Iterative Methods for Image Deblurring: A Matlab Object Oriented Approach

James G. Nagy* Katrina Palmer† Lisa Perrone‡

October 24, 2003

Abstract

In iterative image restoration methods, implementation of efficient matrix vector multiplication, and linear system solves for preconditioners, can be a tedious and time consuming process. Different blurring functions and boundary conditions often require implementing different data structures and algorithms. A complex set of computational methods is needed, each likely having different input parameters and calling sequences. This paper describes a set of Matlab tools that hide these complicated implementation details. Combining the powerful scientific computing and graphics capabilities in Matlab, with the ability to do object oriented programming and operator overloading, results in a set of classes that is easy to use, and easily extensible.

AMS Subject Classification: 65F20, 65F30

Key words: image restoration, ill-posed problem, regularization, Matlab, object-oriented programming, iterative methods, preconditioning

1 Introduction

Image restoration is an example of a linear ill-posed problem, which is often modeled as [4]

$$\mathbf{b} = A\mathbf{x} + \mathbf{n}\,. \tag{1}$$

Here $\mathbf{b}$ is a vector representing the blurred image, A is a large, usually ill-conditioned matrix that models the blurring operation, and $\mathbf{n}$ is a vector that models additive noise. The aim is to compute an approximation of the vector $\mathbf{x}$, which represents the image of the original scene. In most cases, the blur is generally much more significant than the noise, and thus the emphasis is on removing the blur.

Because of the large dimensions of the linear system, iterative methods are typically used to compute approximations of $\mathbf{x}$. These iterative methods include a variety of conjugate gradient type methods [12], the expectation-maximization method (sometimes referred to as the Richardson-Lucy method) [4], and many others [10]. Regularization can be enforced in a variety of ways, including

*Department of Mathematics and Computer Science, Emory University. nagy@mathcs.emory.edu This work was supported by the National Science Foundation under Grant DMS 00-75239.

†Department of Mathematics and Computer Science, Emory University. kpalmer@mathcs.emory.edu

‡Department of Mathematics and Computer Science, Emory University. perrone@mathcs.emory.edu

Tikhonov [11], iteration truncation [12], total variation [32], [23], as well as mixed approaches [22]. Since any one iterative method is not optimal for all image restoration problems, the study of iterative methods is an important and active area of research. One important issue, for example, is the development of reliable stopping criteria. Although significant progress has been made, much more work is needed; the purpose of this paper is to describe a set of software tools we have developed that will facilitate research in these areas.

In this paper we consider iterative methods that have the following general form:

$\mathbf{x}_0$ = initial estimate of $\mathbf{x}$
for $j = 0, 1, 2, \ldots$
- $\mathbf{x}_{j+1}$ = computations involving $\mathbf{x}_j$, A, a preconditioner matrix, P, and other intermediate quantities
- determine if stopping criteria are satisfied

end

The specific computational operations that are required to update $\mathbf{x}_{j+1}$ at each iteration depend on the particular iterative scheme being used, but the most intensive part of these computations usually involves matrix vector products with A and linear system solves with the preconditioner P.

Although efficient algorithms for these operations, in the context of structured linear systems arising in image restoration, are fairly well known (see, for example [6]), specific implementation details depend on the blurring operator as well as on the type of boundary condition being used. These issues dictate the structures of A and P, and in particular, which of several fast algorithms should be used for the matrix vector multiplications and preconditioner solves.

If we want to test an iterative method using a variety of blurring operators and boundary conditions, and be able to use problems arising in both two and three dimensional imaging applications, then we must have a complex set of computational methods at our disposal, each likely having different input parameters and calling sequences. In addition, preconditioning for iterative regularization methods requires the delicate choice of a "truncation" parameter that controls a tradeoff between fast convergence and stability of the iterations (see [14, 15], as well as Section 3 for further details). Even if we use a high level computing environment such as Matlab, prototyping and testing (new) iterative algorithms can be very time consuming. In this paper we describe a set of tools, which use Matlab's object oriented programming capabilities [24], that can greatly simplify this process. Our approach uses the power of operator overloading to hide the details from the user; several examples are provided to illustrate the ease of using these objects. The code and data for these examples may be obtained from our webpage [1]. These codes will be included as a small subset of a more extensive Matlab package for ill-posed problems being developed by Hansen and Jacobsen [17].

2 Matrix Vector Multiplications

In this section we consider matrix vector multiplication with the "blurring" matrix A. The blur can come from a variety of sources, such as atmospheric turbulence, out of focus lens, and motion blurs. Typically the blur is described mathematically with a *point spread function* (PSF); that is, a function that specifies how points in the image are distorted. In some cases a functional

representation of the PSF is known, but in most cases it is constructed experimentally from the imaging system by generating images of "point sources". What constitutes a point source depends on the application. For example, in atmospheric imaging, the point source can be a single bright star [19]. In microscopy, though, the point source is typically a fluorescent microsphere having a diameter which is about half the diffraction limit of the lens [9]. In this paper, along with the image that is to be restored, we assume that we are given an image of one or more point sources. It is this PSF image that will be used to construct the matrix A. Therefore, in the remainder of the paper we will call A a **psfMatrix**.

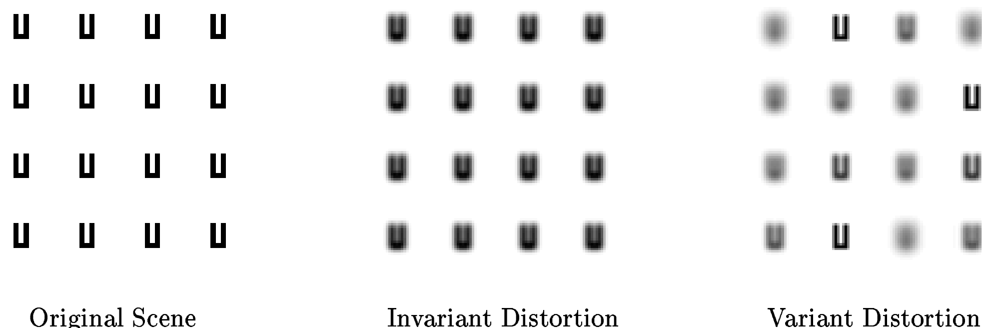
The structure of the **psfMatrix** A, and therefore the implementation details of matrix vector multiplications, depends on several factors. In particular, we will need to classify the blur as either *spatially invariant* or *spatially variant*, and we will need to know what kind of *boundary condition* is to be used.

