Chapter 2: Iterative methods for linear systems

2.1 Conceptual overview

In this chapter, we will develop iterative methods for finding approximate solutions to systems of linear algebraic equations. Such systems arise as part of numerical modeling and simulation in a wide range of scientific disciplines and real-world application areas. Often the size of these systems is extremely large, making them prohibitively slow and computationally intensive to solve using direct methods such as Gauss elimination, or its variants. Partial differential equation models routinely lead to large linear systems that must be solved many times throughout the simulation process. Thus, it is critically important to devise efficient numerical methods for solving such systems, and computer algorithms for their implementation.

This chapter introduces a class of iterative methods that are relatively straightforward to implement, and that work well for certain types of large linear systems. Such methods start from some initial guess of the solution, followed by a process of successively computing improved estimates. We introduce and illustrate the essential ideas using a system of 3 equations in 3 unknowns. Consider, for example, the system

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3 \end{aligned} \tag{1}$$

where the a_{ij} coefficients and the right hand sides b_i are given constants, and the x_i are unknowns to be determined. In matrix form, this system can be written as

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{2}$$

with

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \quad \text{and} \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{3}$$

Here is a specific 3×3 example we will use later to illustrate various iterative methods

$$\begin{aligned} &3x_1 - 0.1x_2 - 0.2x_3 = 7.85 \\ &0.1x_1 + 7x_2 - 0.3x_3 = -19.3 \\ &0.3x_1 - 0.2x_2 + 10x_3 = 71.4 \end{aligned} \quad \Rightarrow \quad \begin{bmatrix} 3 & -0.1 & -0.2 \\ 0.1 & 7 & -0.3 \\ 0.3 & -0.2 & 10 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 7.85 \\ -19.3 \\ 71.4 \end{bmatrix} \tag{4}$$

The exact solution of this system is: $x_1 = 3,\ x_2 = -2.5,\ x_3 = 7$.

We will consider the following three methods for solving such systems

- Jacobi method
- Gauss-Seidel method
- Successive Over-Relaxation (SOR) method

2.2 Jacobi method

This is one of the simplest methods to understand conceptually and to implement algorithmically. The basic idea is to solve each equation for only the diagonal unknown (say, x_i), while treating all the other x_j's as known from the previous iteration. For example, the system (1) can be rewritten as

$$\begin{aligned} x_1 &= \frac{b_1 - a_{12}x_2 - a_{13}x_3}{a_{11}}, \quad a_{11} \neq 0 \\ x_2 &= \frac{b_2 - a_{21}x_1 - a_{23}x_3}{a_{22}}, \quad a_{22} \neq 0 \\ x_3 &= \frac{b_3 - a_{31}x_1 - a_{32}x_2}{a_{33}}, \quad a_{33} \neq 0 \end{aligned} \tag{5}$$

If we let $(x_1^{(k)}, x_2^{(k)}, x_3^{(k)})$ denote the solution estimate at the kth iterate, then the Jacobi approximation for the next iterate has the form

$$\begin{aligned} x_1^{(k+1)} &= \frac{b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)}}{a_{11}}, \quad a_{11} \neq 0 \\ x_2^{(k+1)} &= \frac{b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)}}{a_{22}}, \quad a_{22} \neq 0 \\ x_3^{(k+1)} &= \frac{b_3 - a_{31}x_1^{(k)} - a_{32}x_2^{(k)}}{a_{33}}, \quad a_{33} \neq 0 \end{aligned} \tag{6}$$

Starting from any initial guess for the solution (i.e., at $k = 0$), this formula provides an iterative procedure for computing successive approximations. Later in this chapter we will discuss stopping criteria, and ways to determine whether the process is converging to the solution.

Example: Use the Jacobi method to find the approximate solution of the 3×3 system given in (4).

Solution:
The given system of equations is

$$\begin{aligned} 3x_1 - 0.1x_2 - 0.2x_3 &= 7.85 \\ 0.1x_1 + 7x_2 - 0.3x_3 &= -19.3 \\ 0.3x_1 - 0.2x_2 + 10x_3 &= 71.4 \end{aligned}$$

which can be rewritten as

$$\begin{aligned} x_1 &= \frac{7.85 + 0.1x_2 + 0.2x_3}{3} \\ x_2 &= \frac{-19.3 - 0.1x_1 + 0.3x_3}{7} \\ x_3 &= \frac{71.4 - 0.3x_1 + 0.2x_2}{10} \end{aligned}$$

For the initial guess, we will keep things simple and let $(x_1, x_2, x_3) = (0, 0, 0)$.

Iteration 1: Simply plug the initial guess into the right side and get

$$x_1 = \frac{7.85 + 0 + 0}{3} \approx 2.617$$

$$x_2 = \frac{-19.3 + 0 + 0}{7} \approx -2.757$$

$$x_3 = \frac{71.4 + 0 + 0}{10} = 7.14$$

The solution estimate is: $x_1 = 2.617, x_2 = -2.757, x_3 = 7.14$

Iteration 2: Plug the solution estimate from Iteration 1 into the right side and get

$$x_1 = \frac{7.85 + 0.1(-2.757) + 0.2(7.14)}{3} \approx 3.001$$

$$x_2 = \frac{-19.3 - 0.1(2.617) + 0.3(7.14)}{7} \approx -2.489$$

$$x_3 = \frac{71.4 - 0.3(2.617) + 0.2(-2.757)}{10} \approx 7.006$$

The solution estimate is: $x_1 = 3.001, x_2 = -2.489, x_3 = 7.006$

In this way, the iteration process is repeated till a solution of the desired acuracy is obtained. Notice that even after only two iterations, the solution estimate is quite close to the exact solution, suggesting that the method is converging fairly fast for this problem. As we will discuss later, a key reason for the fast convergence is the favorable properties of the coefficient matrix of the system (see equation 4).

2.3 Gauss-Seidel method

This method is set up in a very similar way to the Jacobi method, but with one important difference: When solving for the diagonal unknown $x_i^{(k+1)}$ the Jacobi method uses the previous iterate for all the other unknowns, namely $x_j^{(k)}$. But in the Gauss-Seidel method, the most recently available values are used for the other unknowns. Thus, for example, if we solve the system (1) sequentially, starting from the first equation, the Gauss-Seidel approximation for the next iterate (going from k to $k+1$) has the form

$$\begin{aligned} x_1^{(k+1)} &= \frac{b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)}}{a_{11}}, \quad a_{11} \neq 0 \\ x_2^{(k+1)} &= \frac{b_2 - a_{21}x_1^{(k+1)} - a_{23}x_3^{(k)}}{a_{22}}, \quad a_{22} \neq 0 \\ x_3^{(k+1)} &= \frac{b_3 - a_{31}x_1^{(k+1)} - a_{32}x_2^{(k+1)}}{a_{33}}, \quad a_{33} \neq 0 \end{aligned} \tag{7}$$

Comparing this with system (6) for Jacobi interations, we notice that while the equation for $x_1^{(k+1)}$ is the same, it is different for $x_2^{(k+1)}$ and $x_3^{(k+1)}$. The difference is that each new $x_i^{(k+1)}$

is used immediately after it is computed, when solving for the remaining unknowns.

Example: Use the Gauss-Seidel method to find the approximate solution of the same 3×3 system in (4).

