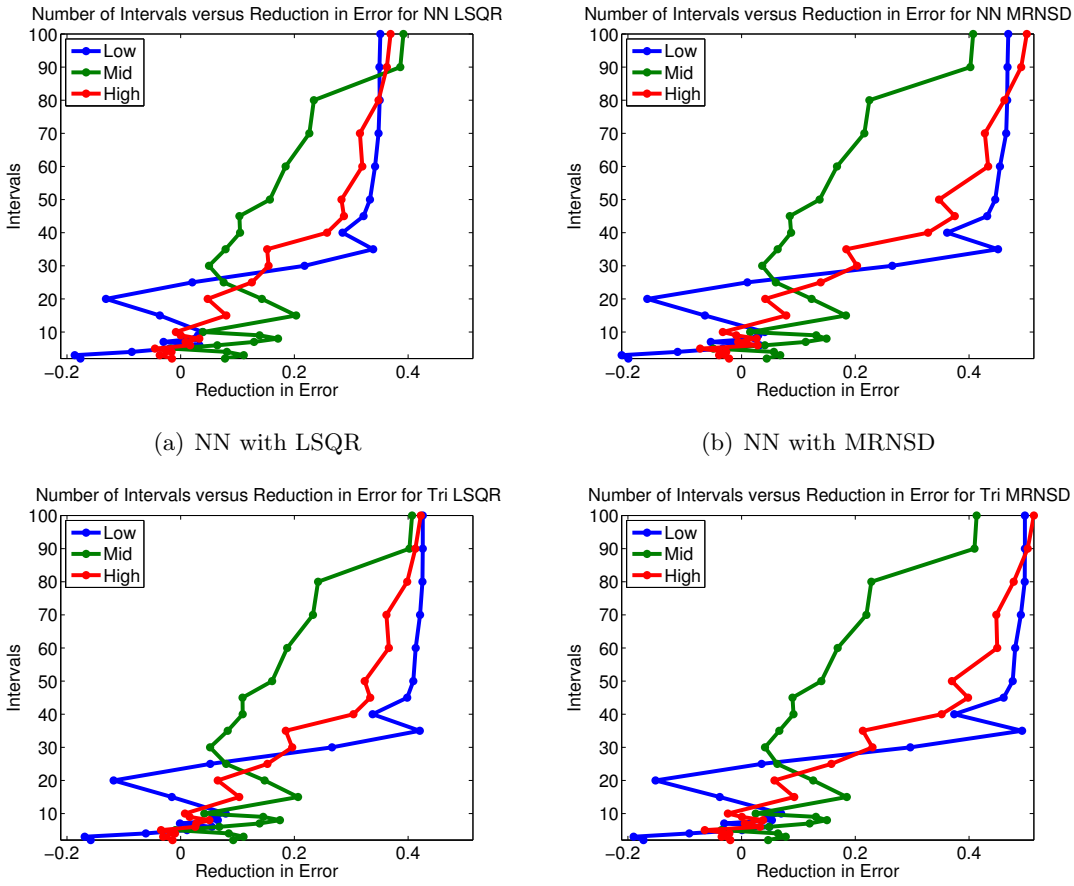
Figure 3.14 compares the number of intervals to the reduction in error for four combinations of interpolation type and solver: nearest neighbor with LSQR, nearest neighbor with MRNSD, trilinear with LSQR, and trilinear with MRNSD.

As can be seen, the initial relative error is less for lower amounts of patient motion. However, the relative error decreases in a similar fashion for each type of patient motion as the number of intervals increases. The number of iterations needed appears not to depend on the type of patient motion as much, though the number of iterations appears to increase somewhat as the number of intervals used increases. As expected, the reduction in error increases as the number of intervals increases, though the reduction seems to level off around 40%. Though the reduction in error differs slightly depending on the combination of interpolation type and iterative method used, the overall trend remains the same.

## 3.6 Remarks and Future Directions

We would like to draw some conclusions from the results presented.

- First, from Section 3.5.1, it appears that the results using single precision or double precision are quite similar. Thus, to save on memory, we can use single precision in solving without significant loss of accuracy.
- The difference between results computed using nearest neighbor or trilinear interpolation appears to be greater when solving with LSQR than with MRNSD.
- Using intervals of equal size may be advisable when just a few or when many intervals are used. When a moderate number is chosen, better results may be obtained by selecting each interval size manually.




(a) NN with LSQR

(b) NN with MRNSD

(c) Tri with LSQR

(d) Tri with MRNSD

Figure 3.14: Comparison of number of intervals used to reduction in error when level of patient motion varies.

- The iteration count is typically less than ten when solving with LSQR, regardless of the level of patient motion or interpolation scheme. However, when MRNSD is used to solve, often five to thirty iterations are required to obtain the solution with least error.
- The relative error is typically lower for less patient motion and greater for more patient motion.
- The initial relative error is less for lower levels of patient motion.
- The reduction in error appears to top out, suggesting that using extra intervals provides little benefit in terms of error reduction. Just where this peak is, though, depends heavily on the motion. In our examples, it seemed to occur around fifty to ninety intervals.
- It may be possible to obtain a negative reduction in error when small numbers of intervals are used. Here the relative error actually gets larger with the reconstructed solution than with the initial blurred image. However, once a certain number of intervals is used for the noise level, the error quickly reduces.

We have a number of ideas for future plans. We would like to further investigate using intervals of unequal length. In particular, we would like to determine a method to automatically choose the ideal number of intervals to use. To do so, we would like to implement a method to generate random patient motion in Matlab. Ideally it could accommodate two types of motion: the drifter (where the head slowly lolls from one location to another) and the jerker (where the head abruptly moves from one location to another), as well as a combination of these two. We would like to use this random patient motion to generate more simulated blurred images. We could then solve them, having more control over the random motion than is available in true patient motion data.

# Chapter 4

# Case Study 2: Adaptive Optics Application

The atmosphere affects the light captured by ground-based telescopes, even when the telescopes are set atop high mountains. The atmosphere causes the light from stars and other celestial objects to arrive in a nonplanar manner. However, the gradient of the atmospheric disturbance may be detected and the disturbance (known as the phase error) compensated for using adaptive optics systems. Thus, a main problem in adaptive optics is to reconstruct the phase error given noisy phase differences; see Section 4.1. Previous approaches to solve this least-squares minimization problem are considered in Section 4.2, with our efficient approach presented in Section 4.3. We consider using either truncated singular value decomposition (TSVD)-type or Tikhonov-type regularization. Both of these approaches make use of Kronecker products and the generalized singular value decomposition. The TSVD-type regularization operates as a direct method whereas the Tikhonov-type regularization uses a preconditioned conjugate gradient-type iterative algorithm to achieve fast convergence. A few implementation considerations are given in Section 4.4. Some results are presented in Section 4.5, with conclusions drawn in Section 4.6.

## 4.1 Motivation

When viewing objects in space using a ground-based telescope, the images are blurred by the atmosphere. To remove the blur, adaptive optics systems solve two inverse problems. First, quantitative information about the blur must be determined; second, this quantitative information is used to remove the blur by a combination of telescope mirror adjustments and computational postprocessing of the images. All of these

computations and adjustments must be performed in real time. This chapter, from [7], focuses on the first problem, namely that of determining quantitative information about the blur.

Using the classic model of image formation relating an object $x$ and a blurring kernel $k$ to an image $b$, we can write:

$$b(s,t) = \int_\Omega k(s,t;\xi,\psi)\ x(\xi,\psi)\ d\xi\ d\psi$$

The kernel is assumed to be spatially invariant; that is, we assume $k(s,t;\xi,\psi) = k(s-\xi,t-\psi)$. The Fourier optics model for $k$ [53, 112] allows us to express it as a function of the phase error $\phi$ of incoming wavefronts of light:

$$k(s,t) = \left|\mathcal{F}^{-1}\{\mathcal{P}(s,t)e^{i\phi(s,t)}\}\right|^2. \tag{4.1}$$

Here $\mathcal{F}^{-1}$ is the inverse Fourier transform, $\mathcal{P}(s,t)$ is the mirror or pupil aperture (a characteristic) function, $i = \sqrt{-1}$, and $\phi$ is the phase error, which is the deviation from planarity of the wavefront that reaches the telescope mirror.

The goal of the adaptive optics system on a telescope is to remove the effects of the phase error $\phi$ on the kernel $k$. This is accomplished by first computing an estimate $\tilde{\phi}$ of $\phi$ and then by constructing a counter wavefront $-\tilde{\phi}$ via a deformable mirror. The kernel $k$ is then modified as follows:

$$\tilde{k}(s,t) = \left|\mathcal{F}^{-1}\{\mathcal{P}(s,t)e^{i(\phi(s,t)-\tilde{\phi}(s,t))}\}\right|^2.$$

If $\phi = \tilde{\phi}$, $\tilde{k}$ is said to have diffraction limited form, where the resolution is dependent only on the telescope.

To obtain an estimate of $\phi$, a wavefront sensor in the telescope collects discrete, noisy measurements of the gradient of incoming wavefronts of light emitted by an object. Since the gradient of the incoming wavefronts equals the gradient of the phase error $\phi$ of those wavefronts, the data noise model takes the form

$$\begin{bmatrix} \boldsymbol{b}_h \\ \boldsymbol{b}_v \end{bmatrix} = \begin{bmatrix} A_h \\ A_v \end{bmatrix} \boldsymbol{\phi} + \boldsymbol{\eta}, \tag{4.2}$$

where $\boldsymbol{b}_h$ and $\boldsymbol{b}_v$ are discrete, noisy measurements of the horizontal and vertical derivatives of $\boldsymbol{\phi}$; $A_h$ and $A_v$ are discrete, horizontal and vertical derivative operators; and $\boldsymbol{\eta}$ represents independent and identically distributed (iid) Gaussian noise. This linear model can then be solved to compute an estimate of $\boldsymbol{\phi}$.

Model (4.2) requires some assumptions that are sufficiently accurate, but not exact. For example, the mapping from the phase $\boldsymbol{\phi}$ to the gradient measurements

contains nonlinearities [12]. Here we model it as linear. Additionally, the assumption of iid Gaussian noise, while common in the literature (see e.g., [12, 24, 34, 46]), in particular for simulations, is not precise as it ignores photon noise.

The operators $A_h$ and $A_v$ are determined by the wavefront sensor geometry that is used. Common wavefront sensor geometries discussed in the literature are those of Hudgin [75] and Fried [44]. In this chapter, we focus on Fried geometry, both because it is more commonly used in operational adaptive optics systems, and because it is more interesting mathematically. For the majority of our discussion, we also assume a rectangular pupil for the telescope. This simplifies the mathematics, however the approach set forth here can be extended to circular or annular pupils in a preconditioned iterative framework, as in [110, 111]. We will briefly consider how our approach works when circular or annular pupils are used.

## 4.2 Previous Work

The problem of reconstructing the shape of an object from measurements of its gradient appears in a variety of applications (see e.g., [40] and the references therein), while the more specific problem of wavefront sensing appears not only in adaptive optics but also in phase imaging and visual optics [6].

Previous approaches to determining the phase error $\boldsymbol{\phi}$ include the preconditioned conjugate gradient (PCG) method and the alternating direction implicit (ADI) method. In [47], PCG is used with the preconditioner based on a multigrid algorithm. Multigrid-preconditioned CG is compared to several other methods, including least squares preconditioned conjugate gradient and gradient denoised least squares, in [10]. Ren and Dekany solve for $\boldsymbol{\phi}$ using the ADI method by utilizing the Sylvester equation in [110].

## 4.3 Methodology of Our Approach

We propose to solve the phase reconstruction problem (4.2) using a preconditioned iterative approach. The preconditioner is formed by using the generalized singular value decomposition (GSVD) and Kronecker products to exploit the structure of the coefficient matrix in (4.2). The preconditioner also suggests a very efficient direct solver using TSVD-type regularization. Though this approach gives good results and is quite efficient, the preconditioned iterative scheme, which uses Tikhonov-type regularization, is preferred by the application people. Real-time solutions may be obtained using either method.

### 4.3.1 Mathematical Background

We make significant use of various properties of the generalized singular value decomposition (GSVD) [52] and Kronecker products [88] in our algorithm. Thus, in this section we briefly describe a few of the relevant properties, as well as the matrices that are formed from both the Hudgin and Fried geometries.

#### 4.3.1.1 Generalized Singular Value Decomposition

The generalized singular value decomposition (GSVD) of two matrices $B$ and $C$ is given as follows:

$$B = U\Sigma X^T, \quad C = V\Delta X^T \tag{4.3}$$

where

- $U$ and $V$ are orthogonal matrices
- $X$ is nonsingular
- $\Sigma$ and $\Delta$ are nonnegative diagonal matrices, though the entries may be on a superdiagonal.

#### 4.3.1.2 Kronecker Products

If $B \in \mathcal{R}^{m\times n}$ and $C$ is another matrix, then the Kronecker product of $B$ and $C$ produces the block structured matrix

$$B \otimes C = \begin{bmatrix} b_{11}C & \cdots & b_{1n}C \\ \vdots & & \vdots \\ b_{m1}C & \cdots & b_{mn}C \end{bmatrix},$$

where $b_{ij}$ is the $ij$th element of $B$. Assume that $B = U_B\Sigma_B V_B^T$ is some decomposition for $B$ (and similarly for $C$). Kronecker products have a plethora of convenient properties, many of which are detailed in Van Loan's survey [88]. We present here but a few of these:

$$(B \otimes C)^T = B^T \otimes C^T, \tag{4.4}$$

$$(B \otimes C)^{-1} = B^{-1} \otimes C^{-1}, \tag{4.5}$$

$$B \otimes C = (U_B\Sigma_B V_B^T) \otimes (U_C\Sigma_C V_C^T) = (U_B \otimes U_C)(\Sigma_B \otimes \Sigma_C)(V_B \otimes V_C)^T. \tag{4.6}$$

One additional property we need relates matrix-matrix multiplication with Kronecker products. As is common, we define the vec($Z$) operation as stacking the columns of

a matrix $Z$; that is, if $Z \in \mathcal{R}^{m \times n}$, then $\text{vec}(Z)$ is an $nm \times 1$ vector. Given matrices $C, Z, B$ such that the product $CZB^T$ is defined, then

$$Y = CZB^T \quad \Leftrightarrow \quad \boldsymbol{y} = (B \otimes C)\boldsymbol{z}, \tag{4.7}$$

where $\boldsymbol{z} = \text{vec}(Z)$ and $\boldsymbol{y} = \text{vec}(Y)$.

#### 4.3.1.3 Matrix Formulation of Geometries

B. R. Hunt came up with a way to develop the wavefront reconstruction problem in a matrix-algebra framework [77]. We assume the two-dimensional phase spectrum exists over a grid of points indexed by $i, j = 1, 2, \ldots, n$, so that $\Phi_{ij}$ is the phase spectrum at the $ij$th coordinate. The wavefront sensor produces phase differences on each of the two coordinates; we denote by $\boldsymbol{b}_h$ the phase differences on the $i$ coordinates and by $\boldsymbol{b}_v$ the phase differences on the $j$ coordinates.

Using the vec operator so that $\text{vec}(\Phi) = \boldsymbol{\phi}$, we then write the problem as

$$\boldsymbol{b} = A\boldsymbol{\phi} + \boldsymbol{\eta}, \tag{4.8}$$

where

$$\boldsymbol{b} = \begin{bmatrix} \boldsymbol{b}_h \\ \boldsymbol{b}_v \end{bmatrix} \quad \text{and} \quad A = \begin{bmatrix} A_h \\ A_v \end{bmatrix}.$$

The vector $\boldsymbol{\eta}$ represents noise and other errors in the measured data, and is typically assumed to be iid normal (Gaussian) random values.

In the case of Hudgin geometry [75],

$$A_h = I \otimes H \quad \text{and} \quad A_v = H \otimes I,$$

where

$$H = \begin{bmatrix} 1 & -1 & & & \\ & 1 & -1 & & \\ & & \ddots & \ddots & \\ & & & 1 & -1 \end{bmatrix} \tag{4.9}$$

is $(n-1) \times n$, and $I$ is the $n \times n$ identity. $H$ is a 1-dimensional phase difference matrix.

In the case of Fried geometry [44],

$$A_h = F \otimes H \quad \text{and} \quad A_v = H \otimes F,$$

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \qquad \begin{bmatrix} -1 & 0 & -1 \\ 0 & 4 & 0 \\ -1 & 0 & -1 \end{bmatrix}$$

Figure 4.1: Grid representations of the Hudgin Laplacian (left) and the Fried Laplacian (right).

where

$$F = \frac{1}{2}\begin{bmatrix} 1 & 1 & & & \\ & 1 & 1 & & \\ & & \ddots & \ddots & \\ & & & 1 & 1 \end{bmatrix}$$

is $(n-1) \times n$, and $H$ is as above. $F$ is an averaging matrix. For the remainder of this chapter, we consider only the Fried geometry because it is used most often in adaptive optics applications.

When least squares estimation is used to obtain a solution of (4.8), the coefficient matrix for the normal equations has the form $A_h^T A_h + A_v^T A_v$. When Hudgin geometry is used, $A_h^T A_h + A_v^T A_v$ is the standard discrete Laplacian, with grid representation given on the left in Figure 4.1, whereas when Fried geometry is used, $A_h^T A_h + A_v^T A_v$ is a discrete Laplacian with grid representation given on the right in Figure 4.1.

### 4.3.2 Wavefront Reconstruction Using TSVD-Type Regularization

We now discuss the mathematical problem at the heart of this reconstruction [110, 111]: solving equation (4.8) for $\boldsymbol{\phi}$, which is a rank-deficient problem. Thus, we must solve for $\boldsymbol{\phi}$ in the least-squares sense.

In the case of Fried geometry, we have:

$$\min_{\boldsymbol{\phi}} \left\| \begin{bmatrix} F \otimes H \\ H \otimes F \end{bmatrix} \boldsymbol{\phi} - \boldsymbol{b} \right\|_2^2 \tag{4.10}$$

We compute the GSVD of $F$ and $H$, such that $F = U_1 \Sigma_1 X_1^T$ and $H = V_1 \Delta_1 X_1^T$. Then, using equations (4.4) and (4.6), we can see that:

$$\begin{bmatrix} F \otimes H \\ H \otimes F \end{bmatrix} = \begin{bmatrix} U_1 \otimes V_1 & \\ & V_1 \otimes U_1 \end{bmatrix} \begin{bmatrix} \Sigma_1 \otimes \Delta_1 \\ \Delta_1 \otimes \Sigma_1 \end{bmatrix} (X_1 \otimes X_1)^T \equiv U \Sigma X^T. \tag{4.11}$$

Since $\Sigma_1$ is a superdiagonal $(n-1) \times n$ matrix and $\Delta_1$ is a diagonal $(n-1) \times n$ matrix, the matrix $\Sigma$ is a $2(n-1)^2 \times n^2$ matrix, composed of stacking two roughly diagonal matrices atop each other; an illustration of the $\Sigma$ matrix for the case $n = 6$

is given in Figure 4.2. This matrix is rank deficient; its first and last columns are all zero. By systematically applying Givens rotations, we can reduce $\Sigma$ to diagonal form; we denote this as $\Sigma = QD$, where the matrix $Q$ contains the Givens rotations.

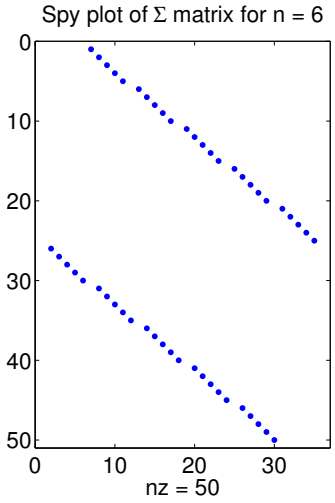



Figure 4.2: Visualization of nonzeros in $\Sigma$ matrix for the case $n = 6$.

Using equation (4.10), we obtain

$$\begin{aligned}
\min_{\boldsymbol{\phi}} \left|\left| \begin{bmatrix} F \otimes H \\ H \otimes F \end{bmatrix} \boldsymbol{\phi} - \boldsymbol{b} \right|\right|_2^2 &= \min_{\boldsymbol{\phi}} \left|\left| U\Sigma X^T \boldsymbol{\phi} - \boldsymbol{b} \right|\right|_2^2 \\
&= \min_{\boldsymbol{\phi}} \left|\left| UQDX^T \boldsymbol{\phi} - \boldsymbol{b} \right|\right|_2^2 \\
&= \min_{\boldsymbol{\phi}} \left|\left| DX^T \boldsymbol{\phi} - Q^T U^T \boldsymbol{b} \right|\right|_2^2 \\
&= \min_{\widehat{\boldsymbol{\phi}}} \left|\left| D\widehat{\boldsymbol{\phi}} - \hat{\boldsymbol{b}} \right|\right|_2^2, \quad \text{where } \widehat{\boldsymbol{\phi}} = X^T \boldsymbol{\phi} \text{ and } \hat{\boldsymbol{b}} = Q^T U^T \boldsymbol{b}.
\end{aligned}$$

Since $D$ is a diagonal matrix, it becomes trivial to solve this minimization problem. Due to our organization of Givens rotations, the last two diagonal elements of $D$ are 0; we have freedom in deciding what to use for the solution terms corresponding to these values. We effectively truncate these two elements by setting their reciprocals to 0 in the pseudo-inverse $\widehat{D}^{-1}$. This is where the term TSVD-like regularization comes from. It is important to note that the remaining $n^2 - 2$ values of $D$ are relatively large; the problem is rank-deficient, but it does not have the classical properties of an ill-posed problem, which are that there are many small singular values, and no gap to indicate a numerical rank.

Because $F$ and $H$ are defined by the geometry of the telescope, the generalized singular value decomposition may be performed once and stored. Thus, $U_1$, $V_1$, $X_1$,

$D$, and the rotators defining $Q$ can be precomputed off line. Additionally, a suitable factorization of $X_1$ (such as the $LU$ decomposition) can be precomputed off line. It is important to note that we never actually explicitly compute $U, \Sigma$, or $X$, but instead we always work with the smaller matrices that make up these larger ones, using the properties of Kronecker products. In particular, if we let $\text{vec}\left(\begin{bmatrix} B_h & B_v \end{bmatrix}\right) = \boldsymbol{b}$ and $\text{vec}(\Phi) = \boldsymbol{\phi}$, the work to be done in real time then becomes

1. Compute $\hat{\boldsymbol{b}} = Q^T U^T \boldsymbol{b} \Leftrightarrow \widehat{B} = \widehat{Q}^T \left( \begin{bmatrix} V_1^T B_h U_1 \\ U_1^T B_v V_1 \end{bmatrix} \right)$, where $\widehat{Q}^T(A)$ applies the Givens rotators described in $Q^T$ to the matrix $A$, retaining matrix form throughout.
2. Solve $\min_{\widehat{\boldsymbol{\phi}}} \left|\left| D\widehat{\boldsymbol{\phi}} - \hat{\boldsymbol{b}} \right|\right|_2^2 \Leftrightarrow \widehat{\Phi} = \widehat{B} \odot \widehat{D}^{-1}$, where $\widehat{D}^{-1}$ contains the pseudo-inverse of the diagonal elements of $D$, with two elements of $\widehat{D}^{-1}$ set to 0. The symbol $\odot$ represents Hadamard (i.e., element-wise) multiplication.
3. Solve $X^T \boldsymbol{\phi} = \widehat{\boldsymbol{\phi}} \Leftrightarrow \Phi = X_1^{-T} \widehat{\Phi} X_1^{-1}$, using our precomputed factorization of $X_1$.

### 4.3.3 Wavefront Reconstruction Using Tikhonov-Type Regularization

While solving equation (4.10) using TSVD-type regularization works well, astronomers prefer a form of Tikhonov regularization motivated by *a priori* knowledge of the phase error statistics. The values of $\boldsymbol{\phi}$ are determined by turbulent temperature variations within certain layers of the earth's atmosphere. The turbulence is often modeled by a Gaussian random vector with mean $\mathbf{0}$ and a covariance matrix $C_{\boldsymbol{\phi}}$ with spectrum given by the von Karman turbulence model [112]. In [34], it is shown that the inverse biharmonic (or squared Laplacian) matrix gives an accurate, sparse approximation of $C_{\boldsymbol{\phi}}$ (see also [10]). Using this approximation, the minimum variance estimator for the true phase error is obtained by solving

$$\min_{\boldsymbol{\phi}} \left|\left| \begin{bmatrix} F \otimes H \\ H \otimes F \end{bmatrix} \boldsymbol{\phi} - \begin{bmatrix} \boldsymbol{b}_h \\ \boldsymbol{b}_v \end{bmatrix} \right|\right|_2^2 + (\sigma^2/c_0)\|L\boldsymbol{\phi}\|^2, \tag{4.12}$$

where $L$ is a discretization of the Laplacian; $\sigma^2$ is the variance of the iid Gaussian noise vector $\boldsymbol{\eta}$ in (4.8); and $c_0$ is a scaling parameter chosen so that $E(\boldsymbol{\phi}^T C_{\boldsymbol{\phi}} \boldsymbol{\phi}) = c_0 E(\boldsymbol{\phi}^T L^{-2} \boldsymbol{\phi})$; see [10, 46] for more details. As is the case in other AO literature, we assume that the user has an estimate of $\sigma^2$ in hand, from which $\alpha$ can be computed directly.

In order to express $L$ in Kronecker product form, we estimate it using the Laplacian arising from the Hudgin geometry. In particular,

$$||L\boldsymbol{\phi}||^2 \stackrel{\text{def}}{=} \left|\left|\left[\begin{array}{c} (I\otimes H)\boldsymbol{\phi} \\ (H\otimes I)\boldsymbol{\phi} \end{array}\right]\right|\right|^2,$$

where $H$ is defined in (4.9); note then that $L^TL = I\otimes(H^TH) + (H^TH)\otimes I$.

Equating $\sigma^2/c_0$ with the regularization parameter $\alpha$, problem (4.12) can be written as

$$\min_{\boldsymbol{\phi}} \left|\left|\left[\begin{array}{c} F\otimes H \\ H\otimes F \\ \alpha I\otimes H \\ H\otimes \alpha I \end{array}\right]\boldsymbol{\phi} - \left[\begin{array}{c} \boldsymbol{b}_h \\ \boldsymbol{b}_v \\ \mathbf{0} \\ \mathbf{0} \end{array}\right]\right|\right|_2^2. \tag{4.13}$$

An important observation may be made regarding the coefficient matrix in (4.13); a permutation matrix $\Pi$ exists such that

$$\Pi\left[\begin{array}{c} F\otimes H \\ H\otimes F \\ \alpha I\otimes H \\ H\otimes \alpha I \end{array}\right] = \left[\begin{array}{c} \left[\begin{array}{c} F \\ \alpha I \end{array}\right]\otimes H \\ H\otimes\left[\begin{array}{c} F \\ \alpha I \end{array}\right] \end{array}\right] \equiv \left[\begin{array}{c} F_\alpha\otimes H \\ H\otimes F_\alpha \end{array}\right].$$

Thus, if the value of $\alpha$ did not change from one data set to another, we could apply the techniques of the previous section and solve this problem in the same manner. Namely, we could compute the GSVD of $F_\alpha$ and $H$, then efficiently solve using matrix-matrix multiplications and Hadamard arithmetic.

Unfortunately, the value of $\alpha$ does not remain the same between data sets. However, though the value of $\alpha$ will typically change, the range of $\alpha$ values is restricted. Thus, we can choose an "average" $\alpha$ value, denoted $\alpha_0$, that approximates the value of $\alpha$ for any data set, and use this $\alpha_0$ value to form a preconditioner. More details are given in Section 4.3.3.1.

To demonstrate how we construct the preconditioner, we rewrite equation (4.13) as

$$\min_{\boldsymbol{\phi}} \left|\left|\left[\begin{array}{c} A_1 \\ A_2 \end{array}\right]\boldsymbol{\phi} - \left[\begin{array}{c} \boldsymbol{b} \\ \mathbf{0} \end{array}\right]\right|\right|_2^2, \quad \text{where } A_1 = F_{\alpha_0}\otimes H, A_2 = H\otimes F_{\alpha_0}.$$

Our goal is to find a preconditioner $M$ such that $M^TM \approx A_1^TA_1 + A_2^TA_2$, while retaining the desirable Kronecker product structure. We begin by calculating the

GSVD of $F_{\alpha_0}$ and $H$: $F_{\alpha_0} = U\Sigma X^T, H = V\Delta X^T$. Observe that

$$
\begin{aligned}
&A_1^T A_1 + A_2^T A_2 \\
&= (X\Sigma^T U^T \otimes X\Delta^T V^T)(U\Sigma X^T \otimes V\Delta X^T) + (X\Delta^T V^T \otimes X\Sigma^T U^T)(V\Delta X^T \otimes U\Sigma X^T) \\
&= (X\Sigma^T U^T U\Sigma X^T) \otimes (X\Delta^T V^T V\Delta X^T) + (X\Delta^T V^T V\Delta X^T) \otimes (X\Sigma^T U^T U\Sigma X^T) \\
&= (X\Sigma^T \Sigma X^T) \otimes (X\Delta^T \Delta X^T) + (X\Delta^T \Delta X^T) \otimes (X\Sigma^T \Sigma X^T) \\
&= (X \otimes X)(\Sigma^T\Sigma \otimes \Delta^T\Delta)(X \otimes X)^T + (X \otimes X)(\Delta^T\Delta \otimes \Sigma^T\Sigma)(X \otimes X)^T \\
&= (X \otimes X)(\Sigma^T\Sigma \otimes \Delta^T\Delta + \Delta^T\Delta \otimes \Sigma^T\Sigma)(X \otimes X)^T \\
&= (X \otimes X)C(X \otimes X)^T,
\end{aligned}
$$

where $C = \Sigma^T\Sigma \otimes \Delta^T\Delta + \Delta^T\Delta \otimes \Sigma^T\Sigma$ is a diagonal matrix.

We want $M^T M \approx (X \otimes X)C(X \otimes X)^T$, which we achieve (exactly) by setting $M = QC^{1/2}(X \otimes X)^T$. $Q$ can be any orthogonal matrix; as long as it has an appropriate Kronecker structure, the computations using $M$ will be efficient. For all the results in this chapter, we set $Q$ to be the identity matrix.

We remark that the preconditioner $M$ is computed once using an appropriate average $\alpha_0$ for all likely data sets. Thus, we do not need to consider the computational costs of computing the GSVD of $F_{\alpha_0}$ and $H$ as part of the real-time time constraints needed by the application. The appropriate $Q$, $X$ and $C^{1/2}$ factors will be precomputed once and stored, or, rather, their inverses will be formed and stored. Since $C$ contains one zero diagonal element, we set the corresponding element in $C^{-1/2}$ to be zero.

Using the preconditioner, the problem now becomes

$$
\min_{\boldsymbol{\phi}} \left\| AM^{-1}M\boldsymbol{\phi} - \begin{bmatrix} \boldsymbol{b} \\ \mathbf{0} \end{bmatrix} \right\|_2^2, \quad \text{where } A = \begin{bmatrix} F \otimes H \\ H \otimes F \\ \alpha I \otimes H \\ H \otimes \alpha I \end{bmatrix}. \tag{4.14}
$$

We use preconditioned LSQR [102, 103] to solve this problem. Note that, again, all matrix-vector multiplications can be performed by exploiting the Kronecker product structure of this problem and using matrix-matrix multiplications. This exploitation occurs both when applying the preconditioner (or its transpose) or when multiplying a vector by $A$ or $A^T$.

#### 4.3.3.1 Approximation Quality of Preconditioner

It is important to consider how well a preconditioner approximates a matrix to determine the quality of the preconditioner. In this section, we consider the approximation

quality of the preconditioner in two ways: first, by looking at the singular values of the original matrix and of the preconditioned system, and second, by looking at the relative errors of the matrix and its preconditioner.

To obtain realistic values for $\alpha_0$, we ran a series of Matlab simulations. We created an $n^2 \times 1$ "true" $\boldsymbol{\phi}$ [10] and formed $\boldsymbol{b}_h$ and $\boldsymbol{b}_v$ from equation (4.8). We then repeated the following 1000 times: we added a realization of a certain noise level (such as 10%) to the gradients $\boldsymbol{b}_h$ and $\boldsymbol{b}_v$, then determined an optimal $\alpha$ value for that realization by solving equation (4.13) for multiple values of $\alpha$. An $\alpha$ value was considered optimal if it produced a reconstructed $\boldsymbol{\phi}$ with minimal relative residual error. After finding the 1000 optimal $\alpha$ values for a given size $n$ and noise level, we determined the $\alpha_0$ for that $n$-noise level combination by using the value at the peak of the histogram of the $\alpha$ values. As $n$ increases, the range of optimal $\alpha$ values decreases, leading to a closer fit of $\alpha_0$ for all $\alpha$ values. Though all of our observations on $\alpha$ and $\alpha_0$ come from simulations on realistic data, we feel that similar results will be obtained once real data is used. As the $\alpha$ value can be found without knowledge of the true $\boldsymbol{\phi}$ (see [10]), the $\alpha_0$ value for real data can be found by again using the peak value of many $\alpha$ values.

We consider the singular values of the matrix $A$ given in equation (4.14) as well as the singular values of the preconditioned matrix $AM^{-1}$. We found that the singular values of the preconditioned system cluster to 1, with tighter clustering the closer $\alpha_0$ is to $\alpha$. Figure 4.3 shows the singular values for six preconditioned systems as well as the singular values for one non-preconditioned system. In this figure, we took $n$ to be 64 and $\alpha_0$ to be 0.058. We varied $\alpha$ to be from 0.03 to 0.124, which were the extremes of the optimal $\alpha$ values we found during the 1000 realizations of 10% noise. The dots above the dashed line show the singular values for these six preconditioned systems. Note that each preconditioned system has one zero singular value, while the remaining singular values cluster toward one. In particular, for the case $\alpha = \alpha_0$, all of the nonzero singular values cluster at one. For comparison, we include the singular values of one non-preconditioned system, for the case $\alpha = 0.058$. These are displayed in the figure below the dashed line. The smallest nonzero singular value for this system is 0.0488 and the largest is 2.0022.

