

## Applied Engineering Problem Solving (CHEN.3170)

### Part II: Numerical Methods and Applications

#### Lesson 6: Solution of Linear and Nonlinear Equations

In Lesson #5 we addressed techniques for the solution of a single nonlinear equation of the form  $f(x) = 0$  by asking the question “What is  $x$  such that  $f(x) = 0$ ?”. In this lesson we extend this methodology to the case of multiple equations and multiple unknowns (where we will restrict our analysis, at present, to the case where the number of equations and number of unknowns are the same). We can represent the case of  $n$  equations and unknowns by allowing  $n$  independent variables,  $x_1, x_2, \dots, x_n$ , and writing each equation with different functional relationships,  $f_1, f_2, \dots, f_n$ , or

$$\begin{aligned} f_1(x_1, x_2, \dots, x_n) &= 0 \\ f_2(x_1, x_2, \dots, x_n) &= 0 \\ &\vdots \\ f_n(x_1, x_2, \dots, x_n) &= 0 \end{aligned} \tag{1}$$

Using the notation from Lesson #2 where a lower case bold letter represents a vector quantity and an upper case bold letter denotes a 2-D array, we have  $\mathbf{x} = [x_1, x_2, \dots, x_n]^T$  and  $\mathbf{f} = [f_1, f_2, \dots, f_n]^T$ . Now, with this simple notation, we can write the sequence of  $n$  equations and their dependence on  $n$  variables as a single vector equation, or

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \tag{2}$$

Thus, the matrix/vector notation in eqn. (2) simply allows writing eqn. (1) in a more concise form. With this form, we can solve this system of equations by asking a similar question as before, “What is  $\mathbf{x}$  such that  $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ ?”. Here, of course, we need to find all  $n$  elements of the vector  $\mathbf{x}$  by simultaneously forcing each function,  $f_j(\mathbf{x})$ , to zero for  $j = 1, 2, \dots, n$ .

Equation (2) is a general relationship that is often used for general nonlinear systems. If, however, the system is linear with respect to the elements of  $\mathbf{x}$  (i.e. the  $x_i$  only appear to the 1<sup>st</sup> power and there are no products of the form  $x_i x_j$ ), then eqns. (1) and (2) are usually written as

$$\begin{aligned} f_1(x_1, x_2, \dots, x_n) &= a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n - b_1 = 0 \\ f_2(x_1, x_2, \dots, x_n) &= a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n - b_2 = 0 \\ &\vdots \\ f_n(x_1, x_2, \dots, x_n) &= a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n - b_n = 0 \end{aligned} \tag{3}$$

$$\text{or } \mathbf{Ax} = \mathbf{b} \tag{4}$$

where

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad \text{and} \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Equation (4) is the linear subset of the more general relationship given in eqn. (2).

In this lesson we will address the solution of eqns. (2) and (4) via a number of methodologies (with focus on a series of numerical techniques -- recall that analytical hand computations for the linear case was already discussed in the **Linear Algebra** section of Lesson #2). We will focus on the linear case first [i.e. the solution of eqn. (4)] and then briefly introduce some techniques for the solution of a system of nonlinear equations written in the form of eqn. (2).

We have already introduced a bunch of notation associated with linear algebra in Lesson #2 and clearly, you should review this material, as needed, before continuing here. The current section of lecture notes introduces the key techniques for computer solution of a system of linear or nonlinear equations and provide some further explanation and insight into Matlab's backslash operator for the solution of linear equations.

Most of the specific subjects of interest here are treated in some detail in Part 3 of your Numerical Methods text by Chapra. This section of the text includes Chapters 8 – 12 which discuss several key concepts and the basic terminology of linear algebra (which we have already discussed), various direct elimination methods and iterative techniques for linear equations, and a brief overview of two methods for the solution of nonlinear equations. These lecture notes will summarize the key points from Chapters 8 – 12 in Chapra and also provide some additional terminology, insight, and applications experience (also please refer back to the Lesson #2 Lecture Notes, as needed).

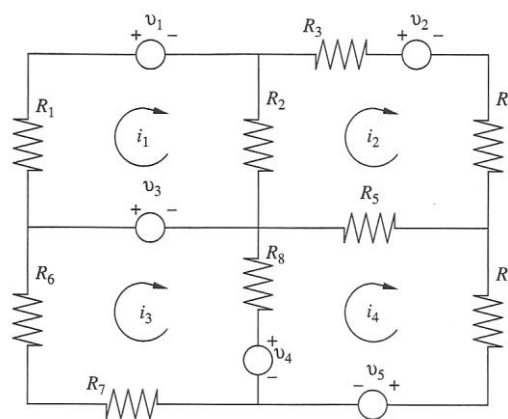
Upon completion of this lesson you should have a good understanding of both direct and iterative methods for the solution of linear equations, and a basic overview of various nonlinear solution strategies. You should leave this lesson with increased insight and confidence in applying a variety of techniques for the solution of both linear and nonlinear systems -- and be able to apply Matlab (and other software tools) for the solution of practical applications involving a system of coupled equations (either linear or nonlinear).

### Motivation

Providing motivation for understanding the material from this lesson is easy since systems of simultaneous equations occur in every field of science and engineering. Here we provide three straightforward illustrative examples where one needs the terminology and tools discussed in this lesson to analyze the system of interest. Actual solution and discussion of these systems will be addressed after the theory and methods have been presented.

#### Problem 1: Resistive Networks

Electric circuits containing only resistors lead to coupled linear algebraic equations. Kirchhoff's voltage law, which states, "the algebraic sum of the voltage drops around a closed loop must be zero," and Kirchhoff's current law, which says, "the sum of the currents entering a node is zero," are often used to develop these equations. For resistors, the voltage drop is given by Ohm's law,  $v = Ri$ , where  $R$  is the resistance (ohms),  $i$  is the current (amperes), and  $v$  is the voltage drop (volts).



In particular, for the resistive network shown in the sketch, we can apply Kirchoff's voltage law in each of the four loops to write four independent equations containing the four unknown currents,  $i_1$  through  $i_4$ , and known values for the resistances,  $R_1 - R_9$ , and voltage sources,  $v_1$  through  $v_5$ . Note that if there is a voltage rise (i.e. an increase in voltage) in the direction of the loop current, then we treat this as a negative voltage drop. Also, in common elements which are contained in more than one loop, the net current in that element in the direction of the loop current is used within the given loop equation.

To illustrate these comments within a real application, let's write the voltage law equations for the specific circuit shown in the sketch on the previous page, as follows:

$$\text{Loop 1:} \quad R_1 i_1 + v_1 + R_2 (i_1 - i_2) - v_3 = 0 \quad (5a)$$

$$\text{Loop 2:} \quad R_2 (i_2 - i_1) + R_3 i_2 + v_2 + R_4 i_2 + R_5 (i_2 - i_4) = 0 \quad (5b)$$

$$\text{Loop 3:} \quad R_6 i_3 + v_3 + R_8 (i_3 - i_4) + v_4 + R_7 i_3 = 0 \quad (5c)$$

$$\text{Loop 4:} \quad -v_4 + R_8 (i_4 - i_3) + R_5 (i_4 - i_2) + R_9 i_4 + v_5 = 0 \quad (5d)$$

where, for example, in Loop 1  $i_1 - i_2$  is the net current through  $R_2$ ,  $v_1$  is a voltage drop, and  $v_3$  is a voltage rise all in the direction of  $i_1$  (and the other equations are developed in a similar fashion).

Now we simply rewrite these equations in a more convenient form [i.e. collecting the coefficients of the four unknown currents in sequential order and putting the known voltage sources on the right-hand side (RHS) of the equations]. Doing this gives

$$\text{Loop 1:} \quad (R_1 + R_2) i_1 - R_2 i_2 = v_3 - v_1 \quad (6a)$$

$$\text{Loop 2:} \quad -R_2 i_1 + (R_2 + R_3 + R_4 + R_5) i_2 - R_5 i_4 = -v_2 \quad (6b)$$

$$\text{Loop 3:} \quad (R_6 + R_7 + R_8) i_3 - R_8 i_4 = -v_3 - v_4 \quad (6c)$$

$$\text{Loop 4:} \quad -R_5 i_2 - R_8 i_3 + (R_5 + R_8 + R_9) i_4 = v_4 - v_5 \quad (6d)$$

Finally, writing this in standard matrix/vector form, we have

$$\begin{bmatrix} R_1 + R_2 & -R_2 & 0 & 0 \\ -R_2 & (R_2 + R_3 + R_4 + R_5) & 0 & -R_5 \\ 0 & 0 & R_6 + R_7 + R_8 & -R_8 \\ 0 & -R_5 & -R_8 & R_5 + R_8 + R_9 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \\ i_3 \\ i_4 \end{bmatrix} = \begin{bmatrix} v_3 - v_1 \\ -v_2 \\ -v_3 - v_4 \\ v_4 - v_5 \end{bmatrix} \quad (7)$$

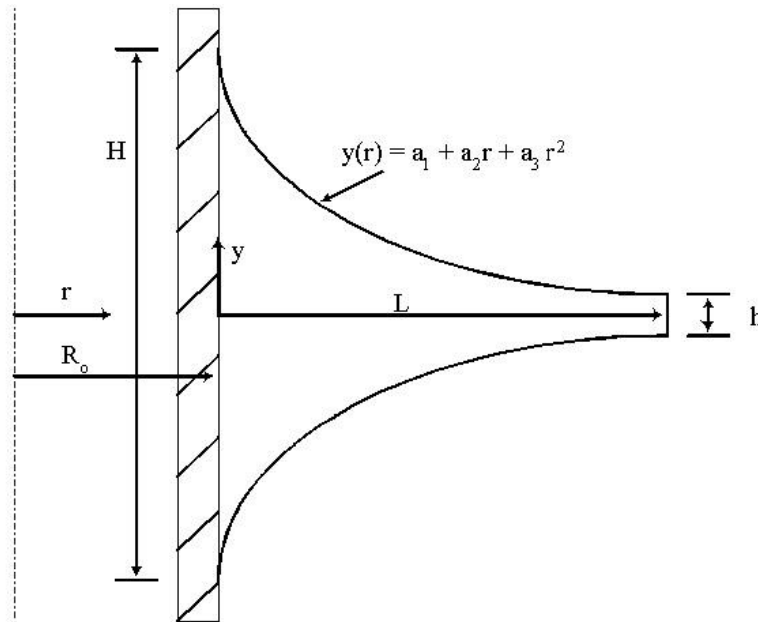
or simply

$$\mathbf{Ax} = \mathbf{b} \quad (8)$$

where the formal definitions of the  $\mathbf{A}$  and  $\mathbf{b}$  coefficient matrices and the solution vector  $\mathbf{x}$  are via a one-to-one correspondence with eqn. (7). The solution of this linear system of equations gives the desired loop currents  $\mathbf{x} = [i_1 \ i_2 \ i_3 \ i_4]^T$  for this problem!!!

### Problem 2: Geometry of a Parabolic Fin

Consider the physical design of a cylindrical fin with a parabolic shape (see sketch below). The purpose of the fin's extended surface is to increase the heat transfer area and to enhance the overall heat transfer from the cylindrical tube. As part of a team assigned to design a fin for peak efficiency, your job is to determine the fin surface area versus length. For this analysis we will keep most of the geometry parameters constant, allowing only the length,  $L$ , to vary (that is,  $R_o$ ,  $H$ , and  $h$  are fixed parameters, and  $L$  is known in any particular case, but it will be allowed to vary so that the surface area vs. length can be determined).



The shape of the fin is given by

$$y(r) = a_1 + a_2 r + a_3 r^2 \quad (9)$$

with the constraints

$$y(R_o) = \frac{H}{2}, \quad y(R_o + L) = \frac{h}{2}, \quad \text{and} \quad \left. \frac{dy}{dr} \right|_{r=R_o+L} = 0 \quad (10)$$

These three constraints give three equations for the unknown coefficients,  $a_1$ ,  $a_2$ , and  $a_3$  -- which vary as a function of the fin's length,  $L$ . For a given set of geometry parameters,  $R_o$ ,  $H$ ,  $h$ , and  $L$ , applying the above constraints gives

$$\frac{H}{2} = a_1 + a_2 R_o + a_3 R_o^2 \quad (11a)$$

$$\frac{h}{2} = a_1 + a_2 (R_o + L) + a_3 (R_o + L)^2 \quad (11b)$$

$$0 = a_2 + 2a_3 (R_o + L) \quad (11c)$$

Now, noting that the unknowns of interest are the  $a_1$ ,  $a_2$ , and  $a_3$  coefficients, we can write these three equations in standard  $\mathbf{Ax} = \mathbf{b}$  form, where

$$\mathbf{A} = \begin{bmatrix} 1 & R_o & R_o^2 \\ 1 & R_o + L & (R_o + L)^2 \\ 0 & 1 & 2(R_o + L) \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \quad \text{and} \quad \mathbf{b} = \begin{bmatrix} H/2 \\ h/2 \\ 0 \end{bmatrix} \quad (12)$$

and the solution vector  $\mathbf{x}$  gives the needed  $a_1$ ,  $a_2$ , and  $a_3$  coefficients and the desired  $y(r)$  profile as given by the quadratic expression in eqn. (9). Thus, we see that the solution of a linear system of equations in the form  $\mathbf{Ax} = \mathbf{b}$  is required here.

However, for this problem, knowing  $y(r)$  for each value of  $L$  is not the final result -- since here we are interested in the fin's surface area vs.  $L$  (since the heat loss from the fin is related to its heat transfer surface area,  $A$ ). From the sketch, we can compute  $A$  for each  $L$  from the following expression:

$$A = 2 \times \text{top surface area} + \text{tip area} \quad (13)$$

$$\text{or} \quad A = 2 \int 2\pi r ds + 2\pi(R_o + L)h \quad (14)$$

where  $ds$  is the differential length along the surface at position  $r$ . From basic calculus, we know that

$$ds = \sqrt{1 + \left(\frac{dy}{dr}\right)^2} dr$$

Therefore, upon substitution we have

$$A = 4\pi \int_{R_o}^{R_o+L} r \sqrt{1 + \left(\frac{dy}{dr}\right)^2} dr + 2\pi(R_o + L)h \quad (15)$$

This integral can be evaluated numerically using Matlab's *quadl* or *integral* function (we will elaborate on this further when we solve this problem within Matlab later in these notes).

Thus, to solve/analyze this problem, a general outline of the solution algorithm would be as follows:

1. Setup basic problem parameters ( $R_o$ ,  $H$ ,  $h$ , and range of  $L$  values)
2. Loop over number of  $L$  values
  - a. Setup coefficient matrices as defined in eqn. (12)
  - b. Solve system of equations to find the  $a_1$ ,  $a_2$ , and  $a_3$  coefficients for the  $y(r)$  expression
  - c. Use *quadl* or *integral* to evaluate the fin's surface area via eqn. (15)
3. Plot and tabulate the key results [i.e.  $y(r)$  and  $A$  for several  $L$  values]

Again, we see that the analysis of this design problem requires the setup and solution of a system of linear equations (which we will solve later in this section of notes)!!!

### Problem 3: Fin Heat Transfer via the FD Method (Revisited)

One of the illustrative applications from Lesson #4 introduced the Finite Difference Method (see [fd\\_intro.pdf](#)) for solving both Initial Value Problems (IVPs) and Boundary Value Problems (BVPs). In that example we saw that the numerical solution of BVPs leads directly to a set of coupled equations where, in general, very large systems are often encountered (it was noted that having greater than 1 million unknowns is fairly routine in multidimensional problems). This application area, by itself, clearly emphasizes the need for efficient numerical schemes to solve large linear systems of the form  $\mathbf{Ax} = \mathbf{b}$ . However, in most real engineering problems, the defining equations have nonlinear elements, which usually increases the computational demands by an order of magnitude or more. Thus, instead of solving *n linear equations* given by  $\mathbf{Ax} = \mathbf{b}$ , we must often solve *n nonlinear equations* of the form  $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ .

To illustrate the added complexity associated with nonlinear problems (and to motivate our need to discuss methods for solution of these problems), let's revisit the rectangular fin heat transfer problem discussed previously (see [rect1d\\_fin\\_1.pdf](#) and [fd\\_intro.pdf](#)). However, this time we will add a new twist to the problem -- by imposing a *temperature dependent heat transfer coefficient,  $h(T)$ , instead of a constant  $h$* . This slight change to the problem specification modifies the problem solution algorithm significantly!!!

To be specific, consider the fin geometry shown in the sketch with a linear dependence of the heat transfer coefficient with temperature, or

$$h = h(T) = a_1 + a_2 T \quad (16)$$

Previously we showed that the defining steady state energy balance equation was given by

$$\frac{d^2 T}{dx^2} - m^2 (T - T_\infty) = 0 \quad (17)$$

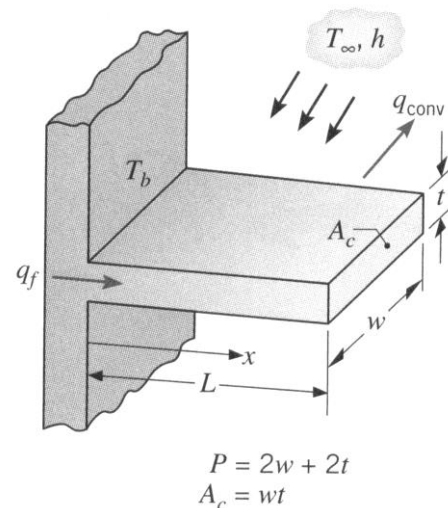
$$\text{with } m^2 = \frac{hP}{kA_c} \quad (18)$$

and that the specific BCs for this problem included a fixed temperature on the left (at  $x = 0$ ) and convection to the environment on the right (at  $x = L$ ), or

$$T(0) = T_b \quad \text{and} \quad -k \left. \frac{dT}{dx} \right|_{x=L} = h(T - T_\infty) \Big|_{x=L} \quad (19)$$

These expressions are still valid for the present problem if we recognize that  $h = h(T)$  is given by eqn. (16) [instead of being constant as done previously].

We will use the Finite Difference (FD) solution method to solve this nonlinear BVP. In particular, in the Lesson #4 example where  $h$  was constant, we showed that discretization of the 1-D problem lead to the following discrete equations for the nodal temperature values (see [fd\\_intro.pdf](#)):



Linear FD Equations (for constant h)

$$\text{Node 1:} \quad -(2 + m^2 \Delta x^2) T_1 + T_2 = -m^2 \Delta x^2 T_\infty - T_b \quad (20a)$$

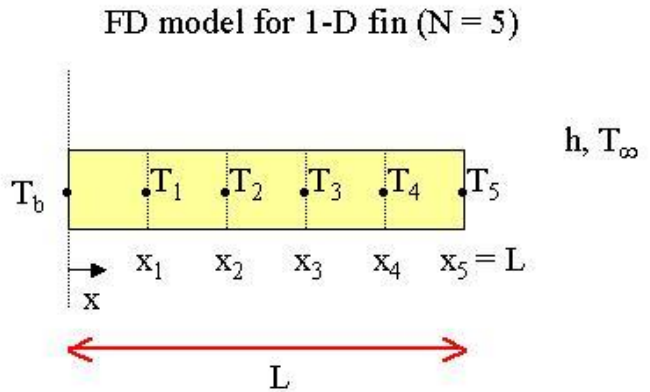
$$\text{Node 2 to N-1:} \quad T_{i-1} - (2 + m^2 \Delta x^2) T_i + T_{i+1} = -m^2 \Delta x^2 T_\infty \quad (20b)$$

$$\text{Node N:} \quad T_{N-2} - \left( 2m^2 \Delta x^2 + 1 + \frac{2h\Delta x}{k} \right) T_N = - \left( \frac{2h\Delta x}{k} + 2m^2 \Delta x^2 \right) T_\infty \quad (20c)$$

where N is the number of unknown temperature points in the discrete model (the general nodal discretization for this problem is illustrated in the sketch to the right for the case of N = 5). These linear equations can be assembled into standard form

$$\mathbf{AT} = \mathbf{b} \quad (21)$$

where  $\mathbf{T}$  is the desired temperature vector and  $\mathbf{A}$  and  $\mathbf{b}$  are constant coefficient matrices for the case where the heat transfer coefficient is constant.



Now, for the case where the heat transfer coefficient is temperature dependent [i.e.  $h = h(T)$ ], the only required change in the model is associated with the fact that, since  $h$  varies with  $T$ , and  $T$  varies with position  $x$ , then  $h$  is indeed spatially dependent -- that is,  $h(T) \rightarrow h(x) \rightarrow h(x_i) \rightarrow h_i$ . Thus, the  $m^2$  parameter in the base model is also space dependent and eqn. (18) needs to be written as

$$m_i^2 = \frac{h_i P}{k A_c} = \frac{P}{k A_c} (c_1 + c_2 T_i) \quad (22)$$

where the last equality is valid for this specific problem because of our assumption that  $h$  varies linearly with  $T$ .

Now to incorporate this new behavior into the discrete FD model, we simply need to include a subscript  $i$  on the  $m^2$  and  $h$  parameters in the nodal balance equations to denote their dependence on space. Doing this gives:

Nonlinear FD Equations (for variable h)

$$\text{Node 1:} \quad -(2 + m_1^2 \Delta x^2) T_1 + T_2 = -m_1^2 \Delta x^2 T_\infty - T_b \quad (23a)$$

$$\text{Node 2 to N-1:} \quad T_{i-1} - (2 + m_i^2 \Delta x^2) T_i + T_{i+1} = -m_i^2 \Delta x^2 T_\infty \quad (23b)$$

$$\text{Node N:} \quad T_{N-2} - \left( 2m_N^2 \Delta x^2 + 1 + \frac{2h_N \Delta x}{k} \right) T_N = - \left( \frac{2h_N \Delta x}{k} + 2m_N^2 \Delta x^2 \right) T_\infty \quad (23c)$$

where  $m_i^2$  and  $h_N$  are defined as above in eqns. (22) and (16), respectively.

Clearly the resultant equations for this problem are nonlinear because of the many  $T_i^2$  terms that occur explicitly if the actual expressions for  $m_i^2$  and  $h_N$  are substituted into the discrete equations. Thus, the sequence of  $N$  coupled nonlinear equations given in eqn. (23) can be written symbolically as  $\mathbf{f}(\mathbf{T}) = \mathbf{0}$ , or as  $\mathbf{A}(\mathbf{T})\mathbf{T} = \mathbf{b}(\mathbf{T})$ , where this latter form explicitly shows that the  $\mathbf{A}$  and  $\mathbf{b}$  coefficient matrices are temperature dependent [implying a general nonlinear dependence on  $T$  because of the multiplication of the  $\mathbf{A}(\mathbf{T})$  matrix into the  $\mathbf{T}$  vector].

Well, from the above discussion, it should be clear that we will need to solve a system of nonlinear equations to compute the discrete temperature profile within the fin for the case of a temperature dependent heat transfer coefficient. We will use this challenge as the motivation for studying solution methods for nonlinear systems as part of this lesson. Once we have discussed the methods in some detail, we will come back and solve this motivation problem as an explicit example of the basic methodology.

-----

**Note:** I personally prefer using the notation  $\mathbf{A}(\mathbf{x})\mathbf{x} = \mathbf{b}(\mathbf{x})$  for many nonlinear problems [instead of  $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ ] because it suggests an iterative solution technique for solving the system of nonlinear equations. Recall the One-Point or Fixed-Point Iteration scheme from Lesson #5 for solving a single nonlinear equation. In this case, because the quantity of interest appeared on both sides of the equation, we added an iteration index  $k$  to denote that the  $k^{\text{th}}$  estimate of the solution,  $\mathbf{x}_k$ , would be used to get a new estimate,  $\mathbf{x}_{k+1}$  -- that is,  $\mathbf{x}_{k+1} = \mathbf{g}(\mathbf{x}_k)$  for the One-Point Iteration method.

Now, for the case of  $n$  nonlinear coupled equations, we can envision an iterative strategy where the current estimate of the solution,  $\mathbf{x}_k$ , is used to estimate the coefficient matrices,  $\mathbf{A}(\mathbf{x}_k)$  and  $\mathbf{b}(\mathbf{x}_k)$ , and then these are used to write a **linear system** (with known  $\mathbf{A}$  and  $\mathbf{b}$ ) that can be solved for a new estimate of the solution vector,  $\mathbf{x}_{k+1}$ . Symbolically this scheme can be written as

$$\mathbf{A}(\mathbf{x}_k)\mathbf{x}_{k+1} = \mathbf{b}(\mathbf{x}_k) \quad (24)$$

This basic idea is referred to as the Linearized Iteration Method or the Successive Substitution Method and we will discuss this method in more detail in a subsequent subsection of these notes (and then actually use this method to solve this motivation problem).

### Solution Techniques for Linear Systems

Hopefully the above three problems have indeed increased your interest in studying solution techniques for both linear and nonlinear systems -- these methods are needed in many analysis situations and they represent essential tools for every practicing engineer. We will focus first on linear systems and, near the end of the lesson, briefly overview some common methods for nonlinear systems. Finally, once we have a good handle on all the terminology and techniques - especially those available within Matlab -- we will come back and actually solve the motivation problems discussed in the previous section (as well as present some additional cases studies for further illustration of the variety of practical situations where these methods are needed).

Our study of linear systems is broken into two subsections that highlight **elimination methods** and **general iterative techniques**, respectively, as practical computational approaches for



computer implementation. A simple example of a 3x3 linear system will be used to illustrate the basic ideas from each method.

### Direct Elimination Methods

There are two general schemes for solving linear systems using the computer: **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. This subsection of notes overviews a formal 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 (or variations of the base methodology) is often the preferred direct solution method for low to medium sized systems (up to a few thousand 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 on iterative methods as part of the Lesson #6 Lecture Notes is on the basic Gauss Seidel scheme and on the use of Successive Relaxation (SR) to help accelerate convergence. Our introductory study of iterative methods will follow the current subsection on direct elimination methods.

As noted in the introductory paragraphs, direct elimination methods formally implement the three standard “row operations” that were discussed in Lesson #2 to transform the original system of linear equations into a form that is easier to solve. In our previous illustrations of the basic procedure (see the hand examples starting on pages 14 and 19 of the Lesson #2 Lecture Notes), we only used the last two row operations (**normalization by a constant** and **addition of a constant times one row to another**) to achieve the desired goals within the given examples. In particular, the first row operation in our list, **the interchange of two rows**, was not utilized in our previous examples.

To illustrate the importance of this row operation (i.e. **the interchange of two rows**), let’s take our example from Lesson #2 for solving  $\mathbf{Ax} = \mathbf{b}$  and re-order (i.e. interchange) the sequence of the equations, as follows;

**Original System:**

$$\begin{bmatrix} 3 & -2 & 0 \\ -1 & 3 & -2 \\ 0 & -1 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -7 \\ 9 \\ -5 \end{bmatrix}$$

**Revised System:**

$$\begin{bmatrix} 0 & -1 & 3 \\ -1 & 3 & -2 \\ 3 & -2 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -5 \\ 9 \\ -7 \end{bmatrix}$$

where, in the revised system, we have simply interchanged rows 1 and 3 -- and, clearly, this re-ordering of the equations does not alter the basic equalities present in the original system.

However, if we now try to impose the remaining two row operations to put the revised system into upper triangular form, we see that a problem quickly develops. For example, in the revised system, if we multiply the 2<sup>nd</sup> equation by 3 and add this to the 3<sup>rd</sup> equation, we get

$$\begin{bmatrix} 0 & -1 & 3 \\ -1 & 3 & -2 \\ 0 & 7 & -6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -5 \\ 9 \\ 20 \end{bmatrix}$$

Now, remembering that our goal here is to put the system in upper triangular form, the pertinent question is, “How do we make the 2,1 element zero?”. The answer, of course, is that we simply cannot achieve this goal – at least, not without a suitable row interchange. What has happened here is that the  $a_{11}$  **pivot element** (i.e. the  $a_{ii}$  element in row or column  $i$ ) in the revised system is zero, and this is a big no-no!!! This element is needed to make the 2,1 element zero, but this is not possible if it is zero. In fact, if a system has one or more diagonal elements that are zero, and these zeros cannot be removed via suitable row interchanges, then the **A** matrix is singular and there is no solution to the system given by **Ax = b**. Clearly, an interchange of rows 1 and 2 in the above systems rectifies the temporary dilemma, and allows one to proceed with applying additional row operations to achieve the desired upper triangular form.

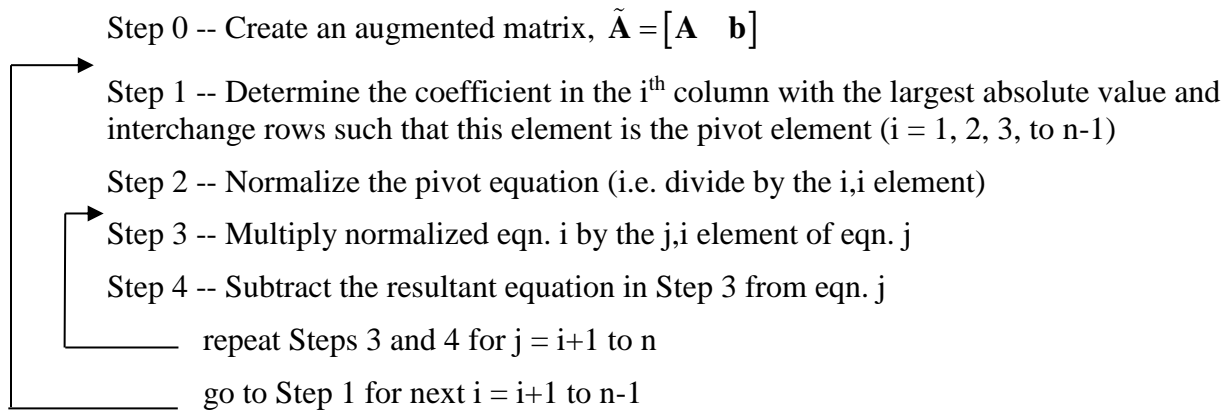
Now, in general system modeling and during subsequent manipulation of the original equations, it is not uncommon to have a temporary zero appear in one of the diagonal pivot elements. This is not usually a concern, however, since a simple row interchange can remedy the temporary situation (unless the matrix is indeed singular). The procedure for checking and performing the row interchange, if it is needed, is referred to as the **partial pivoting** process. The procedure sequentially searches for the coefficient with the largest absolute value in the column below the pivot element, and then the rows are switched so that the largest element becomes the pivot element. The procedure is relatively simple to implement, and it is an absolutely essential component of any general-purpose solution algorithm!!!

It is important to emphasize that, in addition to avoiding a zero diagonal element (and subsequent division by zero during the normalization and back substitution processes), partial pivoting is also used to help minimize the round off error that occurs in each arithmetic computation that is performed. This feature is illustrated nicely in Ex. 9.4 in your text by Chapra -- you should take a look if you haven't already done so...