2.1 Classifying blurring functions

One necessary piece of information we need to know is whether the blur is classified to be *spatially invariant* or *spatially variant*:

> *Spatially invariant* means that the blur is independent of position. That is, a blurred object will look the same regardless of its position in the image.
>
> *Spatially variant* means that the blur does depend on position. That is, an object in an observed image may look different if its position is changed.

To illustrate the difference in these types of blurs, Figure 1 shows an original scene containing the same object repeated in various regions of the image, along with examples of distorted images caused by spatially invariant and spatially variant PSFs.



Original Scene Invariant Distortion Variant Distortion

Figure 1: Example of spatially invariant and spatially variant distortions.

2.2 Boundary conditions

Images are shown only in a finite region, but points near the boundary of a blurred image are likely to have been affected by information outside the field of view. Since this information is not available, for computational purposes, we need to make some assumptions about the *boundary conditions*. In image processing, one of three common assumptions is usually made:

- *Periodic* boundary conditions imply that the image repeats itself endlessly in all directions. That is, we assume the image X has been extracted from a larger image which looks like:
$$\begin{array}{ccc} X & X & X \\ X & X & X \\ X & X & X \end{array}$$

- *Zero* boundary conditions imply a black boundary, so that the pixels outside the borders of the image X are all zeros. That is, we assume the image X has been extracted from a larger image which looks like:
$$\begin{array}{ccc} 0 & 0 & 0 \\ 0 & X & 0 \\ 0 & 0 & 0 \end{array}$$

- *Reflexive* boundary conditions imply that the scene outside the image boundaries is a mirror image of the scene inside the image boundaries. That is, we assume the image X has been extracted from a larger image which looks like:
$$\begin{array}{ccc} X_{rc} & X_r & X_{rc} \\ X_c & X & X_c \\ X_{rc} & X_r & X_{rc} \end{array}$$
where X_c is obtained by "flipping" the columns of X, X_r is obtained by "flipping" the rows of X, and X_{rc} is obtained by "flipping" the rows and columns of X.

Given the boundary condition, and the classification of the blur, we are ready to describe the structure of the **psfMatrix**, A.

2.3 Spatially invariant psfMatrix

In many applications, the PSF is assumed to be, or is approximated by, a spatially invariant model, since computations are generally easier to implement than for spatially variant blurs. If we assume that the blur is spatially invariant, then the PSF is represented by the image of a single point source. In this case, the structure of A depends on the boundary condition:

- Periodic boundary conditions imply that A is a block circulant matrix with circulant blocks (BCCB) [2].

- Zero boundary conditions imply that A is a block Toeplitz matrix with Toeplitz blocks (BTTB) [14, 20, 31].

- Reflexive boundary conditions imply that A is a sum of a BTTB matrix, a block Hankel matrix with Hankel blocks (BHHB), a BTHB matrix, and a BHTB matrix [28].

In the first case, matrix vector multiplication is done using the two dimensional discrete Fourier transform. All BCCB matrices can be written as:

$$C = \mathcal{F}^* \Lambda \mathcal{F},$$

where $\mathcal{F}$ is the two-dimensional discrete Fourier transform matrix, $\mathcal{F}^*$ is the complex conjugate transpose of $\mathcal{F}$, and Λ is a diagonal matrix containing the eigenvalues of C. Moreover, using

properties of the matrix $\mathcal{F}$ it is not difficult to show that the eigenvalues of C can be obtained by computing a two-dimensional FFT of the first column of C. Thus, computing $\mathbf{y} = C\mathbf{x}$ we use:

$$\mathbf{y} = \mathcal{F}^* \Lambda \mathcal{F} \mathbf{x}$$

or, we can think of this as:

$$\mathbf{y} = \texttt{ifft2D}(\texttt{fft2D}(\mathbf{c}) \texttt{ .* } \texttt{fft2D}(\mathbf{x}))\,,$$

where $\mathbf{c}$ is the first column of C, $\texttt{fft2D}(\cdot)$ and $\texttt{ifft2D}(\cdot)$ denote the fast Fourier transform and its inverse, and .* is element-wise multiplication.

In the second case, matrix vector multiplications are done by embedding A into a larger BCCB matrix, padding outside the borders of the image with an appropriate number of zeros, and then using FFTs. The amount of padding depends on the extent of the PSF. For very large problems, the codes use memory efficient methods called overlap-add and overlap-save, which partition the image domain into regions based on the size of the PSF. In linear algebra terms, the approach is equivalent to exploiting sparsity (bandedness) of the matrix A; see [25] for further details.

The third case is similar to zero boundary conditions, except that the values that are padded around the outside of the image are obtained by reflecting the pixel values from the inside of the image boundaries.

2.4 Spatially variant psfMatrix

A generic spatially variant blur would require a point source at every pixel location to fully describe the blurring operation. Since it is not possible to do this, even for small images, some approximations must be made. One such approach is to simply approximate by a spatially invariant blur, which can be done by using a single point source, or by averaging several point sources. Although this approach often works well, in some cases substantially better resolution can be obtained by using a more sophisticated model of the blur [3]. In this paper, we consider a large class of spatially variant PSFs which have the property that in small subregions of the image the PSF is essentially spatially invariant. We use an interpolation approach to piece together the invariant PSFs in each domain. In linear algebra terms, the matrix has the following structure:

$$A = \sum_{i=1}^{p} D_i A_i$$

where A_i are BTTB, except for those corresponding to the border, where the boundary conditions come into play, and D_i are diagonal matrices satisfying $\sum D_i = I$. Piecewise constant interpolation implies that the kth diagonal entry of D_i is one if the kth point is in region i, and zero otherwise. Linear interpolation was also considered in [26], but numerical experiments revealed very little improvement in resolution of the restored images. Therefore we only implement piecewise constant interpolation.