The second measure of the quality of the preconditioner can be obtained by considering the relative errors of the matrix and its preconditioner. Let $\alpha_0$ be the value used in the preconditioner $M$ and suppose $\alpha$ is the actual value specified by the data

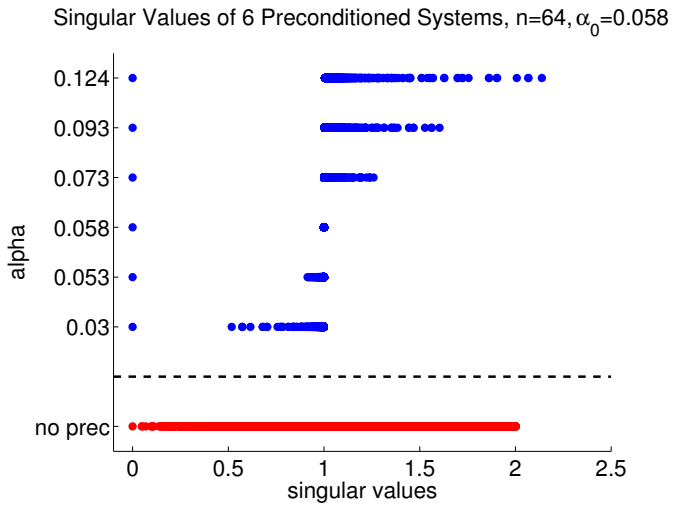



Figure 4.3: Singular values for one non-preconditioned and six preconditioned systems.

collection process. Then,

$$\begin{aligned} A^TA &= (F_\alpha^T F_\alpha) \otimes (H^T H) + (H^T H) \otimes (F_\alpha^T F_\alpha) \\ M^TM &= (F_{\alpha_0}^T F_{\alpha_0}) \otimes (H^T H) + (H^T H) \otimes (F_{\alpha_0}^T F_{\alpha_0}). \end{aligned}$$

We can compute the error between these matrices as

$$E = A^TA - M^TM = (\alpha^2 - \alpha_0^2)I \otimes H^TH + H^TH \otimes (\alpha^2 - \alpha_0^2)I.$$

Due to the simple nature of these matrices, we can explicitly compute the Frobenius norm of each:

$$\begin{aligned} \left|\left|A^TA\right|\right|_F &= \sqrt{4\alpha^4(5n^2 - 8n + 2) + 8\alpha^2(2n^2 - 5n + 3) + 5n^2 - 14n + 10} \\ \left|\left|M^TM\right|\right|_F &= \sqrt{4\alpha_0^4(5n^2 - 8n + 2) + 8\alpha_0^2(2n^2 - 5n + 3) + 5n^2 - 14n + 10} \\ ||E||_F &= 2|\alpha^2 - \alpha_0^2|\sqrt{5n^2 - 8n + 2}. \end{aligned}$$

Thus,

$$\begin{aligned} \lim_{n\to\infty} \frac{||E||_F}{\left|\left|A^TA\right|\right|_F} &= \lim_{n\to\infty} \frac{2\left|\alpha^2 - \alpha_0^2\right| \sqrt{5 - \frac{8}{n} + \frac{2}{n^2}}}{\sqrt{4\alpha^4(5 - \frac{8}{n} + \frac{2}{n^2}) + 8\alpha^2(2 - \frac{5}{n} + \frac{3}{n^2}) + (5 - \frac{14}{n} + \frac{10}{n^2})}} \\ &= \frac{2\sqrt{5}\left|\alpha^2 - \alpha_0^2\right|}{\sqrt{20\alpha^4 + 16\alpha^2 + 5}} \\ &\leq 2\left|\alpha^2 - \alpha_0^2\right|. \end{aligned}$$

## 4.4 Implementation Details

This problem is currently implemented in Matlab and Trilinos for use on a single processor. As of now, there are no matrix-matrix multiplication routines for `Tpetra::CrsMatrix` objects in Trilinos. However, these routines are being developed and should be included in a future release.

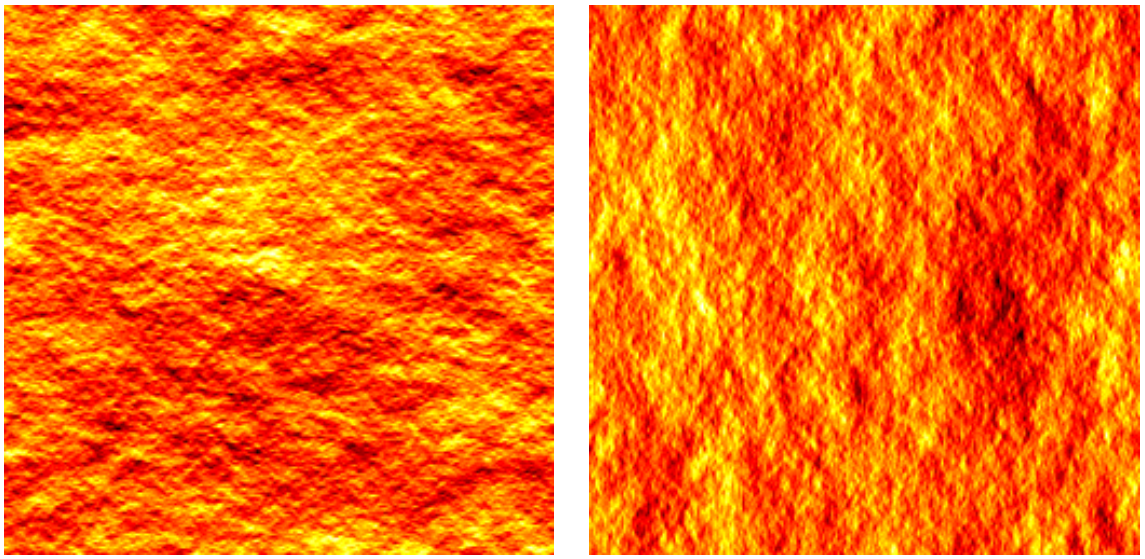
For our implementation, then, we use `Teuchos::SerialDenseMatrix` objects. These are serial objects, but they provide matrix-matrix multiplication operations. When applying the operator or the preconditioner to a vector, we first reshape the vector into one or more appropriately-sized matrices by copying the values to new `SerialDenseMatrix` objects. We then apply the matrices that make up the operator or the preconditioner to these matrices. We finish by copying the values to the output vector at the end. We obtain the matrices that make up the preconditioner via files rather than constructing them from scratch.

Another approach may be to use `Epetra_CrsMatrix` objects and the `EpetraExt::MatrixMatrix` class to perform matrix-matrix multiplications. This requires us to use `double` and `int` datatypes. However, these classes were designed for sparse matrices whereas some of the matrices we use are dense.

## 4.5 Results

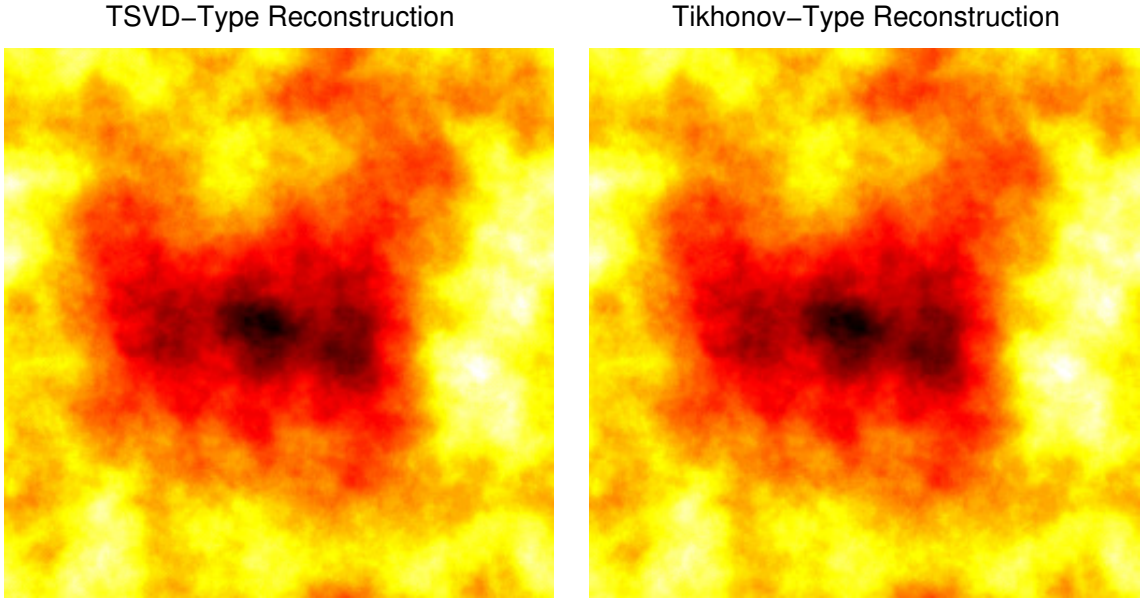
We now present some results showing the effectiveness of our preconditioning approach, even when $\alpha_0$ differs from the $\alpha$ value computed for a data set. We determined the optimal $\alpha_0$ for $n = 256$ with 10% noise using the procedure described in Section 4.3.3.1. We then consider how well our preconditioner works when masking is involved; that is, when the aperture is not a square. First, though, we give an example of the noisy data and reconstructed solution.

Figure 4.4 shows a realization of the noisy phase differences $B_h$ and $B_v$ for the case $n = 256$ with 10% noise. Figure 4.5 shows the reconstructed $\Phi$ for the TSVD-type regularization on the left and the Tikhonov-type regularization on the right. Visually, the results appear indistinguishable. However, the relative errors differ slightly; the relative error for the TSVD solution is 1.058e-2 and for the Tikhonov solution is 9.414e-3 – a roughly 10% improvement. This result, that the Tikhonov approach produces a solution with slightly less relative error than the TSVD solution, is consistent for all the noise realizations, and explains why the regularized approach is often favored by astronomers [34].



(a) Noisy $B_h$ (b) Noisy $B_v$

Figure 4.4: Example noisy gradients for $n = 256$ with 10% noise.




(a) TSVD Reconstruction (b) Tikhonov Reconstruction

Figure 4.5: Comparison of reconstructions for two types of regularization.

### 4.5.1 Effect of Differing $\alpha$ Values

The next set of results we present focus entirely on the Tikhonov-type regularization. We look at the number of preconditioned LSQR iterations required for different data sets, solving to a tolerance of $10^{-6}$. Figure 4.6 compares the value of $\alpha$ to the number of preconditioned LSQR iterations required for 1000 realizations of 10% noise. As expected, when $\alpha$ exactly equals $\alpha_0$, only one preconditioned LSQR iteration is required. As $\alpha$ becomes further from the preconditioner value $\alpha_0$ (either being smaller or larger than $\alpha_0$), the number of LSQR iterations required increases, though it never takes more than a handful of iterations to reach convergence. Figure 4.7 displays the results from this same set of simulations but in a different way. Here, we display the number of times a given number of preconditioned LSQR iterations is required during our 1000 simulations. For example, in nearly one-fourth of the experiments, exactly three LSQR iterations are required to reach a convergence of $10^{-6}$.

It is important to consider different levels of noise. The previous results were based on 10% noise, which may be more or less than in real life. Thus, we wish to see how well our approach works for a larger range of noise, namely from 1% up to 20%. Table 4.1 gives the average number of LSQR iterations required both with and without preconditioning, when solved to different tolerance levels and with different amounts of noise.

As can be seen, the average number of preconditioned LSQR iterations required increases very slowly with the noise level. On the other hand, the number of unpreconditioned iterations required decreases as the noise level increases. Regardless if preconditioning is used, the number of average iterations required goes up as the tolerance level goes down.

### 4.5.2 Using Square-Aperture Preconditioner on Masked Problems

The final results we look at will consider the effectiveness of our preconditioner on a problem with masking. As built, the preconditioner assumes a square aperture. That is, the characteristic function in equation (4.1) returns 1 for all input. Now we wish to solve the problem

$$\min_{\boldsymbol{\phi}} \left\| \begin{bmatrix} F \otimes H \\ H \otimes F \\ \alpha I \otimes H \\ H \otimes \alpha I \end{bmatrix} P\boldsymbol{\phi} - \begin{bmatrix} \boldsymbol{b}_h \\ \boldsymbol{b}_v \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix} \right\|_2^2 ,$$

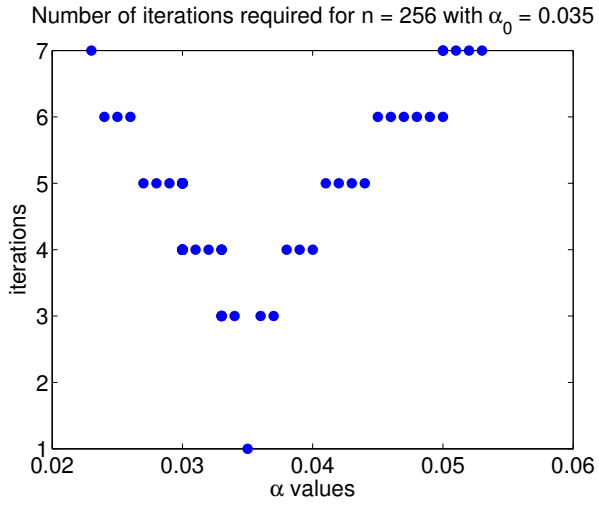



Figure 4.6: Comparison of iterations to $\alpha$ value for 1000 realizations of noise.

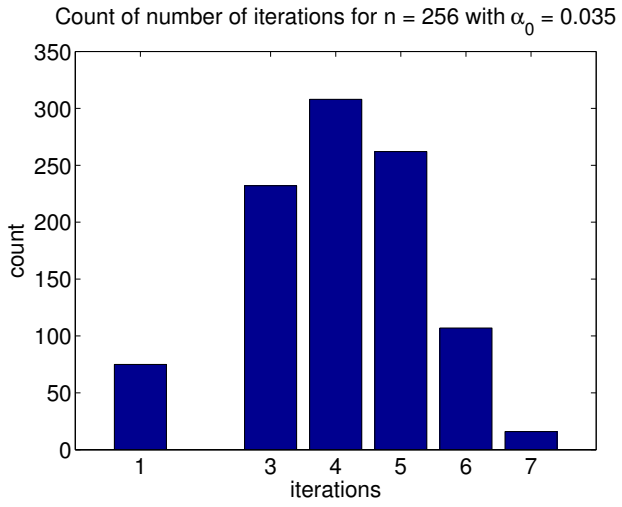



Figure 4.7: Count of number of preconditioned LSQR iterations required for 1000 realizations of noise.

Table 4.1: Number of LSQR iterations required, on average, for $n = 256$ with four different amounts of noise for various tolerance levels.

| Noise level | Preconditioning? | $1e-1$ | $1e-2$ | $1e-3$ | $1e-4$ | $1e-5$ | $1e-6$ |
|---|---|---|---|---|---|---|---|
| 1% | No | 5 | 156 | 346 | 490 | 554 | 611 |
| 1% | Yes | 1 | 1 | 1 | 2 | 3 | 3 |
| 5% | No | 5 | 112 | 220 | 408 | 531 | 578 |
| 5% | Yes | 1 | 1 | 2 | 2 | 3 | 4 |
| 10% | No | 5 | 98 | 193 | 372 | 515 | 564 |
| 10% | Yes | 1 | 1 | 2 | 3 | 3 | 4 |
| 20% | No | 5 | 81 | 175 | 345 | 488 | 554 |
| 20% | Yes | 1 | 1 | 2 | 3 | 3 | 4 |

where $P$ is a diagonal matrix consisting of 0s and 1s on the diagonal. When $P$ is the identity matrix, as is the case of a square aperture, we are solving exactly (4.13). On the other hand, when masking is involved, some of the diagonal entries of $P$ will be zero. We can equivalently consider stripping the diagonal from $P$ and reshaping it into a matrix of size $n \times n$; this will allow us to apply the masking operation via an element-wise matrix multiplication. The two masks we will consider are a circular mask and an annular mask. These are illustrated in Figure 4.8 for the case $n = 64$.

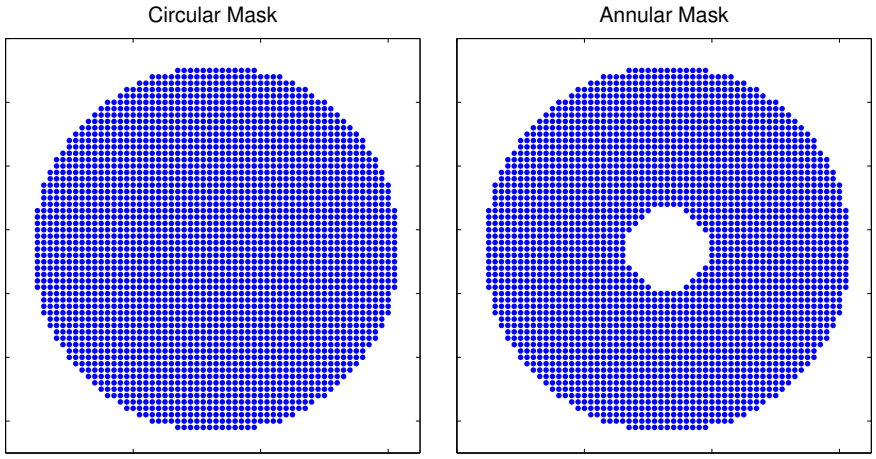
For these results, we again use only Tikhonov regularization. We vary $n$ from 64 to 256, using five different $\alpha$ values for each size. The $\alpha$ values were chosen to span the range of values, including those values that occurred most often. The tolerance levels for solving are the same as those used previously, namely from $1e-1$ to $1e-6$.

We look at the number of iterations required to converge to the desired tolerance level with and without a preconditioner, given in Figure 4.9. We also consider the masked relative error, which is simply the relative error of our reconstructed phase $\Phi$ after the mask has been applied to it. This discards the values of $\Phi$ for which we have no gradient data. The results relating the masked relative error to the tolerance level are shown in Figure 4.10. For these figures, the results using the circular mask are given with a solid dot while those for the annular pupil use an open circle.

As expected, the number of iterations required increases as the tolerance level decreases. Interestingly, though, it appears that the number of iterations does not vary too much as $n$ increases in the preconditioned case, whereas in the unpreconditioned case the iteration count significantly increases as the problem size increases.

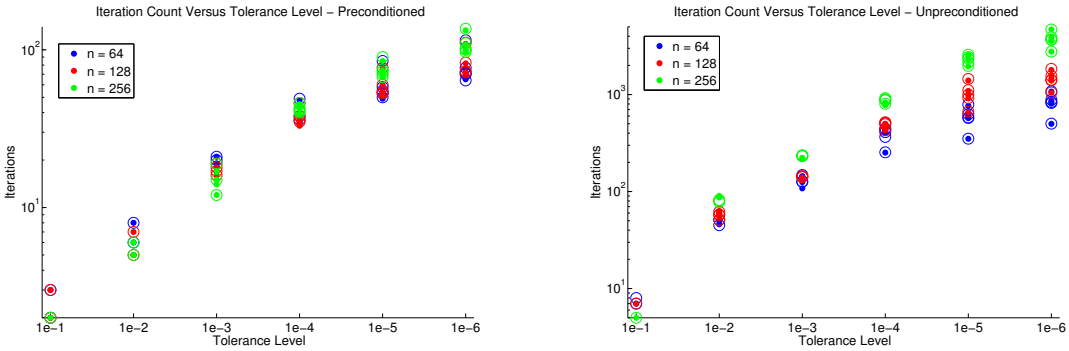
The smallest $\alpha$ value takes significantly more iterations to converge than any other $\alpha$ value; this implies that we may wish to choose $\alpha_0$ a little low to encourage faster convergence in general. The results between the different masks, in terms of iteration count, are quite similar.

When looking at the masked relative error, the results when preconditioning is used vary little as the tolerance level decreases. However, the effect of the size has more impact here than it does on the iteration count, with the largest $n$ yielding results with the smallest masked relative error. On the other hand, without preconditioning, the results are less consistent, though for $n = 256$, there is a general trend of reduced error with more stringent tolerance levels. Whether or not preconditioning is used, the results appear to be more dependent on the particular problem than on the aperture of the telescope.



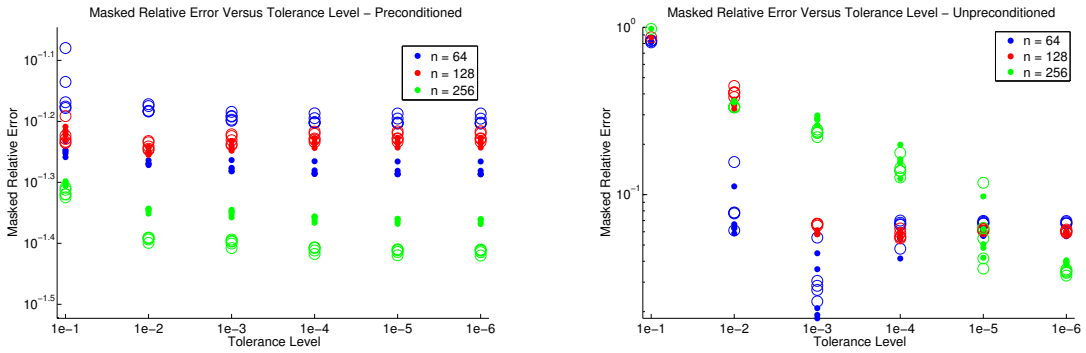

(a) Circular Mask (b) Annular Mask

Figure 4.8: Illustration of masks used.




(a) With a preconditioner (b) Without a preconditioner

Figure 4.9: Comparison of iterations to tolerance level with and without a preconditioner.




(a) With a preconditioner

(b) Without a preconditioner

Figure 4.10: Comparison of masked relative error to tolerance level with and without a preconditioner.

## 4.6 Remarks and Future Directions

A Kronecker product-structured, rank-deficient least squares adaptive optics problem was described, for determining the phase error from noisy phase differences. Efficient techniques for solving on the Fried geometry with square aperture were described. Two types of regularization were considered: a TSVD-type and a preferred Tikhonov-type. Using properties of Kronecker products and the GSVD, a direct solution using TSVD regularization was described. Though this gives good results, a relatively better reconstruction may be obtained using Tikhonov-type regularization. By forming a preconditioner that will work well with any data set, fast convergence is obtained when using LSQR. Approximation quality results of the preconditioner were given, as well as numerical results on simulated data. It was shown that as long as the $\alpha_0$ value used by the preconditioner is suitably chosen, the number of preconditioned LSQR iterations required is small. We have also shown positive results in using the preconditioner developed for square apertures on problems with circular or annular apertures. Exploiting the linear algebra structure of this problem has led to extremely efficient computational approaches that can be solved in real time.

Further investigations including a Trilinos implementation that can run on multiple processors with larger values of $n$ would be interesting.

# Chapter 5

# Concluding Remarks

We have described a variety of inverse problems, from linear to separable nonlinear to fully nonlinear, using a general model problem. By considering the properties of inverse problems, in particular their ill-posedness, we have seen why regularization is needed to obtain a reasonable solution. We have also considered different techniques for solving and including regularization. A common approach to including regularization is to solve using an iterative method, terminating the iterations before full convergence has been reached.

When solving large-scale inverse problems, iterative methods are often the best choice. No structure is required, allowing these methods to work on general problems. Parallel computing also combines well with iterative methods, allowing numerous computing nodes to work in tandem with each other. Trilinos is a robust library of mathematical software for solving large-scale problems in parallel. In particular, the Belos package of Trilinos contains a framework for iterative linear solvers. We have described two iterative solvers, LSQR and MRNSD, that we have implemented using this framework. We have also implemented a status test class that determines the iteration providing least relative error, when the true solution is known.

Next we considered two different linear inverse problems, arising from distinct fields, that both benefit by being solved in an iterative manner. Our first case study came from the field of medical imaging. In particular, we wished to remove patient motion blur in PET brain images. We described the motion information that is collected during a scan and formulated a large linear problem from it. Depending on how much or little of the motion information we include, by averaging over small or large periods of time respectively, the matrix loses or gains sparsity. We then implemented our application in C++, using the Trilinos library, and did a variety of simulations. We found that the type of scalars used, single or double precision, had

little bearing on the results. On the other hand, breaking the motion information into more intervals resulted in significantly better reconstructions. At times the relative interval lengths made a difference, whether the intervals were chosen by hand or were of equal size, though when many intervals were used the difference was minimal. Varying levels of patient motion were also considered; the same trend of better results, in terms of reduction in error, was obtained as the number of intervals used increased.

Our second case study came from adaptive optics, where we wished to reconstruct a distorted wavefront given noisy gradient information. We described two different sensor geometries, which determine the operators in the linear problem. We focused only on the Fried geometry, which is more complex and mathematically interesting. Due to the Kronecker product structure of the problem, we considered using the generalized singular value decomposition in our solving process. From there, we determined a direct method for solving, requiring only matrix-matrix and Hadamard multiplications. However, we could obtain slightly better results by incorporating Tikhonov regularization, as the regularization parameter can be determined by statistics of the data. Here we determined a preconditioner to be used for all problems of a given size and considered its approximation quality. Finally we considered some results by using preconditioned LSQR to solve our problems. We again performed only matrix-matrix and Hadamard multiplications, allowing us great efficiency. We saw that good results could be obtained in just a few iterations, with the iteration count slowly increasing as the noise level increased. We also considered using our square-aperture preconditioner when solving problems with a circular or annular aperture. Using the preconditioner greatly reduced the number of iterations required compared to solving without a preconditioner.

The mathematical and computational contributions we have made using these two applications as case studies can be applied to inverse problems in a variety of other fields.

# Appendix A

# Trilinos Code

We now include our C++ code, utilizing the Trilinos library, described in the previous manuscript. Sections A.1 and A.2 contain the code for implementing LSQR and MRNSD within the Belos framework. Each contains a solver manager, iteration, and status test class. Section A.3 includes our least error status test class. We then present our implementations for our two case studies. Code to solve the PET brain motion deblurring problem is given in Section A.4, while Section A.5 contains our code for the adaptive optics application. For each of these application codes, an example extensible markup language (XML) file is also included. The XML file contains information related to the specific problem to be solved, such as names of files, problem size, and solver parameters.

