Mathematical Methods (10/24.539)

VI. Numerical Solution of Linear Algebraic Equations

Introduction

The solution of linear systems of algebraic equations is an important subject of linear algebra, and the computational considerations needed for computer implementation are usually treated in some detail in introductory numerical methods courses. We have already touched briefly upon this subject (see Section III -- Overview of Linear Algebra), and most students have had some introduction to computer implementation in other courses (numerical methods, heat transfer, etc.). Thus, this section of notes simply represents a quick review or overview of the subject - it is not intended as a complete treatise on this topic. Students with little or no background in this area are referred to one of many good numerical methods texts that treat the subject in more detail.

The numerical solution of large systems of linear algebraic equations is a direct consequence of the Finite Difference method (or Finite Element method -- see Appendix I) for solving ODEs or PDEs. As introduced in the last section on Finite Difference methods, recall that the goal in these techniques is to break the continuous differential equation into a coupled set of algebraic difference equations for each finite volume or node in the system. When one has only a single independent variable (the ODE case), this process can easily lead to several hundred simultaneous equations that need to be solved. For multiple independent variables (the PDE case), systems with hundreds of thousands of equations are common. Thus, in general, we need to be able to solve large systems of linear equations of the form $\underline{\underline{A}}\underline{x} = \underline{b}$ as part of the solution algorithm for general Finite Difference or Finite Element methods.

There are two general schemes for solving linear systems: ***Direct Elimination Methods*** and ***Iterative Methods***. All the direct methods are, in some sense, based on the standard Gauss Elimination technique, which systematically applies row operations to transform the original system of equations into a form that is easier to solve. In particular, this section of notes overviews an algorithm for implementation of the basic Gauss Elimination scheme and it also highlights the LU Decomposition method which, although functionally equivalent to the Gauss Elimination method, does provide some additional flexibility for computer implementation. Thus, the LU decomposition method is often the preferred direct solution method for low to medium sized systems (usually less than 1000 equations or so).

For very large systems, iterative methods (instead of direct elimination methods) are almost always used. This switch is required from accuracy considerations (related to round-off errors), from memory limitations for physical storage of the equation constants, from considerations for treating nonlinear problems, and from overall efficiency concerns. There are several specific iterative schemes that are in common use, but most methods build upon the base Gauss Seidel method, usually with some acceleration scheme to help convergence. Thus, our focus in this brief overview is on the basic Gauss Seidel scheme and on the use of Successive Relaxation (SR) to help accelerate convergence.

Our brief overview of direct and iterative schemes for the computer solution of large systems of linear equations is broken into the following subsections:

Gauss Elimination Method

- Basic Algorithm
- Example 6.1 -- **Illustration of the Gauss Elimination Method**

LU Decomposition Method

- Basic Algorithm
- Example 6.2 -- **Illustration of the LU Decomposition Method**

Iterative Methods

- General Notation
- Gauss Seidel Method
- Successive Relaxation
- Example 6.3 -- **Analysis of the SR Method for a Simple 3x3 System**

The subject of direct solution schemes for linear equations is also treated in my undergraduate introductory numerical methods course. Thus, you may also want to check out the Matlab examples for my ***Applied Problem Solving with Matlab*** course at the following URL: www.profjrwhite.com/courses.htm.

Gauss Elimination Method

Basic Algorithm

The Gauss Elimination Method forms the basis for all elimination techniques. The basic idea is to modify the original equations, using legal row operations, to give a simpler form for actual solution. The basic algorithm can be broken into two stages:

1. Forward Elimination (put equations in upper triangular form)
2. Back Substitution (solve for unknown solution vector)

To see how this works, consider the following system of equations:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots a_{1N}x_N &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots a_{2N}x_N &= b_2 \\ &\vdots \\ a_{N1}x_1 + a_{N2}x_2 + \cdots a_{NN}x_N &= b_N \end{aligned} \qquad (6.1)$$

Now, with reference to this system of N equations and N unknowns, the **Forward Elimination Step** (with partial pivoting) becomes:

```
Step 0 -- Create an augmented matrix, $\tilde{\underline{\underline{A}}} = [\underline{\underline{A}} \quad \underline{b}]$
Step 1 -- Determine the coefficient in the i^th column with the largest absolute value and
          interchange rows such that this element is the pivot element (i = 1, 2, 3, to N-1)
Step 2 -- Normalize the pivot equation (i.e. divide by the i,i element)
    Step 3 -- Multiply normalized eqn. i by the j,i element of eqn. j
    Step 4 -- Subtract the resultant equation in Step 3 from eqn. j
    repeat Steps 3 and 4 for j = i+1 to N
go to Step 1 for next i = i+1 to N-1
```

and the **Back Substitution Step** is given by:

```
Step 5 -- $x_N = b'_N / a'_{NN}$
Step 6 -- $x_i = \left( b'_i - \sum_{j=i+1}^{N} a'_{ij} x_j \right) \Big/ a'_{ii}$
repeat for i = N-1, N-2, to 1
```

where the primes indicate that the coefficients at this stage are different from the original coefficients.