Solution:
As before, we can rewrite the given system as

$$x_1 = \frac{7.85 + 0.1x_2 + 0.2x_3}{3}$$
$$x_2 = \frac{-19.3 - 0.1x_1 + 0.3x_3}{7}$$
$$x_3 = \frac{71.4 - 0.3x_1 + 0.2x_2}{10}$$

We keep the initial guess the same as before: $(x_1, x_2, x_3) = (0, 0, 0)$.

Iteration 1: $x_1 = \frac{7.85 + 0 + 0}{3} \approx 2.617$

$$x_2 = \frac{-19.3 - 0.1(2.617) + 0}{7} \approx -2.795 \quad \text{(using the new value of } x_1\text{)}$$

$$x_3 = \frac{71.4 - 0.3(2.617) + 0.2(-2.795)}{10} \approx 7.006 \quad \text{(with new values for } x_1,\ x_2\text{)}$$

The solution estimate is: $x_1 = 2.617, x_2 = -2.795, x_3 = 7.006$

Iteration 2: Following a similar strategy, where new values are used as soon as they become available, we have

$$x_1 = \frac{7.85 + 0.1(-2.795) + 0.2(7.006)}{3} \approx 2.991$$

$$x_2 = \frac{-19.3 - 0.1(2.991) + 0.3(7.006)}{7} \approx -2.500$$

$$x_3 = \frac{71.4 - 0.3(2.991) + 0.2(-2.5)}{10} \approx 7.000$$

The solution estimate after iteration 2 is: $x_1 = 2.991, x_2 = -2.5, x_3 = 7$

In general, the Gauss-Seidel method converges faster than the Jacobi method. However, convergence issues are a bit more theoretically nuanced and we will address certain aspects of it near the end of this chapter.

2.4 Successive Over-Relaxation (SOR) method

The SOR method is, essentially, a variant that is designed to accelerate the convergence of the Gauss-Seidel method. A useful conceptual way to understand the SOR strategy is as follows:

- Suppose $\mathbf{x}^{(k)}$ is any approximate solution of $\mathbf{Ax} = \mathbf{b}$ at iteration k.
- Let $\mathbf{x}^{(k+1)}$ denote the Gauss-Seidel solution estimate for the next iteration.
- Then, $\mathbf{e}^{(k+1)} = \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}$ can be viewed as the "correction" or improvement produced by Gauss-Seidel when going from iteration k to $k+1$. The vector $\mathbf{e}^{(k+1)}$ provides the direction in which we must move the approximate solution $\mathbf{x}^{(k)}$ in order to improve it.
- Since the direction $\mathbf{e}^{(k+1)}$ improves the solution, the key idea behind the SOR method is to move $\mathbf{x}^{(k)}$ a bit farther in that direction, in order to get a greater improvement.

Accordingly, the SOR formula for going from iteration k to $k+1$ is

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \omega \left(\mathbf{x}_{gs}^{(k+1)} - \mathbf{x}^{(k)}\right) \tag{8}$$

where $0 < \omega < 2$ is a constant parameter value chosen by the user, and $\mathbf{x}_{gs}^{(k+1)}$ denotes the Gauss-Seidel solution estimate at $(k+1)$. Notice that $\omega = 1$ corresponds to a purely Gauss-Seidel process. Typically, ω is taken to be larger than 1, since that corresponds to over-relaxation, or moving farther in the Gauss-Seidel direction. However, there are situations where one might want to choose $\omega \in (0, 1)$, which corresponds to under-relaxation, or making a smaller correction than the corresponding Gauss-Seidel iterate. Unfortunately, there is no simple strategy for determining the optimal value of ω, as it depends on the properties of the linear system under consideration. In practice, one often uses heuristics or trial and error mthods to choose ω.

Another point to note is that while (8) is useful for conceptual understanding, the SOR iteration formula is more commonly written as

$$\begin{aligned}
\mathbf{x}^{(k+1)} &= (1-\omega)\mathbf{x}^{(k)} + \omega\, \mathbf{x}_{gs}^{(k+1)} \\
&= (1-\omega) \begin{bmatrix} x_1^{(k)} \\ x_2^{(k)} \\ x_3^{(k)} \end{bmatrix} + \omega \begin{bmatrix} \frac{b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)}}{a_{11}} \\ \frac{b_2 - a_{21}x_1^{(k+1)} - a_{23}x_3^{(k)}}{a_{22}} \\ \frac{b_3 - a_{31}x_1^{(k+1)} - a_{32}x_2^{(k+1)}}{a_{33}} \end{bmatrix}
\end{aligned} \tag{9}$$

Example: Use the SOR method to find the approximate solution of the same 3×3 system that we previously solved with the Gauss-Seidel method [see (4)].

Solution:

We keep the initial guess the same as before: $(x_1^{(0)}, x_2^{(0)}, x_3^{(0)}) = (0, 0, 0)$.

Suppose $\omega = 1.25$.

Iteration 1: Applying the SOR formula, we have:

$$x_1^{(1)} = -0.25x_1^{(0)} + 1.25\left(\frac{7.85 + 0.1x_2^{(0)} + 0.2x_3^{(0)}}{3}\right) = 1.25\left(\frac{7.85}{3}\right) \approx 3.271$$

$$x_2^{(1)} = -0.25x_2^{(0)} + 1.25\left(\frac{-19.3 - 0.1x_1^{(1)} + 0.3x_3^{(0)}}{7}\right) =$$

$$1.25\left(\frac{-19.3 - 0.1(3.271)}{7}\right)$$

$$\approx -3.505$$

$$x_3^{(1)} = -0.25x_3^{(0)} + 1.25\left(\frac{71.4 - 0.3x_1^{(1)} + 0.2x_2^{(1)}}{10}\right)$$

$$= 1.25\left(\frac{71.4 - 0.3(3.271) + 0.2(-3.505)}{10}\right) \approx 8.715$$

The solution estimate is: $\mathbf{x}^{(1)} = [3.271,\ -3.505,\ 8.715]^T$

Iteration 2: Following a similar process using $\mathbf{x}^{(1)} = [3.271,\ -3.505,\ 8.715]^T$

$$x_1^{(2)} = -0.25(3.271) + 1.25\left[\frac{7.85 + 0.1(-3.505) + 0.2(8.715)}{3}\right] \approx 3.033$$

$$x_2^{(2)} = -0.25(-3.505) + 1.25\left[\frac{-19.3 - 0.1(3.033) + 0.3(8.715)}{7}\right] \approx -2.158$$

$$x_3^{(2)} = -0.25(8.715) + 1.25\left[\frac{71.4 - 0.3(3.033) + 0.2(-2.158)}{10}\right] \approx 6.579$$

The solution estimate after iteration 2 is: $\mathbf{x}^{(2)} = [3.033,\ -2.158,\ 6.579]^T$

We notice that after two iterations the SOR estimate is worse than that from the other two methods. This is, of course, for the particular choice of ω we used in this example. Are there other choices of ω that would work better? Clearly $\omega = 1$ does, since it reduces SOR to the Gauss-Seidel method, whose results were seen in the previous example. There may be other values that work better, and we invite the reader to explore a few others.

2.5 Stopping criteria

In practice, using iterative methods requires monitoring the progress of the computed solution, and deciding when to terminate the process. There are two common approaches to doing this: (1) compute the change in the solution after each iteration; (2) conpute the residual.

