Chapter 4

Solution of Matrix Equations

The discretization and testing of integral equations via the Method of Moments results in a linear system which must be solved numerically. Thus, an understanding of how to do this is necessary. Though engineers will likely not have to implement their own linear equation solver, it is useful to understand how such solvers work, what pitfalls exist, and what software libraries exist for helping in this task. In this chapter we will briefly describe common methods for solving linear equations such as Gaussian Elimination, scalar and block LU factorization, and iterative solvers.

4.1 Direct Methods

In this section we discuss Gaussian elimination (GE) and LU factorization, methods that allow a direct solution of a linear system via matrix factorization. These algorithms require a compute time of the order $O(N^3)$, where N is the number of unknowns. These methods work very well when N is relatively small, however for larger problems, the factorization time may grow prohibitively large, or computer memory may be simply exhausted. Even though computing power continues to grow year to year, these issues will always present challenges as engineers try to solve larger problems.

4.1.1 Gaussian Elimination

Gaussian elimination is a simple method of reducing a matrix to row echelon form through the use of elementary row operations. Once this is done, the unknown vector is obtained through simple back-substitution. To illustrate the method, let us form an

augmented matrix from the $N \times N$ matrix equation $\mathbf{Ax} = \mathbf{b}$, which can be written as

$$\left[\begin{array}{ccccc|c} A_{11} & A_{12} & A_{13} & \dots & A_{1N} & b_1 \\ A_{21} & A_{22} & A_{23} & \dots & A_{2N} & b_2 \\ A_{31} & A_{32} & A_{33} & \dots & A_{3N} & b_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ A_{N1} & A_{N2} & A_{N3} & \dots & A_{NN} & b_N \end{array}\right]. \tag{4.1}$$

We start with the first row and divide every entry by A_{11}, leaving a 1 on the diagonal. For every row below this, we then subtract a multiple of this row so that a zero remains in the first column. The result is

$$\left[\begin{array}{ccccc|c} 1 & A'_{12} & A'_{13} & \dots & A'_{1N} & b'_1 \\ 0 & A'_{22} & A'_{23} & \dots & A'_{2N} & b'_2 \\ 0 & A'_{32} & A'_{33} & \dots & A'_{3N} & b'_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & A'_{N2} & A'_{N3} & \dots & A'_{NN} & b'_N \end{array}\right]. \tag{4.2}$$

We repeat this same operation for the second row and all rows below that, until only ones remain on the diagonal and the lower triangle is zero. The original matrix equation now reads

$$\left[\begin{array}{ccccc|c} 1 & A'_{12} & A'_{13} & \dots & A'_{1N} & b'_1 \\ 0 & 1 & A'_{23} & \dots & A'_{2N} & b'_2 \\ 0 & 0 & 1 & \dots & A'_{3N} & b'_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & b'_N \end{array}\right]. \tag{4.3}$$

The solution to this system is obtained through the following back-substitution operations

$$x_N = b'_N \tag{4.4}$$

and

$$x_i = b'_i - \sum_{k=i+1}^{N} A'_{ik} x_k \qquad i < N. \tag{4.5}$$

The elimination requires a total of $N^3/3$ operations, and $N^2/2$ for the back-substitution. Note that the row operations leading to (4.3) modify the right-hand side vector as well. This is undesirable when dealing with multiple right-hand sides, as they must all be precomputed and available when performing the elimination. This limitation is avoided by using LU factorization.

4.1.1.1 Pivoting

When performing the row operations for Gaussian elimination, the values of the diagonal entries (the *pivots*) were not addressed. A problem arises when one of these entries is zero, and if the relative magnitude of the pivot is small, round-off errors may become a problem in the subsequent multiplications and subtractions. Because the augmented matrix system is not altered by exchanging any two rows, the current row can be exchanged with any of the ones below it that contain the "strongest" diagonal entry (one of greatest magnitude). This is referred to as *partial pivoting*. An exchange of rows *and* columns in the matrix is called *full pivoting*, and requires recording the permutations of the solution vector as well [1]. Many linear algebra packages employ a pivoting strategy for robustness, with partial pivoting being the most common.

4.1.2 LU Factorization

As previously discussed, a factorization that does not modify the right-hand side vector is more advantageous than Gaussian elimination. Therefore, let us consider a factorization of the matrix **A** into lower and upper triangular parts. We can write this as

$$\mathbf{LU} = \mathbf{A} \,, \tag{4.6}$$

where

$$\mathbf{LU} = \begin{bmatrix} L_{11} & 0 & 0 & \dots & 0 \\ L_{21} & L_{22} & 0 & \dots & 0 \\ L_{31} & L_{32} & L_{33} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ L_{N1} & L_{N2} & L_{N3} & \dots & L_{NN} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & U_{13} & \dots & U_{1N} \\ 0 & U_{22} & U_{23} & \dots & U_{2N} \\ 0 & 0 & U_{33} & \dots & U_{3N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & U_{NN} \end{bmatrix} . \tag{4.7}$$

Using this factorization, we can solve the following equation

$$\mathbf{Ax} = (\mathbf{LU})\mathbf{x} = \mathbf{L}(\mathbf{Ux}) = \mathbf{b} \tag{4.8}$$

by first solving the equation

$$\mathbf{Ly} = \mathbf{b} \,, \tag{4.9}$$

where **y** is obtained through the forward substitution

$$y_1 = \frac{b_1}{L_{11}} \,, \tag{4.10}$$

and

$$y_i = \frac{1}{L_{ii}} \left[b_i - \sum_{k=1}^{i-1} L_{ik} y_k \right] \qquad i > 1 \,. \tag{4.11}$$

