Chapter 3

Methods for the Solution of Linear Systems Deriving from Elliptic PDEs

A finite volume discretisation of the elliptic PDEs (Partial Differential Equations) used to describe a fluid mechanics problem generates a large sparse set of linear equations. Typically a CFD algorithm will involve the repetitive solution of a Poisson pressure equation along with scalar transport equations for momentum, enthalpy, concentration, and any other fields of interest. Normally the program will spend most of its execution time in solving these linearised equations, and so the efficiency of the linear solvers underpins the efficiency of the solution method as a whole. Therefore a crucial aspect of any efficient solution of a fluid mechanics problem is the speed that these linear(ised) equations can be solved.

In this chapter a number of different algorithms for the solution of linear equations are discussed and their resource use is compared, both in terms of speed and memory use. Whilst there are many papers comparing two or three linear solvers, comparisons of several classes of linear solver are rare in the literature. Ferziger and Perić’s book [43] compares a number of methods, but the comparisons are made in terms of number of iterations to converge instead of time to converge, a rather meaningless measure due to the variation in computation effort per iteration amongst the solvers. Botta et al [14] compare a number of methods, however the solvers were written by different groups and so there is the possibility that some of the variation in performance is due to different coding standards. In both cases the number of methods covered is less than in the current study.

This chapter is divided into two parts; the first describes the linear solvers, whilst the second discusses their suitability for the solution of the equations that result from a finite volume discretisation of elliptic PDEs.

3.1 A Description of the Linear Solvers

Methods for solving linear equations can be divided into two classes, direct methods (i.e., those which execute in a predetermined number of operations) and iterative methods (i.e., those which attempt to converge to the desired answer in an unknown number of repeated steps). Direct methods are often used for small dense problems, but for the large sparse problems which are typically encountered in the solution of PDEs, iterative methods are usually more efficient.

The direct methods which are discussed here are Gauss Jordan elimination, LU (Lower Upper) factorisation, Cholesky factorisation, LDL (Lower Diagonal Lower\(^T\)) decomposition, Tridiagonal and Block Tridiagonal methods.
Four classes of iterative solver are also discussed:

- simple iterative methods, such as Jacobi iteration, SOR (Successive Over Relaxation), Red-Black SOR, and SSOR (Symmetric Successive Over Relaxation).
- incomplete factorisation schemes, such as Incomplete Cholesky (IC), ILU (Incomplete LU factorisation), SIP (Strongly Implicit Procedure, also known as Stone’s method), and MSI (Modified Strongly Implicit procedure).
- Krylov space methods, such as the CG (Conjugate Gradient), CGS (Conjugate Gradient Squared), GMRES (General Minimalised Residual), QMR (Quasi-Minimalised Residual), BiCG (Bi-Conjugate Gradient) and BiCGSTAB (Bi-Conjugate Gradient Stabilised) methods.
- multigrid schemes.

The distinction between the classes of iterative solvers blurs somewhat since multigrid methods can be considered as an acceleration technique to improve the performance of the simple iterative and incomplete factorisation methods, and Krylov-space methods can use the simple iterative and incomplete factorisation methods as preconditioners. A Krylov space method using an incomplete factorisation smoothed multigrid preconditioner is a solver that combines three of the classes of iterative solver.

### 3.1.1 Linear Equations Resulting from a Finite Volume Discretisation of a PDE

The general form of a set of linear equations can be written,

$$ \mathbf{A} \mathbf{x} = \mathbf{b}, $$

or considering the individual $i^{th}$ equation

$$ \sum_{j=1}^{N} a_{i,j} x_j = b_i. $$

For a set of equations resulting from a finite volume discretisation of a three dimensional PDE on a structured mesh, the coefficient matrix $\mathbf{A}$ will typically take on a hepta-diagonal structure, with the non-zero components occupying only seven diagonals of the matrix. For a two dimensional PDE there will be only five diagonals which are non zero, and for a one dimensional PDE there are three non-zero diagonals. This regular structure enables a considerable reduction in memory use and the number of operations performed since only these seven diagonals need to be stored and operated upon. Using the compass notation of the equations discussed previously in Section 2.1 the above linear equation becomes

$$ a_P x_P + a_E x_E + a_W x_W = b_i $$

for a discretisation of a one dimensional PDE,

$$ a_P x_P + a_E x_E + a_W x_W + a_N x_N + a_S x_S = b_i $$

for a discretisation of a two dimensional PDE, and

$$ a_P x_P + a_E x_E + a_W x_W + a_N x_N + a_S x_S + a_T x_T + a_B x_B = b_i $$

for a three dimensional PDE. The subscript $P$ refers to the point at which the equation is centred, and the $E, W, N, S, T$ and $B$ subscripts refer to the neighbouring East, West, North, South, Top and
Bottom points respectively (see Figure 2.2). As a notational simplification equations (3.3) to (3.5) can be rewritten as

\[ a_P x_P + \sum_{n_b} a_{nb} x_{nb} = b, \]  

(3.6)

where nb represents the neighbouring points E, W, N, S, T and B.

For some systems the equations are symmetric,

\[ A = A^T. \]  

(3.7)

This is commonly the case in diffusion problems, or in the solution of a pressure correction equation. Some of the linear solvers can take advantage of this symmetry at the cost of being only applicable to symmetric systems. In the more general case the equations are non-symmetric, which is the case with a transport equation with advection.

### 3.1.2 Direct Methods

Direct solvers are not commonly used in solving finite volume equations for the very good reason that they scale poorly with problem size, both in terms of memory and number of operations. A notable exception is the Thomas tridiagonal algorithm which is unfortunately only applicable to one-dimensional PDEs. However a multidimensional block tridiagonal method can be developed.

#### LU (Lower Upper) Factorisation

For solving dense systems of equations LU factorisation is commonly used. It has a similar operation count to Gaussian elimination but allows the efficient solution of many right hand sides. The following description of such a solver follows that of Press et al[133]. A more thorough description of the method can be found in Golub and Van Loan[51].

A matrix \( A \) can be factored into lower and upper triangular components such that,

\[ LU = A. \]  

(3.8)

This decomposition can be used to solve the equation,

\[ A x = (LU) x = L (U x) = b, \]  

(3.9)

by first solving for the vector \( y \) such that

\[ Ly = b, \]  

(3.10)

and then solving

\[ U x = y. \]  

(3.11)

The solution of the upper and lower triangular set of equations is trivial, since equation (3.10) can be solved by forward substitution,

\[ y_1 = \frac{b_1}{l_{1,1}}, \]

\[ y_i = \frac{1}{l_{i,i}} \left[ b_i - \sum_{j=1}^{i-1} l_{i,j} y_j \right], \quad i = 2, 3, \ldots N, \]  

(3.12)
where $N$ is the number of equations, followed by the solution of equation (3.11) by backward substitution,

$$x_N = \frac{y_N}{u_{N,N}},$$

$$x_i = \frac{1}{u_{i,i}} \left[ y_i - \sum_{j=i+1}^{N} u_{i,j} x_j \right], \quad i = N-1, N-2, \ldots, 1. \quad (3.13)$$

The only problem remaining is factoring $A$ into $LU$, a process that can be performed using Crout’s algorithm. For each column of the matrix, $j = 1, 2, \ldots, N$,

$$u_{1,j} = a_{1,j},$$

$$u_{i,j} = a_{i,j} - \sum_{k=1}^{i-1} u_{i,k} a_{k,j}, \quad i = 2, 3, \ldots, i-1,$$

$$u_{j,j} = 1, \quad (3.14)$$

$$l_{j,j} = 1 / \left( a_{i,j} - \sum_{k=1}^{j-1} u_{i,k} l_{k,j} \right),$$

$$l_{i,j} = l_{j,j} \left( a_{i,j} - \sum_{k=1}^{j-1} u_{i,k} l_{k,j} \right), \quad i = j+1, j+2, \ldots, N.$$

The algorithm for factoring the $A$ array into the $LU$ arrays is given in Figure 3.1, with the solution of the system $(LU)x = b$ being given in Figure 3.2.

For $j = 1, 2, \ldots, N$

- $u_{1,j} = a_{1,j}$
- for $i = 2, 3, \ldots, j-1$
  - $u_{i,j} = a_{i,j} - \sum_{k=1}^{i-1} u_{i,k} a_{k,j}$
  - $u_{j,j} = 1$
- $l_{j,j} = 1 / \left( a_{i,j} - \sum_{k=1}^{j-1} u_{i,k} l_{k,j} \right)$
- for $i = j+1, j+2, \ldots, N$
  - $l_{i,j} = l_{j,j} \left( a_{i,j} - \sum_{k=1}^{j-1} u_{i,k} l_{k,j} \right)$

Figure 3.1: Factoring $A$ into $LU$.

For $i = 2, 3, \ldots, N$

- $y_i = b_i / l_{i,1}$
- $y_i = \left( b_i - \sum_{j=1}^{i-1} l_{i,j} y_j \right) / l_{i,i}$

For $i = N-1, N-2, \ldots, 1$

- $x_N = y_N / u_{N,N}$
- $x_i = \left( y_i - \sum_{j=i+1}^{N} u_{i,j} x_j \right) / u_{i,i}$

Figure 3.2: Solving the system $(LU)x = b$.

For a system of $N$ equations (which would correspond to the $N$ points on a finite volume mesh) the storage requirement for an LU factorisation is $N^2 + N$, whilst the number of operations is of $O(N^3)$.
for factorisation, and $O(N^2)$ for solution. For a sparse system of equations such a scheme is rather inefficient since $N$ is likely to be large, and most of $A$ is zero. A more efficient band diagonal version where the array is stored as a band only wide enough to store the farthest off-band diagonal is implemented in Press et al[133].

For symmetric matrices where $A = A^T$ a more efficient factorisation which takes advantage of the symmetry is the Cholesky factorisation $A = LL^T$ (sometimes referred to as the square root of the matrix), and the LDL (Lower Diagonal Lower$^T$) factorisation $A = LDL^T[51]$. Both give a threefold reduction in the number of operations compared to LU factorisation, but the LDL method is to be preferred since the Cholesky factorisation requires $N$ square root operations, which can be a slow operation on many computers.

For a general system of equations pivoting, where the equations of the system are reordered, is necessary to ensure numerical stability. However, for the equations resulting from a discretised elliptic PDE, in particular the equations resulting from the pressure equation given in Equation (4.23), the system is diagonally dominant and no pivoting is necessary[51, 133].

**Thomas Tridiagonal Solver**

A finite volume discretisation of a one dimensional PDE results in a system of equations with only the diagonal and one sub- and super-diagonal having non-zero values, commonly referred to as a tridiagonal system of equations. These can be solved extremely easily and efficiently using the Thomas algorithm[169].

A tridiagonal system of equations $A \mathbf{x} = \mathbf{b}$ can be written as,

$$
\begin{bmatrix}
  d_1 & c_1 &  &  &  \\
  a_2 & d_2 & c_2 &  &  \\
    & \ddots & \ddots & \ddots &  \\
  a_{N-1} & d_{N-1} & c_{N-1} & a_N & d_N
\end{bmatrix}
\begin{bmatrix}
  x_1 \\
  x_2 \\
  \vdots \\
  x_{N-1} \\
  x_N
\end{bmatrix}
=
\begin{bmatrix}
  b_1 \\
  b_2 \\
  \vdots \\
  b_{N-1} \\
  b_N
\end{bmatrix}
$$

For the finite volume discretisation of a PDE given in (3.3) the vectors $\mathbf{a}$, $\mathbf{d}$ and $\mathbf{c}$ correspond to $a_W$, $a_P$ and $a_E$ respectively.