The change in the solution is the vector $\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}$, whose magnitude, or norm, can be taken as $\max |x_i^{(k+1)} - x_i^{(k)}|$. It is common practice to scale this and express it in the from of

a relative change. Then, a stopping test would check for the following condition

$$\max_i \left| \frac{x_i^{(k+1)} - x_i^{(k)}}{x_i^{(k+1)}} \right| < \tau \tag{10}$$

where τ is some user-specified error tolerance. An algorithm based on this type of test would exit when the solution updates drop below the user-specified threshold. For convergent processes, this approach is adequate.

A second approach for stopping test is based on the magnitude of the residual vector, which is defined as

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

Clearly, $\mathbf{r} = \mathbf{0}$ for the exact solution $\mathbf{x}$. For this reason, its magnitude is a good measure for how close the approximate solution $\mathbf{x}^{(k)}$ is to $\mathbf{x}$. Thus, after each iteration we compute

$$|\mathbf{r}^{(k+1)}| = |\mathbf{b} - \mathbf{A}\mathbf{x}^{(k+1)}| \tag{11}$$

and exit when $|\mathbf{r}^{(k+1)}| < \tau$ for some specified threshold τ. Although this strategy involves computing a matrix-vector product, the residual does give a reliable estimate of how well the iterative solution approximates the right answer.

2.6 Convergence pointers

In this section we will briefly address certain practically useful aspects related to convergence of the iterative methods discussed here. In keeping with the goals and spirit of this book, we will not attempt to provide a comprehensive treatment of convergence theory.

For linear systems of the form $\mathbf{A}\mathbf{x} = \mathbf{b}$, the properties of matrix $\mathbf{A}$ play a critical role in the convergence behavior. These properties not only affect the speed of convergence, but also determine whether the method will converge at all. By far, the most favorable property for $\mathbf{A}$ to have is diagonal dominance. Mathematically, this requires each row i of $\mathbf{A}$ to satisfy

$$|a_{ii}| \geq \sum_{j \neq i} |a_{ij}|$$

Strict diagonal dominance requires that at least one row satisfies: $|a_{ii}| > \sum_{j \neq i} |a_{ij}|$. All the iterative methods discussed in this chapter are guaranteed to converge, starting from any initial guess, for diagonally dominant linear systems. Intuitively, one way to understand why diagonal dominance is important is to consider the extreme case: suppose $\mathbf{A}$ is a purely diagonal matrix. Then, looking at the iterative method formulation in equations (5)-(7), we can see that the exact solution is found in one iteration. Essentially, the linear system is fully decoupled, and every unknown is independent of all the other unknowns. On the other hand, when $\mathbf{A}$ is diagonally dominant (but not purely diagonal), the unknowns are coupled, but not too strongly. Thus, the diagonal dominance condition captures some approximate measure of the permissible strength of coupling between unknowns.

We note that although diagonal dominance is sufficient, it is not a necessary condition for convergence. Some iterative methods may converge in certain cases when the matrix $\mathbf{A}$ is not diagonally dominant. In fact, there are examples in which the Jacobi method converges but the Gauss-Seidel method does not, and also examples in which the reverse occurs. Other useful properties of $\mathbf{A}$ that often arise in convergence analysis include symmetry, positive definiteness, and sparsity. We will discuss each of these, when needed, in the context of applications and examples.

Moving on to the topic of convergence rate, it depends on the properties of matrix $\mathbf{A}$ and on the specific iterative method under consideration. In general, each iterative method can be written in the form $\mathbf{x}^{(k+1)} = \mathbf{B}\mathbf{x}^{(k)} + \mathbf{c}$, where $\mathbf{B}$ is the iteration matrix, $\mathbf{c}$ is a constant vector, and superscripts (k) denote iteration number. The form of $\mathbf{B}$ and its properties depend on the choice of iterative method. It can be shown that the eigenvalues of $\mathbf{B}$ govern the rate of convergence of the iterative method. In fact, the convergence rate is a function of the spectral radius (eigenvalue of largest magnitude) of $\mathbf{B}$, which must be less than 1 to guarantee convergence. In general, the iteration matrix for the Gauss-Seidel method has smaller spectral radius than for the Jacobi method, which explains why the Gauss-Seidel method converges faster. For the interested reader, we provide a quick overview of convergence theory in the boxed supplement below.

Brief overview of convergence theory
(Optional, but interesting, reading!)

Consider the linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$.
Let us split $\mathbf{A}$ and write it as a sum of 3 matrices

$$\mathbf{A} = \mathbf{D} - \mathbf{L} - \mathbf{U}$$

where $\mathbf{D}, \mathbf{L}$ and $\mathbf{U}$ are the diagonal, lower triangular, and upper triangular parts of $\mathbf{A}$. Then, the linear system is

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

From this perspective, our iterative methods can be written as

$$\begin{aligned} \text{Jacobi:} \quad & \mathbf{D}\mathbf{x}^{(k+1)} = (\mathbf{L} + \mathbf{U})\mathbf{x}^{(k)} + \mathbf{b} \\ \text{Gauss-Seidel:} \quad & (\mathbf{D} - \mathbf{L})\mathbf{x}^{(k+1)} = (\mathbf{U})\mathbf{x}^{(k)} + \mathbf{b} \end{aligned} \tag{12}$$

Or, for the purpose of theoretical analysis, we rewrite this as

$$\begin{aligned} \text{Jacobi:} \quad & \mathbf{x}^{(k+1)} = \mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})\mathbf{x}^{(k)} + \mathbf{D}^{-1}\mathbf{b} \\ \text{Gauss-Seidel:} \quad & \mathbf{x}^{(k+1)} = (\mathbf{D} - \mathbf{L})^{-1}(\mathbf{U})\mathbf{x}^{(k)} + (\mathbf{D} - \mathbf{L})^{-1}\mathbf{b} \end{aligned} \tag{13}$$

Notice that both iterative formulas have the form: $\boxed{\mathbf{x}^{(k+1)} = \mathbf{B}\mathbf{x}^{(k)} + \mathbf{c}}$
where $\mathbf{B}$ is the iteration matrix, $\mathbf{c}$ is a constant vector.

Notice, also, that the exact solution $\mathbf{x}$ satisfies $\mathbf{x} = \mathbf{B}\mathbf{x} + \mathbf{c}$

Thus, we can subtract and get $\boxed{\mathbf{e}^{(k+1)} = \mathbf{B}\,\mathbf{e}^{(k)}}$
where $\mathbf{e}^{(k+1)} = \mathbf{x}^{(k+1)} - \mathbf{x}$ denotes the error after $k+1$ iterations.

Let $\mathbf{e}^{(0)}$ be the initial error. Then, by recursive application we have

$$\mathbf{e}^{(k+1)} = \mathbf{B}^{k+1}\,\mathbf{e}^{(0)}$$

Thus, for convergence, we want $\mathbf{B}^{k+1} \to \mathbf{0}$, which requires the spectral radius $\rho(\mathbf{B}) < \mathbf{1}$.

