Introduction to Scientific Computing

for Emory REU/RET Program

James Nagy

Elizabeth Newman

Lars Ruthotto

Alessandro Veneziani

Yuanzhe Xi

Contents

Chapter 1

Getting Started with Matlab

The computational examples and exercises in this book have been computed using MATLAB, which is an interactive system designed specifically for scientific computation that is used widely in academia and industry. At its core, MATLAB contains an efficient, high level programming language and powerful graphical visualization tools which can be easily accessed through a development environment (that is, a graphical user interface containing various workspace and menu items). MATLAB has many advantages over computer languages such as C, C++, Fortran and Java. For example, when using the MATLAB programming language, essentially no declaration statements are needed for variables. In addition, MATLAB has a built-in extensive mathematical function library that contains such items as simple trigonometric functions, as well as much more sophisticated tools that can be used, for example, to compute difficult integrals. Some of these sophisticated functions are described as we progress through the book. MATLAB also provides additional *toolboxes* that are designed to solve specific classes of problems, such as for image processing. It should be noted that there is no free lunch; because MATLAB is an "interpreted" language, codes written in Fortran and C are usually more efficient for very large problems. (MATLAB may also be compiled.) Therefore, large production codes are usually written in one of these languages, in which case supplementary packages, such as the NAG or IMSL library, or free software from *netlib*, are recommended for scientific computing. However, because it is an excellent package for developing algorithms and problem solving environments, and it can be quite efficient when used properly, all computing in this book uses MATLAB.

We provide a very brief introduction to MATLAB. Though our discussion assumes the use of MATLAB 7.0 or higher, in most cases version 6.0 or 6.5 is sufficient. There are many good sources for more complete treatments on using MATLAB, both on-line, and as books. One excellent source is the *MATLAB Guide, 2nd ed.* by D.J. Higham and N.J. Higham published by SIAM Press, 2005. Another source that we highly recommend is MATLAB's built-in help system, which can be accessed once MATLAB is started. We explain how to access it in section 1.1. The remainder of the chapter then provides several examples that introduce various basic capabilities of MATLAB, such as graph plotting and writing functions.

1.1 Starting, Quitting, and Getting Help

The process by which you start MATLAB depends on your computer system; you may need to request specific commands from your instructor or system administrator. Generally, the process is as follows:

- On a PC with a Windows operating system, double-click the "MATLAB" shortcut icon on your Windows desktop. If there is no "MATLAB" icon on the desktop, you may bring up DOS (on Windows XP by going to the Command Prompt in Accessories) and entering `matlab` at the operating system prompt. Alternatively, you may search for the "MATLAB" icon in a subdirectory and click on it. Where it is to be found depends on how MATLAB was installed; in the simplest case with a default installation it is found in the `C:\$MATLAB` directory, where

`$MATLAB` is the name of the folder containing the MATLAB installation.

- On a Macintosh running OS X 10.1 or higher, there may be a "Matlab" icon on the dock. If so, then clicking this icon should start Matlab. If the "Matlab" icon is not on the dock, then you need to find where it is located. Usually it is found in `/Applications/$MATLAB/`, or `/Applications/$MATLAB/bin/`, where `$MATLAB` is the name of the folder containing the Matlab installation. Once you find the "Matlab" icon, double clicking on it should start Matlab.
- On Unix or Linux platforms, typically you enter `matlab` at the shell prompt. That is, you open a terminal, and enter the command `matlab`.

When you have been successful in getting Matlab to start, then the development tool Graphical User Interface (GUI) should appear on the screen. Although there are slight differences (such as key stroke short cuts) between platforms, in general the GUI should have the same look and feel independently of the platform.

The *command window*, where you will do much of your work, contains a prompt:

```
>>
```

We can enter data and execute commands at this prompt. One very useful command is `doc`, which displays the "help browser". For example, entering the command

```
>> doc matlab
```

opens the help browser to a good location for first time Matlab users to begin reading. Alternatively, you can pull down the *Help* menu, and let go on *MATLAB Help* or *Documentation.* We recommend that you read some of the information on these help pages now, but we also recommend returning periodically to read more as you gain experience using Matlab.

Throughout this book we provide many examples using Matlab. In all cases, we encourage readers to "play along" with the examples provided. While doing so, it may be helpful at times to use the `doc` command to find detailed information about various Matlab commands and functions. For example,

```
>> doc plot
```

opens the help browser, and turns to a page containing detailed information on using the built-in `plot` function.

To exit Matlab, you can pull down the *File* menu, and let go on *Quit MATLAB.* Alternatively, in the command window, you can use the `exit` command:

```
>> exit
```

1.2 Basics of Matlab

Matlab derives its name from MATrix LABoratory because the primary object involved in any Matlab computation is a *matrix.* A matrix A is an array of values, with a certain number of rows and columns that define the "dimension" of the matrix. For example, the array A given by

$$A = \begin{bmatrix} 0 & -6 & 8 & 1 \\ -2 & 5 & 5 & -3 \\ 7 & 8 & 0 & 3 \end{bmatrix}$$

is a matrix with 3 rows and 4 columns, and so is typically referred to as a 3×4 matrix. It is two-dimensional. The values in the matrix are by default all Double Precision numbers which will be discussed in more detail in the next chapter. This fact permits Matlab to avoid using declarations but involves a possible overhead in memory usage and speed of computation. A matrix with only one row (that is, a $1 \times n$ matrix) is often called a row vector, while a matrix with only one column (that is, an $n \times 1$ matrix) is called a column vector. For example if

$$x = \begin{bmatrix} -2 \\ 8 \\ 4 \\ 0 \\ 5 \end{bmatrix} \quad \text{and} \quad y = \begin{bmatrix} -3 & 6 & 3 & -4 \end{bmatrix},$$

then we can say that x is a 5×1 matrix, or that it is a column vector of length 5. Similarly, we can say that y is a 1×4 matrix, or that it is a row vector of length 4. If the shape is obvious from the context, then we may omit the words *row* or *column*, and just refer to the object as a vector.

Matlab is very useful when solving problems whose computations involve matrices and vectors. We explore some of these basic *linear algebra* manipulations, as well as some basic features of Matlab, through a series of examples.

Initializing Vectors. We can easily create row and/or column vectors in Matlab. For example, the following two statements create the same row vector:

```
>> x = [1 2 3 4]
>> x = [1, 2, 3, 4]
```

Similarly, the following two statements create the same column vector:

```
>> x = [1
2
3
4]
>> x = [1; 2; 3; 4]
```

Rather than thinking of these structures as row and column vectors, we should think of them as 1×4 and 4×1 matrices, respectively. Observe that elements in a row may be separated by using either a blank space or by using a comma. Similarly, to indicate that a row has ended, we can use either a carriage return, or a semicolon.

Initializing Matrices. In general, we can create matrices with more than one row and column. The following three statements generate the same matrix:

```
>> A = [1 2 3 4
5 6 7 8
9 10 11 12]
>> A = [1 2 3 4; 5 6 7 8; 9 10 11 12]
>> A = [1, 2, 3, 4; 5, 6, 7, 8; 9, 10, 11, 12]
```

Matrix Arithmetic. Matrices can be combined (provided certain requirements on their dimensions are satisfied) using the operations `+, -, *` to form new matrices. Addition and subtraction of matrices is intuitive; to add or subtract two matrices, we simply add or subtract corresponding entries. The only requirement is that the two matrices have the same dimensions. For example, if

$$A = \begin{bmatrix} -3 & -1 \\ 3 & 2 \\ 3 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 7 & 3 \\ -2 & 2 \\ 3 & -1 \end{bmatrix}, \quad \text{and} \quad C = \begin{bmatrix} -3 & -2 & -2 \\ 5 & 8 & -2 \end{bmatrix}$$

then the Matlab commands

```
>> D = A + B
>> E = B - A
```

produce the matrices

$$D = \begin{bmatrix} 4 & 2 \\ 1 & 4 \\ 6 & -1 \end{bmatrix} \quad \text{and} \quad E = \begin{bmatrix} 10 & 4 \\ -5 & 0 \\ 0 & -1 \end{bmatrix},$$

but the command

```
>> F = C + A
```

produces an error message because the matrices C and A do not have the same dimensions.

Matrix multiplication, which is less intuitive than addition and subtraction, can be defined using linear combinations of vectors. We begin with something that is intuitive, namely the product of a scalar (that is, a number) and a vector. In general, if c is a scalar and a is a vector with entries a_i, then ca is a vector with entries ca_i. For example,

$$\text{if} \quad \mathrm{c} = 9 \quad \text{and} \quad a = \begin{bmatrix} 3 \\ 4 \\ -2 \end{bmatrix} \quad \text{then} \quad \mathrm{c}a = 9 \begin{bmatrix} 3 \\ 4 \\ -2 \end{bmatrix} = \begin{bmatrix} 27 \\ 36 \\ -18 \end{bmatrix}.$$

Once multiplication by a scalar is defined, it is straight forward to form linear combinations of vectors. For example, if

$$\mathrm{c}_1 = 9,\ \mathrm{c}_2 = 3,\ \mathrm{c}_3 = 1, \quad \text{and} \quad a_1 = \begin{bmatrix} 3 \\ 4 \\ -2 \end{bmatrix},\ a_2 = \begin{bmatrix} 3 \\ 8 \\ 0 \end{bmatrix},\ a_3 = \begin{bmatrix} 4 \\ 0 \\ 4 \end{bmatrix},$$

then

$$\mathrm{c}_1 a_1 + \mathrm{c}_2 a_2 + \mathrm{c}_3 a_3 = 9 \begin{bmatrix} 3 \\ 4 \\ -2 \end{bmatrix} + 3 \begin{bmatrix} 3 \\ 8 \\ 0 \end{bmatrix} + 1 \begin{bmatrix} 4 \\ 0 \\ 4 \end{bmatrix} = \begin{bmatrix} 40 \\ 60 \\ -14 \end{bmatrix}.$$

is a linear combination of the vectors a_1, a_2 and a_3 with the scalars c$_1$, c$_2$ and c$_3$, respectively.

Now, suppose we define a matrix A and column vector x as

$$A = \begin{bmatrix} 3 & 3 & 4 \\ 4 & 8 & 0 \\ -2 & 0 & 4 \end{bmatrix}, \quad x = \begin{bmatrix} 9 \\ 3 \\ 1 \end{bmatrix},$$

then the matrix–vector product Ax is simply a compact way to represent a linear combination of the columns of the matrix A, where the scalars in the linear combination are the entries in the vector x. That is,

$$Ax = 9 \begin{bmatrix} 3 \\ 4 \\ -2 \end{bmatrix} + 3 \begin{bmatrix} 3 \\ 8 \\ 0 \end{bmatrix} + 1 \begin{bmatrix} 4 \\ 0 \\ 4 \end{bmatrix} = \begin{bmatrix} 40 \\ 60 \\ -14 \end{bmatrix}.$$

Thus, to form Ax, the number of columns in the matrix A must be the same as the number of entries in the vector x.

Finally we consider matrix-matrix multiplication. If A is an $m \times n$ matrix, and B is an $n \times p$ matrix (that is, the number of columns in A is the same as the number of rows in B), then the product $C = AB$ is an $m \times p$ matrix whose columns are formed by multiplying A by corresponding columns in B viewed as column vectors. For example, if

$$A = \begin{bmatrix} 3 & 3 & 4 \\ 4 & 8 & 0 \\ -2 & 0 & 4 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 9 & -1 & -2 & 0 \\ 3 & 0 & 1 & -3 \\ 1 & 2 & -1 & 5 \end{bmatrix}$$

then

$$C = AB = \begin{bmatrix} 40 & 5 & -7 & 11 \\ 60 & -4 & 0 & -24 \\ -14 & 10 & 0 & 20 \end{bmatrix}.$$

So, the jth column of AB is the linear combination of the columns of A with scalars drawn from the jth column of B. For example, the 3rd column of AB is formed by taking the linear combination of the columns of A with scalars drawn from the 3rd column of B. Thus, the 3rd column of AB is $(-2) * a_1 + (1) * a_2 + (-1) * a_3$ where a_1, a_2 and a_3 denote, respectively, the 1st, 2nd, and 3rd columns of A.

The $*$ operator can be used in MATLAB to multiply scalars, vectors, and matrices, provided the dimension requirements are satisfied. For example, if we define

```
>> A = [1 2; 3 4]
>> B = [5 6; 7 8]
>> c = [1; -1]
>> r = [2 0]
```

and we enter the commands

```
>> C = A*B
>> y = A*c
>> z = A*r
>> w = r*A
```

we find that

$$C = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}, \quad y = \begin{bmatrix} -1 \\ -1 \end{bmatrix}, \quad w = \begin{bmatrix} 2 & 4 \end{bmatrix},$$

but an error message is displayed when trying to calculate `z = A*r` because it is not a legal linear algebra operation; the number of columns in A is not the same as the number of rows in r (that is, the inner matrix dimensions do not agree).

Suppressing Output. Note that each time we execute a statement in the MATLAB command window, the result is redisplayed in the same window. This action can be suppressed by placing a semicolon at the end of the statement. For example, if we enter

```
>> x = 10;
>> y = 3;
>> z = x*y
```

then only the result of the last statement, $z = 30$, is displayed in the command window.

Special Characters In the above examples we observe that the semicolon can be used for two different purposes. When used inside brackets `[ ]` it indicates the end of a row, but when used at the end of a statement, it suppresses display of the result in the command window.

The comma can also be used for two different purposes. When used inside brackets `[ ]` it separates entries in a row, but it can also be used to separate statements in a single line of code. For example, the previous example could have been written in one line as follows:

```
>> x = 10, y = 3, z = x*y
```

The values $x = 10$, $y = 3$ and $z = 30$, are displayed in the command window. If the commas are replaced by semicolons,

```
>> x = 10; y = 3; z = x*y
```

then only $z = 30$ is displayed in the command window.

The semicolon, comma and brackets are examples of certain special characters in MATLAB that are defined for specific purposes. For a full list of special characters, open MATLAB's documentation, which can be done using the MATLAB `Help` menu button, or by entering `doc` in the command window, and navigate to: `MATLAB > Language Fundamentals > Operators and Elementary Operations > MATLAB Operators and Special Characters.`

Transposing Matrices. The *transpose* operation is used in linear algebra to transform an $m \times n$ matrix into an $n \times m$ matrix by transforming rows to columns, and columns to rows. For example, if we have the matrices and vectors:

$$A = \begin{bmatrix} 1 & 2 & 5 \\ 3 & 4 & 6 \end{bmatrix}, \quad c = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \quad r = \begin{bmatrix} 2 & 0 \end{bmatrix},$$

then

$$A^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \\ 5 & 6 \end{bmatrix}, \quad c^T = \begin{bmatrix} 1 & -1 \end{bmatrix}, \quad r^T = \begin{bmatrix} 2 \\ 0 \end{bmatrix}.$$

where superscript T denotes transposition. In MATLAB, the single quote ' is used to perform the transposition operation. For example, consider the following matrices:

```
>> A = [1 2 5; 3 4 6];
>> c = [1; -1];
```

When we enter the commands

```
>> D = A'
>> s = c'*c
>> H = c*c'
```

we obtain

$$D = \begin{bmatrix} 1 & 3 \\ 2 & 4 \\ 5 & 6 \end{bmatrix}, \quad s = 2, \quad H = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}.$$

Other Array Operations. MATLAB supports certain array operations (not normally found in standard linear algebra books) that can be very useful in scientific computing applications. Some of these operations are:

$$.* \quad ./ \quad .\hat{}$$

The *dot* indicates that the operation is to act on the matrices in an element by element (componentwise) way. For example,

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} .* \begin{bmatrix} 5 \\ 6 \\ 7 \\ 8 \end{bmatrix} = \begin{bmatrix} 5 \\ 12 \\ 21 \\ 32 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} .\hat{}3 = \begin{bmatrix} 1 \\ 8 \\ 27 \\ 64 \end{bmatrix}$$

The rules for the validity of these operations are different than for linear algebra. A full list of arithmetic operations, and the rules for their use in MATLAB, can be found by referring to the documentation (help), and navigating to: `MATLAB > Language Fundamentals > Operators and Elementary Operations > Arithmetic`.

Remark on the Problems. The problems in this chapter are designed to help you become more familiar with MATLAB and some of its capabilities. Some problems may include issues not explicitly discussed in the text, but a little exploration in MATLAB (that is, executing the statements and reading appropriate `doc` pages) should provide the necessary information to solve all of the problems.

Problem 1.2.1. *If* `z = [0 -1 2 4 -2 1 5 3]`, *and* `J = [5 2 1 6 3 8 4 7]`, *determine what is produced by the following sequences of* MATLAB *statements:*

```
>> x = z', A = x*x', s = x'*x, w = x*J,
>> length(x), length(z)
>> size(A), size(x), size(z), size(s)
```

Note that to save space, we have placed several statements on each line. Use `doc length` *and* `doc size`, *or search the* MATLAB *Help index, for further information on these commands.*

1.3 Matlab as a Scientific Computing Environment

So far we have used MATLAB as a sophisticated calculator. It is far more powerful than that, containing many programming constructs, such as flow control (if, for, while, etc.), and capabilities for writing functions, and creating structures, classes, objects, etc. Since this is not a MATLAB programming book, we do not discuss these capabilities in great detail. But many MATLAB programming issues are discussed, when needed, as we progress through the book. Here, we introduce a few concepts before beginning our study of scientific computing.

1.3.1 Initializing Vectors with Many Entries

Suppose we want to create a vector, `x`, containing the values $1, 2, \ldots, 100$. We could do this using a `for` loop:

```
n = 100;
for i = 1:n
  x(i) = i;
end
```

In this example, the notation `1:n` is used to create the list of integers $1, 2, 3, \ldots, n$. When MATLAB begins to execute this code, it does not know that `x` will be a row vector of length 100, but it has a very smart *memory manager* that creates space as needed. Forcing the memory manager to work hard can make codes very inefficient. Fortunately, if we know how many entries `x` will have, then we can help out the memory manager by first allocating space using the `zeros` function. For example:

```
n = 100;
x = zeros(1,n);
for i = 1:n
  x(i) = i;
end
```

In general, the function `zeros(m,n)` creates an $m \times n$ array containing all zeros. Thus, in our case, `zeros(1,n)` creates a $1 \times n$ array, that is a row vector with n entries.

A much simpler, and better way, to initialize this simple vector using one of MATLAB's *vector operations*, is as follows:

```
n = 100;
x = 1:n;
```

The colon operator is very useful! Let's see another example where it can be used to create a vector. Suppose we want to create a vector, x, containing n entries equally spaced between $a = 0$ and $b = 1$. The distance between each of the equally spaced points is given by $h = \dfrac{b-a}{n-1} = \dfrac{1}{n-1}$, and the vector, x, should therefore contain the entries:

$$0, \quad 0+h, \quad 0+2*h, \quad \cdots, \quad (i-1)*h, \quad \cdots, \quad 1$$

We can create a vector with these entries, using the colon operator, as follows:

```
n = 101;
h = 1 / (n-1);
x = 0:h:1;
```

or, if we do not need the variable `h` later in our program, we can use:

```
n = 101;
x = 0:1/(n-1):1;
```

We often want to create vectors like this in mathematical computations. Therefore, MATLAB provides a function `linspace` for it. In general, `linspace(a, b, n)` generates a vector of n equally spaced points between a and b. So, in our example with $a = 0$ and $b = 1$, we could instead use:

```
n = 101;
x = linspace(0, 1, n);
```

Note that, for the interval $[0, 1]$, choosing $n = 101$ produces a nice rational spacing between points, namely $h = 0.01$. That is,

$$x = \begin{bmatrix} 0 & 0.01 & 0.02 & \cdots & 0.98 & 0.99 & 1 \end{bmatrix}.$$

A lesson is to be learned from the examples in this subsection. Specifically, if we need to perform a fairly standard mathematical calculation, then it is often worth using the *search* facility in the *help browser* to determine if MATLAB already provides an optimized function for the calculation.

Problem 1.3.1. *Determine what is produced by the* MATLAB *statements:*

```
>> i = 1:10
>> j = 1:2:11
>> x = 5:-2:-3
```

For more information on the use of `":"`*, see* `doc colon`*.*

Problem 1.3.2. *If* `z = [0 -1 2 4 -2 1 5 3]`*, and* `J = [5 2 1 6 3 8 4 7]`*, determine what is produced by the following* MATLAB *statements:*

```
>> z(2:5)
>> z(J)
```

Problem 1.3.3. *Determine what is produced by the following* MATLAB *statements:*

```
>> A = zeros(2,5)
>> B = ones(3)
>> R = rand(3,2)
>> N = randn(3,2)
```

What is the difference between the functions `rand` *and* `randn`*? What happens if you repeat the statement* `R = rand(3,2)` *several times? Now repeat the following pair of commands several times:*

```
>> rng('default')
>> R = rand(3,2)
```

What do you observe? Note: The "up arrow" key can be used to recall statements previously entered in the command window.

Problem 1.3.4. *Determine what is produced by the* MATLAB *statements:*

```
>> x = linspace(1, 1000, 4)
>> y = logspace(0, 3, 4)
```

In each case, what is the spacing between the points?