Example 6.1 illustrates this algorithm for a simple 3x3 system.

Example 6.1 -- Illustration of the Gauss Elimination Method

Problem Description:

Given the following system:

$$\underline{\underline{A}}\underline{x} = \underline{b} \quad \text{with} \quad \underline{\underline{A}} = \begin{bmatrix} 0 & -8 & -2 \\ 6 & 4 & 0 \\ 8 & 0 & -6 \end{bmatrix} \quad \text{and} \quad \underline{b} = \begin{bmatrix} -1 \\ 4 \\ 7 \end{bmatrix}$$

Determine the solution vector using the Gauss Elimination Method.

Problem Solution:

The various steps in the Gauss Elimination Method are as follows:

Step 0 -- form augmented matrix

$$\begin{bmatrix} 0 & -8 & -2 & -1 \\ 6 & 4 & 0 & 4 \\ 8 & 0 & -6 & 7 \end{bmatrix}$$

Step 1 -- i = 1 perform partial pivoting

$$\begin{bmatrix} 8 & 0 & -6 & 7 \\ 6 & 4 & 0 & 4 \\ 0 & -8 & -2 & -1 \end{bmatrix}$$

Step 2 -- i = 1 normalize pivot equation

$$\begin{bmatrix} 1 & 0 & -6/8 & 7/8 \\ 6 & 4 & 0 & 4 \\ 0 & -8 & -2 & -1 \end{bmatrix}$$

Step 3+4 -- multiply eqn. i = 1 by 6 and subtract from eqn. j = i+1 = 2

$$\begin{bmatrix} 1 & 0 & -6/8 & 7/8 \\ 0 & 4 & 36/8 & -10/8 \\ 0 & -8 & -2 & -1 \end{bmatrix}$$

Step 3+4 -- multiply eqn. i = 1 by 0 and subtract from eqn. j = i+2 = 3

$$\begin{bmatrix} 1 & 0 & -6/8 & 7/8 \\ 0 & 4 & 36/8 & -10/8 \\ 0 & -8 & -2 & -1 \end{bmatrix} \qquad \text{(no change)}$$

Step 1 -- i = 2 perform partial pivoting

$$\begin{bmatrix} 1 & 0 & -6/8 & 7/8 \\ 0 & -8 & -2 & -1 \\ 0 & 4 & 36/8 & -10/8 \end{bmatrix}$$

Step 2 -- i = 2 normalize pivot equation

$$\begin{bmatrix} 1 & 0 & -6/8 & 7/8 \\ 0 & 1 & 2/8 & 1/8 \\ 0 & 4 & 36/8 & -10/8 \end{bmatrix}$$

Step 3+4 -- multiply eqn. i = 2 by 4 and subtract from eqn. j = i+1 = 3

$$\begin{bmatrix} 1 & 0 & -6/8 & 7/8 \\ 0 & 1 & 2/8 & 1/8 \\ 0 & 0 & 28/8 & -14/8 \end{bmatrix}$$

Stop elimination phase since i = 2 = N-1 = 2

Step 5 – evaluate the last element of the vector

$$x_3 = -\frac{1}{2}$$

Step 6 – evaluate all other elements (in reverse order)

$$x_2 = \frac{1}{8} - \frac{2}{8}\left(-\frac{1}{2}\right) = \frac{1}{4}$$

$$x_1 = \frac{7}{8} - 0\left(\frac{1}{4}\right) + \frac{6}{8}\left(-\frac{1}{2}\right) = \frac{1}{2}$$

Therefore the final result is $\underline{x} = \begin{bmatrix} 1/2 & 1/4 & -1/2 \end{bmatrix}^T$.

The LU Decomposition Method

Basic Algorithm

The Gauss Elimination Method has the disadvantage that all right-hand sides (i.e. all the $\underline{b}$ vectors of interest for a given problem) must be known in advance for the elimination step to proceed. The LU Decomposition Method outlined here has the property that the matrix modification (or decomposition) step can be performed independent of the right hand side vector. This feature is quite useful in practice - therefore, the LU Decomposition Method is usually the Direct Scheme of choice in most applications.

To develop the basic method, let's break the coefficient matrix into a product of two matrices,

$$\underline{\underline{A}} = \underline{\underline{L}}\,\underline{\underline{U}} \tag{6.2}$$

where $\underline{\underline{L}}$ is a lower triangular matrix and $\underline{\underline{U}}$ is an upper triangular matrix.