2.5 The psfMatrix class

From the previous section we see that the kind of information needed to construct a **psfMatrix**, as well as the implementation details of matrix vector multiplications, vary depending on the type of blur and boundary condition. For computational purposes, we therefore define a **psfMatrix** object, which can be described as a structure containing five fields:

- **psf** is an object used to standardize the definition of a point spread function. The specific details of this object are not important, but it contains, in particular, the PSF images.
- **matdata** is the actual data needed to do the matrix vector multiplications. It contains the eigenvalues of the BCCB matrix in which A is embedded.
- **type** is a character string that indicates if the blur is *invariant* or *variant*. The type is determined by the number of PSFs used to construct A; if only one PSF is used, then the type is set to *invariant*, otherwise it is set to *variant*.
- **boundary** is a character string indicating the type of boundary condition being used.
- **transpose** is an integer flag that indicates if the matrix has been transposed. For spatially variant problems, the algorithms for implementing Ax and $A^T x$ are different, but no data movement is needed. This flag, therefore, simply indicates which algorithm is to be used.

The syntax used to create a **psfMatrix** can take one of several forms, including:

- `A = psfMatrix(PSF);`
- `A = psfMatrix(PSF, boundary);`
- `A = psfMatrix(PSF, center);`
- `A = psfMatrix(PSF, center, boundary);`

where

- `PSF` is either an array containing a single PSF image, or a cell array containing one or more PSF images. Further details about this cell array are provided below.
- `boundary` is a character string indicating the desired boundary condition. Choices are 'zero', 'periodic' or 'reflexive'. If one is not specified, the default is to use zero boundary conditions.
- `center` is either an array specifying the pixel location of the point source for a single PSF image, or it is a cell array containing the pixel locations of point sources for one or more PSF images. If one is not specified, the default is to use the pixel location(s) of the maximum entry in the PSF(s).

Although more detailed examples of using a **psfMatrix** in iterative methods are given in Section 4, it may be useful to take a quick look at how easy it is to set up and use a **psfMatrix**.

Example 2.1. Constructing a **psfMatrix** for a spatially invariant problem only requires that we are given an array containing a PSF image. If this array is denoted as `PSF`, then the following statement will construct A using zero boundary conditions:

```
>> A = psfMatrix(PSF);
```

Alternatively, if we want to use one of the other boundary conditions, we can construct A using one of the following statements:

```
>> A = psfMatrix(PSF, 'reflexive');
>> A = psfMatrix(PSF, 'periodic');
```

Example 2.2. Constructing a **psfMatrix** for a spatially variant problem requires that we are given a cell array containing PSF images in various regions of the (blurred) image domain. The dimension of the cell array, and the size of the PSFs in each cell, are both assumed to be the same as the region partitioning of the image. That is, for example, if the image is partitioned into 6×6 regions of size $k \times k$ each, and a PSF is taken from each region, then we store all of the $k \times k$ PSF images in a 6×6 cell array. If this cell array is denoted as `PSF`, then we can use any of the same statements used in the previous example to construct A.

Example 2.3. By overloading the $*$ operator, matrix vector multiplications with A can be written as:

```
>> b = A * x;
```

Here it is assumed that `x` is an array containing an image, and the result, `b`, is an image having the same dimension as `x`. For example, suppose `PSF` and `x` are the images shown in Figure 2, and we compute:

```
>> A1 = psfMatrix(PSF);
>> b1 = A1 * x;
>> A2 = psfMatrix(PSF,'periodic');
>> b2 = A2 * x;
>> A3 = psfMatrix(PSF,'reflexive');
>> b3 = A3 * x;
```

Then the images `b1`, `b2` and `b3` are shown in Figure 3.

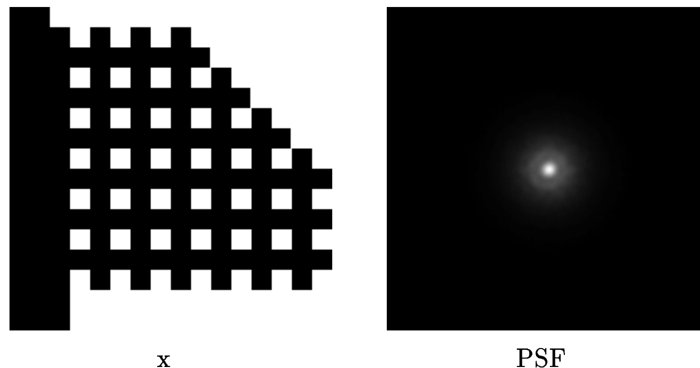


Figure 2: True Image and Point Spread Function

Example 2.4. Once the **psfMatrix**, A, is constructed, then we are ready to use a truncated iteration approach for computing a regularized solution. If, however, we want to use an iterative method to compute a Tikhonov regularized solution, then a little more work needs to be done.

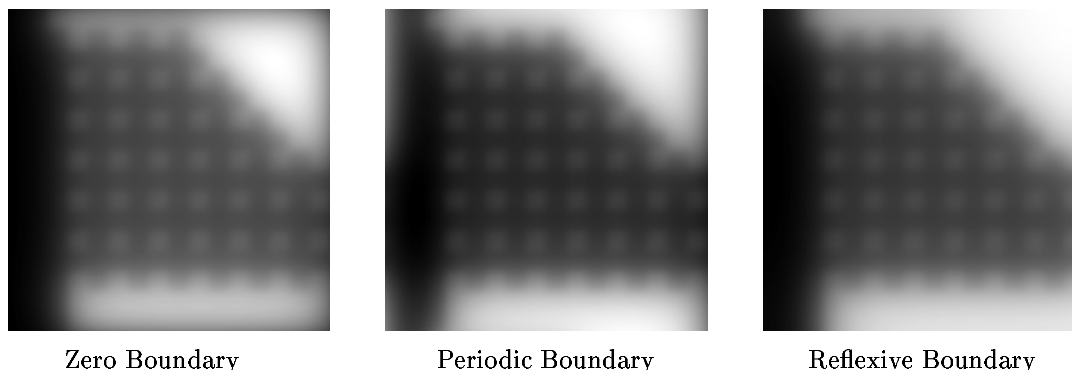


Figure 3: Example of different boundary conditions.