Problem 1.3.5. *Given integers* `a` *and* `b`*, and a rational number* `h`*, determine a formula for* `n` *such that*

```
linspace(a, b, n) = [a, a+h, a+2h, ..., b]
```

1.3.2 Creating Plots

Suppose we want to plot the function $y = x^2 - \sqrt{x+3} + \cos 5x$ on the interval $-3 \le x \le 5$. The basic idea is first to plot several points, (x_i, y_i), and then to connect them together using lines. We can do this in Matlab by creating a vector of x-coordinates, a vector of y-coordinates, and then using the Matlab command `plot(x,y)` to draw the graph in a *figure* window. For example, we might consider the Matlab code:

```
n = 81;
x = linspace(-3, 5, n);
y = zeros(1, n);
for i = 1:n
  y(i) = x(i)^2 - sqrt(x(i) + 3) + cos(5*x(i));
end
plot(x, y)
```

In this code we have used the `linspace` command to create the vector `x` efficiently, and we helped the memory manager by using the `zeros` command to pre-allocate space for the vector `y`. The `for` loop generates the entries of `y` one at time, and the `plot` command draws the graph.

Matlab allows certain operations on arrays that can be used to shorten this code. For example, if `x` is a vector containing entries x_i, then:

- `x + 3` is a vector containing entries $x_i + 3$, and `sqrt(x + 3)` is a vector containing entries $\sqrt{x_i + 3}$.
- Similarly, `5*x` is a vector containing entries $5x_i$, and `cos(5*x)` is a vector containing entries $\cos(5x_i)$.
- Finally, recalling the previous section, we can use the *dot* operation `x.^2` to compute a vector containing entries x_i^2.

Using these properties, the `for` loop above may be replaced with a single *vector operation*:

```
n = 81;
y = zeros(1, n);
x = linspace(-3, 5, n);
y = x.^2 - sqrt(x + 3) + cos(5*x);
plot(x,y)
```

If you can use array operations instead of loops, then you should, as they are more efficient.

Matlab has many more, very sophisticated, plotting capabilities. Three very useful commands are `axis`, `subplot`, and `hold`:

- `axis` is mainly used to scale the x and y-axes on the current plot as follows:
  ```
  axis([xmin xmax ymin ymax])
  ```
- `subplot` is mainly used to put several plots in a single figure window. Specifically,
  ```
  subplot(m, n, p)
  ```
  breaks the figure window into an "$m \times n$ matrix" of small axes, and selects the p^{th} set of axes for the current plot. The axes are counted along the top row of the figure window, then the second row, etc.
- `hold` allows you to overlay several plots on the same set of axes.

The following example illustrates how to use these commands.

Example 1.3.1. Chebyshev polynomials are used in a variety of engineering applications. The j^{th} Chebyshev polynomial $T_j(x)$ is defined by

$$T_j(x) = \cos(j \arccos(x)), \quad -1 \leq x \leq 1 .$$

In section **??** we see that these strange objects are indeed polynomials.

(a) First we plot, in the same figure, the Chebyshev polynomials for $j = 1, 3, 5, 7$. This can be done by executing the following statements in the command window:

```
x = linspace(-1, 1, 201);
T1 = cos(acos(x));
T3 = cos(3*acos(x));
T5 = cos(5*acos(x));
T7 = cos(7*acos(x));
subplot(2,2,1), plot(x, T1)
subplot(2,2,2), plot(x, T3)
subplot(2,2,3), plot(x, T5)
subplot(2,2,4), plot(x, T7)
```

The resulting plot is shown in Fig. 1.1.

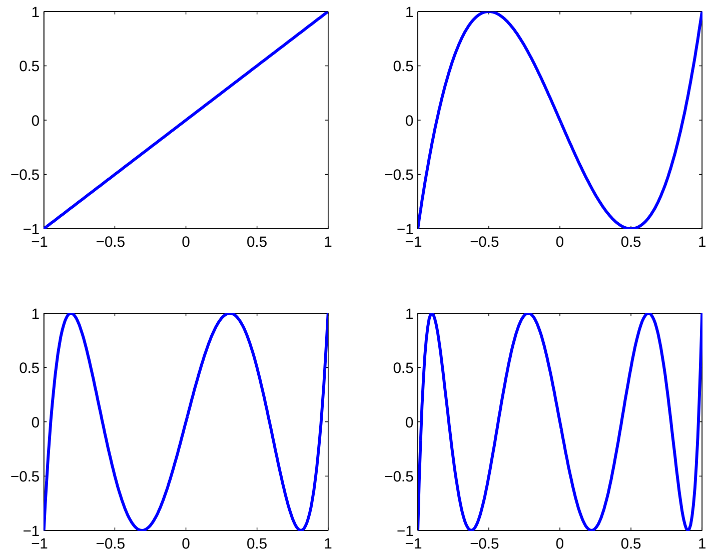


Figure 1.1: Using `subplot` in Example 1.3.1(a).

(b) When you use the `plot` command, MATLAB chooses (usually appropriately) default values for the axes. Here, it chooses precisely the domain and range of the Chebyshev polynomials. These can be changed using the `axis` command, for example:

```
subplot(2,2,1), axis([-1, 1, -2, 2])
subplot(2,2,2), axis([-2, 2, -1, 1])
subplot(2,2,3), axis([-2, 1, -1, 2])
subplot(2,2,4), axis([-2, 2, -2, 2])
```

The resulting plot is shown in Fig. 1.2.

(c) Finally, we can plot all of the polynomials on the same set of axes. This can be achieved as follows (here we use different colors, blue, red, green and cyan, for each):

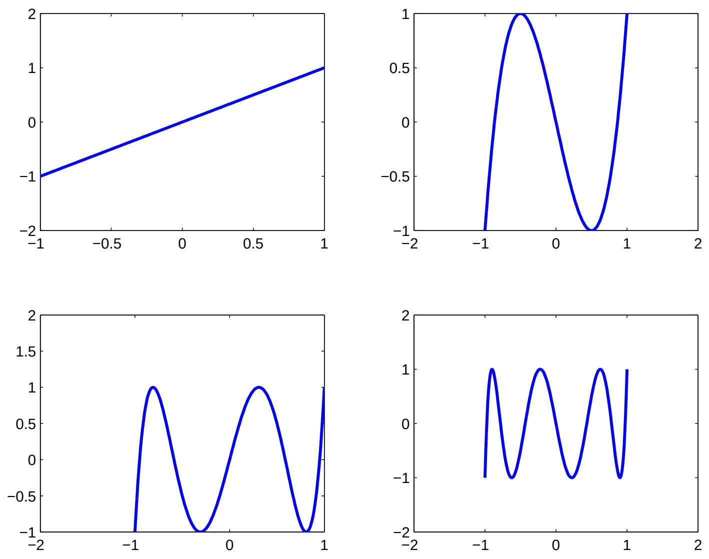


Figure 1.2: Using `axis` in Example 1.3.1(b).

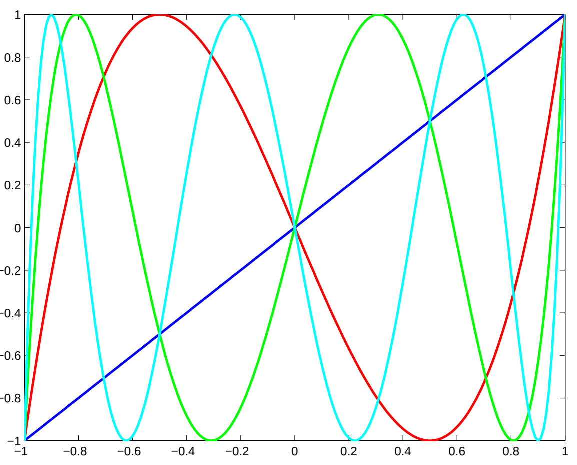


Figure 1.3: Using `hold` in Example 1.3.1(c).

```
subplot(1,1,1)
plot(x, T1, 'b')
hold on
plot(x, T3, 'r')
plot(x, T5, 'g')
plot(x, T7, 'c')
```

The resulting plot is shown in Fig. 1.3. (You should see the plot in color on your screen.)

Remarks on Matlab Figures. We have explained that `hold` can be used to overlay plots on the same axes, and `subplot` can be used to generate several different plots in the *same* figure. In some cases, it is preferable to generate several different plots in *different* figures, using the `figure` command. To clear a figure, so that a new set of plots can be drawn in it, use the `clf` command.

Problem 1.3.6. *Write* MATLAB *code that evaluates and plots the functions:*

(a) $y = 5\cos(3\pi x)$ *for 101 equally spaced points on the interval* $0 \le x \le 1$.

(b) $y = \dfrac{1}{1+x^2}$ *for 101 equally spaced points on the interval* $-5 \le x \le 5$.

(c) $y = \dfrac{\sin 7x - \sin 5x}{\cos 7x + \cos 5x}$ *using 200 equally spaced points on the interval* $-\pi/2 \le x \le \pi/2$. *Use the* `axis` *command to scale the plot so that* $-2 \le x \le 2$ *and* $-10 \le y \le 10$.

In each case, your code should not contain loops but should use arrays directly.

Problem 1.3.7. *A "fixed–point" of a function* $g(x)$ *is a point* x *that satisfies* $x = g(x)$. *An "educated guess" of the location of a fixed–point can be obtained by plotting* $y = g(x)$ *and* $y = x$ *on the same axes, and estimating the location of the intersection point. Use this technique to estimate the location of a fixed–point for* $g(x) = \cos x$.

Problem 1.3.8. *Use* MATLAB *to recreate the plot in Fig. 1.4. Hints: You will need the commands* `hold`, `xlabel`, `ylabel`, *and* `legend`. *The* ± *symbol in the legend can be created using* `\pm`.

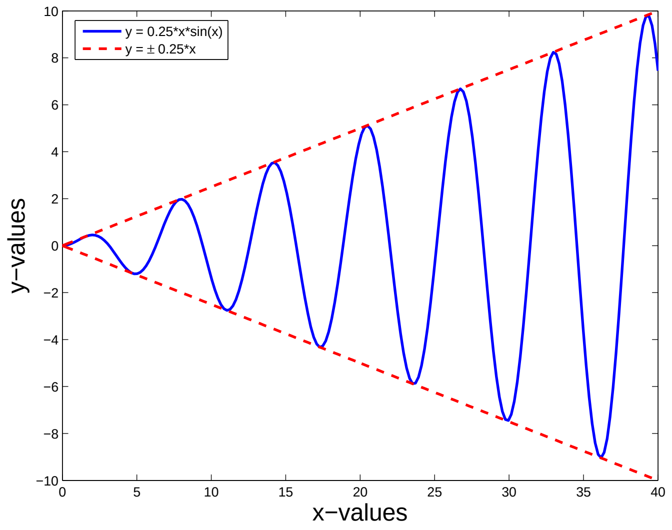


Figure 1.4: Example of a plot that uses MATLAB commands `hold`, `xlabel`, `ylabel`, and `legend`.

Problem 1.3.9. *Explain what happens when the following* MATLAB *code is executed.*

```
for k = 1:6
  x = linspace(0, 4*pi, 2^(k+1)+1);
  subplot(2,3,k), plot(x, sin(x))
  axis([0, 4*pi, -1.5, 1.5])
end
```

What happens if `subplot(2, 3, k)` *is changed to* `subplot(3, 2, k)`*? What happens if the* `axis` *statement is omitted?*

1.3.3 Script and Function M-Files

We introduce *script* and *function* M–files that should be used to write MATLAB programs.

- A *script* is a file containing MATLAB commands that are executed by using the *Run* icon (look for a green, right pointing triangle), under the *Editor* tab, or when the name of the file is entered at the prompt in the MATLAB command window. Scripts are convenient for conducting computational experiments that require entering many commands. However, care must be taken because variables in a script are global to the MATLAB session, and it is therefore easy to unintentionally change their values.

- A *function* is a file containing MATLAB commands that are executed when the function is called. The first line of a function must have the form:

  `function [`*out1, out2,*`... ] =` *FunctionName(in1, in2, ...)*

  By default, any data or variables created within the function are private. You can specify any number of inputs to the function, and if you want it to return results, then you can specify any number of outputs. Functions can call other functions, so you can write sophisticated programs as in any conventional programming language.

In each case, the program must be saved in a M–file; that is, a file with a `.m` extension. Any editor may be used to create an M–file, but we recommend using MATLAB's built-in editor, which can be opened in one of several ways. In addition to clicking on certain menu items, the editor can be opened using the `edit` command. For example, if we enter

```
edit ChebyPlots.m
```

in the command window, then the editor will open the file ChebyPlots.m, and we can begin entering and modifying MATLAB commands.

It is important to consider carefully how the script and function M–files are to be named. As mentioned above, they should all have names of the form:

FunctionName`.m` or *ScriptName*`.m`

The names should be descriptive, but it is also important to avoid using a name already taken be one of MATLAB's many built-in functions. If we happen to use the same name as one of these built-in functions, then MATLAB has a way of choosing which function to use, but such a situation can be very confusing. The command `exist` can be used to determine if a function (or variable) is already defined by a specific name and the command `which` can be used to determine its path. Note, for a function file the name of the function and the name of the M–file must be the same.

To illustrate how to write functions and scripts, we provide two examples.

Example 1.3.2. In this example we write a simple function, `PlotCircle.m`, that generates a plot of a circle with radius r centered at the origin:

```
function PlotCircle(r)
%
%          PlotCircle(r)
%
%   This function plots a circle of radius r centered at the origin.
%   If no input value for r is specified, the default value is chosen
%   as r = 1.  We check for too many inputs and for negative input
%   and report errors in both cases.
%
if nargin < 2
   if nargin == 0
     r = 1;
   elseif r <= 0
     error('The input value should be > 0.')
   end
   theta = linspace(0, 2*pi, 200);
   x = r*cos(theta);
   y = r*sin(theta);
   plot(x, y)
   axis([-2*r,2*r,-2*r,2*r])
   axis square
else
  error('Too many input values.')
end
```

We can use it to generate plots of circles. For example, if we enter

```
>> PlotCircle(2)
```

then a circle of radius 2 centered at the origin is plotted, as shown in Figure 1.5. Note, we have chosen to plot 200 points equally spaced in the variable `theta`. If the resulting plotted circle seems a little "ragged" increase the number of points.

This code introduces some new MATLAB commands that require a little explanation:

- The percent symbol, %, is used to denote a comment in the program. On any line anything after a % symbol is treated as comment. All programs (functions and scripts) should contain comments to explain their purpose, and if there are any input and/or output variables.

- `nargin` is a built-in MATLAB command that can be used to count the number of input arguments to a function. If we run this code using `PlotCircle(2)`, `PlotCircle(7)`, etc., then on entry to this function, the value of `nargin` is 1. However, if we run this code using the command `PlotCircle` (with no specified input argument), then on entry to this function, the value of `nargin` is 0. We also use the command `nargin` to check for too many input values.

- The conditional command `if` is used to execute a statement if the expression is true. First we check that there are not too many inputs. Then, we check if `nargin` is 0 (that is, the input value has not been specified). In the latter case, the code sets the radius r to the default value of 1. Note, the difference between the statements `x = 0` and `x == 0`; the former sets the value of x to 0, while the latter checks to see if x and 0 are equal. Observe that the `if` statements are "nested", and thus the `end` statements are correspondingly nested. Each `end` encountered corresponds to the most recent `if`. The indenting used (and produced automatically by the MATLAB editor) should help you follow the command structure of the program. We also use an `elseif` statement to make sure the input value for the radius is not negative. The `doc` command can be used to find more detailed information on `if` and related statements such as `else` and `elseif`.

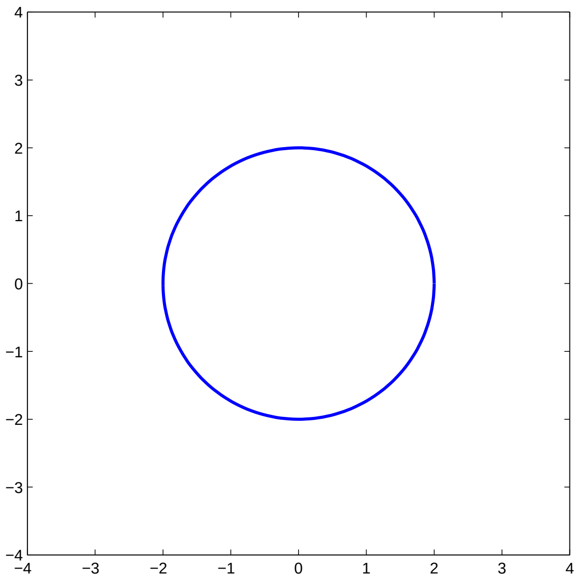


Figure 1.5: Figure created when entering `PlotCirle(2)`.

- `error` is a built-in MATLAB command. When this command is executed, the computation terminates, and the message between the single quotes is printed in the command window.

Example 1.3.3. Here, we illustrate how scripts can be used in combination with functions. Suppose that the concentration of spores of pollen per square centimeter are measured over a 15 day period, resulting in the following data:

day	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
pollen count	12	35	80	120	280	290	360	290	315	280	270	190	90	85	66

(a) First, we write a function that has as input a vector of data, and returns as output the mean and standard deviation of this data.

MATLAB has several built-in functions for statistical analysis. In particular, we can use `mean` and `std` to compute the mean and standard deviation, respectively. Thus, the function we write, named `GetStats.m`, is very simple:

```
function [m, s] = GetStats(x);
%
%   Compute the mean and standard deviation of entries
%   in a vector x.
%
%   Input:  x - vector (either row or column)
%
%   Output:  m - mean value of the elements in x.
%            s - standard deviation of the elements in x.
%
m = mean(x);
s = std(x);
```

```
%
%   Script:  PollenStats
%
%   This script shows a statistical analysis of measured pollen
%   counts on 15 consecutive days.
%

%
%  p is a vector containing the pollen count for each day.
%  d is a vector indicating the day (1 -- 15).
%
p = [12 35 80 120 280 290 360 290 315 280 270 190 90 85 66];
d = 1:length(p);
%
%  Get statistical data, and produce a plot of the results.
%
[m, s] = GetStats(p);
bar(d, p, 'b')
xlabel('Day of Measurement')
ylabel('Pollen Count')
hold on
plot([0, 16], [m, m], 'r')
plot([0, 16], [m-s, m-s], 'r--')
plot([0, 16], [m+s, m+s], 'r--')
text(16, m, '<-- m')
text(16, m-s, '<-- m - s')
text(16, m+s, '<-- m + s')
hold off
```

(b) We now write a script, `PollenStats.m`, to generate and display a statistical analysis of the pollen data.

Once the programs are written, we can run the script by simply entering its name in the command window:

```
>> PollenStats
```

The resulting plot is shown in Figure 1.6.

The script `PollenStats.m` uses several new commands. For detailed information on the specific uses of these commands, use `doc` (for example, `doc bar`). See if you can work out what is going on in the `plot` and `text` lines of `PollenStats.m`.

Problem 1.3.10. *Write a function M–file,* `PlotCircles.m`*, that accepts as input a vector,* `r`*, having positive entries, and plots circles of radius* `r(i)` *centered at the origin on the same axes. If no input value for* `r` *is specified, the default is to plot a single circle with radius* `r = 1`*. Test your function with* `r = 1:5`*. Hint:* MATLAB *commands that might be useful include* `max`*,* `min` *and* `any`*.*

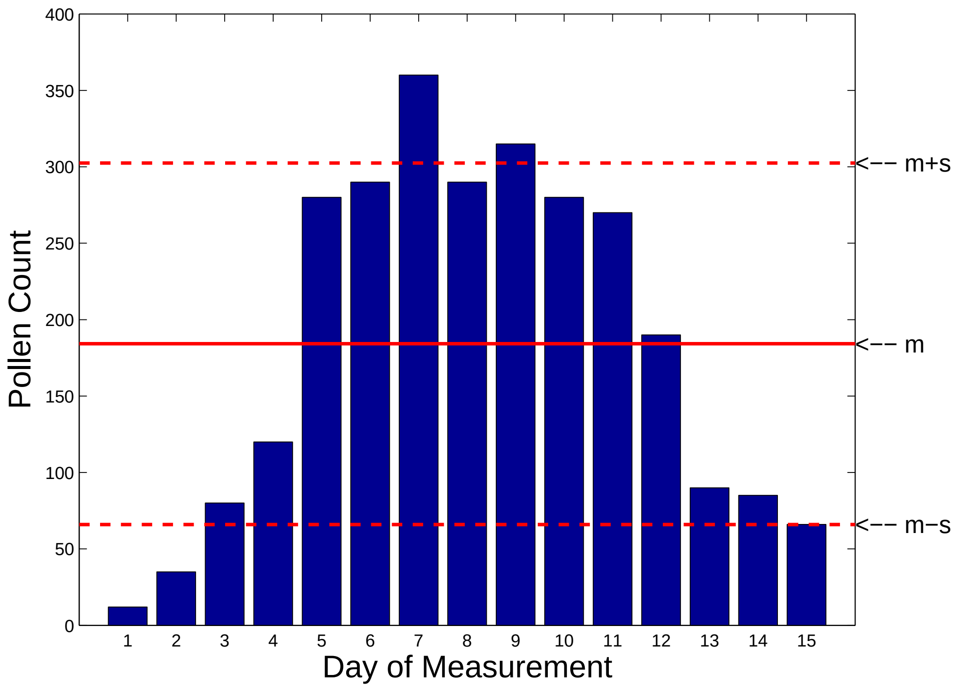


Figure 1.6: Bar graph summarizing the statistical analysis of pollen counts from Example 1.3.3

1.3.4 Creating Reports

Many problems in scientific computing (and in this book) require a combination of mathematical analysis, computational experiments, and a written summary of the results; that is, a report. For simple computer lab reports, a useful tool is MATLAB's `publish` command, which can be used to create a file containing some code and the results it produces. For example, if we enter the code for Example 1.3.1(c) into a script m-file called `Cheby_c.m`, and then use the *Publish* tab, we can create a PDF document that contains the code and the results it produces; see Figure 1.7.