To solve such a set of equations requires only a forward and a backward substitution, the algorithm is given in Figure 3.3.

![Figure 3.3: The Thomas Tridiagonal Algorithm.](image-url)

The Thomas tridiagonal solver requires $N$ words of memory and takes $O(N)$ operations, and so is about as efficient a solver as can be hoped for. Unfortunately it can only be implemented for one-dimensional PDEs. A one-dimensional PDE with cyclic boundary conditions is not truly tridiagonal,
and cannot be solved with the generic Thomas Tridiagonal algorithm. However an implementation for cyclic boundary conditions is given in Press et al[133].

Block Tridiagonal Solvers

A finite volume discretisation of a multidimensional PDE results in equations that do not have the tridiagonal structure that allows the use of the efficient Thomas algorithm. However, they do have a tridiagonal block structure that allows a similar scheme, with dense submatrices being used instead of scalars.

For a finite volume discretisation of a two dimensional PDE as described by equation (3.4), the matrix structure can be viewed as,

$$
\begin{bmatrix}
D_1 & C_1 & & \\
A_2 & D_2 & C_2 & \\
& \ddots & \ddots & \\
A_{N-1} & D_{N-1} & C_{N-1} & A_N & D_N
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
\vdots \\
x_{N-1} \\
x_N
\end{bmatrix}
= 
\begin{bmatrix}
b_1 \\
b_2 \\
\vdots \\
b_{N-1} \\
b_N
\end{bmatrix},
$$

(3.16)

where the submatrices $A_i$, $D_i$, $C_i$ and $b_i$ have the structure,

$$A_i = \begin{bmatrix}
a_{S_1,i} & & \\
& a_{S_2,i} & \\
& & \ddots \\
& & & a_{S_n,i}
\end{bmatrix},$$

(3.17)

$$D_i = \begin{bmatrix}
a_{P_1,i} & a_{E_1,i} & & \\
a_{W_2,i} & a_{P_2,i} & a_{E_2,i} & \\
& \ddots & \ddots & \ddots \\
a_{W_n,i} & a_{P_n,i} & \cdots & a_{E_n,i}
\end{bmatrix},$$

(3.18)

$$C_i = \begin{bmatrix}
a_{N_1,i} & & \\
& a_{N_2,i} & \\
& & \ddots \\
& & & a_{N_n,i}
\end{bmatrix},$$

(3.19)

$$b_i = \begin{bmatrix}
b_{1,i} \\
b_{2,i} \\
\vdots \\
b_{n,i}
\end{bmatrix},$$

(3.20)

In a manner similar to the Thomas Tridiagonal algorithm these equations can be solved by a forward and a backwards block substitution. A similar block structure can be used to solve three dimensional PDEs.

The matrix inverse $Z^{-1}$ used in Figure 3.4, which contains the block tridiagonal algorithm, is purely for notational purposes. Instead of calculating the inverse and performing a matrix multiplication as in $X = B A^{-1}$, the system can be solved as $AX = B$ by factoring $A$ into its $L$ and $U$ components, and then solving $(LU)x = b$ for each column of $X$ and $B$. Unfortunately the sub-matrices $\Gamma$ and $Z$ are dense and so cannot be solved by a tridiagonal solver but instead must be solved with a dense solver such as LU decomposition. For symmetric systems a Cholesky or LDL decomposition can be used instead of LU factorisation, reducing the number of operations.
\[ Z = D_1 \]
\[ y_1 = Z^{-1} b_1 \]
for \( i = 2, 3 \ldots N \)
\[ \Gamma_i = Z^{-1} C_{i-1} \]
\[ Z = D_i - A_i \Gamma_i \]
\[ y_i = (b_i - A_i y_{i-1}) Z^{-1} \]
\[ x_N = y_N \]
for \( i = N-1, N-2 \ldots 1 \)
\[ x_i = y_i - \Gamma_{i+1} x_{i+1} \]

Figure 3.4: The Block Tridiagonal Algorithm.

The two dimensional version of the solver uses \( N^{3/2} \) words of storage and takes \( O(N^2) \) operations (assuming a square mesh), whilst the three dimensional version uses \( N^{5/2} \) words of storage and \( O(N^{7/3}) \) operations. Whilst this is better than LU factorisation, it is not as good as the one dimensional tridiagonal solver, and is less efficient than most iterative schemes. However for cases where one dimension of the problem is much greater than the others the dense submatrices can be made smaller and the efficiency of the method (both in terms of storage and number of operations) can be greatly improved\(^1\).

### 3.1.3 Iterative Methods

An iterative linear solver works by repeatedly apply a series of operations to an approximate solution to the linear system, with the error in the approximate solution being reduced by each application of the operations. Semantically, each application of the operations is an iteration, and the schemes iterate towards a solution. The basic form of all iterative methods is given in Figure 3.5.

\[
\text{set } x^0 \text{ to an initial estimate of } x \\
k = 1 \\
\text{while } k < k_{\text{max}} \text{ and error } > \text{tolerance} \\
\text{perform some operations } x^k = f(x^{k-1}) \\
k = k + 1
\]

Figure 3.5: A generic iterative method.

The three classes of iterative methods discussed in this chapter are

- simple iterative methods, such as Jacobi iteration,
- incomplete factorisation schemes, such as SIP (Strongly Implicit Procedure, also known as Stone’s method),
- Krylov space methods, such as the Conjugate Gradient method, and
- multigrid schemes.

\(^1\)For a \( m \times m \times m \) three dimensional mesh, the storage required is \( n \times m^3 \), and the operation count of order \( O(nm^6) \). By changing from a cubic mesh to one where \( n = 10m \), the storage requirement is reduced by a factor of 215, and the operation count is reduced by a factor of 4637.
Convergence of an Iterative Scheme

Some method must be made to rank the fitness of the approximate solutions so that the decision to terminate the solvers iterative loop can be made. If the solution at iteration $k$ is $x^k$ and the exact solution to the equations is $x$, then the error at iteration $k$ is given by

$$e^k = x^k - x.$$  \hfill (3.21)

Of course the exact solution $x$ is unknown and thus so is the error. However we can easily calculate the residual at any step and the residual is proportional to the error, so if the residual decreases by a factor of $10^{-6}$ so does the error. The residual of the system (3.1) is defined by

$$r^k = b - A x^k.$$  \hfill (3.22)

Most iterative linear solvers include a calculation of the residual (or a close approximation) as part of the solution algorithm, and so with these methods the convergence of the solver can be monitored at no extra computational cost.

A scalar measure of the residual vector $r$’s length is given by its norm. A family of norms is given by

$$
\| r \|_\ell = \left( \frac{\sum_{i=1}^{N} |r_i|^{\ell} }{N} \right)^{1/\ell}.
$$  \hfill (3.23)

The one, two and infinity norms are,

$$
\| r \|_1 = \frac{\sum_{i=1}^{N} |r_i|}{N},
$$  \hfill (3.24)

$$
\| r \|_2 = \sqrt{ \frac{\sum_{i=1}^{N} r_i^2}{N} },
$$  \hfill (3.25)

and

$$
\| r \|_\infty = \max_{i=1\ldots N} |r_i|.
$$  \hfill (3.26)

For the tests described in the latter sections of this chapter the infinity norm was used as a measure of convergence.

The Residual Form of the Equations

If $\delta$ is the difference between successive solutions by an iterative method, $\delta = x^{k+1} - x^k$, then the system

$$A x = b,$$

can be rewritten as

$$A (x^k + \delta) = b,$$

$$A x^k + A \delta = b,$$

$$A \delta = b - A x^k.$$  \hfill (3.28)

The right hand side of the last equation in (3.28) is the residual of the $k^{th}$ iteration, and thus we can rewrite equation (3.27) in its residual form,

$$r^k = b - A x^k,$$

$$A \delta = r^k,$$

$$x^{k+1} = x^k + \delta.$$  \hfill (3.29)
This residual form of the system gives a greater accuracy than the original system when using finite precision arithmetic, since the differences between the $\delta$ values are likely to be of the same magnitude as the $\delta$ values themselves, in contrast to the $x$ terms where differences can be several orders of magnitude smaller than the values of $x$. Thus the rounding errors of the residual formulation will be less than for the original form of the equations.

### 3.1.4 Simple Iterative Methods

The simple iterative methods are characterised by the basic operation

$$ x^k = B x^{k-1} + c, \quad (3.30) $$

which is applied iteratively to the system to be solved. Jacobi’s method, Gauss–Seidel, Successive Over Relaxation (SOR), Symmetric Successive Over Relaxation (SSOR) and Red-Black Successive Over Relaxation (RBSOR) are all linear solvers of this form. These methods are the simplest to implement, but are typically the slowest to converge to a solution. However they can be effectively used as preconditioners for Krylov space methods, or smoothers for multigrid schemes.

#### Jacobi’s Method

In Jacobi’s method each equation is solved in isolation, so the equation (3.6)

$$ a_p x_p + \sum_{n_b} a_{nb} x_{nb} = b, \quad (3.31) $$

is modified into

$$ x_p = \frac{1}{a_p} \left( b - \sum_{n_b} a_{nb} x_{nb} \right). \quad (3.32) $$

This suggests an iterative method defined by,

$$ x_p^k = \frac{1}{a_p} \left( b - \sum_{n_b} a_{nb} x_{nb}^{k-1} \right), \quad (3.33) $$

where the $x$ terms on the right hand side of the equation are all from the previous iteration. This algorithm is given in Figure 3.6.

```
set $x^0$
$k = 1$
while $k \leq k_{\text{max}}$ and $\epsilon > \epsilon_{\text{max}}$
    $\delta x = b - A x^{k-1}$
    $x^k = x^{k-1} + \omega \delta x$
    $\epsilon = \| \delta x \|_{\infty}$
    $k = k + 1$
```

Figure 3.6: The Jacobi method.

The Jacobi method requires $N$ words of storage over that required for the storage of the linear system and its solution. Note that there is no intrinsic order in which to update each equation, so the method is trivial to parallelise.
Successive Over Relaxation (SOR)

If the equations in (3.32) are solved in sequence so that the updated values of \( x \) are used as soon as they are available, then we obtain the Gauss–Seidel algorithm,

\[
x_P^k = \frac{1}{a_P} \left( b - \sum_{nb=W,S,B} a_{nb} x_{nb}^k - \sum_{nb=E,N,T} a_{nb} x_{nb}^{k-1} \right),
\]

assuming a case where the equations are arranged in increasing order from west to east, south to north, bottom to top. If the iterative update is under- or over-relaxed then we arrive at the Successive Over Relaxation scheme,

\[
x_P^k = \frac{\omega}{a_P} \left( b - \sum_{nb=W,S,B} a_{nb} x_{nb}^k - \sum_{nb=E,N,T} a_{nb} x_{nb}^{k-1} \right) - (1 - \omega)x_P^{k-1}.
\]

The equations now contain a data dependency, with the new value of \( x_P^k \) depending on the previous equations updated values. This limits the ability to parallelise or vectorise the algorithm. The dependency can be removed by changing the order in which the equations are solved (such as is done with RBSOR) but this in turn will affect the rate of convergence, typically for the worse.

\[
\begin{align*}
\text{set } x^0 \\
k = 1 \\
\text{while } k \leq k_{\text{max}} \text{ and } \epsilon > \epsilon_{\text{max}} \\
\quad \text{for } i = 1, 2 \ldots N \\
\quad \delta x_i = \frac{1}{a_{ii}} \left( b - \sum_{j=1}^{i-1} a_{ii,j} x_j^k - \sum_{j=i}^{N} a_{ii,j} x_j^{k-1} \right) \\
\quad x_i^k = x_i^{k-1} + \omega \delta x_i \\
\quad \epsilon = || \delta x ||_{\infty} \\
\quad k = k + 1
\end{align*}
\]

Figure 3.7: The SOR algorithm.