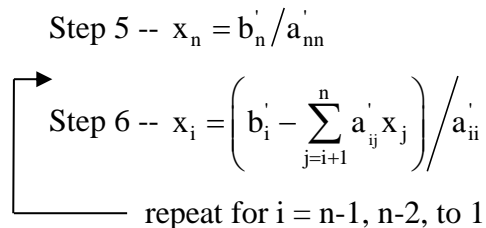
Well, with the above emphasis on partial pivoting and our previous examples of the other row operations, we can now present a formal direct algorithm for solving linear systems on the computer. The formal procedure is referred to as the **Gauss Elimination Method** and, as we have already seen, it involves a forward elimination step followed by the back substitution step. With reference to a system of  $n$  equations and  $n$  unknowns,

$$\mathbf{Ax} = \mathbf{b}$$

the **Forward Elimination Step** (with partial pivoting) becomes:



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



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

To illustrate this formal procedure, let's re-do the example from page 14 of the Lesson #2 Lecture Notes. Note that, although the two cases are very similar, here we follow the above algorithm religiously, whereas in our earlier hand computation we were more interested in finding the solution for a specific problem. However, when implementing a general-purpose algorithm on the computer, the code follows the same sequence of steps every time, independent of the specific problem being solved. Thus, the purpose of the example here is to illustrate the precise steps noted in the above **Gauss Elimination** procedure:

Given

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

$$\begin{bmatrix} 3 & -2 & 0 \\ -1 & 3 & -2 \\ 0 & -1 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -7 \\ 9 \\ -5 \end{bmatrix}$$

Step 0 -- form augmented matrix

$$\begin{bmatrix} 3 & -2 & 0 & -7 \\ -1 & 3 & -2 & 9 \\ 0 & -1 & 3 & -5 \end{bmatrix}$$

Step 1 --  $i = 1$  perform partial pivoting

$$\begin{bmatrix} 3 & -2 & 0 & -7 \\ -1 & 3 & -2 & 9 \\ 0 & -1 & 3 & -5 \end{bmatrix} \quad (\text{no change -- no row interchange needed})$$

Step 2 --  $i = 1$  normalize pivot equation

$$\begin{bmatrix} 1 & -2/3 & 0 & -7/3 \\ -1 & 3 & -2 & 9 \\ 0 & -1 & 3 & -5 \end{bmatrix}$$

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

$$\begin{bmatrix} 1 & -2/3 & 0 & -7/3 \\ 0 & 7/3 & -2 & 20/3 \\ 0 & -1 & 3 & -5 \end{bmatrix}$$

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

$$\begin{bmatrix} 1 & -2/3 & 0 & -7/3 \\ 0 & 7/3 & -2 & 20/3 \\ 0 & -1 & 3 & -5 \end{bmatrix} \quad (\text{no change})$$

Step 1 --  $i = 2$  perform partial pivoting

$$\begin{bmatrix} 1 & -2/3 & 0 & -7/3 \\ 0 & 7/3 & -2 & 20/3 \\ 0 & -1 & 3 & -5 \end{bmatrix} \quad (\text{no change -- no row interchange needed})$$

Step 2 --  $i = 2$  normalize pivot equation

$$\begin{bmatrix} 1 & -2/3 & 0 & -7/3 \\ 0 & 1 & -6/7 & 20/7 \\ 0 & -1 & 3 & -5 \end{bmatrix}$$

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

$$\begin{bmatrix} 1 & -2/3 & 0 & -7/3 \\ 0 & 1 & -6/7 & 20/7 \\ 0 & 0 & 15/7 & -15/7 \end{bmatrix}$$

Stop elimination phase since  $i = 2 = n-1 = 2$

Step 5 -- evaluate the last element of the vector

$$x_3 = \frac{-15/7}{15/7} = -1$$

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

$$x_2 = \frac{20}{7} + \frac{6}{7}x_3 = \frac{20-6}{7} = 2$$

$$x_1 = -\frac{7}{3} + \frac{2}{3}x_2 = \frac{-7+4}{3} = -1$$

Therefore the final result is  $\mathbf{x} = [-1 \ 2 \ -1]^T$  and clearly, this is the same result as before. The only difference here is that we followed a very formal procedure at each step of the process -- and, if we can do this by hand, then it can be coded into the computer for general purpose use!!!

One slight disadvantage of the above algorithm is that the  $\mathbf{b}$  vector is manipulated along with the  $\mathbf{A}$  matrix as part of the augmented matrix formed in Step 0 of the procedure. What if we wanted to solve several systems with the same  $\mathbf{A}$  but different  $\mathbf{b}$  vectors? For example, we might be interested in solving

$$\mathbf{Ax}_1 = \mathbf{b}_1, \quad \mathbf{Ax}_2 = \mathbf{b}_2, \quad \mathbf{Ax}_3 = \mathbf{b}_3, \quad \text{etc.}$$

Clearly, we could simply use the above algorithm multiple times, but this would be quite inefficient, since the same manipulations to transform the  $\mathbf{A}$  matrix would be done each time. Certainly it would be better if we could devise a method to modify the  $\mathbf{A}$  and  $\mathbf{b}$  matrices separately.

In fact, the so-called **LU Decomposition Method** has the desired property that the matrix modification (or decomposition) step can be performed independent of the right-hand side (RHS) vector. This feature is quite useful in practice -- therefore, the LU Decomposition Method, as outlined below, is usually the direct scheme of choice in most applications.

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

$$\mathbf{A} = \mathbf{LU} \tag{25}$$

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

Now, the original system of equations,  $\mathbf{Ax} = \mathbf{b}$ , becomes

$$\mathbf{LUx} = \mathbf{b} \tag{26}$$

This expression can be broken into two problems,

$$\mathbf{Ly} = \mathbf{b} \quad \text{and} \quad \mathbf{Ux} = \mathbf{y} \tag{27}$$

The rationale behind this approach is that the two systems given in eqn. (27) are both easy to solve; one by forward substitution and the other by back substitution. In particular, because  $\mathbf{L}$  is a lower triangular matrix, the expression  $\mathbf{Ly} = \mathbf{b}$  can be solved with a forward substitution step. Similarly, since  $\mathbf{U}$  has upper triangular form,  $\mathbf{Ux} = \mathbf{y}$  can be evaluated with a simple back substitution algorithm.

Thus, the key to this method is the ability to find two matrices,  $\mathbf{L}$  and  $\mathbf{U}$ , that satisfy eqn. (25). Doing this is referred to as the **Decomposition Step** and there are a variety of algorithms available (note that this is performed without knowledge of the  $\mathbf{b}$  vector). For example, **Doolittle Decomposition** (for a  $4 \times 4$  system) would be written as

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ \ell_{21} & 1 & 0 & 0 \\ \ell_{31} & \ell_{32} & 1 & 0 \\ \ell_{41} & \ell_{42} & \ell_{43} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \quad (28)$$

and, because of the specific structure of the matrices, a systematic set of formulae for the components of **L** and **U** results.

For example, a simple scheme (i.e. no partial pivoting) for computing **L** and **U** via Doolittle Decomposition can be developed by simply doing the indicated multiplication of **L** into **U** in a systematic fashion, noting that there is only one unknown per equation (that is, the specific form of **L** and **U** lead to  $n^2$  sequential evaluations -- and the equations are completely decoupled!!!). A few of the steps are as follows:

row 1 of **L** into column 1 of **U**:  $u_{11} = a_{11}$

row 1 of **L** into column 2 of **U**:  $u_{12} = a_{12}$

or  $u_{1j} = a_{1j} \quad \text{for } j = 1, 2, \dots, n$

row 2 of **L** into column 1 of **U**:  $\ell_{21}u_{11} = a_{21} \quad \text{or} \quad \ell_{21} = \frac{a_{21}}{u_{11}} = \frac{a_{21}}{a_{11}}$

$\ell_{21}u_{12} + u_{22} = a_{22} \quad \text{or} \quad u_{22} = a_{22} - \ell_{21}u_{12}$

row 2 of **L** into column 2 of **U**:  $= a_{22} - \frac{a_{21}}{a_{11}}a_{12}$

etc...

and this can be developed into an efficient computational scheme for the unknown elements of the **L** and **U** matrices (with only one unknown per equation).

Note that, since the diagonal elements of **L** are unity, these do not require storage, and one can store all the remaining elements of **L** and **U** within a single  $n \times n$  matrix. Also note that, in any general-purpose implementation, partial pivoting will be required to avoid a division by zero and to minimize associated round-off errors in the calculations. Within this context, we should note that the row interchange information is often stored in a **row permutation matrix**, **P**, where **P** = **I** for no row interchanges. To see this, consider a  $3 \times 3$  system, where initially **A** = **IA** = **PA**, or

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \mathbf{PA} \quad \text{with} \quad \mathbf{P} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Now if we interchange rows 1 and 3, for example, the new matrix becomes

$$\begin{bmatrix} a_{31} & a_{32} & a_{33} \\ a_{21} & a_{22} & a_{23} \\ a_{11} & a_{12} & a_{13} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \mathbf{PA} \quad \text{with} \quad \mathbf{P} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

Thus, we can think of  $\mathbf{PA}$  as the original matrix with all the row interchanges included within the permutation matrix,  $\mathbf{P}$ . Therefore, in practical implementations that include partial pivoting, the LU decomposition technique is really solving a problem of the form

$$\mathbf{PAx} = \mathbf{LUx} = \mathbf{Pb} \quad (29)$$

So that  $\mathbf{PA} = \mathbf{LU}$  in practice, and the permutation matrix is needed to solve for the solution vector  $\mathbf{x}$ .

Now, the original system of equations,  $\mathbf{PAx} = \mathbf{Pb}$ , becomes

$$\mathbf{LUx} = \mathbf{Pb} \quad (30)$$

This expression can be broken into two problems (as discussed above),

$$\mathbf{Ly} = \mathbf{Pb} \quad \text{and} \quad \mathbf{Ux} = \mathbf{y} \quad (31)$$

These equations still only require simple forward and back substitution algorithms once the  $\mathbf{L}$  and  $\mathbf{U}$  (and  $\mathbf{P}$ ) matrices are known.

-----

As you might expect, Matlab has a built-in **lu** function that does the LU factorization discussed above. To illustrate its use, let's solve the "revised" system example from above (I chose this as the test system here since I know that it requires some partial pivoting and that the original system model does not -- thus, the resultant permutation matrix should not be the identity matrix). The revised system model is

$$\begin{bmatrix} 0 & -1 & 3 \\ -1 & 3 & -2 \\ 3 & -2 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -5 \\ 9 \\ -7 \end{bmatrix}$$

and the solution is given by  $\mathbf{x} = [-1 \ 2 \ -1]^T$ , as expected based on our previous work. The Matlab commands needed to generate this result via the **lu** function are:

```
% define matrices for demo
disp('Matrices for the LU demo'), A = [0 -1 3;-1 3 -2;3 -2 0], b = [-5 9 -7]'
%
% illustrate use of lu function
disp('Output from lu(A) function'), [L,U,P] = lu(A)
disp('Intermediate y solution'), y = L\(P*b)
disp('Solution vector x for revised system'), x = U\y
```

and the Matlab screen output using these commands is listed below:

```
Matrices for the LU demo
A =
     0     -1      3
    -1      3     -2
     3     -2      0
b =
    -5
     9
    -7
Output from lu(A) function
L =
    1.0000         0         0
   -0.3333     1.0000         0
         0   -0.4286     1.0000
```

```

U =
    3.0000    -2.0000         0
         0     2.3333    -2.0000
         0         0     2.1429

P =
     0     0     1
     0     1     0
     1     0     0

Intermediate y solution
y =
   -7.0000
    6.6667
   -2.1429

Solution vector x for revised system
x =
    -1
     2
    -1

```

Note here that the permutation matrix is given as expected (an interchange of rows 1 and 3 was required) and that the use of eqn. (31) is required to compute the  $\mathbf{x}$  vector ( $\mathbf{y}$  is simply an intermediate result that is not really useful -- other than to demonstrate the basic method).

-----

Before completing this subsection on Elimination Methods, we need to elaborate slightly upon the equivalency of the Gauss Elimination and LU Decomposition methods and the use of Matlab's backslash operator. The first item is illustrated nicely in Section 10.2 in your text by Chapra. Here the author shows that the  $\mathbf{L}$  and  $\mathbf{U}$  factors can be determined via the same series of row operations as performed in the Gauss Elimination procedure. Thus, the two elimination methods are strongly linked -- and their differences are more of a matter of perception and code implementation. In fact, the LU Decomposition method is often just considered a specific implementation of Gauss Elimination!!!

This last statement brings us to the discussion of Matlab's backslash operator. In practice, solving a system of linear equations of the form  $\mathbf{Ax} = \mathbf{b}$  is accomplished within Matlab as

```
x = A\b
```

The backslash operator (the  $\backslash$  symbol) represents a very sophisticated set of procedures that, after performing a test to determine the specific structure of the matrix, uses a variety of methods depending on the nature of the system. Matlab's command level help for the *mldivide* function simply says that "Gaussian elimination" is used to solve the problem (*mldivide* stands for *matrix left division* and this command is invoked when using the  $\backslash$  operator):

```

help mldivide
\ Backslash or left matrix divide.
  A\B is the matrix division of A into B, which is roughly the
  same as INV(A)*B, except it is computed in a different way.
  If A is an N-by-N matrix and B is a column vector with N
  components, or a matrix with several such columns, then
  X = A\B is the solution to the equation A*X = B computed by
  Gaussian elimination. A warning message is printed if A is
  badly scaled or nearly singular.

```



However, when using the full help documentation for *mldivide*, a discussion of the algorithm details gives a list of several rules and their order of precedence, where a portion of one of the latter (catch all) rules is listed below (see Matlab's full documentation for complete details):

If A is square and does not satisfy criteria 1 through 6, then a general triangular factorization is computed by Gaussian elimination with partial pivoting (see lu). This results in  $A = L*U$  where L is a permutation of a lower triangular matrix and U is an upper triangular matrix. Then X is computed by solving two permuted triangular systems:  $X = U \setminus (L \setminus B)$

Thus, we see that a combination of the basic Gauss Elimination and LU Decomposition methods discussed above do indeed form the basis for Matlab's backslash operator. The specific implementation in Matlab -- we simply solve  $Ax = b$  by using  $x = A \setminus b$  -- makes it easy for the user but, in practice, there is a fair amount of linear algebra, numerical methods, and clever programming behind this "simple" operation. Much of the discussion in this subsection of notes was given so that, as a user, you have a good understanding of what procedures are actually used to solve your problem...

### Iterative Methods

As noted in the introduction to this section of Lecture Notes, for large systems of equations, an iterative solution scheme for the unknown vector is often used. The iterative techniques of interest here are similar to the One-Point Iteration Scheme discussed in Lesson #5, but now we have a system of simultaneous equations instead of a single nonlinear equation. For linear systems, the one point iterative formulation can always be written in the general form

$$x^{p+1} = Bx^p + c \quad (32)$$

where **B** is the iteration matrix, **c** is a constant vector, and p is an iteration counter. In this formulation, the current estimate of the solution vector,  $x^p$ , is used to get a new (hopefully better) estimate of the solution,  $x^{p+1}$ .

As you may recall, the disadvantage of one-point iteration schemes is that they are not guaranteed to converge (i.e. they may or may not work) -- so the convergence properties of a particular scheme is of considerable interest. In particular, for the iterative scheme given by eqn. (32), detailed theoretical analysis has shown that convergence is guaranteed if the largest eigenvalue of the iteration matrix is less than unity, where

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

Therefore, if  $\rho < 1$  the iterative scheme will converge. If  $\rho \ll 1$ , 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, however, that  $\rho$  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).

Since the spectral radius is only practical to compute for small test systems, convergence is tested in real engineering systems during the actual iterative process by computing the largest relative change in the solution vector from one iteration to the next, and comparing the absolute value of this result with some desired user-specified tolerance. If the maximum relative change is less than the specified 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.