By selecting *Edit Publishing Options ...* under the *Publish* tab you can change the type of format to save the published file. Default is html, but this can easily be changed to PDF, which is often much more convenient for reports and presentations.

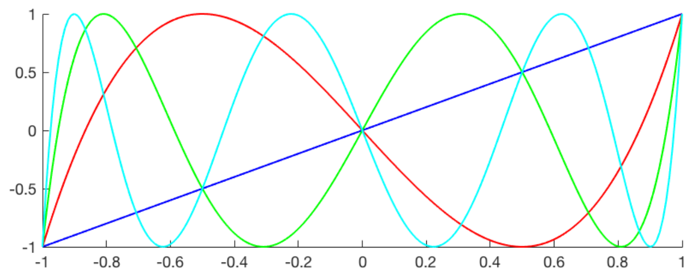
Other things you can do from *Edit Publishing Options ...* include:

- Save only the code, without executing it.
- Save only the result from executing the code, and not the code itself.
- Change the location where the published document is saved on your computer.

```
%
%  Script: Cheby_c
%
%  This creates the figure for the plotting example.  Here
%  we illustrate the use of the hold command.
%
%  Note that we make the line width a bit thicker and the
%  font size a bit larger to make for better displays.
%

x = linspace(-1, 1, 201);
T1 = cos(acos(x));
T3 = cos(3*acos(x));
T5 = cos(5*acos(x));
T7 = cos(7*acos(x));

figure(1), clf
axes('FontSize', 18), hold on
plot(x, T1, 'b', 'LineWidth', 2)
plot(x, T3, 'r', 'LineWidth', 2)
plot(x, T5, 'g', 'LineWidth', 2)
plot(x, T7, 'c', 'LineWidth', 2)
hold off
```



Published with MATLAB® R2017b

Figure 1.7: Example results produced using MATLAB's *Publish* tab.

More extensive reports should be prepared using software such as LaTeX. Plots created in Matlab can be easily printed for inclusion in the report. Probably the easiest approach is to pull down the *File* menu on the desired Matlab figure window, and let go on *Print*.

Another option is to save the plot to a file on your computer using the *Save as* option under the *File* menu on the figure window. Matlab supports many types of file formats, including encapsulated postscript (EPS) and portable document format (PDF), as well as many image compression formats such as TIFF, JPEG and PNG. These files can also be created in the command window, or in any M–file, using the `print` command. For detailed information on creating such files, open the help browser to the associated reference page using the command `doc print`.

An example of a LaTeX document that includes a figure can easily be generated using the *Publish* tool by using *Edit Publishing Options ...* to change the file type to *latex*.

In addition to plots, it may be desirable to create tables of data for a report. Three useful commands that can be use for this purpose are `disp`, `sprintf` and `diary`. When used in combination, `disp` and `sprintf` can be used to write formatted output in the command window. The `diary` command can then be used to save the results in a file. For example, the following Matlab code computes a compound interest table, with annual compounding:

```
a = 35000; n = 5;
r = 1:1:10;
rd = r/100;
t = a*(1 + rd).^n;
diary InterestInfo.dat
disp(' If a = 35,000 dollars is borrowed, to be paid in n = 5 years, then:')
disp(' ')
disp('     Interest rate, r       Total paid after 5 years')
disp('    ==================================================')
for i = 1:length(r)
   disp(sprintf('          %5.2f                    %8.2f     ', r(i), t(i)))
end
diary off
```

If the above code is put into a script or function M–file, then when the code is executed, it prints the following table of data in the command window:

```
 If a = 35,000 dollars is borrowed, to be paid in n = 5 years, then:

     Interest rate, r       Total paid after 5 years
    ==================================================
           1                   36785.35
           2                   38642.83
           3                   40574.59
           4                   42582.85
           5                   44669.85
           6                   46837.90
           7                   49089.31
           8                   51426.48
           9                   53851.84
          10                   56367.85
```

Here, the `diary` command is used to open a file named `InterestInfo.dat`, and any subsequent results displayed in the command window are automatically written in this file until it is closed with the `diary off` command. Thus, after execution, the above table is stored in the file `InterestInfo.dat`.

An alternative approach is to use the `fopen`, `fprintf` and `fclose` commands as follows:

```
a = 35000; n = 5;
r = 1:1:10;
rd = r/100;
t = a*(1 + rd).^n;
fid = fopen('InterestInfo.dat', 'w');
fprintf(fid, ' If a = 35,000 dollars is borrowed, to be paid in n = 5 years, then:\n\n');
fprintf(fid, '      Interest rate, r        Total paid after 5 years\n');
fprintf(fid, '     ==================================================\n');
for i = 1:length(r)
   fprintf(fid, '           %5.2f                    %8.2f\n', r(i), t(i));
end
fclose(fid);
```

In this bit of Matlab code,

- We use `fopen` to request that a file, called `InterestInfo.dat`, be opened for writing. Matlab opens the file, and specifies it with a unique *file indentifier*, which we call `fid`.
- `fprintf` is then used to write data, text, etc., to the file. The commands `\n` indicate that the next printing command should begin on a new line.
- `fclose` is then used to close the file, so it can no longer be modified.

A word of warning about writing to files: If we create a script M–file containing the above code using the `sprintf`, `disp`, and `diary`, and we run that script several times, the file `InterestInfo.dat` will collect the results from each run. In many cases the first set of runs may be used to debug the code, or to get the formatted output to look good, and it is only the final set that we want to save. In this case, it is recommended that you comment out the lines of code containing the `diary` commands (that is, put the symbol % in front of the commands) until you are ready to make the final run.

On the other hand, if we create a script M–file containing the code with `fopen`, `fprintf` and `fclose`, and we run that script several times, the file `InterestInfo.dat` will contain only the results from the final run. This is because we used `'w'` for the file access type in the `fopen` command, which tells Matlab to open the file for writing, and discard any existing contents. If we want to append results to an existing file, without discarding its current contents, then we should replace `'w'` with `'a'` in the `fopen` command. See `doc fopen` for more information.

1.3.5 Defining Mathematical Functions

In many computational science problems it is necessary to evaluate a function. For example, in order to numerically find the maximum of, say, $f(x) = 1/(1 + x^2)$, one way is to evaluate $f(x)$ at many different points. One approach is to write a function M–file, such as:

```
function f = Runge(x)
%
%    f = Runge(x);
%
%  This function evaluates Runge's function, f(x) = 1/(1 + x^2).
%
%  Input:  x - vector of x values
%
%  Output: f - vector of f(x) values.
%
f = 1 ./ (1 + x.*x);
```

To evaluate this function at, say, $x = 1.7$, we can use the MATLAB statement:

```
>> y = Runge(1.7)
```

Or, if we want to plot this function on the interval $-5 \le x \le 5$, we might use the MATLAB statements

```
>> x = linspace(-5, 5, 101);
>> y = Runge(x);
>> plot(x, y)
```

Although this approach works well, it is rather tedious to write a function M–file whose definition is just one line of code. An alternative approach is to use an *anonymous* function:

```
>> Runge = @(x) 1 ./ (1 + x.*x);
```

Here the notation `@(x)` explicitly states that `Runge` is a function of x, and the symbol `@` is referred to as a function handle.

Once `Runge` has been defined, we can evaluate it at, say, $x = 1.7$, using the MATLAB statement:

```
>> y = Runge(1.7);
```

Or, to plot the function on the interval $-5 \le x \le 5$, we might use the MATLAB statements above.

Note that it is always best to use array operators (such as `./` and `.*`) when possible, so that functions can be evaluated at arrays of x-values. For example, if we did not use array operations in the above code, that is, if we had used the statement `Runge = @(x) 1 / (1 + x*x)`, (or the same definition for `f` in the function `Runge`) then we could compute the single value `Runge(1.7)`, but `y = Runge(x)` would produce an error if `x` was a vector, such as that defined by `x = linspace(-5, 5, 101)`.

Anonymous functions provide a powerful and easy to use construct for defining mathematical functions. In general, if $f(x)$ can be defined with a single line of code, we use an anonymous function to define it, otherwise we use a function M–file.

Problem 1.3.11. *The* MATLAB *functions* `tic` *and* `toc` *can be used to find the (wall clock) time it takes to run a piece of code. For example, consider the following* MATLAB *statements:*

```
f = @(x) 1 ./ (1 + x.*x);
tic
for i = 1:n
  x = rand(n,1);
  y = f(x);
end
toc
```

When `tic` *is executed, the timer is started, and when* `toc` *is executed, the timer is stopped, and the time required to run the piece of code between the two statements is displayed in the* MATLAB *command window. Write a script M–file containing these statements. Replace the definition of $f(x)$ by a function M–file. Run the two codes for each of $n = 100, 200, 300, 400, 500$. What do you observe? Repeat this experiment. Do you observe any differences in the timings?*

Problem 1.3.12. *Repeat the experiment in Problem 1.3.11, but this time use*

$$f(x) = \frac{e^x + \sin \pi x}{x^2 + 7x + 4}.$$

Problem 1.3.13. *Write MATLAB code that will create the plot shown in the following Figure 1.8. Your code should be in either a MATLAB script or function m-file. You can use the code given in Example 1.3.2 as a template.*

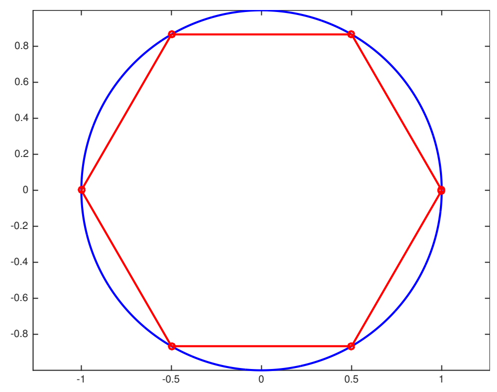


Figure 1.8: Circle with inscribed polygon.

Problem 1.3.14. *To plot a surface, you can use a combination of the MATLAB functions:*

- `linspace` *and* `meshgrid` *to generate a set of* (x, y) *coordinates, and*
- `mesh` *or* `contour` *to plot the surface.*

Read the document pages on these functions, and use them to plot the surfaces of the following (use both `mesh` *and* `contour`*):*

(a) $f(x,y) = (x^2+3y^2)e^{-x^2-y^2}\,, \quad -3 \le x \le 3, \quad -3 \le y \le 3$

(b) $g(x,y) = -3y/(x^2+y^2+1)\,, \quad |x| \le 2, \quad |y| \le 4$

(c) $h(x,y) = |x|+|y|\,, \quad |x| \le 2, \quad |y| \le 1$

Chapter 2

Solution of Linear Systems

Linear systems of equations are ubiquitous in scientific computing – they arise when solving problems in many applications, including biology, chemistry, physics, engineering and economics, and they appear in nearly every chapter of this book. A fundamental numerical problem involving linear systems is that of finding a solution (if one exists) to a set of n linear equations in n unknowns. The first digital computers (developed in the 1940's primarily for scientific computing problems) required about an hour to solve linear systems involving only 10 equations in 10 unknowns. Modern computers are substantially more powerful, and we can now solve linear systems involving thousands of equations in thousands of unknowns in a fraction of a second. Indeed, many problems in science and industry involve millions of equations in millions of unknowns. In this chapter we study the most commonly used algorithm, *Gaussian elimination with partial pivoting*, to solve these important problems.

After a brief introduction to linear systems, we discuss computational techniques for problems (diagonal, lower and upper triangular) that are simple to solve. We then describe Gaussian elimination with partial pivoting, which is an algorithm that reduces a general linear system to one that is simple to solve. Important matrix factorizations associated with Gaussian elimination are described, and issues regarding accuracy of computed solutions are discussed. The chapter ends with a section describing MATLAB implementations, as well as the main tools provided by MATLAB for solving linear systems.

2.1 Linear Systems

A linear system of order n consists of the n linear algebraic equations

$$\begin{array}{c} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n = b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n = b_2 \\ \vdots \\ a_{n,1}x_1 + a_{n,2}x_2 + \cdots + a_{n,n}x_n = b_n \end{array}$$

in the n unknowns $x_1, x_2, \ldots, x_n$. A solution of the linear system is a set of values $x_1, x_2, \ldots, x_n$ that satisfy all n equations simultaneously. Problems involving linear systems are typically formulated using the linear algebra language of matrices and vectors. To do this, first group together the quantities on each side of the equals sign as vectors,

$$\begin{bmatrix} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n \\ a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n \\ \vdots \\ a_{n,1}x_1 + a_{n,2}x_2 + \cdots + a_{n,n}x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

and recall from Section 1.2 that the vector on the left side of the equal sign can be written as a linear combination of column vectors, or equivalently as a matrix–vector product:

$$\begin{bmatrix} a_{1,1} \\ a_{2,1} \\ \vdots \\ a_{n,1} \end{bmatrix} x_1 + \begin{bmatrix} a_{1,2} \\ a_{2,2} \\ \vdots \\ a_{n,2} \end{bmatrix} x_2 + \cdots + \begin{bmatrix} a_{1,n} \\ a_{2,n} \\ \vdots \\ a_{n,n} \end{bmatrix} x_n = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Thus, in matrix–vector notation, the linear system is represented as

$$Ax = b$$

where

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix}$$ is the coefficient matrix of order n,

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$ is the unknown, or solution vector of length n, and

$$b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$ is the right hand side vector of length n.

We consider problems where the coefficients $a_{i,j}$ and the right hand side values b_i are real numbers. A **solution** of the linear system $Ax = b$ of order n is a vector x that satisfies the equation $Ax = b$. The **solution set** of a linear system is the set of all its solutions.

Example 2.1.1. The linear system of equations

$$\begin{aligned} 1x_1 + \ 1x_2 \ + \ 1x_3 \ &= 3 \\ 1x_1 + (-1)x_2 + \ 4x_3 \ &= 4 \\ 2x_1 + \ 3x_2 \ + (-5)x_3 &= 0 \end{aligned}$$

is of order $n = 3$ with unknowns x_1, x_2 and x_3. The matrix–vector form is $Ax = b$ where

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -1 & 4 \\ 2 & 3 & -5 \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \quad b = \begin{bmatrix} 3 \\ 4 \\ 0 \end{bmatrix}$$

This linear system has precisely one solution, given by $x_1 = 1$, $x_2 = 1$ and $x_3 = 1$, so

$$x = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

Linear systems arising in most realistic applications are usually much larger than the order $n = 3$ system of the previous example. However, a lot can be learned about linear systems by looking at small problems. Consider, for example, the 2×2 linear system:

$$\begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \quad \Leftrightarrow \quad \begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 &= b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 &= b_2. \end{aligned}$$

If $a_{1,2} \neq 0$ and $a_{2,2} \neq 0$, then we can write these equations as

$$x_2 = -\frac{a_{1,1}}{a_{1,2}}x_1 + \frac{b_1}{a_{1,2}} \quad \text{and} \quad x_2 = -\frac{a_{2,1}}{a_{2,2}}x_1 + \frac{b_2}{a_{2,2}},$$

which are essentially the slope-intercept form equations of two lines in a plane. Solutions of this linear system consist of all values x_1 and x_2 that satisfy both equations; that is, all points (x_1, x_2) where the two lines intersect. There are three possibilities for this simple example (see Fig. 2.1):

- Unique solution – the lines intersect at only one point.
- No solution – the lines are parallel, with different intercepts.
- Infinitely many solutions – the lines are parallel with the same intercept.

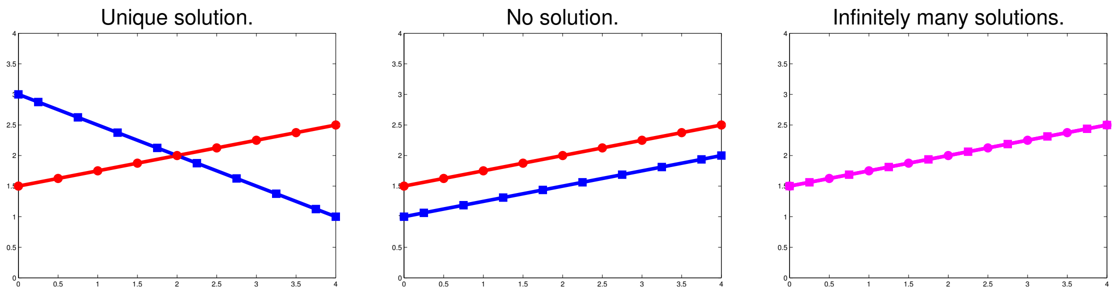


Figure 2.1: Possible solution sets for a general 2×2 linear system. The left plot shows two lines that intersect at only one point (unique solution), the middle plot shows two parallel lines that do not intersect at any points (no solution), and the right plot shows two identical lines (infinitely many solutions).

This conclusion holds for linear systems of any order; that is, **a linear system of order n has either no solution, 1 solution, or an infinite number of distinct solutions**. A linear system $Ax = b$ of order n is **nonsingular** if it has one and only one solution. A linear system is **singular** if it has either no solution or an infinite number of distinct solutions; which of these two possibilities applies depends on the relationship between the matrix A and the right hand side vector b, a matter that is considered in a first linear algebra course.

Whether a linear system $Ax = b$ is singular or nonsingular depends solely on properties of its coefficient matrix A. In particular, the linear system $Ax = b$ is nonsingular if and only if the matrix A is *invertible*; that is, if and only if there is a matrix, A^{-1}, such that $AA^{-1} = A^{-1}A = I$, where I is the identity matrix,

$$I = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1 \end{bmatrix}.$$

So, we say A is nonsingular (invertible) if the linear system $Ax = b$ is nonsingular, and A is singular (non-invertible) if the linear system $Ax = b$ is singular. It is not always easy to determine, a-priori, whether or not a matrix is singular especially in the presence of roundoff errors.

Example 2.1.2. Consider the linear systems of order 2:

(a) $$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5 \\ 6 \end{bmatrix} \quad \Rightarrow \quad \begin{array}{l} x_2 = -\frac{1}{2}x_1 + \frac{5}{2} \\ x_2 = -\frac{3}{4}x_1 + \frac{6}{4} \end{array} \quad \Rightarrow \quad \begin{array}{l} x_2 = -\frac{1}{2}x_1 + \frac{5}{2} \\ x_2 = -\frac{3}{4}x_1 + \frac{3}{2} \end{array}$$

This linear system consists of two lines with unequal slopes. Thus the lines intersect at only one point, and the linear system has a unique solution.

(b) $\begin{bmatrix} 1 & 2 \\ 3 & 6 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5 \\ 6 \end{bmatrix} \quad\Rightarrow\quad \begin{array}{l} x_2 = -\frac{1}{2}x_1 + \frac{5}{2} \\ x_2 = -\frac{3}{6}x_1 + \frac{6}{6} \end{array} \quad\Rightarrow\quad \begin{array}{l} x_2 = -\frac{1}{2}x_1 + \frac{5}{2} \\ x_2 = -\frac{1}{2}x_1 + 1 \end{array}$

This linear system consists of two lines with equal slopes. Thus the lines are parallel. Since the intercepts are not identical, the lines do not intersect at any points, and the linear system has no solution.

(c) $\begin{bmatrix} 1 & 2 \\ 3 & 6 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5 \\ 15 \end{bmatrix} \quad\Rightarrow\quad \begin{array}{l} x_2 = -\frac{1}{2}x_1 + \frac{5}{2} \\ x_2 = -\frac{3}{6}x_1 + \frac{15}{6} \end{array} \quad\Rightarrow\quad \begin{array}{l} x_2 = -\frac{1}{2}x_1 + \frac{5}{2} \\ x_2 = -\frac{1}{2}x_1 + \frac{5}{2} \end{array}$

This linear system consists of two lines with equal slopes. Thus the lines are parallel. Since the intercepts are also equal, the lines are identical, and the linear system has infinitely many solutions.

Problem 2.1.1. *Consider the linear system of Example 2.1.1. If the coefficient matrix A remains unchanged, then what choice of right hand side vector b would lead to the solution vector* $x = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$*?*

Problem 2.1.2. *The determinant of an $n \times n$ matrix, $\det(A)$, is a number that theoretically can be used computationally to indicate if a matrix is singular. Specifically, if $\det(A) \neq 0$ then A is nonsingular. The formula to compute $\det(A)$ for a general $n \times n$ matrix is complicated, but there are some special cases where it can be computed fairly easily. In the case of a 2×2 matrix,*

$$\det\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}\right) = ad - bc.$$

Compute the determinants of the 2×2 matrices in Example 2.1.2. Why do these results make sense?
Important note: The determinant is a good theoretical test for singularity, but it is not a practical test in computational problems. This is discussed in more detail in Section 2.6.

