

Contents

Contents	1
1 MATLAB Basics	3
Chapter 1: MATLAB Basic	3
1.1 WHAT IS MATLAB?	3
1.2 STARTING AND ENDING A MATLAB SESSION	3
1.3 A FIRST MATLAB TUTORIAL	4
1.4 VECTORS AND AN INTRODUCTION TO MATLAB GRAPHICS	6
1.5 A TUTORIAL INTRODUCTION TO RECURSION ON MATLAB	12
2 Basic Concepts of Numerical Analysis with Taylor's Theorem	19
Chapter 2: Basic Concepts of Numerical Analysis with Taylor's Theorem	19
2.1 WHAT IS NUMERICAL ANALYSIS?	19
2.2 TAYLOR POLYNOMIALS	20
2.3 TAYLOR'S THEOREM	26
3 Introduction to M-Files	33
Chapter 3: Introduction to M-Files	33
3.1 WHAT ARE M-FILES?	33
3.2 CREATING AN M-FILE FOR A MATHEMATICAL FUNCTION	35
4 Programming in MATLAB	41
Chapter 4: Programming in MATLAB	41
4.1 SOME BASIC LOGIC	41
4.2 LOGICAL CONTROL FLOW IN MATLAB	43
4.3 WRITING GOOD PROGRAMS	51
5 Floating Point Arithmetic and Error Analysis	59
Chapter 5: Floating Point Arithmetic and Error Analysis	59
5.1 FLOATING POINT NUMBERS	59
5.2 FLOATING POINT ARITHMETIC: THE BASICS	59
5.3 FLOATING POINT ARITHMETIC: FURTHER EXAMPLES AND DETAILS	65

Chapter 1

MATLAB Basics

1.1 WHAT IS MATLAB?

As a student who has already taken courses at least up through calculus, you most likely have seen the power of graphing calculators and perhaps those with symbolic capabilities. MATLAB adds a whole new exciting set of capabilities as a powerful computing tool. Here are a few of the advantages you will enjoy when using MATLAB, as compared to a graphing calculator:

1. It is easy to learn and use. You will be entering commands on your big, familiar computer keyboard rather than on a tiny little keypad where sometimes each key has four different symbols attached.
2. The graphics that MATLAB produces are of very high resolution. They can be easily copied to other documents (with simple clicks of your mouse) and printed out in black/white or color format. The same is true of any numerical and algebraic MATLAB inputs and outputs.
3. MATLAB is an abbreviation for MATrix LABoratory. It is ideally suited for calculations and manipulations involving matrices. This is particularly useful for computer users since the spreadsheet (the basic element for recording data on a computer) is just a matrix.
4. MATLAB has many built-in programs and you can interactively use them to create new programs to perform your desired tasks. It enables you to take advantage of the full computing power of your computer, which has much more memory and speed than a graphing calculator.
5. MATLAB's language is based on the C-family of computer languages. People experienced with such languages will find the transition to MATLAB natural and people who learn MATLAB without much computer background will, as a fringe benefit, be learning skills that will be useful in the future if they need to learn more computer languages.
6. MATLAB is heavily used by mathematicians, scientists, and engineers and there is a tremendous amount of interesting programs and information available on the Internet (much of it is free). It is a powerful computing environment that continues to evolve.

We wish here and now to present a disclaimer. MATLAB is a spectacularly vast computing environment and our plan is not to discuss all of its capabilities, but rather to give a decent survey of enough of them so as to provide the reader with a powerful new arsenal of uses of MATLAB for solving a variety of problems in mathematics and other sciences. Several good books have been written just on using MATLAB; see, for example, references [HiHi-00], [HuLiRo-01], [PSMI98], and [HaLi-00].¹

1.2 STARTING AND ENDING A MATLAB SESSION

We assume that MATLAB has been installed on the system that you are using.² Instructions for starting MATLAB are similar to those for starting any installed software on your system. For example, on most windows-based systems, you should be able to simply double click on MATLAB's icon. Once MATLAB is started, a command window should pop up with a **prompt**:» (or EDU » if you are using the Student Version). In what follows, if we tell you to enter something like » 2+2 (on the command window), you enter 2+2 only at the prompt— which is already there waiting for you to type something. Before we begin our first brief tutorial, we point out that there is a way to create a file containing all interactions with a particular MATLAB session. The command diary will do this. Here is how it works: Say you want to save the session we are about to start to your floppy disk, which you have inserted in the a:/-drive. After the prompt type:

¹Citations in square brackets refer to items in the References section in the back of this book

²MATLAB is available on numerous computing platforms including PC Windows, Linux, MAC, Solaris, Unix, HP-UX. The functionality and use is essentially platform independent although some external interface tasks may vary.

```
>> diary a:/tutor1.txt
```

NOTE: If you are running MATLAB in a computer laboratory or on someone else's machine, you should always save things to your portable storage device or personal account. This will be considerate to the limitations of hard drive space on the machines you are using and will give you better assurance that the files still will be available when you need them.

This causes MATLAB to create a text file called `tutor1.txt` in your `a:/` drive called `tutor1.txt`, which, until you end the current MATLAB session, will be a carbon copy of your entire session on the command window. You can later open it up to edit, print, copy, etc. It is perhaps a good idea to try this out once to see how it works and how you like it (and we will do this in the next section), but in practice, most MATLAB users will often just copy the important parts, of their MATLAB session and paste them appropriately in an open word processing window of their choice.