To illustrate the above ideas, we will discuss the so-called *Gauss Seidel Method* in some detail, since this is the most common one-point iteration scheme in use for linear systems. To start, let's take the original system of equations given by  $\mathbf{Ax} = \mathbf{b}$  and convert it into the classical Gauss Seidel iterative scheme. To do this, we break the original matrix into three specific components, or

$$\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U} \quad (33)$$

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  $\mathbf{L}$ ,  $\mathbf{D}$ , and  $\mathbf{U}$  correspond directly to the original elements of  $\mathbf{A}$ . For example, for a generic  $3 \times 3$  system, we have

$$\mathbf{L} = \begin{bmatrix} 0 & 0 & 0 \\ a_{21} & 0 & 0 \\ a_{31} & a_{32} & 0 \end{bmatrix} \quad \mathbf{D} = \begin{bmatrix} a_{11} & 0 & 0 \\ 0 & a_{22} & 0 \\ 0 & 0 & a_{33} \end{bmatrix} \quad \mathbf{U} = \begin{bmatrix} 0 & a_{12} & a_{13} \\ 0 & 0 & a_{23} \\ 0 & 0 & 0 \end{bmatrix} \quad (34)$$

Now, substituting eqn. (33) into the original balance expression gives

$$(\mathbf{L} + \mathbf{D})\mathbf{x} + \mathbf{U}\mathbf{x} = \mathbf{b}$$

or

$$(\mathbf{L} + \mathbf{D})\mathbf{x} = \mathbf{b} - \mathbf{U}\mathbf{x} \quad (35)$$

If we pre-multiply by  $(\mathbf{L} + \mathbf{D})^{-1}$  and notice that the solution vector appears on both sides of the equation, we can write the resultant equation in an iterative form, with  $p$  as the iteration counter, as

$$\mathbf{x}^{p+1} = -(\mathbf{L} + \mathbf{D})^{-1}\mathbf{U}\mathbf{x}^p + (\mathbf{L} + \mathbf{D})^{-1}\mathbf{b} \quad (36)$$

Clearly this is in the standard form for iterative solutions as defined in eqn. (32), where the iteration matrix is given by

$$\mathbf{B} = -(\mathbf{L} + \mathbf{D})^{-1}\mathbf{U} \quad (37)$$

and the constant vector is written as

$$\mathbf{c} = (\mathbf{L} + \mathbf{D})^{-1}\mathbf{b} \quad (38)$$

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, since explicit evaluation of the inverse matrix is computationally intensive.

For actual implementation on the computer, one writes these equations differently, never having to formally take the inverse as indicated above. In practice, eqn. (35) is written in iterative form as

$$(\mathbf{L} + \mathbf{D})\mathbf{x}^{p+1} = \mathbf{b} - \mathbf{U}\mathbf{x}^p$$

and manipulated to give

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

or

$$\mathbf{x}^{p+1} = \mathbf{D}^{-1}(\mathbf{b} - \mathbf{L}\mathbf{x}^{p+1} - \mathbf{U}\mathbf{x}^p) \quad (39)$$

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

In particular, in discrete form eqn. (39) can be expanded as

$$x_i^{p+1} = \frac{1}{a_{ii}} \left( b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{p+1} - \sum_{j=i+1}^n a_{ij} x_j^p \right) \quad (40)$$

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

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

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \quad (41)$$

The iterative equations for this case can be written individually as

$$x_1^{p+1} = \frac{1}{a_{11}} (b_1 - a_{12}x_2^p - a_{13}x_3^p) \quad (42a)$$

$$x_2^{p+1} = \frac{1}{a_{22}} (b_2 - a_{21}x_1^{p+1} - a_{23}x_3^p) \quad (42b)$$

$$x_3^{p+1} = \frac{1}{a_{33}} (b_3 - a_{31}x_1^{p+1} - a_{32}x_2^{p+1}) \quad (42c)$$

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***.

#### A Note about Diagonal Dominance:

A matrix is said to be ***diagonally dominant*** if the diagonal element is greater than the absolute sum of all the other elements in a row -- and this must be true for every row! This condition can be stated mathematically as

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}| \quad \text{for all } i \quad (43)$$

Now, within this context, studies have shown that diagonal dominance of the original  $\mathbf{A}$  matrix is a sufficient (but not necessary) condition for convergence of the Gauss Seidel method. This means that, if the system is diagonally dominant, it will converge using the Gauss Seidel method! However, if the system is not diagonally dominant then it may or may not converge (we simply do not know).

Unfortunately, most engineering systems are not strictly diagonally dominant. However, since most systems are derived from formal balance equations (where the loss terms are along the diagonal and the production terms appear in the off-diagonal elements), they tend to be nearly diagonally dominant, and these systems usually converge. Of course, this last statement is purposely very vague, since every system is different, and convergence is certainly not guaranteed unless the  $\mathbf{A}$  matrix is strictly diagonally dominant as indicated by eqn. (43). However, a general **rule of thumb** is that systems derived from physical balance equations “usually” converge using the Gauss Seidel approach -- and this is what makes this method so popular (it actually works more often than not)...

Now, since the convergence properties of a system are so important, 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. If the solution is diverging, this might help it to converge. This idea is the basis of the **successive relaxation (SR) method**.

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

$$\mathbf{x}^{p+1}\Big|_{\text{new}} = \alpha \mathbf{x}^{p+1} + (1 - \alpha) \mathbf{x}^p \quad \text{with} \quad 0 < \alpha < 2 \quad (44)$$

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

Note that the formal iteration process using successive relaxation coupled with the Gauss Seidel method is given by

$$\mathbf{x}^{p+1} = (\alpha \mathbf{L} + \mathbf{D})^{-1} ((1 - \alpha) \mathbf{D} - \alpha \mathbf{U}) \mathbf{x}^p + (\alpha \mathbf{L} + \mathbf{D})^{-1} \alpha \mathbf{b} \quad (45)$$

which is in the standard form for iterative solutions as defined in eqn. (32). Thus, we see that the iteration matrix for the SR method is given by

$$\mathbf{B} = (\alpha \mathbf{L} + \mathbf{D})^{-1} ((1 - \alpha) \mathbf{D} - \alpha \mathbf{U}) \quad (46)$$

Thus, eqn. (46) could be used in a formal study involving the spectral radius and the convergence properties of the SR method (for a small system).

The idea, of course, is to choose the relaxation parameter to improve convergence (reduce the spectral radius). In large problems, an optimum choice for  $\alpha$  is most often estimated in a trial

and error fashion for certain classes of problems (since determining the spectral radius for real systems is not practical). Some more advanced codes do try to estimate this quantity as part of the iterative calculation, although this is not particularly easy.

To illustrate some of these concepts and how the relaxation parameter affects convergence, let's implement the SR method for the same standard 3×3 sample problem we have discussed previously. In particular, we would like to analyze the spectral radius,  $\rho$ , and the convergence rate -- in terms of the number of iterations needed for convergence (called  $k$ ) -- as a function of  $\alpha$ . The problem to be solved is the same one that we have discussed earlier, or

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

$$\begin{bmatrix} 3 & -2 & 0 \\ -1 & 3 & -2 \\ 0 & -1 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -7 \\ 9 \\ -5 \end{bmatrix}$$

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 3. This routine is a simple implementation (i.e. no partial pivoting) of eqns. (40) and (44). Because no row interchanges are made internal to the routine, the original **A** matrix cannot have any zeros along the diagonal (since 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  $\mathbf{x}^{p+1}$  and  $\mathbf{x}^p$  are not retained, and the form,  $\mathbf{x} = \mathbf{x} + \dots$ , is used to update the current estimate of  $\mathbf{x}$  based on all the best values available at the time. Also, since two complete iterates of  $\mathbf{x}$  are not available, we have chosen to evaluate convergence based on the maximum residual,  $r_{\max}$ , on iteration  $p$ , which is defined as

$$r_{\max}^p = \max |\mathbf{r}^p| \quad \text{where} \quad \mathbf{r}^p = \mathbf{b} - \mathbf{A}\mathbf{x}^p \quad (47)$$

In **sr.m**, when  $r_{\max} < \text{tol} = 10^{-5}$ , 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 = [3 -2 0;-1 3 -2;0 -1 3]; b = [-7 9 -5]';
>> x1 = A\b
x1 =
    -1
     2
    -1
>> [x2,k] = sr(A,b,[0 0 0]',1.0,1.0e-5,1000)
x2 =
   -1.0000
    2.0000
   -1.0000
k =
    15
```

and we see that the **sr.m** routine gives the correct answer (to within the convergence criterion) in 15 iterations. Thus, we have benchmarked the base methodology!

**Table 3 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 (with 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 (last review: Nov. 2017)
%
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)) < 100*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
        rmax = max(abs(b - A*x));
        k = k + 1;
    end
end
%
% end of function

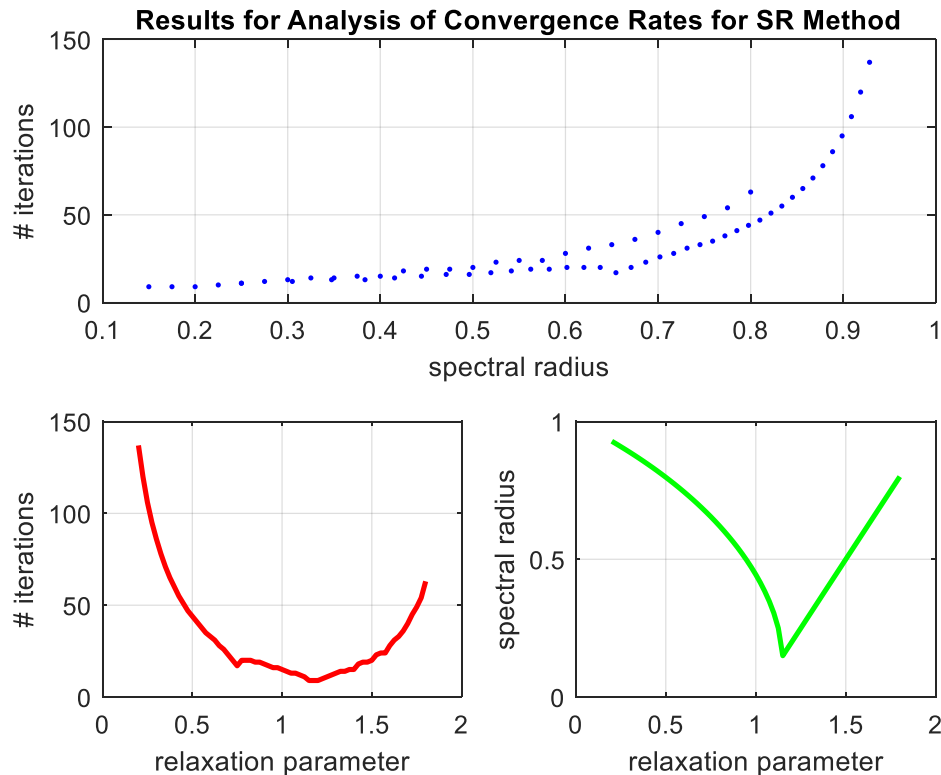
```

Now, to do the desired analyses, a series of computations are required and they have been implemented within Matlab script file **srdemo\_lesson6.m** -- see the code listing in Table 4. The program systematically solves the above 3<sup>rd</sup> order system for several values of  $\alpha$  in the range  $0.2 < \alpha < 1.8$  and stores the number of iterations required for convergence. In addition, for each value of  $\alpha$ , 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 3×3 system, this is a rather trivial, but very informative, computation.

Finally, with the stored values of  $k$  and  $\rho$  for several values of the relaxation parameter,  $\alpha$ , we can create a series of plots --  $k$  vs.  $\alpha$ ,  $\rho$  vs.  $\alpha$ , and  $k$  vs.  $\rho$  -- and these are summarized within three subplots in Fig. 1.

**Table 4 Matlab script file to solve sample problem using the SR method.**

```
%
%
% SRDEMO_LESSON6.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 (as discussed in the Lesson #6 Lecture Notes).
%
% This script file calls the SR.M routine to apply the SR method.
%
% File written by J. R. White, UMass-Lowell (last update: Nov. 2017))
%
    clear all, close all, nfig = 0;
%
% define system of interest and vector of relaxation parameters
    A = [3 -2 0;-1 3 -2;0 -1 3]; b = [-7 9 -5]';
    alf = 0.2:0.025:1.8; Nalf = length(alf);
%
% solve system for range of alf and save k = # of iterations to convergence
    tol = 1e-5; M = 1000; k = zeros(Nalf,1);
    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(Nalf,1);
    for j = 1:Nalf
        B = inv(alf(j)*L + D)*((1-alf(j))*D - alf(j)*U);
        ev = eig(B); p(j) = max(abs(ev));
    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('SRDemo: 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
```



**Fig. 1 Summary results for the SR demonstration example.**

As apparent, we see that the optimum relaxation parameter for this problem is near  $\alpha = 1.15$ . At this point, it takes only 9 iterations to converge (recall that the base Gauss Seidel method with  $\alpha = 1.0$  took 15 iterations). We also see that, as expected, the spectral radius has a minimum at the optimum  $\alpha$ . 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  $\rho$ . 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 a near optimum relaxation parameter, although not easy to do, is usually quite advantageous in larger more realistic applications. Although the methods used in this example are not practical in large more-realistic systems, they do allow us to illustrate many of the concepts discussed above!

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 introductory treatment of the subject, but the interested student is encouraged to see a good text on advanced numerical methods for further details as desired.



### Solution Techniques for Nonlinear Systems

In general, nonlinear equations are much more difficult to solve relative to linear equations. In all cases, the various techniques that are available involve some form of iteration, where a current guess is used to get a new (hopefully better) estimate of the solution. In many cases, the success of a particular method is sensitive to the starting guess and, because nonlinear equations can have multiple solutions, a different starting point can lead to a different solution. In all, the user must invest more thought, effort, and care when dealing with a system of nonlinear equations -- whereas, for linear systems, a simple  $\mathbf{x} = \mathbf{A} \backslash \mathbf{b}$  is all that is needed in most cases!

To illustrate some of the issues and techniques associated with nonlinear equations we will modify our  $3 \times 3$  linear example from the previous section to include several nonlinear terms, and then we will go about solving these equations using several methods.

To start, let's consider our previous linear example:

#### Linear Example:

$$3x_1 - 2x_2 = -7 \quad (48a)$$

$$-x_1 + 3x_2 - 2x_3 = 9 \quad (48b)$$

$$-x_2 + 3x_3 = -5 \quad (48c)$$

Now, let's arbitrarily modify this case with the addition of a few nonlinear terms, as follows:

#### Nonlinear Example:

$$3x_1 - 2x_2^2 = -7 \quad (49a)$$

$$-x_1x_2 + 3x_2 - 2x_3 = 9 \quad (49b)$$

$$-x_2x_3 + 3x_3 = -5 \quad (49c)$$

Our goal is to solve this set of equations for the solution vector  $\mathbf{x} = [x_1 \ x_2 \ x_3]^T$ .

First we should note that there is no simple way to know if a solution even exists and, if there are solutions, how many are there? To illustrate this point, let's work with eqn. (49) a little. In particular, solving eqn. (49a) for  $x_1$  and eqn. (49c) for  $x_3$  gives

$$x_1 = \frac{-7 + 2x_2^2}{3} \quad \text{and} \quad x_3 = \frac{-5}{3 - x_2} \quad (50)$$

Now, putting these expressions into eqn. (49b) gives

$$-\left(\frac{-7 + 2x_2^2}{3}\right)x_2 + 3x_2 - 2\left(\frac{-5}{3 - x_2}\right) = 9$$

or

$$-(-7 + 2x_2^2)(3 - x_2)x_2 + 9x_2(3 - x_2) + 30 = 27(3 - x_2) \quad (51)$$

which clearly shows that  $x_2$  can be found by determining the roots of a 4<sup>th</sup> order polynomial.