2.2 Simply Solved Linear Systems

Some linear systems are easy to solve. Consider the linear systems of order 3 displayed in Fig. 2.2. By design, the solution of each of these linear systems is $x_1 = 1$, $x_2 = 2$ and $x_3 = 3$. The structure of these linear systems makes them easy to solve; to explain this, we first name the structures exhibited.

$$(a)\quad \begin{array}{rl} (-1)x_1 + 0x_2 + 0x_3 &= -1 \\ 0x_1 + 3x_2 + 0x_3 &= 6 \\ 0x_1 + 0x_2 + (-5)x_3 &= -15 \end{array} \quad\Leftrightarrow\quad \begin{bmatrix} -1 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & -5 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -1 \\ 6 \\ -15 \end{bmatrix}$$

$$(b)\quad \begin{array}{rl} (-1)x_1 + 0x_2 + 0x_3 &= -1 \\ 2x_1 + 3x_2 + 0x_3 &= 8 \\ (-1)x_1 + 4x_2 + (-5)x_3 &= -8 \end{array} \quad\Leftrightarrow\quad \begin{bmatrix} -1 & 0 & 0 \\ 2 & 3 & 0 \\ -1 & 4 & -5 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -1 \\ 8 \\ -8 \end{bmatrix}$$

$$(c)\quad \begin{array}{rl} (-1)x_1 + 2x_2 + (-1)x_3 &= 0 \\ 0x_1 + 3x_2 + 6x_3 &= 24 \\ 0x_1 + 0x_2 + (-5)x_3 &= -15 \end{array} \quad\Leftrightarrow\quad \begin{bmatrix} -1 & 2 & -1 \\ 0 & 3 & 6 \\ 0 & 0 & -5 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 24 \\ -15 \end{bmatrix}$$

Figure 2.2: Simply Solved Linear Systems

We say that the linear system in Fig. 2.2(a) is diagonal, the linear system in Fig. 2.2(b) is lower triangular, and the linear system in Fig. 2.2(c) is upper triangular.

2.2.1 Diagonal Linear Systems

A matrix A of order n is **diagonal** if all its nonzero entries are on its diagonal. (This description of a diagonal matrix *does not state* that the entries on the diagonal are nonzero.) A **diagonal linear system** of equations of order n is one whose coefficient matrix is diagonal. Solving a diagonal linear system of order n, like that in Fig. 2.2(a), is easy because each equation determines the value of one unknown, provided that the diagonal entry is nonzero. So, the first equation determines the value of x_1, the second x_2, etc. A linear system with a diagonal coefficient matrix is singular if it contains a diagonal entry that is zero. In this case the linear system may have no solutions or it may have infinitely many solutions.

Example 2.2.1. In Fig. 2.2(a) the solution is $x_1 = \frac{-1}{(-1)} = 1$, $x_2 = \frac{6}{3} = 2$ and $x_3 = \frac{-15}{(-5)} = 3$.

Example 2.2.2. Consider the following singular diagonal matrix, A, and the vectors b and d:

$$A = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -2 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix}, \quad d = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}.$$

(a) The linear system $Ax = b$ has no solution. Although we can solve the first and last equations to get $x_1 = \frac{1}{3}$ and $x_3 = 0$, it is not possible to solve the second equation:

$$0 \cdot x_1 + 0 \cdot x_2 + 0 \cdot x_3 = -1 \quad \Rightarrow \quad 0 \cdot \frac{1}{3} + 0 \cdot x_2 + 0 \cdot 0 = -1 \quad \Rightarrow \quad 0 \cdot x_2 = -1.$$

Clearly there is no value of x_2 that satisfies the equation $0 \cdot x_2 = -1$.

(b) The linear system $Ax = d$ has infinitely many solutions. From the first and last equations we obtain $x_1 = \frac{1}{3}$ and $x_3 = \frac{1}{2}$. The second equation is

$$0 \cdot x_1 + 0 \cdot x_2 + 0 \cdot x_3 = 0 \quad \Rightarrow \quad 0 \cdot \frac{1}{3} + 0 \cdot x_2 + 0 \cdot \frac{1}{2} = 0 \quad \Rightarrow \quad 0 \cdot x_2 = 0,$$

and thus x_2 can be any real number.

Problem 2.2.1. *Consider the matrix*

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Why is this matrix singular? Find a vector b so that the linear system $Ax = b$ has no solution. Find a vector b so that the linear system $Ax = b$ has infinitely many solutions.

2.2.2 Forward Substitution for Lower Triangular Linear Systems

A matrix A of order n is **lower triangular** if all its nonzero entries are either strictly lower triangular entries or diagonal entries. A **lower triangular linear system** of order n is one whose coefficient matrix is lower triangular. Solving a lower triangular linear system, like that in Fig. 2.2(b), is usually carried out by **forward substitution**. Forward substitution determines first x_1, then x_2, and so on, until all x_i are found. For example, in Fig. 2.2(b), the first equation determines x_1. Given x_1, the second equation then determines x_2. Finally, given both x_1 and x_2, the third equation determines x_3. This process is illustrated in the following example.

Example 2.2.3. In Fig. 2.2(b) the solution is $x_1 = \frac{-1}{(-1)} = 1$, $x_2 = \frac{8-2x_1}{3} = \frac{8-2\cdot 1}{3} = 2$ and $x_3 = \frac{-8-(-1)x_1-4x_2}{(-5)} = \frac{-8-(-1)\cdot 1-4\cdot 2}{(-5)} = 3$.

The forward substitution process can break down if a diagonal entry of the lower triangular matrix is zero. In this case, the lower triangular matrix is singular, and a linear system involving such a matrix may have no solution or infinitely many solutions.

Problem 2.2.2. *Use forward substitution to solve the linear system:*

$$\begin{aligned}
3x_1 + \quad 0x_2 + 0x_3 + \quad 0x_4 &= 6\\
2x_1 + (-3)x_2 + 0x_3 + \quad 0x_4 &= 7\\
1x_1 + \quad 0x_2 + 5x_3 + \quad 0x_4 &= -8\\
0x_1 + \quad 2x_2 + 4x_3 + (-3)x_4 &= -3
\end{aligned}$$

Problem 2.2.3. *Consider the matrix and vector*

$$A = \begin{bmatrix} 3 & 0 & 0 \\ 1 & -2 & 0 \\ -1 & 1 & 0 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ -1 \\ c \end{bmatrix}.$$

Why is A singular? For what values c does the linear system $Ax = b$ have no solution? For what values c does the linear system $Ax = b$ have infinitely many solutions?

2.2.3 Backward Substitution for Upper Triangular Linear Systems

A matrix A of order n is **upper triangular** if its nonzero entries are either strictly upper triangular entries or diagonal entries. An **upper triangular linear system** or order n is one whose coefficient matrix is upper triangular. Solving an upper triangular linear system, like that in Fig. 2.2(c), is usually carried out by **backward substitution**. Backward substitution determines first x_n, then x_{n-1}, and so on, until all x_i are found. For example, in Fig. 2.2(c), the third equation determines the value of x_3. Given the value of x_3, the second equation then determines x_2. Finally, given x_2 and x_3, the first equation determines x_1. This process is illustrated in the following example.

Example 2.2.4. In the case in Fig. 2.2(c) the solution is $x_3 = \frac{-15}{(-5)} = 3$, $x_2 = \frac{24-6x_3}{3} = \frac{24-6\cdot 3}{3} = 2$ and $x_1 = \frac{0-(-1)x_3-2x_2}{(-1)} = \frac{0-(-1)\cdot 3-2\cdot 2}{(-1)} = 1$.

The backward substitution process can break down if a diagonal entry of the upper triangular matrix is zero. In this case, the upper triangular matrix is singular, and a linear system involving such a matrix may have no solution or infinitely many solutions.

Problem 2.2.4. *Use backward substitution to solve the linear system:*

$$\begin{aligned}
2x_1 + 2x_2 + 3x_3 + 4x_4 &= 20\\
0x_1 + 5x_2 + 6x_3 + 7x_4 &= 34\\
0x_1 + 0x_2 + 8x_3 + 9x_4 &= 25\\
0x_1 + 0x_2 + 0x_3 + 10x_4 &= 10
\end{aligned}$$

Problem 2.2.5. *Consider the matrix and vector*

$$A = \begin{bmatrix} -1 & 2 & 3 \\ 0 & 0 & 4 \\ 0 & 0 & 2 \end{bmatrix}, \quad b = \begin{bmatrix} 2 \\ c \\ 4 \end{bmatrix}.$$

Why is A singular? For what values c does the linear system $Ax = b$ have no solution? For what values c does the linear system $Ax = b$ have infinitely many solutions?

Problem 2.2.6. *The determinant of a triangular matrix, be it diagonal, lower triangular, or upper triangular, is the product of its diagonal entries.*

(a) Show that a triangular matrix is nonsingular if and only if each of its diagonal entries is nonzero.

(b) Compute the determinant of each of the triangular matrices in Problems 2.2.2, 2.2.3, 2.2.4 and 2.2.5.

Why do these results make sense computationally?

2.3 Gaussian Elimination with Partial Pivoting

The 19th century German mathematician and scientist Carl Friedrich Gauss described a process, called Gaussian elimination in his honor, that uses two elementary operations systematically to transform any given linear system into one that is easy to solve. (Actually, the process was supposedly known centuries earlier.)

The two elementary operations are

(a) exchange two equations

(b) subtract a multiple of one equation from any other equation.

Applying either type of elementary operations to a linear system does not change its solution set. So, we may apply as many of these operations as needed, in any order, and the resulting system of linear equations has the same solution set as the original system. To implement the procedure, though, we need a systematic approach in which we apply the operations. The most commonly implemented scheme is called Gaussian elimination with partial pivoting (GEPP).

For linear systems of order n, GEPP uses n stages to transform the linear system into upper triangular form. At stage k we eliminate variable x_k from all but the first k equations. To achieve this, each stage uses the same two steps. We illustrate the process on the linear system:

$$\begin{aligned} 1x_1 + \quad 2x_2 + (-1)x_3 &= 0 \\ 2x_1 + (-1)x_2 + \quad 1x_3 &= 7 \\ (-3)x_1 + \quad 1x_2 + \quad 2x_3 &= 3 \end{aligned}$$

Stage 1. Eliminate x_1 from all but the first equation.

The **exchange step**, exchanges equations so that among the coefficients multiplying x_1 in all of the equations, the coefficient in the first equation has largest magnitude. If there is more than one such coefficient, choose the first. If this coefficient with largest magnitude is nonzero, then it is underlined and called the **pivot** for stage 1, otherwise stage 1 had no pivot and we terminate the elimination algorithm. In this example, the pivot occurs in the third equation, so equations 1 and 3 are exchanged.

The **elimination step**, eliminates variable x_1 from all but the first equation. For this example, this involves subtracting $m_{2,1} = \frac{2}{-3}$ times equation 1 from equation 2, and $m_{3,1} = \frac{1}{-3}$ times equation 1 from equation 3. Each of the numbers $m_{i,1}$ is a **multiplier**; the first subscript on $m_{i,1}$ indicates from which equation the multiple of the first equation is subtracted. $m_{i,1}$ is computed as the coefficient of x_1 in equation i divided by the coefficient of x_1 in equation 1 (that is, the pivot).

$$\begin{aligned} 1x_1 + \quad 2x_2 + (-1)x_3 &= 0 \\ 2x_1 + (-1)x_2 + \quad 1x_3 &= 7 \\ \underline{(-3)}\, x_1 + \quad 1x_2 + \quad 2x_3 &= 3 \end{aligned}$$

exchange $\downarrow$ rows 1 and 3

$$\begin{aligned} \underline{(-3)}\, x_1 + \quad 1x_2 + \quad 2x_3 &= 3 \\ 2x_1 + (-1)x_2 + \quad 1x_3 &= 7 \\ 1x_1 + \quad 2x_2 + (-1)x_3 &= 0 \end{aligned}$$

eliminate $\downarrow$ x_1

$$\begin{aligned} \underline{(-3)}\, x_1 + \quad 1x_2 + \quad 2x_3 &= 3 \\ 0x_1 + (-\tfrac{1}{3})x_2 + \quad \tfrac{7}{3}x_3 &= 9 \\ 0x_1 + \quad \tfrac{7}{3}x_2 + (-\tfrac{1}{3})x_3 &= 1 \end{aligned}$$

Stage 2. Eliminate x_2 from all but the first two equations.

Stage 2 repeats the above steps for the variable x_2 on a smaller linear system obtained by removing the first equation from the system at the end of stage 1. This new system involves one fewer unknown (the first stage eliminated variable x_1).

The **exchange step**, exchanges equations so that among the coefficients multiplying x_2 in all of the remaining equations, the second equation has largest magnitude. If this coefficient with largest magnitude is nonzero, then it is underlined and called the **pivot** for stage 2, otherwise stage 2 has no pivot and we terminate. In our example, the pivot occurs in the third equation, so equations 2 and 3 are exchanged.

The **elimination step**, eliminates variable x_2 from all below the second equation. For our example, this involves subtracting $m_{3,2} = \frac{-1/3}{7/3} = -\frac{1}{7}$ times equation 2 from equation 3. The **multiplier** $m_{i,2}$ is computed as the coefficient of x_2 in equation i divided by the coefficient of x_2 in equation 2 (that is, the pivot).

$$\begin{aligned}\underline{(-3)}\,x_1 + 1x_2 + \quad 2x_3 &= 3\\ (-\tfrac{1}{3})x_2 + \quad \tfrac{7}{3}x_3 &= 9\\ \underline{\tfrac{7}{3}}\,x_2 + (-\tfrac{1}{3})x_3 &= 1\end{aligned}$$

exchange $\downarrow$ rows 2 and 3

$$\begin{aligned}\underline{(-3)}\,x_1 + 1x_2 + \quad 2x_3 &= 3\\ \underline{\tfrac{7}{3}}\,x_2 + (-\tfrac{1}{3})x_3 &= 1\\ (-\tfrac{1}{3})x_2 + \quad \tfrac{7}{3}x_3 &= 9\end{aligned}$$

eliminate $\downarrow$ x_2

$$\begin{aligned}\underline{(-3)}\,x_1 + 1x_2 + \quad 2x_3 &= 3\\ \underline{\tfrac{7}{3}}\,x_2 + (-\tfrac{1}{3})x_3 &= 1\\ 0x_2 + \quad \tfrac{16}{7}x_3 &= \tfrac{64}{7}\end{aligned}$$

This process continues until the last equation involves only one unknown variable, and the transformed linear system is in upper triangular form. The last stage of the algorithm simply involves identifying the coefficient in the last equation as the final pivot element. In our simple example, we are done at stage 3, where we identify the last pivot element by underlining the coefficient for x_3 in the last equation, and obtain the upper triangular linear system

$$\begin{aligned}\underline{(-3)}\,x_1 + 1x_2 + \quad 2x_3 &= 3\\ \underline{\tfrac{7}{3}}\,x_2 + (-\tfrac{1}{3})x_3 &= 1\\ \underline{\tfrac{16}{7}}\,x_3 &= \tfrac{64}{7}\end{aligned}$$

GEPP is now finished. When the entries on the diagonal of the final upper triangular linear system are nonzero, as it is in our illustrative example, the linear system is nonsingular and its solution may be determined by backward substitution.

The diagonal entries of the upper triangular linear system produced by GEPP play an important role. Specifically, the k^{th} diagonal entry, i.e., the coefficient of x_k in the k^{th} equation, is the pivot for the k^{th} stage of GEPP.

We emphasize that **if GEPP does not find a non-zero pivot at some stage, we can conclude immediately that the original linear system is singular**. Consequently, many GEPP codes simply terminate elimination and return a message that indicates the original linear system is singular.

Example 2.3.1. The previous discussion showed that GEPP transforms the linear system to upper triangular form:

$$\begin{aligned}1x_1 + \quad 2x_2 + (-1)x_3 &= 0\\ 2x_1 + (-1)x_2 + \quad 1x_3 &= 7\\ (-3)x_1 + \quad 1x_2 + \quad 2x_3 &= 3\end{aligned} \quad \to \quad \cdots \quad \to \quad \begin{aligned}\underline{(-3)}\,x_1 + 1x_2 + \quad 2x_3 &= 3\\ \underline{\tfrac{7}{3}}\,x_2 + (-\tfrac{1}{3})x_3 &= 1\\ \underline{\tfrac{16}{7}}\,x_3 &= \tfrac{64}{7}\end{aligned}$$

Using backward substitution, we find that the solution of this linear system is given by:

$$x_3 = \frac{\frac{64}{7}}{\frac{16}{7}} = 4\,, \quad x_2 = \frac{1 + \frac{1}{3}\cdot 4}{\frac{7}{3}} = 1\,, \quad x_1 = \frac{3 - 1\cdot 1 - 2\cdot 4}{-3} = 2\,.$$

Example 2.3.2. Consider the linear system:

$$\begin{aligned} 4x_1 + \quad 6x_2 + (-10)x_3 &= 0 \\ 2x_1 + \quad 2x_2 + \quad 2x_3 &= 6 \\ 1x_1 + (-1)x_2 + \quad 4x_3 &= 4 \end{aligned}$$

Applying GEPP to this example we obtain:

In the first stage, the largest coefficient in magnitude of x_1 is already in the first equation, so no exchange steps are needed. The elimination steps then proceed with the multipliers $m_{2,1} = \frac{2}{4} = 0.5$ and $m_{3,1} = \frac{1}{4} = 0.25$.

$$\begin{aligned} \underline{4}\,x_1 + \quad 6x_2 + (-10)x_3 &= 0 \\ 2x_1 + \quad 2x_2 + \quad 2x_3 &= 6 \\ 1x_1 + (-1)x_2 + \quad 4x_3 &= 4 \end{aligned}$$

$$\downarrow$$

$$\begin{aligned} \underline{4}\,x_1 + \quad 6x_2 + (-10)x_3 &= 0 \\ 0x_1 + \quad (-1)x_2 + \quad 7x_3 &= 6 \\ 0x_1 + \underline{(-2.5)}\,x_2 + \quad 6.5x_3 &= 4 \end{aligned}$$

In the second stage, we observe that the largest x_2 coefficient in magnitude in the last two equations occurs in the third equation, so the second and third equations are exchanged. The elimination step then proceeds with the multiplier $m_{3,2} = \frac{-1}{-2.5} = 0.4$.

$$\downarrow$$

$$\begin{aligned} \underline{4}\,x_1 + \quad 6x_2 + (-10)x_3 &= 0 \\ 0x_1 + \underline{(-2.5)}\,x_2 + \quad 6.5x_3 &= 4 \\ 0x_1 + \quad (-1)x_2 + \quad 7x_3 &= 6 \end{aligned}$$

$$\downarrow$$

In the final stage we identify the final pivot entry, and observe that all pivots are non-zero, and thus the linear system is nonsingular and there is a unique solution.

$$\begin{aligned} \underline{4}\,x_1 + \quad 6x_2 + (-10)x_3 &= 0 \\ 0x_1 + \underline{(-2.5)}\,x_2 + \quad 6.5x_3 &= 4 \\ 0x_1 + \quad 0x_2 + \underline{4.4}\,x_3 &= 4.4 \end{aligned}$$

Using backward substitution, we find the solution of the linear system:

$$x_3 = \frac{4.4}{4.4} = 1\,, \quad x_2 = \frac{4 - 6.5 \cdot 1}{-2.5} = 1\,, \quad x_1 = \frac{0 - 6 \cdot 1 + 10 \cdot 1}{4} = 1\,.$$

Example 2.3.3. Consider the linear system:

$$\begin{aligned} 1x_1 + (-2)x_2 + (-1)x_3 &= 2 \\ (-1)x_1 + \quad 2x_2 + (-1)x_3 &= 1 \\ 3x_1 + (-6)x_2 + \quad 9x_3 &= 0 \end{aligned}$$

Applying GEPP to this example we obtain:

In the first stage, the largest coefficient in magnitude of x_1 is in the third equation, so we exchange the first and third equations. The elimination steps then proceed with the multipliers $m_{2,1} = \frac{-1}{3}$ and $m_{3,1} = \frac{1}{3}$.

$$\begin{aligned} 1x_1 + (-2)x_2 + (-1)x_3 &= 2 \\ (-1)x_1 + \quad 2x_2 + (-1)x_3 &= 1 \\ \underline{3}\,x_1 + (-6)x_2 + \quad 9x_3 &= 0 \end{aligned}$$

$$\downarrow$$

$$\begin{aligned} \underline{3}\,x_1 + (-6)x_2 + \quad 9x_3 &= 0 \\ (-1)x_1 + \quad 2x_2 + (-1)x_3 &= 1 \\ 1x_1 + (-2)x_2 + (-1)x_3 &= 2 \end{aligned}$$

In the second stage, we observe that all coefficients multiplying x_2 in the last two equations are zero. We therefore fail to find a non-zero pivot, and conclude that the linear system is singular.

$$\downarrow$$

$$\begin{aligned} \underline{3}\,x_1 + (-6)x_2 + \quad 9x_3 &= 0 \\ 0x_1 + \quad 0x_2 + \quad 2x_3 &= 1 \\ 0x_1 + \quad 0x_2 + (-4)x_3 &= 2 \end{aligned}$$

Problem 2.3.1. *Use GEPP followed by backward substitution to solve the linear system of Example 2.1.1. Explicitly display the value of each pivot and each multiplier.*

Problem 2.3.2. *Use GEPP followed by backward substitution to solve the following linear system. Explicitly display the value of each pivot and multiplier.*

$$\begin{aligned}
3x_1 + \quad 0x_2 + 0x_3 + \quad 0x_4 &= 6\\
2x_1 + (-3)x_2 + 0x_3 + \quad 0x_4 &= 7\\
1x_1 + \quad 0x_2 + 5x_3 + \quad 0x_4 &= -8\\
0x_1 + \quad 2x_2 + 4x_3 + (-3)x_4 &= -3
\end{aligned}$$

Problem 2.3.3. *Use GEPP and backward substitution to solve the following linear system. Explicitly display the value of each pivot and multiplier.*

$$\begin{aligned}
2x_1 + x_3 &= 1\\
x_2 + 4x_3 &= 3\\
x_1 + 2x_2 &= -2
\end{aligned}$$

Problem 2.3.4. *GEPP provides an efficient way to compute the determinant of any matrix. Recall that the operations used by GEPP are (1) exchange two equations, and (2) subtract a multiple of one equation from another (different) equation. Of these two operations, only the exchange operation changes the value of the determinant of the matrix of coefficients, and then it only changes its sign. In particular, suppose GEPP transforms the coefficient matrix A into the upper triangular coefficient matrix U using m actual exchanges, i.e., exchange steps where an exchange of equations actually occurs. Then*

$$\det(A) = (-1)^m \det(U).$$

(a) Use GEPP to show that

$$\det\left(\begin{bmatrix} 4 & 6 & -10\\ 2 & 2 & 2\\ 1 & -1 & 4 \end{bmatrix}\right) = (-1)^1 \det\left(\begin{bmatrix} 4 & 6 & -10\\ 0 & -2.5 & 6.5\\ 0 & 0 & 4.4 \end{bmatrix}\right) = -(4)(-2.5)(4.4) = 44$$

Recall from Problem 2.2.6 that the determinant of a triangular matrix is the product of its diagonal entries.