Now, the original system of equations,

$$\underline{\underline{A}}\,\underline{x} = \underline{b} \tag{6.3}$$

becomes

$$\underline{\underline{L}}\,\underline{\underline{U}}\,\underline{x} = \underline{b} \tag{6.4}$$

This expression can be broken into two problems,

$$\underline{\underline{L}}\,\underline{y} = \underline{b} \qquad \text{and} \qquad \underline{\underline{U}}\,\underline{x} = \underline{y} \tag{6.5}$$

The rationale behind this approach is that the two systems given in eqn. (6.5) are both easy to solve; one by forward substitution and the other by back substitution. In particular, because $\underline{\underline{L}}$ is a lower triangular matrix, the expression, $\underline{\underline{L}}\,\underline{y} = \underline{b}$, can be solved with a forward substitution step. Similarly, since $\underline{\underline{U}}$ has upper triangular form, $\underline{\underline{U}}\,\underline{x} = \underline{y}$ can be evaluated with a simple back substitution algorithm.

Thus the key to this method is the ability to find two matrices, $\underline{\underline{L}}$ and $\underline{\underline{U}}$, that satisfy eqn. (6.2). Doing this is referred to as the Decomposition Step and there are a variety of algorithms available. Three specific approaches are as follows:

Doolittle Decomposition (4x4 example):

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ \ell_{21} & 1 & 0 & 0 \\ \ell_{31} & \ell_{32} & 1 & 0 \\ \ell_{41} & \ell_{42} & \ell_{43} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \tag{6.6}$$

Because of the specific structure of the matrices, a systematic set of formulae for the components of $\underline{\underline{L}}$ and $\underline{\underline{U}}$ results.

For example, a simple scheme (i.e. no partial pivoting) for computing $\underline{\underline{L}}$ and $\underline{\underline{U}}$ via Doolittle Decomposition directly within the storage space provided by the $\underline{\underline{A}}$ matrix is given below. Note that this algorithm does not explicitly store the diagonal elements of $\underline{\underline{L}}$ (since they are known), so both $\underline{\underline{L}}$ and $\underline{\underline{U}}$ can fit within the space provided by the original matrix (saves on storage):

```
for i = 1:N-1
  for k = i+1:N
    $a_{ki} = a_{ki}/a_{ii}$
    for j = i+1:N
      $a_{kj} = a_{kj} - a_{ki}a_{ij}$
    end
  end
end
```

Example 6.2 illustrates the details of the above algorithm and that the resultant $\underline{\underline{L}}$ and $\underline{\underline{U}}$ matrices do indeed allow easy solution of a system of linear equations written in the matrix-vector form given by eqn. (6.3).

Crout Decomposition (4x4 example):

$$\begin{bmatrix} \ell_{11} & 0 & 0 & 0 \\ \ell_{21} & \ell_{22} & 0 & 0 \\ \ell_{31} & \ell_{32} & \ell_{33} & 0 \\ \ell_{41} & \ell_{42} & \ell_{43} & \ell_{44} \end{bmatrix} \begin{bmatrix} 1 & u_{12} & u_{13} & u_{14} \\ 0 & 1 & u_{23} & u_{24} \\ 0 & 0 & 1 & u_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \tag{6.7}$$

The evaluation of the components of $\underline{\underline{L}}$ and $\underline{\underline{U}}$ is done in a similar fashion as above.

Cholesky Factorization:

For symmetric, positive definite matrices, where

$$\underline{\underline{A}} = \underline{\underline{A}}^T \quad \text{and} \quad \underline{x}^T \underline{\underline{A}}\underline{x} > 0 \quad \text{for all } \underline{x} \neq 0 \tag{6.8}$$

then,

$$\underline{\underline{U}} = \underline{\underline{L}}^T \quad \text{and} \quad \underline{\underline{A}} = \underline{\underline{L}}\underline{\underline{L}}^T$$

and a simple set of expressions for the elements of $\underline{\underline{L}}$ can be obtained (as above).

Other factorizations of this type are also possible (see a good numerical methods text). Once the elements of $\underline{\underline{L}}$ and $\underline{\underline{U}}$ are available (usually stored in a single N×N matrix as done in Example 6.2), the solution step for the unknown vector, $\underline{x}$, is a simple process [as outlined above in eqn. (6.5)]. Matlab's standard equation solver (using the backslash notation, **x = A\b**) uses several variants of the basic LU Decomposition method depending on the form of the original coefficient matrix (see the Matlab help files for details).