We next solve

$$\mathbf{Ux} = \mathbf{y} \,, \tag{4.12}$$

where **x** is obtained through the back-substitution

$$x_N = \frac{y_N}{U_{NN}} \,, \tag{4.13}$$

and

$$x_i = \frac{1}{U_{ii}}\left[y_i - \sum_{k=i+1}^{N} U_{ik}x_k\right] \qquad i < N \,. \tag{4.14}$$

The remaining task is to determine the elements of **L** and **U**. If we carry out the matrix multiplication **LU**, we will have N^2 equations for the $N^2 + N$ unknowns comprising L_{ij} and U_{ij}. Because the diagonal is represented twice, we can specify N of the unknowns ourselves, so we choose

$$L_{ii} = 1 \quad i = 1, 2, \ldots, N \,. \tag{4.15}$$

Thus,

$$\mathbf{LU} = \begin{bmatrix} 1 & 0 & 0 & \ldots & 0 \\ L_{21} & 1 & 0 & \ldots & 0 \\ L_{31} & L_{32} & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ L_{N1} & L_{N2} & L_{N3} & \ldots & 1 \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & U_{13} & \ldots & U_{1N} \\ 0 & U_{22} & U_{23} & \ldots & U_{2N} \\ 0 & 0 & U_{33} & \ldots & U_{3N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & U_{NN} \end{bmatrix} , \tag{4.16}$$

and we can immediately set $U_{11} = A_{11}$. Multiplying **L** by the first column of **U** we obtain

$$\begin{aligned} L_{21} &= \frac{A_{21}}{U_{11}} \,, \\ L_{31} &= \frac{A_{31}}{U_{11}} \,, \\ &\cdots \end{aligned} \tag{4.17}$$

and doing the same with the second column of **U** yields $U_{12} = A_{12}$ and

$$\begin{aligned} U_{22} &= A_{22} - L_{21}U_{12} \,, \\ L_{32} &= \frac{1}{U_{22}}\left[A_{32} - L_{31}U_{12}\right] \,, \\ L_{42} &= \frac{1}{U_{22}}\left[A_{42} - L_{41}U_{12}\right] \,, \\ &\cdots \end{aligned} \tag{4.18}$$

which leads to a generalized method known as *Crout's algorithm*. For each column $j = 1, 2, \ldots, N$, we first solve for the elements of **U** on and above the diagonal, i.e.,

$$U_{ij} = A_{ij} - \sum_{k=1}^{i-1} L_{ik}U_{kj} \qquad i = 1, 2, \ldots, j \,, \tag{4.19}$$

and then for elements of **L** below the diagonal

$$L_{ij} = \frac{1}{U_{jj}}\left[A_{ij} - \sum_{k=1}^{j-1} L_{ik}U_{kj}\right] \qquad i = j+1, j+2, \ldots, N\ . \tag{4.20}$$

Following this algorithm, the elements needed at each step are already computed when they are needed. Note that because **L** is a unit triangular matrix, its diagonal does not need to be stored explicitly. As a result, **A** can be factored *in place*, requiring no additional storage for **L** and **U**. Therefore, throughout the rest of this book we will denote the matrix [**LU**], or often simply the **LU** matrix, as a matrix where **L** and **U** are stored together this way.

A practical LU factorization algorithm will also incorporate partial pivoting to ensure numerical stability at each step of the process. The operation count for the LU factorization and back-substitution are of the same complexity as Gaussian elimination. Once a factorization is completed, it can then be reused for an arbitrary number of right-hand sides. The factored matrix may also be written to disk and read back in later if needed, negating the need to compute and factor the matrix a second time.

4.1.3 Block LU Factorization

The LU factorization described in Section 4.1.2 is a scalar algorithm that operates on individual matrix elements. If implemented as-is, it does not take advantage of CPU vector instructions or efficiently manage the loading of matrix elements to and from system RAM and the CPU cache. If we now partition the matrix **A** into sub-blocks, the scalar operations become block operations, for which optimized kernels can be implemented for various block sizes and CPU architectures.

To illustrate, consider an $N \times N$ matrix **A** partitioned into a 3×3 block matrix, each having approximately $N/3$ rows and columns. This can be written as

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & \mathbf{A}_{13} \\ \mathbf{A}_{21} & \mathbf{A}_{22} & \mathbf{A}_{23} \\ \mathbf{A}_{31} & \mathbf{A}_{32} & \mathbf{A}_{33} \end{bmatrix} = \begin{bmatrix} \mathbf{L}_{11} & 0 & 0 \\ \mathbf{L}_{21} & \mathbf{L}_{22} & 0 \\ \mathbf{L}_{31} & \mathbf{L}_{32} & \mathbf{L}_{33} \end{bmatrix} \begin{bmatrix} \mathbf{U}_{11} & \mathbf{L}_{12} & \mathbf{U}_{13} \\ 0 & \mathbf{U}_{22} & \mathbf{U}_{23} \\ 0 & 0 & \mathbf{U}_{33} \end{bmatrix}, \tag{4.21}$$

where **L** is unit lower triangular, and **U** is upper triangular. As the **L** and **U** blocks are stored together, we write (4.21) using the more compact notation

$$[\mathbf{LU}] = \begin{bmatrix} [\mathbf{LU}]_{11} & \mathbf{U}_{12} & \mathbf{U}_{13} \\ \mathbf{L}_{21} & [\mathbf{LU}]_{22} & \mathbf{U}_{23} \\ \mathbf{L}_{31} & \mathbf{L}_{32} & [\mathbf{LU}]_{33} \end{bmatrix}. \tag{4.22}$$