## A.1 Code for LSQR

### A.1.1 LSQRSolMgr.hpp Code

```
#ifndef LSQR_SOLMGR_HPP
#define LSQR_SOLMGR_HPP

/* LSQRSolMgr.hpp
 * The LSQRSolMgr provides a solver manager for the LSQR linear solver.
 */

#include "BelosConfigDefs.hpp"
#include "BelosTypes.hpp"
#include "BelosLinearProblem.hpp"
#include "BelosSolverManager.hpp"
#include "BelosStatusTestMaxIters.hpp"
#include "BelosStatusTestCombo.hpp"
```

```
#include "BelosStatusTestOutputFactory.hpp"
#include "BelosOutputManager.hpp"
#include "Teuchos_TimeMonitor.hpp"

#include "LSQRIter.hpp"
#include "LSQRStatusTest.hpp"
#include "LeastErrorStatusTest.hpp"

// LSQRSolMgr Exceptions
/* LSQRSolMgrLinearProblemFailure is thrown when the linear problem is not setup
 * (i.e. setProblem() was not called) when solve() is called.
 * This std::exception is thrown from the LSQRSolMgr::solve() method.
 */
class LSQRSolMgrLinearProblemFailure : public Belos::BelosError {public:
    LSQRSolMgrLinearProblemFailure(const std::string& what_arg) : Belos::BelosError(
        what_arg)
  {}};

/* LSQRSolMgrBlockSizeFailure is thrown when the linear problem has more than
 * one RHS.  This std::exception is thrown from the LSQRSolMgr::solve() method.
 */
class LSQRSolMgrBlockSizeFailure : public Belos::BelosError {public:
    LSQRSolMgrBlockSizeFailure(const std::string& what_arg) : Belos::BelosError(what_arg)
  {}};

template<class ScalarType, class MV, class OP>
class LSQRSolMgr : public Belos::SolverManager<ScalarType,MV,OP> {

private:
  typedef Belos::MultiVecTraits<ScalarType,MV> MVT;
  typedef Belos::OperatorTraits<ScalarType,MV,OP> OPT;
  typedef Teuchos::ScalarTraits<ScalarType> SCT;
  typedef typename Teuchos::ScalarTraits<ScalarType>::magnitudeType MagnitudeType;
  typedef Teuchos::ScalarTraits<MagnitudeType> MT;

public:

  // Constructors/Destructor
  /* Empty constructor for LSQRSolMgr.  This constructor takes no arguments and
   * sets the default values for the solver.  The linear problem must be passed
   * in using setProblem() before solve() is called on this object.  The solver
   * values can be changed using setParameters().
   */
  LSQRSolMgr();
```

```
/* Basic constructor for LSQRSolMgr.  This constructor accepts the
 * LinearProblem to be solved in addition to a parameter list of options for
 * the solver manager. These options include the following:
 * - "Maximum Iterations" - an int specifying the maximum number of iterations
 * the underlying solver is allowed to perform. Default: 1000
 * - "Condition Limit" - a MagnitudeType specifying the upper limit of the
 * estimate of the norm of Abar to decide convergence. Default: 0
 * - "Term Iter Max" - the number of consecutive successful iterations
 * required before convergence is declared. Default: 1
 * - "Rel RHS Err" - an estimate of the error in the data defining the RHS.
 * Default: 0
 * - "Rel Mat Err" - an estimate of the error in the data defining the matrix.
 * Default: 0
 * - "Verbosity" - a sum of MsgType specifying the verbosity. Default:
 * Belos::Errors
 * - "Output Style" - an OutputType specifying the style of output. Default:
 * Belos::General
 * - "Output Stream" - a reference-counted pointer to the output stream where
 * all solver output is sent.  Default: Teuchos::rcp(&std::cout,false)
 * - "Output Frequency" - an int specifying how often convergence information
 * should be outputted.  Default: -1 (never)
 * - "Timer Label" - a std::string to use as a prefix for the timer labels.
 * Default: "Belos"
 */
LSQRSolMgr( const Teuchos::RCP<Belos::LinearProblem<ScalarType,MV,OP> > &problem, const
     Teuchos::RCP<Teuchos::ParameterList> &pl );

// Destructor.
virtual ~LSQRSolMgr() {};

// Accessor methods
const Belos::LinearProblem<ScalarType,MV,OP>& getProblem() const {
  return *problem_;
}

// Get a parameter list containing the valid parameters for this object.
Teuchos::RCP<const Teuchos::ParameterList> getValidParameters() const;

//Get a parameter list containing the current parameters for this object.
Teuchos::RCP<const Teuchos::ParameterList> getCurrentParameters() const { return
    params_; }

/* Return the timers for this object.
 * The timers are ordered as follows - time spent in solve() routine.
 */
```

```
Teuchos::Array<Teuchos::RCP<Teuchos::Time> > getTimers() const {
  return tuple(timerSolve_);
}

// Get the iteration count for the most recent call to solve().
int getNumIters() const {
  return numIters_;
}

/* Return whether a loss of accuracy was detected by this solver during the
 * most current solve.
 */
bool isLOADetected() const { return false; }

// Set methods
// Set the linear problem that needs to be solved.
void setProblem( const Teuchos::RCP<Belos::LinearProblem<ScalarType,MV,OP> > &problem )
     { problem_ = problem; }

// Set parameters the solver manager should use to solve the linear problem.
void setParameters( const Teuchos::RCP<Teuchos::ParameterList> &params );

// Reset methods
/* Performs a reset of the solver manager specified by the ResetType.  This
 * informs the solver manager that the solver should prepare for the next call
 * to solve by resetting certain elements of the iterative solver strategy.
 */
void reset( const Belos::ResetType type ) { if ((type & Belos::Problem) && !Teuchos::
    is_null(problem_)) problem_->setProblem(); }

// Solver application methods
/* This method performs possibly repeated calls to the underlying linear
 * solver's iterate() routine until the problem has been solved (as decided by
 * the solver manager) or the solver manager decides to quit.  This method
 * calls LSQRIter::iterate(), which will return either because a specially
 * constructed status test evaluates to Belos::Passed or an std::exception is
 * thrown.  A return from LSQRIter::iterate() signifies one of the following
 * scenarios:
 * - the maximum number of iterations has been exceeded. In this scenario, the
 * current solution to the linear system will be placed in the linear problem
 * and return Belos::Unconverged.
 * - global convergence has been met. In this case, the current solution to
 * the linear system will be placed in the linear problem and the solver
 * manager will return Belos::Converged.
 * Return: Belos::ReturnType specifying:
```

```
     * - Belos::Converged: the linear problem was solved to the specification
     * required by the solver manager.
     * - Belos::Unconverged: the linear problem was not solved to the
     * specification desired by the solver manager.
     */
    Belos::ReturnType solve();

    // Overridden from Teuchos::Describable
    // Method to return description of the LSQR solver manager
    std::string description() const;

  private:

    // Linear problem.
    Teuchos::RCP<Belos::LinearProblem<ScalarType,MV,OP> > problem_;
    // Output manager.
    Teuchos::RCP<Belos::OutputManager<ScalarType> > printer_;
    Teuchos::RCP<std::ostream> outputStream_;
    // Status test.
    Teuchos::RCP<Belos::StatusTest<ScalarType,MV,OP> > sTest_;
    Teuchos::RCP<Belos::StatusTestMaxIters<ScalarType,MV,OP> > maxIterTest_;
    Teuchos::RCP<LSQRStatusTest<ScalarType,MV,OP> > convTest_;
    Teuchos::RCP<LeastErrorStatusTest<ScalarType,MV,OP> > errorTest_;
    Teuchos::RCP<Belos::StatusTestOutput<ScalarType,MV,OP> > outputTest_;
    // Current parameter list.
    Teuchos::RCP<ParameterList> params_;
    // Default solver values.
    static const ScalarType lambda_default_;
    static const MagnitudeType condlim_default_;
    static const MagnitudeType relRhsErr_default_;
    static const MagnitudeType relMatErr_default_;
    static const int maxIters_default_;
    static const int verbosity_default_;
    static const int outputStyle_default_;
    static const int outputFreq_default_;
    static const int termIterMax_default_;
    static const int windowSize_default_;
    static const std::string label_default_;
    static const Teuchos::RCP<std::ostream> outputStream_default_;
    static const Teuchos::RCP< MV > trueSolution_default_;
    // Current solver values.
    ScalarType lambda_;
    MagnitudeType condlim_, relRhsErr_, relMatErr_;
    int maxIters_, numIters_, termIterMax_, windowSize_;
    int verbosity_, outputStyle_, outputFreq_;
```

```
  Teuchos::RCP< MV > trueSolution_;
  // Timers.
  std::string label_;
  Teuchos::RCP<Teuchos::Time> timerSolve_;
  // Internal state variables.
  bool isSet_;
};

// Default solver values.
template<class ScalarType, class MV, class OP>
const ScalarType LSQRSolMgr<ScalarType,MV,OP>::lambda_default_ = 0.0;

template<class ScalarType, class MV, class OP>
const typename LSQRSolMgr<ScalarType,MV,OP>::MagnitudeType LSQRSolMgr<ScalarType,MV,OP>::
    condlim_default_ = 0.0;

template<class ScalarType, class MV, class OP>
const typename LSQRSolMgr<ScalarType,MV,OP>::MagnitudeType LSQRSolMgr<ScalarType,MV,OP>::
    relRhsErr_default_ = 0.0;

template<class ScalarType, class MV, class OP>
const typename LSQRSolMgr<ScalarType,MV,OP>::MagnitudeType LSQRSolMgr<ScalarType,MV,OP>::
    relMatErr_default_ = 0.0;

template<class ScalarType, class MV, class OP>
const Teuchos::RCP< MV > LSQRSolMgr<ScalarType,MV,OP>::trueSolution_default_ = Teuchos::
    null;

template<class ScalarType, class MV, class OP>
const int LSQRSolMgr<ScalarType,MV,OP>::windowSize_default_ = 1;

template<class ScalarType, class MV, class OP>
const int LSQRSolMgr<ScalarType,MV,OP>::maxIters_default_ = 1000;

template<class ScalarType, class MV, class OP>
const int LSQRSolMgr<ScalarType,MV,OP>::termIterMax_default_ = 1;

template<class ScalarType, class MV, class OP>
const int LSQRSolMgr<ScalarType,MV,OP>::verbosity_default_ = Belos::Errors;

template<class ScalarType, class MV, class OP>
const int LSQRSolMgr<ScalarType,MV,OP>::outputStyle_default_ = Belos::General;

template<class ScalarType, class MV, class OP>
const int LSQRSolMgr<ScalarType,MV,OP>::outputFreq_default_ = -1;
```

```
template<class ScalarType, class MV, class OP>
const std::string LSQRSolMgr<ScalarType,MV,OP>::label_default_ = "Belos";

template<class ScalarType, class MV, class OP>
const Teuchos::RCP<std::ostream> LSQRSolMgr<ScalarType,MV,OP>::outputStream_default_ =
    Teuchos::rcp(&std::cout,false);

// Empty Constructor
template<class ScalarType, class MV, class OP>
LSQRSolMgr<ScalarType,MV,OP>::LSQRSolMgr() :
  outputStream_(outputStream_default_),
  lambda_(lambda_default_),
  condlim_(condlim_default_),
  relRhsErr_(relRhsErr_default_),
  relMatErr_(relMatErr_default_),
  trueSolution_(trueSolution_default_),
  windowSize_(windowSize_default_),
  maxIters_(maxIters_default_),
  termIterMax_(termIterMax_default_),
  verbosity_(verbosity_default_),
  outputStyle_(outputStyle_default_),
  outputFreq_(outputFreq_default_),
  label_(label_default_),
  isSet_(false)
{}

// Basic Constructor
template<class ScalarType, class MV, class OP>
LSQRSolMgr<ScalarType,MV,OP>::LSQRSolMgr(const Teuchos::RCP<Belos::LinearProblem<
    ScalarType,MV,OP> > &problem, const Teuchos::RCP<Teuchos::ParameterList> &pl ) :
  problem_(problem),
  outputStream_(outputStream_default_),
  lambda_(lambda_default_),
  condlim_(condlim_default_),
  relRhsErr_(relRhsErr_default_),
  relMatErr_(relMatErr_default_),
  trueSolution_(trueSolution_default_),
  windowSize_(windowSize_default_),
  maxIters_(maxIters_default_),
  termIterMax_(termIterMax_default_),
  verbosity_(verbosity_default_),
  outputStyle_(outputStyle_default_),
  outputFreq_(outputFreq_default_),
  label_(label_default_),
```

```
  isSet_(false)
{
  TEST_FOR_EXCEPTION(problem_ == Teuchos::null, std::invalid_argument, "Problem not given
       to solver manager.");

  /* If the parameter list pointer is null, then set the current parameters to
   * the default parameter list.
   */
  if ( !is_null(pl) ) {
    setParameters( pl );
  }
}

template<class ScalarType, class MV, class OP>
void LSQRSolMgr<ScalarType,MV,OP>::setParameters( const Teuchos::RCP<Teuchos::
    ParameterList> &params )
{
  // Create the internal parameter list if ones doesn't already exist.
  if (params_ == Teuchos::null) {
    params_ = Teuchos::rcp( new Teuchos::ParameterList(*getValidParameters()) );
  }
  else {
    params->validateParameters(*getValidParameters());
  }

  // Check for damping value lambda; update in our list and status test.
  if (params->isParameter("Lambda")) {
    lambda_ = (ScalarType) params->get("Lambda", (double) lambda_default_);
    params_->set("Lambda",lambda_);
  }

  // Check for window size; update in our list and status test.
  if (params->isParameter("Window Size")) {
    windowSize_ = params->get("Window Size", windowSize_default_);
    params_->set("Window Size", windowSize_);
  }

  // Check for maximum number of iterations; update in our list and status test.
  if (params->isParameter("Maximum Iterations")) {
    maxIters_ = params->get("Maximum Iterations",maxIters_default_);
    params_->set("Maximum Iterations", maxIters_);
    if (maxIterTest_!=Teuchos::null)
      maxIterTest_->setMaxIters( maxIters_ );
  }
```

```
// Check to see if timer label changed; update in our list and solver timer.
if (params->isParameter("Timer Label")) {
  std::string tempLabel = params->get("Timer Label", label_default_);
  if (tempLabel != label_) {
    label_ = tempLabel;
    params_->set("Timer Label", label_);
    std::string solveLabel = label_ + ": LSQRSolMgr total solve time";
    timerSolve_ = Teuchos::TimeMonitor::getNewTimer(solveLabel);
  }
}

// Check for a change in verbosity level; update in our list.
if (params->isParameter("Verbosity")) {
  if (Teuchos::isParameterType<int>(*params,"Verbosity")) {
    verbosity_ = params->get("Verbosity", verbosity_default_);
  } else {
    verbosity_ = (int)Teuchos::getParameter<Belos::MsgType>(*params,"Verbosity");
  }
  params_->set("Verbosity", verbosity_);
  if (printer_ != Teuchos::null)
    printer_->setVerbosity(verbosity_);
}

// Check for a change in output style; update in our list.
if (params->isParameter("Output Style")) {
  if (Teuchos::isParameterType<int>(*params,"Output Style")) {
    outputStyle_ = params->get("Output Style", outputStyle_default_);
  } else {
    outputStyle_ = (int)Teuchos::getParameter<Belos::OutputType>(*params,"Output Style"
        );
  }
  params_->set("Output Style", outputStyle_);
  outputTest_ == Teuchos::null;
}

// Check for output stream; update in our list.
if (params->isParameter("Output Stream")) {
  outputStream_ = Teuchos::getParameter<Teuchos::RCP<std::ostream> >(*params,"Output
      Stream");
  params_->set("Output Stream", outputStream_);
  if (printer_ != Teuchos::null)
    printer_->setOStream( outputStream_ );
}

// Check for frequency level; update in our list and output status test.
```

```
if (verbosity_ & Belos::StatusTestDetails) {
  if (params->isParameter("Output Frequency")) {
    outputFreq_ = params->get("Output Frequency", outputFreq_default_);
  }
  params_->set("Output Frequency", outputFreq_);
  if (outputTest_ != Teuchos::null)
    outputTest_->setOutputFrequency( outputFreq_ );
}

// Create output manager if we need to.
if (printer_ == Teuchos::null) {
  printer_ = Teuchos::rcp( new Belos::OutputManager<ScalarType>(verbosity_,
      outputStream_) );
}

// Convergence
typedef Belos::StatusTestCombo<ScalarType,MV,OP>  StatusTestCombo_t;

// Check for convergence tolerance; update in our list and residual tests.
if (params->isParameter("Condition Limit")) {
  condlim_ = params->get("Condition Limit",condlim_default_);
  params_->set("Condition Limit", condlim_);
  if (convTest_ != Teuchos::null)
    convTest_->setCondLim( condlim_ );
}

// Check for number of consecutive passed iterations; update in our list and
// residual tests.
if (params->isParameter("Term Iter Max")) {
  termIterMax_ = params->get("Term Iter Max", termIterMax_default_);
  params_->set("Term Iter Max", termIterMax_);
  if (convTest_ != Teuchos::null)
    convTest_->setTermIterMax( termIterMax_ );
}

// Check for true solution; update in our list and status test.
if (params->isParameter("True Solution")) {
  trueSolution_ = params->get< Teuchos::RCP< MV > >("True Solution",
      trueSolution_default_);
  params_->set("True Solution",trueSolution_);
}

// Check for relative RHS error; update in our list and residual tests.
if (params->isParameter("Rel RHS Err")) {
  relRhsErr_ = params->get("Rel RHS Err",relRhsErr_default_);
```

```
  params_->set("Rel RHS Err", relRhsErr_);
  if (convTest_ != Teuchos::null)
    convTest_->setRelRhsErr( relRhsErr_ );
}

// Check for relative matrix error; update in our list and residual tests.
if (params->isParameter("Rel Mat Err")) {
  relMatErr_ = (MagnitudeType) params->get("Rel Mat Err", (double) relMatErr_default_ )
      ;
  params_->set("Rel Mat Err", relMatErr_);
  if (convTest_ != Teuchos::null)
    convTest_->setRelMatErr( relMatErr_ );
}

// Create status tests if we need to.
// Basic test checks maximum iterations.
if (maxIterTest_ == Teuchos::null)
  maxIterTest_ = Teuchos::rcp( new Belos::StatusTestMaxIters<ScalarType,MV,OP>(
      maxIters_ ) );

// Least error test if needed.
if (trueSolution_ != Teuchos::null)
  errorTest_ = Teuchos::rcp( new LeastErrorStatusTest<ScalarType,MV,OP>(trueSolution_,
      windowSize_) );

// Status test specific to LSQR.
if (convTest_ == Teuchos::null)
  convTest_ = Teuchos::rcp( new LSQRStatusTest<ScalarType,MV,OP>(condlim_, termIterMax_
      , relRhsErr_, relMatErr_) );

if (sTest_ == Teuchos::null)
  sTest_ = Teuchos::rcp( new StatusTestCombo_t( StatusTestCombo_t::OR, maxIterTest_,
      convTest_ ) );

if (errorTest_ != Teuchos::null)
  // add error test to the status test combo
  ((Teuchos::rcp_dynamic_cast< StatusTestCombo_t >) (sTest_))->addStatusTest(errorTest_
      );

if (outputTest_ == Teuchos::null) {
  // Create the status test output class.
  // This class manages and formats the output from the status test.
  Belos::StatusTestOutputFactory<ScalarType,MV,OP> stoFactory( outputStyle_ );
  outputTest_ = stoFactory.create( printer_, sTest_, outputFreq_, Belos::Passed+Belos::
      Failed+Belos::Undefined );
```

```
    // Set the solver string for the output test
    std::string solverDesc = " LSQR ";
    outputTest_->setSolverDesc( solverDesc );
  }

  // Create the timer if we need to.
  if (timerSolve_ == Teuchos::null) {
    std::string solveLabel = label_ + ": LSQRSolMgr total solve time";
    timerSolve_ = Teuchos::TimeMonitor::getNewTimer(solveLabel);
  }

  // Inform the solver manager that the current parameters were set.
  isSet_ = true;
}

template<class ScalarType, class MV, class OP>
Teuchos::RCP<const Teuchos::ParameterList>
LSQRSolMgr<ScalarType,MV,OP>::getValidParameters() const
{
  static Teuchos::RCP<const Teuchos::ParameterList> validPL;
  // Set all the valid parameters and their default values.
  if(is_null(validPL)) {
    Teuchos::RCP<Teuchos::ParameterList> pl = Teuchos::parameterList();
    pl->set("Lambda", lambda_default_, "The default damping parameter.");
    pl->set("Condition Limit", condlim_default_,
      "The upper limit on the estimate of the condition number of Abar");
    pl->set("Maximum Iterations", maxIters_default_,
      "The maximum number of block iterations allowed for each\n"
      "set of RHS solved.");
    pl->set("Term Iter Max", termIterMax_default_,
      "The number of consecutive successful convergent iterations");
    pl->set("Window Size", windowSize_default_,
      "The window size for the least error test");
    pl->set("True Solution", trueSolution_default_,
      "The true solution for the least error test");
    pl->set("Verbosity", verbosity_default_,
      "What type(s) of solver information should be outputted\n"
      "to the output stream.");
    pl->set("Output Style", outputStyle_default_,
      "What style is used for the solver information outputted\n"
      "to the output stream.");
    pl->set("Output Frequency", outputFreq_default_,
      "How often convergence information should be outputted\n"
      "to the output stream.");
    pl->set("Output Stream", outputStream_default_,
```

```
      "A reference-counted pointer to the output stream where all\n"
      "solver output is sent.");
    pl->set("Rel RHS Err", relRhsErr_default_,
      "An estimate in the error in the data defining the\n"
      "right hand side.");
    pl->set("Rel Mat Err", relMatErr_default_,
      "An estimate in the error in the data defining the matrix.");
    pl->set("Timer Label", label_default_,
      "The string to use as a prefix for the timer labels.");
    validPL = pl;
  }
  return validPL;
}

// Solve method
template<class ScalarType, class MV, class OP>
Belos::ReturnType LSQRSolMgr<ScalarType,MV,OP>::solve() {
  // Set the current parameters if they were not set before.
  // NOTE: This may occur if the user generated the solver manager with the
  // default constructor and then didn't set any parameters using setParameters.
  if (!isSet_) {
    setParameters(Teuchos::parameterList(*getValidParameters()));
  }

  TEST_FOR_EXCEPTION(!problem_->isProblemSet(),LSQRSolMgrLinearProblemFailure, "
      LSQRSolMgr::solve(): Linear problem is not ready, setProblem() has not been called.
      ");
  TEST_FOR_EXCEPTION(MVT::GetNumberVecs( *(problem_->getRHS()) ) != 1,
      LSQRSolMgrBlockSizeFailure, "LSQRSolMgr::solve(): Incorrect number of RHS vectors,
      should be exactly 1.");

  // Inform the linear problem of the current linear system to solve.
  std::vector<int> currRHSIdx(1, 0);
  problem_->setLSIndex(currRHSIdx);
  Teuchos::ParameterList plist;
  plist.set("Lambda", lambda_);
  outputTest_->reset();
  // Assume convergence is achieved, then let any failed convergence set this
  // to false.
  bool isConverged = true;

  // LSQR should be given an all-zero initial guess, so ensure that's the case.
  Teuchos::RCP<MV> x = problem_->getLHS();
  x->putScalar((ScalarType) 0.0);
  problem_->updateSolution(x);
```

```
// LSQR solver
Teuchos::RCP<LSQRIter<ScalarType,MV,OP> > lsqr_iter = Teuchos::rcp( new LSQRIter<
    ScalarType,MV,OP>(problem_, printer_, outputTest_, plist) );
Teuchos::TimeMonitor slvtimer(*timerSolve_);

// Reset the number of iterations.
lsqr_iter->resetNumIters();
// Reset the number of calls that the status test output knows about.
outputTest_->resetNumCalls();
// Set the new state and initialize the solver.
LSQRIterationState<ScalarType,MV> newstate;
lsqr_iter->initializeLSQR(newstate);

// Tell lsqr_iter to iterate
try {
  lsqr_iter->iterate();
  // Check convergence first
  if ( convTest_->getStatus() == Belos::Passed ) {
  }
  else if ( (errorTest_ != Teuchos::null) && errorTest_->getStatus() == Belos::Passed )
       {
  }
  else if ( maxIterTest_->getStatus() == Belos::Passed ) {
    // We don't have convergence
    isConverged = false;
  }
  // We returned from iterate(), but none of our status tests Passed.
  // Something is wrong, and it is probably our fault.
  else {
    TEST_FOR_EXCEPTION(true,std::logic_error, "LSQRSolMgr::solve(): Invalid return from
         LSQRIteration::iterate().");
  }
}
catch (const std::exception &e) {
  printer_->stream(Belos::Errors) << "Error! Caught std::exception in LSQRIter::iterate
      () at iteration "  << lsqr_iter->getNumIters() << std::endl << e.what() << std::
      endl;
  throw;
}

// Inform the linear problem that we are finished with this linear system.
problem_->setCurrLS();
// Print final summary and timing information.
sTest_->print( printer_->stream(Belos::FinalSummary) );
```

```
  Teuchos::TimeMonitor::summarize( printer_->stream(Belos::TimingDetails) );
  // Get iteration information for this solve.
  numIters_ = lsqr_iter->getNumIters();

  if (!isConverged) {
    return Belos::Unconverged; // return from LSQRSolMgr::solve()
  }
  return Belos::Converged; // return from LSQRSolMgr::solve()
}

// This method requires the solver manager to return a std::string that
// describes itself.
template<class ScalarType, class MV, class OP>
std::string LSQRSolMgr<ScalarType,MV,OP>::description() const
{
  std::ostringstream oss;
  oss << "LSQRSolMgr<...,"<<Teuchos::ScalarTraits<ScalarType>::name()<<">";
  return oss.str();
}

#endif /* LSQR_SOLMGR_HPP */
```

### A.1.2 LSQRIter.hpp Code

```
#ifndef LSQR_ITER_HPP
#define LSQR_ITER_HPP

/* LSQRIter.hpp
 * Concrete class for performing the LSQR iteration.
 */

#include "BelosConfigDefs.hpp"
#include "BelosTypes.hpp"
#include "BelosIteration.hpp"
#include "BelosLinearProblem.hpp"
#include "BelosOutputManager.hpp"
#include "BelosStatusTest.hpp"
#include "BelosOperatorTraits.hpp"
#include "BelosMultiVecTraits.hpp"
#include "Teuchos_SerialDenseMatrix.hpp"
#include "Teuchos_SerialDenseVector.hpp"
#include "Teuchos_ScalarTraits.hpp"
#include "Teuchos_ParameterList.hpp"
#include "Teuchos_TimeMonitor.hpp"

  // LSQRIteration Structures
```

```
/* Structure to contain pointers to LSQRIteration state variables.
 * This struct is utilized by initialize() and getState().
 */
template <class ScalarType, class MV>
struct LSQRIterationState {
  /*! \brief Bidiagonalization vector. */
  Teuchos::RCP<const MV> U;
  /*! \brief Bidiagonalization vector. */
  Teuchos::RCP<const MV> V;
  /*! \brief The search direction vector. */
  Teuchos::RCP<const MV> W;
  /*! \brief The A^T*U vector. */
  Teuchos::RCP<const MV> Q;
  /*! \brief The M*V vector. */
  Teuchos::RCP<const MV> P;
  /*! \brief The Y vector. */
  Teuchos::RCP<const MV> Y;
  /*! \brief The damping value. */
  ScalarType lambda;
  /*! \brief The current residual norm. */
  ScalarType resid_norm;
  /*! \brief An approximation to the Frobenius norm of A. */
  ScalarType frob_mat_norm;
  /*! \brief An approximation to the condition number of A. */
  ScalarType mat_cond_num;
  /*! \brief An estimate of the norm of A^T*resid. */
  ScalarType mat_resid_norm;
  /*! \brief An estimate of the norm of the solution. */
  ScalarType sol_norm;
  /*! \brief The norm of the RHS vector b. */
  ScalarType bnorm;

  LSQRIterationState() : U(Teuchos::null), V(Teuchos::null),
       W(Teuchos::null), Q(Teuchos::null),
       P(Teuchos::null), Y(Teuchos::null),
       lambda(0.0),
       resid_norm(0.0), frob_mat_norm(0.0),
       mat_cond_num(0.0), mat_resid_norm(0.0),
       sol_norm(0.0), bnorm(0.0)
  {}
};

// LSQRIteration Exceptions
/* LSQRIterationInitFailure is thrown when the LSQRIteration object is unable
 * to generate an initial iterate in the initialize() routine.  This
```

```
   * std::exception is thrown from the initialize() method, which is called by
   * the user or from the iterate() method if isInitialized() == false.  In the
   * case that this std::exception is thrown, isInitialized() will be false and
   * the user will need to provide a new initial iterate to the iteration.
   */
class LSQRIterationInitFailure : public Belos::BelosError {public:
      LSQRIterationInitFailure(const std::string& what_arg) : Belos::BelosError(what_arg)
    {}};

  /* LSQRIterateFailure is thrown when the LSQRIteration object is unable to
   * compute the next iterate in the iterate() routine.  This std::exception is
   * thrown from the iterate() method.
   */
class LSQRIterateFailure : public Belos::BelosError {public:
      LSQRIterateFailure(const std::string& what_arg) : Belos::BelosError(what_arg)
    {}};

template<class ScalarType, class MV, class OP>
class LSQRIter : virtual public Belos::Iteration<ScalarType,MV,OP> {

  public:
  typedef Belos::MultiVecTraits<ScalarType,MV> MVT;
  typedef Belos::OperatorTraits<ScalarType,MV,OP> OPT;
  typedef Teuchos::ScalarTraits<ScalarType> SCT;
  typedef typename SCT::magnitudeType MagnitudeType;

  // Constructors/Destructor
  /* LSQRIter constructor with linear problem, solver utilities, and parameter
   * list of solver options.  This constructor takes pointers required by the
   * linear solver iteration, in addition to a parameter list of options for the
   * linear solver.
   */
  LSQRIter( const Teuchos::RCP<Belos::LinearProblem<ScalarType,MV,OP> > &problem, const
      Teuchos::RCP<Belos::OutputManager<ScalarType> > &printer, const Teuchos::RCP<Belos
      ::StatusTest<ScalarType,MV,OP> > &tester, Teuchos::ParameterList &params );

  // Destructor.
  virtual ~LSQRIter() {};

  // Solver methods
  /* This method performs LSQR iterations until the status test indicates the
   * need to stop or an error occurs (in which case, an std::exception is
   * thrown).  This function will first determine whether the solver is
   * initialized; if not, it will call initialize() using default arguments.
   * After initialization, the solver performs LSQR iterations until the status
```

```
 * test evaluates as Belos::Passed, at which point the method returns to the
 * caller.  The status test is queried at the beginning of the iteration.
 */
void iterate();

/* Initialize the solver to an iterate, providing a complete state.  The
 * LSQRIter contains a certain amount of state, consisting of two
 * bidiagonalization vectors, a descent direction, several other vectors, a
 * damping value, and various estimates of errors and the like.  For any
 * pointer in newstate which directly points to the multivectors in the
 * solver, the data is not copied.
 */
void initializeLSQR(LSQRIterationState<ScalarType,MV> newstate);

// Initialize the solver.
void initialize()
{
  LSQRIterationState<ScalarType,MV> empty;
  initializeLSQR(empty);
}

/* Get the current state of the linear solver.  The data is only valid if
 * isInitialized() == true.
 * Return: A LSQRIterationState object containing const pointers to the
 * current solver state.
 */
LSQRIterationState<ScalarType,MV> getState() const {
  LSQRIterationState<ScalarType,MV> state;
  state.U = U_;
  state.V = V_;
  state.W = W_;
  state.Q = Q_;
  state.P = P_;
  state.Y = Y_;
  state.lambda = lambda_;
  state.resid_norm = resid_norm_;
  state.frob_mat_norm = frob_mat_norm_;
  state.mat_cond_num = mat_cond_num_;
  state.mat_resid_norm = mat_resid_norm_;
  state.sol_norm = sol_norm_;
  state.bnorm = bnorm_;
  return state;
}

// Status methods
```

```
// Get the current iteration count.
int getNumIters() const { return iter_; }

// Reset the iteration count.
void resetNumIters( int iter = 0 ) { iter_ = iter; }

// Get the norms of the residuals native to the solver.
// This method returns a null pointer because residuals aren't used with LSQR.
Teuchos::RCP<const MV> getNativeResiduals( std::vector<MagnitudeType> *norms ) const {
    return Teuchos::null; }

// Get the current update to the linear system.
// This method returns a null pointer because the linear problem is current.
Teuchos::RCP<MV> getCurrentUpdate() const { return Teuchos::null; }

// Accessor methods
// Get a constant reference to the linear problem.
const Belos::LinearProblem<ScalarType,MV,OP>& getProblem() const { return *lp_; }

// Get blocksize to be used by iterative solver in solving this linear problem
int getBlockSize() const { return 1; }

// Set blocksize to be used by iterative solver in solving this linear problem
void setBlockSize(int blockSize) {
  TEST_FOR_EXCEPTION(blockSize!=1,std::invalid_argument, "LSQRIter::setBlockSize():
      Cannot use a block size that is not one.");
}

// States whether the solver has been initialized or not.
bool isInitialized() { return initialized_; }

private:

// Method for initalizing the state storage needed by LSQR.
void setStateSize();

// Classes inputed through constructor that define the linear problem to be
// solved.
const Teuchos::RCP<Belos::LinearProblem<ScalarType,MV,OP> >    lp_;
const Teuchos::RCP<Belos::OutputManager<ScalarType> >          om_;
const Teuchos::RCP<Belos::StatusTest<ScalarType,MV,OP> >       stest_;
// Current solver state.
/* initialized_ specifies that the basis vectors have been initialized and the
 * iterate() routine is capable of running; _initialize is controlled  by the
 * initialize() member method.  For the implications of the state of
```

```
    * initialized_, please see documentation for initialize().
    */
  bool initialized_;
  /* stateStorageInitialized_ specifies that the state storage has been
   * initialized.  This initialization may be postponed if the linear problem
   * was generated without the right-hand side or solution vectors.
   */
  bool stateStorageInitialized_;
  // Current number of iterations performed.
  int iter_;
  // Bidiagonalization vector
  Teuchos::RCP<MV> U_;
  // Bidiagonalization vector
  Teuchos::RCP<MV> V_;
  // Direction vector
  Teuchos::RCP<MV> W_;
  // A^T*U vector
  Teuchos::RCP<MV> Q_;
  // M*V vector
  Teuchos::RCP<MV> P_;
  // Y vector
  Teuchos::RCP<MV> Y_;
  // Damping value
  ScalarType lambda_;
  // Residual norm estimate
  ScalarType resid_norm_;
  // Frobenius norm estimate
  ScalarType frob_mat_norm_;
  // Condition number estimate
  ScalarType mat_cond_num_;
  // A^T*resid norm estimate
  ScalarType mat_resid_norm_;
  // Solution norm estimate
  ScalarType sol_norm_;
  // RHS norm
  ScalarType bnorm_;
};

  // Constructor.
  template<class ScalarType, class MV, class OP>
  LSQRIter<ScalarType,MV,OP>::LSQRIter(const Teuchos::RCP<Belos::LinearProblem<ScalarType
      ,MV,OP> > &problem, const Teuchos::RCP<Belos::OutputManager<ScalarType> > &printer,
       const Teuchos::RCP<Belos::StatusTest<ScalarType,MV,OP> > &tester, Teuchos::
      ParameterList &params ):
    lp_(problem),
```

```
  om_(printer),
  stest_(tester),
  initialized_(false),
  stateStorageInitialized_(false),
  iter_(0),
  lambda_(params.get("Lambda", (ScalarType) 0.0))
{
}

// Setup the state storage.
template <class ScalarType, class MV, class OP>
void LSQRIter<ScalarType,MV,OP>::setStateSize ()
{
  if (!stateStorageInitialized_) {
    // Check if there is any multivector to clone from.
    Teuchos::RCP<const MV> lhsMV = lp_->getLHS();
    Teuchos::RCP<const MV> rhsMV = lp_->getRHS();
    if (lhsMV == Teuchos::null || rhsMV == Teuchos::null) {
stateStorageInitialized_ = false;
return;
    }
    else {
// Initialize the state storage.  If the subspace has not been
// initialized before, generate it using the LHS and RHS from lp_.
if (U_ == Teuchos::null) {
  TEST_FOR_EXCEPTION(rhsMV == Teuchos::null, std::invalid_argument, "LSQRIter::
      setStateSize(): linear problem does not specify right hand multivector to clone
      from.");
  TEST_FOR_EXCEPTION(lhsMV == Teuchos::null, std::invalid_argument, "LSQRIter::
      setStateSize(): linear problem does not specify left hand multivector to clone
      from.");

  U_ = MVT::Clone( *rhsMV, 1 );
  V_ = MVT::Clone( *lhsMV, 1 );
  W_ = MVT::Clone( *lhsMV, 1 );
  if ( lp_->isRightPrec() ) {
    Q_ = MVT::Clone( *lhsMV, 1 );
    P_ = MVT::Clone( *lhsMV, 1 );
    Y_ = MVT::Clone( *lhsMV, 1 );
  }
}
// State storage has now been initialized.
stateStorageInitialized_ = true;
    }
  }
```

```
}

// Initialize this iteration object
template <class ScalarType, class MV, class OP>
void LSQRIter<ScalarType,MV,OP>::initializeLSQR(LSQRIterationState<ScalarType,MV>
    newstate)
{
  // Initialize the state storage if it isn't already.
  if (!stateStorageInitialized_)
    setStateSize();
  TEST_FOR_EXCEPTION(!stateStorageInitialized_,std::invalid_argument, "LSQRIter::
      initialize(): Cannot initialize state storage!");
  std::string errstr("LSQRIter::initialize(): Specified multivectors must have a
      consistent length and width.");

  // Create convenience variables for zero and one.
  const ScalarType one = Teuchos::ScalarTraits<ScalarType>::one();
  const ScalarType zero = Teuchos::ScalarTraits<ScalarType>::zero();

  // Compute initial bidiagonalization vectors and search direction.
  Teuchos::RCP<const MV> lhsMV = lp_->getLHS();
  Teuchos::RCP<const MV> rhsMV = lp_->getRHS();

  lp_->applyOp( *lhsMV, *U_);
  MVT::MvAddMv( one, *rhsMV, -one, *U_, *U_);
  if ( lp_->isRightPrec() ) {
    lp_->getOperator()->apply( *U_, *Q_, Teuchos::TRANS );
    lp_->getRightPrec()->apply( *Q_, *V_, Teuchos::TRANS );
  } else {
    lp_->getOperator()->apply( *U_, *V_, Teuchos::TRANS);
  }
  MVT::MvAddMv( one, *V_, zero, *V_, *W_);
  frob_mat_norm_ = zero;
  mat_cond_num_ = zero;
  sol_norm_ = zero;

  // The solver is initialized.
  initialized_ = true;
}

// Iterate until the status test informs us we should stop.
template <class ScalarType, class MV, class OP>
void LSQRIter<ScalarType,MV,OP>::iterate()
{
  // Allocate/initialize data structures
```

```
if (initialized_ == false) {
  initialize();
}

// Create convenience variables for zero and one.
const ScalarType one = Teuchos::ScalarTraits<ScalarType>::one();
const MagnitudeType zero = Teuchos::ScalarTraits<MagnitudeType>::zero();

// Allocate memory for scalars.
std::vector<typename Teuchos::ScalarTraits<ScalarType>::magnitudeType> alpha(1);
std::vector<typename Teuchos::ScalarTraits<ScalarType>::magnitudeType> beta(1);
std::vector<typename Teuchos::ScalarTraits<ScalarType>::magnitudeType> wnorm2(1);
ScalarType rhobar, phibar, cs1, phi, rho, cs, sn, theta, xxnorm = zero, common;
ScalarType zetabar, sn1, psi, res = zero, bbnorm = zero, ddnorm = zero, gamma, tau;
ScalarType cs2 = -one, sn2 = zero, gammabar, zeta = zero, delta;

// Allocate memory for working vectors.
// Operator applied to bidiagonalization vector
Teuchos::RCP<MV> AV;
// Transpose of operator applied to bidiagonalization vector
Teuchos::RCP<MV> AtU;
AV = MVT::Clone( *U_, 1);
AtU = MVT::Clone( *V_, 1);

// Get the current solution vector.
Teuchos::RCP<MV> cur_soln_vec = lp_->getCurrLHSVec();

// Check that the current solution vector only has one column.
TEST_FOR_EXCEPTION( MVT::GetNumberVecs(*cur_soln_vec) != 1, LSQRIterateFailure, "
    LSQRIter::iterate(): current linear system has more than one vector!" );

// Compute alpha and beta and scale bidiagonalization vectors
MVT::MvNorm( *U_, beta );
MVT::MvScale( *U_, one / beta[0] );
if ( lp_->isRightPrec() ) {
  MVT::MvScale( *Q_, one / beta[0] );
  lp_->getRightPrec()->apply( *Q_, *V_, Teuchos::TRANS );
  MVT::MvNorm( *V_, alpha );
  MVT::MvScale( *V_, one / alpha[0] );
} else {
  MVT::MvScale( *V_, one / beta[0] );
  MVT::MvNorm( *V_, alpha );
  MVT::MvScale( *V_, one / alpha[0] );
}
MVT::MvScale( *W_, one / (beta[0] * alpha[0]) );
```

```
rhobar = alpha[0];
phibar = beta[0];

resid_norm_ = beta[0];
mat_resid_norm_ = alpha[0] * beta[0];
bnorm_ = beta[0];

// Iterate until the status test tells us to stop.
while (stest_->checkStatus(this) != Belos::Passed) {
  // Increment the iteration.
  iter_++;

  // Perform the next step of the bidiagonalization.
  if (lp_->isRightPrec() ) {
    lp_->applyRightPrec( *V_, *P_ );
    lp_->applyOp( *P_, *AV);
  } else {
    lp_->applyOp( *V_, *AV);
  }
  MVT::MvAddMv( one, *AV, -alpha[0], *U_, *U_ );
  MVT::MvNorm( *U_, beta);
  // Check that beta is a positive number.
  TEST_FOR_EXCEPTION( SCT::real(beta[0]) <= zero, LSQRIterateFailure, "LSQRIter::
      iterate(): non-positive value for beta encountered!");
  bbnorm += alpha[0]*alpha[0] + beta[0]*beta[0] + lambda_*lambda_;
  MVT::MvScale( *U_, one / beta[0] );

  if (lp_->isRightPrec() ) {
    lp_->getOperator()->apply( *U_, *Q_, Teuchos::TRANS );
    lp_->getRightPrec()->apply( *Q_, *AtU, Teuchos::TRANS );
  } else {
    lp_->getOperator()->apply( *U_, *AtU, Teuchos::TRANS);
  }
  MVT::MvAddMv( one, *AtU, -beta[0], *V_, *V_ );
  MVT::MvNorm( *V_, alpha );
  // Check that alpha is a positive number.
  TEST_FOR_EXCEPTION( SCT::real(alpha[0]) <= zero, LSQRIterateFailure, "LSQRIter::
      iterate(): non-positive value for alpha encountered!");
  MVT::MvScale( *V_, one / alpha[0] );

  // Use a plane rotation to eliminate the damping parameter.  This alters
  // the diagonal (rhobar) of the lower-bidiagonal matrix.
  common = Teuchos::ScalarTraits< ScalarType >::squareroot(rhobar*rhobar + lambda_*
      lambda_);
```

```
cs1 = rhobar / common;
sn1 = lambda_ / common;
psi = sn1 * phibar;
phibar = cs1 * phibar;

// Use a plane rotation to eliminate the subdiagonal element (beta) of the
// lower-bidiagonal matrix, giving an upper-bidiagonal matrix.
rho = Teuchos::ScalarTraits< ScalarType >::squareroot(rhobar*rhobar + lambda_*
    lambda_ + beta[0]*beta[0]);
cs = common / rho;
sn = beta[0] / rho;
theta = sn * alpha[0];
rhobar = -cs * alpha[0];
phi = cs * phibar;
phibar = sn * phibar;
tau = sn * phi;

delta = sn2 * rho;
gammabar = -cs2 * rho;
zetabar = (phi - delta*zeta) / gammabar;
sol_norm_ = Teuchos::ScalarTraits< ScalarType >::squareroot(xxnorm + zetabar*
    zetabar);
gamma = Teuchos::ScalarTraits< ScalarType >::squareroot(gammabar*gammabar + theta*
    theta);
cs2 = gammabar / gamma;
sn2 = theta / gamma;
zeta = (phi - delta*zeta) / gamma;
xxnorm += zeta*zeta;

// Update the solution vector and search direction vector.
if (lp_->isRightPrec() ) {
  MVT::MvAddMv( phi / rho, *W_, one, *Y_, *Y_ );
  lp_->applyRightPrec( *Y_, *cur_soln_vec);
} else {
  MVT::MvAddMv( phi / rho, *W_, one, *cur_soln_vec, *cur_soln_vec);
}
lp_->updateSolution();
MVT::MvNorm( *W_, wnorm2 );
ddnorm += (one / rho)*(one / rho) * wnorm2[0]*wnorm2[0];
MVT::MvAddMv( one, *V_, -theta / rho, *W_, *W_ );

frob_mat_norm_ = Teuchos::ScalarTraits< ScalarType >::squareroot(bbnorm);
mat_cond_num_ = frob_mat_norm_ * Teuchos::ScalarTraits< ScalarType >::squareroot(
    ddnorm);
res+= psi*psi;
```

```
      resid_norm_ = Teuchos::ScalarTraits< ScalarType >::squareroot(phibar*phibar + res);
      mat_resid_norm_ = alpha[0] * Teuchos::ScalarTraits< ScalarType >::magnitude(tau);
    } // end while (sTest_->checkStatus(this) != Passed)
  }

#endif /* LSQR_ITER_HPP */
```