Example 6.2 -- Illustration of the LU Decomposition Method

Problem Description:

Given the following system:

$$\underline{\underline{A}}\underline{x} = \underline{b} \quad \text{with} \quad \underline{\underline{A}} = \begin{bmatrix} 8 & 0 & -6 \\ 0 & -8 & -2 \\ 6 & 4 & 0 \end{bmatrix} \quad \text{and} \quad \underline{b} = \begin{bmatrix} 7 \\ -1 \\ 4 \end{bmatrix}$$

Determine the solution vector using LU Decomposition (with Doolittle Decomposition).

Note: This is the same system of equations as used in Example 6.1 except that the rows have been interchanged. This was done so that no partial pivoting would be necessary as part of the formal solution scheme. In general, however, partial pivoting must be implemented as an integral part of any solution scheme, including the LU Decomposition method. Here, we want to keep this example as simple as possible to highlight the computation of the $\underline{\underline{L}}$ and $\underline{\underline{U}}$ matrices (in place).

Problem Solution:

Following the above algorithm for the decomposition step, we have the following sequence of computations:

i = 1 and k = 2: $a_{21} = a_{21}/a_{11} = 0/8 = 0$

then for j = 2: $a_{22} = a_{22} - a_{21}a_{12} = -8 - (0)(0) = -8$

j = 3: $a_{23} = a_{23} - a_{21}a_{13} = -2 - (0)(-6) = -2$

i = 1 and k = 3: $a_{31} = a_{31}/a_{11} = 6/8 = 3/4$

then for j = 2: $a_{32} = a_{32} - a_{31}a_{12} = 4 - (3/4)(0) = 4$

j = 3: $a_{33} = a_{33} - a_{31}a_{13} = 0 - (3/4)(-6) = 9/2$

i = 2 and k = 3: $a_{32} = a_{32}/a_{22} = 4/(-8) = -1/2$

then for j = 3: $a_{33} = a_{33} - a_{32}a_{23} = 9/2 - (-1/2)(-2) = 7/2$

Putting all these values into the "new" $\underline{\underline{A}}$ matrix gives

$$\underline{\underline{A}}_{\text{new}} = \begin{bmatrix} 8 & 0 & -6 \\ 0 & -8 & -2 \\ 3/4 & -1/2 & 7/2 \end{bmatrix}$$

where we see that row 1 is unchanged as dictated by eqn. (6.6).

Now, recalling that the diagonal of $\underline{\underline{L}}$ contains all ones, we can easily break this into the desired $\underline{\underline{L}}$ and $\underline{\underline{U}}$ matrices, or

$$\underline{\underline{L}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 3/4 & -1/2 & 1 \end{bmatrix} \qquad \text{and} \qquad \underline{\underline{U}} = \begin{bmatrix} 8 & 0 & -6 \\ 0 & -8 & -2 \\ 0 & 0 & 7/2 \end{bmatrix}$$

And, of course, $\underline{\underline{L}}$ times $\underline{\underline{U}}$ should give back the original $\underline{\underline{A}}$ matrix, or

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 3/4 & -1/2 & 1 \end{bmatrix} \begin{bmatrix} 8 & 0 & -6 \\ 0 & -8 & -2 \\ 0 & 0 & 7/2 \end{bmatrix} = \begin{bmatrix} 8 & 0 & -6 \\ 0 & -8 & -2 \\ 6 & 4 & 0 \end{bmatrix}$$

As a final step, two simple forward and back substitution sequences will solve the original system of equation. Doing these explicitly gives

$$\underline{\underline{L}}\underline{y} = \underline{b} \quad \Rightarrow \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 3/4 & -1/2 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 7 \\ -1 \\ 4 \end{bmatrix}$$

which leads to

$y_1 = 7$, $\quad y_2 = -1$, $\quad$ and $\quad y_3 = 4 - (3/4)(7) - (-1/2)(-1) = -7/4$

and

$$\underline{\underline{U}}\underline{x} = \underline{y} \quad \Rightarrow \quad \begin{bmatrix} 8 & 0 & -6 \\ 0 & -8 & -2 \\ 0 & 0 & 7/2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 7 \\ -1 \\ -7/4 \end{bmatrix}$$

which gives

$x_3 = (-7/4)/(7/2) = -1/2$

$x_2 = [-1 - (-2)(-1/2)]/(-8) = 1/4$

and $\quad x_1 = [7 - (0)(1/4) - (-6)(-1/2)]/(8) = 1/2$

Thus, we see that the final answer, $\underline{x} = \begin{bmatrix} 1/2 & 1/4 & -1/2 \end{bmatrix}^T$, is the same as that obtained in Example 6.1. Note that, as expected, the ***a priori*** row interchange did not affect the solution!!!