In general, a 4<sup>th</sup> order polynomial has four roots, but these are not necessarily real valued. In fact, the possible combinations are: 4 real roots, 2 real roots and a pair of complex conjugate roots, or two pairs of complex conjugate roots (i.e. no real roots). Thus, if we focus only on the real solutions to eqn. (49), this analysis says that we should expect four, two, or no solutions to the original nonlinear equations.

Well, further manipulation of eqn. (51) shows that there are two real roots for the particular set of coefficients in the current problem and, upon substitution into eqn. (50), we get two real solutions to the original nonlinear system:

$$\mathbf{x}_1 = \begin{bmatrix} -1.80843 \\ 0.887328 \\ -2.36667 \end{bmatrix} \quad \text{and} \quad \mathbf{x}_2 = \begin{bmatrix} 5.20859 \\ -3.36346 \\ -0.785736 \end{bmatrix}$$

These represent analytical solutions to the system of three nonlinear equations. However, for a large system of nonlinear equations, the above analytical manipulations are not practical, so we need to develop some numerical techniques for solving general problems of this type -- that is, problems of the form

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \quad \text{or} \quad \mathbf{A}(\mathbf{x}) \mathbf{x} = \mathbf{b}(\mathbf{x}) \quad (52)$$

### Linearized Iteration Method

We have already noted that the matrix formulation [2<sup>nd</sup> form in eqn. (52)], where the  $\mathbf{A}$  and  $\mathbf{b}$  coefficient matrices are expressed as functions of the solution vector  $\mathbf{x}$ , can be easily written in iterative form as

$$\mathbf{A}(\mathbf{x}^k) \mathbf{x}^{k+1} = \mathbf{b}(\mathbf{x}^k) \quad (53)$$

With this iterative strategy, the current estimate of the solution,  $\mathbf{x}^k$ , is used to estimate the coefficient matrices,  $\mathbf{A}(\mathbf{x}^k)$  and  $\mathbf{b}(\mathbf{x}^k)$ , and then these are used to write a **linear system** (with known  $\mathbf{A}$  and  $\mathbf{b}$ ) that can be solved for a new estimate of the solution vector,  $\mathbf{x}^{k+1}$ . This basic idea is referred to as the **Linearized Iteration Method** or the **Successive Substitution Method**.

Our sample problem, given by eqn. (49), can be expressed in this fashion with different matrix representations. Two obvious choices are

$$\text{Form \#1:} \quad \begin{bmatrix} 3 & -2x_2^k & 0 \\ -x_2^k & 3 & -2 \\ 0 & -x_3^k & 3 \end{bmatrix} \begin{bmatrix} x_1^{k+1} \\ x_2^{k+1} \\ x_3^{k+1} \end{bmatrix} = \begin{bmatrix} -7 \\ 9 \\ -5 \end{bmatrix} \quad (54a)$$

$$\text{Form \#2:} \quad \begin{bmatrix} 3 & -2x_2^k & 0 \\ 0 & 3-x_1^k & -2 \\ 0 & 0 & 3-x_2^k \end{bmatrix} \begin{bmatrix} x_1^{k+1} \\ x_2^{k+1} \\ x_3^{k+1} \end{bmatrix} = \begin{bmatrix} -7 \\ 9 \\ -5 \end{bmatrix} \quad (54b)$$

With a known estimate of the solution vector  $\mathbf{x}^k$ , the  $\mathbf{A}$  matrix can be evaluated, and the resultant “linear equations” can be easily solved for  $\mathbf{x}^{k+1}$ . The process is continued with successive substitution of the new estimate, to compute a better estimate, etc. until the maximum relative change in two successive estimates is less than some specified tolerance (here  $\text{tol} = 10^{-5}$ ).

This scheme was implemented in Matlab script file **nldemo1\_lesson6.m** (see Table 5). The code asks the user to select the form of eqn. (54) to use and to input an initial guess for  $x_2$  -- with  $x_2$  we can then compute initial values of  $x_1$  and  $x_3$  via eqn. (50). With the matrix formulation and initial solution vector fixed, the code simply executes the above algorithm, giving intermediate edit along the way (so that we can monitor the progress of the iterative process).

**Table 5 Listing of the nldemo1\_lesson6.m file (uses the Linearized Iteration Method).**

```
%
% NLDEMO1_LESSON6.M Solve nonlinear demo using the linearized iteration method
%
% This file computes the solution for a 3rd order nonlinear system via
% the linearized iteration (successive substitution) method. In this method,
% the coefficient matrix is a function of the solution vector. However, with
% a given guess for x, A(x) can be determined, and the solution of a linear
% system, A(x) * x = b, can be obtained. This process is continued until
% convergence (or we hit the max number of iterations).
%
% Note that NLDEMO2 and NLDEMO3... solve this same problem using different methods.
%
% File prepared by J. R. White, UMass-Lowell (last update: Nov. 2017)
%

clear all; close all; nfig = 0;

% set reasonable initial guess
x2 = input(' input a guess for x2: ');
if x2 == 3, x2 = 3.001; end % don't allow x2 = 3 as guess
x1 = (-7+2*x2*x2)/3; % satisfies eqn. 1
x3 = -5/(3-x2); % satisfies eqn. 3
xold = [x1 x2 x3]';

% select eqn form (there were two obvious forms for A(x))
form = menu('Select Equation Form', ...
            'Form #1', ...
            'Form #2', ...);

% start iteration loop
itmax = 30; it = 0; tol = 1e-5; emax = 1; n = length(xold);
fprintf(1,'\n Intermediate edit for NLDemo1 (Form #1i) \n',form);
while emax > tol && it <= itmax
    it = it+1;
    x = xold;

% set up A and b matrices given current guess and solve for new solution vector
switch form
    case 1
        A = [ 3 -2*x(2) 0; % coefficient matrix (form #1)
              -x(2) 3 -2;
              0 -x(3) 3];
    case 2
        A = [ 3 -2*x(2) 0; % coefficient matrix (form #2)
              0 3-x(1) -2;
              0 0 3-x(2)];
end
b = [-7 9 -5]'; % RHS vector (constant for this case)
xnew = A\b; % find solution vector

% calc & edit error (intermediate results)
emax = max(abs((xnew-xold)./xnew));
fprintf(1,' it = %3d max error = %8.3e \n',it,emax);
fprintf(1,' xnew xold \n');
for j = 1:n
    fprintf(1,' %10.5f %10.5f \n',xnew(j),xold(j));
end
%
```

```

        xold = xnew; % use current estimate as guess for next iteration
    end
%
% print final max relative error and iteration count
    fprintf(1,'\n Number of iterations to convergence = %3d\n',it);
    fprintf(1,' Max relative error at convergence = %8.3e\n',emax);
    if it >= itmax
        fprintf(1,' ***** WARNING -- Hit max number of iterations!!! *****\n');
    end
%
% end of file

```

Selected code output for  $x_2^0 = 1.0$  with Form #1 [eqn. (54a)] is:

```

>> nldemo1_lesson6
input a guess for x2: 1.0

Intermediate edit for NLDemo1 (Form #1)
it = 1      max error = 2.000e-01
    xnew      xold
-1.77778    -1.66667
 0.83333     1.00000
-2.36111    -2.50000
it = 2      max error = 7.960e-02
    xnew      xold
-1.83033    -1.77778
 0.90541     0.83333
-2.37925    -2.36111
it = 3      max error = 2.912e-02
    xnew      xold
-1.80229    -1.83033
 0.87979     0.90541
-2.36441    -2.37925

Deleted some output for brevity

it = 10      max error = 3.507e-05
    xnew      xold
-1.80844    -1.80841
 0.88734     0.88731
-2.36668    -2.36666
it = 11      max error = 1.335e-05
    xnew      xold
-1.80843    -1.80844
 0.88732     0.88734
-2.36667    -2.36668
it = 12      max error = 5.102e-06
    xnew      xold
-1.80843    -1.80843
 0.88733     0.88732
-2.36667    -2.36667

Number of iterations to convergence = 12
Max relative error at convergence = 5.102e-06

```

and similar output for  $x_2^0 = 1.0$  with Form #2 [eqn. (54b)] is:

```

>> nldemo1_lesson6
input a guess for x2: 1.0

Intermediate edit for NLDemo1 (Form #2)
it = 1      max error = 1.667e-01

```

```

      xnew      xold
-1.76190      -1.66667
 0.85714      1.00000
-2.50000      -2.50000
it = 2      max error = 7.143e-02
      xnew      xold
-1.81333      -1.76190
 0.91000      0.85714
-2.33333      -2.50000
it = 3      max error = 3.910e-02
      xnew      xold
-1.80204      -1.81333
 0.87576      0.91000
-2.39234      -2.33333

```

Deleted some output for brevity

```

it = 15      max error = 3.106e-05
      xnew      xold
-1.80843      -1.80844
 0.88732      0.88735
-2.36669      -2.36664
it = 16      max error = 1.720e-05
      xnew      xold
-1.80844      -1.80843
 0.88733      0.88732
-2.36666      -2.36669
it = 17      max error = 9.527e-06
      xnew      xold
-1.80843      -1.80844
 0.88733      0.88733
-2.36668      -2.36666

```

```

Number of iterations to convergence = 17
Max relative error at convergence = 9.527e-06

```

Here we see that both formulations converge to the correct solution,  $\mathbf{x}_1$ , but that the 2<sup>nd</sup> form takes a few more iterations to converge (17 vs. 12 iterations).

In an attempt to converge to the second solution,  $\mathbf{x}_2$ , a sequence of new initial guesses was made with  $\mathbf{x}_2^0$  becoming closer and closer to the analytical result of  $\mathbf{x}_2 = -3.36346$ . In all cases, however, the code converged to  $\mathbf{x}_1$  instead of  $\mathbf{x}_2$ . Only when  $\mathbf{x}_2^0$  was set to the actual analytical result to within the specified convergence criterion did the second solution  $\mathbf{x}_2$  emerge.

The behavior observed here is typical of arbitrary *one-point or fixed-point iteration schemes* as implemented in this example. The setup and solution strategy are relatively straightforward, but the iterative scheme is not guaranteed to converge and, when it does, there is no way to predict what solution will be found (if multiple solutions exist). Usually, if the initial guess is close to the actual solution, the iterative strategy will converge to that solution vector -- however, as we have seen here, this is not always the case. Also, because the selection of  $\mathbf{A}(\mathbf{x})$  is not unique in a given problem (here we had two obvious choices and others are available), different formulations can result -- and these can often have significantly different convergence properties (even to the point where one formulation may converge and another diverge). Thus, the variability and unpredictability of the linearized iteration strategy make this technique difficult to use within a general-purpose nonlinear equation solver. However, when applied on a case-specific basis for a

given problem, it can often be tailored to be an effective, easy-to-use solution methodology. This solution method is certainly not applicable for every case but, in my experience over the years, I have found the linearized iteration approach to be quite effective for many practical engineering problems that involve a system of mildly nonlinear equations (see the solution to **Motivation Problem #3** and the **parallel\_flow\_1 Case Study**, for example). Thus, the **linearized iteration method** is one technique that you should put in your toolbox for use as one possible option for solving nonlinear problems...

### Newton's Method

Probably the most common one-point approach for solving nonlinear equations is **Newton's Method**. We are already familiar with this technique for a single equation from our discussions in Lesson #5. By way of review for a single nonlinear equation, we start with a Taylor series expansion for the function at the point  $x_{i+1}$  written in terms of the function and its derivatives evaluated at point  $x_i$ , or

$$f(x_{i+1}) = f(x_i) + f'(x_i)(x_{i+1} - x_i) + O(h^2) \quad (55)$$

where  $O(h^2)$  says that the truncation error will be “on the order of  $h^2$ ,” with  $h = x_{i+1} - x_i$ . Now, we drop the error term and solve eqn. (55) for  $x_{i+1}$  and set  $f(x_{i+1}) = 0$ , since the goal here is to estimate the value of  $x$  such that  $f(x) = 0$ . Doing this gives

$$x_{i+1} = x_i - f'(x_i)^{-1}f(x_i) \quad (56)$$

which is the iteration equation for Newton's method for the case of a single nonlinear equation.

Now, of interest here, is finding the solution vector  $\mathbf{x}$  to a system of coupled nonlinear equations written in the form  $\mathbf{f}(\mathbf{x}) = \mathbf{0}$  -- that is, what is  $\mathbf{x}$  such that  $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ ? This is essentially the same problem as described above except we now have  $n$  nonlinear equations and  $n$  unknowns. Thus, we can perform the same type of manipulations as above, as long as we are careful with the notation and treatment of the Taylor series for the case of  $n$  independent variables. In particular, we can write the Taylor series expansion for each function, as

$$\begin{aligned} f_1(\mathbf{x}_{i+1}) &= f_1(\mathbf{x}_i) + \left. \frac{\partial f_1(\mathbf{x})}{\partial x_1} \right|_{\mathbf{x}_i} (x_{1,i+1} - x_{1,i}) + \left. \frac{\partial f_1(\mathbf{x})}{\partial x_2} \right|_{\mathbf{x}_i} (x_{2,i+1} - x_{2,i}) \\ &\quad + \cdots + \left. \frac{\partial f_1(\mathbf{x})}{\partial x_n} \right|_{\mathbf{x}_i} (x_{n,i+1} - x_{n,i}) + O(h^2) \\ f_2(\mathbf{x}_{i+1}) &= f_2(\mathbf{x}_i) + \left. \frac{\partial f_2(\mathbf{x})}{\partial x_1} \right|_{\mathbf{x}_i} (x_{1,i+1} - x_{1,i}) + \left. \frac{\partial f_2(\mathbf{x})}{\partial x_2} \right|_{\mathbf{x}_i} (x_{2,i+1} - x_{2,i}) \\ &\quad + \cdots + \left. \frac{\partial f_2(\mathbf{x})}{\partial x_n} \right|_{\mathbf{x}_i} (x_{n,i+1} - x_{n,i}) + O(h^2) \end{aligned}$$

etc. for  $n$  equations...

Using summation notation to treat the  $n$  first-derivative terms for the  $k^{\text{th}}$  function,  $f_k(\mathbf{x})$ , we can generalize the above expressions, as follows:

$$f_k(\mathbf{x}_{i+1}) = f_k(\mathbf{x}_i) + \sum_{\ell=1}^n \left. \frac{\partial f_k(\mathbf{x})}{\partial x_\ell} \right|_{\mathbf{x}_i} (x_{\ell,i+1} - x_{\ell,i}) \quad (57)$$

where we have truncated the 2<sup>nd</sup> and higher order terms (for convenience and since, in practice, we only keep up through the 1<sup>st</sup> order terms).

Now, the second term on the right hand side of eqn. (57) looks like a matrix times a vector -- recall that  $\mathbf{w} = \mathbf{A}\mathbf{z}$  is written in discrete form as

$$w_k = \sum_{\ell} a_{k\ell} z_{\ell} \quad (58)$$

Therefore, defining the **Jacobian matrix**,  $\mathbf{J}(\mathbf{x})$ , as

$$\mathbf{J}(\mathbf{x}) = \left[ \frac{\partial f_k(\mathbf{x})}{\partial x_\ell} \right] \quad (59)$$

and the increment vector on the  $i^{\text{th}}$  step,  $\mathbf{h}_i$ , as

$$\mathbf{h}_i = \mathbf{x}_{i+1} - \mathbf{x}_i = \begin{bmatrix} x_{\ell,i+1} - x_{\ell,i} \end{bmatrix} \quad (60)$$

eqn. (57) becomes

$$\mathbf{f}(\mathbf{x}_{i+1}) = \mathbf{f}(\mathbf{x}_i) + \mathbf{J}(\mathbf{x})|_{\mathbf{x}_i} \mathbf{h}_i \quad (61)$$

where we note that the Jacobian matrix is evaluated at state  $i$  (i.e. with vector  $\mathbf{x}_i$ ).