The choice of the value of \( \omega \) is crucial to the convergence of the SOR method. A theorem due to Kahan[74] shows that SOR fails to converge if \( \omega \) is outside the interval (0,2). If the relaxation term \( \omega = 1 \) then SOR reduces to the Gauss-Seidel method.

Symmetric SOR and Red-Black SOR utilise the same algorithm but with changes made to the order of the operations. With Symmetric SOR every sweep through the equations in the order \( i = 1, 2 \ldots N \) is followed by a sweep in the reverse order, \( i = N, N-1, \ldots 1 \). With Red-Black SOR the operations are done in two passes, the first pass operating on the odd elements of the array, \( i = 1, 3 \ldots N-1 \), followed by the second pass which operates on the even elements, \( i = 2, 4 \ldots N \).

The advantages of Red-Black SOR is that dependencies between adjacent array values are removed, enabling the method to be vectorised or parallelised. However, as is seen in Figures (3.22) to (3.25), this is at the cost of a greatly reduced rate of convergence.

The SOR methods require no storage above that of the original system.

3.1.5 Incomplete Factorisation Methods

A second class of iterative methods is developed by using an approximate decomposition of the \( A \) array into it’s \( L, U \) factors. Instead of doing a complete factorisation only a restricted number of off-diagonal elements of the \( L \) and \( U \) matrices are allowed to take on a non-zero value, typically only
the diagonals that are non-zero in the \( A \) matrix and one or two further bands. For each iteration a forward and backward substitution process using this incomplete factorisation is applied to the residual formulation of the system of equations.

These methods are efficient in their own right, but also have value as preconditioners for the Krylov space solvers, and as smoothers for the multigrid schemes. The most commonly encountered implementations are Incomplete Cholesky factorisation (commonly used as a preconditioner for the Conjugate Gradient method) and the Strongly Implicit method of Stone[163].

**Incomplete Cholesky Factorisation (IC)**

The simplest incomplete factorisation scheme is the Incomplete Cholesky method, where a symmetric matrix \( A \) is factored into a \( LL^T \) system, with infill only in the locations where \( a_{i,j} \neq 0 \).

\[
\begin{align*}
\text{set } & x^0 \quad \text{set } x^0 \\
k = 1 \quad k = 1 \\
\text{while } k \leq k_{\text{max}} \text{ and } \epsilon > \epsilon_{\text{max}} \quad \text{while } k \leq k_{\text{max}} \text{ and } \epsilon > \epsilon_{\text{max}} \\
r^{k-1} = b - A x^{k-1} \quad r^{k-1} = b - A x^{k-1} \\
\epsilon = \|r^{k-1}\|_\infty \quad \epsilon = \|r^{k-1}\|_\infty \\
\text{for } l = 1, 2 \ldots n_z \quad \text{for } l = 1, 2 \ldots n_z \\
\text{for } j = 1, 2 \ldots n_y \quad \text{for } j = 1, 2 \ldots n_y \\
\text{for } i = 1, 2 \ldots n_x \quad \text{for } i = 1, 2 \ldots n_x \\
y_{i,j,l} = e_{i,j,l} (r_{i,j,l} - a_{w_{i,j,l}} y_{i-1,j,l} e_{i-1,j,l}) \quad y_{i,j,l} = e_{i,j,l} (r_{i,j,l} - a_{w_{i,j,l}} y_{i-1,j,l} e_{i-1,j,l}) \quad y_{i,j,l} = e_{i,j,l} (r_{i,j,l} - a_{w_{i,j,l}} y_{i-1,j,l} e_{i-1,j,l}) \\
& - a_{s_{i,j,l}} y_{i,j-1,l} e_{i,j-1,l} \quad - a_{s_{i,j,l}} y_{i,j-1,l} e_{i,j-1,l} \quad - a_{s_{i,j,l}} y_{i,j-1,l} e_{i,j-1,l} \\
& - a_{b_{i,j,l}} y_{i,j,l-1} e_{i,j,l-1} \quad - a_{b_{i,j,l}} y_{i,j,l-1} e_{i,j,l-1} \quad - a_{b_{i,j,l}} y_{i,j,l-1} e_{i,j,l-1} \\
x^{k} = x^{k-1} + y \quad x^{k} = x^{k-1} + y \\
k = k+1 \quad k = k+1
\end{align*}
\]

Figure 3.8: The Incomplete Cholesky method.

\[
\begin{align*}
\text{for } l = 1, 2 \ldots n_z \\
\text{for } j = 1, 2 \ldots n_y \\
\text{for } i = 1, 2 \ldots n_x \\
e_{i,j,l} &= \frac{1}{\sqrt{a_{p_{i,j,l}}^2 e_{i-1,j,l}^2 + a_{s_{i,j,l}}^2 e_{i,j-1,l}^2 + a_{b_{i,j,l}}^2 e_{i,j,l-1}^2}}
\end{align*}
\]

Figure 3.9: The Incomplete Cholesky and Incomplete LU factorisations.

If the original symmetric matrix is factored into diagonal, lower and upper triangular parts as

\[
A = D + L + L^T,
\]

then the incomplete factorisation \( M \) can be written as

\[
M = (E + L)(E + L^T),
\]
where the \( E \) array only has non-zero components on the diagonal, the values being
\[
e_{i,i} = \frac{1}{\sqrt{a_{i,i} - \sum_{j=1}^{i-1} (a_{i,j} e_{j,j})^2}},
\]
whilst the off-diagonal values of the \( L \) matrix have the same values as the corresponding elements in the \( A \) array.

The system of equations is then solved iteratively by the following process; for each iteration \( k \) the residual is calculated,
\[
r^{k-1} = b - A x^{k-1},
\]
then an update is calculated by solving
\[
(E + L) y = r^{k-1},
\]
\[
(E + L^T) z = y,
\]
with the update then being added to the previous iterations solution,
\[
x^k = x^{k-1} + z.
\]

The solution algorithm is given in Figure 3.8, with the factorisation being given in Figure 3.9.

```
set x^0
k = 1
while k \leq k_{\text{max}} \text{ and } \epsilon > \epsilon_{\text{max}}
\quad r^{k-1} = b - A x^{k-1}
\quad \epsilon = \| r^{k-1} \|_{\infty}
\quad \text{for } l = 1, 2 \ldots n_z
\quad \quad \text{for } j = 1, 2 \ldots n_y
\quad \quad \quad \text{for } i = 1, 2 \ldots n_x
\quad \quad \quad \quad y_{i,j,l} = e_{i,j,l} (r_{i,j,l} - a_{W_{i,j,l}} y_{i-1,j,l} - a_{S_{i,j,l}} y_{i,j-1,l} - a_{B_{i,j,l}} y_{i,j,l-1})
\quad \quad \text{for } l = n_z, n_z-1, \ldots 1
\quad \quad \text{for } j = n_y, n_y-1, \ldots 1
\quad \quad \quad \text{for } i = n_x, n_x-1, \ldots 1
\quad \quad \quad \quad y_{i,j,l} = y_{i,j,l} - e_{i,j,l} (a_{E_{i,j,l}} y_{i+1,j,l} + a_{N_{i,j,l}} y_{i,j+1,l} + a_{T_{i,j,l}} y_{i,j,l+1})
\quad \quad \text{x^k = x^{k-1} + y}
\quad k = k+1
```

Figure 3.10: The Incomplete LU method.

The method is not as fast as the SIP and MSI formulations described below, and is not often used as a solver. However it has frequently been used as a preconditioner for the Conjugate Gradient method, the combination being referred to as the Incomplete Cholesky–Conjugate Gradient method or ICCG. The solver can be modified to remove the need to take square roots in the factorisation of the equations (the Incomplete LDL method), and to be applied to non-symmetric systems (the Incomplete LU method (ILU)). The factorisation and solution algorithms for the ILU solver are given in Figures 3.10 and 3.11.

The factorisation step of the IC and ILU solvers only requires the storage of the diagonal of the \( E \) matrix, whilst the solvers themselves require the storage of the residual \( r \). Therefore the memory usage of the method is only \( 2N \) words.
Strongly Implicit Procedure (SIP)

The Strongly Implicit Procedure (SIP) of Stone[163] is a more advanced version of the Incomplete LU scheme, but unlike the IC and ILU methods it is only applicable to the equations resulting from a finite volume discretisation of a PDE. The approximate LU decomposition of $A$ that is used is

$$M = LU = A + N,$$  \hspace{1cm} (3.42)

where $N$ is the error between the exact and approximate factorisations.

If $L$ and $U$ are constrained to be only non-zero in the locations that $A$ is non-zero then the matrix $M$ will have more non-zero diagonals than $A$: two extra non-zero diagonals if $A$ is a five-diagonal matrix resulting from a two dimensional PDE, or six extra diagonals if it is a seven-diagonal matrix from a three dimensional PDE.

To make $M$ a good approximation of $A$, the $N$ array is set such that

$$Nx \approx 0.$$  \hspace{1cm} (3.43)

This is done by recognising that the system being solved is from a finite volume approximation of a PDE. Thus the values of the $x$ field in the extra diagonals of $N$ can be approximated by a second order extrapolation of the values of $x$. By putting the terms for the extrapolation into the elements of $N$ and cancelling with the values of $x$ in the extra diagonals of $N$ then the system can be made to approximate equation (3.43). Finally, to make the LU factorisation unique the diagonal elements of $U$ are set to 1.