(b) Use GEPP to calculate the determinant of each of the matrices:

$$\begin{bmatrix} 1 & 1 & 1\\ 2 & 1 & 2\\ 4 & -3 & 0 \end{bmatrix} \quad \textit{and} \quad \begin{bmatrix} 1 & 1 & 1 & 1\\ 2 & 3 & 4 & 5\\ -1 & 2 & -2 & 1\\ 2 & 6 & 3 & 7 \end{bmatrix}$$

2.4 Gaussian Elimination and Matrix Factorizations

The concept of matrix factorizations is fundamentally important in the process of numerically solving linear systems, $Ax = b$. The basic idea is to decompose the matrix A into a product of *simply solved systems*, from which the solution of $Ax = b$ can be easily computed. The idea is similar to what we might do when trying to find the roots of a polynomial. For example, the equations

$$x^3 - 6x^2 + 11x - 6 = 0 \quad \text{and} \quad (x-1)(x-2)(x-3) = 0$$

are equivalent, but the factored form is clearly much easier to solve. In general, we cannot solve linear systems so easily (i.e., by inspection), but decomposing A makes solving the linear system $Ax = b$ computationally simpler.

When we apply Gaussian elimination with partial pivoting by rows to reduce A to upper triangular form, we obtain an LU factorization of a *permuted* version of A. That is,

$$PA = LU$$

where P is a permutation matrix representing *all* row interchanges in the order that they are applied.

If we can compute the factorization $PA = LU$, then

$$Ax = b \;\Rightarrow PAx = Pb \;\Rightarrow\; LUx = Pb \;\Rightarrow\; Ly = Pb, \text{ where } Ux = y.$$

Therefore, to solve $Ax = b$:

- Compute the factorization $PA = LU$.
- Permute entries of b to obtain $d = Pb$.
- Solve $Ly = d$ using forward substitution.
- Solve $Ux = y$ using backward substitution.

It is important to emphasize the importance, and power of matrix factorizations. The cost of computing $A = LU$ or $PA = LU$ requires $O(n^3)$ FLOPS, but forward and backward solves require only $O(n^2)$ FLOPS[1]. Thus, if we need to solve multiple linear systems where the matrix A does not change, but with different right hand side vectors, such as

$$Ax_1 = b_1\,, \quad Ax_2 = b_2 \quad \cdots$$

then we need only compute the (relatively expensive) matrix factorization once, and reuse it for all linear system solves.

Matrix factorizations might be useful for other calculations, such as computing the determinant of A. Recall the following properties of determinants:

- The determinant of a product of matrices is the product of their determinants. Thus, in particular,
$$\det(PA) = \det(P)\det(A) \quad \text{and} \quad \det(LU) = \det(L)\det(U)\,.$$
- The determinant of the permutation matrix P is ± 1; it is $+1$ if an even number of row swaps were performed, and -1 if odd number of row swaps were performed.
- The determinant of a triangular matrix is the product of its diagonal entries. In particular, $\det(L) = 1$ because it is a unit lower triangular matrix.

Using these properties, we see that

$$\begin{aligned} \det(PA) &= \det(LU) \\ \det(P)\det(A) &= \det(L)\det(U) \\ \pm\det(A) &= \det(U) \\ \det(A) &= \pm\det(U) = \pm u_{11}u_{22}\cdots u_{nn} \end{aligned}$$

Thus, once we have the $PA = LU$ factorization, it is trivial to compute the determinant of A.

[1]A floating point operation (FLOP) is basic operation (addition, subtraction, multiplication, division) performed by the computer. Counting FLOPs gives a rough idea of the cost (time) it takes for the computer to complete the algorithm; the more FLOPs means more expensive.

Example 2.4.1. Consider the linear system, $Ax = b$, where

$$A = \begin{bmatrix} 1 & 2 & 4 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

The $PA = LU$ factorization is given by

$$P = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \quad L = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{7} & 1 & 0 \\ \frac{4}{7} & \frac{1}{2} & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 7 & 8 & 9 \\ 0 & \frac{6}{7} & \frac{19}{7} \\ 0 & 0 & -\frac{1}{2} \end{bmatrix}$$

Since the $PA = LU$ factorization is given, to solve $Ax = b$ we need only:

- Obtain $d = Pb = \begin{bmatrix} 3 \\ 1 \\ 2 \end{bmatrix}$
- Solve $Ly = d$, or

$$\begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{7} & 1 & 0 \\ \frac{4}{7} & \frac{1}{2} & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 1 \\ 2 \end{bmatrix}$$

  Using forward substitution, we obtain

$$\begin{aligned} & y_1 = 3 \\ & \tfrac{1}{7}y_1 + y_2 = 1 \Rightarrow y_2 = 1 - \tfrac{1}{7}(3) = \tfrac{4}{7} \\ & \tfrac{4}{7}y_1 + \tfrac{1}{2}y_2 + y_3 = 2 \Rightarrow y_3 = 2 - \tfrac{4}{7}(3) - \tfrac{1}{2}(\tfrac{4}{7}) = 0 \end{aligned}$$

- Solve $Ux = y$, or

$$\begin{bmatrix} 7 & 8 & 9 \\ 0 & \frac{6}{7} & \frac{19}{7} \\ 0 & 0 & -\frac{1}{2} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ \frac{4}{7} \\ 0 \end{bmatrix}$$

  Using backward substitution, we obtain

$$\begin{aligned} & -\tfrac{1}{2}x_3 = 0 \Rightarrow x_3 = 0. \\ & \tfrac{6}{7}x_2 + \tfrac{19}{7}x_3 = \tfrac{4}{7} \Rightarrow x_2 = \tfrac{7}{6}(\tfrac{4}{7} - \tfrac{19}{7}(0)) = \tfrac{2}{3}. \\ & 7x_1 + 8x_2 + 9x_3 = 3 \Rightarrow x_1 = \tfrac{1}{7}(3 - 8(\tfrac{2}{3}) - 9(0)) = -\tfrac{1}{3}. \end{aligned}$$

  Therefore, the solution of $Ax = b$ is given by

$$x = \begin{bmatrix} -\frac{1}{3} \\ \frac{2}{3} \\ 0 \end{bmatrix}.$$

factorization of the matrices:

$$A = \begin{bmatrix} 1 & 3 & -4 \\ 0 & -1 & 5 \\ 2 & 0 & 4 \end{bmatrix}, \; B = \begin{bmatrix} 3 & 6 & 9 \\ 2 & 5 & 2 \\ -3 & -4 & -11 \end{bmatrix}, \; C = \begin{bmatrix} 2 & -1 & 0 & 3 \\ 0 & -\frac{3}{4} & \frac{1}{2} & 4 \\ 1 & 1 & 1 & -\frac{1}{2} \\ 2 & -\frac{5}{2} & 1 & 13 \end{bmatrix}$$

Problem 2.4.1. *Suppose that* $b^T = \begin{bmatrix} 3 & 60 & 1 & 5 \end{bmatrix}$, *and suppose that Gaussian elimination with partial pivoting has been used on a matrix* A *to obtain its* $PA = LU$ *factorization, where*

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \quad L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -3/4 & 1 & 0 & 0 \\ 1/4 & 0 & 1 & 0 \\ 1/2 & -1/5 & 1/3 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 4 & 8 & 12 & -8 \\ 0 & 5 & 10 & -10 \\ 0 & 0 & -6 & 6 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Use this factorization (do not compute the matrix A*) to solve* $Ax = b$*.*

Problem 2.4.2. *Use the* $PA = LU$ *factorizations of the matrices in the previous two problems to compute* $\det(A)$*.*

2.5 Other Matrix Factorizations

As mentioned in the previous section, the idea of matrix factorization is very important and powerful when performing linear algebra computations. There are many types of matrix factorization, besides $PA = LU$, that may be preferred for certain problems. In this section we consider three other matrix factorizations: Cholesky, QR, and singular value decomposition (SVD).

2.5.1 Cholesky factorization

A matrix $A \in \mathcal{R}^{n\times n}$ is *symmetric* if $A = A^T$, which means that the entries are symmetric about the main diagonal.

Example 2.5.1. Consider the matrices

$$A = \begin{bmatrix} 3 & 0 & -1 & 5 \\ 0 & 2 & 4 & 8 \\ -1 & 4 & 1 & -2 \\ 5 & 8 & -2 & 6 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 3 & 0 & -1 & 5 \\ 1 & 2 & 4 & 8 \\ 0 & 3 & 1 & -2 \\ -2 & 1 & 1 & 6 \end{bmatrix}$$

then

$$A^T = \begin{bmatrix} 3 & 0 & -1 & 5 \\ 0 & 2 & 4 & 8 \\ -1 & 4 & 1 & -2 \\ 5 & 8 & -2 & 6 \end{bmatrix} = A \quad \text{but} \quad B^T = \begin{bmatrix} 3 & 1 & 0 & -2 \\ 0 & 2 & 3 & 1 \\ -1 & 4 & 1 & 1 \\ 5 & 8 & -2 & 6 \end{bmatrix} \neq B\,.$$

Thus A is symmetric, but B is *not* symmetric.

A matrix $A \in \mathcal{R}^{n\times n}$ is *positive definite* if $x^T Ax > 0$ for all $x \in \mathcal{R}^n$, $x \neq 0$. The special structure of a symmetric and positive definite matrix (which we will abbreviate as SPD) allows us to compute a special LU factorization, which is called the *Cholesky factorization*. Specifically, it can be shown that an $n \times n$ matrix A is SPD **if and only if** it can be factored as[2]

$$A = R^T R \quad \text{(called the Cholesky factorization)}$$

where R is an upper triangular matrix with positive entries on the diagonal. Pivoting is generally not needed to compute this factorization. The **"if and only if"** part of the above statement is important. This means that if we are given a matrix A, and the Cholesky factorization fails, then we know A is not SPD, but if it succeeds, then we know A is SPD.

[2]In some books, the Cholesky factorization is defined as $A = LL^T$ where L is lower triangular. This is the same as the notation used in this book (which better matches what is the default from computed in MATLAB), with $L = R^T$.

To give a brief outline on how to compute a Cholesky factorization, it is perhaps easiest to begin with a small 3×3 example. That is, consider

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{12} & a_{22} & a_{23} \\ a_{13} & a_{23} & a_{33} \end{bmatrix}.$$

To find the Cholesky factorization:

- Set $R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ 0 & r_{22} & r_{23} \\ 0 & 0 & r_{33} \end{bmatrix}$

- Form the matrix $R^TR = \begin{bmatrix} r_{11}^2 & r_{11}r_{12} & r_{11}r_{13} \\ r_{11}r_{12} & r_{12}^2 + r_{22}^2 & r_{12}r_{13} + r_{22}r_{23} \\ r_{11}r_{13} & r_{12}r_{13} + r_{22}r_{23} & r_{13}^2 + r_{23}^2 + r_{33}^2 \end{bmatrix}$

- Now set $A = R^TR$ and match corresponding components to solve for r_{ij}. That is,

$$\begin{aligned}
r_{11}^2 = a_{11} \quad &\Rightarrow \quad r_{11} = \sqrt{a_{11}} \\
r_{11}r_{12} = a_{12} \quad &\Rightarrow \quad r_{12} = a_{12}/r_{11} \\
r_{11}r_{13} = a_{13} \quad &\Rightarrow \quad r_{13} = a_{13}/r_{11} \\
r_{12}^2 + r_{22}^2 = a_{22} \quad &\Rightarrow \quad r_{22} = \sqrt{a_{22} - r_{12}^2} \\
r_{13}r_{12} + r_{23}r_{22} = a_{23} \quad &\Rightarrow \quad r_{23} = (a_{23} - r_{13}r_{12})/r_{22} \\
r_{13}^2 + r_{23}^2 + r_{33}^2 = a_{33} \quad &\Rightarrow \quad r_{33} = \sqrt{a_{33} - r_{13}^2 - r_{23}^2}
\end{aligned}$$

The above process can easily be generalized for any $n \times n$ SPD matrix. We skip efficient implementation details, but make two observations. First, generally we do not need to consider pivoting when computing the Cholesky factorization of an SPD matrix, and the values inside the square root symbols are always positive. The second observation is that because the matrix is symmetric, it should not be surprising that an efficient implementation costs approximately half the number of FLOPS needed for standard $PA = LU$ factorizations.

Solving linear systems with the Cholesky factorization is essentially the same as with $PA = LU$. That is, if $A = R^TR$, then

$$Ax = b \quad \Rightarrow \quad R^TRx = b \quad \Rightarrow \quad R^T(Rx) = b\,,$$

and so to compute x,

- use forward substitution to solve $R^Ty = b$, and

- use backward substitution to solve $Rx = y$.

Problem 2.5.1. *Compute the Cholesky factorization of the matrix:*

$$A = \begin{bmatrix} 4 & 1 & 1 \\ 1 & 3 & -1 \\ 1 & -1 & 2 \end{bmatrix}$$

Problem 2.5.2. *Compute the Cholesky factorization of the matrix:*

$$A = \begin{bmatrix} 1 & 2 & -1 \\ 2 & 8 & -4 \\ -1 & -4 & 6 \end{bmatrix}$$

Problem 2.5.3. *Compute the Cholesky factorization of*

$$A = \begin{bmatrix} 25 & 15 & -5 \\ 15 & 25 & 1 \\ -5 & 1 & 6 \end{bmatrix}$$

and use the factorization to solve $Ax = b$, where

$$b = \begin{bmatrix} -5 \\ -7 \\ 12 \end{bmatrix} .$$

2.5.2 QR factorization

A matrix $Q \in \mathcal{R}^{n\times n}$ is called *orthogonal* if the columns of Q form an orthonormal set. That is, if we write

$$Q = \begin{bmatrix} q_1 & q_2 & \cdots & q_n \end{bmatrix} ,$$

where q_j is the jth column of Q, then

$$q_i^T q_j = \begin{cases} 1 & \text{if} \quad i = j \\ 0 & \text{if} \quad i \neq j \end{cases} .$$

This means that if Q is an orthogonal matrix, then

$$Q^T Q = \begin{bmatrix} q_1^T \\ q_2^T \\ \vdots \\ q_n^T \end{bmatrix} \begin{bmatrix} q_1 & q_2 & \cdots & q_n \end{bmatrix} = \begin{bmatrix} q_1^T q_1 & q_1^T q_2 & \cdots & q_1^T q_n \\ q_2^T q_1 & q_2^T q_2 & \cdots & q_2^T q_n \\ \vdots & \vdots & & \vdots \\ q_n^T q_1 & q_n^T q_2 & \cdots & q_n^T q_n \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} .$$

That is, $Q^T Q = I$, and thus the inverse of Q is simply Q^T. This is a very nice property!

A first course on linear algebra often covers a topic called Gram-Schmidt orthonormalization, which transforms a linearly independent set of vectors into an orthonormal set. We will not review the Gram-Schmidt method in this book, but state that if it is applied to the columns of a nonsingular matrix $A \in \mathcal{R}^{n\times n}$, then it results in a matrix factorization of the form

$$A = QR ,$$

where $Q \in \mathcal{R}^{n\times n}$ is an orthogonal matrix, and $R \in \mathcal{R}^{n\times n}$ is upper triangular. This is called the QR factorization of A.

We remark that if the columns of the (possibly over-determined rectangular) matrix $A \in \mathcal{R}^{m\times n}$, $m \geq n$, then it is still possible to compute a QR factorization of A. This will be discussed when we consider the topic of least squares in the chapter on curve fitting. We also remark that there are other (often better) approaches than Gram-Schmidt for computing $A = QR$ (e.g., Householder and Givens methods), but these are best left for a more advanced course on numerical linear algebra.

Computing solutions of $Ax = b$ with the QR factorization is also straight forward:

$$Ax = b \quad \Rightarrow \quad QRx = b \quad \Rightarrow \quad Rx = Q^T b ,$$

and so to solve $Ax = b$,

- compute $d = Q^T b$, and
- use backward substitution to solve $Rx = d$.

Although we do not discuss algorithms for computing $A = QR$ in this book, we should note that if $A \in \mathcal{R}^{n\times n}$ is nonsingular, then an efficient implementation requires $O(n^3)$ FLOPS. But the hidden constant in the $O(\cdot)$ notation is approximately two times that for the $PA = LU$ factorization. Thus, $PA = LU$ is usually the preferred factorization for solving $n \times n$ nonsingular systems of equations. However, the QR factorization is superior for solving least squares problems.

2.5.3 Singular Value Decomposition

We end this section with arguably the most important matrix factorization. Let $A \in \mathcal{R}^{m\times n}$, $m \geq n$. Then there exist orthogonal matrices

$$\begin{aligned} U &= \begin{bmatrix} u_1 & u_2 & \cdots & u_m \end{bmatrix} \in \mathcal{R}^{m\times m} \\ V &= \begin{bmatrix} v_1 & v_2 & \cdots & v_n \end{bmatrix} \in \mathcal{R}^{n\times n} \end{aligned}$$

and a diagonal matrix

$$\Sigma = \text{diag}(\sigma_1, \sigma_2, \ldots, \sigma_n) = \begin{bmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_n \\ & & \end{bmatrix} \in \mathcal{R}^{m\times n}$$

such that $A = U\Sigma V^T$, with $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$. The factorization, $A = U\Sigma V^T$ is called the *singular value decomposition* (SVD).

As with the QR factorization, algorithms for computing the SVD are very complicated, and best left for a more advanced course on numerical linear algebra. Here we just define the decomposition, and discuss some of its properties. We use the following notation and terminology:

- σ_i are called *singular values* of the matrix A.
- u_i, which are the columns of U, are called *left singular vectors* of the matrix A.
- v_i, which are the columns of V, are called *right singular vectors* of the matrix A.

Notice that, because V is an orthogonal matrix, we know $V^TV = I$. Thus, for the case $m \geq n$,

$$A = U\Sigma V^T \quad \Rightarrow \quad AV = U\Sigma$$

and so,

$$A \begin{bmatrix} v_1 & \cdots & v_n \end{bmatrix} = \begin{bmatrix} u_1 & \cdots & u_n & u_{n+1} & \cdots & u_m \end{bmatrix} \begin{bmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_n \\ & & \end{bmatrix}.$$

The above can be written as:

$$\begin{bmatrix} Av_1 & \cdots & Av_n \end{bmatrix} = \begin{bmatrix} \sigma_1 u_1 & \cdots & \sigma_n u_n \end{bmatrix}$$

That is,

$$Av_i = \sigma_i u_i\,,\ i = 1, 2, \ldots, n$$

The SVD has the following properties:

- If rank$(A) = r$, then
$$\sigma_1 \geq \cdots \geq \sigma_r > \sigma_{r+1} = \cdots = \sigma_n = 0$$
. In particular, if A is $n \times n$ and nonsingular, then all singular values are nonzero.
- If rank$(A) = r$, then the nullspace of A is:
$$\text{null}(A) = \text{span}\{v_{r+1},\ \ldots,\ v_n\}$$
That is, $Ax = 0$ if and only if x is a linear combination of the vectors $v_{r+1},\ \ldots,\ v_n$.

- If rank$(A) = r$, then the range space of A is:

$$\text{range}(A) = \text{span}\{u_1, \ldots, u_r\}$$

Computing solutions of $Ax = b$ with the SVD is straight forward:

$$Ax = b \quad \Rightarrow \quad U\Sigma V^T x = b \quad \Rightarrow \quad \Sigma V^T x = U^T b\,,$$

and so to solve $Ax = b$,

- compute $d = U^T b$,
- solve the diagonal system $\Sigma y = d$,
- compute $x = Vy$.