This matrix equation is of the same form as eqn. (55) for a single nonlinear function  $f(\mathbf{x})$ . As before, we solve eqn. (61) for  $\mathbf{x}_{i+1}$  and set  $\mathbf{f}(\mathbf{x}_{i+1}) = \mathbf{0}$ , since this represents the next estimate of the vector  $\mathbf{x}$  such that  $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ . Doing this gives

$$\mathbf{J}(\mathbf{x})|_{\mathbf{x}_i} \mathbf{h}_i = -\mathbf{f}(\mathbf{x}_i) \quad (62)$$

or

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{h}_i \quad \text{where} \quad \mathbf{h}_i = -\mathbf{J}(\mathbf{x})|_{\mathbf{x}_i}^{-1} \mathbf{f}(\mathbf{x}_i) = -\mathbf{J}(\mathbf{x})|_{\mathbf{x}_i} \backslash \mathbf{f}(\mathbf{x}_i) \quad (63)$$

where the last equality uses Matlab's backslash operator to actually solve for the increment in the  $\mathbf{x}$  vector on step  $i$  (instead of using the matrix inverse operator -- since we know that the LU decomposition procedure embedded within the backslash operation is much more efficient than taking the formal matrix inverse).

Well, eqn. (63) represents the iteration equation for Newton's method for a system of nonlinear equations. One starts with an initial guess  $\mathbf{x}_1$  for the full vector of unknowns, computes the function  $\mathbf{f}(\mathbf{x})$  and Jacobian  $\mathbf{J}(\mathbf{x})$  evaluated at the current guess, and then calculates  $\mathbf{h}_1$  and  $\mathbf{x}_2$  via eqn. (63). The solution vector  $\mathbf{x}_2$  is then used to compute  $\mathbf{x}_3$ ,  $\mathbf{x}_3$  is used to compute  $\mathbf{x}_4$ , etc. until the maximum relative change in successive iterates is less than some specified convergence criterion.

To demonstrate Newton's method, the above solution algorithm was implemented within Matlab script file **nldemo2\_lesson6.m** (see Table 6) for the same problem solved previously with the linearized iteration method. The basic code setup is similar to the **nldemo1\_lesson6.m** code

discussed previously (see Table 5 in previous subsection); however, within the iteration loop, we must compute  $\mathbf{f}(\mathbf{x}_i)$  and  $\mathbf{J}(\mathbf{x}_i)$  at each step before actually solving eqn. (63). These expressions are given explicitly as

$$f_1(\mathbf{x}) = 3x_1 - 2x_2^2 + 7 \quad (64a)$$

$$f_2(\mathbf{x}) = -x_1x_2 + 3x_2 - 2x_3 - 9 \quad (64b)$$

$$f_3(\mathbf{x}) = -x_2x_3 + 3x_3 + 5 \quad (64c)$$

and

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} \\ \frac{\partial f_3}{\partial x_1} & \frac{\partial f_3}{\partial x_2} & \frac{\partial f_3}{\partial x_3} \end{bmatrix} = \begin{bmatrix} 3 & -4x_2 & 0 \\ -x_2 & -x_1 + 3 & -2 \\ 0 & -x_3 & -x_2 + 3 \end{bmatrix} \quad (65)$$

where the components of the  $\mathbf{x}$  vector represent the current values at stage  $i$ .

**Table 6 Listing of the nldemo2\_lesson6.m script file (uses Newton's Method).**

```
%
% NLDEMO2_LESSON6.M      Solve nonlinear demo using Newton's method
%
% This file computes the solution for a 3rd order nonlinear system via
% Newton's method.  This method requires explicit evaluation of the nonlinear
% function and the matrix of first partial derivatives (Jacobian matrix).  With
% a given guess for x, f(x) and J(x) can be determined, and the solution of a
% linear system, J(x) * h = f(x), can be obtained to find the next estimate,
% x_i+1 = x_i - h.  This process is continued until convergence (or we hit the
% max number of iterations).
%
% Note that NLDEMO1 and NLDEMO3... solve this same problem using other methods...
%
% File prepared by J. R. White, UMass-Lowell (last update: Nov. 2017)
%

clear all; close all; nfig = 0;

%
% set a reasonable initial guess
x2 = input(' input a guess for x2: ');
if x2 == 3, x2 = 3.001; end % don't allow x2 = 3 as guess
x1 = (-7+2*x2*x2)/3; % satisfies eqn. 1
x3 = -5/(3-x2); % satisfies eqn. 3
xold = [x1 x2 x3]';

%
% start iteration loop
itmax = 30; it = 0; tol = 1e-5; emax = 1; n = length(xold);
fprintf(1,'\n Intermediate edit for NLDemo2 \n');
while emax > tol && it <= itmax
    it = it+1;
    x = xold;

%
% compute function vector using xold
f = [3*x(1)-2*x(2)*x(2)+7;
     -x(1)*x(2)+3*x(2)-2*x(3)-9;
     -x(2)*x(3)+3*x(3)+5];
```



```
% compute Jacobian matrix evaluated at xold
J = [ 3      -4*x(2)      0;
      -x(2)  -x(1)+3     -2;
      0      -x(3)     -x(2)+3];

%
% compute xnew
xnew = xold - J\f;

%
% calc & edit error (intermediate results)
emax = max(abs((xnew-xold)./xnew));
fprintf(1,' it = %3d      max error = %8.3e \n',it,emax);
fprintf(1,'          xnew          xold      \n');
for j = 1:n
    fprintf(1,' %10.5f    %10.5f    \n',xnew(j),xold(j));
end

%
xold = xnew; % use current estimate as guess for next iteration
end

%
% print final max relative error and iteration count
fprintf(1,'\n Number of iterations to convergence = %3d\n',it);
fprintf(1,' Max relative error at convergence = %8.3e\n',emax);
if it >= itmax
    fprintf(1,' ***** WARNING -- Hit max number of iterations!!! *****\n');
end

%
% end of file
```

The code outputs from **nldemo2\_lesson6** with  $x_2^0 = 1.0$  and  $x_2^0 = -2.0$  are shown below:

```
>> nldemo2_lesson6
input a guess for x2: 1.0

Intermediate edit for NLDemo2
it = 1      max error = 1.290e-01
      xnew      xold
-1.81905     -1.66667
 0.88571      1.00000
-2.35714     -2.50000
it = 2      max error = 5.868e-03
      xnew      xold
-1.80844     -1.81905
 0.88733      0.88571
-2.36666     -2.35714
it = 3      max error = 3.397e-06
      xnew      xold
-1.80843     -1.80844
 0.88733      0.88733
-2.36667     -2.36666

Number of iterations to convergence = 3
Max relative error at convergence = 3.397e-06
```

```
>> nldemo2_lesson6
input a guess for x2: -2.0

Intermediate edit for NLDemo2
it = 1      max error = 1.233e+01
      xnew      xold
14.84314      0.33333
-7.44118     -2.00000
 0.08824     -1.00000
it = 2      max error = 1.193e+00
```

```

      xnew      xold
      8.30225    14.84314
      -4.79255    -7.44118
      -0.45649     0.08824
it =   3      max error = 4.260e-01
      xnew      xold
      5.82192     8.30225
      -3.67252    -4.79255
      -0.70725    -0.45649
it =   4      max error = 1.095e-01
      xnew      xold
      5.24733     5.82192
      -3.38438    -3.67252
      -0.77988    -0.70725
it =   5      max error = 7.407e-03
      xnew      xold
      5.20878     5.24733
      -3.36357    -3.38438
      -0.78570    -0.77988
it =   6      max error = 4.091e-05
      xnew      xold
      5.20859     5.20878
      -3.36346    -3.36357
      -0.78574    -0.78570
it =   7      max error = 1.110e-09
      xnew      xold
      5.20859     5.20859
      -3.36346    -3.36346
      -0.78574    -0.78574

```

```

Number of iterations to convergence = 7
Max relative error at convergence = 1.110e-09

```

Clearly the 1<sup>st</sup> case converged to the  $\mathbf{x}_1$  result and the 2<sup>nd</sup> case converged on the  $\mathbf{x}_2$  solution vector from the analytical work discussed previously (see page 26). Notice also that both solutions converged quite quickly, with only 3 and 7 iterations, respectively (much faster, in general, than the linearized iteration method). Thus, for this case, Newton's method was relatively simple to implement and it works quite nicely!!!

One of the disadvantages of Newton's method is the need for explicit evaluation of the Jacobian matrix -- since, in many situations, this is not easy to compute. In these cases, we simply estimate the matrix of partial derivatives using a finite difference method (recall the discussion of the Secant Method in Lesson #5...). A good example of doing this is given in the Case Study in Section 12.3 in your numerical methods text by Chapra -- you should definitely check it out if you have not already done so...

### Matlab's *fsolve* Function

Before leaving this section on nonlinear equations, we should note that Matlab has several built-in functions for the solution of constrained and unconstrained nonlinear equations (many of these are associated with the Optimization Toolbox). One of the most often used functions for the solution of a set of unconstrained nonlinear equation is the *fsolve* function. Although the details of the methods implemented within *fsolve* are beyond the scope of this introductory course, the function is actually relatively easy to use -- based only on the knowledge and insight gained from our discussions in the previous subsections. The user must supply a function file to evaluate  $\mathbf{f}(\mathbf{x})$  for any given solution guess  $\mathbf{x}$  and, on option, one can either provide explicit code

to compute the Jacobian or let the code generate it internally using a finite difference approximation [via multiple calls to the user file that evaluates  $\mathbf{f}(\mathbf{x})$  for any  $\mathbf{x}$ ] -- the reader is encouraged to type *help fsolve* in Matlab's command window to obtain further information...

The best way to illustrate the use of *fsolve* is to use it within the context of a familiar example -- the same 3×3 nonlinear sample problem as solved previously. In particular, Table 7 gives a listing of the program **nldemo3\_lesson6.m**. In this case, *fsolve* does all the iterative calculations (which are hidden from the user) using the user-defined anonymous function that simply evaluates the nonlinear equations of interest [eqn. (64) for this example]. The default option is to compute the Jacobian internally, and this selection was retained here -- thus, keeping things very simple for the user.

As before we show the Matlab abbreviated output obtained with two different initial guesses:

```
>> nldemo3_lesson6
input a guess for x2: 1.0
```

```
Solution edit for NLDemo3
  x(1) =  -1.80843
  x(2) =   0.88733
  x(3) =  -2.36667
```

```
>> nldemo3_lesson6
input a guess for x2: -2.0
```

```
Solution edit for NLDemo3
  x(1) =   5.20859
  x(2) =  -3.36346
  x(3) =  -0.78574
```

Thus, with relatively little effort, we have solved the given nonlinear problem with the same results as computed previously. Personally, when I have a need to solve **unconstrained nonlinear problems**, Matlab's *fsolve* function is the method of choice for me...

**Table 7 Solution of sample nonlinear problem using Matlab's *fsolve* command.**

```
%
% NLDemo3_LESSON6.M Solve nonlinear demo using Matlab's fsolve function
%
% This file computes the solution for a 3rd order nonlinear system using
% Matlab's built-in fsolve function.
%
% Note that NLDemo1 and NLDemo2... solve this same problem using other methods...
%
% File prepared by J. R. White, UMass-Lowell (last update: Nov. 2017)
%

clear all; close all; nfig = 0;

%
% set reasonable initial guess
x2 = input('input a guess for x2: ');
if x2 == 3, x2 = 3.001; end % don't allow x2 = 3 as guess
x1 = (-7+2*x2*x2)/3; % satisfies eqn. 1
x3 = -5/(3-x2); % satisfies eqn. 3
xo = [x1 x2 x3]';
n = length(xo);
%
```

```
% call fsolve and edit solution
fx = @(x) [3*x(1)-2*x(2)*x(2)+7;
          -x(1)*x(2)+3*x(2)-2*x(3)-9;
          -x(2)*x(3)+3*x(3)+5];
x = fsolve(fx,xo);
fprintf(1,'\n Solution edit for NLDemo3 \n');
for j = 1:n
    fprintf(1,'    x(%1i) = %10.5f    \n',j,x(j));
end
%
% end of file
```

## Solution of the Motivation Problems

Now that we have reviewed several basic methods for solving linear and nonlinear equations, we should be able to solve the problems presented earlier to motivate the need for such techniques. In particular, several Matlab script and function files were written to solve the three problems posed earlier. Each problem is briefly discussed in the following subsections (please refer back, as needed, to the problem descriptions given at the beginning of this lesson):

### Problem 1 Solution

This problem deals with a 4-loop resistive circuit where we want to compute the current in each of the four loops. Kirchoff's voltage law leads to a series of four linear equations that we wrote in standard matrix form in eqns. (7) and (8) on page 3 of this lesson. To solve a problem of this type, we simply specify the particular resistor and voltage values, set up the **A** and **b** coefficient matrices, solve the linear system using Matlab's backslash operator (i.e.  $\mathbf{x} = \mathbf{A} \backslash \mathbf{b}$ ), and then edit the results of the problem. This sequence of steps was coded into the **resistive\_networks0.m** file as given in Table 8.

The screen output from running **resistive\_networks0** is given below:

```
>> resistive_networks0

Results for Resistive Networks #0
--> Current I1 = -1.35465 ma
--> Current I2 = -1.02740 ma
--> Current I3 = -1.26321 ma
--> Current I4 = -0.78963 ma
```

Thus, all the currents are negative (meaning that they really move opposite to the assumed direction as shown in the sketch on page 3).

**Table 8 Listing of the program to solve Problem 1.**

```
%
% RESISTIVE_NETWORKS0.M Find the current in a particular resistive network
%
% A particular resistive network is given along with the applied voltages. The
% goal here is simply to determine the currents in each path within the circuit.
%
% File prepared by J. R. White, UMass-Lowell (last update: Nov. 2017)
%

clear all, close all

%
% define model parameters (note that resistance is given in kohms and voltage
% in volts, so the current will be in milliamps -- I = V/R)
R1 = 1; R2 = 3.5; R3 = 4; R4 = 1.75; R5 = 1 ;
```

```

R6 = 1; R7 = 2; R8 = 1.5; R9 = 5;
V1 = 5; V2 = 5; V3 = 2.5; V4 = 2; V5 = 5;
%
% set up coefficient matrices (see development in Notes)
A = [ (R1+R2)    -R2    0    0
      -R2    (R2+R3+R4+R5)    0    -R5;
      0    0    (R6+R7+R8)    -R8;
      0    -R5    -R8    (R5+R8+R9)];
b = [V3-V1; -V2; -V3-V4; V4-V5];
%
I = A\b; % solve linear system
%
% edit results
fprintf('\n Results for Resistive Networks #0 \n ');
for i = 1:length(I)
    fprintf(' --> Current I%i = %8.5f ma \n ',i,I(i));
end
%
% end of problem

```

## Problem 2 Solution

Problem 2 deals with the geometry of a parabolic fin. The goal here is to compute the surface area of the fin as a function of the fin's length (since the amount of heat transfer is related to the heat transfer surface area). The expression for  $y(r)$ , which defines the coordinates of the outer surface of the fin, along with three geometry constraints, lead to a system of three linear equations for the equation constants in the  $y(r)$  expression -- and these are a function of the fin length,  $L$ . Once the coefficients that define  $y(r)$  are known, an integral expression can be evaluated to determine the fin surface area for each length. A sketch of the fin geometry and all the pertinent equations for this problem were developed on pages 4 and 5 of these notes (please review these as needed). Our job now is to actually solve the problem and to present the results.