The system of equations is then solved iteratively in a similar manner to the solution of the Incomplete Cholesky system; for each iteration $k$ the residual is calculated,

$$r^{k-1} = b - A x^{k-1},$$  \hspace{1cm} (3.44)

then an update is calculated by solving

$$L y = r^{k-1},$$  \hspace{1cm} (3.45)

$$U z = y,$$

with the update then being added to the previous iterations solution,

$$x^k = x^{k-1} + z.$$  \hspace{1cm} (3.46)

As is seen in Figures (3.22) to (3.25), the SIP solver is much faster than the simpler ILU and IC schemes, and so is suitable as a solver in its own right, as well as a smoother with other iterative methods. It requires $6N$ words of storage for the solution of two dimensional PDEs, and $8N$ words for three dimensional PDEs.
set $x^0$
\[ k = 1 \]
while $k \leq k_{\text{max}}$ and $\epsilon > \epsilon_{\text{max}}$
\[ r_{k-1} = b - A x_k \]
\[ \epsilon = \| r_{k-1} \|_\infty \]
for $l = 1, 2 \ldots n_z$
for $j = 1, 2 \ldots n_y$
for $i = 1, 2 \ldots n_x$
\[ y_{i,j,l} = f_{i,j,l} \left( r_{i,j,l} - e_{i,j,l} y_{i-1,j,l} - d_{i,j,l} y_{i,j-1,l} - c_{i,j,l} y_{i,j,l-1} \right) \]
for $l = n_z, n_z-1, \ldots 1$
for $j = n_y, n_y-1, \ldots 1$
for $i = n_x, n_x-1, \ldots 1$
\[ y_{i,j,l} = y_{i,j,l} - s_{i,j,l} y_{i+1,j,l} - u_{i,j,l} y_{i,j+1,l} - v_{i,j,l} y_{i,j,l+1} \]
\[ x^k = x^{k-1} + y \]
\[ k = k+1 \]

Figure 3.12: The Strongly Implicit Procedure (SIP) of Stone.

for $l = 1, 2 \ldots n_z$
for $j = 1, 2 \ldots n_y$
for $i = 1, 2 \ldots n_x$
\[ c_{i,j,l} = a_{B_{i,j,l}} / (1 + \alpha (s_{i,j,l-1} + u_{i,j,l-1})) \]
\[ d_{i,j,l} = a_{S_{i,j,l}} / (1 + \alpha (s_{i,j-1,l} + u_{i,j-1,l})) \]
\[ e_{i,j,l} = a_{W_{i,j,l}} / (1 + \alpha (s_{i-1,j,l} + u_{i-1,j,l})) \]
\[ \beta = c_{i,j,l} s_{i,j,l-1} + d_{i,j,l} s_{i,j-1,l} \]
\[ \gamma = d_{i,j,l} v_{i,j-1,l} + e_{i,j,l} s_{i-1,j,l} \]
\[ \rho = e_{i,j,l} u_{i-1,j,l} + c_{i,j,l} s_{i,j,l-1} \]
\[ f_{i,j,l} = 1 / \left( a_{P_{i,j,l}} + \alpha (\beta + \gamma + \rho) - (e_{i,j,l} s_{i-1,j,l}) \right) \]
\[ s_{i,j,l} = f_{i,j,l} (a_{E_{i,j,l}} - \alpha \beta) \]
\[ u_{i,j,l} = f_{i,j,l} (a_{N_{i,j,l}} - \alpha \gamma) \]
\[ v_{i,j,l} = f_{i,j,l} (a_{T_{i,j,l}} - \alpha \rho) \]

Figure 3.13: The incomplete LU factorisation used in SIP.
CHAPTER 3. LINEAR SOLVERS

Modified Strongly Implicit procedure (MSI)

The Strongly Implicit Procedure was further developed by Schneider and Zedan[151, 185] into the Modified Strongly Implicit procedure (MSI). In this method the $L$ and $U$ arrays chosen in the decomposition in equation (3.42) are allowed to have more non-zero elements than the equation matrix $A$.

$$
\begin{align*}
\text{set } & x^0 \\
& k = 1 \\
\text{while } & k \leq k_{\text{max}} \text{ and } \varepsilon > \varepsilon_{\text{max}} \\
& r^{k-1} = b - A x^{k-1} \\
& \varepsilon = \| r^{k-1} \|_{\infty} \\
\text{for } l = 1, 2 \ldots n_z \\
& \text{for } j = 1, 2 \ldots n_y \\
& \text{for } i = 1, 2 \ldots n_x \\
& y_{i,j,l} = p_{i,j,l} \left(r_{i,j,l} - c_{i,j,l} y_{i,j,l-1} - d_{i,j,l} y_{i+1,j,l-1} - e_{i,j,l} y_{i,j,l-1} - f_{i,j,l} y_{i,j-1,l} - g_{i,j,l} y_{i+1,j,l-1} - h_{i,j,l} y_{i,j-1,l} \right) \\
& \text{for } l = n_z, n_z-1, \ldots 1 \\
& \text{for } j = n_y, n_y-1, \ldots 1 \\
& \text{for } i = n_x, n_x-1, \ldots 1 \\
& y_{i,j,l} = y_{i,j,l} - q_{i,j,l} y_{i+1,j,l} - s_{i,j,l} y_{i-1,j,l} + t_{i,j,l} y_{i,j+1,l} - u_{i,j,l} y_{i,j-1,l+1} - v_{i,j,l} y_{i-1,j,l+1} - w_{i,j,l} y_{i,j,l+1} \\
& x^k = x^{k-1} + y \\
& k = k + 1
\end{align*}
$$

Figure 3.14: The Modified Strongly Implicit procedure (MSI) of Schneider and Zedan.

The code for the MSI method is more complicated than for the SIP scheme and it requires more memory, using $8N$ and $14N$ words of storage for two and three dimensional PDEs respectively. However it converges much faster than the other incomplete factorisation methods. It is infrequently reported in the literature and seems to be less frequently used, presumably due the increased complexity of the implementation and the extra storage used.
for \( l = 1, 2 \ldots n_z \)
for \( j = 1, 2 \ldots n_y \)
for \( i = 1, 2 \ldots n_x \)
\[
c_{i,j,l} = a_{B_{i,j,l}}/(1 + \alpha(s_{i,j,l-1} - q_{i,j,l-1}(q_{i+1,j,l-1} + t_{i+1,j,l-1}))
\]
\[
d_{i,j,l} = -c_{i,j,l} q_{i,j,l-1}
\]
\[
e_{i,j,l} = -c_{i,j,l} t_{i,j,l-1} - d_{i,j,l} s_{i+1,j,l-1}
\]
\[
f_{i,j,l} = (a_{S_{i,j,l}} - c_{i,j,l} u_{i,j,l-1}
+ \alpha(d_{i,j,l} u_{i+1,j,l}(q_{i+1,j,l-1} + 2u_{i+1,j,l-1} + w_{i+1,j,l-1})
- u_{i-1,j,l}(a_{W_{i,j,l}} - c_{i,j,l} v_{i,j,l-1})))
/(1 + \alpha(2 u_{i-1,j,l} + v_{i,j,l-1} - u_{i-1,j,l} s_{i,j,l-1}
- q_{i,j,l-1}(q_{i+1,j,l-1} + 2u_{i+1,j,l-1} + w_{i+1,j,l-1})))
\]
\[
g_{i,j,l} = -d_{i,j,l} u_{i+1,j,l-1} - f_{i,j,l} q_{i,j,l-1}
\]
\[
h_{i,j,l} = (a_{W_{i,j,l}} - c_{i,j,l} u_{i,j,l-1} - f_{i,j,l} s_{i,j,l-1}
- \alpha(c_{i,j,l} s_{i,j,l-1} + e_{i,j,l} s_{i+1,j,l-1} + d_{i,j,l} v_{i,j,l-1})))
/(1 + \alpha(2 * s_{i-1,j,l} + u_{i-1,j,l} + 2v_{i-1,j,l})))
\]
\[
\theta_1 = d_{i,j,l} q_{i+1,j,l-1}
\]
\[
\theta_2 = c_{i,j,l} s_{i,j,l-1}
\]
\[
\theta_3 = d_{i,j,l} t_{i-1,j,l-1} + e_{i,j,l} q_{i+1,j,l-1}
\]
\[
\theta_4 = e_{i,j,l} s_{i,j,l-1}
\]
\[
\theta_5 = c_{i,j,l} t_{i-1,j,l-1}
\]
\[
\theta_6 = G_{i,j,l} q_{i+1,j,l-1}
\]
\[
\theta_7 = h_{i,j,l} s_{i-1,j,l}
\]
\[
\theta_8 = f_{i,j,l} u_{i,j,l-1}
\]
\[
\theta_9 = g_{i,j,l} u_{i-1,j,l}
\]
\[
\theta_{10} = f_{i,j,l} w_{i+1,j,l-1} + h_{i,j,l} u_{i-1,j,l}
\]
\[
\theta_{11} = g_{i,j,l} w_{i+1,j,l-1}
\]
\[
\theta_{12} = h_{i,j,l} v_{i-1,j,l}
\]
\[
q_{i,j,l} = p_{i,j,l}(a_{E_{i,j,l}} - d_{i,j,l} w_{i+1,j,l-1} - g_{i,j,l} t_{i+1,j,l-1}
- \alpha(2(\theta_1 + \theta_6) + \theta_3 + \theta_9 + \theta_{11})))
\]
\[
s_{i,j,l} = -p_{i,j,l}(e_{i,j,l} w_{i+1,j,l-1} + h_{i,j,l} t_{i-1,j,l})
\]
\[
t_{i,j,l} = p_{i,j,l}(a_{N_{i,j,l}} - c_{i,j,l} v_{i+1,j,l-1}
- \alpha(2(\theta_2 + \theta_7 + \theta_9 + \theta_4)))
\]
\[
u_{i,j,l} = -p_{i,j,l}(f_{i,j,l} w_{i-1,j,l} + g_{i,j,l} u_{i+1,j-1,l})
\]
\[
v_{i,j,l} = -p_{i,j,l} h_{i,j,l} w_{i-1,j,l}
\]
\[
w_{i,j,l} = p_{i,j,l}(a_{T_{i,j,l}} - \alpha(\theta_8 + \theta_9 + \theta_{10} + \theta_{11} + \theta_{12})
\]

Figure 3.15: The incomplete LU factorisation used in the MSI method.
3.1.6 Krylov Space Methods

In Krylov space methods the solution of the system of equations is viewed as an optimisation problem, with the goal being the minimisation of the residual of the system. The Conjugate Gradient method is the oldest of the Krylov space solvers, the classical description of the method being given in the joint paper by Hestenes and Stiefel[60]. It was originally developed as a direct method, but Reid[138] popularised the solver as an iterative scheme for sparse matrix problems.

Since the development of the Conjugate Gradient scheme a number of other Krylov space schemes have been devised, with good summaries of the methods being given in the books by Barrett et al[7] and Golub and Van Loan[51]. A more intuitive introduction to the methods is given in the paper by Shewchuk[154].

In this chapter the Conjugate Gradient (CG), the Bi-Conjugate Gradient Stabilised (BiCGSTAB), and the General Minimalised Residual (GMRES) methods are discussed. Other methods which are commonly encountered in the literature include the Steepest Descent (SD), the Bi-Conjugate Gradient (BiCG), the Conjugate Gradient Squared (CGS), and the Quasi-Minimal Residual (QMR) methods. These additional methods were also implemented for this study but offered no advantages over the three methods discussed below. They are briefly discussed at the end of the section, but for a fuller description the reader is referred to Barrett et al[7] and Golub and Van Loan[51].

Conjugate Gradient Method (CG)

The Conjugate Gradient method is the oldest of the Krylov space methods. It was independently developed by Hestenes and Stiefel[60] as a direct method, but its popularity dates from the discovery by Reid[138] that it could be effectively used as an iterative method.

Following the presentation given in Barrett et al[7], for each iteration the residual is minimised along a path orthogonal to the previous search. Thus the solver steps along the residual surface in the solution space to find the minimum residual. At each iteration the iterate \( x^k \) is updated by a multiple of the search direction vector \( p^k \),

\[
x^k = x^{k-1} + \alpha_k p^k.
\]

The residual \( r^k = b - Ax^k \) is updated as,

\[
r^k = r^{k-1} - \alpha_k q^k \quad \text{where} \quad q^k = Ap^k.
\]

The choice \( \alpha_k = \frac{r^T r}{p^T A p} \) minimises \( r^T A r^k \) for all choices of \( \alpha \).