Iterative Methods

General Notation

For large systems of equations, an iterative solution scheme for the unknown vector can always be written in the form

$$\underline{x}^{p+1} = \underline{\underline{B}}\,\underline{x}^{p} + \underline{c} \qquad (6.9)$$

where $\underline{\underline{B}}$ is the iteration matrix, $\underline{c}$ is a constant vector and p is an iteration counter. Convergence of this scheme is guaranteed if the largest eigenvalue of the iteration matrix is less that unity, where

$$\rho = \text{spectral radius} = |\lambda_{\max}|$$

Therefore, if $\rho < 1$ the iterative scheme will converge. If $\rho << 1$, the iterative scheme converges very rapidly. If $\rho \approx 1$ but less than unity, the scheme will be slowly converging. The iteration algorithm will diverge if the spectral radius is greater than unity. Note that ρ is not normally known for large systems, but the above convergence criterion is useful for formal analyses and for the evaluation of the utility of various iterative schemes on model problems (relatively small systems).

In most systems, convergence is tested during the iterative process by computing the largest relative change from one iteration to the next, and comparing the absolute value of this result with some desired tolerance. If the maximum relative change is less than the desired accuracy, then the process is terminated. If this condition is not satisfied, then another iteration is performed. If the relative change continually increases from iteration to iteration, then the process is diverging and some alternate method or a change in formulation is needed.

The Gauss Seidel Method

Let's take the original system of equations given by $\underline{\underline{A}}\,\underline{x} = \underline{b}$ and convert it into the classical Gauss Seidel iterative scheme. To do this, let's break the original matrix into three specific components, or

$$\underline{\underline{A}} = \underline{\underline{L}} + \underline{\underline{D}} + \underline{\underline{U}} \qquad (6.10)$$

where the three matrices on the right hand side, in sequence, are strictly lower triangular, diagonal, and strictly upper triangular matrices. Note that the primary non-zero elements of $\underline{\underline{L}}$, $\underline{\underline{D}}$, and $\underline{\underline{U}}$ correspond to the original elements of $\underline{\underline{A}}$.

Now, substituting eqn. (6.10) into the original balance expression gives

$$\left(\underline{\underline{L}} + \underline{\underline{D}}\right)\underline{x} + \underline{\underline{U}}\,\underline{x} = \underline{b} \qquad (6.11)$$

or

$$\left(\underline{\underline{L}} + \underline{\underline{D}}\right)\underline{x} = \underline{b} - \underline{\underline{U}}\,\underline{x} \qquad (6.12)$$

If we premultiply by $\left(\underline{\underline{L}}+\underline{\underline{D}}\right)^{-1}$ and notice that the solution vector appears on both sides of the equation, we can write the equation in an iterative form as

$$\underline{x}^{p+1} = -\left(\underline{\underline{L}}+\underline{\underline{D}}\right)^{-1}\underline{\underline{U}}\underline{x}^{p} + \left(\underline{\underline{L}}+\underline{\underline{D}}\right)^{-1}\underline{b} \tag{6.13}$$

Clearly this is in the standard form for iterative solutions as defined in eqn. (6.9), where the iteration matrix is given by

$$\underline{\underline{B}} = -\left(\underline{\underline{L}}+\underline{\underline{D}}\right)^{-1}\underline{\underline{U}} \tag{6.14}$$

and the constant vector is written as

$$\underline{c} = \left(\underline{\underline{L}}+\underline{\underline{D}}\right)^{-1}\underline{b} \tag{6.15}$$

This form of the iteration strategy is useful for the study of the convergence properties of model problems. It is, however, not particularly useful as a program algorithm for code implementation.

For actual implementation on the computer, one writes these equations differently, never having to formally take the inverse as indicated above. In practice, eqn. (6.12) is written in iterative form as

$$\underline{\underline{D}}\underline{x}^{p+1} = \underline{b} - \underline{\underline{L}}\underline{x}^{p+1} - \underline{\underline{U}}\underline{x}^{p}$$

or

$$\underline{x}^{p+1} = \underline{\underline{D}}^{-1}\left(\underline{b} - \underline{\underline{L}}\underline{x}^{p+1} - \underline{\underline{U}}\underline{x}^{p}\right) \tag{6.16}$$

This specific form is somewhat odd at first glance, since $\underline{x}^{p+1}$ appears on both sides of the equation. This is justified because of the special form of the strictly lower triangular matrix, $\underline{\underline{L}}$. This can be seen more clearly if the matrix equations are written using discrete notation.

In discrete form eqn. (6.16) can be expanded as

$$x_i^{p+1} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{p+1} - \sum_{j=i+1}^{N} a_{ij}x_j^{p}\right) \tag{6.17}$$

where the diagonal elements of $\underline{\underline{D}}^{-1}$ are simply $1/a_{ii}$ and the limits associated with the summations account for the special structure of the $\underline{\underline{L}}$ and $\underline{\underline{U}}$ matrices.