Let us now compute a scalar LU factorization of the entire first column of **A**, but leave the rest of the matrix unchanged. This results in

$$\begin{bmatrix} [\mathbf{LU}]_{11} & \mathbf{A}_{12} & \mathbf{A}_{13} \\ \mathbf{L}_{21} & \mathbf{A}_{22} & \mathbf{A}_{23} \\ \mathbf{L}_{31} & \mathbf{A}_{32} & \mathbf{A}_{33} \end{bmatrix}. \tag{4.23}$$

We next compute the first trailing sub-row of $\mathbf{U}$, which can be written as

$$\left[\begin{array}{cc}\mathbf{U}_{12} & \mathbf{U}_{13}\end{array}\right] = \mathbf{L}_{11}^{-1}\left[\begin{array}{cc}\mathbf{A}_{12} & \mathbf{A}_{13}\end{array}\right], \tag{4.24}$$

where $\mathbf{L}_{11}^{-1}$ indicates a triangular back-substitution. This results in

$$\left[\begin{array}{ccc}[\mathbf{LU}]_{11} & \mathbf{U}_{12} & \mathbf{U}_{13} \\ \mathbf{L}_{21} & \mathbf{A}_{22} & \mathbf{A}_{23} \\ \mathbf{L}_{31} & \mathbf{A}_{32} & \mathbf{A}_{33}\end{array}\right]. \tag{4.25}$$

We now update the lower-right sub-matrix, yielding

$$\left[\begin{array}{ccc}[\mathbf{LU}]_{11} & \mathbf{U}_{12} & \mathbf{U}_{13} \\ \mathbf{L}_{21} & \tilde{\mathbf{A}}_{22} & \tilde{\mathbf{A}}_{23} \\ \mathbf{L}_{31} & \tilde{\mathbf{A}}_{32} & \tilde{\mathbf{A}}_{33}\end{array}\right], \tag{4.26}$$

where

$$\tilde{\mathbf{A}} = \left[\begin{array}{cc}\tilde{\mathbf{A}}_{22} & \tilde{\mathbf{A}}_{23} \\ \tilde{\mathbf{A}}_{32} & \tilde{\mathbf{A}}_{33}\end{array}\right] = \left[\begin{array}{cc}\mathbf{A}_{22} & \mathbf{A}_{23} \\ \mathbf{A}_{32} & \mathbf{A}_{33}\end{array}\right] - \left[\begin{array}{c}\mathbf{L}_{21} \\ \mathbf{L}_{31}\end{array}\right]\left[\begin{array}{cc}\mathbf{U}_{12} & \mathbf{U}_{13}\end{array}\right]. \tag{4.27}$$

The notation $\tilde{\mathbf{A}}$ indicates that the factorization is not yet complete for that sub-matrix. This process is now repeated for $\tilde{\mathbf{A}}$, where a scalar factorization of the first column yields

$$\left[\begin{array}{cc}[\mathbf{LU}]_{22} & \tilde{\mathbf{A}}_{23} \\ \mathbf{L}_{32} & \tilde{\mathbf{A}}_{33}\end{array}\right]. \tag{4.28}$$

Updating the upper-right block yields

$$\left[\begin{array}{cc}[\mathbf{LU}]_{22} & \mathbf{U}_{23} \\ \mathbf{L}_{32} & \tilde{\mathbf{A}}_{33}\end{array}\right], \tag{4.29}$$

where $\mathbf{U}_{23} = \mathbf{L}_{22}^{-1}\tilde{\mathbf{A}}_{23}$. Updating the trailing sub-matrix $\tilde{\mathbf{A}}_{33}$ yields

$$\left[\begin{array}{cc}[\mathbf{LU}]_{22} & \mathbf{U}_{23} \\ \mathbf{L}_{32} & \tilde{\mathbf{B}}_{33}\end{array}\right] \tag{4.30}$$

where $\tilde{\mathbf{B}}_{33} = \tilde{\mathbf{A}}_{33} - \mathbf{L}_{32}\mathbf{U}_{23}$. As the trailing sub-matrix comprises a single block, we now simply compute $\mathrm{LU}(\tilde{\mathbf{B}}_{33}) = [\mathbf{LU}]_{33}$, yielding finally the $\mathbf{LU}$ matrix in (4.22).

The algorithm just described comprises a right-looking block LU factorization [2, 3]. It is so-called because at each step, the trailing submatrix is located to the lower-right. With the exception of the scalar factorization, all other steps are matrix-matrix operations, which can be carried out efficiently using Level 3 BLAS functions (`gemm` and `trsm`). A similar right-looking algorithm is used in LAPACK (`getrf`), which also incorporates row pivoting. The BLAS and LAPACK software libraries are discussed further in Sections 4.3.1 and 4.3.2.