The search directions are updated using the residual,

\[
p^k = r^k + \beta_{k-1} p^{k-1},
\]

where the choice

\[
\beta_k = \frac{r^T r}{r^T r^{k-1}}
\]

ensures that \( r^k \) and \( r^{k-1} \), or \( p^k \) and \( A p^{k-1} \) are orthogonal.

The pseudo code for the preconditioned conjugate gradient method is given in Figure 3.16, the preconditioning being the “solve \( M z^{k-1} = r^{k-1} \)” operation. If \( M \) is set to the identity matrix \( I \) then for each iteration \( z = r \) and the algorithm simplifies to its unpreconditioned form. The preconditioned form of the solver requires \( 4N \) words of storage (not including that required by the preconditioner), whilst the unpreconditioned form requires \( 3N \) words.
set $x^0$
$r^0 = b - A x^0$
$k = 1$
while $k \leq k_{\text{max}}$ and $\epsilon > \epsilon_{\text{max}}$
solve $M z^{k-1} = r^{k-1}$
$\rho_{k-1} = r^{k-1} T z^{k-1}$
if $k = 0$
$p^1 = z^0$
else
$\beta_{k-1} = \frac{\rho_{k-1}}{\rho_{k-2}}$
$p^k = z^{k-1} + \beta_{k-1} p^k$
$q^k = \Lambda p^k$
$\alpha_k = \frac{\rho_{k-1}}{p^k q^k T q^k}$
$x^k = x^{k-1} + \alpha_k p^k$
$r^k = r^{k-1} - \alpha_k q^k$
$\epsilon = ||r^k||_\infty$
$k = k + 1$

Figure 3.16: The preconditioned Conjugate Gradient algorithm.

The Conjugate Gradient method works only for symmetric positive definite systems. Thus it cannot be used to solve the general transport equation, but can be used for diffusion equations or the pressure correction equation of the SIMPLE coupling schemes described in Section 4.2.1.

Since it requires a symmetric system, any preconditioner used with the solver must preserve this symmetry. Thus Jacobi’s method, Red Black SOR and Symmetric-SOR can be used as preconditioners, but not SOR. The incomplete LU solvers make excellent preconditioners, with the incomplete Cholesky method commonly being used in early implementations of the solver. This combination of solver and preconditioner is referred to as the ICCG (Incomplete Cholesky-Conjugate Gradient) method.

**Generalised Minimal Residual (GMRES)**

The Generalised Minimal Residual method is designed to solve non-symmetric linear systems, using a sequence of orthogonal vectors. Thus unlike the conjugate gradient method it can be applied to the transport equation, but it does so at the cost of requiring the storage and use of the sequence of solution vectors.

The GMRES iterates are constructed as the series

$$x^k = x^0 + y_1 v^1 + \ldots + y_k v^k,$$

where the $y_i$ coefficients have been chosen to minimise the residual norm. The number of operations in the calculation of the $x^k$ iterate thus increases linearly with the number of iterations, as does the storage used. To place an upper limit on the storage required by the scheme, the solver is commonly implemented with a restart after $k_{\text{restart}}$ iterations, limiting the memory usage to $N + k_{\text{restart}}(N + k_{\text{restart}} + 4)$ words of storage.
The algorithm for the restarted GMRES solver is given in Figure 3.17. It is taken from the method suggested by Saad and Schultz[145].

\[
\begin{align*}
\text{set } x^0 \\
k = 1 \\
\text{while } k \leq k_{\text{max}} \text{ and } \epsilon > \epsilon_{\text{max}}
\text{r} &= b - Ax \\
\text{if } (k = 1) \quad \epsilon^1 &= ||\text{r}||_2 \\
solve & M v^1 = r \\
\text{if } (k = 1) \quad \sigma &= \epsilon^1 / ||v^1||_2 \\
s &= 0 \\
s_1 &= ||v^1||_2 \\
v^1 &= v^1 / ||v^1||_2 \\
\text{for } i = 1, 2 \ldots k_{\text{restart}} \text{ while } \epsilon < \epsilon_{\text{max}}
\text{r} &= A v^i \\
solve & M v^{i+1} = r \\
\text{for } j = 1, 2 \ldots i \\
h_{j,i} &= v^{i+1}^T v^j \\
v^{i+1} &= v^{i+1} - h_{j,i} v^j \\
h_{i+1,i} &= ||v^{i+1}||_2 \\
v^{i+1} &= v^{i+1} / h_{i+1,i} \\
\text{for } j = 1, 2 \ldots i \text{ } i - 1 \\
\alpha &= t_i h_{j,i} + u_j h_{j+1,i} \\
\beta &= -u_i h_{j,i} + t_j h_{j+1,i} \\
h_{j,i} &= \alpha \\
h_{j+1,i} &= \beta \\
t_i &= h_{i,i} / \sqrt{h_{i,i}^2 + h_{i+1,i}^2} \\
u_i &= h_{i+1,i} / \sqrt{h_{i,i}^2 + h_{i+1,i}^2} \\
h_{i,i} &= \sqrt{h_{i,i}^2 + h_{i+1,i}^2} \\
h_{i+1,i} &= 0 \\
s_{i+1} &= -u_i s_i \\
s_i &= t_i s_i \\
\epsilon &= |s_{i+1}| \\
k &= k + 1 \\
\text{for } j = i, i - 1 \ldots 1 \\
y_j &= s_j / h_{j,j} \\
\text{for } l = j - 1, j - 2 \ldots 1 \\
s_l &= s_l - y_j h_{l,j} \\
\text{for } j = i, i - 1 \ldots 1 \\
x &= x + y_j v^j
\end{align*}
\]

Figure 3.17: The Preconditioned restarted GMRES Method.
Bi-Conjugate Gradient Stabilised (BiCGSTAB)

The Bi-Conjugate Gradient Stabilised method was developed by Van der Vorst[174] from the CGS, BiCG and GMRES methods, in order to solve non-symmetric systems of equations whilst avoiding the often highly irregular convergence patterns of CGS and BiCG, and the large storage requirements of GMRES.

The method can be applied to non-symmetric systems, and is quite robust and efficient. Like the conjugate gradient method it only stores a limited number of vectors at any iteration, requiring $8N$ words of memory ($6N$ in its unpreconditioned form), and so unlike the GMRES solver doesn’t increase its per-iteration memory use and operation count with increasing number of iterations.

Other Krylov Space Methods

Other Krylov space solvers implemented and tested in this study were the Steepest Descent (SD), the Bi-Conjugate Gradient (BiCG), the Conjugate Gradient Squared (CGS), and the Quasi-Minimal Residual (QMR) methods.

The Steepest Descent scheme (SD) is an optimisation method—by minimising the residuals of the linear equations it arrives at the equations solution[154]. For the current iteration the solver corrects the solution in the direction of the steepest downward gradient. It is simple but inefficient.

The preconditioned Bi-Conjugate Gradient Stabilised algorithm.

set $x^0$
$r^0 = b - A x^0$
$f^r = r^0$
$k = 1$

while $k \leq k_{\text{MAX}}$ and $\varepsilon > \varepsilon_{\text{MAX}}$

$\rho_{k-1} = f^r T x^{k-1}$

if $\rho_{k-1} = 0$ or $\omega_{k-1} = 0$ method fails

if $k = 1$

$p^1 = r^0$

else

$\beta_{k-1} = \frac{\rho_{k-1} \omega_{k-1}}{\rho_{k-1} \omega_{k-1} + \rho_k}$

$p^k = x^{k-1} + \beta_{k-1} (p^{k-1} - \omega_{k-1} v^{k-1})$

solve $M \tilde{p} = p$

$v^k = A \tilde{p}$

$\alpha_k = \frac{\rho_k}{\rho_k + \omega_{k-1} \rho_{k-1}}$

$s = r^{k-1} - \alpha_k v^k$

$\varepsilon = \| s^k \|_{\infty}$

if $\varepsilon > \varepsilon_{\text{MAX}}$

$x^k = x^{k-1} + \alpha_k \tilde{p}$

stop

solve $M \tilde{s} = s$

$t = A \tilde{s}$

$\omega_k = \frac{t^T t}{t^T \tilde{s}}$

$x^k = x^{k-1} + \alpha_k p^k + \omega_k \tilde{s}$

$r^k = s - \omega_k t$

$\varepsilon = \| r^k \|_{\infty}$

$k = k + 1$
The Bi-Conjugate Gradient (BiCG) and Conjugate Gradient Squared (CGS) methods were developed to free the CG solver from its limitation of only being applicable to symmetric systems. For the BiCG method the orthogonal sequence used in the CG method is replaced by two mutually orthogonal sequences, one based on the system $A$ and the other on its transpose $A^T$. The CGS solver is a modification of the BiCG solver that applies the updating operations for the $A$ sequence and the $A^T$ sequence to both vectors. Ideally this would double the convergence rate, but in practice convergence is very irregular.

The Quasi-Minimal Residual (QMR) method applies a least squares solve and update to the BiCG residuals, smoothing out the convergence of the method and preventing the breakdowns that can occur with BiCG.

None of these methods was found to be superior to the methods discussed above, and so for the sake of brevity detailed descriptions of the methods and data of their behaviour have been omitted. For further information on these methods the reader is directed to Barrett et al and Golub and Van Loan.

Preconditioners

The rate of convergence of the Krylov space methods is often improved by the use of a preconditioner. This is expressed in the algorithms above as a step of the form,

$$Mz = r,$$

where $r$ is a current residual field, $z$ is the preconditioned residual, and $M$ is a matrix having similar properties to $A$. If $M$ is identical to $A$, then preconditioning entails solving the set of equations, which is what we were using the Krylov space method to do in the first place. However if $M$ only resembles $A$, then the convergence of the Krylov space method will still be improved by the preconditioning.

![Convergence of MSI, CG and CG-MSI Solvers](image)

Figure 3.19: Comparison of the convergence of the MSI, the unpreconditioned CG, and the MSI preconditioned CG solvers.

The convergence of an unpreconditioned CG solver, and a MSI solver together with a MSI preconditioned CG solver are shown in Figure 3.19. The conjugate gradient solver converges slowly at first, with the convergence rate increasing after 10 seconds of solution time. In contrast, the incomplete
factorisation method has an initially fast rate of convergence that slows in the later iterations. Combining the two methods by using the MSI as a preconditioner for the CG scheme results in a method that is faster than either the CG or MSI solvers.

Among the easiest preconditioning methods to implement are the Jacobi and Symmetric SOR algorithms. More complex methods such as incomplete Cholesky decomposition, Incomplete LU methods, and the multigrid schemes give faster convergence, at the expense of greater memory usage and a more complex implementation.

### 3.1.7 Multigrid Methods

Whilst the simple iterative and incomplete factorisation solvers are fast for small mesh sizes, their rate of convergence decreases with an increasing number of equations. This is partly due to the increased number of operations with the increased number of mesh points, but mostly due to the fact that these methods are much faster at smoothing the small wavelength components than the long wavelength components of the error. One way to improve their efficiency is the multigrid technique, whereby a PDE is solved on a series of meshes with a varying number of mesh points. The seminal paper for the method is that of Brandt[16], whilst Briggs[18] gives an excellent tutorial on the technique.