Furthermore, this expression also shows exactly how the convergence rate is related to $\rho(\mathbf{B})$. For example, suppose we want to estimate the number of iterations it will take to reduce the initial error by a factor of 10. Let M be the number of iterations needed. Then,

$$\begin{aligned}
\left|\mathbf{e}^{(M)}\right| &\leq \frac{\left|\mathbf{e}^{(0)}\right|}{10} \\
\Rightarrow \quad \left|\mathbf{B}^{M}\right|\left|\mathbf{e}^{(0)}\right| &\leq \frac{\left|\mathbf{e}^{(0)}\right|}{10} \\
\Rightarrow \quad |\rho(\mathbf{B})|^{M} &\leq \frac{1}{10} \\
\therefore \quad M &\approx \frac{\ln(1/10)}{\ln(\rho(\mathbf{B}))}
\end{aligned}$$

Notice that $\rho(\mathbf{B}) \to \mathbf{0} \Rightarrow \mathbf{M} \to \mathbf{0}$, and $\rho(\mathbf{B}) \to \mathbf{1} \Rightarrow \mathbf{M} \to \infty$.

Since $\mathbf{e}^{(k+1)} = \mathbf{B}\,\mathbf{e}^{(k)}$, we have $|\mathbf{e}^{(k+1)}| \leq \rho(\mathbf{B})|\mathbf{e}^{(k)}|$. Thus, the convergence rate is strongly dependent on the magnitude of $\rho(\mathbf{B})$, since at each iteration the error decreases by $\rho(\mathbf{B})$.

In closing, we note that there are many other types of iterative methods that are widely used in numerical modeling and simulation. In particular, gradient descent methods such as conjugate gradients, and Krylov subspace methods are popular for more general linear systems. For the interested reader we recommend the references [**??**].

2.7 Sample Python codes

Implementing the iterative methods discussed in this chapter using Python is quite straightforward. We provide sample code segments that show how to do this for the following example

$$\begin{aligned} -4\,x_1 + 3\,x_2 + 2\,x_3 &= 1 && (14)\\ 2\,x_1 + 3\,x_2 + 1\,x_3 &= -2 && (15)\\ 2\,x_1 + 3\,x_2 + 4\,x_3 &= 2 && (16) \end{aligned}$$

We will solve this system for the unknown vector $[x_1, x_2, x_3]^T$ using each of the three iterative methods, starting from the intial guess $[0, 0, 0]^T$. An error tolerance of 10^{-5} will be used, and the maximum permissible number of iterations will be 100. In the code listings that follow, the linear system is input in the form of a single 3×4 matrix, which consists of the 3×3 coefficient matrix, augmented with the right hand side vector.

Listing 1: Jacobi method

```
def column(m, c):
  return [m[i][c] for i in range(len(m))]

def height(m):
  return len(m)

def jacobi(m, x0=None, eps=1e-5, max_iteration=100):
  n  = height(m)
  x0 = [0] * n if x0 == None else x0
  x1 = [None] * n

  for k in range(max_iteration):
    for i in range(n):
      s = sum(-m[i][j] * x0[j] for j in range(n) if i != j)
      x1[i] = (m[i][n] + s) / m[i][i]
    if all(abs(x1[i]-x0[i]) < eps for i in range(n)):
      return x1
    x0, x1 = x1, x0
  raise ValueError('Solution␣does␣not␣converge')

if __name__ == '__main__':
  m = [[-4,3,2,1], [2,3,1,-2], [2,3,4,2]]
  print(jacobi(m))
```

Listing 2: Gauss-Seidel method

```
def column(m, c):
  return [m[i][c] for i in range(len(m))]

def height(m):
  return len(m)

def gauss_seidel(m, x0=None, eps=1e-5, max_iteration=100):
  n  = height(m)
  x0 = [0] * n if x0 == None else x0
  x1 = x0[:]

  for k in range(max_iteration):
    for i in range(n):
      s = sum(-m[i][j] * x1[j] for j in range(n) if i != j)
      x1[i] = (m[i][n] + s) / m[i][i]
    if all(abs(x1[i]-x0[i]) < eps for i in range(n)):
      return x1
    x0 = x1[:]
  raise ValueError('Solution does not converge')

if __name__ == '__main__':
  m = [[-4,3,2,1], [2,3,1,-2], [2,3,4,2]]
  print(gauss_seidel(m))
```

Listing 3: SOR method

```
def column(m, c):
  return [m[i][c] for i in range(len(m))]

def height(m):
  return len(m)

def sor(m, w=1.2, x0=None, eps=1e-5, max_iteration=100):
  n  = height(m)
  x0 = [0] * n if x0 == None else x0
  x1 = x0[:]

  for k in range(max_iteration):
    for i in range(n):
      s = sum(-m[i][j] * x1[j] for j in range(n) if i != j)
      x1[i] = w*(m[i][n]+s)/m[i][i] + (1-w)*x0[i]
    if all(abs(x1[i]-x0[i]) < eps for i in range(n)):
      return x1
    x0 = x1[:]
  raise ValueError('Solution␣does␣not␣converge')

if __name__ == '__main__':
  m = [[-4,3,2,1], [2,3,1,-2], [2,3,4,2]]
  print(sor(m))
```

Exercises

1. Solve the following system using the Gauss-Seidel method, starting from some suitable initial guess.

$$\begin{aligned} 11x_1 - 2x_2 + 3x_3 + 2x_4 &= 1 \\ -3x_1 - 21x_2 + 4x_3 - 4x_4 &= 0 \\ -4x_2 + 30x_3 + 5x_4 &= 0 \\ -6x_1 - x_2 + 20x_4 &= 1 \end{aligned}$$

2. Solve the following system using the SOR method, starting from the initial guess $[0, 0, 0, 0]^T$ and with an error tolerance of 10^{-5}. Try different values of ω and see if you can find one that converges faster than Gauss-Seidel (i.e., $\omega = 1$).

$$\begin{aligned} 9x_1 - 3x_2 + 3x_4 &= 10 \\ -2x_1 + 15x_2 + 2x_3 - 3x_4 &= 20 \\ -2x_2 + 10x_3 + 5x_4 &= -1 \\ -x_2 + 3x_3 + 14x_4 &= -2 \end{aligned}$$

3. Write a Python program that implements the Jacobi, Gauss-Seidel and SOR methods for solving each of the following linear systems. Start from any reasonable initial guess, and assume an error tolerance of 10^{-5}. For the SOR method, explore a few different values of ω and see how the convergence rate varies. Summarize your observations in the form of a brief written discussion.

   (a)

$$a_{i,j} = \begin{cases} 4\,i\,, & \text{if } j = i \text{ and } i = 1, 2, ..., 16 \\ -1\,, & \text{if } \begin{cases} j = i+1 & \text{and } i = 1, 2, 3, 5, 6, 7, 9, 10, 11, 13, 14, 15 \\ j = i-1 & \text{and } i = 2, 3, 4, 6, 7, 8, 10, 11, 12, 14, 15, 16 \\ j = i+4 & \text{and } i = 1, 2, ..., 12 \\ j = i-4 & \text{and } i = 5, 6, ..., 16 \end{cases} \\ 0\,, & \text{otherwise} \end{cases}$$

   and $\mathbf{b} = (1.902207, 1.051143, 1.175689, 3.480083, 0.819600, -0.264419, -0.412789, 1.175689, 0.913337, -0.150209, -0.264419, 1.051143, 1.966694, 0.913337, 0.819600, 1.902207)^T$

(b)

$$a_{i,j} = \begin{cases} 2\,i\,, & \text{if } j = i \text{ and } i = 1, 2, ..., 40 \\ -1\,, & \text{if } \begin{cases} j = i+1 & \text{and } i = 1, 2, ..., 39 \\ j = i-1 & \text{and } i = 2, 3, ..., 40 \end{cases} \\ 0\,, & \text{otherwise} \end{cases}$$

and $b_i = 1.5\,i - 6$, for $i = 1, 2, ..., 40$.