4.1.4 Condition Number

Given the matrix system $\mathbf{Ax} = \mathbf{b}$, it is desirable to know the impact on the solution $\mathbf{x}$ given any inaccuracies in the estimate of $\mathbf{b}$. If after computing the singular value decomposition (SVD) of $\mathbf{A}$ we observe a large difference between the largest and smallest singular values, we know that small changes in $\mathbf{b}$ may result in large changes in $\mathbf{x}$. We now define the *condition number* of a matrix, which is

$$\kappa(\mathbf{A}) = \|\mathbf{A}^{-1}\| \, \|\mathbf{A}\| \tag{4.31}$$

where $\|\cdot\|$ is a matrix norm. If we choose the 2-norm, then the condition number is given by

$$\kappa(\mathbf{A}) = \frac{\lambda_{max}(\mathbf{A})}{\lambda_{min}(\mathbf{A})} \tag{4.32}$$

where $\lambda_{min}(\mathbf{A})$ and $\lambda_{max}(\mathbf{A})$ are the minimum and maximum eigenvalues of $\mathbf{A}$, respectively [4]. The condition number is important because many elements of the MOM system matrix and right-hand side vectors are obtained through numerical integration. Small errors in the integration can be amplified by a poorly conditioned system matrix. The condition number is also important to iterative methods such as those in the next section, as it has a direct influence on their rate of convergence.

4.2 Iterative Methods

We now discuss iterative solver techniques, which have become increasingly popular in recent years. In these algorithms, the bulk of the compute time is spent computing the matrix-vector products between the system matrix and one or more vectors at each iteration. Though the full matrix must still be stored, the overall compute time is of the order $O(MN^2)$, where N^2 is the operations for the matrix-vector product and M is the number of iterations.

Iterative methods do not modify the original matrix, and instead minimize a residual vector at each iteration. At the core of each lies the update operation

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k \, , \tag{4.33}$$

where the new estimate of the solution $\mathbf{x}_{k+1}$ is obtained from the previous solution $\mathbf{x}_k$ and a search direction $\mathbf{p}_k$, where α_k is a constant. The residual at each step is

$$\mathbf{r}_{k+1} = \mathbf{A}\mathbf{x}_{k+1} - \mathbf{b} \, . \tag{4.34}$$

In this section we will briefly discuss several commonly used iterative methods and summarize templates that the reader can use to write routines of their own. Because the iterative solver algorithms comprise simple operations such as simple vector and matrix-vector products, they are easy and quick to implement in software, even if the user is not familiar with their inner workings. Some of these methods involve matrix-vector products involving the transpose or conjugate transpose of the system matrix, whereas some of the others do not. The latter are referred to as transpose-free.

4.2.1 Conjugate Gradient

The Conjugate Gradient (CG) algorithm is a method for solving a matrix system where **A** is symmetric and positive definite. It is related to the Method of Steepest Descent, and minimizes a quadratic function by generating a sequence of conjugate-direction search vectors that are **A**-orthogonal at each step [5, 6]. Because the system matrix obtained for integral equation problems via the Moment Method is neither symmetric or positive definite, this algorithm cannot be applied directly. Instead, we can apply it to the following system called the "normal equations"

$$\mathbf{A}^\dagger \mathbf{A}\mathbf{x} = \mathbf{A}^\dagger \mathbf{b} \,, \tag{4.35}$$

where $\mathbf{A}^\dagger$ denotes the conjugate transpose, and $\mathbf{A}^\dagger\mathbf{A}$ is symmetric and positive definite. Pseudocode [6] for the preconditioned CG method for the normal equations is provided in Algorithm 1, where **M** is a preconditioner matrix. This method requires two matrix-vector products per iteration, one with **A** and one with $\mathbf{A}^\dagger$.

The CG method has been used extensively for electromagnetic field problems, and the relationship between the matrix eigenvalues and its convergence studied [7, 8, 9, 10]. For a matrix system with N unknowns, CG theoretically converges to the exact solution in at most N iterations assuming there are no round-off errors, and the residual error decreases at each step. For many practical radiation and scattering problems, CG performs very well. However, when the system matrix is poorly conditioned, its convergence rate can be very slow and may stagnate with no additional decrease in the residual norm.

4.2.2 Biconjugate Gradient

The Biconjugate Gradient (BiCG) method was developed by Lanczos [11] and is applicable to general, nonsymmetric systems. BiCG approaches the problem by generating a pair of bi-orthogonal residual sequences using **A** and $\mathbf{A}^T$. Pseudocode [12] for the preconditioned BiCG method is provided in Algorithm 2. This method requires two matrix-vector products per iteration, one with **A** and one with $\mathbf{A}^T$. Though the residual error of the BiCG does not necessarily decrease at each step and its convergence may be erratic, it does exhibit good performance for many problems.

4.2.3 Conjugate Gradient Squared

The Conjugate Gradient Squared (CGS) algorithm [13] is a method that avoids using the transpose of **A** and attempts to obtain a faster rate of convergence than CG and BiCG. The method does converge faster in many cases, though because of its more aggressive formulation it is also more sensitive to residual errors and may diverge quickly if the system is ill-conditioned. Pseudocode [12] for the preconditioned CGS method is provided in Algorithm 3. This method requires two matrix-vector products with **A** per iteration.

4.2.4 Biconjugate Gradient Stabilized

The Biconjugate Gradient Stabilized (BiCG-Stab) algorithm is similar to CGS but attempts to avoid its irregular convergence patterns. Pseudocode [12] for the preconditioned BiCG-Stab method is provided in Algorithm 4. This method requires two matrix-vector products with $\mathbf{A}$ per iteration. Note that there are two end-condition tests in this method.