Although we do not discuss algorithms for computing the SVD in this book, we should note that if $A \in \mathcal{R}^{n\times n}$ is nonsingular, then an efficient implementation requires $O(n^3)$ FLOPS. But the hidden constant in the $O(\cdot)$ notation is approximately nine times that for the QR factorization, and eighteen times that for the $PA = LU$ factorization. Because it is so expensive, it is rarely used to solve linear systems, but it is a superior tool to analyze sensitivity of linear systems, and it finds use in important applications such as rank deficient least squares problems, principle component analysis (PCA), and even data compression.

Problem 2.5.4. *Our definition of the SVD assumes that $m \geq n$ (that is, A has at least as many rows as columns). Show that a similar definition, and hence decomposition, can be written for the case $m < n$ (that is, A has more columns than rows).*
Hint: Consider using our original definition of the SVD for A^T.

Problem 2.5.5. *Show that $A^T u_i = \sigma_i v_i\,,\ i = 1, 2, \ldots, n$.*

Problem 2.5.6. *Suppose $A \in \mathcal{R}^{n\times n}$. Show that det$(A) = \pm\sigma_1\sigma_2\cdots\sigma_n$.*
Hint: First show that for any orthogonal matrix U, det$(U) = \pm 1$.

Problem 2.5.7. *In this problem we consider relationships between singular values and eigenvalues. Recall from your basic linear algebra class that if $B \in \mathcal{R}^{n\times n}$, then λ is an* eigenvalue *of B if there is a nonzero vector $x \in \mathcal{R}^n$ such that*

$$Bx = \lambda x\,.$$

The vector x is called an eigenvector *of B. There are relationships between singular values/vectors and eigenvalue/vectors. To see this, assume $A \in \mathcal{R}^{m\times n}$, $m \geq n$, and $A = U\Sigma V^T$ is the SVD of A. Then from above, we know:*

$$Av_i = \sigma_i u_i \quad \textit{and} \quad A^T u_i = \sigma_i v_i\,.$$

Using these relationships, show:

- *$A^T A v_i = \sigma_i^2 v_i$, and thus σ_i^2 is an eigenvalue of $A^T A$ with corresponding eigenvector v_i.*
- *$AA^T u_i = \sigma_i^2 u_i$, and thus σ_i^2 is an eigenvalue of AA^T with corresponding eigenvector u_i.*
- *If A is square and symmetric, that is $A = A^T$, then the singular values of A are the absolute values of the eigenvalues of A.*

2.6 The Accuracy of Computed Solutions

Methods for determining the accuracy of the computed solution of a linear system are discussed in more advanced courses in numerical linear algebra. However, in this section we attempt to at least qualitatively describe some of the factors affecting the accuracy.

2.6.1 Vector norms

First we need a tool to measure errors between vectors. Suppose we have a vector $\hat{x}$ that is an approximation of the vector x. How do we determine if $\hat{x}$ is a good approximation of x? It may seem natural to consider the error vector $e = \hat{x} - x$ and determine if e is small. However, e is a vector with possibly many entries, so what does it mean to say "e is small"? To answer this question, we need the concept of *vector norm*, which uses the notation $\| \cdot \|$, and must satisfy the following properties:

1. $\|v\| \geq 0$ for all vectors $v \in \mathcal{R}^n$, and $\|v\| = 0$ if and only if $v = 0$ (that is, the vector with all zero entries),
2. $\|v + w\| \leq \|v\| + \|w\|$ for all vectors $v \in \mathcal{R}^n$ and $w \in \mathcal{R}^n$,
3. $\|cv\| = |c|\|v\|$ for all vectors $v \in \mathcal{R}^n$ and all scalars c.

There many vector norms, so sometimes we include a subscript, such as $\| \cdot \|_p$, to indicate precisely which norm we are using. Here are some examples:

- The 2-norm is the standard Euclidean length of a vector taught in multivariable calculus and linear algebra courses. Specifically, if

$$e = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_n \end{bmatrix}$$

  then we define the vector 2-norm as

$$\|e\|_2 = \sqrt{e^T e} = \sqrt{e_1^2 + e_2^2 + \cdots + e_n^2}\,.$$

- The vector 1-norm is defined as

$$\|e\|_1 = |e_1| + |e_2| + \cdots + |e_n|\,.$$

- The vector ∞-norm is defined as

$$\|e\|_\infty = \max_{1 \leq i \leq n} \{|e_i|\}\,.$$

- In general, if $1 \leq p < \infty$, then the p-norm is defined as

$$\|e\|_p = \left(\sum_{i=1}^{n} |e_i|^p \right)^{1/p}\,.$$

Although other norms are used in certain applications, we usually use the 2-norm. However, any norm gives us a single number, and if the norm of the error,

$$\|e\| = \|\hat{x} - x\|$$

is small, then we say the error is small.

We should note that "small" may be relative to the magnitude of the values in the vector x, and thus it is perhaps better to use the **relative error**,

$$\frac{\|\hat{x} - x\|}{\|x\|}$$

provided $x \neq 0$.

Example 2.6.1. Suppose

$$x = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \quad \text{and} \quad \hat{x} = \begin{bmatrix} 0.999 \\ -1.001 \end{bmatrix}$$

then

$$\hat{x} - x = \begin{bmatrix} -10^{-3} \\ -10^{-3} \end{bmatrix}$$

and so, using the 2-norm, we get

$$\|\hat{x} - x\|_2 = \sqrt{10^{-6} + 10^{-6}} = \sqrt{2} \cdot 10^{-3} \approx 0.0014142$$

and the relative error

$$\frac{\|\hat{x} - x\|_2}{\|x\|_2} = \frac{\sqrt{2} \cdot 10^{-3}}{\sqrt{2}} = 10^{-3}\,.$$

Problem 2.6.1. *Consider the previous example, and compute the relative error using the 1-norm and the ∞-norm.*

Problem 2.6.2. *If $x = \begin{bmatrix} 1 & 2 & -3 & 0 & 1 \end{bmatrix}^T$, compute $\|x\|_1$, $\|x\|_2$, and $\|x\|_\infty$.*

2.6.2 Matrix norms

The idea of vector norms can be extended to matrices. Formally, we say that $\|\cdot\|$ is a *matrix norm* if it satisfies the following properties:

1. $\|A\| \geq 0$ for all matrices $A \in \mathcal{R}^{m\times n}$, and $\|A\| = 0$ if and only if $A = 0$ (that is, the matrix with all zero entries),
2. $\|A + B\| \leq \|A\| + \|B\|$ for all matrices $A \in \mathcal{R}^{m\times n}$ and $B \in \mathcal{R}^{m\times n}$,
3. $\|cA\| = |c|\|A\|$ for all matrices $A \in \mathcal{R}^{m\times n}$ and all scalars c.

Given what we know about vector norms, it may be perhaps most natural to first consider the *Frobenius* matrix norm, which is defined as

$$\|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij}^2}\,.$$

Other matrix norms that are often used in scientific computing are the class of *p-norms*, which are said to be *induced by* the corresponding vector norms, and are defined as

$$\|A\|_p = \max_{x\neq 0} \frac{\|Ax\|_p}{\|x\|_p}\,.$$

This is not a very useful definition for actual computations, but fortunately there are short cut formulas that can be used for three of the most popular matrix p-norms. We will not prove these, but proofs can be found in more advanced books on numerical analysis.

- The matrix 2-norm is defined as
$$\|A\|_2 = \max_{x\neq 0} \frac{\|Ax\|_2}{\|x\|_2}\,,$$
but can be computed as
$$\|A\|_2 = \sigma_1\,,$$
where σ_1 is the largest singular value of A.
- The matrix 1-norm is defined as
$$\|A\|_1 = \max_{x\neq 0} \frac{\|Ax\|_1}{\|x\|_1}\,,$$
but can be computed as
$$\|A\|_1 = \max_{1\le j\le n} \sum_{i=1}^{m} |a_{ij}|\,,$$
that is, the maximum column sum.
- The matrix ∞-norm is defined as
$$\|A\|_\infty = \max_{x\neq 0} \frac{\|Ax\|_\infty}{\|x\|_\infty}\,,$$
but can be computed as
$$\|A\|_\infty = \max_{1\le i\le m} \sum_{j=1}^{n} |a_{ij}|\,,$$
that is, the maximum row sum.

The induced matrix norms satisfy two important and useful properties,
$$\|Ax\| \le \|A\|\|x\|,$$
and
$$\|AB\| \le \|A\|\|B\|\,,$$
provided the matrix multiplication of AB is defined.

Problem 2.6.3. *If*
$$A = \begin{bmatrix} 1 & -2 & 3 \\ 2 & 0 & 5 \\ -1 & 1 & -1 \\ 2 & 4 & 0 \end{bmatrix},$$
compute $\|A\|_1$, $\|A\|_\infty$, and $\|A\|_F$.

Problem 2.6.4. *In the case of the Frobenius norm, show that*
$$\|A\|_F^2 = \sum_{j=1}^{n} \|a_j\|_2^2 = trace(A^TA) = trace(AA^T)$$
where a_j is the j-th column vector of the matrix A, and trace is the sum of diagonal entries of the given matrix.

Problem 2.6.5. *In general, for an induced matrix norm, the following inequality holds:*

$$\|Ax\| \le \|A\|\|x\|$$

In some special (important) cases, equality holds. In particular, assume Q is an orthogonal matrix, and show that

$$\|Qx\|_2 = \|x\|_2 \,.$$

This property says that the Euclidean length (2-norm) of the vector x does not change if the vector is modified by an orthogonal transformation. In this case, we say the 2-norm is invariant under orthogonal transformations.

Problem 2.6.6. *We know that if $A = U\Sigma V^T$ then*

$$\|A\|_2 = \sigma_1 \quad \textit{(largest singular value of } A\textit{).}$$

Show that if A is nonsingular, then

$$\|A^{-1}\|_2 = \frac{1}{\sigma_n} \quad \textit{(reciprocal of the smallest singular value of } A\textit{).}$$

Hint: If $A = U\Sigma V^T$ is the SVD or A, what is the SVD of A^{-1}?

2.6.3 Measuring accuracy of computed solutions

Suppose we compute an approximate solution, $\hat{x}$, of $Ax = b$. How do we determine if $\hat{x}$ is a good approximation of x? If we know the exact solution, then we can simply compute the relative error,

$$\frac{\|\hat{x} - x\|}{\|x\|}$$

using any vector norm.

If we do not know the exact solution, then we may try to see if the computed solution is a good fit to the data. That is, we consider the *residual error*,

$$\|r\| = \|b - A\hat{x}\| \,,$$

or the relative residual error,

$$\frac{\|r\|}{\|b\|} = \frac{\|b - A\hat{x}\|}{\|b\|} \,.$$

We might ask the question:

> *If the relative residual error is small, does this mean $\hat{x}$ is a good approximation of the exact solution x?*

Unfortunately the answer is: Not always. Consider the following example.

Example 2.6.2. Consider the matrix

$$A = \begin{bmatrix} 0.835 & 0.667 \\ 0.333 & 0.266 \end{bmatrix}, \quad b = \begin{bmatrix} 0.168 \\ 0.067 \end{bmatrix}.$$

It is easy to verify that the exact solution to $Ax = b$ is $x = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$. Suppose we somehow compute the approximation $\hat{x} = \begin{bmatrix} 267 \\ -334 \end{bmatrix}$, which is clearly a very poor approximation of x. But the residual vector is

$$r = b - A\hat{x} \approx \begin{bmatrix} 0.001000000000019 \\ 0.000000000000003 \end{bmatrix},$$

and the relative residual error is

$$\frac{\|r\|_2}{\|b\|_2} = \frac{\|b - A\hat{x}\|_2}{\|b\|_2} \approx 0.005528913725860\,.$$

Thus in this example a small residual **does not imply** $\hat{x}$ is a good approximation of x.

Why does this happen? We can gain a little insight by examining a simple 2×2 linear system,

$$Ax = b \quad \Rightarrow \quad \begin{array}{c} a_{11}x_1 + a_{12}x_2 = b_1 \\ a_{21}x_1 + a_{22}x_2 = b_2 \end{array}$$

and assume that $a_{12} \neq 0$ and $a_{22} \neq 0$. Each equation is a line, which then can be written as

$$x_2 = -\frac{a_{11}}{a_{12}}x_1 + \frac{b_1}{a_{12}} \quad \text{and} \quad x_2 = -\frac{a_{21}}{a_{22}}x_1 + \frac{b_2}{a_{22}}\,.$$

The solution of the linear system, $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ is the point where the two lines intersect. Consider the following two very different cases (it might also help to look at the plots in Figure 2.1):

> **Case 1:** The slopes, $-\frac{a_{11}}{a_{12}}$ and $-\frac{a_{21}}{a_{22}}$ are very different, e.g., the two lines are nearly perpendicular. Then small changes in b or A (e.g., due to round off error) will not dramatically change the intersection point. In this case, *the rows of A are linearly independent – very far from being linearly dependent, and A is very far from being singular.* In this case, we say the matrix A, and hence the linear system $Ax = b$, is **well-conditioned**.
>
> **Case 2:** The slopes, $-\frac{a_{11}}{a_{12}}$ and $-\frac{a_{21}}{a_{22}}$ are nearly equal, e.g., the two lines are nearly parallel. In this case, small changes in b or A (e.g., due to round off error) can cause a dramatic change the intersection point. Here, *the rows of A are nearly linearly dependent, and thus A is very close to being singular.* In this case we say that the matrix A, and hence the linear system $Ax = b$, is **ill-conditioned**.

This idea of conditioning can be extended to larger systems. In general, if the matrix is nearly singular (i.e., the columns or rows are nearly linearly dependent), then we say the matrix A is ill-conditioned.

So far our discussion of conditioning is a bit vague, and it would be nice to have a formal definition and/or way to determine if a matrix is ill-conditioned. We can do this by recalling the SVD; that is, if $A = U\Sigma V^T$, where

$$\Sigma = \text{diag}(\sigma_1, \sigma_2, \ldots, \sigma_n)\,,$$

where $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$. Recall that the rank($A$) is the number of nonzero singular values. That is, A is singular if the smallest singular value, $\sigma_n = 0$.

Now suppose A is nonsingular, so that $\sigma_n \neq 0$. How do we determine if A "is nearly singular"? One way is to consider the ratio of the largest and smallest singular values, $\frac{\sigma_1}{\sigma_n}$. This ratio will be ≈ 1 if the matrix is well-conditioned (e.g., as in the case of the identity matrix, I), and very large if the matrix is ill-conditioned. We know that $\|A\|_2 = \sigma_1$, and from Problem 2.6.6 we also know that $\|A^{-1}\|_2 = \frac{1}{\sigma_n}$. Thus, we can define the **condition number associated with the matrix 2-norm** to be

$$\kappa_2(A) = \|A\|_2\|A^{-1}\|_2 = \frac{\sigma_1}{\sigma_n}\,.$$

More generally, for any matrix norm, we define the condition number as

$$\kappa(A) = \|A\|\|A^{-1}\|\,.$$

Example 2.6.3. Consider the matrix

$$A = \begin{bmatrix} 0.835 & 0.667 \\ 0.333 & 0.266 \end{bmatrix}, \quad b = \begin{bmatrix} 0.168 \\ 0.067 \end{bmatrix}.$$

Using MATLAB's `svd` function, we find that $\sigma_1 \approx 1.1505e+00$ and $\sigma_2 \approx 8.6915e-07$, and hence

$$\kappa_2(A) = \|A\|_2\|A^{-1}\|_2 = \frac{\sigma_1}{\sigma_n} \approx 1.3238e+06\,.$$

This is a very large number, and so we conclude that A is ill-conditioned.

Let us now return back to the question:

If the relative residual error is small, does this mean $\hat{x}$ is a good approximation of the exact solution x?

To see how the residual relates to the relative error, suppose A is a nonsingular matrix and $\hat{x}$ is an approximate solution of $Ax = b$. Then using an induced matrix norm (e.g., 2-norm), we obtain:

- $r = b - A\hat{x} = Ax - A\hat{x} = A(x - \hat{x})$, which means
$$x - \hat{x} = A^{-1}r\,. \tag{2.1}$$
- If we take norms on both sides of (2.1), we obtain
$$\|x - \hat{x}\| = \|A^{-1}r\| \le \|A^{-1}\|\|r\|\,. \tag{2.2}$$
- Next observe that $Ax = b$ implies $\|b\| = \|Ax\| \le \|A\|\|x\|$, and so
$$\frac{1}{\|x\|} \le \frac{\|A\|}{\|b\|}\,. \tag{2.3}$$
- Putting together the inequalities (2.2) and (2.3), we obtain:
$$\frac{\|x - \hat{x}\|}{\|x\|} \le \|A\|\|A^{-1}\|\frac{\|r\|}{\|b\|} = \kappa(A)\|\frac{\|r\|}{\|b\|}\,. \tag{2.4}$$

The result in (2.4) is important! It tells us:

- If the matrix A is well conditioned, e.g. $\kappa(A) \approx 1$, and if the relative residual is small, then we can be sure to have an accurate solution.
- However, if A is ill-conditioned (e.g., $\kappa(A)$ is very large), then the relative error can be large even if the relative residual is small. Notice that we **cannot** say "will be large" because the result is given in terms of an upper bound. But the fact that it "can be large" is important to know, and should be taken into account when attempting to solve ill-conditioned linear systems.

We conclude this subsection with the following remarks. The two categories, **well-conditioned** and **ill-conditioned** are separated by a grey area. That is, while we can say that a matrix with condition number $\kappa(A)$ in the range of 1 to 100 would be considered well-conditioned, and a condition number $\kappa(A) > 10^8$ is considered ill-conditioned, there is a large "grey" area between these extremes, and it depends on machine precision. Although we cannot get rid of this grey area, we can say that if the condition number of A is about 10^p and the machine epsilon, ϵ, is about 10^{-s} then the solution of the linear system $Ax = b$ *may* have no more than about $s - p$ decimal digits accurate. Recall that in SP arithmetic, $s \approx 7$, and in DP arithmetic, $s \approx 16$.

We should keep in mind that a well-conditioned linear system has the property that *all small changes* in the matrix A and the right hand side b lead to a small change in the solution of $Ax = b$, and that an ill-conditioned linear system has the property that *some small changes* in the matrix A and/or the right hand side b can lead to a large change in the solution of $Ax = b$.

2.7 Matlab Notes

Software is available for solving linear systems where the coefficient matrices have a variety of structures and properties. We restrict our discussion to "dense" systems of linear equations. (A "dense" system is one where all the coefficients are treated as non-zero values. So, the matrix is considered to have no special structure.) Most of today's best software for solving "dense" systems of linear equations, including that found in Matlab, was developed in the *LAPACK* project. We describe the main tools (i.e., the backslash operator and the `linsolve` function) provided by Matlab for solving dense linear systems. These, and other useful built-in Matlab functions that are relevant to the topics of this chapter, and which will be discussed in this section, include:

`linsolve`	used to solve linear systems $Ax = b$
\ (backslash)	used to solve linear systems $Ax = b$
`lu`	used to compute $A = LU$ and $PA = LU$ factorizations
`cond`	used to compute condition number of a matrix
`triu`	used to get upper triangular part of a matrix
`tril`	used to get lower triangular part of a matrix
`diag`	used to get diagonal part of a matrix, or to make a diagonal matrix

2.7.1 Built-in Matlab Tools for Linear Systems

An advantage of using a powerful scientific computing environment like Matlab is that we do not need to write our own implementations of standard algorithms, like Gaussian elimination and triangular solves. Matlab provides two powerful tools for solving linear systems:

- The *backslash* operator: \
  Given a matrix A and vector b, \ can be used to solve $Ax = b$ with the single command:

  ```
  x = A \ b;
  ```

  Matlab first checks if the matrix A has a special structure, including diagonal, upper triangular, and lower triangular. If a special structure is recognized (for example, upper triangular), then a special method (for example, column-oriented backward substitution) is used to solve $Ax = b$. If a special structure is not recognized then Gaussian elimination with partial pivoting is used to solve $Ax = b$. During the process of solving the linear system, Matlab estimates the *reciprocal* of the condition number of A. If A is ill–conditioned, then a warning message is printed along with the estimate, `RCOND`, of the reciprocal of the condition number.

- The function `linsolve`.
  If we know a-priori that A has a special structure recognizable by Matlab, then we can improve efficiency by avoiding checks on the matrix, and skipping directly to the special solver. This can be especially helpful if, for example, it is known that A is upper triangular. A-priori information on the structure of A can be provided to Matlab using the `linsolve` function. For more information, see `help linsolve` or `doc linsolve`.

Because the backslash operator is so powerful, we use it almost exclusively to solve general linear systems.