In particular, the following algorithm was implemented into the **parabolic\_fin1.m** code (see Table 9 for the code listing):

1. Set up basic problem parameters ( $R_o$ ,  $H$ ,  $h$ , and range of  $L$  values)
2. Loop over number of  $L$  values
  - a. Setup coefficient matrices as defined in eqn. (12) on page 5
  - b. Solve system of equations to find the  $a_1$ ,  $a_2$ , and  $a_3$  coefficients for the  $y(r)$  expression
  - c. Use **integral** to evaluate the fin's surface area via eqn. (15) on page 5
3. Plot and tabulate the key results [i.e.  $y(r)$  and  $A$  for several  $L$  values]

The **parabolic\_fin1** code follows the above algorithm quite closely and it has lots of comments for good internal documentation -- thus, it is relatively easy to follow the code logic that was needed to solve this problem. Concerning the subject of this Lesson -- the solution of linear equations -- we simply used the  $\mathbf{x} = \mathbf{A} \backslash \mathbf{b}$  command to accomplish this step of the algorithm, and you should be quite comfortable by now with using Matlab's backslash operator for this purpose. Although the use of the **integral** routine to perform the numerical integration is new, a quick **help integral** in the Matlab command window will explain how this function is used -- it is really pretty simple to use and we will elaborate on the actual methods used within numerical integration routines like **integral** in Lesson #8 (if time permits). Note that, since the expression for the integrand was quite simple, we used an anonymous Matlab function to define this quantity (instead of writing a separate function file).

For the code output, the *fprintf* command was used to get a short table of results that summarize the equation constants and the fin surface area versus fin length. Finally, in making the plot of the fin geometry, we took a few extra steps to plot both positive and negative  $y(r)$  values, as well as the vertical lines at the end of the fins (Matlab's *hold on* and *hold off* commands were used here). This was done so that the fin geometries are easier to visualize.

**Table 9 Listing of the program to solve Problem 2.**

```
%
% PARABOLIC_FIN1.M Find the surface area of a parabolic fin
%
% Assume that a parabolic fin has a shape described by
%   y = a1 + a2*r + a3*r^2
% and it must satisfy three constraints
%   y(Ro) = H/2,   y(Ro+L) = h/2,   and   dy/dr at Ro+L = 0
% These three constraints can be used to find the equation coefficients
% for each value of L.
%
% Once these are known, one can evaluate the following expression for the
% surface area of the fin for each L:
%   A = 2*top surface area + tip area
%   A = 4*pi*int (r*sqrt(1+(a2+2*a3*r)^2) dr + 2*pi*(Ro+L)*h
% The integral goes from Ro to Ro+L. We will use the built-in integral routine
% to do these integrals. An anonymous function, f(r), is used to define the
% integrand for the integration.
%
% Finally, a plot and summary table of results are generated to describe
% the fin's geometry versus length.
%
% File prepared by J. R. White, UMass-Lowell (last update: Nov. 2017)
%

clear all, close all

%
% model constants
Ro = 5; % cylindrical tube's outside radius (cm)
H = 6; % height of fin at base of cylindrical tube (cm)
h = 1; % height of fin at the tip (cm)

%
% loop over different values of fin length, L
L = 2:2:10; NL = length(L); % set vector of fin lengths (cm)
Sarea = zeros(1,NL); % initialize storage for surface area
Coeff = zeros(3,NL); % initialize storage for coeff vector
for i = 1:NL

%
% set up coeff matrices and solve resultant linear system
RoL = Ro + L(i);
A = [1 Ro Ro^2; 1 RoL RoL^2; 0 1 2*RoL]; b = [H/2 h/2 0]';
x = A\b; Coeff(:,i) = x; % store solution vector for ith length

%
% find surface area for given L
a2 = x(2); a3 = x(3);
f = @(r) r.*sqrt(1 + (a2 + 2*a3*r).^2); % anonymous function for use in integral
Sarea(i) = 4*pi*integral(f,Ro,RoL) + 2*pi*RoL*h; % compute integral to find area
end

%
% plot and tabulate results
fprintf('\n Parabolic_Fin1: Summary Data for Geometry of Parabolic Fin \n\n');
fprintf(' Length (cm)   a1           a2           a3       Surface Area (cm^2) \n');
for i = 1:NL
    fprintf(' %6.2f %12.3e %12.3e %12.3e %13.4e \n ',L(i),Coeff(:,i),Sarea(i));
end

%
style = ['r- ' 'b--' 'g-.' 'c: ' 'm- ']; % 5 different plot styles
for i = 1:NL
    r = linspace(Ro,Ro+L(i),20); % need a new r vector for each length, L
```

```

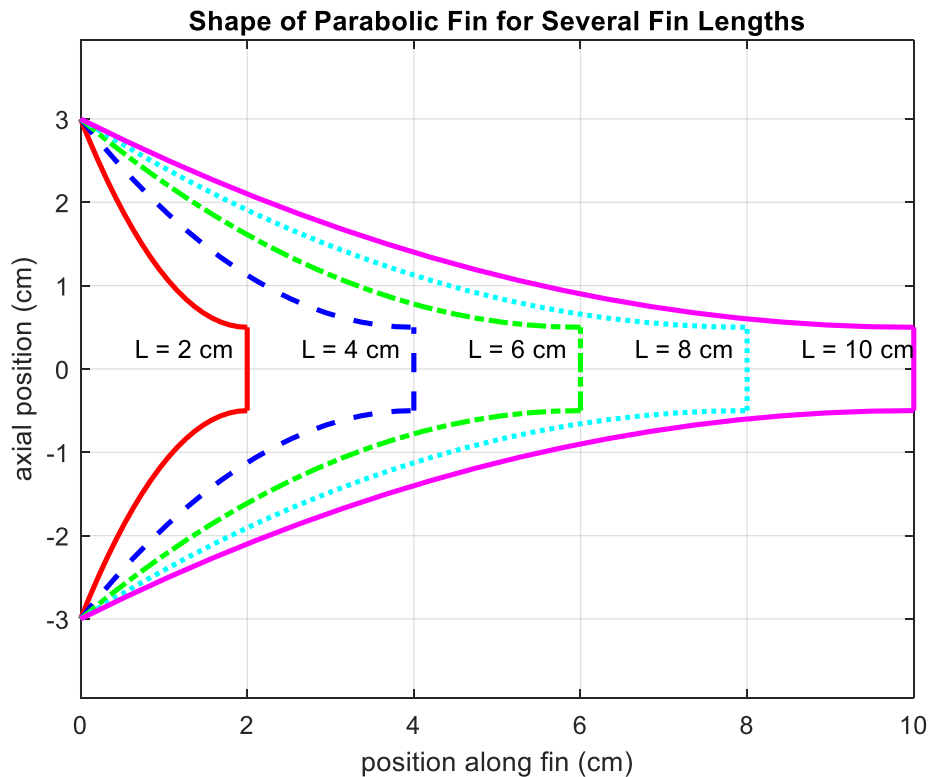
y = Coeff(1,i) + Coeff(2,i)*r + Coeff(3,i)*r.^2;
plot(r-Ro,y,style(i,:), 'LineWidth',2),hold on      % plots +y(r)
plot(r-Ro,-y,style(i,:), 'LineWidth',2)           % plots -y(r)
plot([L(i) L(i)],[-h/2 h/2],style(i,:), 'LineWidth',2) % plots vertical end
end
axis equal; grid % use axis equal to display proper aspect ratio for real geometry
title('Shape of Parabolic Fin for Several Fin Lengths')
xlabel('position along fin (cm)'),ylabel('axial position (cm)')
for i=1:NL, text(L(i)-1.35,0.25,['L = ',num2str(L(i)),' cm']); end
%
% end of problem

```

The graphical output from **parabolic\_fin1** is shown in Fig. 2 and the tabular output is given below:

Parabolic\_Fin1: Summary Data for Geometry of Parabolic Fin

Length (cm)	a1	a2	a3	Surface Area (cm <sup>2</sup> )
2.00	3.113e+01	-8.750e+00	6.250e-01	2.8914e+02
4.00	1.316e+01	-2.813e+00	1.563e-01	4.7515e+02
6.00	8.903e+00	-1.528e+00	6.944e-02	7.2456e+02
8.00	7.102e+00	-1.016e+00	3.906e-02	1.0301e+03
10.00	6.125e+00	-7.500e-01	2.500e-02	1.3889e+03



**Fig. 2 Fin geometry versus fin length.**

Note that, although there are no real surprises here (i.e. everything behaves as expected), this was not a particularly easy problem to solve. I have given this problem several times as a homework assignment in previous years and many students had lots of difficulty -- some with the integration routine, some with the plots, some with the use of the *fprintf* function, and some with the looping structure needed to parameterize the fin length. Thus, since previous students had so much difficulty, I decided to convert this problem into a demonstration problem instead of a HW assignment. Thus, you should take advantage to this change and make sure you have a good understanding of both the solution algorithm and the actual Matlab code needed to solve this problem -- since you may see something similar in your HW assignment for this lesson!!!

### Problem 3 Solution

In contrast to the first two motivation problems that involved linear equations, Problem 3 requires the solution of a system of nonlinear equations. This example models the steady state behavior of a simple rectangular fin with a temperature dependent heat transfer coefficient. The problem is summarized (including the equations to be implemented) on pages 6 – 8 of this set of notes, and the base linear version of the problem (with constant  $h$ ) is described in detail in the **fd\_intro.pdf** file from the Lesson #4 Notes.

The **Linearized Iteration Method** is the procedure employed to solve this particular nonlinear problem. The case-specific iterative algorithm follows:

1. Guess the discrete temperature profile (here we use a constant profile that represents a simple average between the base temperature,  $T_b$ , and the ambient temperature,  $T_\infty$ )
2. Compute  $h(T) \rightarrow h(x_i) \rightarrow h_i$  based on the current estimate of  $T_i$  (here we used the linear relationship that  $h(T) = c_1 + c_2 T$ , with  $c_1 = 200 \text{ W/m}^2\text{-C}$  and  $c_2 = 2 \text{ W/m}^2\text{-C}^2$ ) -- these are called  $a_1$  and  $a_2$  in the code.
3. Calculate  $m^2_i$  and the **A** and **b** coefficient matrices via eqns. (22) and (23) on page 7.
4. Solve the “linearized” system for a new estimate of the discrete temperature profile using Matlab’s backslash operator,  $\mathbf{T} = \mathbf{A} \backslash \mathbf{b}$
5. Compute the maximum relative change in the new and old temperature profiles -- and go back to Step #2 if the error is too large with the new guess as the current best estimate of **T**
6. Edit/plot desired results

The above algorithm was implemented into the **rect1d\_fin\_3.m** and **rect1d\_fin\_3a.m** files, as given in Table 10. The **rect1d\_fin\_3** script file is the main program that sets up the overall solution scheme, including specifying the base problem parameters, initiation of the iteration loop, solving of the linearized system, checking on convergence, and plotting of the converged temperature and heat transfer coefficient profiles along the length of the fin. Inside the iteration loop, the main program calls the **rect1d\_fin\_3a** function file to set up the **A** and **b** coefficient matrices given the current estimate of the discrete **h** vector. Note that, although the basic code within the **rect1d\_fin\_3a** function could have been included directly within the main program, the current two-subprogram format makes the overall logic and program flow more explicit and easier to follow. Note also that some intermediate edit is available for small problems, so that one can easily check the details of the iterative process to assure that things are working as designed (this is always a good idea to help in debugging your programs). For larger problems, when the number of equations exceeds some reasonable value, the additional edit is skipped.



**Table 10 Listing of the programs to solve Problem 3.**

```

%
% RECT1D_FIN_3.M   Introduction to FD Methods for Solving Nonlinear BVPs
%                  Heat Transfer Analysis of a Rectangular Fin Arrangement with
%                  a Temperature-Dependent Heat Transfer Coefficient (Nonlinear Eqns.)
%
% This file illustrates the finite difference (FD) method for solving nonlinear
% BVPs. We solve the same rectangular fin heat transfer problem that was treated
% previously in program RECT1D_FIN_2.M, except now we allow the heat transfer
% coefficient to vary with temperature. This makes the BVP nonlinear and, upon
% discretization, the system of coupled algebraic equations that results also
% becomes nonlinear. We will use the Linearized Iteration method (as discussed
% in the Lecture Notes) to solve this nonlinear problem.
%
% The basic iterative procedure is as follows:
%   1. guess the temperature profile (here let  $T(x)$  = constant for simplicity)
%   2. compute  $h(x)$  with the correlation given using the current estimate of  $T(x)$ 
%   3. set up the matrix balance equations and solve for new  $T(x)$ 
%   4. check convergence (go back to Step 2 if not converged)
% Hopefully this scheme will converge rather quickly (remember that this one-point
% iteration scheme is not guaranteed to converge). Let's see...
%
% The development of the FD equations are done in the Lecture Notes and the
% coefficient matrices for each new  $T(x)$  guess are constructed in RECT1D_FIN_3A.M.
%
% File prepared by J. R. White, UMass-Lowell (last update: Nov. 2017)
%
%
%   clear all, close all, nfig = 0;
%
% identify basic problem data
%   w = 1; % unit width of fin (m)
%   thk = 0.01; % fin thickness (m)
%   L = 0.05; % fin length (m)
%   Tb = 200; % fin base temperature (C)
%   Tinf = 30; % environment temperature (C)
%   k = 200; % fin thermal conductivity (W/m-C)
%   a1 = 200; a2 = 2; % constants in heat transfer coeff correlation
%
% define # of points and set x-vector
%   N = input(' Input number of unknowns in problem (N): ');
%   dx = L/N; x = dx:dx:L; x = x';
%
% set initial guess and start iterative loop
%   Told = 0.5*(Tb+Tinf)*ones(size(x));
%   itmax = 20; it = 0; tol = 1e-5; merr = 1;
%   while merr > tol && it <= itmax
%       h = a1 + a2*Told; % determine h(T) -> h(x)
%
% compute temperature profile for given h(x)
%   [A,b] = rect1d_fin_3a(h,Tb,Tinf,k,L,w,thk,N); % determine coeff matrices
%   Tnew = A\b; % find solution vector
%   it = it+1; % increment iteration counter
%   merr = max(abs(Tnew-Told)./Tnew); % compute max relative change
%
% print intermediate edit if the number of eqns is relatively small
%   if N < 15
%       fprintf('\n Intermediate edit for RECT1D_FIN_3.M \n');
%       fprintf(' it = %3d max error = %8.2e \n',it,merr);
%       fprintf(' Tnew Told h used \n');
%       fprintf(' (C) (C) (W/m^2-C) \n');
%       for j = 1:N
%           fprintf(' %8.2f %8.2f %8.2f \n',Tnew(j),Told(j),h(j));
%       end
%   end
%
%   Told = Tnew; % set Told (i.e. the guess) for the next pass
%
% end % end of iteration loop
%

```

```

% print max relative error and iteration count
    fprintf(1,'\n\n Number of iterations to convergence = %3d\n',it);
    fprintf(1,' Max relative error at convergence = %8.2e\n',merr);
%
% add on the known base temperature to the solution vector [do same for h(x)]
    xp = [0; x];    Tp = [Tb; Tnew];    hb = a1 + a2*Tb;    hp = [hb; h];
%
% plot temperature profile
    nfig = nfig+1;    figure(nfig)
    plot(xp*100,Tp,'r--','LineWidth',2),grid
    title('T(x) Profile in Rectangular Fin (Nonlinear Case)')
    xlabel('Spatial Position (cm)'), ylabel('Temperature (^oC)')
%
% plot heat transfer coeff profile
    nfig = nfig+1;    figure(nfig)
    plot(xp*100,hp,'r--',xp*100,500*ones(size(hp)),'b-','LineWidth',2),grid
    title('h(x) Profile in Rectangular Fin (Nonlinear Case)')
    xlabel('Spatial Position (cm)'), ylabel('Heat Transfer Coeff (W/m^2-^oC)')
    legend('variable h', 'constant h')
%
% end of problem

%
% RECT1D_FIN_3A.M    Function file to compute the coefficient matrices (A & b)
%                    for the nonlinear rectangular fin heat transfer problem.
%
% This routine is called from RECT1D_FIN_3.M.  Several variables are passed
% into this routine via input arguments, but only the convective heat transfer
% coeff is position dependent -- this is the only term that varies with each call
% to this function.  The desired A and b coefficient matrices are returned.
%
% File prepared by J. R. White, UMass-Lowell (last update:  Nov. 2017)
%
%
%    function [A,b] = rect1d_fin_3a(h,Tb,Tinf,k,L,w,thk,N)
%
% initialize variables
    dx = L/N;    dx2 = dx*dx;
    A = zeros(N,N);    b = zeros(N,1);
%
% compute m2 variable (this is space dependent because h is a function of space)
    P = 2*w + 2*thk;    % perimeter
    Ac = w*thk;    % cross section area (conduction area)
    m2 = h*P/(k*Ac);    % parameter in derived equations (see notes)
%
% compute coeff matrices
% node 1
    A(1,1) = -(2 + m2(1)*dx2);    A(1,2) = 1;    b(1) = -m2(1)*dx2*Tinf - Tb;
% interior nodes
    for i = 2:N-1
        A(i,i-1) = 1;    A(i,i) = -(2 + m2(i)*dx2);    A(i,i+1) = 1;
        b(i) = -m2(i)*dx2*Tinf;
    end
% node N
    A(N,N-2) = 1;    A(N,N) = -(2*m2(N)*dx2 + 1 + 2*h(N)*dx/k);
    b(N) = -(2*h(N)*dx/k + 2*m2(N)*dx2)*Tinf;
%
% end of routine

```

The results from the **rect1d\_fin\_3** program are discussed in the next few pages.