For the multigrid method discussed here, a set of equations is given for a PDE discretised on a fine mesh. The multigrid scheme then transforms the equations onto a series of progressively coarser meshes, solving the equations fully on the coarsest mesh. The solution is then solved on the series of successively finer meshes, using the solution from the previous (coarser) mesh as an initial estimate of the solution, finally solving on the finest mesh. By transferring from a fine to a coarse mesh the medium wavelength errors in the fine mesh solution are transformed into short wavelength errors on the coarse mesh which are much easier to smooth. The computation cost for solving on the coarse meshes is low, and the cost for solving on the finer meshes is reduced by using the coarse mesh solution as an initial estimate.

The three basic operators for the multigrid technique are the smoother, which improves the current estimate of the solution on a given mesh, and the restriction and prolongation operators, which map a set of equations and a solution between a fine and a coarse mesh.

Given two meshes, with the fine mesh having a mesh spacing of $h$, and the coarse mesh having a spacing of $2h$. The restriction operator maps the fine mesh solution onto the coarse mesh, and is written

$$x^{2h} = R x^h,$$

whilst the prolongation operator performs the inverse operation, interpolating a coarse mesh solution onto a fine mesh,

$$x^h = P x^{2h}.$$

The basic algorithm for a multigrid scheme is given in Figure 3.20 For each iteration the current solution is smoothed using some smoothing operator, typically a few iterations of one of the simple iterative or incomplete factorisation methods. The residual for the current solution is calculated and then restricted onto the coarse mesh and solved. The solution from the coarse mesh system is then interpolated onto the finer mesh using the prolongation operator, and is added to the previous iterations solution.

For all but the coarsest mesh the multigrid solver is recursively applied to solve the restricted system of equations. On the coarsest mesh the restricted system is solved either by an iterative method, solving the equations to full convergence, or by the use of a direct method.

On the finest mesh the infinity norm of the residual is taken and compared to a supplied tolerance. If the norm of the residual is less than the tolerance then the solver is taken to have converged and the
process terminates. Otherwise the solver is applied repeatedly to the system.

For the algorithm given in Figure 3.20 the superscript \( k \) refers to the iteration number, whilst the superscript \( 2^h \) specifies that the variable applies to the coarse mesh.

### Prolongation and Restriction Operators

For the solution of the equations resulting from a finite volume discretisation of a PDE two types of prolongation and restriction schemes can be used – either the equations can be rederived on each mesh, or some form of operation can be used to generate the coarse mesh equations from the fine mesh system.

For the first type of prolongation and restriction scheme, where the equations are rederived and the boundary conditions re-applied on each mesh, the solver is necessarily closely tied to the discretisation of the PDE. The resulting code is not very general, and for complicated differencing schemes the calculation of the equations can be slow\(^2\). For such a scheme the value of the solution at each point can be transferred to the corresponding point on the coarser/finer mesh, an operation called straight injection. For a finite volume solver this carries some extra overhead in that the cell centres of two meshes don’t align, and instead of a simple injection process the solution must be averaged over the cells.

For a black box solver that is not directly coupled to the discretisation process the prolongation and restriction operators must be derived from the fine mesh equations rather than re-discretised from the underlying equations. A method that applies to the solution of PDEs is developed below. The method is applied to meshes that have \( 2^n + 3 \) nodes along each axis including boundary nodes (ie: \( 2^n + 1 \) internal points on each axis). However it can be used for systems with \( m2^n + 3 \) nodes on each axis, where \( m + 1 \) is the number of points along the axis on the coarsest mesh. The method can be used for the equations arising from both finite volume and finite difference differencing, and straight injection can be used for transferring fields between different meshes without the problems of averaging solutions as can the case with the finite volume schemes. In addition the boundary

\[^2\text{Moreover, for some equations such as the pressure correction equation in SIMPLE coupling schemes, the variables are defined only upon the mesh they were created on.}\]
conditions are automatically applied in the equations restriction operation.

For a simple system resulting from the finite volume or finite difference discretisation of a one dimensional PDE, the equations are (using the natural ordering),

\[
\begin{bmatrix}
  a_{P_1} & a_{E_1} \\
  a_{W_2} & a_{P_1} & a_{E_2} \\
  a_{W_3} & a_{P_2} & a_{E_3} \\
  a_{W_4} & a_{P_3} & a_{E_4} \\
  a_{W_5} & a_{P_4} & a_{E_5}
\end{bmatrix}
\begin{bmatrix}
  x_1 \\
  x_2 \\
  x_3 \\
  x_4 \\
  x_5
\end{bmatrix}
= \begin{bmatrix}
  b_1 \\
  b_2 \\
  b_3 \\
  b_4 \\
  b_5
\end{bmatrix} .
\] (3.55)

For the above system the \( x_1 \) and \( x_5 \) nodes are interior boundary points (ie: the first row of points on the interior of a solution domain), and the boundary conditions have been applied in a form that removes the need for the points physically located on the boundary (see Section 2.3).

For an elliptic PDE the solution must be reasonably smooth, and so the solution values at the even numbered points in the mesh can be estimated from a second order centred interpolation from the odd numbered points,

\[
x_2 = \frac{1}{2}(x_1 + x_3),
\]

\[
x_4 = \frac{1}{2}(x_3 + x_5).
\] (3.56)

By substituting these equations into the initial system, a system with only half the equations of the original system is generated,

\[
\begin{bmatrix}
  2a_{P_1} + a_{E_1} \\
  2a_{W_5} + a_{E_1} \\
  2a_{W_3} + a_{P_1} & a_{E_3} & a_{W_5} \\
  2a_{P_3} + a_{W_5} + a_{E_3} \\
  2a_{P_5} + a_{W_3} + a_{E_5}
\end{bmatrix}
\begin{bmatrix}
  x_1 \\
  x_2 \\
  x_3 \\
  x_4 \\
  x_5
\end{bmatrix}
= \begin{bmatrix}
  b_1 \\
  b_2 \\
  b_3 \\
  b_4 \\
  b_5
\end{bmatrix} .
\] (3.57)

which can be rewritten

\[
\begin{bmatrix}
  a_{P_1}^{2h} & a_{E_1}^{2h} \\
  a_{W_5}^{2h} & a_{E_3}^{2h} & a_{W_5}^{2h} \\
  a_{W_3}^{2h} & a_{P_1}^{2h} & a_{E_3}^{2h} \\
  a_{P_3}^{2h} & a_{W_5}^{2h} + a_{E_3}^{2h} \\
  a_{P_5}^{2h} & a_{W_3}^{2h} + a_{E_5}^{2h}
\end{bmatrix}
\begin{bmatrix}
  x_1 \\
  x_2 \\
  x_3 \\
  x_4 \\
  x_5
\end{bmatrix}
= \begin{bmatrix}
  b_1^{2h} \\
  b_2^{2h} \\
  b_3^{2h} \\
  b_4^{2h} \\
  b_5^{2h}
\end{bmatrix} ,
\] (3.58)

where

\[
a_{P_i}^{2h} = 2a_{P_{i-1}} + a_{E_{2i-1}},
\]

\[
a_{E_i}^{2h} = 2a_{E_{2i-1}},
\]

\[
a_{W_i}^{2h} = 2a_{W_{2i-1}},
\]

\[
b_i^{2h} = 2b_{2i-1}.
\] (3.59)

For these equations, restricting the field from the one mesh to the next coarser mesh can be accomplished using simple injection,

\[
x_i^{2h} = x_{2i-1}.
\] (3.60)

The corresponding prolongation operator is

\[
x_i = x_{\frac{i}{2} + \frac{1}{4}}, \quad i = 1, 3, ...
\] (3.61)

for the odd numbered points, whilst the even numbered points are found by linear interpolation

\[
x_i = \frac{1}{2}(x_{\frac{i}{2} + \frac{1}{4}} + x_{\frac{i}{2}}), \quad i = 2, 4 ...
\] (3.62)

For a two dimensional system the equation restriction operator becomes

\[
a_{P_{i,j}}^{2h} = 2a_{P_{i+1,j}}, a_{E_{i,j}} + a_{E_{i+1,2j+1}}, a_{W_{i,j}} + a_{W_{i+1,2j+1}}, a_{S_{i-1,j+1}}, a_{S_{i-1,j+1}},
\]

\[
a_{P_{i,j}}^{2h} = a_{E_{i+1,2j+1}},
\]

\[
a_{W_{i,j}}^{2h} = a_{W_{i+1,2j+1}},
\]

\[
a_{E_{i,j}}^{2h} = a_{E_{i+1,2j+1}},
\]

\[
a_{S_{i,j}}^{2h} = a_{S_{i-1,2j+1}},
\]

\[
b_{i,j}^{2h} = 2b_{2i+1,2j+1},
\] (3.63)
the field restriction is
\[ x_{i,j}^{2h} = x_{2i-1,2j-1}, \tag{3.64} \]
and the prolongation operator becomes
\[ x_{i,j} = x_{h}^{2h}, \quad i = 1, 3, \ldots, j = 1, 3, \ldots \tag{3.65} \]
with the remaining points being found by bilinear interpolation.

Similarly in three dimensions the equation restriction operators are
\[
\begin{align*}
  a_{l,j}^{2h} &= 2 a_{l,j-1,2j-1} + a_{E,l-1,2j-1,2h-1} + a_{W,l+1,2j-1,2h-1} + a_{N,l-1,2j-1,2h-1} + a_{S,l+1,2j-1,2h-1} \\
  a_{l,j}^{2h} &= a_{E,l-1,2j-1,2h-1} + a_{W,l+1,2j-1,2h-1} \\
  a_{l,j}^{2h} &= a_{N,l-1,2j-1,2h-1} + a_{S,l+1,2j-1,2h-1} \\
  a_{l,j}^{2h} &= a_{T,l-1,2j-1,2h-1} + a_{H,l+1,2j-1,2h-1} \\
  a_{l,j}^{2h} &= a_{E,l-1,2j-1,2h-1} + a_{W,l+1,2j-1,2h-1} + a_{N,l-1,2j-1,2h-1} + a_{S,l+1,2j-1,2h-1} + a_{T,l-1,2j-1,2h-1} + a_{H,l+1,2j-1,2h-1} \\
  b_{l,j}^{2h} &= 2 b_{l,j-1,2j-1,2h-1} \\
  c_{l,j}^{2h} &= b_{l,j-1,2j-1,2h-1} + b_{l,j+1,2j-1,2h-1} \tag{3.66}
\end{align*}
\]
the field restriction is
\[ x_{i,j,k}^{2h} = x_{2i-1,2j-1,2k-1}, \tag{3.67} \]
and the prolongation becomes
\[ x_{i,j,k} = x_{h}^{2h}, \quad i = 1, 3, \ldots, j = 1, 3, \ldots, k = 1, 3, \ldots \tag{3.68} \]
with the remaining points found by trilinear interpolation.

**Smoothing and Solving**

The smoother for the multigrid method is typically a simple iterative scheme such as SOR, Gauss Seidel or Jacobi iteration, or an incomplete factorisation method such as SIP or MSI. At each recursive call of the multigrid solver the solution is smoothed by applying several iterations of the smoother to the solution. The smoother doesn’t have to be applied to convergence since the aim is not to solve the equations but rather to smooth the solution on the current mesh.

The solve operation in Figure 3.20 varies depending on what level of the mesh hierarchy the solver is on–for all but the coarsest mesh the solution to
\[ A^{2h} x^{2h} = b^{2h}, \tag{3.69} \]
is found by recursing and applying the multigrid solver to the system. At the coarsest mesh however the system is solved either by an iterative technique applied to convergence, or by using a direct method. Since this system is for the coarsest mesh the computational cost of its solution is minimal.