When using Tikhonov regularization, we want to compute a solution of the damped least squares problem:

$$\min \left\| \begin{bmatrix} A \\ \lambda L \end{bmatrix} \mathbf{x} - \begin{bmatrix} \mathbf{b} \\ 0 \end{bmatrix} \right\|_2 = \min ||\hat{A}\mathbf{x} - \hat{\mathbf{b}}||_2 \,,$$

where the regularization operator, L, is usually chosen to be the identity matrix or differentiation matrix. In most cases, L can be thought of as a **psfMatrix**, where the "PSF" is the convolution kernel for the operator. For example, when using the identity matrix, the corresponding "PSF" is simply the scalar 1, and when using the two-dimensional (discrete) Laplacian, the corresponding "PSF" is the array:

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

The **psfMatrix** class can easily be used to construct the matrix for the above damped least squares problem. In particular, given the PSF image, `PSF`, and a regularization parameter, `lambda`, the following Matlab statements will construct the damped least squares matrix, `Ahat`, for use in iterative methods:

```
>> A = psfMatrix(PSF);
>> L = psfMatrix(1);
>> Ahat = [A; lambda*L];
```

If we want to use the Laplacian for regularization, we instead use:

```
>> A = psfMatrix(PSF);
>> L = psfMatrix([0 -1 0; -1 4 -1; 0 -1 0]);
>> Ahat = [A; lambda*L];
```

Further examples given in Section 4 show how to use the **psfMatrix** class in iterative methods, but first we need to discuss preconditioning.

3 Preconditioning

Preconditioning is often used with Krylov subspace iterative methods, such as conjugate gradients [5], to accelerate the rate of convergence; that is, to reduce the number of iterations needed to compute a good approximation of the solution. Preconditioning is often presented in the context of solving linear systems. Suppose we are to use a conjugate gradient method to solve $A\mathbf{x} = \mathbf{b}$. The standard approach to preconditioning is to construct a matrix, P, that satisfies the following properties:

- It should be relatively inexpensive to construct P.
- It should be relatively inexpensive to solve linear systems of the form $P\mathbf{z} = \mathbf{w}$.
- The preconditioned system should satisfy $P^{-1}A \approx I$ in the sense that the singular values of $P^{-1}A$ are clustered around 1.

The first two requirements are related to the additional computational costs of preconditioning; constructing P is a one time cost, where as linear system solves with P (and with P^T for non-symmetric problems) are required at each iteration. The last requirement determines the speed of convergence; more singular values clustered around one, as well as tighter clusters, usually implies faster convergence.

For ill-posed problems, we must incorporate regularization, and therefore this standard approach may need some modifications. In the case of Tikhonov regularization, we simply apply the standard approach outlined above to the linear system $\hat{A}\mathbf{x} = \hat{\mathbf{b}}$, where

$$\hat{A} = \begin{bmatrix} A \\ \lambda L \end{bmatrix}, \quad \hat{\mathbf{b}} = \begin{bmatrix} \mathbf{b} \\ 0 \end{bmatrix}.$$

However, in the case when regularization is enforced by truncating the iterations, the situation is a bit more delicate. This is explained in more detail in the following subsection.

3.1 Preconditioned iterative regularization

For ill-posed problems, such as image restoration, the matrix A is severely ill-conditioned. If P is a good approximation to A, then P is likely to be very ill-conditioned. In this case, inaccuracies in the data will be highly amplified with the initial linear system solve with P. To see this more clearly, suppose we were able to compute the singular value decomposition (SVD) of A:

$$A = U\Sigma V^T$$

where U and V are orthogonal matrices, and Σ is a diagonal matrix. Consider the following:

- For well-posed problems, the ideal preconditioner would be $P = A = U\Sigma V^T$, and thus solutions of $P\mathbf{z} = \mathbf{w}$ are given by:

  $$\mathbf{z} = V\Sigma^{-1}U^T\mathbf{w}.$$

  For ill-posed problems, though, it is well known that this approach will substantially amplify noise and other data errors in $\mathbf{w}$. In this case the iterative method has little hope of recovering, and it is very unlikely that a reasonable approximation of the solution will be computed.

- We could use the idea of a truncated SVD (TSVD) to regularize the preconditioner solves; that is, we could compute solutions of the form

$$\mathbf{z} = V\Sigma^{\dagger}U^{T}\mathbf{w}\,,$$

where $\Sigma^{\dagger} = \text{diag}(\frac{1}{\sigma_1}, \ldots, \frac{1}{\sigma_k}, 0, \ldots 0)$. The problem with this approach is that the preconditioner is singular, and therefore we cannot guarantee that $\mathbf{z} \neq 0$, even when $\mathbf{w} \neq 0$. The preconditioned conjugate gradient method will break down if $\mathbf{z} = 0$.

- An alternative approach proposed in [15] is to construct a preconditioner of the form $P_\tau = U\Sigma_\tau V^T$, where $\Sigma_\tau = \text{diag}(\sigma_1, \ldots, \sigma_k, 1, \ldots 1)$. In this case, we simply compute solutions as:

$$\mathbf{z} = V\Sigma_\tau^{-1}U^{T}\mathbf{w}\,.$$

To understand why this last approach works, we need to understand a little about truncated iteration regularization. It can be shown (see [13]) that the early iterations of the conjugate gradient method filter out components of the solution corresponding to the small singular values of the matrix. That is, the early iterations tend to reconstruct mostly the good part of the solution, and noise components of the solution are filtered out. It is this part of the iteration that we want to accelerate. At some point, though, the noise components start to be reconstructed, and the iterations begin to be corrupted with noise; this part of the iteration we do **not** want to accelerate.

For preconditioned conjugate gradients, it is the singular values of the preconditioned system $P^{-1}A$ that we must consider. By clustering all of the singular values around one, we no longer have the information to distinguish between the signal and noise subspaces. However, if we use the preconditioner P_τ, then the preconditioned system has the form:

$$P_\tau^{-1}A = V\Delta V^T\,,$$

where $\Delta = \text{diag}(1, \ldots, 1, \sigma_{k+1}, \ldots, \sigma_n)$. In this case, the large singular values (i.e., those corresponding to the signal subspace) are clustered at one, and are well separated from the small singular values (those corresponding to the noise subspace). In this special case, one iteration computes a regularized solution.