### A.1.3 LSQRStatusTest.hpp Code

```
#ifndef LSQR_STATUS_TEST_HPP
#define LSQR_STATUS_TEST_HPP

/* LSQRStatusTest.hpp
 * Belos::StatusTest class for specifying convergence of LSQR.
 */

#include "BelosStatusTest.hpp"

template <class ScalarType, class MV, class OP>
class LSQRStatusTest: public Belos::StatusTest<ScalarType,MV,OP> {

public:

  typedef Teuchos::ScalarTraits<ScalarType> SCT;
  typedef typename SCT::magnitudeType MagnitudeType;
  typedef Belos::MultiVecTraits<ScalarType,MV>  MVT;

  // Constructor/Destructor.
  /* The constructor takes four optional arguments, specifying the upper limit
   * of the apparent condition number of Abar, the number of successful
   * convergent iterations, an estimate of the relative error in the data
   * defining b, and an estimate of the relative error in the data defining A.
   * The value of 0 will be used as default for all of these arguments, except
   * for the number of iterations will have a default of 1.
   */
  LSQRStatusTest( MagnitudeType cond_lim = 0.0, int term_iter_max = 1, MagnitudeType
      rel_rhs_err = 0.0, MagnitudeType rel_mat_err = 0.0 );

  // Destructor
  virtual ~LSQRStatusTest();

  // Parameter definition methods.
  /* Set the value of the tolerance.  We allow the limit of the condition number
   * of Abar to be reset for cases where, in the process of testing convergence,
   * we find that the initial limit was too tight or too lax.
  */
```

```
int setCondLim(MagnitudeType cond_lim) {
  cond_lim_ = cond_lim;
  cond_tol_ = (cond_lim > 0) ? (Teuchos::ScalarTraits< MagnitudeType >::one() /
      cond_lim) : Teuchos::ScalarTraits< MagnitudeType >::eps();
  return(0);}

int setTermIterMax(int term_iter_max) {
  term_iter_max_ = term_iter_max;
  if (term_iter_max_ < 1)
    term_iter_max_ = 1;
  return(0);}

int setRelRhsErr(MagnitudeType rel_rhs_err) {
  rel_rhs_err_ = rel_rhs_err;
  return(0);}

int setRelMatErr(MagnitudeType rel_mat_err) {
  rel_mat_err_ = rel_mat_err;
  return(0);}

// Status methods
/* This method checks to see if the convergence criteria are met using the
 * current information from the iterative solver, returning one of
 * Belos::Unconverged, Belos::Converged, or Belos::Failed.
 */
Belos::StatusType checkStatus(Belos::Iteration<ScalarType,MV,OP> *iSolver );

// Return the result of the most recent CheckStatus call.
Belos::StatusType getStatus() const {return(status_);}

// Reset the status test to the initial internal state.
void reset();

// Print methods
// Output formatted description of stopping test to output stream.
void print(std::ostream& os, int indent = 0) const;

// Print message for each status specific to this stopping test.
void printStatus(std::ostream& os, Belos::StatusType type) const;

// Methods to access data members.
// Return value of upper limit of condition number of Abar set in constructor.
MagnitudeType getCondLim() const {return(cond_lim_);};

// Return number of successful convergent iterations required.
```

```
  int getTermIterMax() const {return(term_iter_max_);};

  // Return value of the estimate of the relative error in the data defining b.
  MagnitudeType getRelRhsErr() const {return(rel_rhs_err_);};

  // Return value of the estimate of the relative error in the data defining A.
  MagnitudeType getMatErr() const {return(rel_mat_err_);};

  // Call to setup initialization.
  Belos::StatusType firstCallCheckStatusSetup(Belos::Iteration<ScalarType,MV,OP>* iSolver
      );

  // Overridden from Teuchos::Describable
  // Method to return description of the LSQR status test.
  std::string description() const
  {
    std::ostringstream oss;
    oss << "LSQRStatusTest<>: [ limit of condition number = " << cond_lim_ << " ]";
    return oss.str();
  }

private:

  // Upper limit of condition number of Abar
  MagnitudeType cond_lim_;
  // Error in data defining b
  MagnitudeType rel_rhs_err_;
  // Error in data defining A
  MagnitudeType rel_mat_err_;
  // Tolerance used to determine convergence: the reciprocal of cond_lim_ or, if
  // that is zero, machine epsilon
  MagnitudeType cond_tol_;
  // Status
  Belos::StatusType status_;
  // Is this the first time CheckStatus is called?
  bool firstcallCheckStatus_;
  // How many iterations in a row a test for convergence has passed.
  int term_iter_;
  // How many iterations in a row a passing test for convergence is required.
  int term_iter_max_;
};

template <class ScalarType, class MV, class OP>
LSQRStatusTest<ScalarType,MV,OP>::LSQRStatusTest( MagnitudeType cond_lim /* = 0 */, int
    term_iter_max /* = 1 */, MagnitudeType rel_rhs_err /* = 0 */, MagnitudeType
```

```
    rel_mat_err /* = 0 */)
  : cond_lim_(cond_lim),
    term_iter_max_(term_iter_max),
    rel_rhs_err_(rel_rhs_err),
    rel_mat_err_(rel_mat_err),
    status_(Belos::Undefined),
    firstcallCheckStatus_(true)
{}

template <class ScalarType, class MV, class OP>
LSQRStatusTest<ScalarType,MV,OP>::~LSQRStatusTest()
{}

template <class ScalarType, class MV, class OP>
void LSQRStatusTest<ScalarType,MV,OP>::reset()
{
  status_ = Belos::Undefined;
  firstcallCheckStatus_ = true;
}

template <class ScalarType, class MV, class OP>
Belos::StatusType LSQRStatusTest<ScalarType,MV,OP>::firstCallCheckStatusSetup(Belos::
    Iteration<ScalarType,MV,OP>* iSolver)
{
  if (firstcallCheckStatus_) {
      firstcallCheckStatus_ = false;
      term_iter_ = -1;
      if (cond_lim_ > 0)
        cond_tol_ = 1.0 / cond_lim_;
      else
        cond_tol_ = Teuchos::ScalarTraits< MagnitudeType >::eps();
    }
    return Belos::Undefined;
}

template <class ScalarType, class MV, class OP>
Belos::StatusType LSQRStatusTest<ScalarType,MV,OP>::checkStatus( Belos::Iteration<
    ScalarType,MV,OP>* iSolver)
{
  if (firstcallCheckStatus_) {
    Belos::StatusType status = firstCallCheckStatusSetup(iSolver);
    if(status==Belos::Failed) {
      status_ = Belos::Failed;
      return(status_);
    }
```

```
  }

  const ScalarType one = Teuchos::ScalarTraits<ScalarType>::one();
  const MagnitudeType zero = Teuchos::ScalarTraits<MagnitudeType>::zero();

  bool flag = false;
  LSQRIter<ScalarType,MV,OP>* solver = dynamic_cast< LSQRIter<ScalarType,MV,OP>* > (
      iSolver);
  LSQRIterationState< ScalarType, MV > state = solver->getState();

  // Compute the three stopping criteria
  ScalarType stop_crit_1 = state.resid_norm / state.bnorm;
  ScalarType stop_crit_2 = (state.resid_norm > zero) ? state.mat_resid_norm / (state.
      frob_mat_norm * state.resid_norm) : zero;
  ScalarType stop_crit_3 = one / state.mat_cond_num;
  ScalarType resid_tol = rel_rhs_err_ + rel_mat_err_ * state.mat_resid_norm * state.
      sol_norm / state.bnorm;
  ScalarType resid_tol_mach = Teuchos::ScalarTraits< MagnitudeType >::eps() + Teuchos::
      ScalarTraits< MagnitudeType >::eps() * state.mat_resid_norm * state.sol_norm /
      state.bnorm;

  // Check if any stopping criteria have been met
  if (stop_crit_1 <= resid_tol || stop_crit_2 <= rel_mat_err_ || stop_crit_3 <= cond_tol_
       || stop_crit_1 <= resid_tol_mach || stop_crit_2 <= Teuchos::ScalarTraits<
      MagnitudeType >::eps() || stop_crit_3 <= Teuchos::ScalarTraits< MagnitudeType >::
      eps()) {
    flag = true;
  }

  // Check if stopping criteria have been met for enough consecutive iterations.
  if (!flag) {
    term_iter_ = -1;
  }
  term_iter_++;
  status_ = (term_iter_ < term_iter_max_) ? Belos::Failed : Belos::Passed;
  return status_;
}

template <class ScalarType, class MV, class OP>
void LSQRStatusTest<ScalarType,MV,OP>::print(std::ostream& os, int indent) const
{
  for (int j = 0; j < indent; j++)
    os << ' ';
  printStatus(os, status_);
  os << "limit of condition number = " << cond_lim_ << std::endl;
```

```
}

template <class ScalarType, class MV, class OP>
void LSQRStatusTest<ScalarType,MV,OP>::printStatus(std::ostream&os, Belos::StatusType
    type) const
{
  os << std::left << std::setw(13) << std::setfill('.');
  switch (type) {
  case Belos::Passed:
    os << "OK";
    break;
  case Belos::Failed:
    os << "Failed";
    break;
  case Belos::Undefined:
  default:
    os << "**";
    break;
  }
  os << std::left << std::setfill(' ');
  return;
}

#endif /* LSQR_STATUS_TEST_HPP */
```

## A.2 Code for MRNSD

### A.2.1 MRNSDSolMgr.hpp Code

```
#ifndef MRNSD_SOLMGR_HPP
#define MRNSD_SOLMGR_HPP

/* MRNSDSolMgr.hpp
 * The MRNSDSolMgr provides a solver manager for the MRNSD linear solver.
 */

#include "BelosConfigDefs.hpp"
#include "BelosTypes.hpp"
#include "BelosLinearProblem.hpp"
#include "BelosSolverManager.hpp"
#include "BelosStatusTestMaxIters.hpp"
#include "BelosStatusTestCombo.hpp"
#include "BelosOutputManager.hpp"
#include "Teuchos_TimeMonitor.hpp"
```

```
#include "MRNSDIter.hpp"
#include "MRNSDStatusTest.hpp"
#include "LeastErrorStatusTest.hpp"

// MRNSDSolMgr Exceptions
/* MRNSDSolMgrLinearProblemFailure is thrown when the linear problem is not
 * setup (i.e. setProblem() was not called) when solve() is called.
 * This std::exception is thrown from the MRNSDSolMgr::solve() method.
 */
class MRNSDSolMgrLinearProblemFailure : public Belos::BelosError {public:
    MRNSDSolMgrLinearProblemFailure(const std::string& what_arg) : Belos::BelosError(
        what_arg)
  {}};

/* MRNSDSolMgrBlockSizeFailure is thrown when the linear problem has more than
 * one RHS.  This std::exception is thrown from the MRNSDSolMgr::solve() method.
 */
class MRNSDSolMgrBlockSizeFailure : public Belos::BelosError {public:
    MRNSDSolMgrBlockSizeFailure(const std::string& what_arg) : Belos::BelosError(what_arg
        )
  {}};

template<class ScalarType, class MV, class OP>
class MRNSDSolMgr : public Belos::SolverManager<ScalarType,MV,OP> {

private:
  typedef Belos::MultiVecTraits<ScalarType,MV> MVT;
  typedef Belos::OperatorTraits<ScalarType,MV,OP> OPT;
  typedef Teuchos::ScalarTraits<ScalarType> SCT;
  typedef typename Teuchos::ScalarTraits<ScalarType>::magnitudeType MagnitudeType;
  typedef Teuchos::ScalarTraits<MagnitudeType> MT;

public:

  // Constructors/Destructor
  /* Empty constructor for MRNSDSolMgr.  This constructor takes no arguments and
   * sets the default values for the solver.  The linear problem must be passed
   * in using setProblem() before solve() is called on this object.  The solver
   * values can be changed using setParameters().
   */
  MRNSDSolMgr();

  /* Basic constructor for MRNSDSolMgr.  This constructor accepts the
   * LinearProblem to be solved in addition to a parameter list of options for
   * the solver manager. These options include the following:
```

```
 * - "Maximum Iterations" - an int specifying the maximum number of iterations
 * the underlying solver is allowed to perform. Default: 1000
 * - "Tolerance" - a MagnitudeType specifying the convergence tolerance.
 * Default: -1, resulting in a tolerance computed to be sqrt(eps)*norm(A^T*b).
 * - "Verbosity" - a sum of Belos::MsgType specifying the verbosity. Default:
 * Belos::Errors
 * - "Output Style" - a Belos::OutputType specifying the style of output.
 * Default: Belos::General
 * - "Output Stream" - a reference-counted pointer to the output stream where
 * all solver output is sent.  Default: Teuchos::rcp(&std::cout,false)
 * - "Output Frequency" - an int specifying how often convergence information
 * should be outputted.  Default: -1 (never)
 * - "Timer Label" - a std::string to use as a prefix for the timer labels.
 * Default: "Belos"
 */
MRNSDSolMgr( const Teuchos::RCP<Belos::LinearProblem<ScalarType,MV,OP> > &problem,
    const Teuchos::RCP<Teuchos::ParameterList> &pl );

// Destructor.
virtual ~MRNSDSolMgr() {};

// Accessor methods
const Belos::LinearProblem<ScalarType,MV,OP>& getProblem() const {
  return *problem_;
}

// Get a parameter list containing the valid parameters for this object.
Teuchos::RCP<const Teuchos::ParameterList> getValidParameters() const;

// Get a parameter list containing the current parameters for this object.
Teuchos::RCP<const Teuchos::ParameterList> getCurrentParameters() const { return
    params_; }

/* Return the timers for this object.
 * The timers are ordered as follows - time spent in solve() routine.
 */
Teuchos::Array<Teuchos::RCP<Teuchos::Time> > getTimers() const {
  return tuple(timerSolve_);
}

// Get the iteration count for the most recent call to solve().
int getNumIters() const {
  return numIters_;
}
```

```
/* Return whether a loss of accuracy was detected by this solver during the
 * most current solve.
 */
bool isLOADetected() const { return false; }

// Set methods
// Set the linear problem that needs to be solved.
void setProblem( const Teuchos::RCP<Belos::LinearProblem<ScalarType,MV,OP> > &problem )
     { problem_ = problem; }

// Set parameters the solver manager should use to solve the linear problem.
void setParameters( const Teuchos::RCP<Teuchos::ParameterList> &params );

// Reset methods
/* Performs a reset of the solver manager specified by the ResetType.  This
 * informs the solver manager that the solver should prepare for the next call
 * to solve by resetting certain elements of the iterative solver strategy.
 */
void reset( const Belos::ResetType type ) { if ((type & Belos::Problem) && !Teuchos::
    is_null(problem_)) problem_->setProblem(); }

// Solver application methods
/* This method performs possibly repeated calls to the underlying linear
 * solver's iterate() routine until the problem has been solved (as decided by
 * the solver manager) or the solver manager decides to quit.  This method
 * calls MRNSDIter::iterate(), which will return either because a specially
 * constructed status test evaluates to Belos::Passed or an std::exception is
 * thrown.  A return from MRNSDIter::iterate() signifies one of the following
 * scenarios:
 * - the maximum number of iterations has been exceeded. In this scenario, the
 * current solution to the linear system will be placed in the linear problem
 * and return Belos::Unconverged.
 * - global convergence has been met. In this case, the current solution to
 * the linear system will be placed in the linear problem and the solver
 * manager will return Belos::Converged.
 * Return: Belos::ReturnType specifying:
 * - Belos::Converged: the linear problem was solved to the specification
 * required by the solver manager.
 * - Belos::Unconverged: the linear problem was not solved to the
 * specification desired by the solver manager.
 */
Belos::ReturnType solve();

// Overridden from Teuchos::Describable
// Method to return description of the MRNSD solver manager
```

```
  std::string description() const;

private:

  // Linear problem.
  Teuchos::RCP<Belos::LinearProblem<ScalarType,MV,OP> > problem_;
  // Output manager.
  Teuchos::RCP<Belos::OutputManager<ScalarType> > printer_;
  Teuchos::RCP<std::ostream> outputStream_;
  // Status test.
  Teuchos::RCP<Belos::StatusTest<ScalarType,MV,OP> > sTest_;
  Teuchos::RCP<Belos::StatusTestMaxIters<ScalarType,MV,OP> > maxIterTest_;
  Teuchos::RCP<MRNSDStatusTest<ScalarType,MV,OP> > convTest_;
  Teuchos::RCP<LeastErrorStatusTest<ScalarType,MV,OP> > errorTest_;
  Teuchos::RCP<Belos::StatusTestOutput<ScalarType,MV,OP> > outputTest_;
  // Current parameter list.
  Teuchos::RCP<ParameterList> params_;
  // Default solver values.
  static const ScalarType tolerance_default_;
  static const int maxIters_default_;
  static const int verbosity_default_;
  static const int outputStyle_default_;
  static const int outputFreq_default_;
  static const int windowSize_default_;
  static const std::string label_default_;
  static const Teuchos::RCP<std::ostream> outputStream_default_;
  static const Teuchos::RCP< MV > trueSolution_default_;
  // Current solver values.
  ScalarType tolerance_;
  int maxIters_, numIters_, windowSize_;
  int verbosity_, outputStyle_, outputFreq_;
  Teuchos::RCP< MV > trueSolution_;
  // Timers.
  std::string label_;
  Teuchos::RCP<Teuchos::Time> timerSolve_;
  // Internal state variables.
  bool isSet_;
};

// Default solver values.
template<class ScalarType, class MV, class OP>
const ScalarType MRNSDSolMgr<ScalarType,MV,OP>::tolerance_default_ = -1.0;

template<class ScalarType, class MV, class OP>
```

```
const Teuchos::RCP< MV > MRNSDSolMgr<ScalarType,MV,OP>::trueSolution_default_ = Teuchos::
    null;

template<class ScalarType, class MV, class OP>
const int MRNSDSolMgr<ScalarType,MV,OP>::windowSize_default_ = 1;

template<class ScalarType, class MV, class OP>
const int MRNSDSolMgr<ScalarType,MV,OP>::maxIters_default_ = 1000;

template<class ScalarType, class MV, class OP>
const int MRNSDSolMgr<ScalarType,MV,OP>::verbosity_default_ = Belos::Errors;

template<class ScalarType, class MV, class OP>
const int MRNSDSolMgr<ScalarType,MV,OP>::outputStyle_default_ = Belos::General;

template<class ScalarType, class MV, class OP>
const int MRNSDSolMgr<ScalarType,MV,OP>::outputFreq_default_ = -1;

template<class ScalarType, class MV, class OP>
const std::string MRNSDSolMgr<ScalarType,MV,OP>::label_default_ = "Belos";

template<class ScalarType, class MV, class OP>
const Teuchos::RCP<std::ostream> MRNSDSolMgr<ScalarType,MV,OP>::outputStream_default_ =
    Teuchos::rcp(&std::cout,false);

// Empty Constructor
template<class ScalarType, class MV, class OP>
MRNSDSolMgr<ScalarType,MV,OP>::MRNSDSolMgr() :
  outputStream_(outputStream_default_),
  tolerance_(tolerance_default_),
  trueSolution_(trueSolution_default_),
  windowSize_(windowSize_default_),
  maxIters_(maxIters_default_),
  verbosity_(verbosity_default_),
  outputStyle_(outputStyle_default_),
  outputFreq_(outputFreq_default_),
  label_(label_default_),
  isSet_(false)
{}


// Basic Constructor
template<class ScalarType, class MV, class OP>
MRNSDSolMgr<ScalarType,MV,OP>::MRNSDSolMgr(const Teuchos::RCP<Belos::LinearProblem<
    ScalarType,MV,OP> > &problem, const Teuchos::RCP<Teuchos::ParameterList> &pl ) :
```

```
  problem_(problem),
  outputStream_(outputStream_default_),
  tolerance_(tolerance_default_),
  trueSolution_(trueSolution_default_),
  windowSize_(windowSize_default_),
  maxIters_(maxIters_default_),
  verbosity_(verbosity_default_),
  outputStyle_(outputStyle_default_),
  outputFreq_(outputFreq_default_),
  label_(label_default_),
  isSet_(false)
{
  TEST_FOR_EXCEPTION(problem_ == Teuchos::null, std::invalid_argument, "Problem not given
       to solver manager.");

  /* If the parameter list pointer is null, then set the current parameters to
   * the default parameter list.
   */
  if ( !is_null(pl) ) {
    setParameters( pl );
  }
}

template<class ScalarType, class MV, class OP>
void MRNSDSolMgr<ScalarType,MV,OP>::setParameters( const Teuchos::RCP<Teuchos::
    ParameterList> &params )
{
  // Create the internal parameter list if ones doesn't already exist.
  if (params_ == Teuchos::null) {
    params_ = Teuchos::rcp( new Teuchos::ParameterList(*getValidParameters()) );
  }
  else {
    params->validateParameters(*getValidParameters());
  }

  // Check for window size; update in our list and status test.
  if (params->isParameter("Window Size")) {
    windowSize_ = params->get("Window Size", windowSize_default_);
    params_->set("Window Size", windowSize_);
  }

  // Check for maximum number of iterations; update in our list and status test.
  if (params->isParameter("Maximum Iterations")) {
    maxIters_ = params->get("Maximum Iterations",maxIters_default_);
    params_->set("Maximum Iterations", maxIters_);
```

```
  if (maxIterTest_!=Teuchos::null)
    maxIterTest_->setMaxIters( maxIters_ );
}

// Check to see if timer label changed; update in our list and solver timer.
if (params->isParameter("Timer Label")) {
  std::string tempLabel = params->get("Timer Label", label_default_);
  if (tempLabel != label_) {
    label_ = tempLabel;
    params_->set("Timer Label", label_);
    std::string solveLabel = label_ + ": MRNSDSolMgr total solve time";
    timerSolve_ = Teuchos::TimeMonitor::getNewTimer(solveLabel);
  }
}

// Check for a change in verbosity level; update in our list.
if (params->isParameter("Verbosity")) {
  if (Teuchos::isParameterType<int>(*params,"Verbosity")) {
    verbosity_ = params->get("Verbosity", verbosity_default_);
  } else {
    verbosity_ = (int)Teuchos::getParameter<Belos::MsgType>(*params,"Verbosity");
  }
  params_->set("Verbosity", verbosity_);
  if (printer_ != Teuchos::null)
    printer_->setVerbosity(verbosity_);
}

// Check for a change in output style; update in our list.
if (params->isParameter("Output Style")) {
  if (Teuchos::isParameterType<int>(*params,"Output Style")) {
    outputStyle_ = params->get("Output Style", outputStyle_default_);
  } else {
    outputStyle_ = (int)Teuchos::getParameter<Belos::OutputType>(*params,"Output Style"
        );
  }
  params_->set("Output Style", outputStyle_);
  outputTest_ == Teuchos::null;
}

// Check for output stream; update in our list.
if (params->isParameter("Output Stream")) {
  outputStream_ = Teuchos::getParameter<Teuchos::RCP<std::ostream> >(*params,"Output
      Stream");
  params_->set("Output Stream", outputStream_);
  if (printer_ != Teuchos::null)
```

```
    printer_->setOStream( outputStream_ );
}

// Check for frequency level; update in our list and output status test.
if (verbosity_ & Belos::StatusTestDetails) {
  if (params->isParameter("Output Frequency")) {
    outputFreq_ = params->get("Output Frequency", outputFreq_default_);
  }
  params_->set("Output Frequency", outputFreq_);
  if (outputTest_ != Teuchos::null)
    outputTest_->setOutputFrequency( outputFreq_ );
}

// Create output manager if we need to.
if (printer_ == Teuchos::null) {
  printer_ = Teuchos::rcp( new Belos::OutputManager<ScalarType>(verbosity_,
      outputStream_) );
}

// Convergence
typedef Belos::StatusTestCombo<ScalarType,MV,OP>  StatusTestCombo_t;

// Check for convergence tolerance; update in our list and residual tests.
if (params->isParameter("Tolerance")) {
  tolerance_ = params->get("Tolerance",tolerance_default_);
  params_->set("Tolerance", tolerance_);
  if (convTest_ != Teuchos::null)
    convTest_->setTolerance( tolerance_ );
}

// Check for true solution; update in our list and status test.
if (params->isParameter("True Solution")) {
  trueSolution_ = params->get< Teuchos::RCP< MV > >("True Solution",
      trueSolution_default_);
  params_->set("True Solution",trueSolution_);
}

// Create status tests if we need to.
// Basic test checks maximum iterations.
if (maxIterTest_ == Teuchos::null)
  maxIterTest_ = Teuchos::rcp( new Belos::StatusTestMaxIters<ScalarType,MV,OP>(
      maxIters_ ) );

// Least error test if needed.
if (trueSolution_ != Teuchos::null)
```

```
    errorTest_ = Teuchos::rcp( new LeastErrorStatusTest<ScalarType,MV,OP>(trueSolution_,
        windowSize_) );

  // Status test specific to MRNSD.
  if (convTest_ == Teuchos::null)
    convTest_ = Teuchos::rcp( new MRNSDStatusTest<ScalarType,MV,OP>(tolerance_) );

  if (sTest_ == Teuchos::null)
    sTest_ = Teuchos::rcp( new StatusTestCombo_t( StatusTestCombo_t::OR, maxIterTest_,
        convTest_ ) );

  if (errorTest_ != Teuchos::null)
    // add error test to the status test combo
    ((Teuchos::rcp_dynamic_cast< StatusTestCombo_t >) (sTest_))->addStatusTest(errorTest_
        );

  if (outputTest_ == Teuchos::null) {
    // Create the status test output class.
    // This class manages and formats the output from the status test.
    Belos::StatusTestOutputFactory<ScalarType,MV,OP> stoFactory( outputStyle_ );
    outputTest_ = stoFactory.create( printer_, sTest_, outputFreq_, Belos::Passed+Belos::
        Failed+Belos::Undefined );

    // Set the solver string for the output test
    std::string solverDesc = " MRNSD ";
    outputTest_->setSolverDesc( solverDesc );

  }

  // Create the timer if we need to.
  if (timerSolve_ == Teuchos::null) {
    std::string solveLabel = label_ + ": MRNSDSolMgr total solve time";
    timerSolve_ = Teuchos::TimeMonitor::getNewTimer(solveLabel);
  }

  // Inform the solver manager that the current parameters were set.
  isSet_ = true;
}


template<class ScalarType, class MV, class OP>
Teuchos::RCP<const Teuchos::ParameterList>
MRNSDSolMgr<ScalarType,MV,OP>::getValidParameters() const
{
  static Teuchos::RCP<const Teuchos::ParameterList> validPL;
```

```
  // Set all the valid parameters and their default values.
  if(is_null(validPL)) {
    Teuchos::RCP<Teuchos::ParameterList> pl = Teuchos::parameterList();
    pl->set("Tolerance", tolerance_default_,
      "The convergence tolerance.");
    pl->set("Maximum Iterations", maxIters_default_,
      "The maximum number of block iterations allowed for each\n"
      "set of RHS solved.");
    pl->set("Window Size", windowSize_default_,
      "The window size for the least error test");
    pl->set("True Solution", trueSolution_default_,
      "The true solution for the least error test");
    pl->set("Verbosity", verbosity_default_,
      "What type(s) of solver information should be outputted\n"
      "to the output stream.");
    pl->set("Output Style", outputStyle_default_,
      "What style is used for the solver information outputted\n"
      "to the output stream.");
    pl->set("Output Frequency", outputFreq_default_,
      "How often convergence information should be outputted\n"
      "to the output stream.");
    pl->set("Output Stream", outputStream_default_,
      "A reference-counted pointer to the output stream where all\n"
      "solver output is sent.");
    pl->set("Timer Label", label_default_,
      "The string to use as a prefix for the timer labels.");
    validPL = pl;
  }
  return validPL;
}


// Solve method
template<class ScalarType, class MV, class OP>
Belos::ReturnType MRNSDSolMgr<ScalarType,MV,OP>::solve() {
  // Set the current parameters if they were not set before.
  // NOTE:  This may occur if the user generated the solver manager with the
  // default constructor and then didn't set any parameters using setParameters.
  if (!isSet_) {
    setParameters(Teuchos::parameterList(*getValidParameters()));
  }

  TEST_FOR_EXCEPTION(!problem_->isProblemSet(),MRNSDSolMgrLinearProblemFailure, "
      MRNSDSolMgr::solve(): Linear problem is not ready, setProblem() has not been called
```

```
    .");
TEST_FOR_EXCEPTION(MVT::GetNumberVecs( *(problem_->getRHS()) ) != 1,
    MRNSDSolMgrBlockSizeFailure, "MRNSDSolMgr::solve(): Incorrect number of RHS vectors
    , should be exactly 1.");

// Inform the linear problem of the current linear system to solve.
std::vector<int> currRHSIdx(1, 0);
problem_->setLSIndex(currRHSIdx);
Teuchos::ParameterList plist;
outputTest_->reset();
// Assume convergence is achieved, then let any failed convergence set this
// to false.
bool isConverged = true;

// MRNSD solver
Teuchos::RCP<MRNSDIter<ScalarType,MV,OP> > mrnsd_iter = Teuchos::rcp( new MRNSDIter<
    ScalarType,MV,OP>(problem_, printer_, outputTest_, plist) );
Teuchos::TimeMonitor slvtimer(*timerSolve_);

// Reset the number of iterations.
mrnsd_iter->resetNumIters();
// Reset the number of calls that the status test output knows about.
outputTest_->resetNumCalls();
// Set the new state and initialize the solver.
MRNSDIterationState<ScalarType,MV> newstate;
mrnsd_iter->initializeMRNSD(newstate);
// Tell mrnsd_iter to iterate
try {
  mrnsd_iter->iterate();
  // Check convergence first
  if ( convTest_->getStatus() == Belos::Passed ) {
  }
  else if ( (errorTest_ != Teuchos::null) && errorTest_->getStatus() == Belos::Passed )
       {
  }
  else if ( maxIterTest_->getStatus() == Belos::Passed ) {
    // We don't have convergence
    isConverged = false;
  }
  // We returned from iterate(), but none of our status tests Passed.
  // Something is wrong, and it is probably our fault.
  else {
    TEST_FOR_EXCEPTION(true,std::logic_error,
     "MRNSDSolMgr::solve(): Invalid return from MRNSDIteration::iterate().");
  }
```

```
  }
  catch (const std::exception &e) {
    printer_->stream(Belos::Errors) << "Error! Caught std::exception in MRNSDIter::
        iterate() at iteration "
            << mrnsd_iter->getNumIters() << std::endl
            << e.what() << std::endl;
    throw;
  }

  // Inform the linear problem that we are finished with this linear system.
  problem_->setCurrLS();
  // Print final summary and timing information.
  sTest_->print( printer_->stream(Belos::FinalSummary) );
  Teuchos::TimeMonitor::summarize( printer_->stream(Belos::TimingDetails) );
  // Get iteration information for this solve.
  numIters_ = mrnsd_iter->getNumIters();

  if (!isConverged) {
    return Belos::Unconverged; // return from MRNSDSolMgr::solve()
  }
  return Belos::Converged; // return from MRNSDSolMgr::solve()
}

// This method requires the solver manager to return a std::string that
// describes itself.
template<class ScalarType, class MV, class OP>
std::string MRNSDSolMgr<ScalarType,MV,OP>::description() const
{
  std::ostringstream oss;
  oss << "MRNSDSolMgr<...,"<<Teuchos::ScalarTraits<ScalarType>::name()<<">";
  return oss.str();
}

#endif /* MRNSD_SOLMGR_HPP */
```