The test for overall convergence of the scheme is performed on the finest mesh, where the norm of the residual is calculated and compared with the user supplied solution tolerance. Once the norm reduces below the specified error bound the solver is assumed to have converged and the process is terminated.
3.2 A Comparison of the Solvers

The linear solvers were compared in terms of their speed, and memory usage. When comparing the speeds of the solvers several factors come into play. For direct solvers the number of operations is fixed for a given number of equations, but for iterative methods the time taken to converge to a solution depends not only on the number of equations but also on the properties of the equations themselves (such as the boundary conditions and diagonal dominance), and the convergence criteria and tolerance chosen.

The number of equations to be solved, the layout of the data in memory, and minor implementation details such as the syntax used to perform a matrix-vector operation also have a big effect on speed. These effects are discussed in the following chapter, but it is important to note that comparisons of different codes should be made for a range of array sizes and with a consistent coding style to reduce variability due to these factors.

In the following sections the test case used to compare the solvers is described, and then the solvers are compared on the basis of their convergence characteristics and scaling.

3.2.1 The Solver Test Case

To compare the speeds of the linear solvers they were used to solve two finite volume problems, one with Dirichlet and the other with Neumann boundary conditions, which simulate the equations encountered in a finite volume CFD code. The test cases were run for both two and three dimensional problems, and were solved to full convergence.

The test case was a finite volume discretisation of the Laplace equation applied to a unit square or cubic domain, with a sinusoidally varying source term. For the three dimensional case the equations were,

\[
\nabla^2 \phi = \sin(\pi x) \sin(2\pi y) \sin(2\pi z),
\]

with Dirichlet boundary conditions, and

\[
\nabla^2 \phi = \sin(2\pi x) \sin(2\pi y) \sin(2\pi z),
\]

with Neumann boundary conditions. For Neumann boundaries a zero normal gradient was applied to all boundaries, and for the Dirichlet problem the \(x = 0\) and \(x = 1\) boundaries were set to \(\phi = -1\) and \(\phi = 1\) respectively, with all other boundaries set to \(\phi = 0\). With the Neumann problem, the solution is not unique, and a further condition of

\[
\phi = 0
\]

was imposed at the centre of the solution domain to ensure uniqueness. For the two dimensional test cases the \(z\) forcing component of the source term was dropped. The solutions to the two dimensional forms of the test functions are shown in Figure 3.21.

For each equation two runs were made. The first compared the convergence of the iterative methods at one mesh size—a \(256^2\) mesh in two dimensions, and \(64^3\) mesh in three. The other run was made for a range of mesh sizes, with results obtained for the time to reduce the maximum residual by a factor of \(10^8\), the solution then being considered fully converged. The direct methods were also timed for the same range of array sizes. All runs were made from an initial field of \(\phi = 0\), and the relaxation parameters for the simple iterative and incomplete LU solvers were set to \(\omega = 1\) and \(\alpha = 0.9\) respectively.

The runs were made on a DEC Alpha 500au workstation running Digital Unix 4.0E, using Fortran 90 code compiled with the Digital Fortran compiler and using double precision storage for the floating point numbers.

---

3The Neumann boundary problem is an approximation of the pressure correction equation from a cavity flow problem, whilst the Dirichlet boundary problem approximates a scalar field from such a flow.
point data. Timings were made using the C `getrusage` and `gettimeofday` functions which provide accuracy to \(1/1000^{th}\) of a second, with multiple runs being made at the small array sizes to ensure an accurate resolution of the runtime.

One failing of the test problem (and one which the Author has not had time to remedy) is the square or cubic dimensions of the solution domains, and the unit aspect ratio of the mesh cells. Iterative methods can stall when the mesh is distorted from such a regular topology\[43\], an analysis of the mechanisms behind the failing being given by Brandt\[17\]. Unfortunately, timer prevented a thorough investigation of this phenomena, although Ferziger and Perić\[43\] claim it’s effect is less pronounced on convection–diffusion problems.
Figure 3.21: The 2D solutions to equation (3.70) (top) and (3.71) (bottom).
CHAPTER 3. LINEAR SOLVERS

3.2.2 Convergence of the Solvers

To compare the convergence rates of the iterative solvers, runs were made using $259^2$ and $67^3$ meshes for both the Neumann and Dirichlet boundary condition problems, with the residual at each iteration being printed out for plotting, the abscissa being plotted as solution time. Other studies of iterative solver have either studied only one or two classes of solver (such as Briggs[18], Yu[183]), or have made comparisons in terms of number of iterations (Ferziger and Peric[43], Iserles[68], Barrett et al[7], and Kershaw[77]). The speed of each iteration can vary widely between different solvers, and so comparisons in terms of iterations alone is rather meaningless. One study by Botta et al[14] makes a comparison of a number of different methods in terms of computation speed, but the rate of convergence is not given, and the coding of different solvers by different groups allows for a wide range of coding styles which may affect the relative solver speeds.

In general the iterative methods took longer to solve the problems with Neumann boundary conditions than those with Dirichlet boundaries. The system from the Neumann boundary problem is singular unless the solution is specified at some point in the solution domain. Most methods performed more efficiently if the system was left singular, with the exception of the direct Cholesky factorisation method which would fail with the singular system.

Figures 3.22 and 3.23 show the convergence of the solvers for the 2D test cases with Dirichlet and Neumann boundary conditions respectively. Figures 3.24 and 3.25 show the convergence for the 3D Dirichlet and Neumann test runs.

The Simple Iterative Methods

The Jacobi and RBSOR methods were typically the slowest to converge for the Dirichlet boundary problems (Figures 3.22 and 3.24), convergence patterns that are typical in the literature[43]. For large system SOR and SSOR had similar speeds to each other and were the fastest of the simple iterative schemes. All solvers exhibited a smooth monotonic reduction in the residual. It should be noted that for two dimensional problems the Block Tridiagonal direct method is faster to solve than these simple iterative schemes.

The Neumann problems were slower to solve than those with Dirichlet boundary conditions. For the Dirichlet boundary conditions the convergence exhibits an initial sharp drop in the maximum residual, followed by a much longer period where the relative residual drops by a constant ratio with each iteration. For the Neumann boundary conditions this initial sharp drop was not observed, and it was thought to be due to the rapid smoothing of the short wavelength structures adjacent to the boundaries in the Dirichlet problem.

Incomplete Factorisation Methods

The more complicated incomplete factorisation solvers tended to converge faster than their simpler counterparts, with MSI being the fastest to converge, and the incomplete Cholesky and incomplete LU methods being slowest. The ILU solver gave a similar convergence rate to the fastest of the simple iterative methods, whilst the MSI and SIP solvers give a much improved rate of convergence. All solvers exhibited a smooth monotonic reduction in residual.

Multigrid Schemes

For the Dirichlet problem, the multigrid solvers were not as fast as the simple and incomplete factorisation methods at reducing the residual for the first two of orders of magnitude. However they
exhibited vastly superior performance in generating a fully converged solution, being an order of magnitude faster than the best of the incomplete factorisation methods, and two orders of magnitude better than the simple iterative schemes. They exhibited a linear reduction in the residual. This behaviour is generally agreed upon in the literature; see [16], [55] and [49] for example.

Typically the SIP smoothed multigrid was fastest, being faster than the MSI smoother version, but in many 2D cases the SSOR smoothed multigrid had similar speed to the SIP smoothed scheme.

![Graphs showing convergence with time of 2D solvers on a $259^2$ mesh with Dirichlet boundary conditions.](image1)

![Graphs showing convergence with time of 2D solvers on a $259^2$ mesh with Dirichlet boundary conditions.](image2)

![Graphs showing convergence with time of 2D solvers on a $259^2$ mesh with Dirichlet boundary conditions.](image3)

![Graphs showing convergence with time of 2D solvers on a $259^2$ mesh with Dirichlet boundary conditions.](image4)

![Graphs showing convergence with time of 2D solvers on a $259^2$ mesh with Dirichlet boundary conditions.](image5)

Figure 3.22: Convergence with time of 2D solvers on a $259^2$ mesh with Dirichlet boundary conditions. Note the scale of the x axis varies from graph to graph.

Krylov Space Methods

The Krylov space methods all exhibited an irregular convergence pattern for both test problems, with a non-monotonic reduction in the residual.

- CG: The unpreconditioned CG solver showed an initial period of approximately linear convergence, after which the rate of convergence increased.
Preconditioning improved the smoothness of the convergence of the CG solver, with the incomplete factorisation smoothed multigrid preconditioners forcing a smooth linear reduction in the residual. The Jacobi smoothed multigrid preconditioner however gave a very poor performance for the Neumann boundary problem.

The fastest solvers in all cases were the multigrid-SIP and multigrid-ILU preconditioned methods, both giving very similar rates of convergence.

For all cases the Jacobi preconditioner reduced the number of iterations required to converge, but increased the overall run time.

- BiCGSTAB: The BiCGSTAB solvers typically had a more irregular rate of convergence than their CG counterparts. However experience of their use in a CFD code has proved them to be more robust than CG, not breaking down in cases that break down for a CG scheme. Otherwise they show the same overall convergence behaviour as the CG solvers, with the exception that the Jacobi smoothed multigrid preconditioner converged with a similar speed to the multigrid-incomplete factorisation schemes for the Neumann boundaries, unlike the case for the CG solver.
where it gave a much slower rate of convergence.

- GMRES: The GMRES solver typically displayed a monotonic convergence. The heavily preconditioned versions with multigrid or incomplete factorisation preconditioners converged with a similar rate to the CG or BiCGSTAB solvers, but the unpreconditioned solver had a very slow convergence rate, comparable to the simple iterative schemes.

**Summary**

The incomplete LU and simple iterative schemes all exhibited a smooth monotonic reduction in the residual, with the MSI and SIP schemes being up to an order of magnitude faster than the other methods. The multigrid schemes also exhibited a smooth reduction in residual. Whilst the incomplete LU smoothed multigrid schemes were the fastest methods to converge, the performance of the multigrid solver when using one of the simple iterative schemes as a smoother was still faster than the use of the best of the incomplete LU methods without multigrid acceleration. However, for the initial smoothing
of the Dirichlet boundary problems the incomplete factorisation methods give a faster initial reduction in the residual, behaviour that was thought to be due to their rapid smoothing of the short wavelength components in the residual at the boundary.

The Krylov space methods typically exhibited an irregular convergence, but when coupled with a multigrid preconditioner these solvers gave a smooth regular reduction in the residual. Generally the use of a preconditioner smoothed and speeded the convergence, with the exception of the Jacobi preconditioner, which whilst decreasing the number of iterations taken to converge to the solution, actually increased the overall solution time.

The multigrid-SIP and Krylov-multigrid-incomplete factorisation methods give the best performance, both exhibiting a smooth monotonic reduction in the residual, whilst being faster than the other methods.
3.2.3 Scaling of the Solvers

The previous section compared the convergence of the iterative solvers for a single mesh size. For a more general comparison of the solvers, the scaling of the speed of the solvers with mesh size is useful in predicting the performance of the methods for very large and very small problems.