We can compute explicitly the $PA = LU$ factorization using the `lu` function:

```
[L, U, P] = lu(A)
```

Given this factorization, and a vector b, we could solve the $Ax = b$ using the statements:

```
d = P * b;, y = L \ d;, x = U \ y;
```

These statements could be combined into one instruction:

```
x = U \ ( L \ (P * b) );
```

Thus, given a matrix A and vector b we could solve the linear system $Ax = b$ as follows:

```
[L, U, P] = lu(A);
x = U \ ( L \ (P * b) );
```

Note, MATLAB follows the rules of operator precedence thus backslash and `*` are of equal precedence. With operators of equal precedence MATLAB works from the left. So, without parentheses the instruction

```
x = U \ L \ P * b ;
```

would be interpreted as

```
x = ((U \ L) \ P) * b;
```

The cost and accuracy of this approach is essentially the same as using the backslash operator. So when would we prefer to explicitly compute the $PA = LU$ factorization? One situation is when we need to solve several linear systems with the same coefficient matrix, but different right hand side vectors. For large systems it is far more expensive to compute the $PA = LU$ factorization than it is to use forward and backward substitution to solve corresponding lower and upper triangular systems. Thus, if we can compute the factorization just once, and use it for the various different right hand side vectors, we can make a substantial savings. This is illustrated in problem 2.7.4, which involves a relatively small linear system.

Problem 2.7.1. *Use the backslash operator to solve the systems given in problems 2.3.2 and 2.3.3.*

Problem 2.7.2. *Use the backslash operator to solve the linear system*

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} .$$

Explain your results.

Problem 2.7.3. *Consider the linear system defined by the following* MATLAB *commands:*

```
A = eye(500) + triu(rand(500));, x = ones(500,1);, b = A * x;
```

Does this matrix have a special structure? Suppose we slightly perturb the entries in A:

```
C = A + eps*rand(500);
```

Does the matrix C have a special structure? Execute the following MATLAB *commands:*

```
tic, x1 = A \ b;, toc
tic, x2 = C \ b;, toc
tic, x3 = triu(C) \ b;, toc
```

Are the solutions `x1`, `x2` *and* `x3` *good approximations to the exact solution,* `x = ones(500,1)`*? What do you observe about the time required to solve each of the linear systems?*

Problem 2.7.4. *Create a script M–file containing the following* MATLAB *statements:*

```
n = 50;, A = rand(n);
tic
for k = 1:n
  b = rand(n,1);, x = A \ b;
end
toc

tic
[L, U, P] = lu(A);
for k = 1:n
  b = rand(n,1);, x = U \ ( L \ (P * b) );
end
toc
```

The dimension of the problem is set to $n = 50$. Experiment with other values of n, such as $n = 100, 150, 200$. What do you observe?

2.7.2 Matlab Tools for Other Matrix Factorizations

Just like the built-in Matlab function `lu` function can be used to compute the $PA = LU$ factorization, Matlab has tools to find other matrix factorizations, including:

- `R = chol(A)` can be used to compute the Cholesky factorization, $A = R^T R$, provided A is symmetric positive definite (if A is not SPD, an error message will be produced).
- `[Q, R] = qr(A)` can be used to compute the QR factorization, $A = QR$.
- `[U, S, V] = svd(A)` can be used to compute the SVD, $A = USV^T$ (here we used the letter S instead of Σ for the diagonal matrix containing the singular values).

Problem 2.7.5. *Suppose*

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Find each of the following matrix factorizations:

(a) $PA = LU$. What does the matrix U tell you about A?

(b) Cholesky, $A = R^T R$. Why should you expect to get an error message when using `chol`*)?*

(c) $A = QR$. What does the matrix R tell you about A?

(d) SVD, $A = USV^T$. What does S tell you about A?

Problem 2.7.6. *Suppose*

$$A = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix}$$

Obviously A is a symmetric matrix. Show that A is also positive definite by using `chol` *to find the Cholesky factorization.*

Chapter 3

Nonlinear Equations and Root Finding

We were introduced to the concept of finding roots of a function in basic algebra courses. For example, we learned the quadratic formula so that we could factor quadratic polynomials, and we found x-intercepts provided key points when we sketched graphs. In many practical applications, though, we are presented with problems involving very complicated functions for which basic algebraic techniques cannot be used, and we must resort to computational methods. In this chapter we consider the problem of computing roots of a general nonlinear function of one variable. We describe and analyze several techniques, including the well known Newton iteration. We show how the Newton iteration can be applied to the rather special, but important, problem of calculating square roots. We also discuss how to calculate real roots in the special case when the nonlinear function is a polynomial of arbitrary degree. The chapter ends with a discussion of implementation details, and we describe how to use some of MATLAB's built-in root finding functions.

3.1 Root Finding Methods

Let us turn now to finding roots – that is, points x_* such that $f(x_*) = 0$. The first two methods we consider, the Newton and secant iterations, are closely related to the fixed–point iteration discussed in Section **??**. We then describe the bisection method, which has a slower rate of convergence than the Newton and secant iterations, but has the advantage that convergence can be guaranteed for relative accuracies greater than ϵ_{DP}. The final method we discuss is rather different than the first three, and is based on inverse interpolation.

3.1.1 The Newton Iteration

Suppose $f(x)$ is a given function, and we wish to compute a simple root of $f(x) = 0$. A systematic procedure for constructing an equivalent fixed–point problem, $x = g(x)$, is to choose the **Newton Iteration Function**

$$g(x) = x - \frac{f(x)}{f'(x)}\,.$$

The associated fixed–point iteration $x_{n+1} = g(x_n)$ leads to the well–known **Newton Iteration**

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}\,.$$

Notice that if x_* satisfies $x_* = g(x_*)$, and if $f'(x_*) \neq 0$, then $f(x_*) = 0$, and hence x_* is a simple root of $f(x) = 0$.

How does this choice of iteration function arise? Here are two developments:

- An algebraic derivation can be obtained by expanding $f(z)$ around x using a Taylor series:

$$f(z) = f(x) + (z-x)f'(x) + (z-x)^2 \frac{f''(x)}{2} + \cdots$$

A Taylor polynomial approximation is obtained by terminating the infinite series after a finite number of terms. In particular, a linear polynomial approximation is given by

$$f(z) \approx f(x) + (z-x)f'(x) \,.$$

Setting $z = x_{n+1}$, $f(x_{n+1}) = 0$ and $x = x_n$ we obtain

$$0 \approx f(x_n) + (x_{n+1} - x_n)f'(x_n) \,.$$

Rearranging this relation gives the Newton iteration. Note, this is "valid" because we assume x_{n+1} is much closer to x_* than is x_n, so, usually, $f(x_{n+1})$ is much closer to zero than is $f(x_n)$.

- For a geometric derivation, observe that the curve $y = f(x)$ crosses the x-axis at $x = x_*$. In point-slope form, the tangent to $y = f(x)$ at the point $(x_n, f(x_n))$ is given by $y - f(x_n) = f'(x_n)(x - x_n)$. Let x_{n+1} be defined as the value of x where this tangent crosses the x-axis (see Fig. 3.1). Then the point $(x_{n+1}, 0)$ is on the tangent line, so we obtain

$$0 - f(x_n) = f'(x_n)(x_{n+1} - x_n) \,.$$

Rearranging this equation, we obtain the Newton iteration.

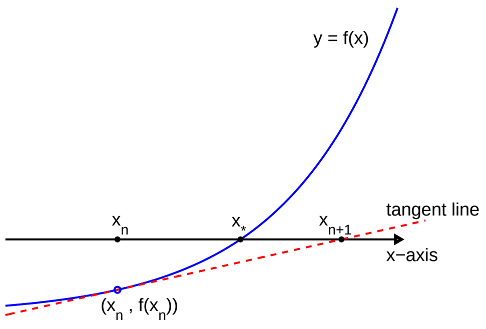


Figure 3.1: Illustration of Newton's method. x_{n+1} is the location where the tangent line to $y = f(x)$ through the point $(x_n, f(x_n))$ crosses the x-axis.

Example 3.1.1. Suppose $f(x) = x - e^{-x}$, and consider using Newton's method to compute a root of $f(x) = 0$. In this case, $f'(x) = 1 + e^{-x}$, and the Newton iteration is

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n - e^{-x_n}}{1 + e^{-x_n}} = \frac{e^{x_n}(x_n + 1)}{1 + e^{x_n}} \,.$$

Example 3.1.2. Suppose $f(x) = x^3 - x - 1$, and consider using Newton's method to compute a root of $f(x) = 0$. In this case, $f'(x) = 3x^2 - 1$, and the Newton iteration is

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n^3 - x_n - 1}{3x_n^2 - 1} = \frac{2x_n^3 + 1}{3x_n^2 - 1} \,.$$

Summarizing basic convergence properties for a simple root x_* of f:

- The Newton iteration is locally convergent, which means the initial guess x_0 needs to be "sufficiently close" to the root x_*.
- The number of correct digits in x_n approximately doubles at each Newton iteration. This is called *quadratic convergence.*

Example 3.1.3. Consider the equation $f(x) \equiv x^{20} - 1 = 0$ with real roots ± 1. If we use the Newton iteration the formula is

$$x_{n+1} = x_n - \frac{x_n^{20} - 1}{20x_n^{19}}$$

Starting with $x_0 = 0.5$, which we might consider to be "close" to the root $x_* = 1$ we compute $x_1 = 26213$, $x_2 = 24903$, $x_3 = 23658$ to five digits. The first iteration takes an enormous leap across the root then the iterations "creep" back towards the root. So, we didn't start "close enough" to get quadratic convergence; the iteration is converging linearly. If, instead, we start from $x_0 = 1.5$, to five digits we compute $x_1 = 1.4250$, $x_2 = 1.3538$ and $x_3 = 1.2863$, and convergence is again linear. If we start from $x_0 = 1.1$, to five digits we compute $x_1 = 1.0532$, $x_2 = 1.0192$, $x_3 = 1.0031$, and $x_4 = 1.0001$. We observe quadratic convergence in this last set of iterates.

Problem 3.1.1. *Construct the Newton iteration by deriving the tangent equation to the curve $y = f(x)$ at the point $(x_n, f(x_n))$ and choosing the point x_{n+1} as the place where this line crosses the x-axis. (See Fig. 3.1.)*

Problem 3.1.2. *Some computers perform the division $\frac{N}{D}$ by first computing the reciprocal $\frac{1}{D}$ and then computing the product $\frac{N}{D} = N \cdot \frac{1}{D}$. Consider the function $f(x) \equiv D - \frac{1}{x}$. What is the root of $f(x)$? Show that, for this function $f(x)$, the Newton iteration function $g(x) = x - \frac{f(x)}{f'(x)}$ can be written so that evaluating it does not require divisions. Simplify this iteration function so that evaluating it uses as few adds, subtracts, and multiplies as possible.*

Use the iteration that you have derived to compute $\frac{N}{D} = \frac{2}{3}$ without using any divisions.

3.1.2 Secant Iteration

The major disadvantage of the Newton iteration is that the function f' must be evaluated at each iteration. This derivative may be difficult to compute or may be far more expensive to evaluate than the function f. One quite common circumstance where it is difficult to derive f' is when f is defined by a computer program. Recently, the technology of Automatic Differentiation has made computing automatically the values of derivatives defined in such complex ways a possibility but methods that do not require values of $f'(x)$ remain in widespread use.

When the cost of evaluating the derivative is the problem, this cost can often be reduced by using instead the iteration

$$x_{n+1} = x_n - \frac{f(x_n)}{m_n},$$

where m_n is an approximation of $f'(x_n)$. For example, we could use a *difference quotient*

$$m_n = \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}.$$

This difference quotient uses the already–computed values $f(x)$ from the current and the previous iterate. It yields the **secant iteration**:

$$\begin{aligned} x_{n+1} &= x_n - \frac{f(x_n)}{\left(\dfrac{f(x_n)-f(x_{n-1})}{x_n - x_{n-1}}\right)} \\ &= x_n - \left(\frac{f(x_n)}{f(x_n)-f(x_{n-1})}\right)(x_n - x_{n-1}) \quad \text{[preferred computational form]} \\ &= \frac{x_n f(x_{n-1}) - x_{n-1} f(x_n)}{f(x_{n-1}) - f(x_n)} \quad \text{[computational form to be avoided]} \end{aligned}$$

This iteration requires two starting values x_0 and x_1. Of the three expressions for the iterate x_{n+1}, the second expression

$$x_{n+1} = x_n - \left(\frac{f(x_n)}{f(x_n)-f(x_{n-1})}\right)(x_n - x_{n-1}), \quad n = 1, 2, \ldots$$

provides the "best" implementation. It is written as a *correction* of the most recent iterate in terms of a scaling of the difference of the current and the previous iterate. All three expressions suffer from cancelation; in the last, cancelation can be catastrophic.

When the error e_n is small enough the secant iteration converges, and it can be shown that the error ultimately behaves like $|e_{n+1}| \approx K|e_n|^p$ where $p = \dfrac{1+\sqrt{5}}{2} \approx 1.618$ and K is a positive constant. (Derivation of this result is not trivial.) The value $\dfrac{1+\sqrt{5}}{2}$ is the well-known golden ratio which we will meet again in the next chapter. Since $p > 1$, the error exhibits *superlinear convergence* but at a rate less than that exemplified by quadratic convergence. However, this superlinear convergence property has a similar effect on the behavior of the error as the quadratic convergence property of the Newton iteration. As long as $f''(x)$ is continuous in an interval containing x_* and the root x_* is simple, we have the following properties:

- If the initial iterates x_0 and x_1 are both sufficiently close to the root x_* then the secant iteration is guaranteed to converge to x_*.
- When the iterates x_n and x_{n-1} are sufficiently close to the root x_* then the secant iteration converges superlinearly; that is, the number of correct digits increases faster than linearly at each iteration. Ultimately the number of correct digits increases by a factor of approximately 1.6 per iteration.

As with the Newton iteration we may get convergence on larger intervals containing x_* than we observe superlinearity.

Problem 3.1.3. *Derive the secant iteration geometrically as follows: Write down the formula for the straight line joining the two points $(x_{n-1}, f(x_{n-1}))$ and $(x_n, f(x_n))$, then for the next iterate x_{n+1} choose the point where this line crosses the x-axis. (The figure should be very similar to Fig. 3.1.)*

3.1.3 Other Methods

Other methods for finding roots include the *bisection method* and *quadratic inverse interpolation.* Discussion of these methods can be found in standard books on numerical analysis.

3.2 Calculating Square Roots

How does a computer or a calculator compute the square root of a positive real number? In fact, how did engineers calculate square roots before computers were invented? There are many approaches, one dating back to the ancient Babylonians! For example, we could use the identity

$$\sqrt{a} = e^{\frac{1}{2}\ln a}$$

and then use tables of logarithms to find approximate values for $\ln(a)$ and e^b. In fact, some calculators still use this identity along with good routines to calculate logarithms and exponentials.

In this section we describe how to apply root finding techniques to calculate square roots. For any real value $a > 0$, the function $f(x) = x^2 - a$ has two square roots, $x_* = \pm\sqrt{a}$. The Newton iteration applied to $f(x)$ gives

$$\begin{aligned} x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} &= x_n - \frac{x_n^2 - a}{2x_n} \\ &= \frac{\left(x_n + \dfrac{a}{x_n}\right)}{2} \end{aligned}$$

Note that we only need to perform simple addition, multiplication and division operations to compute x_{n+1}. The (last) algebraically simplified expression is less expensive to compute than the original expression because it involves just one divide, one add, and one division by two, whereas the original expression involves a multiply, two subtractions, a division and a multiplication by two. [When using IEEE Standard floating–point arithmetic, multiplication or division by a power of two of a machine floating–point number involves simply increasing or reducing, respectively, that number's exponent by one, known as a binary shift. This is a low cost operation relative to the standard arithmetic operations.]

From the iteration $x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$ we observe that since $a > 0$, x_{n+1} has the same sign as x_n for all n. Hence, if the Newton iteration converges, it converges to that root with the same sign as x_0.

Now consider the error in the iterates.

$$e_n \equiv x_n - \sqrt{a}$$

Algebraically, we have

$$\begin{aligned} e_{n+1} &= x_{n+1} - \sqrt{a} &= \frac{\left(x_n + \dfrac{a}{x_n}\right)}{2} - \sqrt{a} \\ &= \frac{(x_n - \sqrt{a})^2}{2x_n} &= \frac{e_n^2}{2x_n} \end{aligned}$$

From this formula, we see that when $x_0 > 0$ we have $e_1 = x_1 - \sqrt{a} > 0$ so that $x_1 > \sqrt{a} > 0$. It follows similarly that $e_n \geq 0$ for all $n \geq 1$. Hence, if x_n converges to $\sqrt{a}$ then it converges from above. To prove that the iteration converges, note that

$$0 \leq \frac{e_n}{x_n} = \frac{x_n - \sqrt{a}}{x_n - 0} = \frac{\text{distance from } x_n \text{ to } \sqrt{a}}{\text{distance from } x_n \text{ to } 0} < 1$$

for $n \geq 1$, so

$$e_{n+1} = \frac{e_n^2}{2x_n} = \left(\frac{e_n}{x_n}\right)\frac{e_n}{2} < \frac{e_n}{2}$$

Hence, for all $n \geq 1$, the error e_n is reduced by a factor of at least $\frac{1}{2}$ at each iteration. Therefore the iterates x_n converge to $\sqrt{a}$. Of course, "quadratic convergence rules!" as this is the Newton iteration, so when the error is sufficiently small the number of correct digits in x_n approximately doubles with each iteration. However, if x_0 is far from the root we may not have quadratic convergence initially, and, at worst, the error may only be approximately halved at each iteration.

Problem 3.2.1. *Consider the function $f(x) = x^3 - a$ which has only one real root, namely $x = \sqrt[3]{a}$. Show that a simplified Newton iteration for this function may be derived as follows*

$$\begin{aligned} x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} &= x_n - \frac{1}{3}\left(\frac{x_n^3 - a}{x_n^2}\right) \\ &= \frac{1}{3}\left(2x_n + \frac{a}{x_n^2}\right) \end{aligned}$$

Problem 3.2.2. *In the cube root iteration above show that the second (algebraically simplified) expression for the cube root iteration is less expensive to compute than the first. Assume that the value of $\frac{1}{3}$ is precalculated and in the iteration a multiplication by this value is used where needed.*

Problem 3.2.3. *Let $a > 0$. Write down the roots of the function*

$$f(x) = a - \frac{1}{x^2}$$

Write down the Newton iteration for computing the positive root of $f(x) = 0$. Show that the iteration formula can be simplified so that it does not involve divisions. Compute the arithmetic costs per iteration with and without the simplification; recall that multiplications or divisions by two (binary shifts) are less expensive than arithmetic. How would you compute $\sqrt{a}$ from the result of the iteration, without using divisions or using additional iterations?

Problem 3.2.4. *Let $a > 0$. Write down the real root of the function*

$$f(x) = a - \frac{1}{x^3}$$

Write down the Newton iteration for computing the real root of $f(x) = 0$. Show that the iteration formula can be simplified so that it does not involve a division at each iteration. Compute the arithmetic costs per iteration with and without the simplification; identify any parts of the computation that can be computed a priori, *that is they can be computed just once independently of the number of iterations. How would you compute $\sqrt[3]{a}$ from the result of the iteration, without using divisions or using additional iterations?*

3.3 Matlab Notes

By now our familiarity with MATLAB should lead us to suspect that there are built-in functions that can be used to compute roots, and it is not necessary to write our own implementations of the methods discussed in this chapter. The MATLAB functions that are relevant to the topics discussed in this chapter include:

`roots`	used to compute all roots of $p(x) = 0$ were $p(x)$ is a polynomial
`fzero`	used to find a root of $f(x) = 0$ for a general function of one variable

It is, however, instructive to implement some of the methods; at the very least, such exercises can help to improve our programming skills. We therefore begin this section by developing implementations of fixed–point, Newton and bisection methods. We then conclude this section with a thorough discussion of the built-in MATLAB functions `roots` and `fzero`. We emphasize that it is typically best to use one of these built-in functions, which have been thoroughly tested, rather than one of our own implementations.

3.3.1 Fixed–Point Iteration

Recall that the basic fixed–point iteration is given by

$$x_{n+1} = g(x_n), \quad n = 0, 1, 2, ...$$

The iterates, x_{n+1}, can be computed using a `for` loop. To begin the iteration, though, we need an initial guess, x_0, and we need a function, $g(x)$. If $g(x)$ is simple, then it is usually best to define it as an anonymous function, and if $g(x)$ is complicated, then it is usually best to define it as a function m-file.

Example 3.3.1. Suppose we want to compute 12 iterations of $g(x) = e^{-x}$, using the initial guess $x_0 = 0.5$ (see Table **??**). Since $g(x)$ is a very simple function, we define it as an anonymous function, and use a `for` loop to generate the iterates:

```
g = @(x) exp(-x);
x = 0.5;
for n = 1:12
  x = g(x);
end
```

If we want to test the fixed–point iteration for several different $g(x)$, then it would be helpful to have a MATLAB program (that is, a function m-file) that works for arbitrary $g(x)$. In order to write this function, we need to consider a few issues:

- In the previous example, we chose, somewhat arbitrarily, to compute 12 iterations. For other problems this may be too many or two few iterations. To increase flexibility, we should:
  - Stop the iteration if the computed solution is sufficiently accurate. This can be done, for example, by stopping the iteration if the relative change between successive iterates is less than some specified tolerance. That is, if

    $$|x_{n+1} - x_n| \leq \text{tol} * |x_n|.$$

    This can be implemented by inserting the following conditional statement inside the `for` loop:

    ```
    if ( abs(x(n+1)-x(n)) <= tol*max(abs(x(n)), abs(x(n+1))))
      break
    end
    ```

    A value of `tol` must be either defined in the code, or specified by the user. The command `break` terminates execution of the `for` loop.
  - Allow the user to specify a maximum number of iterations, and execute the loop until the above convergence criterion is satisfied, or until the maximum number of iterations is reached.
- It might be useful to return all of the computed iterations. This can be done easily by creating a vector containing $x_0, x_1, \cdots$.