### A.2.2 MRNSDIter.hpp Code

```
#ifndef MRNSD_ITER_HPP
#define MRNSD_ITER_HPP

/* MRNSDIter.hpp
 * Concrete class for performing the MRNSD iteration.
 */

#include "BelosConfigDefs.hpp"
#include "BelosTypes.hpp"
```

```
#include "BelosIteration.hpp"
#include "BelosLinearProblem.hpp"
#include "BelosOutputManager.hpp"
#include "BelosStatusTest.hpp"
#include "BelosOperatorTraits.hpp"
#include "BelosMultiVecTraits.hpp"
#include "Teuchos_SerialDenseMatrix.hpp"
#include "Teuchos_SerialDenseVector.hpp"
#include "Teuchos_ScalarTraits.hpp"
#include "Teuchos_ParameterList.hpp"
#include "Teuchos_TimeMonitor.hpp"

  // MRNSDIteration Structures
  /* Structure to contain pointers to MRNSDIteration state variables.
   * This struct is utilized by initialize() and getState().
   */
  template <class ScalarType, class MV>
  struct MRNSDIterationState {
    /* Vector g. */
    Teuchos::RCP<const MV> G;
    /* Vector xg. */
    Teuchos::RCP<const MV> XG;
    /* Storage vector. */
    Teuchos::RCP<const MV> AV;
    /* Storage vector. */
    Teuchos::RCP<const MV> AtV;
    /* The gamma value. */
    ScalarType gamma;

    MRNSDIterationState() : G(Teuchos::null), XG(Teuchos::null),
          AV(Teuchos::null), AtV(Teuchos::null),
          gamma(0.0)
    {}
  };

  // MRNSDIteration Exceptions
  /* MRNSDIterationInitFailure is thrown when the MRNSDIteration object is
   * unable to generate an initial iterate in the initialize() routine.  This
   * std::exception is thrown from the initialize() method, which is called by
   * the user or from the iterate() method if isInitialized() == false.  In the
   * case that this std::exception is thrown, isInitialized() will be false and
   * the user will need to provide a new initial iterate to the iteration.
   */
class MRNSDIterationInitFailure : public Belos::BelosError {public:
```

```
      MRNSDIterationInitFailure(const std::string& what_arg) : Belos::BelosError(what_arg
          )
    {}};

  /* MRNSDIterateFailure is thrown when the MRNSDIteration object is unable to
   * compute the next iterate in the iterate() routine.  This std::exception is
   * thrown from the iterate() method.
   */
class MRNSDIterateFailure : public Belos::BelosError {public:
      MRNSDIterateFailure(const std::string& what_arg) : Belos::BelosError(what_arg)
    {}};

template<class ScalarType, class MV, class OP>
class MRNSDIter : virtual public Belos::Iteration<ScalarType,MV,OP> {

  public:
  typedef Belos::MultiVecTraits<ScalarType,MV> MVT;
  typedef Belos::OperatorTraits<ScalarType,MV,OP> OPT;
  typedef Teuchos::ScalarTraits<ScalarType> SCT;
  typedef typename SCT::magnitudeType MagnitudeType;

  // Constructors/Destructor
  /* MRNSDIter constructor with linear problem, solver utilities, and parameter
   * list of solver options.  This constructor takes pointers required by the
   * linear solver iteration, in addition to a parameter list of options for the
   * linear solver.
   */
  MRNSDIter( const Teuchos::RCP<Belos::LinearProblem<ScalarType,MV,OP> > &problem, const
      Teuchos::RCP<Belos::OutputManager<ScalarType> > &printer, const Teuchos::RCP<Belos
      ::StatusTest<ScalarType,MV,OP> > &tester, Teuchos::ParameterList &params );

  // Destructor.
  virtual ~MRNSDIter() {};

  // Solver methods
  /* This method performs MRNSD iterations until the status test indicates the
   * need to stop or an error occurs (in which case, an std::exception is
   * thrown).  This function will first determine whether the solver is
   * initialized; if not, it will call initialize() using default arguments.
   * After initialization, the solver performs MRNSD iterations until the status
   * test evaluates as Belos::Passed, at which point the method returns to the
   * caller.  The status test is queried at the beginning of the iteration.
   */
  void iterate();
```

```
/* Initialize the solver to an iterate, providing a complete state.  The
 * MRNSDIter contains a certain amount of state, consisting of a gradient
 * vector, helper vectors, and gamma.  For any pointer in newstate which
 * directly points to the multivectors in the solver, the data is not copied.
 */
void initializeMRNSD(MRNSDIterationState<ScalarType,MV> newstate);

// Initialize the solver.
void initialize()
{
  MRNSDIterationState<ScalarType,MV> empty;
  initializeMRNSD(empty);
}

/* Get the current state of the linear solver.  The data is only valid if
 * isInitialized() == true.
 * Return: A MRNSDIterationState object containing const pointers to the
 * current solver state.
 */
MRNSDIterationState<ScalarType,MV> getState() const {
  MRNSDIterationState<ScalarType,MV> state;
  state.G = G_;
  state.XG = XG_;
  state.AV = AV_;
  state.AtV = AtV_;
  state.gamma = gamma_;
  return state;
}

// Status methods
// Get the current iteration count.
int getNumIters() const { return iter_; }

// Reset the iteration count.
void resetNumIters( int iter = 0 ) { iter_ = iter; }

// Get the norms of the residuals native to the solver.
// This method returns a null pointer because these aren't used with MRNSD.
Teuchos::RCP<const MV> getNativeResiduals( std::vector<MagnitudeType> *norms ) const {
    return Teuchos::null; }

// Get the current update to the linear system.
// This method returns a null pointer because the linear problem is current.
Teuchos::RCP<MV> getCurrentUpdate() const { return Teuchos::null; }
```

```
// Accessor methods
// Get a constant reference to the linear problem.
const Belos::LinearProblem<ScalarType,MV,OP>& getProblem() const { return *lp_; }

// Get blocksize to be used by iterative solver in solving this linear problem
int getBlockSize() const { return 1; }

// Set blocksize to be used by iterative solver in solving this linear problem
void setBlockSize(int blockSize) {
  TEST_FOR_EXCEPTION(blockSize!=1,std::invalid_argument,
         "MRNSDIter::setBlockSize(): Cannot use a block size that is not one.");
}

// States whether the solver has been initialized or not.
bool isInitialized() { return initialized_; }

private:

// Method for initalizing the state storage needed by MRNSD.
void setStateSize();

// Classes inputed through constructor that define the linear problem to be
// solved.
const Teuchos::RCP<Belos::LinearProblem<ScalarType,MV,OP> >    lp_;
const Teuchos::RCP<Belos::OutputManager<ScalarType> >          om_;
const Teuchos::RCP<Belos::StatusTest<ScalarType,MV,OP> >       stest_;
// Current solver state.
/* initialized_ specifies that the basis vectors have been initialized and the
 * iterate() routine is capable of running; _initialize is controlled  by the
 * initialize() member method.  For the implications of the state of
 * initialized_, please see documentation for initialize().
 */
bool initialized_;
/* stateStorageInitialized_ specifies that the state storage has been
 * initialized.  This initialization may be postponed if the linear problem
 * was generated without the right-hand side or solution vectors.
 */
bool stateStorageInitialized_;
// Current number of iterations performed.
int iter_;
// Vector g
Teuchos::RCP<MV> G_;
// Vector xg
Teuchos::RCP<MV> XG_;
// Helper vector
```

```
  Teuchos::RCP<MV> AV_;
  // Helper vector
  Teuchos::RCP<MV> AtV_;
  // Gamma value
  ScalarType gamma_;
};

  // Constructor.
  template<class ScalarType, class MV, class OP>
  MRNSDIter<ScalarType,MV,OP>::MRNSDIter(const Teuchos::RCP<Belos::LinearProblem<
      ScalarType,MV,OP> > &problem, const Teuchos::RCP<Belos::OutputManager<ScalarType> >
       &printer, const Teuchos::RCP<Belos::StatusTest<ScalarType,MV,OP> > &tester,
      Teuchos::ParameterList &params ):
    lp_(problem),
    om_(printer),
    stest_(tester),
    initialized_(false),
    stateStorageInitialized_(false),
    iter_(0)
  {
  }

  // Setup the state storage.
  template <class ScalarType, class MV, class OP>
  void MRNSDIter<ScalarType,MV,OP>::setStateSize ()
  {
    if (!stateStorageInitialized_) {
      // Check if there is any multivector to clone from.
      Teuchos::RCP<const MV> lhsMV = lp_->getLHS();
      Teuchos::RCP<const MV> rhsMV = lp_->getRHS();
      if (lhsMV == Teuchos::null || rhsMV == Teuchos::null) {
        stateStorageInitialized_ = false;
        return;
      }
      else {
        if (G_ == Teuchos::null) {
          TEST_FOR_EXCEPTION(rhsMV == Teuchos::null, std::invalid_argument, "MRNSDIter::
              setStateSize(): linear problem does not specify right hand multivector to
              clone from.");
          TEST_FOR_EXCEPTION(lhsMV == Teuchos::null, std::invalid_argument, "MRNSDIter::
              setStateSize(): linear problem does not specify left hand multivector to
              clone from.");
          // Initialize the state storage
          G_ = MVT::Clone( *lhsMV, 1 );
          XG_ = MVT::Clone( *lhsMV, 1 );
```

```
        AV_ = MVT::Clone( *rhsMV, 1 );
        AtV_ = MVT::Clone( *lhsMV, 1 );
        // State storage has now been initialized.
        stateStorageInitialized_ = true;
      }
    }
  }
}

// Initialize this iteration object
template <class ScalarType, class MV, class OP>
void MRNSDIter<ScalarType,MV,OP>::initializeMRNSD(MRNSDIterationState<ScalarType,MV>
    newstate)
{
  // Initialize the state storage if it isn't already.
  if (!stateStorageInitialized_)
    setStateSize();

  TEST_FOR_EXCEPTION(!stateStorageInitialized_,std::invalid_argument, "MRNSDIter::
      initialize(): Cannot initialize state storage!");
  std::string errstr("MRNSDIter::initialize(): Specified multivectors must have a
      consistent length and width.");

  // Create convenience variables for zero, one, and tau.
  const ScalarType zero = Teuchos::ScalarTraits<ScalarType>::zero();
  const ScalarType one = Teuchos::ScalarTraits<ScalarType>::one();
  const MagnitudeType tau = Teuchos::ScalarTraits<MagnitudeType>::squareroot(Teuchos::
      ScalarTraits< MagnitudeType >::eps());
  Teuchos::RCP<MV> cur_soln_vec = lp_->getCurrLHSVec();
  Teuchos::RCP<const MV> rhsMV = lp_->getRHS();
  std::vector<MagnitudeType> norm(1);
  std::vector<ScalarType> meanValue(1);

  // Compensate for negative values of initial guess.
  Teuchos::ArrayRCP<const ScalarType> xView = cur_soln_vec->get1dView();
  ScalarType minx = xView[0];
  for (int i = 1; i < xView.size(); i++) {
    if (minx > xView[i])
      minx = xView[i];
  }
  xView = Teuchos::null;
  if (cur_soln_vec->isDistributed()) {
    Teuchos::Array<ScalarType> lminxPacket(1), minxPacket(1);
    lminxPacket[0] = minx;
```

```
    Teuchos::reduceAll(*cur_soln_vec->getMap()->getComm(), Teuchos::REDUCE_MIN, 1, &
        lminxPacket[0], &minxPacket[0]);
    minx = minxPacket[0];
  }
  if (minx < 0) {
    AtV_->putScalar(tau - minx);
    MVT::MvAddMv( one, *cur_soln_vec, one, *AtV_, *cur_soln_vec );
    lp_->updateSolution();
  }

  // Change initial guess if all zeros.
  MVT::MvNorm( *cur_soln_vec, norm );
  if (norm[0] == zero) {
    rhsMV->meanValue(meanValue);
    cur_soln_vec->putScalar( std::max(meanValue[0], tau) );
    lp_->updateSolution();
  }

  // Compute initial g, xg, and gamma.
  lp_->applyOp ( *cur_soln_vec, *AV_);
  MVT::MvAddMv( one, *rhsMV, -one, *AV_, *AV_ );
  lp_->getOperator()->apply( *AV_, *G_, Teuchos::TRANS);
  MVT::MvScale( *G_, -one );
  XG_->elementWiseMultiply( one, *(G_->getVector(0)), *cur_soln_vec, zero );
  MVT::MvDot( *G_, *XG_, norm );
  gamma_ = norm[0];

  // The solver is initialized
  initialized_ = true;
}

// Iterate until the status test informs us we should stop.
template <class ScalarType, class MV, class OP>
void MRNSDIter<ScalarType,MV,OP>::iterate()
{
  // Allocate/initialize data structures.
  if (initialized_ == false) {
    initialize();
  }

  // Create convenience variables for zero and one.
  const ScalarType one = Teuchos::ScalarTraits<ScalarType>::one();
  const MagnitudeType zero = Teuchos::ScalarTraits<MagnitudeType>::zero();

  // Allocate memory for scalars and pointers.
```

```
ScalarType alpha, beta;
Teuchos::Array<ScalarType> lalphaPacket(1), alphaPacket(1);
std::vector<MagnitudeType> norm(1);
Teuchos::ArrayRCP<const ScalarType> xView, xgView;

// Get the current solution vector.
Teuchos::RCP<MV> cur_soln_vec = lp_->getCurrLHSVec();

// Check that the current solution vector only has one column.
TEST_FOR_EXCEPTION( MVT::GetNumberVecs(*cur_soln_vec) != 1, MRNSDIterateFailure, "
    MRNSDIter::iterate(): current linear system has more than one vector!" );

// Iterate until the status test tells us to stop.
while (stest_->checkStatus(this) != Belos::Passed) {
  // Increment the iteration
  iter_++;
  lp_->applyOp( *XG_, *AV_ );
  MVT::MvScale( *AV_, -one );
  MVT::MvDot( *AV_, *AV_, norm );
  alpha = gamma_ / norm[0];
  xView = cur_soln_vec->get1dView();
  xgView = XG_->get1dView();
  for (int i = 0; i < xgView.size(); i++) {
    if (xgView[i] > 0) {
      beta = xView[i] / xgView[i];
      if (beta < alpha)
        alpha = beta;
    }
  }
  xView = Teuchos::null;
  xgView = Teuchos::null;
  if (cur_soln_vec->isDistributed()) {
    lalphaPacket[0] = alpha;
    Teuchos::reduceAll( *cur_soln_vec->getMap()->getComm(), Teuchos::REDUCE_MIN, 1, &
        lalphaPacket[0], &alphaPacket[0]);
    alpha = alphaPacket[0];
  }

  MVT::MvAddMv( one, *cur_soln_vec, -alpha, *XG_, *cur_soln_vec );
  lp_->getOperator()->apply( *AV_, *AtV_, Teuchos::TRANS );
  MVT::MvAddMv( one, *G_, alpha, *AtV_, *G_ );
  XG_->elementWiseMultiply( one, *(G_->getVector(0)), *cur_soln_vec, zero );
  MVT::MvDot( *G_, *XG_, norm );
  gamma_ = norm[0];
```

```
      // Update the solution vector.
      lp_->updateSolution();

    } // end while (sTest_->checkStatus(this) != Passed)
  }

#endif /* MRNSD_ITER_HPP */
```

### A.2.3 MRNSDStatusTest.hpp Code

```
#ifndef MRNSD_STATUS_TEST_HPP
#define MRNSD_STATUS_TEST_HPP

/* MRNSDStatusTest.hpp
 * Belos::StatusTest class for specifying convergence of MRNSD.
 */

#include "BelosStatusTest.hpp"

template <class ScalarType, class MV, class OP>
class MRNSDStatusTest: public Belos::StatusTest<ScalarType,MV,OP> {

public:

  typedef Teuchos::ScalarTraits<ScalarType> SCT;
  typedef typename SCT::magnitudeType MagnitudeType;
  typedef Belos::MultiVecTraits<ScalarType,MV>  MVT;

  // Constructor/Destructor.
  /* The constructor takes one argument, specifying the tolerance.  If negative,
   * then the tolerance is computed as sqrt(eps)*norm(A^T*b).
   */
  MRNSDStatusTest( ScalarType tolerance );

  // Destructor
  virtual ~MRNSDStatusTest();

  // Parameter definition methods.
  /* Set the value of the tolerance.  We allow the tolerance to be reset for
   * cases where, in the process of testing convergence, we find that the
   *initial limit was too tight or too lax.
  */
  int setTolerance(ScalarType tolerance) {
    tolerance_ = tolerance;
    return(0);}
```

```
  // Status methods
  /* This method checks to see if the convergence criteria are met using the
   * current information from the iterative solver, returning one of:
   * Belos::Unconverged, Belos::Converged, or Belos::Failed.
   */
  Belos::StatusType checkStatus(Belos::Iteration<ScalarType,MV,OP> *iSolver );

  // Return the result of the most recent CheckStatus call.
  Belos::StatusType getStatus() const {return(status_);}

  // Reset the status test to the initial internal state.
  void reset();

  // Print methods
  // Output formatted description of stopping test to output stream.
  void print(std::ostream& os, int indent = 0) const;

  // Print message for each status specific to this stopping test.
  void printStatus(std::ostream& os, Belos::StatusType type) const;

  // Return the value of the tolerance set in the constructor.
  ScalarType getTolerance() const {return(tolerance_);};

  // Call to setup initialization.
  Belos::StatusType firstCallCheckStatusSetup(Belos::Iteration<ScalarType,MV,OP>* iSolver
      );

  // Overridden from Teuchos::Describable
  // Method to return description of the MRNSD status test.
  std::string description() const
  {
    std::ostringstream oss;
    oss << "MRNSDStatusTest<>: [ tolerance = " << tolerance_ << " ]";
    return oss.str();
  }

private:

  // Tolerance used to determine convergence
  ScalarType tolerance_;
  // Status
  Belos::StatusType status_;
  // Is this the first time CheckStatus is called?
  bool firstcallCheckStatus_;
};
```

```
template <class ScalarType, class MV, class OP>
MRNSDStatusTest<ScalarType,MV,OP>::MRNSDStatusTest( ScalarType tolerance )
  : tolerance_(tolerance),
    status_(Belos::Undefined),
    firstcallCheckStatus_(true)
{}

template <class ScalarType, class MV, class OP>
MRNSDStatusTest<ScalarType,MV,OP>::~MRNSDStatusTest()
{}

template <class ScalarType, class MV, class OP>
void MRNSDStatusTest<ScalarType,MV,OP>::reset()
{
  status_ = Belos::Undefined;
  firstcallCheckStatus_ = true;
}

template <class ScalarType, class MV, class OP>
Belos::StatusType MRNSDStatusTest<ScalarType,MV,OP>::firstCallCheckStatusSetup(Belos::
    Iteration<ScalarType,MV,OP>* iSolver)
{
  if (firstcallCheckStatus_) {
      firstcallCheckStatus_ = false;
      if (tolerance_ < 0) {
        Belos::LinearProblem<ScalarType,MV,OP> lp = iSolver->getProblem();
        Teuchos::RCP<const MV> lhsMV = lp.getLHS();
        Teuchos::RCP<const MV> rhsMV = lp.getRHS();
        Teuchos::RCP<MV> trAb = MVT::Clone( *lhsMV, 1 );
        std::vector<MagnitudeType> norm(1);
        lp.applyOp( *rhsMV, *trAb );
        MVT::MvNorm( *trAb, norm );
        tolerance_ = Teuchos::ScalarTraits< MagnitudeType >::squareroot(Teuchos::
            ScalarTraits< MagnitudeType >::eps() ) * norm[0];
      }
  }
  return Belos::Undefined;
}

template <class ScalarType, class MV, class OP>
Belos::StatusType MRNSDStatusTest<ScalarType,MV,OP>::checkStatus( Belos::Iteration<
    ScalarType,MV,OP>* iSolver)
{
  if (firstcallCheckStatus_) {
```

```
    Belos::StatusType status = firstCallCheckStatusSetup(iSolver);
    if(status==Belos::Failed) {
      status_ = Belos::Failed;
      return(status_);
    }
  }

  MRNSDIter<ScalarType,MV,OP>* solver = dynamic_cast< MRNSDIter<ScalarType,MV,OP>* > (
      iSolver);
  MRNSDIterationState< ScalarType, MV > state = solver->getState();
  ScalarType rnrm = Teuchos::ScalarTraits< MagnitudeType >::squareroot(std::abs(state.
      gamma));
  status_ = (rnrm <= tolerance_) ? Belos::Passed : Belos::Failed;
  return status_;
}

template <class ScalarType, class MV, class OP>
void MRNSDStatusTest<ScalarType,MV,OP>::print(std::ostream& os, int indent) const
{
  for (int j = 0; j < indent; j++)
    os << ' ';
  printStatus(os, status_);
  os << "tolerance = " << tolerance_ << std::endl;
}

template <class ScalarType, class MV, class OP>
void MRNSDStatusTest<ScalarType,MV,OP>::printStatus(std::ostream&os, Belos::StatusType
    type) const
{
  os << std::left << std::setw(13) << std::setfill('.');
  switch (type) {
  case Belos::Passed:
    os << "OK";
    break;
  case Belos::Failed:
    os << "Failed";
    break;
  case Belos::Undefined:
  default:
    os << "**";
    break;
  }
  os << std::left << std::setfill(' ');
  return;
}
```

```
#endif /* MRNSD_STATUS_TEST_HPP */
```

## A.3 Code for Least Error Status Test

### A.3.1 LeastErrorStatusTest.hpp Code

```
#ifndef LEAST_ERROR_STATUS_TEST_HPP
#define LEAST_ERROR_STATUS_TEST_HPP

/* LeastErrorStatusTest.hpp
 * Belos::StatusTest class for specifying convergence of iterative solvers based
 * on lowest error.
 */

#include "BelosStatusTest.hpp"
#include "Teuchos_Array.hpp"
#include "BelosTpetraAdapter.hpp"

template <class ScalarType, class MV, class OP>
class LeastErrorStatusTest: public Belos::StatusTest<ScalarType,MV,OP> {

public:

  typedef Teuchos::ScalarTraits<ScalarType> SCT;
  typedef typename SCT::magnitudeType MagnitudeType;
  typedef Belos::MultiVecTraits<ScalarType,MV>  MVT;

  // Constructor/Destructor.
  /* The constructor takes one argument, a pointer to the exact solution, plus
   * an optional argument that specifies the "windowSize" for this test.  If an
   * approximate solution has the least error of the next windowSize approximate
   * solutions, convergence is considered achieved.  By default, the windowSize
   * is 1, meaning that an approximate solution is best if the next solution has
   * a larger error.
   */
  LeastErrorStatusTest( const Teuchos::RCP< const MV > &trueSolution, int windowSize = 1
      );

  // Destructor
  virtual ~LeastErrorStatusTest() {};

  // Status method
  /* This method checks to see if the convergence criteria are met using the
   * current information from the iterative solver, returning one of
```

```
     * Belos::Unconverged, Belos::Converged, or Belos::Failed.
     */
    Belos::StatusType checkStatus(Belos::Iteration<ScalarType,MV,OP> *iSolver );

    // Return the result of the most recent CheckStatus call.
    Belos::StatusType getStatus() const {return(status_);}

    // Reset the status test to the initial internal state.
    void reset();

    // Print methods
    // Output formatted description of stopping test to output stream.
    void print(std::ostream& os, int indent = 0) const;

    // Print message for each status specific to this stopping test.
    void printStatus(std::ostream& os, Belos::StatusType type) const;

    // Methods to access data members.
    // Return the value of the window size set in the constructor.
    int getWindowSize() const {return(window_size_);};

    // Call to setup initialization.
    Belos::StatusType firstCallCheckStatusSetup(Belos::Iteration<ScalarType,MV,OP>* iSolver
        );

    // Overridden from Teuchos::Describable
    // Method to return description of the least error status test.
    std::string description() const
    {
      std::ostringstream oss;
      oss << "LeastErrorStatusTest<>: [ window size = " << window_size_ << " ]";
      return oss.str();
    }

  private:

    // True solution
    const Teuchos::RCP< const MV > true_solution_;
    // Norm of true solution
    MagnitudeType true_norm_;
    // Current best approximate solution
    Teuchos::RCP< MV > best_solution_;
    // Helper multivector to compute difference between approximate and truth
    Teuchos::RCP< MV > diff_mv_;
    // Window size
```

```
  int window_size_;
  // Current window size
  int best_window_;
  // Current best error
  MagnitudeType best_error_;
  // Status
  Belos::StatusType status_;
  // Is this the first time CheckStatus is called?
  bool firstcallCheckStatus_;
};

template <class ScalarType, class MV, class OP>
LeastErrorStatusTest<ScalarType,MV,OP>::LeastErrorStatusTest( const Teuchos::RCP< const
    MV > &trueSolution, int windowSize /* = 1 */ )
  : true_solution_(trueSolution),
    window_size_(windowSize),
    status_(Belos::Undefined),
    firstcallCheckStatus_(true)
{
}

template <class ScalarType, class MV, class OP>
void LeastErrorStatusTest<ScalarType,MV,OP>::reset()
{
  status_ = Belos::Undefined;
  firstcallCheckStatus_ = true;
}

template <class ScalarType, class MV, class OP>
Belos::StatusType LeastErrorStatusTest<ScalarType,MV,OP>::firstCallCheckStatusSetup(Belos
    ::Iteration<ScalarType,MV,OP>* iSolver)
{
  const ScalarType one = Teuchos::ScalarTraits<ScalarType>::one();
  std::vector< MagnitudeType > norm_(1);
  if (firstcallCheckStatus_) {
      firstcallCheckStatus_ = false;
      MVT::MvNorm( *true_solution_, norm_ );
      true_norm_ = norm_[0];
      Belos::LinearProblem< ScalarType, MV, OP > problem = iSolver->getProblem();
      diff_mv_ = MVT::Clone( *problem.getLHS(), 1 );
      best_solution_ = MVT::CloneCopy( *problem.getLHS() );
      MVT::MvAddMv( one, *best_solution_, -one, *true_solution_, *diff_mv_);
      MVT::MvNorm( *diff_mv_, norm_ );
      best_error_ = norm_[0];
      best_window_ = -1;
```

```
    }
    return Belos::Undefined;
}

template <class ScalarType, class MV, class OP>
Belos::StatusType LeastErrorStatusTest<ScalarType,MV,OP>::checkStatus( Belos::Iteration<
    ScalarType,MV,OP>* iSolver)
{
  if (firstcallCheckStatus_) {
    Belos::StatusType status = firstCallCheckStatusSetup(iSolver);
    if(status==Belos::Failed) {
      status_ = Belos::Failed;
      return(status_);
    }
  }

  int iter = iSolver->getNumIters();
  const ScalarType one = Teuchos::ScalarTraits<ScalarType>::one();
  const ScalarType zero = Teuchos::ScalarTraits<ScalarType>::zero();
  std::vector< MagnitudeType > norm_(1);
  Belos::LinearProblem< ScalarType, MV, OP > problem = iSolver->getProblem();
  MVT::MvAddMv( one, *(problem.getLHS()), -one, *true_solution_, *diff_mv_ );
  MVT::MvNorm( *diff_mv_, norm_ );
  if (problem.getLHS()->getMap()->getComm()->getRank() == 0)
    std::cout << "iter " << iter << " norm = " << norm_[0] / true_norm_ << std::endl;

  // Check if our best solution is better than this iteration's solution.
  if ( best_error_ <= norm_[0] ) {
    // If yes, increment window size.
    best_window_++;
    // Now check if we have a best solution for the given window size.
    if ( best_window_ >= window_size_ ) {
      iSolver->resetNumIters( iter - window_size_ );
      MVT::MvAddMv( one, *best_solution_, zero, *(problem.getLHS()), *(problem.getLHS())
          );
      problem.updateSolution();
      status_ = Belos::Passed;
      std::cout << "norm of truth = " << true_norm_ << std::endl;
      std::cout << "norm of solution = " << best_error_ << std::endl;
    } else {
      status_ = Belos::Failed;
    }
  } else {
    // If no, save current solution and its error and reset current window.
    best_solution_ = MVT::CloneCopy( *(problem.getLHS()) );
```

```
    best_error_ = norm_[0];
    best_window_ = 0;
    status_ = Belos::Failed;
  }
  return status_;
}

template <class ScalarType, class MV, class OP>
void LeastErrorStatusTest<ScalarType,MV,OP>::print(std::ostream& os, int indent) const
{
  for (int j = 0; j < indent; j++)
    os << ' ';
  printStatus(os, status_);
  os << "window size = " << window_size_ << std::endl;
}

template <class ScalarType, class MV, class OP>
void LeastErrorStatusTest<ScalarType,MV,OP>::printStatus(std::ostream&os, Belos::
    StatusType type) const
{
  os << std::left << std::setw(13) << std::setfill('.');
  switch (type) {
  case Belos::Passed:
    os << "OK";
    break;
  case Belos::Failed:
    os << "Failed";
    break;
  case Belos::Undefined:
  default:
    os << "**";
    break;
  }
  os << std::left << std::setfill(' ');
  return;
}

#endif /* LEAST_ERROR_STATUS_TEST_HPP */
```