One final point about this preconditioning approach needs to be addressed: How do we choose the cutoff parameter, τ? As with deciding the cutoff parameter for TSVD regularization, there is not one best approach. The three most popular approaches that have been used in the literature include:

- *Picard Condition*: The index where the Fourier coefficients level off indicates where the random errors start to dominate the right hand side, so the magnitude of the eigenvalue corresponding to this index is the truncation parameter. See [15] for more details.

- *L-Curve*: The corner of the L-curve gives a good balance of the solution size and residual size, so the τ that corresponds to the corner is the truncation parameter. See [18] for more details.

- *Generalized Cross Validation (GCV)*: We assume that a solution computed on a reduced set of data points should give a good estimate of missing points. The GCV method finds a function of τ that measures the errors in these estimates. The minimum of this GCV function corresponds to the truncation parameter. See [18] for more details.

We remark that proper choice of the truncation parameter is important. If chosen too small, the iterations will be too corrupted with noise, and it will be impossible to compute a good solution. If chosen too large, then very little preconditioning is done, and convergence speed will not be improved. Our main goal is to compute a good approximation of the solution, so we should be conservative when choosing τ; that is, we should carefully avoid choosing parameters that are too small.

Of course if it is feasible to compute the SVD of A, then we do not need to use an iterative method. But the discussion in this section does give us the basic idea of how to precondition ill-posed problems: We should attempt to construct a preconditioner that clusters large singular values around one, and leaves small singular values alone.

3.2 Preconditioners for Image Restoration

As we saw in Section 2, the matrices that arise in image restoration are highly structured, involving circulant, Toeplitz and Hankel matrices. Preconditioning such matrices has been thoroughly investigated in the literature; see the survey paper [6] and the references therein. Moreover, many of these approaches have been applied to image restoration [15, 26].

Although a variety of approaches to preconditioning have been proposed, the most popular is to use circulant approximations, and that is the approach we consider in this paper. But it should be noted that, in principal, many of the other fast transform based preconditioners can be used as well.

For spatially invariant blurs, our approach is to approximate the **psfMatrix**, A, with a BCCB matrix. Two popular approximations include:

$$\min ||A - P||_F \quad \text{and} \quad \min ||A - P||_1$$

where subscript F denotes the Frobenius norm, and where the minimizations are done over all BCCB matrices, P. It is very inexpensive to construct these preconditioners; see [6, 7, 8, 30] for more details. We remark that for spatially invariant blurs, using periodic boundary conditions implies that A is a BCCB matrix, and therefore in this case $A = P$. Therefore, a BCCB preconditioner for spatially invariant blurs (with zero or reflexive boundary conditions) is essentially a **psfMatrix** with periodic boundary conditions.

For spatially variant blurs, each PSF requires its own preconditioner. In this case, as with invariant blurs, the preconditioner for any given PSF is (equivalent to) a **psfMatrix** with periodic boundary conditions.

An important property of BCCB matrices is that their spectral decomposition is easy to compute using FFTs. With this spectral decomposition, we can easily construct preconditioners for both the damped least squares problem in Tikhonov regularization, as well as for truncated iteration regularization:

- For Tikhonov regularization, we use the approach described in [12]:

  $$P = \mathcal{F}^* \Lambda \mathcal{F} = \mathcal{F}^* \left(|\Lambda_A|^2 + \lambda^2 |\Lambda_L|^2 \right)^{1/2} \mathcal{F}\,,$$

  where $P_A = \mathcal{F}^* \Lambda_A \mathcal{F}$ and $P_L = \mathcal{F}^* \Lambda_L \mathcal{F}$ are, respectively, the BCCB approximations to A and L.

- For truncated iteration regularization, we use the approach outlined in the previous subsection, except we replace the SVD of P with its spectral decomposition.

3.3 The psfPrec class

In this subsection we define a **psfPrec** object that will construct data and information needed to solve linear systems with the BCCB preconditioner, P. A **psfPrec** can be described as a structure containing three fields, very similar to **psfMatrix**:

- **matdata** is the actual data needed to do linear system solves; that is, it contains the eigenvalues of P.
- **type** is a character string indicating if the blur is *invariant* or *variant*.
- **transpose** an integer flag that indicates if P has been transposed.

The syntax used to create a **psfPrec** can take one of several forms, including:

- `P = psfPrec(A, b);`
- `P = psfPrec(A, b, tol);`
- `P = psfPrec(A, b, 'help');`

where

- `A` is a **psfMatrix**, or, in the case of Tikhonov regularization, a concatenation of **psfMatrix** objects. (See below for some examples.)
- `b` is the blurred image to be restored. This input parameter is needed only because the `psfPrec` codes utilize the size of `b`.
- `tol` is the truncation tolerance for the preconditioner. Eigenvalues below `tol` are replaced by the value one. If `tol` is not specified, then the default is `tol = 0`; that is, no truncation is used.
- If `'help'` is specified, a truncation parameter will be chosen using the GCV method. In addition, diagnostic plots (the GCV curve, Picard condition, and L-Curve) will be displayed to help the user determine if GCV has chosen an appropriate value for `tol`.

Example 3.1. Constructing a **psfPrec** only requires that we are given a **psfMatrix** and its right hand side `b`. If this **psfMatrix** is denoted as `A`, then the following statement will construct P using a tolerance of zero:

```
>> P = psfPrec(A,b);
```

However, if we want to specify a particular value for the tolerance, we can construct P using the following statement:

```
>> P = psfPrec(A, b, tol);
```

Alternately, if we want the tolerance chosen for us by the GCV function, we can construct P using the following:

```
>> P = psfPrec(A, b, 'help')
```

Using `'help'` produces a plot with some information about the chosen truncation tolerance. In the next section we provide a specific example to show how this information can be used to determine if an appropriate tolerance has been found.