As a specific example, consider the 3x3 system shown below:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

The iterative equations for this case can be written individually as

$$x_1^{p+1} = \frac{1}{a_{11}}\left(b_1 - a_{12}x_2^p - a_{13}x_3^p\right)$$

$$x_2^{p+1} = \frac{1}{a_{22}}\left(b_2 - a_{21}x_1^{p+1} - a_{23}x_3^p\right)$$

$$x_3^{p+1} = \frac{1}{a_{33}}\left(b_3 - a_{31}x_1^{p+1} - a_{32}x_2^{p+1}\right)$$

Thus, if the equations are taken in sequence, all the terms on the right hand side are known, thereby allowing computation of a new estimate for the full solution vector in terms of the very latest information available for the calculation. This is the basic idea behind the Gauss Seidel method.

The Successive Relaxation (SR) Method

To improve the rate of convergence, one might consider using a weighted average of the results of the two most recent estimates to obtain the next best guess of the solution. If the solution is converging, this might help extrapolate to the real solution more quickly. This idea is the basis of the SR method.

In particular, let α be some weight factor with a value between 0 and 2. Now, let's compute the next value of $\underline{x}^{p+1}$ to use in the Gauss Seidel method as a linear combination of the current value, $\underline{x}^{p+1}$, and the previous solution, $\underline{x}^p$, as follows:

$$\underline{x}^{p+1}\Big|_{\text{new}} = \alpha\underline{x}^{p+1} + (1-\alpha)\underline{x}^p \quad \text{with} \quad 0 < \alpha < 2 \tag{6.18}$$

Note that if α is unity, we simply get the standard Gauss Seidel method (or whatever base iterative scheme is in use). When α is greater that unity, the system is said to be over-relaxed, indicating that the latest value, $\underline{x}^{p+1}$, is being weighted more heavily (weight for $\underline{x}^p$ is negative). If, however, α is less than one, the system is under-relaxed, this time indicating that the previous solution, $\underline{x}^p$, is more heavily weighted (positive weight values).

Note that the formal iteration process for the SR method is given by

$$\underline{x}^{p+1} = \left(\alpha\underline{\underline{L}} + \underline{\underline{D}}\right)^{-1}\left((1-\alpha)\underline{\underline{D}} - \alpha\underline{\underline{U}}\right)\underline{x}^p + \left(\alpha\underline{\underline{L}} + \underline{\underline{D}}\right)^{-1}\alpha\underline{b} \tag{6.19}$$

which is in the standard form for iterative solutions as defined in eqn. (6.9). Thus, we see that the iteration matrix for the SR method is given by

$$\underline{\underline{B}} = \left(\alpha\underline{\underline{L}} + \underline{\underline{D}}\right)^{-1}\left((1-\alpha)\underline{\underline{D}} - \alpha\underline{\underline{U}}\right) \tag{6.20}$$

Thus, eqn. (6.20) would be used in a formal study involving the spectral radius and the convergence properties of the SR method.

The idea, of course, is to choose the relaxation parameter to improve convergence (reduce the spectral radius). This is illustrated for a simple problem in Example 6.3. In large problems an optimum choice for α is most often estimated in a trial and error fashion for certain classes of problems (experience helps here). Some more advanced codes do try to estimate this quantity as part of the iterative calculation, although this is not particularly easy.

A variety of schemes for improving convergence have been developed over the years, with many taking advantage of the particular structure of the algebraic equations or some characteristic of the physical system under study. The specifics of these algorithms are beyond the scope of this course, but the interested student is encouraged to see a good text on advanced numerical methods for further details as desired.

Example 6.3 - Analysis of the SR Method for a Simple 3x3 System

Problem Description:

Solve the following 3x3 system using the Successive Relaxation (SR) method with several values of relaxation parameter, α. Plot and analyze the convergence rate -- in terms of the number of iterations needed for convergence (called k) -- as a function of α. Use a tolerance of 10^{-5}.

Also formally compute the spectral radius, ρ, using Matlab's ***eig*** function to determine the eigenvalues of the iteration matrix given by eqn. (6.20). Also plot ρ vs. α and discuss the relative rate of convergence as a function of the spectral radius, k vs. ρ. Do your analyses for the following system:

$$\underline{\underline{A}}\underline{x} = \underline{b} \quad \text{with} \quad \underline{\underline{A}} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 9 & 6 \\ 3 & 6 & 27 \end{bmatrix} \quad \text{and} \quad \underline{b} = \begin{bmatrix} 2 \\ -4 \\ 24 \end{bmatrix}$$

Problem Solution:

The first step here is to implement a basic SR scheme into a Matlab function file. This was done within routine **sr.m**, which is listed in Table 6.1. This routine is a simple implementation (i.e. no partial pivoting) of eqns. (6.17) and (6.18). Because no row interchanges are made internal to the routine, the original $\underline{\underline{A}}$ matrix cannot have any zeros along the diagonal (the $1/a_{ii}$ term would cause a divide by zero). Thus, a check is made and an appropriate warning message is given if needed. Note also that explicit variables for $\underline{x}^{P+1}$ and $\underline{x}^{P}$ are not retained, and the form, $\underline{x} = \underline{x} + \cdots$, is used to update the current estimate of $\underline{x}$ based on all the best values available at the time. Also, since two complete iterates of $\underline{x}$ are not available, we have chosen to evaluate convergence based on the maximum residual, r_{max}, on iteration p, which is defined as

$$r_{max}^{p} = \max|\underline{r}^{p}| \qquad \text{where} \qquad \underline{r}^{p} = \underline{b} - \underline{\underline{A}}\underline{x}^{p}$$

In **sr.m**, when $r_{max} < tol = 10^{-5}$, we say the problem has converged!

Now, to perform the desired analyses, we start by comparing the solution given by Matlab's backslash operator with the use of our **sr.m** function for $\alpha = 1$ (i.e. the standard Gauss Seidel method). At the Matlab prompt, we have

```
>> clear all, format compact
>> A = [1 2 3;2 9 6;3 6 27];  b = [2 -4 24]';
>> x1 = A\b
x1 =
    2.2000
   -1.6000
    1.0000
>> [x2,k] = sr(A,b,[0 0 0]',1.0,1.0e-5,1000)
x2 =
    2.2000
   -1.6000
    1.0000
k =
    18
```

and we see that the **sr.m** routine gives the correct answer (to within the convergence criterion) in 18 iterations. Thus, we have benchmarked the base methodology!

Now, to do the desired analyses, a series of computations are required and they have been implemented within Matlab script file **sr_demo1.m --** see Table 6.2 for a listing. The program systematically solves the above 3x3 system for several values of α in the range $0.8 < \alpha < 1.6$ and stores the number of iterations required for convergence. In addition, for each value of α, an explicit value for the spectral radius is computed. Note that this task is straightforward conceptually, but it is very computationally intensive for large systems. However, for a simple 3x3 system, this is a rather trivial, but very informative, computation. Finally, with the stored values of k and ρ for several values of the relaxation parameter, α, we can create a series of plots -- k vs. α, ρ vs. α, and k vs. ρ -- and these are summarized within three subplots in Fig. 6.1.

As apparent, we see that the optimum relaxation parameter for this problem is near $\alpha = 1.1$. At this point, it takes about 14 iterations to converge (recall that the base Gauss Seidel method with $\alpha = 1.0$ took 18 iterations). We also see that, as expected, the spectral radius has a minimum at the optimum α. Finally, in the upper subplot, we see that there is a direct relationship between convergence rate and spectral radius -- with increased convergence rate (smaller # of iterations) with decreasing ρ. This is all quite consistent with expected trends, and this example shows quite vividly the usual behavior associated with iterative methods for solving linear equation. Unfortunately, however, most realistic analyses have many thousands of equations and the spectral radius is usually much closer to unity -- which leads to much slower convergence rates and significantly increased computational times. Thus, finding the optimum relaxation is usually quite advantageous in larger more realistic applications.