(c)

$$a_{i,j} = \begin{cases} 2\,i\,, & \text{if } j = i \text{ and } i = 1, 2, ..., 80 \\ 0.5\,i\,, & \text{if } \begin{cases} j = i+2 & \text{and } i = 1, 2, ..., 78 \\ j = i-2 & \text{and } i = 3, 4, ..., 80 \end{cases} \\ 0.25\,i\,, & \text{if } \begin{cases} j = i+4 & \text{and } i = 1, 2, ..., 76 \\ j = i-4 & \text{and } i = 5, 6, ..., 80 \end{cases} \\ 0\,, & \text{otherwise} \end{cases}$$

and $b_i = \pi$, for $i = 1, 2, ..., 80$

4. Construct an example of a linear system that does NOT satisfy the diagonal dominance condition, whose exact solution you know. [Note: It is easy to construct linear systems with known solution, because you can choose any matrix and exact solution you want, and multiply them to find the corresponding right hand side vector.]

   4.1 Use the Gauss-Seidel method to try and solve your linear system.

   4.2 At each iteration, compute and monitor the error in your approximate solution (compared to the exact solution).

   4.3 Observe the number of iterations needed for convergence.

   4.4 Vary your initial guess and repeat the above explorations.

   4.5 Summarize your observations in the form of a brief written discussion.

5. The goal of this exercise is to compare and contrast the behavior of different error indicators used in iterative solution methods. Two commonly used indicators, based on the magnitude of the iterative update when going from k to $k+1$ are: (i) the absolute error, or $\max_i |x_i^{(k+1)} - x_i^{(k)}|$, and (ii) the relative error, or $\max_i \left| \dfrac{x_i^{(k+1)} - x_i^{(k)}}{x_i^{(k)}} \right|$.

Pick an example of a linear system whose exact solution you know. Solve your system using both the Jacobi, and the Gauss-Seidel method. Compute and record the absolute error and the relative error at each iteration, together with the true error, which can be computed using the known exact solution. Observe how each error indicator behaves compared to the true error. A simple way to do this is by plotting error versus iteration number for each type of error. Write a brief paragraph summarizing your observations.

Chapter 3: Nonlinear algebraic systems

3.1 Conceptual overview

Let us begin with an illustration. A data scientist wants to fit a regression model to the data set shown in Fig. **??**. A model of the form $y = ae^{bx} + c$ is selected, due to the strong curvature seen in the plot. Here a, b, c are parameters whose optimal values must be determined in order to produce the best fit. Suppose the optimization process leads to the following system of equations that must be solved for determining the best values of a, b, c [1]

$$\begin{aligned} a\, e^{5070\, b} + c &= 69 \\ a\, e^{9060\, b} + c &= 75 \\ a\, e^{26900\, b} + c &= 84 \end{aligned} \tag{17}$$

There is no direct algebraic process for solving such systems, as the unknowns are nonlinearly coupled. The usual process for solving such systems is through iterative linearization, as we will learn in this chapter.

The need to solve nonlinear algebraic systems arises in a vast range of applications across many disciplines. Before we proceed to solution methods, let us consider one more example. The radio navigation of ships with the help of land-based tracking stations was widely used in the past. One example is LORAN, which stands for Long Range Navigation. In this navigation system, two (or more) widely-spaced transmitting stations on the coast send out synchronized, pulsed signals, which the ship receives. Using these signals, the ship determines its difference in distance from the two transmitting stations. Once the difference in distance is known, it identifies the ship's position to be somewhere on a hyperbola, every point of which is at a constant difference in distance from the associated pair of transmitting stations.

Assume the surface of the sea is the two-dimensional x-y plane (see Fig. 3). Then, the time difference between a pair of tracking signals locates the ship on a unique hyperbola. By using a third transmitting station, its signals can be paired with one of the other two to locate the ship on another unique hyperbola. Its exact location will then be at a point of intersection of the two hyperbolas. In this situation, we seek solutions in the form of (x, y) pairs that satisfy a quadratic system such as, for example

$$\frac{x^2}{186^2} - \frac{y^2}{300^2 - 186^2} = 1 \tag{18}$$

$$\frac{(y-500)^2}{279^2} - \frac{(x-300)^2}{500^2 - 279^2} = 1 \tag{19}$$

[1] Note this system is for illustration pusposes only, and will not yield the correct optimal values. A least squares optimization strategy to find the best values of a, b, c would yield a more complicated system of three nonlinear equations.

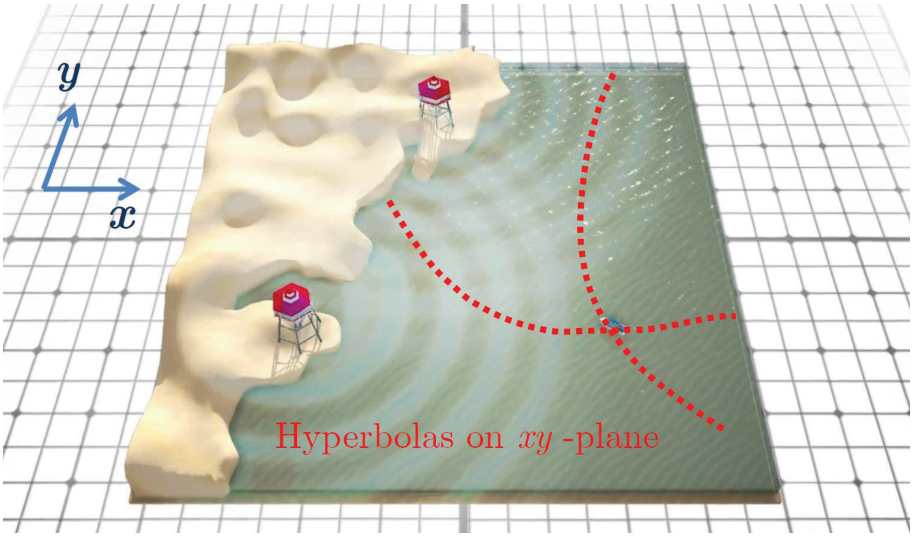

Figure 1: Long Range Navigation, or LORAN

Nonlinear algebraic systems can be generally written in the form

$$\begin{aligned} f_1(x_1, x_2, \ldots, x_n) &= 0 \\ f_2(x_1, x_2, \ldots, x_n) &= 0 \\ &\vdots \\ f_n(x_1, x_2, \ldots, x_n) &= 0 \end{aligned} \tag{20}$$

where each f_i is some (generally nonlinear) function of the n unknowns $x_1, x_2, \ldots, x_n$. It is often convenient to express this in vector form

$$\mathbf{F}(\mathbf{x}) = \mathbf{0}$$

where the vector function $\mathbf{F}(\mathbf{x}) = [f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_n(\mathbf{x})]^T$, with $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$. On the right hand side is an n-vector of zeroes.