4.2.5 GMRES

In 1986 Saad and Schultz proposed [14] an algorithm known as the Generalized Minimum Residual (GMRES) method, which can be used for the iterative solution of general, nonsymmetric matrices. In the Conjugate Gradient method, the residuals form an orthogonal basis for the space span $\{\mathbf{r}_0, \mathbf{A}\mathbf{r}_0, \mathbf{A}^2\mathbf{r}_0, \dots\}$. In GMRES, this basis is formed explicitly via the following algorithm:

$\mathbf{w}_i = \mathbf{A}\mathbf{v}_i$
for $k = 1, 2, \dots, i$ **do**
  $\mathbf{w}_i = \mathbf{w}_i - (\mathbf{w}_i \cdot \mathbf{v}_k)\mathbf{v}_k$
end for
$\mathbf{v}_{i+1} = \mathbf{w}_i / ||\mathbf{w}_i||$

which is a modified Gram-Schmidt orthogonalization. Applied to the Krylov sequence $\{\mathbf{A}^k\mathbf{r}_0\}$, this is called the Arnoldi method. The inner product coefficients $\mathbf{w}_i \cdot \mathbf{v}_k$ and $||\mathbf{w}_i||$ are stored in an upper Hessenberg matrix. The GMRES iterates are then constructed as

$$\mathbf{x}_i = \mathbf{x}_0 + y_1\mathbf{v}_1 + y_2\mathbf{v}_2 + \cdots y_i\mathbf{v}_i \,, \tag{4.36}$$

where the coefficients y_k are chosen to minimize the residual norm $||\mathbf{b} - \mathbf{A}\mathbf{x}_i||$. GMRES has the property that this residual norm can be computed without the iterate having been formed. Thus, the expensive action of forming the iterate can be postponed until the residual norm has grown small enough. The drawback to GMRES is that the work and storage required per iteration increase linearly with the iteration count. Unless convergence happens quickly, the cost can quickly become prohibitive. One common way of handling this is a reset of the iteration: after m iterations, the accumulated data are cleared and the intermediate results used as the initial data for the next m iterations. This procedure is then repeated until convergence is achieved. The difficulty now lies in choosing an appropriate value for m. If m is too small, the algorithm may be slow to converge, or fail to converge entirely. If m is too large, the required computations and storage may be excessive.

Pseudocode [12] for the preconditioned, restarted GMRES(m) method is provided in Algorithm 5. GMRES is employed often to solve electromagnetic problems due to its good convergence properties, robust performance, and the fact it is transpose-free. We will use it along with the Fast Multipole Method in Chapter 11.

Algorithm 1 Preconditioned Conjugate Gradient (CGNR) Method

Make an initial guess for $\mathbf{x}_0$
$\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$
$\tilde{\mathbf{r}}_0 = \mathbf{A}^\dagger \mathbf{r}_0$
Solve $\mathbf{M}\mathbf{z}_0 = \tilde{\mathbf{r}}_0$
$\mathbf{p}_0 = \mathbf{z}_0$
for $i = 1, 2, ...,$ until convergence **do**
  $\mathbf{w}_{i-1} = \mathbf{A}\mathbf{p}_{i-1}$
  $\alpha_{i-1} = (\mathbf{z}^*_{i-1} \cdot \tilde{\mathbf{r}}_{i-1})/||\mathbf{w}_{i-1}||_2^2$
  $\mathbf{x}_i = \mathbf{x}_{i-1} + \alpha_{i-1}\mathbf{p}_{i-1}$
  $\mathbf{r}_i = \mathbf{r}_{i-1} - \alpha_{i-1}\mathbf{w}_{i-1}$
  Convergence check
  $\tilde{\mathbf{r}}_i = \mathbf{A}^\dagger \mathbf{r}_i$
  Solve $\mathbf{M}\mathbf{z}_i = \tilde{\mathbf{r}}_i$
  $\beta_{i-1} = (\mathbf{z}^*_i \cdot \tilde{\mathbf{r}}_i)/(\mathbf{z}^*_{i-1} \cdot \tilde{\mathbf{r}}_{i-1})$
  $\mathbf{p}_i = \mathbf{z}_i + \beta_{i-1}\mathbf{p}_{i-1}$
end for

Algorithm 2 Preconditioned Biconjugate Gradient (BiCG) Method

Make an initial guess for $\mathbf{x}_0$
$\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$
$\tilde{\mathbf{r}}_0 = \mathbf{r}_0$
for $i = 1, 2, ...,$ until convergence **do**
  Solve $\mathbf{M}\mathbf{z}_{i-1} = \mathbf{r}_{i-1}$
  Solve $\mathbf{M}^T\tilde{\mathbf{z}}_{i-1} = \tilde{\mathbf{r}}_{i-1}$
  $\rho_{i-1} = \mathbf{z}_{i-1} \cdot \tilde{\mathbf{r}}_{i-1}$
  if $\rho_{i-1} = 0$ **then**
    method fails
  end if
  if $i = 1$ **then**
    $\mathbf{p}_i = \mathbf{z}_{i-1}$
    $\tilde{\mathbf{p}}_i = \tilde{\mathbf{z}}_{i-1}$
  else
    $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
    $\mathbf{p}_i = \mathbf{z}_{i-1} + \beta_{i-1}\mathbf{p}_{i-1}$
    $\tilde{\mathbf{p}}_i = \tilde{\mathbf{z}}_{i-1} + \beta_{i-1}\tilde{\mathbf{p}}_{i-1}$
  end if
  $\mathbf{q}_i = \mathbf{A}\mathbf{p}_i$
  $\tilde{\mathbf{q}}_i = \mathbf{A}^T\tilde{\mathbf{p}}_i$
  $\alpha_i = \rho_{i-1}/(\tilde{\mathbf{p}}_i \cdot \mathbf{q}_i)$
  $\mathbf{x}_i = \mathbf{x}_{i-1} + \alpha_i\mathbf{p}_i$
  $\mathbf{r}_i = \mathbf{r}_{i-1} - \alpha_i\mathbf{q}_i$
  $\tilde{\mathbf{r}}_i = \tilde{\mathbf{r}}_{i-1} - \alpha_i\tilde{\mathbf{q}}_i$
  Convergence check
end for