Example 3.2. To construct a **psfPrec** for Tikhonov regularization, we concatenate A and `lambda`*L, where A and L are `psfMatrices`. The following will construct P:

```
>> P = psfPrec([A; lambda*L], b)
```

Example 3.3. We have overloaded the \ operator, so linear system solves with P can be written as:

```
>> z = P \ w;
```

4 Iterative Methods using psfMatrix and psfPrec

Now that we've gathered the tools together, it's time to consider some iterative methods for solving $A\mathbf{x} = \mathbf{b}$ with a **psfMatrix** A. For instance, consider the preconditioned conjugate gradient method for least squares problems (PCGLS) [5]:

Given: Matrix A, preconditioner P, initial guess, $\mathbf{x}_0$
Set $\mathbf{r} = \mathbf{b} - A\mathbf{x}_0$, $\mathbf{s} = P^{-T}(A^T\mathbf{r})$, and $\gamma = ||\mathbf{s}||_2^2$;
for $k = 1, 2, \ldots$
  if ($k == 1$), set $\mathbf{p} = \mathbf{s}$;
  otherwise compute $\beta = \gamma \;/\; \gamma_{\text{old}}$ and $\mathbf{p} = \mathbf{s} + \beta * \mathbf{p}$;
  $\mathbf{t} = P^{-1}\mathbf{p}$;
  $\mathbf{q} = A * \mathbf{t}$;
  $\alpha = \gamma \;/\; ||\mathbf{q}||_2^2$;
  $\mathbf{x} = \mathbf{x} + \alpha * \mathbf{t}$;
  $\mathbf{r} = \mathbf{r} - \alpha * \mathbf{q}$;
  $\mathbf{s} = P^{-T}(A^T\mathbf{r})$;
  $\gamma_{\text{old}} = \gamma$, $\gamma = ||\mathbf{r}||_2^2$;
  determine if stopping criteria are satisfied
end

We note that during each iteration there are two matrix-vector multiplies and two preconditioner system solves. With the **psfMatrix** and **psfPrec** classes, a Matlab implementation of an iterative method like this is straightforward. In particular, with the * and \ operators overloaded, a Matlab implementation to compute $\mathbf{s}$, for example, can be done as:

$$\mathbf{s} = P' \setminus (A' * \mathbf{r})$$

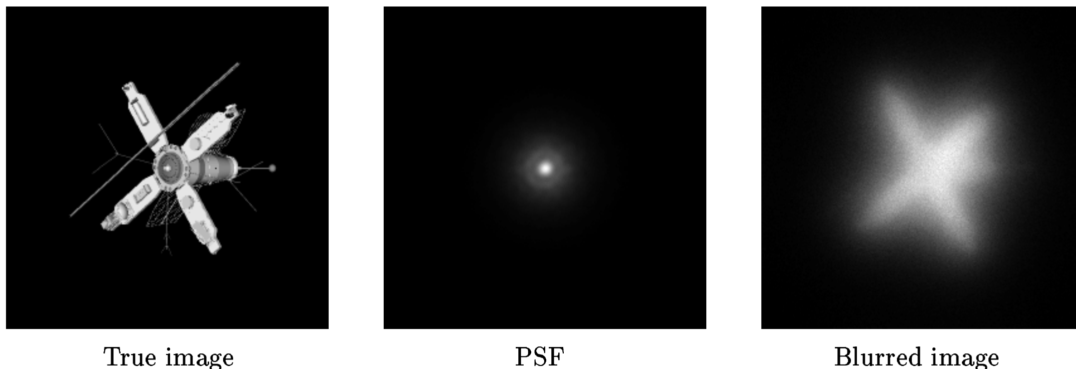
As mentioned previously, our efficient implementation of these operations makes a computation such as this relatively fast.

We have implemented several iterative methods, including:

- CGLS and PCGLS: For use on nonsymmetric problems, or when Tikhonov regularization is used. CGLS is the conjugate gradient method for least squares problems without preconditioning; see [5].
- MR2 and PMR2: These are minimum residual type methods (unpreconditioned and preconditioned), for use on symmetric problems; see [12, 14].
- MRNSD and PMRNSD: These are iterative regularization methods that can be used to enforce nonnegativity in the computed solution; see [21, 27].

The implementations are similar, requiring the user to specify, at the very least, a matrix `A`, a right hand side vector `b`, and an initial guess, `x0`. If a preconditioned version is used, then a preconditioner, `P`, must also be specified. Additional inputs include a maximum number of iterations and a stopping tolerance.

In the following examples (for simplicity) we use only spatially invariant PSFs, but the extension to the variant case is straightforward (in terms of command-line entries, there is no obvious difference to the user). For the 2-dimensional examples, we use a set of test data, shown in Figure 4, which was developed at the US Air Force Phillips Laboratory, Lasers and Imaging Directorate, Kirtland Air Force Base, New Mexico. The images are from a computer simulation of a field experiment showing a blurred image of a satellite and the corresponding PSF, as taken from a ground based telescope. This data has been used widely in the literature for testing algorithms for ill-posed image restoration problems; see, for example [29].



True image PSF Blurred image

Figure 4: Satellite test data.

This test data, along with the following examples, can be obtained by downloading the software from our web page [1], and changing to the directory `./RestoreTools/Examples/`. If you do not start Matlab while in this directory, then you should first run the script `startup.m` so that the search paths are properly set.

Example 4.1. The following code solves $A\mathbf{x} = \mathbf{b}$ under zero boundary conditions with an unknown truncation tolerance for preconditioning. The blurred image `b` is used as the initial guess, and the prescribed number of iterations is 15.

```
>> load satellite
>> A = psfMatrix(PSF);
>> P = psfPrec(A, b, 'help');
>> x0 = b;
>> x = PCGLS( A, P, b, x0, 15);
```

Using the string `'help'` in constructing the preconditioner produces a plot shown in Figure 5. The truncation tolerance for the preconditioner was chosen as the minimum of the GCV plot, which corresponds nicely with the corner of the L-curve, and with the flat part of the Picard condition plot, indicating a good tolerance. The computed solution, `x`, is shown in Figure 6.

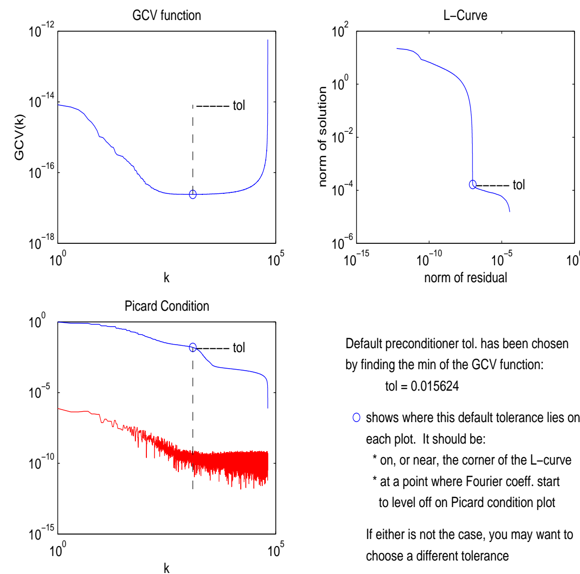


Figure 5: Plot produced when using `'help'` in `psfPrec`.