In certain special cases it may be possible to solve a system like (20) exactly through a series of algebraic operations. But in the general case, it is usually necessary to use iterative strategies to try and find approximate solutions. In fact, general nonlinear systems can be quite complex, and it may not be known a priori whether solutions exist, and if they do, how many. Furthermore, even if a solution is known to exist, there is no guarantee that it can be found.

3.2 Newton method

Consider the general nonlinear system

$$\mathbf{F}(\mathbf{x}) = \mathbf{0} \tag{21}$$

with

$$\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$$

and

$$\mathbf{F}(\mathbf{x}) = \begin{bmatrix} f_1(x_1, x_2, \ldots, x_n) \\ f_2(x_1, x_2, \ldots, x_n) \\ \vdots \\ f_n(x_1, x_2, \ldots, x_n) \end{bmatrix}$$

For a concrete example we will use

$$\begin{aligned} x_2^2 - \sin(x_1 x_3) - 4 &= 0 \\ x_1^3 - 4(x_2 + x_3)^2 + \cos(x_3) + 2 &= 0 \\ e^{x_1 + x_2} - 5x_2 + x_3^2 - 4 &= 0 \end{aligned} \tag{22}$$

In this example, $n = 3$ and

$$\begin{aligned} f_1(x_1, x_2, x_3) &= x_2^2 - \sin(x_1 x_3) - 4 \\ f_2(x_1, x_2, x_3) &= x_1^3 - 4(x_2 + x_3)^2 + \cos(x_3) + 2 \\ f_3(x_1, x_2, x_3) &= e^{x_1 + x_2} - 5x_2 + x_3^2 - 4 \end{aligned} \tag{23}$$

The Newton method for solving $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ is an iterative process that essentially transforms the nonlinear system into a sequence of linear systems. It does this by approximating $\mathbf{F}$ to be a linear function of $\mathbf{x}$, similar to how we use tangent line approximations for funtions of one variable. As with tangent line approximations, the idea is to pick an $\mathbf{x}$-value and write a linear expression for $\mathbf{F}$ that is locally valid. Accordingly, suppose we want to linearly approximate $\mathbf{F}(\mathbf{x})$ at $\mathbf{x} = \mathbf{x}^{(k)}$. Then, we can write

$$\mathbf{F}(\mathbf{x}) \approx \mathbf{F}(\mathbf{x^{(k)}}) + \mathbf{J}(\mathbf{x^{(k)}})(\mathbf{x} - \mathbf{x^{(k)}})$$

where $\mathbf{J}(\mathbf{x}^{(k)})$ denotes the $n \times n$ Jacobian matrix of $\mathbf{F}$ evaluated at $\mathbf{x}^{(k)}$. The Jacobian matrix is

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix} \tag{24}$$

To solve $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ we set the linearized form to $\mathbf{0}$ and find $\mathbf{x}$

$$\mathbf{F}(\mathbf{x}^{(k)}) + \mathbf{J}(\mathbf{x}^{(k)})(\mathbf{x} - \mathbf{x}^{(k)}) = \mathbf{0} \quad \Rightarrow \quad \mathbf{x} = \mathbf{x}^{(k)} - \ [\mathbf{J}(\mathbf{x}^{(k)})]^{-1}\mathbf{F}(\mathbf{x}^{(k)})$$

Since the resulting value of $\mathbf{x}$ is an iterative approximation for the solution, it is generally denoted $\mathbf{x}^{(k+1)}$.

To summarize, the Newton method starts from an initial guess $\mathbf{x}^{(0)}$, and applies the iterative formula

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - [\mathbf{J}(\mathbf{x}^{(k)})]^{-1}\mathbf{F}(\mathbf{x}^{(k)}), \qquad k = 0, 1, 2, \ldots \tag{25}$$

to compute successive updates. For practical implementation, it is common to rewrite the iterative formula as

$$[\mathbf{J}(\mathbf{x}^{(k)})](\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}) = -\mathbf{F}(\mathbf{x}^{(k)}) \qquad \text{OR} \qquad [\mathbf{J}(\mathbf{x}^{(k)})]\, \Delta\mathbf{x}^{(k)} = -\mathbf{F}(\mathbf{x}^{(k)}) \tag{26}$$

since it reduces the problem to a series of linear algebraic solves, and the matrix inversion can be avoided. Equation (26) is solved for the correction $\Delta\mathbf{x}^{(k)}$, and the next approximation is computed as $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \Delta\mathbf{x}^{(k)}$.

At each iteration, the magnitude of the correction vector $\Delta\mathbf{x}^{(k)}$ can also be used as a stopping test. When $|\Delta\mathbf{x}^{(k)}| < \epsilon$, for some prescribed error tolerance ϵ, the iterations can be terminated.

Algorithm for the Newton method

1. Pick an initial guess $\mathbf{x}^{(0)}$ and a stopping tolerance ϵ.
2. For $k = 0, 1, 2, \ldots$
   (a) Compute $\mathbf{F}(\mathbf{x}^{(k)})$ and $\mathbf{J}(\mathbf{x}^{(k)})$.
   (b) Find $\Delta\mathbf{x}^{(k)}$ by solving
   $$[\mathbf{J}(\mathbf{x}^{(k)})]\, \Delta\mathbf{x}^{(k)} = -\mathbf{F}(\mathbf{x}^{(k)}), \quad k = 0, 1, 2, \ldots$$
   (c) Compute $\mathbf{x}^{(k+1)}$ using
   $$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \Delta\mathbf{x}^{(k)}$$
   (d) Check whether $|\Delta\mathbf{x}^{(k)}| < \epsilon$.
   If yes, exit. Otherwise, set $k := k + 1$ and repeat steps (a)-(d).

Example: Use the Newton method to find an approximate solution of the system (23), starting from the initial guess $x_1 = 0, x_2 = 2, x_3 = -3$, with an error tolerance of 10^{-6}.

Solution: The given system is

$$\begin{aligned} x_2^2 - \sin(x_1 x_3) - 4 &= 0 \\ x_1^3 - 4(x_2 + x_3)^2 + \cos(x_3) + 2 &= 0 \\ e^{x_1 + x_2} - 5x_2 + x_3^2 - 4 &= 0 \end{aligned}$$

for which we have

$$\begin{aligned} f_1(x_1, x_2, x_3) &= x_2^2 - \sin(x_1 x_3) - 4 \\ f_2(x_1, x_2, x_3) &= x_1^3 - 4(x_2 + x_3)^2 + \cos(x_3) + 2 \\ f_3(x_1, x_2, x_3) &= e^{x_1 + x_2} - 5x_2 + x_3^2 - 4 \end{aligned}$$

In vector form, our system is

$$\mathbf{F}(\mathbf{x}) = [f_1(x_1, x_2, x_3), f_2(x_1, x_2, x_3), f_3(x_1, x_2, x_3)]^T$$

The Jacobian matrix for this system is

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} -x_3\cos(x_1x_3) & 2x_2 & -x_1\cos(x_1x_3) \\ 3x_1^2 & -8(x_2+x_3) & -8(x_2+x_3)-\sin(x_3) \\ e^{x_1+x_2} & e^{x_1+x_2}-5 & 2x_3 \end{bmatrix}$$