Algorithm 3 Preconditioned Conjugate Gradient Squared (CGS) Method

Make an initial guess for $\mathbf{x}_0$
$\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$
$\tilde{\mathbf{r}} = \mathbf{r}_0$
for $i = 1, 2, ...,$ until convergence **do**
  $\rho_{i-1} = \tilde{\mathbf{r}} \cdot \mathbf{r}_{i-1}$
  if $\rho_{i-1} = 0$ **then**
    method fails
  end if
  if $i = 1$ **then**
    $\mathbf{u}_1 = \mathbf{r}_0$
    $\mathbf{p}_1 = \mathbf{u}_1$
  else
    $\beta_{i-i} = \rho_{i-1}/\rho_{i-2}$
    $\mathbf{u}_i = \mathbf{r}_{i-1} + \beta_{i-1}\mathbf{q}_{i-1}$
    $\mathbf{p}_i = \mathbf{u}_i + \beta_{i-1}(\mathbf{q}_{i-1} + \beta_{i-1}\mathbf{p}_{i-1})$
  end if
  Solve $\mathbf{M}\hat{\mathbf{p}} = \mathbf{p}_i$
  $\hat{\mathbf{v}} = \mathbf{A}\hat{\mathbf{p}}$
  $\alpha_i = \rho_{i-1}/(\tilde{\mathbf{r}} \cdot \hat{\mathbf{v}})$
  $\mathbf{q}_i = \mathbf{u}_i - \alpha_i\hat{\mathbf{v}}$
  Solve $\mathbf{M}\hat{\mathbf{u}} = \mathbf{u}_i + \mathbf{q}_i$
  $\mathbf{x}_i = \mathbf{x}_{i-1} + \alpha_i\hat{\mathbf{u}}$
  $\hat{\mathbf{q}} = \mathbf{A}\hat{\mathbf{u}}$
  $\mathbf{r}_i = \mathbf{r}_{i-1} - \alpha_i\hat{\mathbf{q}}$
  Convergence check
end for

Algorithm 4 Preconditioned Biconjugate Gradient Stabilized (BiCG-Stab) Method

Make an initial guess for $\mathbf{x}_0$
$\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$
$\tilde{\mathbf{r}} = \mathbf{r}_0$
for $i = 1, 2, ...,$ until convergence **do**
  $\rho_{i-1} = \tilde{\mathbf{r}} \cdot \mathbf{r}_{i-1}$
  if $\rho_{i-1} = 0$ **then**
    method fails
  end if
  if $i = 1$ **then**
    $\mathbf{p}_1 = \mathbf{r}_0$
  else
    $\beta_{i-1} = (\rho_{i-1}/\rho_{i-2})(\alpha_{i-1}/\omega_{i-1})$
    $\mathbf{p}_i = \mathbf{r}_{i-1} + \beta_{i-1}(\mathbf{p}_{i-1} - \omega_{i-1}\mathbf{v}_{i-1})$
  end if
  Solve $\mathbf{M}\hat{\mathbf{p}} = \mathbf{p}_i$
  $\mathbf{v}_i = \mathbf{A}\hat{\mathbf{p}}$
  $\alpha_i = \rho_{i-1}/(\tilde{\mathbf{r}} \cdot \mathbf{v}_i)$
  $\mathbf{s} = \mathbf{r}_{i-1} - \alpha_i \mathbf{v}_i$
  Check norm of **s**, if small enough set $\mathbf{x}_i = \mathbf{x}_{i-1} + \alpha_i \hat{\mathbf{p}}$, stop
  Solve $\mathbf{M}\hat{\mathbf{s}} = \mathbf{s}$
  $\mathbf{t} = \mathbf{A}\hat{\mathbf{s}}$
  $\omega_i = (\mathbf{t} \cdot \mathbf{s})/(\mathbf{t} \cdot \mathbf{t})$
  $\mathbf{x}_i = \mathbf{x}_{i-1} + \alpha_i \hat{\mathbf{p}} + \omega_i \hat{\mathbf{s}}$
  $\mathbf{r}_i = \mathbf{s} - \omega_i \mathbf{t}$
  Convergence check
  For continuation, it is necessary that $\omega_i \neq 0$
end for

Algorithm 5 Preconditioned GMRES(m) Method

Make an initial guess for $\mathbf{x}_0$
for $j = 1, 2, ...,$ **do**
  Solve $\mathbf{Mr} = \mathbf{b} - \mathbf{Ax}_0$
  $\mathbf{v}_1 = \mathbf{r}/||\mathbf{r}||_2$
  $\mathbf{s} := ||\mathbf{r}||_2\mathbf{e}_1$
  for $i = 1, ..., m$ **do**
    Solve $\mathbf{Mw} = \mathbf{Av}_i$
    for $k = 1, ..., i$ **do**
      $h_{k,i} = \mathbf{w} \cdot \mathbf{v}_k$
      $\mathbf{w} = \mathbf{w} - h_{k,i}\mathbf{v}_k$
    end for
    $h_{i+1,i} = ||\mathbf{w}||_2$
    $\mathbf{v}_{i+1} = \mathbf{w}/h_{i+1,i}$
    Apply $J_1, \ldots, J_{i-1}$ on $(h_{1,i}, \ldots, h_{i+1,i})$
    Construct J_i acting on the ith and $(i+1)$st component
    of $h_{\cdot,i}$ such that $(i+1)$st component of $J_i h_{\cdot,i}$ is 0
    $\mathbf{s} := J_i\mathbf{s}$
    if $\mathbf{s}(i+1)$ small enough **then**
      UPDATE($\tilde{\mathbf{x}}, i$)
      stop
    end if
  end for
  UPDATE($\tilde{\mathbf{x}}, m$)
end for