## A.4 Code for PET Application

### A.4.1 HRRT.hpp Code

```
#include <Teuchos_RCP.hpp>
#include <Teuchos_ParameterList.hpp>
```

```
#include "Tpetra_DefaultPlatform.hpp"
#include "Tpetra_Map.hpp"
#include "Tpetra_MultiVector.hpp"
#include "Tpetra_Vector.hpp"
#include "Tpetra_CrsMatrix.hpp"
#include "Teuchos_SerialDenseMatrix.hpp"
#include "Tpetra_Export.hpp"
#include "Tpetra_Operator.hpp"

#include "BelosLinearProblem.hpp"
#include "BelosBlockCGSolMgr.hpp"
#include "BelosBlockGmresSolMgr.hpp"
#include "BelosTpetraAdapter.hpp"

#include "LSQRSolMgr.hpp"
#include "MRNSDSolMgr.hpp"

using Tpetra::MultiVector;
using Tpetra::Operator;

// Explicitly invert a four-by-four matrix.
/*
  \param mat - (In) Four-by-four matrix to invert.
  \param inv - (Out) Inverse of the matrix.
*/
template <class Scalar, class Ordinal>
void invert(const Teuchos::SerialDenseMatrix< Ordinal, Scalar > &mat, Teuchos::
    SerialDenseMatrix< Ordinal, Scalar > &inv);

// Convert a quaternion to a four-by-four matrix.
/*
  \param outtrans1 - (Out) Matrix formed from the quaternion.
  \param w, x, y, z - (In) Elements of the quaternion.
  \param tx, ty, tz - (In) Elements of the quaternion.
*/
template <class Scalar, class Ordinal>
void quatToMatrix(Teuchos::SerialDenseMatrix< Ordinal, Scalar > &outtrans1, Scalar w,
    Scalar x, Scalar y, Scalar z, const Scalar tx, const Scalar ty, const Scalar tz);

// Read duration information from a CSV file.  Each processor has its own copy
// of the duration information.
/*
  \param file - (In) File in CSV form containing duration information.
  \param durationsVec - (Out) Vector to store the durations.
```

```
*/
template <class Ordinal>
void readDurations(const char *file, Teuchos::SerialDenseVector< Ordinal, Ordinal > &
    durationsVec);

//! Read motion information from a CSV file and create average motion matrix for
// each bin.
/*
  \param file - (In) File in CSV form containing motion information.
  \param motionsMat - (Out) Average quaternions for each bin on this processor.
  \param mapMotions - (In) Map for motion matrices.
  \param durationsVec - (In) Vector containing the durations.
  \param calib - (In) Calibration matrix.
  \param invInitial - (Out) Inverse of initial motion matrix.
  \param timeOffset - (In) Number of seconds' worth of information to discard.
  \param samplingRate - (In) Sampling rate of machine.  The product of
    samplingRate and timeOffset will be the number of motion data discarded at
    the beginning of the file.
*/
template <class Scalar, class Ordinal>
void readMotions(const char *file, Teuchos::SerialDenseMatrix< Ordinal, Scalar > &
    motionsMat,  const Teuchos::RCP<const Tpetra::Map<Ordinal> > &mapMotions, const
    Teuchos::SerialDenseVector< Ordinal, Ordinal > &durationsVec,  const Teuchos::
    SerialDenseMatrix< Ordinal, Scalar > &calib, Teuchos::SerialDenseMatrix< Ordinal,
    Scalar > &invInitial, const Ordinal timeOffset, const Ordinal samplingRate);

// Read noisy, blurred RHS data given in raw form (4 bytes per number).
/*
  \param file - (In) File in raw data form containing RHS.
  \param b - (Out) Vector to store data.
*/
template <class Scalar, class Ordinal>
void readData(const char *file, const Teuchos::RCP<Tpetra::Vector< Scalar, Ordinal,
    Ordinal > > &b);

// Write reconstructed solution in raw form (4 bytes per number).
/*
  \param file - (In) File to store solution in raw form.
  \param x - (In) Reconstructed solution.
*/
template <class Scalar, class Ordinal>
void writeData(const char *file, const Teuchos::RCP<const Tpetra::Vector< Scalar, Ordinal
    , Ordinal > > &x);

// Form system matrix A using nearest neighbor interpolation.
```

```
/*
  \param A - (Out) System matrix.
  \param splitMap - (In) Map describing distribution of b/x.
  \param motionsMat - (In) Average quaternions for each bin on this processor.
  \param durationsVec - (In) Vector containing the durations.
  \param calib - (In) Calibration matrix.
  \param invInitial - (In) Inverse of initial motion matrix.
  \param mapMotions - (In) Map for motion matrices.
  \param size_m1, size_m2, size_m3 - (In) Problem size.
*/
template <class Scalar, class Ordinal>
void formA(const Teuchos::RCP<Tpetra::CrsMatrix< Scalar, Ordinal > > &A, const Teuchos::
    RCP<const Tpetra::Map<Ordinal> > &splitMap, const Teuchos::SerialDenseMatrix< Ordinal
    , Scalar > &motionsMat, const Teuchos::SerialDenseVector< Ordinal, Ordinal > &
    durationsVec, const Teuchos::SerialDenseMatrix< Ordinal, Scalar > &calib, const
    Teuchos::SerialDenseMatrix< Ordinal, Scalar > &invInitial, const Teuchos::RCP<const
    Tpetra::Map<Ordinal> > &mapMotions, const Ordinal size_m1, const Ordinal size_m2,
    const Ordinal size_m3);

// Form system matrix A using trilinear interpolation.
/*
  \param A - (Out) System matrix.
  \param splitMap - (In) Map describing distribution of b/x.
  \param motionsMat - (In) Average quaternions for each bin on this processor.
  \param durationsVec - (In) Vector containing the durations.
  \param calib - (In) Calibration matrix.
  \param invInitial - (In) Inverse of initial motion matrix.
  \param mapMotions - (In) Map for motion matrices.
  \param size_m1, size_m2, size_m3 - (In) Problem size.
*/
template <class Scalar, class Ordinal>
void formATrilinear(const Teuchos::RCP<Tpetra::CrsMatrix< Scalar, Ordinal > > &A, const
    Teuchos::RCP<const Tpetra::Map<Ordinal> > &splitMap, const Teuchos::SerialDenseMatrix
    < Ordinal, Scalar > &motionsMat, const Teuchos::SerialDenseVector< Ordinal, Ordinal >
     &durationsVec, const Teuchos::SerialDenseMatrix< Ordinal, Scalar > &calib, const
    Teuchos::SerialDenseMatrix< Ordinal, Scalar > &invInitial, const Teuchos::RCP<const
    Tpetra::Map<Ordinal> > &mapMotions, const Ordinal size_m1, const Ordinal size_m2,
    const Ordinal size_m3);

// Solver linear system using LSQR.
/*
  \param A - (In) System matrix.
  \param X - (In/Out) Solution vector.
  \param B - (In) Data vector.
  \param Truth - (In) True solution vector.
```

```
  \param WindowSize - (In) Window size for least error status test if true
    solution is known.
  \param maxIters - (In) Maximum number of allowable iterations.
*/
template <class Scalar, class Ordinal>
void solveNewLSQR(const Teuchos::RCP<const Tpetra::CrsMatrix< Scalar, Ordinal > > &A,
    const Teuchos::RCP< Tpetra::Vector< Scalar, Ordinal, Ordinal > > &X, const Teuchos::
    RCP<const Tpetra::Vector< Scalar, Ordinal, Ordinal > > &B, const Teuchos::RCP< Tpetra
    ::Vector< Scalar, Ordinal > > &Truth = Teuchos::null, const int WindowSize = 1, const
     int maxIters = 10000);

// Solve linear system using MRNSD.
/*
  \param A - (In) System matrix.
  \param X - (In/Out) Solution vector.
  \param B - (In) Data vector.
  \param Truth - (In) True solution vector.
  \param WindowSize - (In) Window size for least error status test if true
    solution is known.
  \param Tolerance - (In) Tolerance level.
  \param maxIters - (In) Maximum number of allowable iterations.
*/
template <class Scalar, class Ordinal>
void solveNewMRNSD(const Teuchos::RCP<const Tpetra::CrsMatrix< Scalar, Ordinal > > &A,
    const Teuchos::RCP<Tpetra::Vector< Scalar, Ordinal, Ordinal > > &X, const Teuchos::
    RCP<const Tpetra::Vector< Scalar, Ordinal, Ordinal > > &B, const Teuchos::RCP< Tpetra
    ::Vector< Scalar, Ordinal, Ordinal > > &Truth = Teuchos::null, const int WindowSize =
     1, const Scalar Tolerance = 0.1, const int maxIters = 10000);

// Main method for reconstructing.
/*
  \param pl - (In) Parameter list containing all the necessary information.
*/
template <class Scalar, class Ordinal>
void reconstruct(const Teuchos::RCP< const Teuchos::ParameterList > &pl);

template <class Scalar, class Ordinal>
void invert(const Teuchos::SerialDenseMatrix< Ordinal, Scalar > &mat, Teuchos::
    SerialDenseMatrix< Ordinal, Scalar > &inv) {
  Scalar det = mat(0,3) * mat(1,2) * mat(2,1) * mat(3,0) -
    mat(0,2) * mat(1,3) * mat(2,1) * mat(3,0) -
    mat(0,3) * mat(1,1) * mat(2,2) * mat(3,0) +
    mat(0,1) * mat(1,3) * mat(2,2) * mat(3,0) +
    mat(0,2) * mat(1,1) * mat(2,3) * mat(3,0) -
    mat(0,1) * mat(1,2) * mat(2,3) * mat(3,0) -
```

```
  mat(0,3) * mat(1,2) * mat(2,0) * mat(3,1) +
  mat(0,2) * mat(1,3) * mat(2,0) * mat(3,1) +
  mat(0,3) * mat(1,0) * mat(2,2) * mat(3,1) -
  mat(0,0) * mat(1,3) * mat(2,2) * mat(3,1) -
  mat(0,2) * mat(1,0) * mat(2,3) * mat(3,1) +
  mat(0,0) * mat(1,2) * mat(2,3) * mat(3,1) +
  mat(0,3) * mat(1,1) * mat(2,0) * mat(3,2) -
  mat(0,1) * mat(1,3) * mat(2,0) * mat(3,2) -
  mat(0,3) * mat(1,0) * mat(2,1) * mat(3,2) +
  mat(0,0) * mat(1,3) * mat(2,1) * mat(3,2) +
  mat(0,1) * mat(1,0) * mat(2,3) * mat(3,2) -
  mat(0,0) * mat(1,1) * mat(2,3) * mat(3,2) -
  mat(0,2) * mat(1,1) * mat(2,0) * mat(3,3) +
  mat(0,1) * mat(1,2) * mat(2,0) * mat(3,3) +
  mat(0,2) * mat(1,0) * mat(2,1) * mat(3,3) -
  mat(0,0) * mat(1,2) * mat(2,1) * mat(3,3) -
  mat(0,1) * mat(1,0) * mat(2,2) * mat(3,3) +
  mat(0,0) * mat(1,1) * mat(2,2) * mat(3,3);

inv(0,0) = mat(1,2)*mat(2,3)*mat(3,1) - mat(1,3)*mat(2,2)*mat(3,1) + mat(1,3)*mat(2,1)*
    mat(3,2) - mat(1,1)*mat(2,3)*mat(3,2) - mat(1,2)*mat(2,1)*mat(3,3) + mat(1,1)*mat
    (2,2)*mat(3,3);
inv(0,1) = mat(0,3)*mat(2,2)*mat(3,1) - mat(0,2)*mat(2,3)*mat(3,1) - mat(0,3)*mat(2,1)*
    mat(3,2) + mat(0,1)*mat(2,3)*mat(3,2) + mat(0,2)*mat(2,1)*mat(3,3) - mat(0,1)*mat
    (2,2)*mat(3,3);
inv(0,2) = mat(0,2)*mat(1,3)*mat(3,1) - mat(0,3)*mat(1,2)*mat(3,1) + mat(0,3)*mat(1,1)*
    mat(3,2) - mat(0,1)*mat(1,3)*mat(3,2) - mat(0,2)*mat(1,1)*mat(3,3) + mat(0,1)*mat
    (1,2)*mat(3,3);
inv(0,3) = mat(0,1)*mat(1,3)*mat(2,2) + mat(0,2)*mat(1,1)*mat(2,3) + mat(0,3)*mat(1,2)*
    mat(2,1) - mat(0,1)*mat(1,2)*mat(2,3) - mat(0,2)*mat(1,3)*mat(2,1) - mat(0,3)*mat
    (1,1)*mat(2,2);

inv(1,0) = mat(1,3)*mat(2,2)*mat(3,0) - mat(1,2)*mat(2,3)*mat(3,0) - mat(1,3)*mat(2,0)*
    mat(3,2) + mat(1,0)*mat(2,3)*mat(3,2) + mat(1,2)*mat(2,0)*mat(3,3) - mat(1,0)*mat
    (2,2)*mat(3,3);
inv(1,1) = mat(0,2)*mat(2,3)*mat(3,0) - mat(0,3)*mat(2,2)*mat(3,0) + mat(0,3)*mat(2,0)*
    mat(3,2) - mat(0,0)*mat(2,3)*mat(3,2) - mat(0,2)*mat(2,0)*mat(3,3) + mat(0,0)*mat
    (2,2)*mat(3,3);
inv(1,2) = mat(0,3)*mat(1,2)*mat(3,0) - mat(0,2)*mat(1,3)*mat(3,0) - mat(0,3)*mat(1,0)*
    mat(3,2) + mat(0,0)*mat(1,3)*mat(3,2) + mat(0,2)*mat(1,0)*mat(3,3) - mat(0,0)*mat
    (1,2)*mat(3,3);
inv(1,3) = mat(0,0)*mat(1,2)*mat(2,3) + mat(0,2)*mat(1,3)*mat(2,0) + mat(0,3)*mat(1,0)*
    mat(2,2) - mat(0,0)*mat(1,3)*mat(2,2) - mat(0,2)*mat(1,0)*mat(2,3) - mat(0,3)*mat
    (1,2)*mat(2,0);
```

```
  inv(2,0) = mat(1,1)*mat(2,3)*mat(3,0) - mat(1,3)*mat(2,1)*mat(3,0) + mat(1,3)*mat(2,0)*
      mat(3,1) - mat(1,0)*mat(2,3)*mat(3,1) - mat(1,1)*mat(2,0)*mat(3,3) + mat(1,0)*mat
      (2,1)*mat(3,3);
  inv(2,1) = mat(0,3)*mat(2,1)*mat(3,0) - mat(0,1)*mat(2,3)*mat(3,0) - mat(0,3)*mat(2,0)*
      mat(3,1) + mat(0,0)*mat(2,3)*mat(3,1) + mat(0,1)*mat(2,0)*mat(3,3) - mat(0,0)*mat
      (2,1)*mat(3,3);
  inv(2,2) = mat(0,1)*mat(1,3)*mat(3,0) - mat(0,3)*mat(1,1)*mat(3,0) + mat(0,3)*mat(1,0)*
      mat(3,1) - mat(0,0)*mat(1,3)*mat(3,1) - mat(0,1)*mat(1,0)*mat(3,3) + mat(0,0)*mat
      (1,1)*mat(3,3);
  inv(2,3) = mat(0,0)*mat(1,3)*mat(2,1) + mat(0,1)*mat(1,0)*mat(2,3) + mat(0,3)*mat(1,1)*
      mat(2,0) - mat(0,0)*mat(1,1)*mat(2,3) - mat(0,1)*mat(1,3)*mat(2,0) - mat(0,3)*mat
      (1,0)*mat(2,1);

  inv(3,0) = mat(1,2)*mat(2,1)*mat(3,0) - mat(1,1)*mat(2,2)*mat(3,0) - mat(1,2)*mat(2,0)*
      mat(3,1) + mat(1,0)*mat(2,2)*mat(3,1) + mat(1,1)*mat(2,0)*mat(3,2) - mat(1,0)*mat
      (2,1)*mat(3,2);
  inv(3,1) = mat(0,1)*mat(2,2)*mat(3,0) - mat(0,2)*mat(2,1)*mat(3,0) + mat(0,2)*mat(2,0)*
      mat(3,1) - mat(0,0)*mat(2,2)*mat(3,1) - mat(0,1)*mat(2,0)*mat(3,2) + mat(0,0)*mat
      (2,1)*mat(3,2);
  inv(3,2) = mat(0,2)*mat(1,1)*mat(3,0) - mat(0,1)*mat(1,2)*mat(3,0) - mat(0,2)*mat(1,0)*
      mat(3,1) + mat(0,0)*mat(1,2)*mat(3,1) + mat(0,1)*mat(1,0)*mat(3,2) - mat(0,0)*mat
      (1,1)*mat(3,2);
  inv(3,3) = mat(0,0)*mat(1,1)*mat(2,2) + mat(0,1)*mat(1,2)*mat(2,0) + mat(0,2)*mat(1,0)*
      mat(2,1) - mat(0,0)*mat(1,2)*mat(2,1) - mat(0,1)*mat(1,0)*mat(2,2) - mat(0,2)*mat
      (1,1)*mat(2,0);
  inv.scale(1.0/det);
}

template <class Scalar, class Ordinal>
void quatToMatrix(Teuchos::SerialDenseMatrix< Ordinal, Scalar > &outtrans1, Scalar w,
    Scalar x, Scalar y, Scalar z, const Scalar tx, const Scalar ty, const Scalar tz) {
  Scalar qlen = sqrt(x*x + y*y + z*z + w*w);

  w /= qlen;
  x /= qlen;
  y /= qlen;
  z /= qlen;

  outtrans1(0,0) = w*w + x*x - y*y - z*z;
  outtrans1(0,1) = 2*x*y - 2*w*z;
  outtrans1(0,2) = 2*x*z + 2*w*y;
  outtrans1(0,3) = tx;

  outtrans1(1,0) = 2*x*y + 2*w*z;
  outtrans1(1,1) = w*w + y*y - x*x - z*z;
```

```
  outtrans1(1,2) = 2*y*z - 2*w*x;
  outtrans1(1,3) = ty;

  outtrans1(2,0) = 2*x*z - 2*w*y;
  outtrans1(2,1) = 2*y*z + 2*w*x;
  outtrans1(2,2) = w*w + z*z - x*x - y*y;
  outtrans1(2,3) = tz;

  outtrans1(3,0) = 0;
  outtrans1(3,1) = 0;
  outtrans1(3,2) = 0;
  outtrans1(3,3) = 1;
}

template <class Ordinal>
void readDurations(const char *file, Teuchos::SerialDenseVector< Ordinal, Ordinal > &
    durationsVec) {
  Ordinal dur;
  int i;
  char comma;
  std::ifstream DurationsFileName;
  DurationsFileName.open(file);
  if (DurationsFileName.fail()) {
    std::cerr << "Error: failed to open durations file: " << file << std::endl;
    exit(1);
  }
  DurationsFileName >> dur;
  durationsVec(0) = dur;
  for (i = 1; i < durationsVec.length(); i++) {
    DurationsFileName >> comma;
    DurationsFileName >> dur;
    durationsVec(i) = dur;
  }
  DurationsFileName.close();
}

template <class Scalar, class Ordinal>
void readMotions(const char *file, Teuchos::SerialDenseMatrix< Ordinal, Scalar > &
    motionsMat,  const Teuchos::RCP<const Tpetra::Map<Ordinal> > &mapMotions, const
    Teuchos::SerialDenseVector< Ordinal, Ordinal > &durationsVec,  const Teuchos::
    SerialDenseMatrix< Ordinal, Scalar > &calib, Teuchos::SerialDenseMatrix< Ordinal,
    Scalar > &invInitial, const Ordinal timeOffset, const Ordinal samplingRate) {

  std::string line;
  Scalar tempScalar;
```

```
char comma;
Ordinal i, j, k, skipLines = 0, curDuration;
std::ifstream MotionsFileName;
Scalar avgquat[7];
Teuchos::SerialDenseMatrix< Ordinal, Scalar >o1(4, 4, true);
Teuchos::SerialDenseMatrix< Ordinal, Scalar >o2(4, 4, true);
MotionsFileName.open(file);
if (MotionsFileName.fail()) {
  std::cerr << "Error: failed to open motions file: " << file << std::endl;
  exit(1);
}
// Remove first timeOffset*samplingRate lines
for (i = 0; i < timeOffset*samplingRate; i++) {
  getline(MotionsFileName, line);
}
// Now read the first motion info for the first position and find the inverse
for (i = 0; i < 7; i++) {
  avgquat[i] = 0.0;
}
for (i = 0; i < durationsVec[0]; i++) {
  getline(MotionsFileName, line);
  std::stringstream ss(line);
  for (j = 0; j < 3; j++) {
    ss >> tempScalar;
    ss >> comma;
  }
  ss >> comma >> comma >> comma;
  for (j = 0; j < 7; j++) {
    ss >> tempScalar;
    avgquat[j] += tempScalar;
    ss >> comma;
  }
}
for (i = 0; i < 7; i++) {
  avgquat[i] /= durationsVec[0];
}
quatToMatrix(o1, avgquat[0], avgquat[1], avgquat[2], avgquat[3], avgquat[4], avgquat
    [5], avgquat[6]);
o2.multiply(Teuchos::NO_TRANS, Teuchos::NO_TRANS, 1.0, calib, o1, 0.0);
invert<Scalar,Ordinal>(o2, invInitial);

/* Now, for each interval, find the average quaternion.  First skip the
 * correct number of rows by determining how many rows are on previous
 * processors.
 */
```

```
  for (i = 1; i < mapMotions->getMinGlobalIndex(); i++) {
    skipLines += durationsVec[i];
  }
  for (i = 0; i < skipLines; i++) {
    getline(MotionsFileName, line);
  }
  // Now get the average quaternion for each of its own intervals
  for (i = 0; i < motionsMat.numRows(); i++) {
    curDuration = mapMotions->getComm()->getRank()==0 ? durationsVec[i + 1] :
        durationsVec[i + mapMotions->getMinGlobalIndex()];
    for (j = 0; j < 7; j++) {
      avgquat[j] = 0.0;
    }
    for (j = 0; j < curDuration; j++) {
      getline(MotionsFileName, line);
      std::stringstream ss(line);
      for (k = 0; k < 3; k++) {
        ss >> tempScalar;
        ss >> comma;
      }
      ss >> comma >> comma >> comma;
      for (k = 0; k < 7; k++) {
        ss >> tempScalar;
        avgquat[k] += tempScalar;
        ss >> comma;
      }
    }
    for (j = 0; j < 7; j++) {
      avgquat[j] /= curDuration;
      motionsMat(i, j) = avgquat[j];
    }
  }
  MotionsFileName.close();
}

template <class Scalar, class Ordinal>
void readData(const char *file, const Teuchos::RCP<Tpetra::Vector< Scalar, Ordinal,
    Ordinal > > &b) {
  Ordinal NumMyElements_target;
  if (b->getMap()->getComm()->getRank()==0) {
    NumMyElements_target = b->getGlobalLength();
  }
  else {
    NumMyElements_target = 0;
  }
```

```
  Teuchos::RCP<Tpetra::Map< Ordinal > > TargetMap = Teuchos::rcp( new Tpetra::Map<
      Ordinal > (Teuchos::OrdinalTraits<Ordinal>::invalid(), NumMyElements_target, b->
      getMap()->getIndexBase(), b->getMap()->getComm()) );
  Teuchos::RCP<Tpetra::Vector < Scalar, Ordinal, Ordinal > > Y = Teuchos::rcp( new Tpetra
      ::Vector< Scalar, Ordinal, Ordinal > (TargetMap) );
  std::ifstream BlurredImageFileName;
  float* tempNumArray = new float[NumMyElements_target];

  if (b->getMap()->getComm()->getRank()==0) {
    BlurredImageFileName.open(file, std::ios::out | std::ios::binary);
    BlurredImageFileName.read((char*)tempNumArray, NumMyElements_target*sizeof(float));
    for (int i = 0; i < NumMyElements_target; i++) {
      Y->replaceGlobalValue(i, tempNumArray[i]);
    }
    BlurredImageFileName.close();
  }
  Tpetra::Export< Ordinal > Exporter(TargetMap, b->getMap());
  b->doExport(*Y, Exporter, Tpetra::INSERT);
}

template <class Scalar, class Ordinal>
void writeData(const char *file, const Teuchos::RCP<const Tpetra::Vector< Scalar, Ordinal
    , Ordinal > > &x) {
  Ordinal NumMyElements_target;
  if (x->getMap()->getComm()->getRank()==0) {
    NumMyElements_target = x->getGlobalLength();
  }
  else {
    NumMyElements_target = 0;
  }
  Teuchos::RCP<Tpetra::Map< Ordinal > > TargetMap = Teuchos::rcp( new Tpetra::Map<
      Ordinal > (Teuchos::OrdinalTraits<Ordinal>::invalid(), NumMyElements_target, x->
      getMap()->getIndexBase(), x->getMap()->getComm()) );
  Tpetra::Export< Ordinal > Exporter(x->getMap(), TargetMap);
  Teuchos::RCP<Tpetra::Vector < Scalar, Ordinal, Ordinal > > Y = Teuchos::rcp( new Tpetra
      ::Vector< Scalar, Ordinal, Ordinal > (TargetMap) );
  Y->doExport(*x, Exporter, Tpetra::INSERT);
  std::ofstream SolutionFileName;
  if (x->getMap()->getComm()->getRank()==0) {
    Teuchos::ArrayRCP<const Scalar> yView = Y->get1dView();
    Ordinal i;
    float temp;
    SolutionFileName.open(file, std::ios::out | std::ios::binary);
    for (i = 0; i < x->getGlobalLength(); i++) {
      temp = (float) yView[i];
```

```
      SolutionFileName.write((char*)&temp, sizeof(float));
    }
    SolutionFileName.close();
  }
}

template <class Scalar, class Ordinal>
void formA(const Teuchos::RCP<Tpetra::CrsMatrix< Scalar, Ordinal > > &A, const Teuchos::
    RCP<const Tpetra::Map<Ordinal> > &splitMap, const Teuchos::SerialDenseMatrix< Ordinal
    , Scalar > &motionsMat, const Teuchos::SerialDenseVector< Ordinal, Ordinal > &
    durationsVec, const Teuchos::SerialDenseMatrix< Ordinal, Scalar > &calib, const
    Teuchos::SerialDenseMatrix< Ordinal, Scalar > &invInitial, const Teuchos::RCP<const
    Tpetra::Map<Ordinal> > &mapMotions, const Ordinal size_m1, const Ordinal size_m2,
    const Ordinal size_m3) {
  Scalar weight;
  Ordinal x, y, z;
  Ordinal i, j, k;
  Ordinal sumDurations = 0;
  Ordinal dataSize = size_m1 * size_m2 * size_m3;
  Teuchos::SerialDenseMatrix< Ordinal, Scalar > outtrans1(4, 4, false);
  Teuchos::SerialDenseMatrix< Ordinal, Scalar > outtrans2(4, 4, false);
  Teuchos::SerialDenseMatrix< Ordinal, Scalar > outtrans(4, 4, false);
  Teuchos::SerialDenseMatrix< Ordinal, Scalar > inv(4, 4, true);
  Teuchos::SerialDenseMatrix< Ordinal, Scalar >P(dataSize, 4, false);
  Teuchos::SerialDenseMatrix< Ordinal, Scalar >W(dataSize, 4, false);
  for (i = 0; i < durationsVec.length(); i++) {
    sumDurations += durationsVec[i];
  }
  if ( A->getComm()->getRank()==0) {
    weight = (1.0*durationsVec(0)) / sumDurations;
    for (i = 0; i < dataSize; i++) {
      A->insertGlobalValues(i, Teuchos::ArrayView<const Ordinal>(&i, 1), Teuchos::
          ArrayView<const Scalar>(&weight, 1));
    }
  }
  for (k = 0; k < size_m3; k++) {
    for (j = 0; j < size_m2; j++) {
      for (i = 0; i < size_m1; i++) {
        P(k*(size_m1 * size_m2) + j*size_m1 + i, 0) = j;
        P(k*(size_m1 * size_m2) + j*size_m1 + i, 1) = i;
        P(k*(size_m1 * size_m2) + j*size_m1 + i, 2) = k;
        P(k*(size_m1 * size_m2) + j*size_m1 + i, 3) = 1.0;
      }
    }
  }
```

```
  // Each processor now needs to form its Ai matrices.
  for (i = 0; i < motionsMat.numRows(); i++) {
    quatToMatrix(outtrans1, motionsMat(i, 0), motionsMat(i, 1), motionsMat(i, 2),
        motionsMat(i, 3), motionsMat(i, 4), motionsMat(i, 5), motionsMat(i, 6));
    outtrans2.multiply(Teuchos::NO_TRANS, Teuchos::NO_TRANS, 1, calib, outtrans1, 0);
    outtrans.multiply(Teuchos::NO_TRANS, Teuchos::NO_TRANS, 1, outtrans2, invInitial, 0);
    invert<Scalar,Ordinal>(outtrans, inv);
    W.multiply(Teuchos::NO_TRANS, Teuchos::TRANS, 1, P, inv, 0);
    weight = (1.0*(A->getComm()->getRank()==0 ? durationsVec[i + 1] : durationsVec[i +
        mapMotions->getMinGlobalIndex()])) / sumDurations;
    for (j = 0; j < W.numRows(); j++) {
      x = (Ordinal) round(W(j,1));
      y = (Ordinal) round(W(j,0));
      z = (Ordinal) round(W(j,2));
      if (x >= 0 && x < size_m2 && y >= 0 && y < size_m1 && z >= 0 && z < size_m3) {
        k = x + y*size_m1 + z*size_m1*size_m2;
        A->insertGlobalValues(j, Teuchos::ArrayView<const Ordinal>(&k, 1), Teuchos::
            ArrayView<const Scalar>(&weight, 1));
      }
    }
  }
  // Needed in case no entries are inserted on some processor
  k = 0; weight = 0;
  A->insertGlobalValues(0, Teuchos::ArrayView<const Ordinal>(&k, 1), Teuchos::ArrayView<
      const Scalar>(&weight, 1));
  A->fillComplete(splitMap, splitMap);
}

template <class Scalar, class Ordinal>
void formATrilinear(const Teuchos::RCP<Tpetra::CrsMatrix< Scalar, Ordinal > > &A, const
    Teuchos::RCP<const Tpetra::Map<Ordinal> > &splitMap, const Teuchos::SerialDenseMatrix
    < Ordinal, Scalar > &motionsMat, const Teuchos::SerialDenseVector< Ordinal, Ordinal >
     &durationsVec, const Teuchos::SerialDenseMatrix< Ordinal, Scalar > &calib, const
    Teuchos::SerialDenseMatrix< Ordinal, Scalar > &invInitial, const Teuchos::RCP<const
    Tpetra::Map<Ordinal> > &mapMotions, const Ordinal size_m1, const Ordinal size_m2,
    const Ordinal size_m3) {
  Scalar weight;
  Ordinal x, y, z;
  Scalar xi, eta, gam;
  Ordinal i, j, k;
  Ordinal sumDurations = 0;
  Ordinal dataSize = size_m1 * size_m2 * size_m3;
  Ordinal cols[8];
  Scalar vals[8];
```

```
Teuchos::SerialDenseMatrix< Ordinal, Scalar > outtrans1(4, 4, false);
Teuchos::SerialDenseMatrix< Ordinal, Scalar > outtrans2(4, 4, false);
Teuchos::SerialDenseMatrix< Ordinal, Scalar > outtrans(4, 4, false);
Teuchos::SerialDenseMatrix< Ordinal, Scalar > inv(4, 4, true);
Teuchos::SerialDenseMatrix< Ordinal, Scalar >P(dataSize, 4, false);
Teuchos::SerialDenseMatrix< Ordinal, Scalar >W(dataSize, 4, false);
for (i = 0; i < durationsVec.length(); i++) {
  sumDurations += durationsVec[i];
}
if ( A->getComm()->getRank()==0) {
  weight = (1.0*durationsVec(0)) / sumDurations;
  for (i = 0; i < dataSize; i++) {
    A->insertGlobalValues(i, Teuchos::ArrayView<const Ordinal>(&i, 1), Teuchos::
        ArrayView<const Scalar>(&weight, 1));
  }
}
for (k = 0; k < size_m3; k++) {
  for (j = 0; j < size_m2; j++) {
    for (i = 0; i < size_m1; i++) {
      P(k*(size_m1 * size_m2) + j*size_m1 + i, 0) = j;
      P(k*(size_m1 * size_m2) + j*size_m1 + i, 1) = i;
      P(k*(size_m1 * size_m2) + j*size_m1 + i, 2) = k;
      P(k*(size_m1 * size_m2) + j*size_m1 + i, 3) = 1.0;
    }
  }
}

// Each processor now needs to form its Ai matrices
for (i = 0; i < motionsMat.numRows(); i++) {
  quatToMatrix(outtrans1, motionsMat(i, 0), motionsMat(i, 1), motionsMat(i, 2),
      motionsMat(i, 3), motionsMat(i, 4), motionsMat(i, 5), motionsMat(i, 6));
  outtrans2.multiply(Teuchos::NO_TRANS, Teuchos::NO_TRANS, 1, calib, outtrans1, 0);
  outtrans.multiply(Teuchos::NO_TRANS, Teuchos::NO_TRANS, 1, outtrans2, invInitial, 0);
  invert<Scalar,Ordinal>(outtrans, inv);
  W.multiply(Teuchos::NO_TRANS, Teuchos::TRANS, 1, P, inv, 0);
  weight = (1.0*(A->getComm()->getRank()==0 ? durationsVec[i + 1] : durationsVec[i +
      mapMotions->getMinGlobalIndex()])) / sumDurations;
  for (j = 0; j < W.numRows(); j++) {
    x = (Ordinal) floor(W(j,1));
    xi = W(j,1) - x;
    y = (Ordinal) floor(W(j,0));
    eta = W(j,0) - y;
    z = (Ordinal) floor(W(j,2));
    gam = W(j,2) - z;
```

```
      if (x >= 0 && x < (size_m2-1) && y >= 0 && y < (size_m1-1) && z >= 0 && z < (
          size_m3-1) ) {
        cols[0] = x + y*size_m1 + z*size_m1*size_m2;
        cols[1] = (x+1) + y*size_m1 + z*size_m1*size_m2;
        cols[2] = x + (y+1)*size_m1 + z*size_m1*size_m2;
        cols[3] = x + y*size_m1 + (z+1)*size_m1*size_m2;
        cols[4] = (x+1) + y*size_m1 + (z+1)*size_m1*size_m2;
        cols[5] = x + (y+1)*size_m1 + (z+1)*size_m1*size_m2;
        cols[6] = (x+1) + (y+1)*size_m1 + z*size_m1*size_m2;
        cols[7] = (x+1) + (y+1)*size_m1 + (z+1)*size_m1*size_m2;
        vals[0] = (1-xi)*(1-eta)*(1-gam)*weight;
        vals[1] = xi*(1-eta)*(1-gam)*weight;
        vals[2] = (1-xi)*eta*(1-gam)*weight;
        vals[3] = (1-xi)*(1-eta)*gam*weight;
        vals[4] = xi*(1-eta)*gam*weight;
        vals[5] = (1-xi)*eta*gam*weight;
        vals[6] = xi*eta*(1-gam)*weight;
        vals[7] = xi*eta*gam*weight;
        A->insertGlobalValues(j, Teuchos::ArrayView<const Ordinal>(cols, 8), Teuchos::
            ArrayView<const Scalar>(vals, 8));
      }
      else if (x >= 0 && x < (size_m2-1) && y >= 0 && y < (size_m1-1) && z == (size_m3-1)
          ) { // include last image in stack
        cols[0] = x + y*size_m1 + z*size_m1*size_m2;
        cols[1] = (x+1) + y*size_m1 + z*size_m1*size_m2;
        cols[2] = x + (y+1)*size_m1 + z*size_m1*size_m2;
        cols[3] = (x+1) + (y+1)*size_m1 + z*size_m1*size_m2;
        vals[0] = (1-xi)*(1-eta)*(1-gam)*weight;
        vals[1] = xi*(1-eta)*(1-gam)*weight;
        vals[2] = (1-xi)*eta*(1-gam)*weight;
        vals[3] = xi*eta*(1-gam)*weight;
        A->insertGlobalValues(j, Teuchos::ArrayView<const Ordinal>(cols, 4), Teuchos::
            ArrayView<const Scalar>(vals, 4));
      }
    }
  }
  // Needed in case no entries are inserted on some processor
  k = 0; weight = 0;
  A->insertGlobalValues(0, Teuchos::ArrayView<const Ordinal>(&k, 1), Teuchos::ArrayView<
      const Scalar>(&weight, 1));
  A->fillComplete(splitMap, splitMap);
}

template <class Scalar, class Ordinal>
```

```
void solveNewLSQR(const Teuchos::RCP<const Tpetra::CrsMatrix< Scalar, Ordinal > > &A,
    const Teuchos::RCP< Tpetra::Vector< Scalar, Ordinal, Ordinal > > &X, const Teuchos::
    RCP<const Tpetra::Vector< Scalar, Ordinal, Ordinal > > &B, const Teuchos::RCP< Tpetra
    ::Vector< Scalar, Ordinal > > &Truth /* = Teuchos::null */, const int WindowSize /* =
     1 */, const int maxIters /* = 10000 */) {
  typedef Tpetra::MultiVector< Scalar, Ordinal, Ordinal > MV;
  typedef Tpetra::Operator< Scalar, Ordinal > OP;

  Teuchos::RCP< Belos::LinearProblem< Scalar, MV, OP > >myProblem = Teuchos::rcp(new
      Belos::LinearProblem< Scalar, MV, OP>(A, X, B));
  bool set = myProblem->setProblem();
  if (A->getComm()->getRank()==0 && !set)
    std::cout << "ERROR! LSQR not set" << std::endl;

  Teuchos::RCP< Teuchos::ParameterList > pl = Teuchos::rcp(new Teuchos::ParameterList());
  pl->set("Adaptive Block Size", false);
  pl->set("Maximum Iterations", (Ordinal) maxIters);
  pl->set("Condition Limit", (Scalar) 1000);
  pl->set("Term Iter Max", 1);
  pl->set("Rel Mat Err", 0.0);
  if (Truth != Teuchos::null) {
    pl->set< Teuchos::RCP < MV > >("True Solution", Truth);
    pl->set("Window Size", WindowSize);
  }

  LSQRSolMgr< Scalar, MV, OP > solver( myProblem, pl );
  solver.solve();
  Ordinal numIters = solver.getNumIters();
  if (A->getComm()->getRank()==0)
    std::cout << "LSQR num iters = " << numIters << std::endl;
}

template <class Scalar, class Ordinal>
void solveNewMRNSD(const Teuchos::RCP<const Tpetra::CrsMatrix< Scalar, Ordinal > > &A,
    const Teuchos::RCP<Tpetra::Vector< Scalar, Ordinal, Ordinal > > &X, const Teuchos::
    RCP<const Tpetra::Vector< Scalar, Ordinal, Ordinal > > &B, const Teuchos::RCP< Tpetra
    ::Vector< Scalar, Ordinal, Ordinal > > &Truth /* = Teuchos::null */, const int
    WindowSize /* = 1 */, const Scalar Tolerance /* = 0.1 */, const int maxIters /* =
    10000 */) {
  typedef Tpetra::MultiVector< Scalar, Ordinal, Ordinal > MV;
  typedef Tpetra::Operator< Scalar, Ordinal > OP;

  Teuchos::RCP< Belos::LinearProblem< Scalar, MV, OP > >myProblem = Teuchos::rcp(new
      Belos::LinearProblem< Scalar, MV, OP>(A, X, B));
  bool set = myProblem->setProblem();
```

```
  if (A->getComm()->getRank()==0 && !set)
    std::cout << "ERROR! MRNSD not set" << std::endl;

  Teuchos::RCP< Teuchos::ParameterList > pl = Teuchos::rcp(new Teuchos::ParameterList());
  pl->set("Maximum Iterations", (Ordinal) maxIters);
  pl->set("Tolerance", Tolerance);
  if (Truth != Teuchos::null) {
    pl->set< Teuchos::RCP< MV > >("True Solution", Truth);
    pl->set("Window Size", WindowSize);
  }

  MRNSDSolMgr< Scalar, MV, OP > solver( myProblem, pl );
  solver.solve();
  Ordinal numIters = solver.getNumIters();
  if (A->getComm()->getRank()==0)
    std::cout << "MRNSD num iters = " << numIters << std::endl;
}

template <class Scalar, class Ordinal>
void reconstruct(const Teuchos::RCP< const Teuchos::ParameterList > &pl) {
  Teuchos::RCP<const Teuchos::Comm<int> > comm = Tpetra::DefaultPlatform::
      getDefaultPlatform().getComm();

  size_t myRank = comm->getRank();
  size_t numProc = comm->getSize();
  bool verbose = (myRank==0);

  const Ordinal indexBase = 0;
  const Ordinal quatSize = 7;
  const Ordinal calibSize = 4;
  Ordinal motionSize;
  Ordinal numDurations;
  const Ordinal size_m1 = pl->get<Ordinal>("nx");
  const Ordinal size_m2 = pl->get<Ordinal>("ny");
  const Ordinal size_m3 = pl->get<Ordinal>("nz");
  const Ordinal dataSize = size_m1*size_m2*size_m3;

  // Maps
  Teuchos::RCP<const Tpetra::Map<Ordinal> > splitMap;
  Teuchos::RCP<const Tpetra::Map<Ordinal> > allMap;
  Teuchos::RCP<const Tpetra::Map<Ordinal> > mapMotions;

  // Matrices and vectors
  Teuchos::RCP< Tpetra::CrsMatrix< Scalar, Ordinal > > A;
  Teuchos::SerialDenseMatrix< Ordinal, Scalar > motionsMat;
```

```
Teuchos::SerialDenseVector< Ordinal, Ordinal > durationsVec;
Teuchos::RCP<Tpetra::Vector< Scalar, Ordinal, Ordinal > > B;
Teuchos::RCP<Tpetra::Vector< Scalar, Ordinal, Ordinal > > X;
Teuchos::SerialDenseMatrix< Ordinal, Scalar > invInitial(4, 4, true);
Teuchos::RCP<Tpetra::Vector< Scalar, Ordinal, Ordinal > > Truth;

// Read Calibration Matrix
Teuchos::SerialDenseMatrix< Ordinal, Scalar >calib(calibSize, calibSize, true);
std::ifstream CalibFileName;
Scalar num;
std::string fname = pl->get<std::string>("Calibration File");
CalibFileName.open(fname.c_str());
if (CalibFileName.fail()) {
  std::cout << "Error: failed to open calibration file: " << pl->get<const char*>("
      Calibration File") << std::endl;
  exit(1);
}
if (pl->get<std::string>("Calibration Orientation").compare("Row") == 0) {
  for (int i = 0; i < calibSize; i++) {
    for (int j = 0; j < calibSize; j++) {
      CalibFileName >> num;
      calib(i,j) = num;
    }
  }
} else {
  for (int j = 0; j < calibSize; j++) {
    for (int i = 0; i < calibSize; i++) {
      CalibFileName >> num;
      calib(i,j) = num;
    }
  }
}
CalibFileName.close();

// Read Durations
numDurations = pl->get<Ordinal>("Number of Intervals");
durationsVec = Teuchos::SerialDenseVector< Ordinal, Ordinal >(numDurations, false);
readDurations<Ordinal>(pl->get<std::string>("Interval File").c_str(), durationsVec);

// Read Motions
mapMotions = Teuchos::rcp( new Tpetra::Map<Ordinal>(numDurations, indexBase, comm) );
motionsMat = Teuchos::SerialDenseMatrix< Ordinal, Scalar >(verbose ? mapMotions->
    getNodeNumElements() - 1 : mapMotions->getNodeNumElements(), quatSize, false);
readMotions<Scalar,Ordinal>(pl->get<std::string>("Motion File").c_str(), motionsMat,
    mapMotions, durationsVec, calib, invInitial, pl->get<Ordinal>("Motion Offset"), pl
```

```
    ->get<Ordinal>("Sampling Rate"));

// First make allMap, then form A
std::vector<Ordinal> indices(dataSize);
for (Ordinal i = 0; i < dataSize; i++) {
  indices[i] = i;
}
allMap = Teuchos::rcp( new Tpetra::Map<Ordinal>(Teuchos::OrdinalTraits<Ordinal>::
    invalid(), Teuchos::ArrayView<const Ordinal>(indices), indexBase, comm) );
splitMap = Teuchos::rcp( new Tpetra::Map<Ordinal>(dataSize, indexBase, comm) );

if (pl->isParameter("Interpolation")) {
  if (pl->get<std::string>("Interpolation") == "Trilinear") {
    A = Teuchos::rcp( new Tpetra::CrsMatrix< Scalar, Ordinal, Ordinal >(allMap, allMap,
         numDurations*8) );
    formATrilinear<Scalar,Ordinal>(A, splitMap, motionsMat, durationsVec, calib,
        invInitial, mapMotions, size_m1, size_m2, size_m3);
  } else if (pl->get<std::string>("Interpolation") == "Nearest Neighbor") {
    A = Teuchos::rcp( new Tpetra::CrsMatrix< Scalar, Ordinal, Ordinal >(allMap, allMap,
         numDurations) );
    formA<Scalar,Ordinal>(A, splitMap, motionsMat, durationsVec, calib, invInitial,
        mapMotions, size_m1, size_m2, size_m3);
  } else {
    if (verbose)
      std::cerr << "Error: undefined Interpolation type: " << pl->get<std::string>("
          Interpolation") << std::endl;
    exit(1);
  }
} else { // no interpolation given; used NN
  A = Teuchos::rcp( new Tpetra::CrsMatrix< Scalar, Ordinal, Ordinal >(allMap, allMap,
      numDurations) );
  formA<Scalar,Ordinal>(A, splitMap, motionsMat, durationsVec, calib, invInitial,
      mapMotions, size_m1, size_m2, size_m3);
}

// Read Data
B = Teuchos::rcp( new Tpetra::Vector< Scalar, Ordinal, Ordinal >(splitMap, false) );
readData<Scalar,Ordinal>(pl->get<std::string>("Input File").c_str(), B);

// Read Truth, if available
if (pl->isParameter("True Solution File")) {
  Truth = Teuchos::rcp( new Tpetra::Vector< Scalar, Ordinal, Ordinal >(splitMap, false)
       );
  readData<Scalar,Ordinal>(pl->get<std::string>("True Solution File").c_str(), Truth);
} else {
```

```
    Truth = Teuchos::null;
  }

  X = Teuchos::rcp( new Tpetra::Vector< Scalar, Ordinal, Ordinal >(splitMap, true) );
  Teuchos::RCP<Tpetra::Vector< Scalar, Ordinal, Ordinal > > diff_mv = Teuchos::rcp( new
      Tpetra::Vector< Scalar, Ordinal, Ordinal > (splitMap, false) );
  // Solve
  if (pl->isSublist("LSQR Solver")) {
    solveNewLSQR<Scalar,Ordinal>(A, X, B, Truth, pl->sublist("LSQR Solver").get<Ordinal>(
        "Window Size"), pl->sublist("LSQR Solver").get<Ordinal>("Maximum Iterations") );
    fname = pl->get<std::string>("LSQR Output File");
    writeData<Scalar,Ordinal>(fname.c_str(), X);
  }
  X->putScalar( (Scalar) 0.0 );
  if (pl->isSublist("MRNSD Solver")) {
    solveNewMRNSD<Scalar,Ordinal>(A, X, B, Truth, pl->sublist("MRNSD Solver").get<Ordinal
        >("Window Size"), (Scalar) pl->sublist("MRNSD Solver").get<double>("Tolerance"),
        pl->sublist("MRNSD Solver").get<Ordinal>("Maximum Iterations") );
    fname = pl->get<std::string>("MRNSD Output File");
    writeData<Scalar,Ordinal>(fname.c_str(), X);
  }
}
```