Example 4.2. In this example, we use Tikhonov regularization,

$$\min \left\| \begin{bmatrix} A \\ \lambda L \end{bmatrix} \mathbf{x} - \begin{bmatrix} \mathbf{b} \\ 0 \end{bmatrix} \right\|_2 = \min ||\hat{A}\mathbf{x} - \hat{\mathbf{b}}||_2 \,,$$

with $\lambda = .00023$ and L the identity matrix. The number of iterations is 35, and `b` is the initial guess.

```
>> load satellite
>> A = psfMatrix(PSF);
>> L = psfMatrix(1);
>> lambda = .00023;
>> Ahat = [A; lambda*L];
>> bhat = [b; zeros( size(b) )];
>> P = psfPrec(Ahat, b);
>> x0 = b;
>> x = PCGLS(Ahat, P, bhat, x0, 35);
```

Because we used Tikhonov regularization in this example, we used the default tolerance, 0, for the preconditioner. The regularization parameter, λ, was chosen through experimentation; we do not address the topic of choosing Tikhonov regularization parameters in this paper. The computed solution, `x`, is shown in Figure 6.

Example 4.3. The satellite data was used in [14], where it was shown that algorithms for symmetric indefinite matrices, such as MR2 and PMR2, could be used by constructing a symmetric approximation to the **psfMatrix**. This can be done easily with our tools as follows:

```
>> load satellite
>> A = psfMatrix(PSF);
>> As = (A + A')/2;
>> P = psfPrec(As, b, 'help');
>> x0 = b;
>> x = PMR2(As, P, b, x0, 15);
```

The solution, `x`, is shown in Figure 6.

Example 4.4. As a last 2-dimensional example, we use a method that enforces nonnegativity at each iteration, PMRNSD [21, 27]. This can be done as follows:

```
>> load satellite
>> A = psfMatrix(PSF);
>> P = psfPrec(A, b, 'help');
>> x0 = b;
>> x = PMRNSD(A, P, b, x0, 10);
```

The solution, `x`, is shown in Figure 6.

In the next example, we work with a 3-dimensional test image. Our true, undistorted image, denoted in the following example by `x_true`, is a size $128 \times 128 \times 27$ MRI of a human brain, available in the Matlab Image Processing Toolbox. To produce $\mathbf{b}$, we build a $64 \times 64 \times 14$ Gaussian PSF using the function `psfGauss`, then convolve it with the MRI image. We then add 1% Gaussian noise to the result. Shown in figure 7 are 2-dimensional slices of the test data.

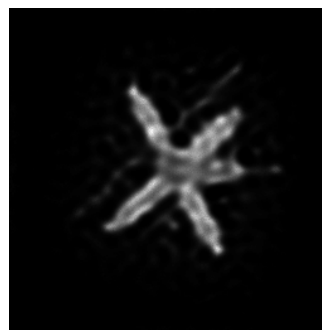
Example 4.5. Observe that the commands for this 3-D PCGLS restoration are fundamentally the same as those used in example 4.1, where we used PCGLS to deblur a 2-dimensional image.

```
>> load mri
>> x_true = double(squeeze(D));
>> PSF = psfGauss([64 64 14]);
>> A = psfMatrix(PSF);
>> b = A*x_true;
>> n = randn(size(b));
>> n = 0.01*n*norm(b(:))/norm(n(:));
>> b = b + n;
>> P = psfPrec(A, b, 0.01);
>> x0 = b;
>> x = PCGLS(A, P, b, x0, 25);
```

Iterative reg. (PCGLS)

Tikhonov Regularization

Iterative reg. (PMR2)

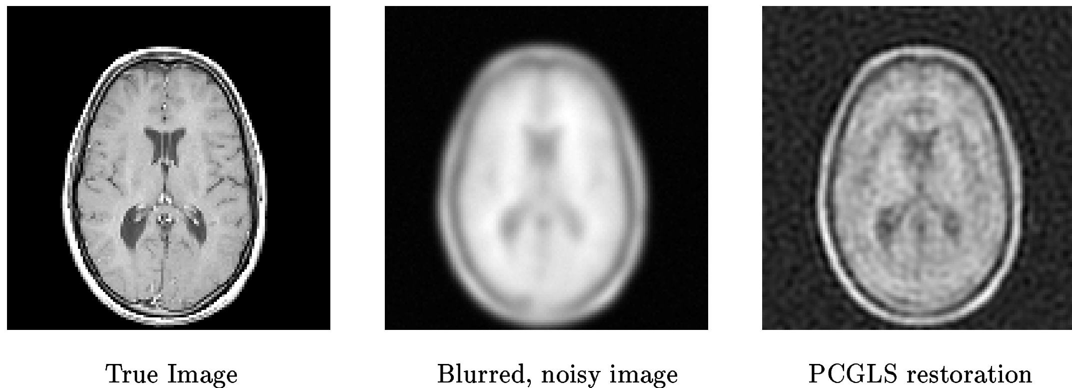


Iterative reg. (PMRNSD)

Figure 6: Reconstructions of the blurred image using various algorithms.

One layer (slice) of the solution, $\mathbf{x}$, is shown in Figure 7.

It should be noted that in this example, unlike those presented for the 2-dimensional image, a truncation tolerance was specified for the preconditioner. This was necessary because the GCV function initially produced 0.0 as the recommended truncation tolerance, which in turn produced an ineffective preconditioner (this was obvious from the GCV, L-curve, and Picard condition plots (not shown) that were displayed when the 'help' option was invoked). We selected the value 0.01 to correspond with the noise level, but in practice, further experimentation might lead to a more appropriate choice.



True Image   Blurred, noisy image   PCGLS restoration

Figure 7: MRI data. The slice shown corresponds to layer 15 in the 3-D image.