procedure UPDATE($\tilde{\mathbf{x}}, i$)
  Solve $\mathbf{Hy} = \tilde{\mathbf{s}}$
  Where the upper $i \times i$ triangular part of $\mathbf{H}$ has $h_{i,j}$ as
  its elements, and $\tilde{\mathbf{s}}$ represents the first i components of $\mathbf{s}$.
  $\tilde{\mathbf{x}} = \mathbf{x}_0 + y_1\mathbf{v}_1 + y_2\mathbf{v}_2 + \cdots y_i\mathbf{v}_i$
  $\mathbf{s}(i+1) = ||\mathbf{b} - \mathbf{A}\tilde{\mathbf{x}}||_2$
  if $\tilde{\mathbf{x}}$ is accurate enough **then**
    stop
  else
    $\mathbf{x}_0 = \tilde{\mathbf{x}}$
  end if
end procedure

4.2.6 Stopping Criteria

When using an iterative method, we need to know when to terminate the iteration. Since we do not know the solution vector $\mathbf{x}$, we cannot compute the error vector

$$\mathbf{e}_i = \mathbf{x} - \mathbf{x}_i \tag{4.37}$$

but instead compute the residual norm N_i at iteration i, which is

$$N_i = \frac{\|\mathbf{r}_i\|}{\|\mathbf{b}\|} = \frac{\|\mathbf{A}\mathbf{x}_i - \mathbf{b}\|}{\|\mathbf{b}\|} \,. \tag{4.38}$$

We terminate the iteration once the residual norm is below a certain value, such as 10^{-3}. Note that the residual norm is only an indirect measure of the amount of error, because [4]

$$\frac{\|\mathbf{e}_n\|}{\|\mathbf{e}_0\|} \leq \kappa(\mathbf{A}) \frac{\|\mathbf{r}_n\|}{\|\mathbf{r}_0\|} \,, \tag{4.39}$$

and if $\mathbf{A}$ is badly conditioned, $\mathbf{x}_i$ may not be a very good estimate of the solution for a given N_i. In practice, the iteration is stopped when the residual norm goes below a predefined value, or a maximum number of iterations is reached.

4.2.7 Preconditioning

Previous sections have pointed out the relationship between increasing condition number of a matrix and slow convergence of iterative solvers. In some cases, the solution may converge but only after many iterations and a long compute time. In others, the solution may be unobtainable because the iteration stagnates or even diverges. In both cases, though especially in the latter, a method of improving the convergence behavior of the solver is desirable. To investigate such a method, let us modify the original linear system to create the new system

$$\mathbf{M}^{-1}\mathbf{A}\mathbf{x} = \mathbf{M}^{-1}\mathbf{b} \,, \tag{4.40}$$

where $\mathbf{M}$ is a *preconditioner matrix*. If $\mathbf{M}$ resembles $\mathbf{A}$ in some way, then the solution of the above matrix system remains the same, however the eigenvalues of $\mathbf{M}^{-1}\mathbf{A}$ may be more attractive and allow better performance in the iterative solver. Algorithms 1-5 include a preconditioning step, requiring the solution of one or more auxiliary linear systems of the form $\mathbf{M}\mathbf{z} = \mathbf{r}$ at each iteration.

The problem we now face is finding the value of $\mathbf{M}^{-1}$. Obviously, if $\mathbf{M}^{-1}$ is exactly equal to $\mathbf{A}^{-1}$ we have not improved anything, as this requires a complete factorization of $\mathbf{A}$. The key is determining a value of $\mathbf{M}^{-1}$ that has a reasonable setup time and memory demand, and that does not impose an unacceptable overhead when solving the auxiliary problem at each iteration. In some cases, the application of a preconditioner is absolutely necessary as the original system does not converge at all. In other cases, the hope is that the improved rate of convergence will result in fewer iterations and an overall savings in compute time. This is particularly attractive when the calculations involve multiple right-hand sides, as the preconditioner setup time can be amortized quickly. Several effective preconditioning methods are discussed at length in [6] for general matrices. We present versions of some of these in Section 11.5, adapted specifically for use with the Fast Multipole Method.

4.3 Software for Linear Systems

There are many off-the-shelf software libraries available for working with and solving linear systems of equations. Fortunately, many of these are well written, time-tested, robust and freely available. This means that the programmer does not have to worry about writing their own routines, and can instead simply install the libraries and follow the provided API. In this section we will briefly discuss some of the software most widely used for matrix algebra problems.