### A.4.2 HRRTmain.cpp Code

```
#include "HRRT.hpp"
#include <time.h>
#include "Teuchos_XMLParameterListHelpers.hpp"

int main(int argc, char *argv[]) {
  time_t time0 = time(NULL);
  typedef int Ordinal;
  typedef float Scalar;

  Teuchos::oblackholestream blackhole;
  Teuchos::GlobalMPISession mpiSession(&argc,&argv,&blackhole);
  Teuchos::RCP< Teuchos::ParameterList > pl = Teuchos::getParametersFromXmlFile(argv[1]);
  reconstruct<Scalar,Ordinal>(pl);
  time_t time1 = time(NULL);
  std::cout << "Total time = " << time1 - time0 << " seconds " << std::endl;
  return 0;
}
```

### A.4.3 Example XML File

```
<ParameterList>
```

```
  <Parameter name="Motion File" type="string" value="motions.csv"/>
  <Parameter name="Number of Intervals" type="int" value="17"/>
  <Parameter name="Interval File" type="string" value="intervals.csv"/>
  <Parameter name="Input File" type="string" value="rhs.img"/>
  <Parameter name="True Solution File" type="string" value="true.img"/>
  <Parameter name="Motion Offset" type="int" value="0"/>
  <Parameter name="Sampling Rate" type="int" value="20"/>
  <Parameter name="nx" type="int" value="128"/>
  <Parameter name="ny" type="int" value="128"/>
  <Parameter name="nz" type="int" value="48"/>
  <Parameter name="Amount of Noise" type="float" value="0.10"/>
  <Parameter name="Scalar Type" type="string" value="float"/>
  <Parameter name="Ordinal Type" type="string" value="int"/>
  <Parameter name="Interpolation" type="string" value="Trilinear"/>
  <Parameter name="Calibration File" type="string" value="calib.txt"/>
  <Parameter name="Calibration Orientation" type="string" value="Row"/>
  <Parameter name="LSQR Output File" type="string" value="LSQRsol.img"/>
  <Parameter name="MRNSD Output File" type="string" value="MRNSDsol.img"/>
  <ParameterList name="LSQR Solver">
    <Parameter name="Maximum Iterations" type="int" value="500"/>
    <Parameter name="Condition Limit" type="float" value="1000"/>
    <Parameter name="Term Iter Max" type="int" value="1"/>
    <Parameter name="Rel RHS Err" type="float" value="0.0"/>
    <Parameter name="Rel Mat Err" type="float" value="0.0"/>
    <Parameter name="Window Size" type="int" value="4"/>
  </ParameterList>
  <ParameterList name="MRNSD Solver">
    <Parameter name="Maximum Iterations" type="int" value="1000"/>
    <Parameter name="Tolerance" type="float" value="0.0"/>
    <Parameter name="Window Size" type="int" value="8"/>
  </ParameterList>
</ParameterList>
```

## A.5 Code for AO Application

### A.5.1 AOOperator.hpp

```
#ifndef AO_OPERATOR_HPP
#define AO_OPERATOR_HPP

/* AOOperator.hpp
 * Concrete class representing the Adaptive Optics operator.
 */

#include "Tpetra_Operator.hpp"
```

```
#include "Teuchos_SerialDenseMatrix.hpp"
#include "Tpetra_Map.hpp"

template<class Scalar, class LocalOrdinal = int, class GlobalOrdinal = LocalOrdinal,
    class Node = Kokkos::DefaultNode::DefaultNodeType>
class AOOperator : public Tpetra::Operator<Scalar,LocalOrdinal,GlobalOrdinal,Node> {

  public:
  // Constructors/Destructor
  /* AOOperator constructor with mask.
   * This class is templated on Scalar, LocalOrdinal, GlobalOrdinal and Node.
   * The LocalOrdinal type, if omitted, defaults to int.
   * The GlobalOrdinal type defaults to the LocalOrdinal type.
   * The Node type defaults to the default node in Kokkos.
   * This constructor takes a Teuchos::SerialDenseMatrix describing the mask
   * for this operator.  The mask is of size (n-1)xn, so the problem size is
   * known.  It also takes a pointer to the domain and range maps.
   */
  AOOperator( const Teuchos::RCP<Teuchos::SerialDenseMatrix<LocalOrdinal, Scalar> > &mask
      , const Scalar alpha, const Teuchos::RCP<const Tpetra::Map<LocalOrdinal,
      GlobalOrdinal,Node> > &domainMap, const Teuchos::RCP<const Tpetra::Map<LocalOrdinal
      ,GlobalOrdinal,Node> > &rangeMap);

  // Destructor.
  ~AOOperator() {};

  // Getter methods
  const Teuchos::RCP< const Tpetra::Map<LocalOrdinal,GlobalOrdinal,Node> > & getDomainMap
      () const { return dMap_; }

  const Teuchos::RCP< const Tpetra::Map<LocalOrdinal,GlobalOrdinal,Node> > & getRangeMap
      () const { return rMap_; }

  bool hasTransposeApply() const { return true; }

  // Apply method
  /* This method applies the adaptive optics operator, or its transpose, to a
   * vector X.  It scales the result by alpha, then adds that to beta*Y, storing
   * the result in Y.  Thus, Y = alpha*AO^{mode}*X + beta*Y.
   */
  void apply(const Tpetra::MultiVector<Scalar,LocalOrdinal,GlobalOrdinal,Node> &X, Tpetra
      ::MultiVector<Scalar,LocalOrdinal,GlobalOrdinal,Node> &Y, Teuchos::ETransp mode=
      Teuchos::NO_TRANS, Scalar alpha = Teuchos::ScalarTraits<Scalar>::one(), Scalar beta
       = Teuchos::ScalarTraits<Scalar>::zero()) const;
```

```
private:
  // H and F matrices
  Teuchos::RCP<Teuchos::SerialDenseMatrix<LocalOrdinal, Scalar> > H_;
  Teuchos::RCP<Teuchos::SerialDenseMatrix<LocalOrdinal, Scalar> > F_;
  // Mask matrix
  Teuchos::RCP<Teuchos::SerialDenseMatrix<LocalOrdinal, Scalar> > mask_;
  // Domain and range maps
  Teuchos::RCP<const Tpetra::Map<LocalOrdinal,GlobalOrdinal,Node> > dMap_;
  Teuchos::RCP<const Tpetra::Map<LocalOrdinal,GlobalOrdinal,Node> > rMap_;
  // Alpha value
  Scalar alpha_;
  // Size
  LocalOrdinal n_;
};

template<class Scalar, class LocalOrdinal, class GlobalOrdinal, class Node>
AOOperator<Scalar,LocalOrdinal,GlobalOrdinal,Node>::AOOperator( const Teuchos::RCP<
    Teuchos::SerialDenseMatrix<LocalOrdinal, Scalar> > &mask, const Scalar alpha, const
    Teuchos::RCP<const Tpetra::Map<LocalOrdinal,GlobalOrdinal,Node> > &domainMap, const
    Teuchos::RCP<const Tpetra::Map<LocalOrdinal,GlobalOrdinal,Node> > &rangeMap ) :
  mask_(mask),
  alpha_(alpha),
  dMap_(domainMap),
  rMap_(rangeMap),
  n_(mask_->numRows() + 1)
{
  H_ = Teuchos::rcp( new Teuchos::SerialDenseMatrix<LocalOrdinal, Scalar>(n_ - 1, n_) );
  F_ = Teuchos::rcp( new Teuchos::SerialDenseMatrix<LocalOrdinal, Scalar>(n_ - 1, n_) );

  const Scalar one = Teuchos::ScalarTraits<Scalar>::one();
  const Scalar negOne = -Teuchos::ScalarTraits<Scalar>::one();
  const Scalar half = static_cast<Scalar>(one / 2.0);

  for (LocalOrdinal i = 0; i < n_ - 1; i++) {
    (*H_)(i, i) = one;
    (*H_)(i, i+1) = negOne;
  }
  for (LocalOrdinal i = 0; i < n_ - 1; i++) {
    (*F_)(i, i) = half;
    (*F_)(i, i+1) = half;
  }
}

template<class Scalar, class LocalOrdinal, class GlobalOrdinal, class Node>
```

```
void AOOperator<Scalar,LocalOrdinal,GlobalOrdinal,Node>::apply(const Tpetra::MultiVector<
    Scalar,LocalOrdinal,GlobalOrdinal,Node> &X, Tpetra::MultiVector<Scalar,LocalOrdinal,
    GlobalOrdinal,Node> &Y, Teuchos::ETransp mode /* = Teuchos::NO_TRANS */, Scalar alpha
     /* = Teuchos::ScalarTraits<Scalar>::one() */, Scalar beta /* = Teuchos::ScalarTraits
    <Scalar>::zero() */) const
{
  const Scalar one = Teuchos::ScalarTraits<Scalar>::one();
  const Scalar zero = Teuchos::ScalarTraits<Scalar>::zero();
  const LocalOrdinal STRIDE = X.getStride();
  if (mode == Teuchos::NO_TRANS) {
    Teuchos::RCP<Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> > V = Teuchos::rcp(new
        Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> (n_, n_, false) );
    Teuchos::ArrayRCP<const Scalar> xView = X.get1dView();
    for (LocalOrdinal i = 0; i < n_; i++) {
      for (LocalOrdinal j = 0; j < n_; j++) {
        (*V)(i,j) = xView[j*n_ + i];
      }
    }
    xView = Teuchos::null;
    /* Note: We need the Teuchos::Copy because, otherwise, we need to reshape V,
     * which effectively copies the entries anyway.  This is because the lda
     * will be 0 otherwise (due to the stride) and mat-mat multiplies won't work
     */
    Teuchos::RCP<Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> > HV = Teuchos::rcp(new
        Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> (n_ - 1, n_, false) );
    HV->multiply(Teuchos::NO_TRANS, Teuchos::NO_TRANS, one, *H_, *V, zero);
    Teuchos::RCP<Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> > VHt = Teuchos::rcp(new
         Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> (n_, n_ - 1, false) );
    int res = VHt->multiply(Teuchos::NO_TRANS, Teuchos::TRANS, one, *V, *H_, zero);
    Teuchos::RCP<Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> > Nm1Nm1 = Teuchos::rcp(
        new Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> (n_ - 1, n_ - 1, false) );
    Nm1Nm1->multiply(Teuchos::NO_TRANS, Teuchos::TRANS, one, *HV, *F_, zero);
    Nm1Nm1->scale(*mask_);
    Scalar *values = Nm1Nm1->values();
    for (LocalOrdinal i = 0; i < (n_-1)*(n_-1); i++) {
      Y.replaceGlobalValue(i, 0, values[i]);
    }
    Nm1Nm1->multiply(Teuchos::NO_TRANS, Teuchos::NO_TRANS, one, *F_, *VHt, zero);
    Nm1Nm1->scale(*mask_);
    values = Nm1Nm1->values();
    for (LocalOrdinal i = 0; i < (n_-1)*(n_-1); i++) {
      Y.replaceGlobalValue(i + (n_-1)*(n_-1), 0, values[i]);
    }
    HV->scale(alpha_);
    values = HV->values();
```

```
  for (LocalOrdinal i = 0; i < (n_-1)*n_; i++) {
    Y.replaceGlobalValue(i + 2*(n_-1)*(n_-1), 0, values[i]);
  }
  VHt->scale(alpha_);
  values = VHt->values();
  for (LocalOrdinal i = 0; i < n_*(n_-1); i++) {
    Y.replaceGlobalValue(i + 2*(n_-1)*(n_-1) + (n_-1)*n_, 0, values[i]);
  }
} else /* mode == Teuchos::TRANS */ {
  // Get pointer to values
  Teuchos::ArrayRCP<const Scalar> Xview = X.get1dView();
  Teuchos::RCP<Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> > V1 = Teuchos::rcp(new
      Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> (n_ - 1, n_ - 1, false) );
  Teuchos::RCP<Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> > V2 = Teuchos::rcp(new
      Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> (n_ - 1, n_ - 1, false) );
  Teuchos::RCP<Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> > V3 = Teuchos::rcp(new
      Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> (n_ - 1, n_, false) );
  Teuchos::RCP<Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> > V4 = Teuchos::rcp(new
      Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> (n_, n_ - 1, false) );
  Teuchos::ArrayRCP<const Scalar> xView = X.get1dView();
  for (LocalOrdinal i = 0; i < n_ - 1; i++) {
    for (LocalOrdinal j = 0; j < n_ - 1; j++) {
      (*V1)(i,j) = xView[j*(n_-1) + i];
    }
  }
  for (LocalOrdinal i = 0; i < n_ - 1; i++) {
    for (LocalOrdinal j = 0; j < n_ - 1; j++) {
      (*V2)(i,j) = xView[j*(n_-1) + i + (n_-1)*(n_-1)];
    }
  }
  for (LocalOrdinal i = 0; i < n_ - 1; i++) {
    for (LocalOrdinal j = 0; j < n_; j++) {
      (*V3)(i,j) = xView[j*(n_-1) + i + 2*(n_-1)*(n_-1)];
    }
  }
  for (LocalOrdinal i = 0; i < n_; i++) {
    for (LocalOrdinal j = 0; j < n_ - 1; j++) {
      (*V4)(i,j) = xView[j*n_ + i + 2*(n_-1)*(n_-1)+(n_-1)*n_];
    }
  }
  xView = Teuchos::null;

  Teuchos::RCP<Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> > MAV = Teuchos::rcp(
      new Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> (n_, n_, true) );
```

```
    Teuchos::RCP<Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> > Nm1Nm1 = Teuchos::rcp(
         new Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> (*V1) );
    Nm1Nm1->scale(*mask_);
    Teuchos::RCP<Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> > NNm1 = Teuchos::rcp(
        new Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> (n_, n_ - 1, false) );
    NNm1->multiply(Teuchos::TRANS, Teuchos::NO_TRANS, one, *H_, *Nm1Nm1, zero);
    Teuchos::RCP<Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> > NN = Teuchos::rcp( new
         Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> (n_, n_, false) );
    NN->multiply(Teuchos::NO_TRANS, Teuchos::NO_TRANS, one, *NNm1, *F_, zero);
    (*MAV) += (*NN);
    (*Nm1Nm1) = (*V2);
    Nm1Nm1->scale(*mask_);
    NNm1->multiply(Teuchos::TRANS, Teuchos::NO_TRANS, one, *F_, *Nm1Nm1, zero);
    NN->multiply(Teuchos::NO_TRANS, Teuchos::NO_TRANS, one, *NNm1, *H_, zero);
    (*MAV) += (*NN);
    NN->multiply(Teuchos::TRANS, Teuchos::NO_TRANS, one, *H_, *V3, zero);
    NN->scale(alpha_);
    (*MAV) += (*NN);
    NN->multiply(Teuchos::NO_TRANS, Teuchos::NO_TRANS, one, *V4, *H_, zero);
    NN->scale(alpha_);
    (*MAV) += (*NN);
    Scalar *values = MAV->values();
    for (LocalOrdinal i = 0; i < n_*n_; i++) {
      Y.replaceGlobalValue(i, 0, values[i]);
    }
  }
}

#endif /* AO_OPERATOR_HPP */
```

### A.5.2 AOPreconditioner.hpp

```
#ifndef AO_PRECONDITIONER_HPP
#define AO_PRECONDITIONER_HPP

/* AOPreconditioner.hpp
 * Concrete class representing the Adaptive Optics preconditioner.
 */

#include "Tpetra_Operator.hpp"
#include "Teuchos_SerialDenseMatrix.hpp"
#include "Tpetra_Map.hpp"

template<class Scalar, class LocalOrdinal = int, class GlobalOrdinal = LocalOrdinal,
    class Node = Kokkos::DefaultNode::DefaultNodeType>
```

```
class AOPreconditioner : public Tpetra::Operator<Scalar,LocalOrdinal,GlobalOrdinal,Node>
    {

  public:

  // Constructors/Destructor
  /* AOPreconditioner constructor.
   * This class is templated on Scalar, LocalOrdinal, GlobalOrdinal and Node.
   * The LocalOrdinal type, if omitted, defaults to int.
   * The GlobalOrdinal type defaults to the LocalOrdinal type.
   * The Node type defaults to the default node in Kokkos.
   * This constructor takes a Teuchos::SerialDenseMatrix describing the diagonal
   * values of C^(-1/2) for this operator as well as one for the explicit
   * inverse of X, both of size n x n.  It also takes a pointer to the
   * domain and range maps.
   */
  AOPreconditioner( const Teuchos::RCP<Teuchos::SerialDenseMatrix<LocalOrdinal, Scalar> >
       &Chahlfinv, const Teuchos::RCP<Teuchos::SerialDenseMatrix<LocalOrdinal, Scalar> >
      &Xinv, const Teuchos::RCP<const Tpetra::Map<LocalOrdinal,GlobalOrdinal,Node> > &
      domainMap, const Teuchos::RCP<const Tpetra::Map<LocalOrdinal,GlobalOrdinal,Node> >
      &rangeMap);

  // Destructor.
  ~AOPreconditioner() {};

  // Getter methods
  const Teuchos::RCP< const Tpetra::Map<LocalOrdinal,GlobalOrdinal,Node> > & getDomainMap
      () const { return dMap_; }

  const Teuchos::RCP< const Tpetra::Map<LocalOrdinal,GlobalOrdinal,Node> > & getRangeMap
      () const { return rMap_; }

  bool hasTransposeApply() const { return true; }

  // Apply method
  /* This method applies the preconditioner for the adaptive optics problem, or
   * its transpose to a vector X.  It scales the result by alpha, then adds that
   * to beta*Y, storing the result in Y.  Thus, Y = alpha*P^{mode}*X + beta*Y.
   */
  void apply(const Tpetra::MultiVector<Scalar,LocalOrdinal,GlobalOrdinal,Node> &X, Tpetra
      ::MultiVector<Scalar,LocalOrdinal,GlobalOrdinal,Node> &Y, Teuchos::ETransp mode=
      Teuchos::NO_TRANS, Scalar alpha = Teuchos::ScalarTraits<Scalar>::one(), Scalar beta
       = Teuchos::ScalarTraits<Scalar>::zero()) const;

private:
```

```
  // Inverse of C^.5 and X matrices
  Teuchos::RCP<Teuchos::SerialDenseMatrix<LocalOrdinal, Scalar> > Chalfinv_;
  Teuchos::RCP<Teuchos::SerialDenseMatrix<LocalOrdinal, Scalar> > Xinv_;
  // Domain and range maps
  Teuchos::RCP<const Tpetra::Map<LocalOrdinal,GlobalOrdinal,Node> > dMap_;
  Teuchos::RCP<const Tpetra::Map<LocalOrdinal,GlobalOrdinal,Node> > rMap_;
  // Size
  LocalOrdinal n_;
};

template<class Scalar, class LocalOrdinal, class GlobalOrdinal, class Node>
AOPreconditioner<Scalar,LocalOrdinal,GlobalOrdinal,Node>::AOPreconditioner( const Teuchos
    ::RCP<Teuchos::SerialDenseMatrix<LocalOrdinal, Scalar> > &Chalfinv, const Teuchos::
    RCP<Teuchos::SerialDenseMatrix<LocalOrdinal, Scalar> > &Xinv, const Teuchos::RCP<
    const Tpetra::Map<LocalOrdinal,GlobalOrdinal,Node> > &domainMap, const Teuchos::RCP<
    const Tpetra::Map<LocalOrdinal,GlobalOrdinal,Node> > &rangeMap ) :
  Chalfinv_(Chalfinv),
  Xinv_(Xinv),
  dMap_(domainMap),
  rMap_(rangeMap),
  n_(Chalfinv_->numRows())
{
}

template<class Scalar, class LocalOrdinal, class GlobalOrdinal, class Node>
void AOPreconditioner<Scalar,LocalOrdinal,GlobalOrdinal,Node>::apply(const Tpetra::
    MultiVector<Scalar,LocalOrdinal,GlobalOrdinal,Node> &X, Tpetra::MultiVector<Scalar,
    LocalOrdinal,GlobalOrdinal,Node> &Y, Teuchos::ETransp mode /* = Teuchos::NO_TRANS */,
     Scalar alpha /* = Teuchos::ScalarTraits<Scalar>::one() */, Scalar beta /* = Teuchos
    ::ScalarTraits<Scalar>::zero() */) const
{
  const Scalar one = Teuchos::ScalarTraits<Scalar>::one();
  const Scalar zero = Teuchos::ScalarTraits<Scalar>::zero();
  const LocalOrdinal STRIDE = 0;
  if (mode == Teuchos::NO_TRANS) {
    Teuchos::RCP<Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> > V = Teuchos::rcp(new
        Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> (n_, n_, false) );
    Teuchos::ArrayRCP<const Scalar> xView = X.get1dView();
    for (LocalOrdinal i = 0; i < n_; i++) {
      for (LocalOrdinal j = 0; j < n_; j++) {
        (*V)(i,j) = xView[j*n_ + i];
      }
    }
    xView = Teuchos::null;
    V->scale(*Chalfinv_);
```

```
    Teuchos::RCP<Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> > NN1 = Teuchos::rcp(new
         Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> (n_, n_, false) );
    NN1->multiply(Teuchos::TRANS, Teuchos::NO_TRANS, one, *Xinv_, *V, zero);
    Teuchos::RCP<Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> > NN2 = Teuchos::rcp(new
         Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> (n_, n_, false) );
    NN2->multiply(Teuchos::NO_TRANS, Teuchos::NO_TRANS, one, *NN1, *Xinv_, zero);
    Scalar *values = NN2->values();
    for (LocalOrdinal i = 0; i < n_*n_; i++) {
      Y.replaceGlobalValue(i, 0, values[i]);
    }
  } else /* mode == Teuchos::TRANS */ {
    Teuchos::RCP<Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> > V = Teuchos::rcp(new
        Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> (n_, n_, false) );
    Teuchos::ArrayRCP<const Scalar> xView = X.get1dView();
    for (LocalOrdinal i = 0; i < n_; i++) {
      for (LocalOrdinal j = 0; j < n_; j++) {
        (*V)(i,j) = xView[j*n_ + i];
      }
    }
    xView = Teuchos::null;
    Teuchos::RCP<Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> > NN1 = Teuchos::rcp(new
         Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> (n_, n_, false) );
    NN1->multiply(Teuchos::NO_TRANS, Teuchos::NO_TRANS, one, *Xinv_, *V, zero);
    Teuchos::RCP<Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> > NN2 = Teuchos::rcp(new
         Teuchos::SerialDenseMatrix<LocalOrdinal,Scalar> (n_, n_, false) );
    NN2->multiply(Teuchos::NO_TRANS, Teuchos::TRANS, one, *NN1, *Xinv_, zero);
    NN2->scale(*Chalfinv_);
    Scalar *values = NN2->values();
    for (LocalOrdinal i = 0; i < n_*n_; i++) {
      Y.replaceGlobalValue(i, 0, values[i]);
    }
  }
}

#endif /* AO_PRECONDITIONER_HPP */
```