Iteration 1: Set $\mathbf{x}^{(0)} = (0, 2, -3)^T$ and solve the following linear system for $\Delta\mathbf{x}^{(0)}$

$$[\mathbf{J}(\mathbf{x}^{(0)})]\ \Delta\mathbf{x}^{(0)} = -\mathbf{F}(\mathbf{x}^{(0)})$$

We have

$$\begin{aligned} \mathbf{J}(\mathbf{x_0}) &= \begin{bmatrix} -(-3)\cos(0) & 2(2) & -(0)\cos((2)(-3)) \\ 3(0)^2 & -8(2-3) & -8(2-3)-\sin(-3) \\ e^{0+2} & e^{0+2}-5 & 2(-3) \end{bmatrix} \\ &= \begin{bmatrix} 3 & 4 & 0 \\ 0 & 8 & 8-\sin(-3) \\ e^2 & e^2-5 & -6 \end{bmatrix} \\ \mathbf{F}(\mathbf{x_0}) &= \begin{bmatrix} 2^2-\sin((0)(-3))-4 \\ 0^3-4(2-3)^2+\cos(-3)+2 \\ e^{0+2}-5(2)+0^2-4 \end{bmatrix} = \begin{bmatrix} 0 \\ 2+\cos(-3) \\ e^2-14 \end{bmatrix} \end{aligned}$$

The linear system to be solved is

$$[\mathbf{J}(\mathbf{x}^{(0)})]\ \Delta\mathbf{x}^{(0)} = -\mathbf{F}(\mathbf{x}^{(0)})$$

$$\begin{bmatrix} 3 & 4.0000000 & 0 \\ 0.0000000 & 8.0000000 & 8.1411200 \\ 7.3890591 & 2.3890561 & -6.0000000 \end{bmatrix} \begin{bmatrix} \Delta x_1^{(0)} \\ \Delta x_2^{(0)} \\ \Delta x_3^{(0)} \end{bmatrix} = \begin{bmatrix} 0.0000000 \\ 1.0100075 \\ -6.6109439 \end{bmatrix}$$

Its solution is

$$\Delta\mathbf{x}^{(0)} = (\Delta x_1^{(0)}, \Delta x_2^{(0)}, \Delta x_3^{(0)})^T = (-0.8048, 0.0090, 0.1152)^T$$

This is the estimated change in $\mathbf{x}^{(0)}$, from which we compute the new approximation

$$\mathbf{x_1} = \begin{bmatrix} x_1^{(1)} \\ x_2^{(1)} \\ x_3^{(1)} \end{bmatrix} = \begin{bmatrix} 0.0000 \\ 2.0000 \\ -3.0000 \end{bmatrix} + \begin{bmatrix} -0.8048 \\ 0.0090 \\ 0.1152 \end{bmatrix}$$

We then check whether the condition $|\mathbf{x_1} - \mathbf{x_0}| < 10^{-6}$ is satisfied. If yes, the process terminates, and the current iterate is assumed to be a solution that satisfies the nonlinear system. If the condition is not satisfied, then the iterative process repeats, and we use $\mathbf{x}^{(1)}$ to compute the next approximation $\mathbf{x}^{(2)}$.

In this example, the general setup to compute the kth iterate involves solving the linear system

$$\mathbf{J}(\mathbf{x}^{(k-1)})\ \Delta\mathbf{x}^{(k-1)} = -\mathbf{F}(\mathbf{x}^{(k-1)})$$

for $\Delta\mathbf{x}^{(k-1)}$, with

$$\mathbf{J}(\mathbf{x^{(k-1)}}) = \begin{bmatrix} -x_3^{(k-1)} \cos\left(x_1^{(k-1)} x_3^{(k-1)}\right) & 2x_2^{(k-1)} & -x_1^{(k-1)} \cos\left(x_1^{(k-1)} x_3^{(k-1)}\right) \\ 3\left(x_1^{(k-1)}\right)^2 & -8\left(x_2^{(k-1)} + x_3^{(k-1)}\right) & -8\left(x_2^{(k-1)} + x_3^{(k-1)}\right) - \sin\left(x_3^{(k-1)}\right) \\ e^{(x_1^{(k-1)} + x_2^{(k-1)})} & e^{(x_1^{(k-1)} + x_2^{(k-1)})} - 5 & 2x_3^{(k-1)} \end{bmatrix}$$

and

$$\mathbf{F}(\mathbf{x^{(k-1)}}) = \begin{bmatrix} \left(x_2^{(k-1)}\right)^2 - \sin\left(x_1^{(k-1)} x_3^{k-1}\right) - 4 \\ \left(x_1^{(k-1)}\right)^3 - 4\left(x_2^{(k-1)} + x_3^{(k-1)}\right)^2 + \cos x_3^{(k-1)} + 2 \\ e^{(x_1^{(k-1)} + x_2^{(k-1)})} - 5x_2^{(k-1)} + \left(x_3^{(k-1)}\right)^2 - 4 \end{bmatrix}$$

After computing $\Delta\mathbf{x}^{(k-1)}$, we get the next iterative approximation using

$$\mathbf{x^{(k)}} = \begin{bmatrix} x_1^{(k)} \\ x_2^{(k)} \\ x_3^{(k)} \end{bmatrix} = \begin{bmatrix} x_1^{(k-1)} \\ x_2^{(k-1)} \\ x_3^{(k-1)} \end{bmatrix} + \begin{bmatrix} \Delta x_1^{(k-1)} \\ \Delta x_2^{(k-1)} \\ \Delta x_3^{(k-1)} \end{bmatrix}$$

As before, we check the condition $|\mathbf{x^{(k)}} - \mathbf{x^{(k-1)}}| < 10^{-6}$, and if it is satisfied the process terminates. Otherwise, we continue to the next iteration.

Example:

The LORAN illustration discussed earlier in this chapter outlines a strategy for identifying the position of a ship sailing in the sea. The surface of the sea is assumed to be a two-dimensional plane. The ship receives synchronized, timed signals from two different tracking stations on the coast. The time difference between the signals received locates the ship at the point of intersection of the following two hyperbolas

$$\frac{x^2}{186^2} - \frac{y^2}{300^2 - 186^2} - 1 = 0 \tag{27}$$

$$\frac{(y-500)^2}{279^2} - \frac{(x-300)^2}{500^2 - 279^2} - 1 = 0 \tag{28}$$

(need to correct this info): where the sea surface corresponds to the x-y plane, with one tracking station located at the origin, and the other at the coordinates (300,500) miles. Solve this quadratic system and determine the (x, y) coordinates of the ship's position.

Solution:
As before, we introduce functions f_1, f_2 so that

$$\begin{aligned} f_1(x,y) &= \frac{x^2}{186^2} - \frac{y^2}{300^2 - 186^2} - 1 \\ f_2(x,y) &= \frac{(y-500)^2}{279^2} - \frac{(x-300)^2}{500^2 - 279^2} - 1 \end{aligned}$$

The Jacobian matrix is

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} \\ \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} \end{bmatrix} = \begin{bmatrix} \frac{2}{186^2}x & \frac{-2}{300^2-186^2}y \\ \\ \frac{-2}{500^2-279^2}(x-300) & \frac{2}{279^2}(y-500) \end{bmatrix}$$

We will solve this problem by writing a computer code, a copy of which is given below. The linear systems that arise within each Newton iteration will be solved using either Gauss-Seidel iterations, or directly with Gaussian elimination.