4.3.1 BLAS

The BLAS (Basic Linear Algebra Subprograms) comprises a set of subroutines that act as building blocks for more complex vector and matrix operations [15]. BLAS comprises three levels of functions: BLAS level 1 for scalar, vector and vector-vector operations, BLAS level 2 for matrix-vector operations, and BLAS level 3 for matrix-matrix operations. Separate routines exist for real and complex arithmetic, and in single and double precision. The BLAS source code is freely available from the Netlib BLAS repository[1], and may be compiled and included in commercial applications at no cost. This version comprises the original FORTRAN "reference implementation", which is the standard from which other versions are derived. There are several highly optimized versions of BLAS available from various vendors. For Intel CPUs, the most popular are the Intel oneAPI Math Kernel Library (oneMKL) [16] and OpenBLAS [17], both of which are distributed for free as of 2026. These libraries are written to take advantage of CPU vector instructions, and carefully manage the use of the CPU cache and the slower system RAM. These libraries also possess their own internal thread-level parallelism, and can take advantage of multiple processors without the user having to implement a multithreaded program themselves.

There also exist BLAS libraries for systems having one or more graphics processing units (GPUs). These units are massively parallel, having thousands of small processing cores, and are capable of orders of magnitude higher performance in BLAS operations than on a single CPU. For users with NVIDIA GPUs, the cuBLAS library [18] is a GPU-accelerated implementation of BLAS made specifically for NVIDIA GPUs, and uses the NVIDIA CUDA API programming model [19]. Another GPU-accelerated implementation of BLAS is MAGMA [20], which provides implementations for CUDA as well as OpenCL, which is an open, parallel programming standard supported by AMD GPUs.

4.3.2 LAPACK

LAPACK is a library for solving dense linear systems, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems [21]. It is also available from the Netlib LAPACK repository[2], and may be used freely in commercial applications. LAPACK routines use BLAS functions for low-level work, and a BLAS implementation must accompany an installation of LAPACK. Separate versions

[1] netlib.org/blas
[2] netlib.org/lapack

of most LAPACK routines are available for real and complex arithmetic, and in single and double precision. As with BLAS, there are also highly optimized versions of LAPACK available. The Intel oneAPI library also contains an implementation of LAPACK for CPUs. Similarly, MAGMA also implements LAPACK functions for GPUs.

4.3.3 MATLAB

MATLAB is a popular numerical computing environment and programming language commercially available from The Mathworks, Inc. It was originally developed to aid in solving matrix problems and many of its matrix functions operate via calls to BLAS and LAPACK. The MATLAB scripting language is easy to learn and fairly complex simulations can be written in it. Though its scripts are interpreted and less efficient than compiled code, MATLAB is easy to learn, accurate, and can be used to solve many of the examples in this book. It is also excellent at generating plots, and was used to generate all the plots in this book.

References

[1] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C : The Art of Scientific Computing*. Cambridge University Press, 1992.

[2] J. W. Demmel, N. J. Higham, and R. S. Schreiber, "Stability of block LU factorization," *Numer. Lin. Algebra Applic*, vol. 2, pp. 173–190, 1995.

[3] J. Dongarra, S. Hammarlin, and D. Walker, "Key concepts for parallel out-of-core LU factorization," *Computers and Mathematics With Applications*, vol. 35, pp. 13–31, April 1998.

[4] A. F. Peterson, S. L. Ray, and R. Mittra, *Computational Methods for Electromagnetics*. IEEE Press, 1998.

[5] M. Hestenes and E. Steifel, "Methods of conjugate gradients for solving linear systems," *J. Res. Nat. Bur. Stand.*, vol. 49, pp. 409–435, 1952.

[6] Y. Saad, *Iterative Methods for Sparse Linear Systems*. PWS, 1st ed., 1996.

[7] T. K. Sarkar, K. R. Siarkiewicz, and R. F. Stratton, "Survey of numerical methods for solution of large systems of linear equations for electromagnetic field problems," *IEEE Trans. Antennas Propagat.*, vol. 29, pp. 847–856, November 1981.

[8] T. K. Sarkar, "The conjugate gradient method as applied to electromagnetic field problems," *IEEE Antennas Propagat. Soc. Newsletter*, pp. 5–14, August 1986.

[9] A. F. Peterson and R. Mittra, "Convergence of the conjugate gradient method when applied to matrix equations representing electromagnetic scattering problems," *IEEE Trans. Antennas Propagat.*, vol. 34, pp. 1447–1454, December 1986.

[10] A. F. Peterson, C. F. Smith, and R. Mittra, "Eigenvalues of the moment-method matrix and their effect on the convergence of the conjugate gradient algorithm," *IEEE Trans. Antennas Propagat.*, vol. 36, pp. 1177–1179, August 1988.

[11] C. Lanczos, "Solution of systems of linear equations by minimized iterations," *J. Res. Nat. Bur. Standards*, vol. 49, pp. 33–53, 1952.

[12] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, second ed., 1994.

[13] P. Sonneveld, "CGS, a fast Lanczos-type solver for nonsymmetric linear systems," *SIAM J. Sci. Statist. Comput.*, vol. 10, pp. 36–52, 1989.

[14] Y. Saad and M. H. Schultz, "GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM J. Sci. Stat. Comput.*, vol. 7, pp. 856–869, July 1986.

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

[16] Intel oneAPI Math Kernel Library (oneMKL), www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html.

[17] OpenBLAS, An optimized BLAS library, www.openblas.net.

[18] cuBLAS, docs.nvidia.com/cuda/cublas/index.html.

[19] CUDA Toolkit Documentation, docs.nvidia.com/cuda.

[20] J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, and I. Yamazaki, "Accelerating numerical dense linear algebra calculations with gpus," *Numerical Computations with GPUs*, pp. 1–26, 2014.

[21] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics, third ed., 1999.