### A.5.3 AOmain.hpp

```
#ifndef AO_MAIN_HPP
#define AO_MAIN_HPP

#include "AOOperator.hpp"
#include "AOPreconditioner.hpp"
#include "LSQRSolMgr.hpp"
#include "Tpetra_Map.hpp"
#include "Tpetra_MultiVector.hpp"
```

```
#include "Tpetra_DefaultPlatform.hpp"
#include "BelosTpetraAdapter.hpp"
#include "BelosLinearProblem.hpp"

enum Orientation {
  ROW_MAJOR,
  COL_MAJOR
};

// Read in a file in raw data form that contains a matrix and store the values
// in a given SerialDenseMatrix object.
/*
  \param file - (In) File in raw data form containing a matrix.
  \param matrix - (Out) Matrix of correct size to store data.
  \param orientation - (In) Orientation of data in file (row or column major).
*/
template <class Scalar, class Ordinal>
void readFile(const char *file, const Teuchos::RCP< Teuchos::SerialDenseMatrix< Ordinal,
    Scalar > > &matrix, Orientation orientation);

// Write reconstructed solution in raw form.
/*
  \param file - (In) File to store solution in raw form.
  \param Phi - (In) Reconstructed solution.
*/
template <class Scalar, class Ordinal>
void writeFile(const char *file, const Teuchos::RCP<const Tpetra::Vector< Scalar, Ordinal
    , Ordinal > > &Phi);

// Generate a mask.  If r0 is omitted, the mask is circular with radius r1.
// Otherwise, it is annular with inner radius r0.  The largest circle contained
// in the n by n square has radius r1 = 1.
/*
  \param mask - (Out) Matrix of size n x n to hold mask.  Consists of 1s and 0s.
  \param r1 - (In) Outer radius.  Default value is 0.5.
  \param r0 - (In) Inner radius.  Default value is 0.
*/
template <class Scalar, class Ordinal>
void makeMask(const Teuchos::RCP< Teuchos::SerialDenseMatrix< Ordinal, Scalar > > &mask,
    Scalar r1 = 0.5, Scalar r0 = 0.0);

// Solver function with or without preconditioning.
/*
  \param alpha - (In) Alpha value.
  \param ChalfinvFileName - (In) File containing diagonal portion of
```

```
    preconditioner.
  \param XinvFileName - (In) File containing other portion of preconditioner.
  \param mask - (In) Mask matrix.
  \param Phi - (Out) Solution vector.
  \param B - (In) Data vector.
  \param solverParams - (In) Solver parameters.
*/
template <class Scalar, class Ordinal>
void solve(Scalar alpha, const char *ChalfinvFileName, const char *XinvFileName, Teuchos
    ::RCP< Teuchos::SerialDenseMatrix< Ordinal, Scalar > > &mask, const Teuchos::RCP<
    Tpetra::MultiVector< Scalar, Ordinal > > &Phi, const Teuchos::RCP< Tpetra::
    MultiVector< Scalar, Ordinal> > &B, Teuchos::RCP< Teuchos::ParameterList > &
    solverParams );

// Main method for reconstructing.
/*
  \param pl - (In) Parameter list containing all the necessary information.
*/
template <class Scalar, class Ordinal>
void reconstruct(const Teuchos::RCP< const Teuchos::ParameterList > &pl);

template <class Scalar, class Ordinal>
void readFile(const char *file, const Teuchos::RCP< Teuchos::SerialDenseMatrix< Ordinal,
    Scalar > > &matrix, Orientation orientation) {
  std::ifstream fileName;
  int size = matrix->numRows() * matrix->numCols();
  double* tempNumArray = new double[size];
  fileName.open(file, std::ios::out);
  if (fileName.fail()) {
    std::cerr << "Error: failed to open file: " << file << std::endl;
    exit(1);
  }
  fileName.read((char*)tempNumArray, size*sizeof(double));
  if (orientation == ROW_MAJOR) {
    for (Ordinal i = 0; i < matrix->numRows(); i++) {
      for (Ordinal j = 0; j < matrix->numCols(); j++) {
  (*matrix)(i,j) = (Scalar) tempNumArray[i*matrix->numCols() + j];
      }
    }
  } else if (orientation == COL_MAJOR) {
    for (Ordinal j = 0; j < matrix->numCols(); j++) {
      for (Ordinal i = 0; i < matrix->numRows(); i++) {
  (*matrix)(i,j) = (Scalar) tempNumArray[j*matrix->numRows() + i];
      }
    }
```

```
  } else {
    std::cerr << "Incorrect orientation type" << std::endl;
  }
  fileName.close();
  delete [] tempNumArray;
}

template <class Scalar, class Ordinal>
void writeData(const char *file, const Teuchos::RCP<const Tpetra::Vector< Scalar, Ordinal
    , Ordinal > > &Phi) {
  std::ofstream SolutionFileName;
  Teuchos::ArrayRCP<const Scalar> phiView = Phi->get1dView();
  Ordinal i;
  Scalar temp;
  SolutionFileName.open(file, std::ios::out | std::ios::binary);
  for (i = 0; i < Phi->getGlobalLength(); i++) {
    temp = (Scalar) phiView[i];
    SolutionFileName.write((char*)&temp, sizeof(Scalar));
  }
  SolutionFileName.close();
  phiView = Teuchos::null;
}

template <class Scalar, class Ordinal>
void makeMask(const Teuchos::RCP< Teuchos::SerialDenseMatrix< Ordinal, Scalar > > &mask,
    Scalar r1 /* = 0.5 */, Scalar r0 /* = 0.0 */) {
  Ordinal n = mask->numRows();
  Scalar zero = Teuchos::ScalarTraits<Scalar>::zero();
  Scalar one = Teuchos::ScalarTraits<Scalar>::one();
  Scalar negOne = -one;
  Scalar two = 2*one;

  Scalar h = two / n;
  Teuchos::RCP< Teuchos::SerialDenseVector< Ordinal, Scalar > > x = Teuchos::rcp(new
      Teuchos::SerialDenseVector< Ordinal, Scalar > (n, false));
  for (Ordinal i = 0; i < n; i++) {
    (*x)(i) = negOne + i*h;
  }
  Teuchos::RCP< Teuchos::SerialDenseVector< Ordinal, Scalar > > onevec = Teuchos::rcp(new
       Teuchos::SerialDenseVector< Ordinal, Scalar > (n, false));
  (*onevec) = one;
  Teuchos::RCP< Teuchos::SerialDenseMatrix< Ordinal, Scalar > > xonet = Teuchos::rcp(new
      Teuchos::SerialDenseMatrix< Ordinal, Scalar > (n, n, false));
  xonet->multiply(Teuchos::NO_TRANS, Teuchos::TRANS, one, *x, *onevec, zero);
```

```
  Teuchos::RCP< Teuchos::SerialDenseMatrix< Ordinal, Scalar > > onext = Teuchos::rcp(new
      Teuchos::SerialDenseMatrix< Ordinal, Scalar > (n, n, false));
  onext->multiply(Teuchos::NO_TRANS, Teuchos::TRANS, one, *onevec, *x, zero);
  Teuchos::RCP< Teuchos::SerialDenseMatrix< Ordinal, Scalar > > r = Teuchos::rcp(new
      Teuchos::SerialDenseMatrix< Ordinal, Scalar > (n, n, false));
  for (Ordinal i = 0; i < n; i++) {
    for (Ordinal j = 0; j < n; j++) {
      (*r)(i,j) = Teuchos::ScalarTraits<Scalar>::squareroot( (*xonet)(i,j) * (*xonet)(i,j
          ) + (*onext)(i,j) * (*onext)(i,j) );
    }
  }
  if (r0 <= zero) {
    // Making a circular mask
    for (Ordinal i = 0; i < n; i++) {
      for (Ordinal j = 0; j < n; j++) {
        (*mask)(i,j) = (*r)(i,j) <= r1 ? one : zero;
      }
    }
  } else {
    // Making an annular mask
    for (Ordinal i = 0; i < n; i++) {
      for (Ordinal j = 0; j < n; j++) {
        (*mask)(i,j) = ( (r0 <= (*r)(i,j)) && ((*r)(i,j) <= r1) ) ? one : zero;
      }
    }
  }
}

template <class Scalar, class Ordinal>
void solve(Scalar alpha, const char *ChalfinvFileName, const char *XinvFileName, Teuchos
    ::RCP< Teuchos::SerialDenseMatrix< Ordinal, Scalar > > &mask, const Teuchos::RCP<
    Tpetra::MultiVector< Scalar, Ordinal > > &Phi, const Teuchos::RCP< Tpetra::
    MultiVector< Scalar, Ordinal > > &B, Teuchos::RCP< Teuchos::ParameterList > &
    solverParams ) {
  typedef Tpetra::MultiVector< Scalar, Ordinal, Ordinal > MV;
  typedef Tpetra::Operator< Scalar, Ordinal > OP;

  // Determine n
  Ordinal n = mask->numRows();
  bool havePrec = (ChalfinvFileName != NULL && XinvFileName != NULL);
  Teuchos::RCP< Teuchos::SerialDenseMatrix< Ordinal, Scalar > > Chalfinv;
  Teuchos::RCP< Teuchos::SerialDenseMatrix< Ordinal, Scalar > > Xinv;
  if (havePrec) {
    Chalfinv = Teuchos::rcp( new Teuchos::SerialDenseMatrix< Ordinal, Scalar > (n, n,
        false) );
```

```
    Xinv = Teuchos::rcp( new Teuchos::SerialDenseMatrix< Ordinal, Scalar > (n, n, false)
        );
    readFile(ChalfinvFileName, Chalfinv, COL_MAJOR);
    readFile(XinvFileName, Xinv, COL_MAJOR);
  }

  Teuchos::RCP< AOOperator<Scalar,Ordinal> > AO = Teuchos::rcp( new AOOperator<Scalar,
      Ordinal> (mask, alpha, Phi->getMap(), B->getMap()) );
  Teuchos::RCP< Belos::LinearProblem< Scalar, MV, OP > > myProblem = Teuchos::rcp(new
      Belos::LinearProblem< Scalar, MV, OP>(AO, Phi, B));
  Teuchos::RCP< AOPreconditioner<Scalar,Ordinal> > AP;
  if (havePrec) {
    AP = Teuchos::rcp( new AOPreconditioner<Scalar,Ordinal> (Chalfinv, Xinv, Phi->getMap
        (), Phi->getMap()) );
    myProblem->setRightPrec(AP);
  }
  bool set = myProblem->setProblem();
  if (!set) {
    std::cout << "Error! LSQR not set" << std::endl;
  }
  LSQRSolMgr< Scalar, MV, OP > solver(myProblem, solverParams);
  solver.solve();
  Ordinal numIters = solver.getNumIters();
  std::cout << "LSQR num iters = " << numIters << std::endl;
}

template <class Scalar, class Ordinal>
void reconstruct(const Teuchos::RCP< const Teuchos::ParameterList > &pl) {
  Ordinal n = pl->get<Ordinal>("n");
  Scalar r0 = -1;
  Scalar r1 = -1;
  if (pl->isParameter("r0")) {
    r0 = pl->get<Scalar>("r0");
  }
  if (pl->isParameter("r1")) {
    r1 = pl->get<Scalar>("r1");
  }
  Teuchos::RCP< Teuchos::SerialDenseMatrix<Ordinal,Scalar> > mask = Teuchos::rcp( new
      Teuchos::SerialDenseMatrix<Ordinal,Scalar>(n-1, n) );
  if (r0 != -1 && r1 != -1) {
    makeMask<Scalar,Ordinal>(mask, r1, r0);
  } else if (r1 != -1) {
    makeMask<Scalar,Ordinal>(mask, r1);
  } else {
    makeMask<Scalar,Ordinal>(mask);
```

```
}
std::string sxFileName = pl->get<std::string>("sx File");
std::string syFileName = pl->get<std::string>("sy File");
Scalar alpha = pl->get<Scalar>("alpha");
std::string ChalfinvFileName, XinvFileName;
bool prec = false;
if (pl->isParameter("Chalfinv File") && pl->isParameter("Xinv File")) {
  ChalfinvFileName = pl->get<std::string>("Chalfinv File");
  XinvFileName = pl->get<std::string>("Xinv File");
  prec = true;
}

Teuchos::RCP<const Teuchos::Comm<int> > comm = Tpetra::DefaultPlatform::
    getDefaultPlatform().getComm();
Teuchos::RCP<const Tpetra::Map<Ordinal> > domainMap = Teuchos::rcp( new Tpetra::Map<
    Ordinal>(n*n, 0, comm));
Teuchos::RCP<Tpetra::Vector< Scalar, Ordinal, Ordinal > > Phi = Teuchos::rcp( new
    Tpetra::Vector< Scalar, Ordinal, Ordinal >(domainMap) );
Teuchos::RCP<const Tpetra::Map<Ordinal> > rangeMap = Teuchos::rcp( new Tpetra::Map<
    Ordinal> (2*(n-1)*(n-1)+2*n*(n-1), 0, comm) );
Teuchos::RCP<Tpetra::Vector< Scalar, Ordinal, Ordinal > > B = Teuchos::rcp( new Tpetra
    ::Vector<Scalar, Ordinal, Ordinal > (rangeMap) );
Teuchos::RCP< Teuchos::SerialDenseMatrix< Ordinal, Scalar > > sx = Teuchos::rcp( new
    Teuchos::SerialDenseMatrix< Ordinal, Scalar > ((n-1)*(n-1), 1, false) );
Teuchos::RCP< Teuchos::SerialDenseMatrix< Ordinal, Scalar > > sy = Teuchos::rcp( new
    Teuchos::SerialDenseMatrix< Ordinal, Scalar > ((n-1)*(n-1), 1, false) );
readFile(sxFileName.c_str(), sx, COL_MAJOR);
readFile(syFileName.c_str(), sy, COL_MAJOR);
for(Ordinal i = 0; i < (n-1)*(n-1); i++) {
  B->replaceGlobalValue(i, (*sx)(i,0) );
}
for(Ordinal i = 0; i < (n-1)*(n-1); i++) {
  B->replaceGlobalValue(i+(n-1)*(n-1), (*sy)(i,0) );
}

Teuchos::ParameterList spl = (pl->sublist("LSQR Solver"));
Teuchos::RCP< Teuchos::ParameterList > solverPL = Teuchos::rcp( &spl, false );
if (prec) {
  solve<Scalar,Ordinal>(alpha, ChalfinvFileName.c_str(), XinvFileName.c_str(), mask,
      Phi, B, solverPL);
} else {
  solve<Scalar,Ordinal>(alpha, NULL, NULL, mask, Phi, B, solverPL);
}
std::string fname = pl->get<std::string>("Output File");
writeData<Scalar, Ordinal>(fname.c_str(), Phi);
```

```
}

#endif /* AO_MAIN_HPP */
```

### A.5.4 AOmain.cpp

```
#include "AOmain.hpp"
#include <Teuchos_GlobalMPISession.hpp>
#include <Teuchos_oblackholestream.hpp>
#include "Teuchos_XMLParameterListHelpers.hpp"

int main(int argc, char *argv[]) {
  typedef int Ordinal;
  typedef double Scalar;
  Teuchos::oblackholestream blackhole;
  Teuchos::GlobalMPISession mpiSession(&argc, &argv, &blackhole);
  Teuchos::RCP< Teuchos::ParameterList > pl = Teuchos::getParametersFromXmlFile(argv[1]);
  reconstruct<Scalar,Ordinal>(pl);
  return 0;
}
```

### A.5.5 Example XML File

```
<ParameterList>
  <Parameter name="n" type="int" value="64"/>
  <Parameter name="r1" type="double" value="0.9"/>
  <Parameter name="alpha" type="double" value="0.0"/>
  <Parameter name="sx File" type="string" value="sx64.double"/>
  <Parameter name="sy File" type="string" value="sy64.double"/>
  <Parameter name="Chalfinv File" type="string" value="Chalfinv64.double"/>
  <Parameter name="Xinv File" type="string" value="Xinv64.double"/>
  <ParameterList name="LSQR Solver">
    <Parameter name="Condition Limit" type="double" value="1000"/>
    <Parameter name="Maximum Iterations" type="int" value="200"/>
    <Parameter name="Rel Mat Err" type="double" value="0"/>
    <Parameter name="Term Iter Max" type="int" value="1"/>
  </ParameterList>
  <Parameter name="Ordinal Type" type="string" value="int"/>
  <Parameter name="Scalar Type" type="string" value="double"/>
  <Parameter name="Output File" type="string" value="reconstructedPhi64.double"/>
</ParameterList>
```

# Bibliography


[1] H. C. Andrews and B. R. Hunt. *Digital Image Restoration*. Prentice Hall Professional Technical Reference, Englewood Cliffs, NJ, 1977.

[2] M. Bachmayr and M. Burger. Iterative total variation schemes for nonlinear inverse problems. *Inverse Problems*, 25(10), 2009.

[3] S. Balay, J. Brown, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.1, Argonne National Laboratory, 2010.

[4] S. Balay, J. Brown, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc Web page, 2011. http://www.mcs.anl.gov/petsc.

[5] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[6] S. Barbero, J. Rubinstein, and L. N. Thibos. Wavefront sensing and reconstruction from gradient and Laplacian data measured with a Hartmann-Shack sensor. *Optics Letters*, 31(12):1845–1847, 2006.

[7] J. Bardsley, S. Knepper, and J. Nagy. Structured linear algebra problems in adaptive optics imaging. *Adv. Comput. Math., to appear*, 2011.

[8] J. M. Bardsley. An efficient computational method for total variation-penalized Poisson likelihood estimation. *Inverse Prob. Imag.*, 2(2):167–185, 2008.

[9] J. M. Bardsley. Stopping rules for a nonnegatively constrained iterative method for ill-posed Poisson imaging problems. *BIT*, 48(4):651–664, 2008.

[10] J. M. Bardsley. Wavefront reconstruction methods for adaptive optics systems on ground-based telescopes. *SIAM J. Matrix Anal. Appl.*, 30(1):67–83, 2008.

[11] J. M. Bardsley and C. R. Vogel. A nonnegatively constrained convex programming method for image reconstruction. *SIAM J. Sci. Comput.*, 25(4):1326–1343, 2003.

[12] H. H. Barrett, C. Dainty, and D. Lara. Maximum-likelihood methods in wavefront sensing: stochastic models and likelihood functions. *J. Opt. Soc. Am. A*, 24(2):391–414, 2007.

[13] J. Barzilai and J. M. Borwein. Two-point step size gradient methods. *IMA J. Numer. Anal.*, 8(1):141–148, 1988.

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

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

[16] Å. Björck, E. Grimme, and P. van Dooren. An implicit shift bidiagonalization algorithm for ill-posed systems. *BIT*, 34(4):510–534, 1994.

[17] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, PA, 1997.

[18] P. M. Bloomfield, T. 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(8):959–978, 2003.

[19] H. Brakhage. On ill-posed problems and the method of conjugate gradients. In H. W. Engl and C. W. Groetsch, editors, *Inverse and Ill-Posed Problems*, pages 165–175. Academic Press, Boston, 1987.

[20] 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. Imag.*, 23(9):1176–1185, 2004.

[21] D. Calvetti and L. Reichel. Tikhonov regularization of large linear problems. *BIT*, 43(2):263–283, 2003.

[22] D. Calvetti and E. Somersalo. *Introduction to Bayesian Scientific Computing: Ten Lectures on Subjective Computing.* Springer-Verlag New York, Inc., Secaucus, NJ, 2007.

[23] E. J. Candès, J. K. Romberg, and T. Tao. Stable signal recovery from incomplete and inaccurate measurements. *Comm. Pure Appl. Math.*, 59(8):1207–1223, 2006.

[24] R. C. Cannon. Global wave-front reconstruction using Shack-Hartmann sensors. *J. Opt. Soc. Am. A*, 12(9):2031–2039, 1995.

[25] K. Chadan, D. Colton, L. Päivärinta, and W. Rundell. *An Introduction to Inverse Scattering and Inverse Spectral Problems.* SIAM, Philadelphia, PA, 1997.

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

[27] M. Cheney and B. Borden. *Fundamentals of Radar Imaging.* SIAM, Philadelphia, PA, 2009.

[28] J. Chung, E. Haber, and J. Nagy. Numerical methods for coupled superresolution. *Inverse Problems*, 22(4):1261–1272, 2006.

[29] J. Chung, S. Knepper, and J. Nagy. Large-scale inverse problems in imaging. In O. Scherzer, editor, *Handbook of Mathematical Methods in Imaging*, pages 43–86. Springer, 2011.

[30] 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.

[31] J. Chung, P. Sternberg, and C. Yang. High performance 3-dimensional image reconstruction for molecular structure determination. *Int. J. High Perform. Comput. Appl.*, 24(2):117–135, 2010.

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

[33] M. Elad and A. Feuer. Restoration of a single superresolution image from several blurred, noisy, and undersampled measured images. *IEEE Trans. Image Proc.*, 6(12):1646–1658, 1997.

[34] B. L. Ellerbroek. Efficient computation of minimum-variance wave-front reconstructors with sparse matrix techniques. *J. Opt. Soc. Am. A*, 19(9):1803–1816, 2002.

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

[36] H. W. Engl and P. Kügler. Nonlinear inverse problems: Theoretical aspects and some industrial applications. In V. Capasso and J. Périaux, editors, *Multidisciplinary Methods for Analysis Optimization and Control of Complex Systems*, pages 3–48. Springer, Berlin, 2005.

[37] H. W. Engl, K. Kunisch, and A. Neubauer. Convergence rates for Tikhonov regularisation of nonlinear ill-posed problems. *Inverse Problems*, 5(4):523–540, 1989.

[38] H. W. Engl, A. K. Louis, and W. Rundell, editors. *Inverse Problems in Geophysical Applications*, Philadelphia, PA, 1997. SIAM.

[39] J. Eriksson and P. Wedin. Truncated Gauss-Newton algorithms for ill-conditioned nonlinear least squares problems. *Optim. Methods Softw.*, 19(6):721–737, 2004.

[40] S. Ettl, J. Kaminski, M. C. Knauer, and G. Häusler. Shape reconstruction from gradient data. *Applied Optics*, 47(12):2091–2097, 2008.

[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, 2009.

[42] M. A. T. Figueiredo, R. D. Nowak, and S. J. Wright. Gradient projection for sparse reconstruction: Application to compressed sensing and other inverse problems. *IEEE J. Sel. Topics Signal Process.*, 1(4):586–597, 2007.

[43] J. Frank. *Three-Dimensional Electron Microscopy of Macromolecular Assemblies*. Oxford University Press, New York, 2006.

[44] D. L. Fried. Least-square fitting a wave-front distortion estimate to an array of phase-difference measurements. *J. Opt. Soc. Am.*, 67(3):370–375, 1977.

[45] R. R. Fulton, S. R. Meikle, S. Eberl, J. Pfeiffer, C. J. Constable, and M. J. Fulham. Correction for head movements in positron emission tomography using

an optical motion-tracking system. *IEEE Trans. Nucl. Sci.*, 49(1):116–123, 2002.

[46] L. Gilles. Order-N sparse minimum-variance open-loop reconstructor for extreme adaptive optics. *Optics Letters*, 28(20):1927–1929, 2003.

[47] L. Gilles, C. R. Vogel, and B. L. Ellerbroek. Multigrid preconditioned conjugate-gradient method for large-scale wave-front reconstruction. *J. Opt. Soc. Am. A*, 19(9):1817–1822, 2002.

[48] 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.

[49] G. H. Golub, F. T. Luk, and M. L. Overton. A block Lanczos method for computing the singular values and corresponding singular vectors of a matrix. *ACM Trans. Math. Softw.*, 7(2):149–169, 1981.

[50] G. H. Golub and V. Pereyra. The differentiation of pseudo-inverses and nonlinear least squares problems whose variables separate. *SIAM J. Numer. Anal.*, 10(2):413–432, 1973.

[51] G. H. Golub and V. Pereyra. Separable nonlinear least squares: the variable projection method and its applications. *Inverse Problems*, 19(2):R1–R26, 2003.

[52] G. H. Golub and C. F. Van Loan. *Matrix Computations, 3rd edition*. Johns Hopkins University Press, Baltimore, MD, 1996.

[53] J. W. Goodman. *Introduction to Fourier Optics, 2nd edition*. McGraw-Hill, New York, 1996.

[54] 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(9):1538–1546, 1994.

[55] E. Haber, U. M. Ascher, and D. Oldenburg. On optimization techniques for solving nonlinear inverse problems. *Inverse Problems*, 16(5):1263–1280, 2000.

[56] E. Haber and D. Oldenburg. A GCV based method for nonlinear ill-posed problems. *Computational Geosciences*, 4(1):41–63, 2000.

[57] M. Hanke. *Conjugate Gradient Type Methods for Ill-Posed Problems*. Pitman Research Notes in Mathematics, Longman Scientific & Technical, Harlow, Essex, 1995.

[58] M. Hanke. Limitations of the L-curve method in ill-posed problems. *BIT*, 36(2):287–301, 1996.

[59] M. Hanke. On Lanczos based methods for the regularization of discrete ill-posed problems. *BIT*, 41(5):1008–1018, 2001.

[60] M. Hanke, J. G. Nagy, and C. Vogel. Quasi-Newton approach to nonnegative image restorations. *Linear Algebra Appl.*, 316(1–3):223–236, 2000.

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

[62] P. C. Hansen. Numerical tools for analysis and solution of Fredholm integral equations of the first kind. *Inverse Problems*, 8(6):849–872, 1992.

[63] P. C. Hansen. Regularization Tools: A Matlab package for analysis and solution of discrete ill-posed problems. *Numerical Algorithms*, 6(1):1–35, 1994.

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

[65] P. C. Hansen. Regularization Tools version 4.0 for Matlab 7.3. *Numerical Algorithms*, 46(2):189–194, 2007.

[66] P. C. Hansen. *Discrete Inverse Problems: Insight and Algorithms*. SIAM, Philadelphia, PA, 2010.

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

[68] P. C. Hansen and D. P. O'Leary. The use of the L-curve in the regularization of discrete ill-posed problems. *SIAM J. Sci. Comput.*, 14(6):1487–1503, 1993.

[69] M. Heroux, R. Bartlett, V. Howle, R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, and A. Williams. An overview of Trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, 2003.

[70] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

[71] M. A. Heroux and J. M. Willenbring. Trilinos users guide. Technical Report SAND2003-2952, Sandia National Laboratories, 2003.

[72] B. Hofmann. Regularization of nonlinear problems and the degree of ill-posedness. In G. Anger, R. Gorenflo, H. Jochmann, H. Moritz, and W. Webers, editors, *Inverse Problems: Principles and Applications in Geophysics, Technology, and Medicine*. Akademie Verlag, Berlin, 1993.

[73] M. Hohn, G. Tang, G. Goodyear, P. R. Baldwin, Z. Huang, P. A. Penczek, C. Yang, R. M. Glaeser, P. D. Adams, and S. J. Ludtke. SPARX, a new environment for Cryo-EM image processing. *J. Struct. Biol.*, 157(1):47–55, 2007.

[74] J. Howse. Software for LSQR (C version), 1999. http://www.stanford.edu/group/SOL/software/lsqr.html.

[75] R. H. Hudgin. Wave-front reconstruction for compensated imaging. *J. Opt. Soc. Am.*, 67(3):375–378, 1977.

[76] H. M. Hudson and R. S. Larkin. Accelerated image reconstruction using ordered subsets of projection data. *IEEE Trans. Med. Imag.*, 13(4):601–609, 1994.

[77] B. R. Hunt. Matrix formulation of the reconstruction of phase values from phase differences. *J. Opt. Soc. Am.*, 69(3):393–399, 1979.

[78] M. Jacobsen. *Modular Regularization Algorithms*. PhD thesis, Informatics and Mathematical Modelling, Technical Unversity of Denmark, Lyngby, 2004.

[79] T. K. Jensen. *Stabilization Algorithms for Large-Scale Problems*. PhD thesis, Informatics and Mathematical Modelling, Technical University of Denmark, Lyngby, 2006.

[80] M. G. Kang and S. Chaudhuri. Super-resolution image reconstruction. *IEEE Signal Process. Mag.*, 20(3):19–20, 2003.

[81] S. Karimi, D. K. Salkuyeh, and F. Toutounian. A preconditioner for the LSQR algorithm. *J. Appl. Math. and Informatics*, 26(1–2):213–222, 2008.

[82] L. Kaufman. A variable projection method for solving separable nonlinear least squares problems. *BIT*, 15(1):49–57, 1975.

[83] M. E. Kilmer, P. C. Hansen, and M. I. Español. A projection-based approach to general-form Tikhonov regularization. *SIAM J. Sci. Comput.*, 29(1):315–330, 2007.

[84] M. E. Kilmer and D. P. O'Leary. Choosing regularization parameters in iterative methods for ill-posed problems. *SIAM J. Matrix Anal. Appl.*, 22(4):1204–1221, 2001.

[85] L. Landweber. An iteration formula for Fredholm integral equations of the first kind. *American J. of Mathematics*, 73(3):615–624, 1951.

[86] R. M. Larsen. *Lanczos Bidiagonalization with Partial Reorthogonalization*. PhD thesis, Dept. of Computer Science, University of Aarhus, Denmark, 1998.

[87] C. L. Lawson and R. J. Hanson. *Solving Least Squares Problems*. SIAM, Philadelphia, PA, 1995.

[88] C. Van Loan. The ubiquitous Kronecker product. *J. Comp. and Appl. Math.*, 123(1–2):85–100, 2000.

[89] R. Marabini, G. T. Herman, and J. M. Carazo. 3D reconstruction in electron microscopy using ART with smooth spherically symmetric volume elements (blobs). *Ultramicroscopy*, 72(1–2):53–65, 1998.

[90] M. Menke, M. S. Atkins, and K. R. Buckley. Compensation methods for head motion detected during PET imaging. *IEEE Trans. Nucl. Sci.*, 43(1):310–317, 1996.

[91] K. Miller. Least squares methods for ill-posed problems with a prescribed bound. *SIAM J. Math. Anal.*, 1(1):52–74, 1970.

[92] J. Modersitzki. *Numerical Methods for Image Registration*. Oxford University Press, Oxford, 2004.

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

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

[95] J. G. Nagy and Z. Strakoš. Enforcing nonnegativity in image reconstruction algorithms. In D. C. Wilson, H. D. Tagare, F. L. Bookstein, F. J. Preteux, and E. R. Dougherty, editors, *Mathematical Modeling, Estimation, and Imaging*, volume 4121, pages 182–190. SPIE, 2000.

[96] F. Natterer. *The Mathematics of Computerized Tomography*. SIAM, Philadelphia, PA, 2001.

[97] F. Natterer and F. Wübbeling. *Mathematical Methods in Image Reconstruction*. SIAM, Philadelphia, PA, 2001.

[98] N. Nguyen, P. Milanfar, and G. Golub. Efficient generalized cross-validation with applications to parametric image restoration and resolution enhancement. *IEEE Trans. Image Proc.*, 10(9):1299–1308, 2001.

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

[100] 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(4):474–489, 1981.

[101] M. R. Osborne. Separable least squares, variable projection, and the Gauss-Newton algorithm. *Elec. Trans. Numer. Anal.*, 28:1–15, 2007.

[102] C. C. Paige and M. A. Saunders. Algorithm 583: LSQR: Sparse linear equations and least squares problems. *ACM Trans. Math. Softw.*, 8(2):195–209, 1982.

[103] C. C. Paige and M. A. Saunders. LSQR: An algorithm for sparse linear equations and sparse least squares. *ACM Trans. Math. Softw.*, 8(1):43–71, 1982.

[104] P. Penczek, M. Radermacher, and J. Frank. Three-dimensional reconstruction of single particles embedded in ice. *Ultramicroscopy*, 40(1):33–53, 1992.

[105] D. L. Phillips. A technique for the numerical solution of certain integral equations of the first kind. *J. ACM*, 9(1):84–97, 1962.

[106] Y. Picard and C. J. Thompson. Motion correction of PET images using multiple acquisition frames. *IEEE Trans. Med. Imag.*, 16(2):137–144, 1997.

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

[108] 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(5):2588–2596, 2004.

[109] 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(6):574–581, 2008.

[110] H. Ren and R. Dekany. Fast wave-front reconstruction by solving the Sylvester equation with the alternating direction implicit method. *Opt. Express*, 12(14):3279–3296, 2004.

[111] H. Ren, R. Dekany, and M. Britton. Large-scale wave-front reconstruction for adaptive optics systems by use of a recursive filtering algorithm. *Applied Optics*, 44(13):2626–2637, 2005.

[112] M. Roggemann and B. Welsh. *Imaging Through Turbulence*. CRC Press, 1996.

[113] L. I. Rudin, S. Osher, and E. Fatemi. Nonlinear total variation based noise removal algorithms. *Physica D*, 60(1–4):259–268, 1992.

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

[115] Y. Saad. On the rates of convergence of the Lanczos and the block-Lanczos methods. *SIAM J. Numer. Anal.*, 17(5):687–706, 1980.

[116] S. D. Saban, M. Silvestry, G. R. Nemerow, and P. L. Stewart. Visualization of $\alpha$-helices in a 6-Ångstrom resolution cryoelectron microscopy structure of adenovirus allows refinement of capsid protein assignments. *J. Virol.*, 80(24):12049–12059, 2006.

[117] A. N. Tikhonov. Regularization of incorrectly posed problems. *Soviet Math. Dokl.*, 4:1624–1627, 1963.

[118] A. N. Tikhonov. Solution of incorrectly formulated problems and the regularization method. *Soviet Math. Dokl.*, 4:1035–1038, 1963.

[119] A. N. Tikhonov and V. Y. Arsenin. *Solutions of Ill-Posed Problems*. Winston, Washington, D.C., 1977.

[120] A. N. Tikhonov, A. S. Leonov, and A. G. Yagola. *Nonlinear Ill-Posed Problems, Volumes I and II*. Chapman and Hall, London, 1998.

[121] J. Tomlin. Software for LSQR (C++ version), 2007. http://www.stanford.edu/group/SOL/software/lsqr.html.

[122] Y. Tsaig and D. L. Donoho. Extensions of compressed sensing. *Signal Processing*, 86(3):549–571, 2006.

[123] R. S. Tuminaro, M. A. Heroux, S. A. Hutchinson, and J. N. Shadid. Official Aztec user's guide: Version 2.1. Technical report, Sandia National Laboratories, 1999.

[124] J. M. Varah. Pitfalls in the numerical solution of linear ill-posed problems. *SIAM J. Sci. Stat. Comp.*, 4(2):164–176, 1983.

[125] C. R. Vogel. Optimal choice of a truncation level for the truncated SVD solution of linear first kind integral equations when data are noisy. *SIAM J. Numer. Anal.*, 23(1):109–117, 1986.

[126] C. R. Vogel. An overview of numerical methods for nonlinear ill-posed problems. In H. W. Engl and C. W. Groetsch, editors, *Inverse and Ill-Posed Problems*, pages 231–245. Academic Press, Boston, 1987.

[127] C. R. Vogel. Non-convergence of the L-curve regularization parameter selection method. *Inverse Problems*, 12(4):535–547, 1996.

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

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

[130] P. Wendykier, N. Raghunath, T. L. Faber, J. Chung, S. Knepper, J. G. Nagy, and J. R. Votaw. Deblurring PET images using motion information. Technical Report TR-2010-028, Emory University, 2010.