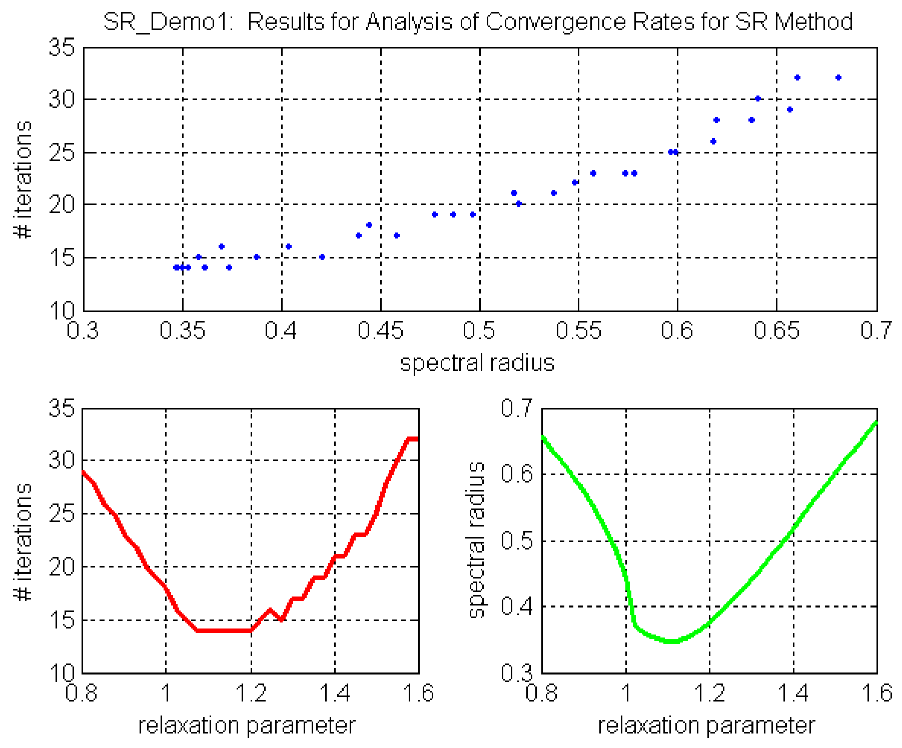

Fig. 6.1 Summary results for Example 6.3.

Table 6.1 Listing of the sr.m function file.

```
%
%   SR.M   Function to implement the Successive Relaxation (SR) Method
%
%   This routine implements a simple version of the SR method -- that is,
%   no partial pivoting is performed.  Thus, the user must be sure to arrange
%   the equations appropriately so that the diagonal elements do not have any zeros.
%
%   Inputs:  A     = n by n coefficient matrix (nonzero diagonal elements)
%            b     = n by 1 right-hand side vector
%            x     = n by 1 vector containing initial guess
%            alpha = relaxation parameter (alpha > 0)
%            tol   = error tolerance used to terminate search
%            M     = maximum number of iterations
%
%   Outputs: x = n by 1 solution vector
%            k = number of iterations performed
%
%   Note:  This file is a modified version of a similar routine from the
%   text "Applied Numerical Methods for Engineers Using Matlab and C," by
%   Schilling and Harris, Brooks Cole Publishing (2000).
%
%   File generated by J. R. White, UMass-Lowell (Aug. 2003)
%
      function [x,k] = sr(A,b,x,alpha,tol,M)
%
%   set some iteration parameters
      k = 0;   rmax = 1;
%
%   check for zeros along diagonal
      n = length(x);
      for i = 1:n
        if abs(A(i,i)) < eps
          disp('   WARNING:  Check A matrix for a zero along the diagonal!!!')
          disp(' ')
          return;
        end
      end
%
%   perform iteration
      while (k < M & rmax >= tol)
        for i = 1:n;
          d = A(i,i);
          x(i) = (1 - alpha)*x(i) + alpha*b(i)/d;
          for j = 1:n
            if j ~= i
              x(i) = x(i) - alpha*A(i,j)*x(j)/d;
            end
          end
        end
        rmax = max(abs(b - A*x));
        k = k + 1;
      end
%
%   end of function
```

Table 6.2 Matlab script file to solve Example 6.3.

```
%
%   SR_DEMO1.M   Analyze SR Method for a Simple 3x3 System
%
%   This program solves a system of equations using the successive
%   relaxation (SR) method for a range of relaxation parameters.  It also
%   computes the spectral radius for each case.  The idea here is to
%   illustrate several key points concerning iterative methods for the
%   solution of linear equations.
%
%   This script file calls the SR.M routine to apply the SR method.
%
%   File written by J. R. White, UMass-Lowell (Aug. 2003)
%

%
%   getting started
      clear all,  close all,  nfig = 0;
%
%   define system of interest and vector of relaxation parameters
      A = [1 2 3;2 9 6;3 6 27];  b = [2 -4 24]';
      alf = 0.8:0.025:1.6;   Nalf = length(alf);
%
%   solve system for range of alf and save k = # of iterations to convergence
      tol = 1e-5;   M = 1000;   k = zeros(size(b));
      for j = 1:Nalf
        xo = zeros(size(b));  [x,k(j)] = sr(A,b,xo,alf(j),tol,M);
      end
%
%   now compute the spectral radius
      L = tril(A,-1);  D = diag(diag(A));  U = triu(A,1);
      p = zeros(size(b));
      for j = 1:Nalf
        B = inv(alf(j)*L + D)*((1-alf(j))*D - alf(j)*U);
        v = eig(B);   p(j) = max(abs(v));
      end
%
%   plot results
      nfig = nfig+1;   figure(nfig)
      subplot(2,1,1), plot(p,k,'b.','LineWidth',2),grid
      xlabel('spectral radius'),ylabel('# iterations')
      title('SR\_Demo1:  Results for Analysis of Convergence Rates for SR Method')
      subplot(2,2,3), plot(alf,k,'r-','LineWidth',2),grid
      xlabel('relaxation parameter'),ylabel('# iterations')
      subplot(2,2,4), plot(alf,p,'g-','LineWidth',2),grid
      xlabel('relaxation parameter'),ylabel('spectral radius')
%
%   end of program
```