5 Concluding Remarks

Hansen's Matlab package, *Regularization Tools* [16], has been very influential in research on algorithms for ill-posed inverse problems. In particular, the toolbox contains several test problems and analysis tools that are often used in research papers, and the work has been heavily cited in the literature. *Regularization Tools*, though, was designed mainly for small scale problems, where the coefficient matrix, A, can be constructed explicitly, and where it is possible to compute an SVD of A. In its current form, it is not possible to directly use the toolbox on an application such as image restoration, since explicit construction of the matrix A is usually not feasible; an average size problem involving images with 256×256 pixels implies A would have dimension 65536×65536. Iterative methods can be used effectively, but efficient implementations of the basic kernels, such as construction of a preconditioner, matrix vector multiplications, and preconditioner solves, can be tedious and time consuming. In this paper we described a set of software tools that can easily be used in iterative image restoration methods. Using an object oriented design means that with very little work, a wide variety of problems can be solved, including ones with spatially invariant or spatially variant blurs, with different boundary conditions, as well as for two and three dimensional images. It is our hope that this work will help facilitate research in image restoration, and more generally, in research efforts on iterative algorithms for large scale ill-posed inverse problems.

Finally, we remark that for separable blurs (such as a Gaussian PSF), the **psfMatrix** can be represented as a Kronecker product, and thus direct methods such as the singular value decomposition (SVD) can be used. Although it is not discussed in this paper, the current version of our codes allows for some limited Kronecker product computations. For example, if A is a **psfMatrix**, then

`svd(A)` exploits Kronecker product structure to compute an SVD of A (if the PSF is not separable, then a separable approximation is computed). In future work we plan to include several additional tools and algorithms that exploit the Kronecker product structure of separable blurs. Updates of this work will be posted on our webpage [1].

Acknowledgments: We are very grateful to Per Christian Hansen and Michael Jacobsen for many helpful suggestions during the development of the codes.

References

[1] RestoreTools: An Object Oriented Matlab Package for Image Restoration, 2002. http://www.mathcs.emory.edu/~nagy/RestoreTools.

[2] H. Andrews and B. Hunt. *Digital Image Restoration.* Prentice-Hall, Englewood Cliffs, NJ, 1977.

[3] M. R. Banham and A. K. Katsaggelos. Digital image restoration. *IEEE Signal Processing Magazine*, pages 24–41, March 1997.

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

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

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

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

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

[9] A. Diaspro, M. Corosu, P. Ramoino, and M. Robello. Two-photon excitation imaging based on a compact scanning head. *IEEE Engineering in Medicine and Biology*, pages 18–22, September/October 1999.

[10] G. H. Golub and C. Van Loan. *Matrix Computations, third edition.* Johns Hopkins Press, 1989.

[11] C. W. Groetsch. *The Theory of Tikhonov Regularization for Fredholm Integral Equations of the First Kind.* Pitman, Boston, 1984.

[12] M. Hanke. *Conjugate Gradient Type Methods for Ill-Posed Problems.* Pitman Research Notes in Mathematics, Longman Scientific & Technical, Harlow, Essex, 1995.

[13] M Hanke and P. C. Hansen. Regularization methods for large-scale problems. *Surv. Math. Ind.*, 3:253–315, 1993.

[14] M. Hanke and J. G. Nagy. Restoration of atmospherically blurred images by symmetric indefinite conjugate gradient techniques. *Inverse Problems*, 12:157–173, 1996.

[15] M. Hanke, J. G. Nagy, and R. J. Plemmons. Preconditioned iterative regularization. In L. Reichel, A. Ruttan, and R. S. Varga, editors, *Numerical Linear Algebra.* de Gruyter, Berlin, 1993.

[16] P. C. Hansen. Regularization tools: A Matlab package for the analysis and solution of discrete ill-posed problems. *Numerical Algorithms*, 6:1–35, 1994.

[17] P. C. Hansen and M. Jacobsen. Regularization tools: The milenium edition, in progress.

[18] 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.*, 14:1487–1503, 1993.

[19] J. W. Hardy. Adaptive optics. *Scientific American*, 270(6):60–65, 1994.

[20] J. Kamm and J. G. Nagy. Kronecker product and SVD approximation in image restoration. *Linear Algebra Appl.*, 284:177–192, 1998.

[21] L. Kaufman. Maximum likelihood, least squares, and penalized least squares for PET. *IEEE Trans. Med. Imag.*, 12:200–214, 1993.

[22] M. E. Kilmer and D. P. O'Leary. Choosing regularization parameters in iterative methods for ill-posed problems. *SIAM J. Matrix Anal. Appl.*, 22:1204–1221, 2000.

[23] S. Osher L. Rudin and E. Fatemi. Nonlinear total variation based noise removal algorithms. *Numerica D*, 60:259–268, 1992.

[24] The Math Works, Inc. *Using* MATLAB*, Version 6*, 2000.

[25] J. G. Nagy and D. P. O'Leary. Fast iterative image restoration with a spatially variant PSF. In F. T. Luk, editor, *Advanced Signal Processing Algorithms, Architectures, and Implementations VII*, volume 3162, pages 388–399. SPIE, 1997.

[26] J. G. Nagy and D. P. O'Leary. Restoring images degraded by spatially-variant blur. *SIAM J. Sci. Comput.*, 19:1063–1082, 1998.

[27] J. G. Nagy and Z. Strakoš. Enforcing nonnegativity in image reconstruction algorithms. In et. al. D. C. Wilson, editor, *Mathematical Modeling, Estimation, and Imaging*, volume 3461, pages 182–190. SPIE, 2000.

[28] M. K. Ng, R. H. Chan, and W.-C. Tang. A fast algorithm for deblurring models with Neumann boundary conditions. *SIAM J. Sci. Comput.*, 21:851–866, 1999.

[29] M. C. Roggemann and B. Welsh. *Imaging Through Turbulence.* CRC Press, Boca Raton, Florida, 1996.

[30] G. Strang. A proposal for Toeplitz matrix calculations. *Studies in Appl. Math.*, 74:171–176, 1986.

[31] H. J. Trussell. Convergence criteria for iterative restoration methods. *IEEE Trans. Acoust., Speech, Signal Processing*, 31:129–136, 1983.

[32] C. Vogel and M. Oman. A fast, robust algorithm for total variation based reconstruction of noisy blurred images. *IEEE Trans. Image Processing*, 1996.