First we show a portion of the intermediate results for the case of  $N = 10$ . As seen below, the initial constant guess for the temperature profile leads to a constant  $h$  on the first iteration. However, with  $h = 430 \text{ W/m}^2\text{-C}$  everywhere, the resultant temperature profile, **Tnew**, varied from 200 C at the base of the fin to about 129 C at the fin's tip. With this profile as **Told** on the 2<sup>nd</sup> iteration, a better estimate of the real space-dependent heat transfer coefficient was computed, and this was then used to get a new estimate of the actual temperature profile. This process was continued for 7 iterations, until the maximum relative change between two successive iterations was less than the specified tolerance of  $1 \times 10^{-5}$ . Careful inspection of the code output reveals that the Linearized Iteration Method worked exactly as expected, and that, for this case, it converged rather quickly to a consistent solution.

```
>> rect1d_fin_3
```

```
Input number of unknowns in problem (N): 10
```

```
Intermediate edit for RECT1D_FIN_3.M
```

```
it = 1      max error = 3.83e-01
  Tnew      Told      h used
  (C)       (C)       (W/m^2-C)
186.45     115.00     430.00
174.60     115.00     430.00
164.33     115.00     430.00
155.51     115.00     430.00
148.05     115.00     430.00
141.87     115.00     430.00
136.91     115.00     430.00
133.11     115.00     430.00
130.43     115.00     430.00
128.84     115.00     430.00
```

```
Intermediate edit for RECT1D_FIN_3.M
```

```
it = 2      max error = 4.17e-02
  Tnew      Told      h used
  (C)       (C)       (W/m^2-C)
184.69     186.45     572.91
171.61     174.60     549.21
160.50     164.33     528.65
151.13     155.51     511.01
143.32     148.05     496.10
136.94     141.87     483.74
131.85     136.91     473.82
127.99     133.11     466.22
125.28     130.43     460.86
123.68     128.84     457.68
```

```
Some edit deleted for brevity
```

```
Intermediate edit for RECT1D_FIN_3.M
```

```
it = 6      max error = 3.30e-05
  Tnew      Told      h used
  (C)       (C)       (W/m^2-C)
184.83     184.83     569.67
171.89     171.90     543.79
160.90     160.91     521.81
151.64     151.64     503.28
143.92     143.92     487.84
137.60     137.60     475.21
```

```

132.57      132.58      465.16
128.75      128.76      457.52
126.07      126.08      452.16
124.49      124.49      448.99

Intermediate edit for RECT1D_FIN_3.M
it = 7      max error = 5.34e-06
  Tnew      Told      h used
  (C)      (C)      (W/m^2-C)
184.83      184.83      569.67
171.89      171.89      543.79
160.90      160.90      521.81
151.64      151.64      503.28
143.92      143.92      487.84
137.60      137.60      475.20
132.58      132.57      465.15
128.75      128.75      457.51
126.07      126.07      452.15
124.49      124.49      448.98

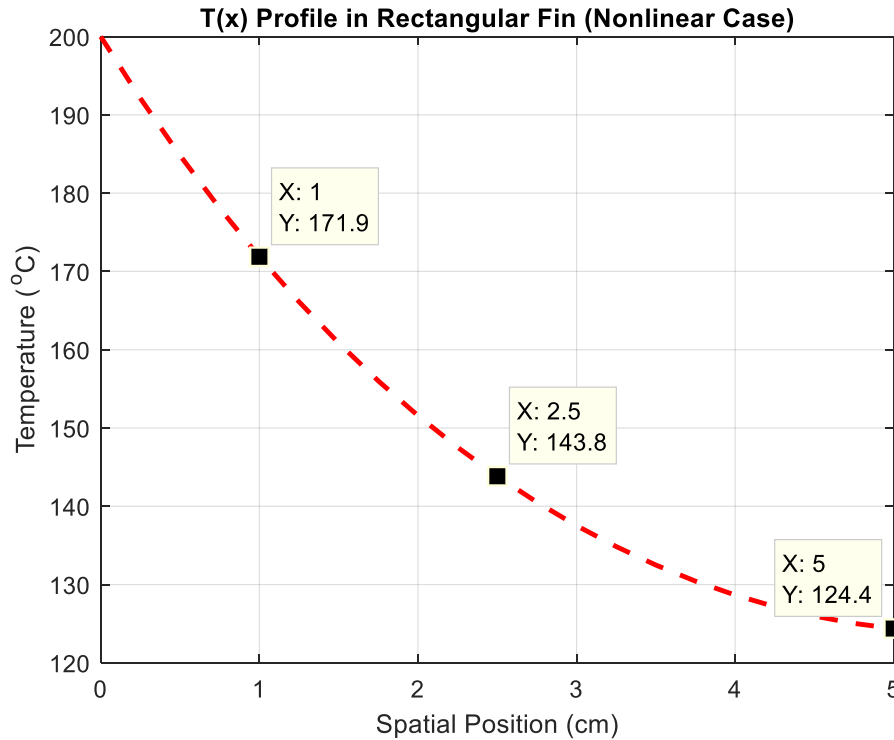
Number of iterations to convergence = 7
Max relative error at convergence = 5.34e-06

```

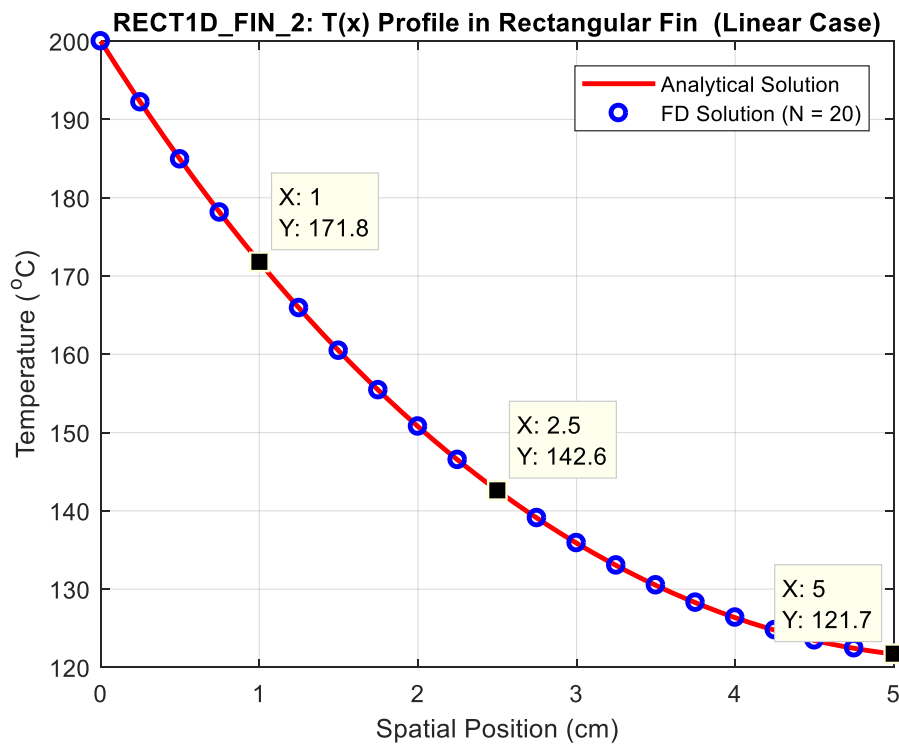
Once the basic iterative procedure had been validated, the code was rerun with  $N = 20$  and the graphical results for  $T(x)$  are given below in Fig. 3. Here we see the expected temperature profile, with  $T(x)$  monotonically decreasing from  $T_b$  at the base of the fin to the fin's tip temperature (which will be some value above  $T_\infty$ ). For comparison purposes, we have also included, in Fig. 4, the results from the linear version of this same problem (with  $h = \text{constant}$ ) as discussed previously in Lesson #4 (see the **fd\_intro.pdf** documentation file and the **rect1d\_fin\_2.m** code). Clearly, the two profiles are quite similar and, to facilitate a more quantitative comparison, several discrete temperature values are annotated directly on the plots in both Figs. 3 and 4 (the annotation was made using the **Data Cursor tool** that is available within any standard Matlab figure window). Using the data given, we see that the two profiles differ in the worst case by under 3 % (less than 2-3 °C).

To investigate why the profiles are so similar -- when one case uses  $h = f(T)$  and the other has  $h = \text{constant}$  -- a plot of the spatial dependence of the heat transfer coefficient for both cases was also generated as shown in Fig. 5. Clearly there is a rather strong spatial dependence for the nonlinear case. However, the key here is that the linear approximation actually used a suitable average value of  $h$  ( $h = 500 \text{ W/m}^2\text{-C}$ ) and, with a reasonable average  $h$ , we would expect to get a reasonable estimate of the real  $T(x)$  profile using a simple linear model.

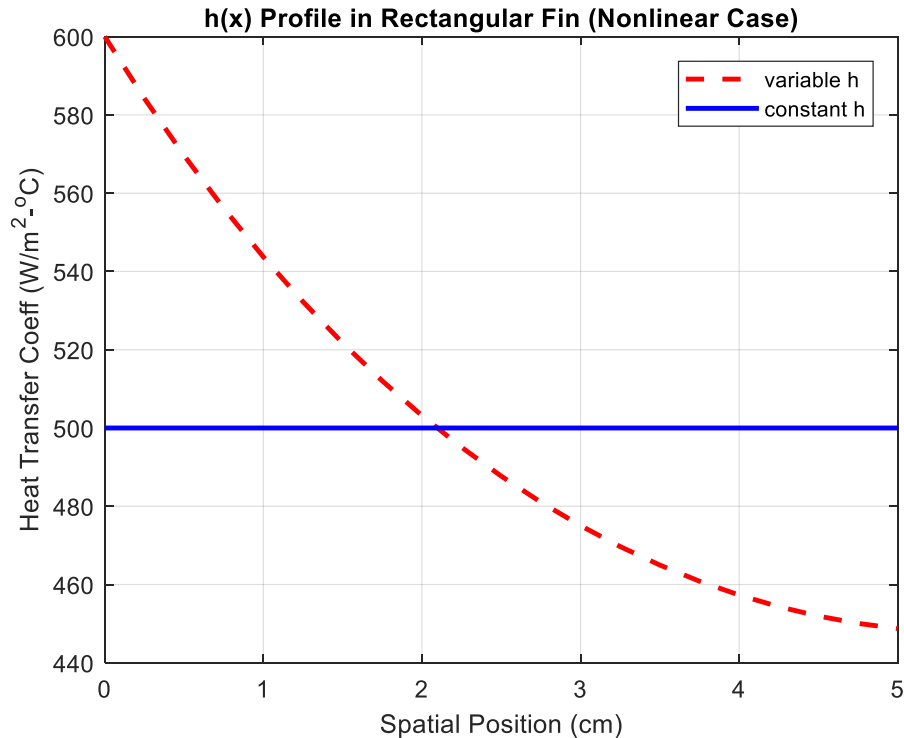
**Note:** Since the general mathematics and numerical solutions associated with a linear approximation to an engineering problem is so much easier to work with, we will often first generate a linear solution to a problem in the early design phase. This allows us to get a good understanding of the basic physics and the interrelationships among several key design variables -- allowing us to make some preliminary design decisions. Then, once the focus of the design problem has been narrowed, a more complete nonlinear model is used to do the final design and performance analyses. Thus, both linear and nonlinear models of the same system are often simulated to help us fully understand the system under study. This was the approach taken here -- with the goal of gaining a lot of insight about the solution techniques for linear vs. nonlinear



**Fig. 3**  $T(x)$  profile for the nonlinear version of the rectangular fin problem.



**Fig. 4**  $T(x)$  profile for the linear version of the rectangular fin problem.



**Fig. 5  $h(x)$  profiles for the nonlinear and linear versions of the rectangular fin problem.**

systems and, hopefully, in the process of studying this simple fin heat transfer problem, you have also gained some further knowledge and understanding of this area of study (this should definitely help you out when you take your Heat Transfer course...).

### Summary and Additional Applications

Well, this completes our brief overview of several solution techniques for both hand and computer solution of linear and nonlinear systems. You should have a good understanding of the “language” associated with linear systems and some insight into several methods for solving both linear and nonlinear systems. Within Matlab, we have found that the backslash operator will handle most linear problems and that the *fsolve* command is one possible tool for working with unconstrained nonlinear systems -- and you should have a good understanding of the basic algorithms used within these powerful tools. In addition, from an applications perspective, the setup and solution of the motivation problems should be quite illustrative of the types of problems that can be addressed and the actual procedures needed to solve a given problem.

To give some further experience with these techniques, I have also included two additional worked-out examples that show how to solve some interesting engineering problems involving both linear and nonlinear systems within the areas of chemical reaction stoichiometry and fluid mechanics, and a third example that discusses, in more detail, the relationship between diagonal dominance and the iterative solution of linear systems. These examples are available as separate pdf files, as follows:

**reaction\_eqns.pdf -- Reaction Stoichiometry**

**parallel\_flows1.pdf -- A Two-Pipe Parallel Flow System**

**conv\_demo1.pdf -- On the Convergence of Iterative Methods**

After you have finished your reading assignment for this lesson (Chapters 8 – 12 in Chapra and these Lecture Notes) and after reviewing the above additional examples, you should be ready to do HW #6 (see **hw6xxx.pdf**). This homework involves several questions that require you to set up and solve a series of problems involving both linear and/or nonlinear systems. As before, I prefer that you collect any hand calculations/manipulations for developing the pertinent models, the Matlab m-files, the resultant plots and/or tabular results, and a brief description of the results of each problem in a separate solution packet for each HW problem -- keeping things well organized will make life easier for you and for me...

Well, good luck with HW #6 -- I think you will find the problems to be both interesting and informative!!!