We assume the initial guess is $(x, y) = (200, 200)$ miles, and the error tolerance is 10^{-6}. With these inputs, the computer program converges to the solution $(x, y) \approx (107.20, 192.35)$. These coordinates correspond to the position of the ship with reference to the origin of the x-y plane (usually, the location of the primary tracking station) on which the model is based. We invite the reader to explore other initial guesses, as well as the effect of varying the error tolerance.

There are some important theoretical and practical questions to consider when using the Newton method:

- The initial guess can have a significant impact on the success of the method. It determines whether the iterative process will converge, and if it does, what solution it converges to. Depending on the problem, different initial guesses may converge to different solutions, or they may diverge. Thus, if a problem has multiple solutions, the user will need to try different initial guesses to try and find different solutions.

- When the method converges, the rate of convergence is often quite good. In fact, under certain theoretical conditions, it can be shown that the iterations converge at a rate $O(h^2)$, where h is some measure of the distance of the approximate solution from the exact solution.

- For large problems (with many unknowns and many equations) it can get computationally expensive to find the Jacobian matrix and to solve the linear systems that arise at each Newton step. It is important to explore and implement efficient methods to carry out these operations.

3.3 Computer program for implementing the Newton method

This section provides a sample Python code that demonstrates how to solve the nonlinear system given in the Example 3.2.2 for the LORAN application study. The primary coding modules consist of two linear system solvers (Gauss-Seidel iterations, Gaussian elimination), and one nonlinear solver (Newton method). The function names used in the programs make it self-evident which solver is coded where. The inputs specific to our example are coded in the main program. This includes the right hand side function $\mathbf{F}(\mathbf{x})$, the Jacobian matrix $\mathbf{J}(\mathbf{x})$, and the initial guess $\mathbf{x}^{(0)}$. The code includes a few utility functions that are helpful for doing certain matrix manipulations.

Listing 4: Utility functions

```
def column(m, c):
        return [m[i][c] for i in range(len(m))]

def row(m, r):
        return m[r][:]

def height(m):
        return len(m)

def width(m):
        return len(m[0])
```

Listing 5: Gauss-Seidel method

```
def gauss_seidel(m, x0=None, eps=1e-6, round=1000):
        n  = height(m)
        b  = column(m, n)
        x0 = [0] * n if x0 == None else x0
        x1 = x0[:]

        for k in range(round):
                for i in range(n):
                        s = sum(-m[i][j] * x1[j] for j in range
                           (n) if i != j)
                        x1[i] = (s + b[i]) / m[i][i]
                if all(abs(x1[i]-x0[i]) < eps for i in range(n)
                   ):
                        return x1
                x0 = x1[:]
        raise ValueError('Solution␣does␣not␣converge')
```

Listing 6: Gaussian elemination method

```
def gaussian_elimination(m):
        # forward elimination
        n = height(m)
        for i in range(n):
                for j in range(i+1, n):
                        v = [m[j][k] - (m[i][k] * m[j][i] / m[i
                           ][i]) for k in range(n+1)]
                        m[j] = v
        if m[n-1][n-1] == 0: raise ValueError('No␣unique␣
           solution')

        # backward substitution
        x = [0] * n
        x[n-1] = m[n-1][n] / m[n-1][n-1]
        for i in range(n-2, -1, -1):
                s = sum(m[i][j] * x[j] for j in range(i, n))
                x[i] = (m[i][n] - s) / m[i][i]
        return x
```

Listing 7: Newton-Raphson method

```
def newton_nonlinear(f, j, x0, sys_lin_method=gauss_seidel, eps
   =1e-6, max_iteration=100):
        """
        Parameters
        ----------
        f  : list of functions
        j  : list of list of functions : Jacobian matrix
        x0 : list of floats : initial guess
        sys_lin_meth : function : solve system of linear
           equations

        Returns
        -------
        list of floats

        Raises
        ------
        ValueError : if solution doesn't converge
        """
        for n in range(max_iteration):
                # solve for dx
                m = [[fn(x0) for fn in j[i]] for i in range(len
                   (j))]
                for i in range(len(f)):
                        m[i].append(-f[i](x0))
                dx = sys_lin_method(m)
                # update x0
                x0 = [x0[i] + dx[i] for i in range(len(dx))]
                # terminate?
                if all(abs(i) < eps for i in dx):
                        return x0
        raise ValueError('Solution does not converge')

if __name__ == '__main__':
        f = [   lambda x : (x[0]**2)/(186**2) + (x[1]**2)
           /(300**2 - 186**2) - 1,
                        lambda x : ((x[1]-500)**2)/(279**2) -
                           ((x[0]-300)**2)/(500**2 - 279**2) -
                           1]
        j = [[lambda x :  2*x[0]/(186**2), lambda x : 2*x
           [1]/(300**2 - 186**2)],
                        [lambda x : -2*(x[0]-300)/(500**2 -
                           279**2) , lambda x : 2*(x[1]-500)
```

```
                        /(279**2)]]

    # initial guess
    x0 = [200, 200]

    # solve by Newton method
    x = newton_nonlinear(f, j, x0)

    print('solution is ', x)
    for i in range(len(f)):
            print('f[%d]=%f' % (i, f[i](x)))
```

Exercises

1. Consider the nonlinear system

$$\begin{aligned} x_1^3 + x_1^2 x_2 - x_1 x_3 &= 6 \\ e^{x_1} + e^{x_2} - x_3 &= 0 \\ x_2^2 - 2x_1 x_3 &= 4 \end{aligned}$$

   1.1 Estimate its solution starting from the initial guess $\mathbf{x}^{(0)} = [-1, -1, 1]^T$, with an error tolerance $\epsilon = 0.00001$.

   1.2 Compare your solution with one of the known solutions $(-1.4560, -1.6642, 0.4225)^T$.

   1.3 Try other initial guesses and error tolerances, and observe any differences in solution behavior of results.

2. Discuss the advantages and limitations of the Newton method in the context of Exercise 1, and also more generally.

3. Find the point(s) of intersection of the circle $x^2 + y^2 = 4$ and the hyperbola $xy = 1$.

4. Suppose the circle

$$(x - a)^2 + (y - b)^2 = R^2$$

   passes through the three points $(-1.052, 6.960)$, $(6.017, 5.977)$ and $(3.470, -1.779)$. Find its center (a, b) and radius R.

5. A sphere of the form

$$(x - a)^2 + (y - b)^2 + (z - c)^2 = \rho^2$$

   passes through the points $(-1.465, -5.000, 3.536)$, $(5.000, -9.938, 0.782)$ and $(0.955, -2.061, 0.000)$. Find its center (a, b, c) and radius ρ.

6. Solve the LORAN example given in the text using geometry. Plot the graph of the two given hyperbolas and locate the approximate points of intersection graphically.

7. Consider the problem of trying to locate the position of an object on the earth's surface using GPS (Global Positioning System). How might a mathematical model be used to accomplish this, in conjunction with information from satellites? Discuss some ideas. What are some ways in which the LORAN strategy might be generalized to a GPS strategy?

Unfinished business in Ch 2-3:
*** Include iteration matrix analysis for the examples in Ch 2.**
*** Ch 3 opening illustration on data science is incomplete.**
*** Ch 3 LORAN example is incomplete.**