Combining these items with the basic code given in Example 3.3.1 we obtain the following function:

```
function [x, flag] = FixedPoint(g, x0, tol, nmax)
%
%     [x, flag] = FixedPoint(g, x0, tol, nmax);
%
% Find a fixed--point of g(x) using the iteration:  x(n+1) = g(x(n))
%
% Input:  g - an anonymous function or function handle,
%        x0 - an initial guess of the fixed--point,
%       tol - a (relative) stopping tolerance, and
%      nmax - specifies maximum number of iterations.
%
% Output: x - a vector containing the iterates, x0, x1, ...
%         flag - set to 0 for normal exit and to 1
%                for exceeding maximum number of iterations
%
x(1) = x0; flag = 1;
for n = 1:nmax
  x(n+1) = g(x(n));
  if ( abs(x(n+1)-x(n)) <= tol*max(abs(x(n)), abs(x(n+1))))
    flag = 0;
    break
  end
end
x = x(:);
```

The final statement in the function, `x = x(:)`, is used to ensure `x` is returned as a column vector. This, of course, is not necessary, and is done only for cosmetic purposes.

Example 3.3.2. To illustrate how to use the function `FixedPoint`, consider $g(x) = e^{-x}$ with the initial guess $x_0 = 0.5$.

(a) If we want to compute exactly 12 iterations, we can set `nmax = 12` and `tol = 0`. That is, we use the statements:

```
g = @(x) exp(-x);
[x, flag] = FixedPoint(g, 0.5, 0, 12);
```

The same result could be computed using just one line of code:

```
[x, flag] = FixedPoint(@(x) exp(-x), 0.5, 0, 12);
```

Note that on exit, `flag = 1`, indicating that the maximum number of iterations has been exceeded, and that the iteration did not converge within the specified tolerance. Of course this is not surprising since we set `tol = 0`!

(b) If we want to see how many iterations it takes to reach a certain stopping tolerance, say `tol =` 10^{-8}, then we should choose a relatively large value for `nmax`. For example, we could compute

```
[x, flag] = FixedPoint(@(x) exp(-x), 0.5, 1e-8, 1000);
```

On exit, we see that `flag = 0`, and so the iteration has converged within the specified tolerance. The number of iterations is then `length(x) - 1` (recall the first entry in the vector is the initial guess). For this example, we find that it takes 31 iterations.

Problem 3.3.1. *Implement* `FixedPoint` *and use it to:*

(a) Produce a table corresponding to Table **??** *starting from the value* $x_0 = 0.6$

(b) Produce a table corresponding to Table **??** *starting from the value* $x_0 = 0.0$

(c) Produce a table corresponding to Table **??** *starting from the value* $x_0 = 2.0$

Recall that it is possible to produced formatted tables in MATLAB *using the* `sprintf` *and* `disp` *commands; see Section 1.3.4.*

Problem 3.3.2. *Implement* `FixedPoint` *and use it to compute 10 iterations using each* $g(x)$ *given in Problem* **??***, with* $x_0 = 1.5$*.*

Problem 3.3.3. *Rewrite the function* `FixedPoint` *using* `while` *instead of the combination of* `for` *and* `if`*. Note that it is necessary to implement a counter to determine when the number of iterations has reached* `nmax`*.*

3.3.2 Newton and Secant Methods

The Newton and secant methods are used to find roots of $f(x) = 0$. Implementation of these methods is very similar to the fixed–point iteration. For example, Newton's method can be implemented as follows:

```
function x = Newton(f, df, x0, tol, nmax)
%
%     x = Newton(f, df, x0, tol, nmax);
%
% Use Newton's method to find a root of f(x) = 0.
%
% Input:  f - an anonymous function or function handle for f(x)
%        df - an anonymous function or function handle for f'(x)
%        x0 - an initial guess of the root,
%       tol - a (relative) stopping tolerance
%      nmax - specifies maximum number of iterations.
%
% Output: x - a vector containing the iterates, x0, x1, ...
%
x(1) = x0;
for n = 1:nmax
  x(n+1) = x(n) - f(x(n)) / df(x(n));
  if ( abs(x(n+1)-x(n)) <= tol*abs(x(n)) )
    break
  end
end
x = x(:);
```

Note that in Newton's method we must input two functions, $f(x)$ and $f'(x)$. It might be prudent to include a check, and to print an error message, if `df(x(n))` is zero, or smaller in magnitude than a specified value.

Example 3.3.3. Suppose we want to find a root of $f(x) = 0$ where $f(x) = x - e^{-x}$, with initial guess $x_0 = 0.5$. Then, using `Newton`, we can compute an approximation of this root using the following MATLAB statements:

```
f = @(x) x - exp(-x);
df = @(x) 1 + exp(-x);
x = Newton(f, df, 0.5, 1e-8, 10);
```

or, in one line of code, using

```
x = Newton(@(x) x - exp(-x), @(x) 1 + exp(-x), 0.5, 1e-8, 10);
```

For this problem, only 4 iterations of Newton's method are needed to converge to the specified tolerance.

Problem 3.3.4. *Using the* `Fixed Point` *code above as a template, modify* `Newton` *to check more carefully for convergence and to use a flag to indicate whether convergence has occurred before the maximum number of iterations has been exceeded. Also include in your code a check for small derivative values.*

Problem 3.3.5. *Using the* `Newton` *code above, write a similar implementation* `Secant` *for the secant iteration. Test both codes at a simple root of a quadratic polynomial $f(x)$ that is concave up, choosing the initial iterates x_0 (and, in the case of secant, x_1) close to this root. Construct tables containing the values of the iterates, x_n and corresponding function values, $f(x_n)$. Is there any pattern to the signs of the consecutive $f(x)$ values?*

Problem 3.3.6. *Using the* `Newton` *code above, write a similar implementation* `Secant` *for the secant iteration. Use both codes to compute the root of $f(x) = x - e^{-x}$. For Newton, use the initial guess $x_0 = 0.5$, and for secant use $x_0 = 0$ and $x_1 = 1$. Check the ratios of the errors in successive iterates as in Table* **??** *to determine if they are consistent with the convergence rates discussed in this chapter. Repeat the secant method with initial iterates $x_0 = 1$ and $x_1 = 0$.*

Problem 3.3.7. *Using the* `Newton` *code above, write a similar implementation* `Secant` *for the secant iteration. Use both methods to compute the root of $f(x) = \ln(x-3) + \sin(x) + 1$, $x > 3$. For Newton, use the initial guess $x_0 = 4$, and for secant use $x_0 = 3.25$ and $x_1 = 4$. Check the ratios of the errors in successive iterates as in Table* **??** *to determine if they are consistent with the convergence rates discussed in this chapter.*

3.3.3 Bisection Method

Recall that bisection begins with an initial bracket, $[a, b]$ containing the root, and proceeds as follows:

- First compute the midpoint, $m = (a + b)/2$.
- Determine if a root is in $[a, m]$ or $[m, b]$. If the root is in $[a, m]$, rename m as b, and continue. Similarly, if the root is in $[m, b]$, rename m as a and continue.
- Stop if the length of the interval becomes sufficiently small.

Note that we can determine if the root is in $[a, m]$ by checking to see if $f(a) \cdot f(m) \leq 0$. With these observations, an implementation of the bisection method could have the form:

```
function x = Bisection0(f, a, b, tol, nmax)
%
%     x = Bisection0(f, a, b, tol, nmax);
%
% Use bisection method to find a root of f(x) = 0.
%
% Input:  f - an anonymous function or function handle for f(x)
%     [a,b] - an initial bracket containing the root
%       tol - a (relative) stopping tolerance, and
%      nmax - specifies maximum number of iterations.
%
% Output: x - a vector containing the iterates, x0, x1, ...
%
x(1) = a;   x(2) = b;
for n = 2:nmax
  m = (a+b)/2;   x(n+1) = m;
  if f(a)*f(m) < 0
    b = m;
  else
    a = m;
  end
  if ( abs(b-a) <= tol*abs(a) )
    break
  end
end
x = x(:);
```

Note that this implementation requires that we evaluate $f(x)$ twice for every iteration (to compute $f(a)$ and $f(b)$). Since this is typically the most expensive part of the method, it is a good idea to reduce the number of function evaluations as much as possible. The bisection method can be implemented so that only one function evaluation is needed at each iteration. In the algorithm below we have taken care to keep track of whether the maximum number of iterations has been exceeded and to check for zeros and avoid underflows in our treatment of the function values generated during the iteration.

Problem 3.3.8. *Use the* `Bisection` *code to compute a root of* $f(x) = x - e^{-x}$*. Use the initial bracket* $[0, 1]$*. Produce a table of results similar to Table* **??***.*

Problem 3.3.9. *Using the* `Bisection`, `Secant` *and* `Newton` *codes, experimentally compare the convergence behavior of each method for the function* $f(x) = x - e^{-x}$*.*

```
function [x, flag] = Bisection(f, a, b, tol, nmax)
%
%     [x, flag]  = Bisection(f, a, b, tol, nmax);
%
% Use bisection method to find a root of f(x) = 0.
%
% Input:  f - an anonymous function or function handle for f(x)
%     [a,b] - an initial bracket containing the root
%       tol - a (relative) stopping tolerance, and
%      nmax - specifies maximum number of iterations.
%
% Output: x - a vector containing the iterates, x0, x1, ...,
%             with last iterate the final approximation to the root
%         flag - set to 0 for normal exit and to 1
%                for exceeding maximum number of iterations
%
x(1) = a;  x(2) = b;  fa = f(a);  fb = f(b);  flag = 1;
if (sign(fa) ~= 0 & sign(fb) ~= 0)
  for n = 2:nmax
    c = (a+b)/2;  x(n+1) = c;  fc = f(c);
    if sign(fc) == 0
      flag = 0;
      break
    end
    if sign(fa) ~= sign(fc)
      b = c;  fb = fc;
    else
      a = c;  fa = fc;
    end
    if (abs(b-a) <= tol*max(abs(a), abs(b)))
      flag = 0;
      break
    end
  end
else
  if sign(fa) == 0
    x(2) = a; x(1) = b;
  end
  flag = 0;
end
x = x(:);
```

3.3.4 The `roots` and `fzero` Functions

As previously mentioned, MATLAB provides two built-in root finding methods, `roots` (to find roots of polynomials) and `fzero` (to find a root of a more general function). In this section we discuss in more detail how to use these built-in MATLAB functions.

Computing a solution of $f(x) = 0$.

The MATLAB function `fzero` is used to find zeros of a general function of one variable. The implementation uses a combination of bisection, secant and inverse quadratic interpolation. The basic calling syntax for `fzero` is:

```
x = fzero(fun, x0)
```

where `fun` can be an inline or anonymous function, or a handle to a function M–file. The initial guess, `x0` can be a scalar (that is, a single initial guess), or it can be a bracket (that is, a vector containing two values) on the root. Note that if `x0` is a single initial guess, then `fzero` first attempts to find a bracket by sampling on successively wider intervals containing `x0` until a bracket is determined. If a bracket for the root is known, then it is usually best to supply it, rather than a single initial guess.

Example 3.3.4. Consider the function $f(x) = x - e^{-x}$. Because $f(0) < 0$ and $f(1) > 0$, a bracket for a root of $f(x) = 0$ is $[0, 1]$. This root can be found as follows:

```
x = fzero(@(x) x-exp(-x), [0, 1])
```

We could have used a single initial guess, such as:

```
x = fzero(@(x) x-exp(-x), 0.5)
```

but as mentioned above, it is usually best to provide a bracket if one is known.

Problem 3.3.10. *Use `fzero` to find a root of $f(x) = 0$, with $f(x) = \sin x$. For initial guess, use $x_0 = 1$, $x_0 = 5$, and the bracket $[1, 5]$. Explain why a different root is computed in each case.*

Problem 3.3.11. *Consider the function $f(x) = \frac{1}{x} - 1$, and note that $f(1) = 0$. Thus, we might expect that $x_0 = 0.5$ is a reasonable initial guess for `fzero` to compute a solution of $f(x) = 0$. Try this, and see what `fzero` computes. Can you explain what happens here? Now use the initial guess to be the bracket $[0.5, 1.5]$. What does `fzero` compute in this case?*

If a root, x_*, has multiplicity greater than one, then $f'(x_*) = 0$, and hence the root is not simple. `fzero` can find roots which have odd multiplicity, but it cannot find roots of even multiplicity, as illustrated in the following example.

Example 3.3.5. Consider the function $f(x) = x^2 e^x$, which has a root $x_* = 0$ of multiplicity 2.

(a) If we use the initial guess $x_0 = 1$, then

```
f = @(x) x.*x.*exp(x);
xstar = fzero(f, 1)
```

computes $x \approx -925.8190$. At first this result may seem completely ridiculous, but if we compute

```
f(xstar)
```

the result is 0. Notice that $\lim\limits_{x \to -\infty} x^2 e^x = 0$, thus it appears that `fzero` "finds" $x \approx -\infty$.

(b) If we try the initial guess $x_0 = -1$, then

```
f = @(x) x.*x.*exp(x);
xstar = fzero(f, -1)
```

then `fzero` fails because it cannot find an interval containing a sign change.

(c) We know the root is $x_* = 0$, so we might try the bracket $[-1, 1]$. However, if we compute

```
f = @(x) x.*x.*exp(x);
xstar = fzero(f, [-1, 1])
```

an error occurs because the initial bracket does not satisfy the sign change condition, $f(-1)f(1) < 0$.

Problem 3.3.12. *The functions $f(x) = x^2 - 4x + 4$ and $f(x) = x^3 - 6x^2 + 12x - 8$ satisfy $f(2) = 0$. Attempt to find this root using* `fzero`*. Experiment with a variety of initial guesses. Why does* `fzero` *have trouble finding the root for one of the functions, but not for the other?*

It is possible to reset certain default parameters used by `fzero` (such as stopping tolerance and maximum number of iterations), and it is possible to obtain more information on exactly what is done during the computation of the root (such as number of iterations and number of function evaluations) by using the calling syntax

```
[x, fval, exitflag, output] = fzero(fun, x0, options)
```

where

- `x` is the computed approximation of the root.
- `fval` is the function value of the computed root, $f(x_*)$. Note that if a root is found, then this value should be close to zero.
- `exitflag` provides information about what condition caused `fzero` to terminate; see `doc fzero` for more information.
- `output` is a structure array that contains information such as which algorithms were used (e.g., bisection, interpolation, secant), number of function evaluations and number of iterations.
- `options` is an input parameter that can be used, for example, to reset the stopping tolerance, and to request that results of each iteration be displayed in the command window. The options are set using the built-in MATLAB function `optimset`.

The following examples illustrate how to use `options` and `output`.

Example 3.3.6. Consider $f(x) = x - e^{-x}$. We know there is a root in the interval $[0, 1]$. Suppose we want to investigate how the cost of computing the root is affected by the choice of the initial guess. Since the cost is reflected in the number of function evaluations, we use `output` for this purpose.

(a) Using the initial guess $x_0 = 0$, we can compute:

```
[x, fval, exitflag, output] = fzero(@(x) x-exp(-x), 0);
output
```

we see that `output` displays the information:

```
intervaliterations: 10
        iterations: 6
         funcCount: 27
         algorithm: 'bisection, interpolation'
           message: 'Zero found in the interval [-0.64, 0.64]'
```

The important quantity here is `funcCount`, which indicates that a total of 27 function evaluations was needed to compute the root.

(b) On the other hand, if we use $x_0 = 0.5$, and compute:

```
[x, fval, exitflag, output] = fzero(@(x) x-exp(-x), 0.5);
output
```

we see that 18 function evaluations are needed.

(c) Finally, if we use the bracket $[0, 1]$, and compute:

```
[x, fval, exitflag, output] = fzero(@(x) x-exp(-x), [0, 1]);
output
```

we see that only 8 function evaluations are needed.

Notice from this example that when we provide a bracket, the term `interval iterations`, which is returned by the structure `output`, is 0. This is because `fzero` does not need to do an initial search for a bracket, and thus the total number of function evaluations is substantially reduced.

Example 3.3.7. Again, consider $f(x) = x - e^{-x}$, which has a root in $[0, 1]$. Suppose we want to investigate how the cost of computing the root is affected by the choice of stopping tolerance on x. To investigate this, we must first use the MATLAB function `optimset` to define the structure `options`.

(a) Suppose we want to set the tolerance to 10^{-4}. Then we can use the command:

```
options = optimset('TolX', 1e-4);
```

If we then execute the commands:

```
[x, fval, exitflag, output] = fzero(@(x) x-exp(-x), [0, 1], options);
output
```

we see that only 5 function evaluations are needed.

(b) To increase the tolerance to 10^{-8}, we can use the command:

```
options = optimset('TolX', 1e-8);
```

Executing the commands:

```
[x, fval, exitflag, output] = fzero(@(x) x-exp(-x), [0, 1], options);
output
```

shows that 7 function evaluations are needed to compute the approximation of the root.

Example 3.3.8. Suppose we want to display results of each iteration when we compute an approximation of the root of $f(x) = x - e^{-x}$. For this, we use `optimset` as follows:

```
options = optimiset('Display', 'iter');
```

If we then use `fzero` as:

```
x = fzero(@(x) x-exp(-x), [0, 1], options);
```

then the following information is displayed in the MATLAB command window:

```
 Func-count    x          f(x)             Procedure
    2               1       0.632121        initial
    3          0.6127      0.0708139        interpolation
    4         0.56707   -0.000115417        interpolation
    5        0.567144    9.44811e-07        interpolation
    6        0.567143    1.25912e-11        interpolation
    7        0.567143   -1.11022e-16        interpolation
    8        0.567143   -1.11022e-16        interpolation

Zero found in the interval [0, 1]
```

Notice that a total of 8 function evaluations were needed, which matches (as it should) the number found in part (c) of Example 3.3.6.

Problem 3.3.13. *Note that we can use* `optimset` *to reset* `TolX` *and to request display of the iterations in one command, such as:*

```
options = optimset('TolX', 1e-7, 'Display', 'iter');
```

Use this set of options with $f(x) = x - e^{-x}$, and various initial guesses (e.g., 0, 1 and [0,1]) in `fzero` *and comment on the computed results.*

Roots of polynomials

The MATLAB function `fzero` cannot, in general, be effectively used to compute roots of polynomials. For example, it cannot compute the roots of $f(x) = x^2$ because $x = 0$ is a root of even multiplicity. However, there is a special purpose method, called `roots`, that can be used to compute all roots of a polynomial.

Recall from our discussion of polynomial interpolation (section **??**) MATLAB assumes polynomials are written in the canonical form

$$p(x) = a_1 x^n + a_2 x_{n-1} + \cdots + a_n x + a_{n+1},$$

and that they are represented with a vector (either row or column) containing the coefficients:

$$a = \left[\begin{array}{ccccc} a_1 & a_2 & \cdots & a_n & a_{n+1} \end{array} \right].$$

If such a vector is defined, then the roots or $p(x)$ can be computed using the built-in function `roots`. Because it is beyond the scope of this book, we have not discussed the basic algorithm used by `roots` (which computes the eigenvalues of a *companion matrix* defined by the coefficients of $p(x)$), but it is a very simple function to use. In particular, execution of the statement

```
r = roots(a);
```

produces a (column) vector `r` containing the roots of the polynomial defined by the coefficients in the vector `a`.

Example 3.3.9. Consider $x^4 - 5x^2 + 4 = 0$. The following MATLAB commands:

```
a = [1 0 -5 0 4];
r = roots(a)
```

compute a vector `r` containing the roots 2, 1, -2 and -1.

The MATLAB function `poly` can be used to create the coefficients of a polynomial with given roots. That is, if `r` is a vector containing the roots of a polynomial, then

```
a = poly(r);
```

constructs a vector containing the coefficients of a polynomial whose roots are given by the entries in the vector `r`. Basically, `poly` multiplies out the factored form

$$p(x) = (x - r_1)(x - r_2)\cdots(x - r_n)$$

to obtain the canonical power series form

$$p(x) = x^n + a_2 x^{n-1} + \cdots + a_n x + a_{n+1}.$$

Example 3.3.10. The roots of $(2x + 1)(x - 3)(x + 7) = 0$ are obviously $-\frac{1}{2}, 3$ and -7. Executing the MATLAB statements

```
r = [-1/2, 3, -7];
a = poly(r)
```

constructs the vector `a = [1, 4.5, -19, -10.5]`.

Problem 3.3.14. *Consider the polynomial given in Example* **??**. *Use the* MATLAB *function* `poly` *to construct the coefficients of the power series form of* $p_{12}(x)$. *Then compute the roots using the* MATLAB *function* `roots`. *Use the* `format` *command so that computed results display 15 digits.*

Problem 3.3.15. *Consider the polynomial* $p_{20}(x)$ *given in Example* **??**. *Compute the results shown in that example using the* MATLAB *functions* `poly` *and* `roots`. *Use the* `format` *command so that computed results display 15 digits.*

Problem 3.3.16. *Consider the polynomial* $p_{22}(x)$ *given in Example* **??**. *Compute the results shown in that example using the* MATLAB *functions* `poly` *and* `roots`. *Use the* `format` *command so that computed results display 15 digits.*