The solvers were run for a range of mesh sizes, with a reduction of $10^6$ in the infinity norm of the residual ($\| r \|_\infty$) being used as a convergence tolerance. The solvers were run for the equations resulting from the Dirichlet and Neumann boundary condition problems (see Equations (3.70) and (3.71) and Figure 3.21), with both the 2D and 3D versions of the solvers being tested. The time for solution of the test codes is plotted as a function of mesh size in Figures 3.26 to 3.29, with Figures 3.26 and 3.27 showing the solution time for 2D problems with Dirichlet and Neumann boundary conditions, whilst Figures 3.28 and 3.29 show the solution times for the 3D problems.

Direct Solvers

These solvers run for the same length of time regardless of the boundary conditions – no pivoting was done so there was a set number and order of operations regardless of the equations being solved. With large systems of equations the time that the dense LU and LDL solvers took to solve had an $O(N^4)$ scaling, where $N$ was the number of equations. However, from an operation count one expects an $O(N^3)$ scaling, and no explanation is available for the scaling discrepancy.

For two dimensional problems the banded LU and Cholesky solvers were both $O(N^2)$ in run time. A jump in the solver runtime is clearly visible at $N = 1000$ and $N = 3000$ for the 2D banded LU and Cholesky solvers respectively. This occurs when the solver data exceeded the size of the cache of the test machine. The block tridiagonal solvers also exhibit $O(N^2)$ scaling, but run 15 and 4 times faster than the banded LU and Cholesky solvers respectively.

The three dimensional solvers exhibit slightly different behaviour to those solving two dimensional problems. As might be expected the dense LU and LDL solvers scale identically to the two dimensional versions. However the tridiagonal solvers scale like $O(N^3)$, as does the banded LU solver.

The direct methods were rarely faster than the iterative schemes. However, for the two dimensional test problems the block-tridiagonal solver was faster than the simple iterative schemes, and for large two dimensional systems ($N > 20000$) with Neumann boundaries it was faster than the incomplete factorisation schemes.

The Simple Iterative Methods

For the two dimensional problem the time taken by the simple iterative schemes to solve a system of equations scaled like $O(N^2)$, but for the three dimensional test cases they scaled like $O(N^{1.7})$. For all but the smallest systems the simple iterative methods were the slowest of the iterative methods.

Incomplete Factorisation Methods

For the two dimensional problems the solution time for the incomplete factorisation solvers scales like $O(N)$ for small numbers of equations, but as the numbers of equations increases they started to exhibit $O(N^2)$ behaviour. For the three dimensional problem the solvers exhibited $O(N^{1.7})$ behaviour.

For small to moderate sized problems ($N < 1000$ for Neumann problems, and $N < 10000$ for Dirichlet problems) the MSI solver was typically the fastest solver of all the methods tested.

---

4The effects of cache size are discussed in Chapter 6
Figure 3.26: The time taken to solve a two dimensional discretisation of the Laplace equation with Dirichlet boundary conditions.
Figure 3.27: The time taken to solve a two dimensional discretisation of the Laplace equation with Neumann boundary conditions.
Figure 3.28: The time taken to solve a three dimensional discretisation of the Laplace equation with Dirichlet boundary conditions.
Figure 3.29: The time taken to solve a three dimensional discretisation of the Laplace equation with Neumann boundary conditions.
CHAPTER 3. LINEAR SOLVERS

Multigrid Schemes

For the two dimensional test cases the time taken by the multigrid methods to solve the system of equations scaled like $O(N^{1.15})$ for the Dirichlet boundary condition problem, and $O(N^{1.3})$ for the problem with Neumann boundaries. With the three dimensional test case the scaling was $O(N^{1.15})$ for both boundary conditions.

For small problems the multigrid scheme are typically the slowest, for $N < 100$ being slower than the slowest of the direct methods. However their superior scaling means that for large problem sizes ($N > 20000$) the multigrid solvers are the fastest of the methods studied.

Krylov Space Methods

The Krylov space solvers are the fastest of the methods for moderate sized problems, with either the preconditioned conjugate gradient or BiCGSTAB solvers being fastest for $N \approx 10000$ when coupled with the MSI or SIP smoothed multigrid preconditioners. For larger problems the slightly superior scaling of the multigrid solvers ensures that they become faster. The multigrid-preconditioned Krylov space methods scaled better than methods that used other preconditioners, due to the better scaling of multigrid schemes as a whole.

- CG: For the two dimensional test case the time taken by the conjugate gradient method to solve the system of equations scaled between $O(N^{1.3})$ and $O(N^{1.6})$, with the multigrid preconditioned solvers scaling like $O(N^{1.3})$ and the other preconditioners and the unpreconditioned solver scaling $O(N^{1.6})$. For the Neumann boundaries the scaling was of order $O(N^{1.5})$ and $O(N^{1.6})$ for the multigrid-preconditioned and other solvers respectively.

For the three dimensional test cases the scaling was $O(N^{1.4})$ and $O(N^{1.2})$ for the general and multigrid preconditioned versions of the solver respectively.

- BiCGSTAB: For the two dimensional tests the scaling was $O(N^{1.6})$ and $O(N^{1.4})$ for the general and multigrid-preconditioned codes, and with the three dimensional problems the scaling was of $O(N^{1.4})$ and $O(N^{1.2})$.

- GMRES This solver exhibited similar scaling to its CG and BiCGSTAB brethren. Generally it was slower than those two solvers.

Summary

For small problems (where the number of equations were less than 5000) the MSI solver, an incomplete factorisation method, was typically the fastest method to solve the equations. For moderate sized problems (where the number of equations was of the order of 10000) the CG and BiCGSTAB Krylov space method were fastest when coupled with either the MSI or SIP preconditioners or the multigrid preconditioners that use MSI or SIP smoothing. For large systems (where the number of equations exceeded 30000) the multigrid solvers were the fastest.

The solution times for the multigrid methods scaled like $O(N^{1.15})$ for large sets of equations, unlike the other iterative schemes for which the solution time generally scaled between $O(N^{1.9})$ to $O(N^2)$. For preconditioned Krylov space methods the scaling was somewhere between the scaling of the Krylov method and that of it’s preconditioner.

The direct methods were typically the slowest of the methods tested. However it is noteworthy that for two dimensional systems with Neumann boundary conditions and more than 20000 equations the block tridiagonal solver was faster than the incomplete factorisation schemes such as MSI. The
solution time for the direct LU solver exhibited $O(N^4)$ scaling of the solution time, whilst the block tridiagonal solver had $O(N^2)$ and $O(N^3)$ scaling for the 2D and 3D cases respectively.

### 3.2.4 Memory Usage

The number of words of memory used by the solvers is shown in Figure 3.30. The storage shown includes the storage of the equations, solution, and right hand side ($A$, $x$ and $b$) which are $7N$ and $9N$ for the 2D and 3D finite volume discretisations used.

![Figure 3.30: The memory usage in words of the 2D (left) and 3D (right) linear solvers.](image)

The simple iterative schemes such as SOR used the least amount of memory, with SOR, RBSOR and SSOR not requiring any memory above the storage requirements of the equations, and Jacobi requiring only $N$ words of memory. The simplest of the incomplete factorisation methods also are efficient in memory use, with the Incomplete Cholesky and ILU methods both only requiring $2N$ words of memory above the storage of the equations. However the SIP and MSI solvers require more memory, using $6N$ and $8N$ words of memory for the 2D versions, and $8N$ and $14N$ in 3D.

The Krylov space methods similarly vary in their storage requirements. The unpreconditioned CG solver is the most efficient requiring $3N$ words of memory above the storage of the equations, with the unpreconditioned BiCGSTAB solver requiring $6N$ words of memory. The GMRES solver is comparatively greedy in its memory usage, requiring $(1 + N_T)N + N_T^2$ words of memory, where $N_T$ is the recurrence length of the solver. The recurrence length should be as large as possible, but the fact that memory is finite will set an upper limit available. For the data in Figure 3.30 a recurrence length of 50 was used. If a preconditioner is used the above figures must be increased to account for any storage required by the preconditioner.

The memory usage of the multigrid solvers depends both on the number of equations and also upon the number of grid levels used. With $n$ being the number of equations on a particular grid level, the 2D multigrid solver requires $9n$, $13n$ and $15n$ words of memory for that level for the Jacobi/ILU, SIP and MSI versions of the solver respectively. For the 3D solver the requirements are $11n$, $17n$ and $23n$. For each restriction from a fine to a coarse mesh the number of equations decreases by a factor of $2^2$ for the 2D solver and $2^3$ for the 3D solver, so the number of points on all meshes is given by the series $(1 + \frac{1}{2} + \frac{1}{2^2} + \ldots)N$ for the 2D solver and $(1 + \frac{1}{2} + \frac{1}{2^2} + \ldots)N$ for the 3D solver. Truncating both of these series at three terms gives an approximate estimate of the overall memory requirement for the solvers as $11.8N$, $17N$ and $19.7N$ for the Jacobi/ILU, SIP and MSI versions of the 2D multigrid solver, and $12.5N$, $19.4N$ and $26.2N$ for the Jacobi/ILU, SIP and MSI versions of the 3D solver.

Finally the direct methods have a much larger memory footprint than the iterative schemes. The naïve LU decomposition scheme uses $N^2$ words of memory, whilst the memory usage of the block tridiagonal methods depends on the relative axis lengths of the mesh upon which the finite volume equations have been discretised. For the worst case, a square or cubic mesh, the 2D block tridiagonal
solver uses $N^{3/2}$ words of memory, whilst the 3D solver uses $N^{5/3}$ words. If the meshes are much shorter along one axis however the memory usage is reduced.

3.3 Conclusions

A number of linear solvers suitable for the solution of elliptic PDEs have been described, and tested on the equations arising from a two and three dimensional finite volume discretisation of the Laplace equation. Comparisons of the solvers have been made in terms of the speed to solve the systems, and the memory used.

For small problems (where the number of equations is less than 5000) the MSI (Modified Strongly Implicit) solver, an incomplete factorisation method, is typically the fastest method to solve the equations. For moderate sized problems (where the number of equations is of the order of 10000) the CG (Conjugate Gradient) and BiCGSTAB (Bi-Conjugate Gradient Stabilised) Krylov space methods become fastest when coupled with either the MSI or SIP (Strongly Implicit Procedure) preconditioners or the multigrid preconditioners that use MSI or SIP smoothing.

For large systems (where the number of equations exceeds 20000) the multigrid solvers become the fastest. The solution times for the multigrid methods scale like $O(N^{1.15})$ for large sets of equations, unlike the other iterative schemes for which the solution time scales like $O(N^{1.5})$ to $O(N^2)$.

The direct methods were typically the slowest methods tested. However in one case (two dimensional systems with more than 20000 equations and Neumann boundary conditions) the block tridiagonal solver was faster than the simple iterative schemes such as SOR (Successive Over Relaxation) and the incomplete factorisation schemes such as MSI.

In terms of memory usage the most efficient solvers are the simple iterative methods such as SOR family of solvers. However none of the iterative methods are especially greedy in terms of memory use, with the notable exception of the GMRES (General Minimalised Residual) Krylov space solver which required the storage of successive iterates of the solution, and all required approximately $O(N)$ words of memory. The direct methods however use more memory than the iterative methods, with their memory requirement scaling as $O(N^2)$ for LU decomposition and up to $O(N^{3/2})$ or $O(N^{5/2})$ for the 2D and 3D block tridiagonal methods.