On most platforms, you can end a MATLAB session by clicking down your left mouse button after you have moved the cursor to the "File" menu (located on the upper-left corner of the MATLAB command window). This will cause a menu of commands to appear that you can choose from. With the mouse button still held down, slide the cursor down to the "Exit MATLAB" option and release it. This will end the session. Another way to accomplish the same would be to simply click (and release) the left mouse button after you have slid it on top of the "X" button at the upper-right corner of the command window. Yet another way is to simply enter the command:

```
>> quit
```

Any diary file you created in the session will now be accessible.

1.3 A FIRST MATLAB TUTORIAL

As with all tutorials we present, this is intended to be worked by the reader on a computer with MATLAB installed. Begin by starting a MATLAB session as described earlier. If you like, you may begin a diary as shown in the previous section on which to document this session. MATLAB will not respond to or execute any command until you press the "enter key," and you can edit a command (say, if you made a typo) and press enter no matter where the cursor is located in a given command line. Let us start with some basic calculations: First enter the command:

```
>> 5 + 3
-> ans = 8
```

The arrow (`->`) notation indicates that MATLAB has responded by giving us `ans = 8`. As a general rule we will print MATLAB input in a different font (Courier New) than the main font of the text (Times New Roman). It does not matter to MATLAB whether you leave spaces around the `+` sign.³ (This is usually just done to make the printout more legible.) Instead of adding, if we wanted to divide 5 by 3, we would enter (the operation `-5-` is represented by the keyboard symbol `/` in MATLAB)

```
>> 5/3
-> ans =1.6667
```

The output "1.6667" is a four-decimal approximation to the unending decimal approximation. The exact decimal answer here is 1.6666666666... (where the 6's go on forever). The four-decimal display mode is the default format in which MATLAB displays decimal answers. The previous example demonstrates that if the inputs and outputs are integers (no decimals), MATLAB will display them as such. MATLAB does its calculations using about 16 digits—we shall discuss this in greater detail in Chapters 2 and 5. There are several ways of changing how your outputs are displayed. For example, if we enter:

```
>> format long
>> 5/3
-> ans =1.666666666666667
```

we will see the previous answer displayed with 15 digits. All subsequent calculations will be displayed in this format until you change it again. To change back to the default format, enter `» format short`. Other popular formats are `» format bank` (displays two decimal places, useful for applications to finance) and `» format rat` (approximates all answers as fractions of small integers and displays them as such). It is not such a good idea to work in `format rat` unless you know for sure the numbers you are working with are fractions as opposed to irrational numbers, like $71 = 3.14159265\dots$, whose decimals go on forever without repetition and are impossible to express via fractions.

In MATLAB, a single equals sign (`=`) stands for "is assigned the value." For example, after switching back to the default format, let us store the following constants into MATLAB's workspace memory:

³The format of actual output that MATLAB gives can vary slightly depending on the platform and version being used. In general it will take up more lines and have more blank spaces than as we have printed it. We adopt this convention throughout the book in order to save space.

```
>>format short
>> a = 2.5
>> a = 2.5000
>> b = 64
-> b = 64
```

Notice that after each of these commands, MATLAB will produce an output of simply what you have inputted and assigned. You can always suppress the output on any given MATLAB command by tacking on a semicolon (;) at the end of the command (before you press enter). Also, you can put multiple MATLAB commands on a single line by separating them with commas, but these are not necessary after a semicolon. For example, we can introduce two new constants a and bb without having any output using the single line:

```
>>aa = 11; bb = 4;
```

Once variables have been assigned in a MATLAB session, computations involving them can be done using any of MATLAB's built-in functions. For example, to evaluate $aa + a\sqrt{b}$, we could enter

```
>> aa + a*sqrt(b)
>> ans=31
```

Note that a stands for the single variable that we introduced above rather than a1, so the output should be 31. MATLAB has many built-in functions, many of which are listed in the MATLAB Command Index at the end of this book.

MATLAB treats all numerical objects as matrices, which are simply rectangular arrays of numbers. Later we will see how easy and flexible MATLAB is in manipulating such arrays. Suppose we would like to store in MATLAB the following two matrices:

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

We do so using the following syntax:

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

    2    4
   -1    6

>> B = [2 5 -3 ; 1 0 -1 ; 8 4 0]
>>-> B=

    2    5   -3
    1    0   -1
    8    4    0
```

(note that the rows of a matrix are entered in order and separated by semicolons; also, adjacent entries within a row are given at least one space between). You can see from the outputs that MATLAB displays these matrices pretty much in their mathematical form (but without the brackets).

In MATLAB it is extremely simple to edit a previous command into a new one. Let's say in the matrix B above, we wish to change the bottom-left entry from eight to three. Since the creation of matrix B was the last command we entered, we simply need to press the up-arrow key (↑) once and magically the whole last command appears at the cursor (do this!). If you continue to press this up-arrow key, the preceding commands will continue to appear in order. Try this now! Next press the down arrow key (↓) several times to bring you back down again to the most recent command you entered (i.e., where we defined the matrix B). Now simply use the mouse and/or left- and right-arrow keys to move the cursor to the 8 and change it to 3, then press enter. You have now overwritten your original matrix for B with this modified version. Very nice indeed! But there is more. If on the command line you type a sequence of characters and then press the uparrow key, MATLAB will then retrieve only those input lines (in order of most recent occurrence) that begin with the sequence of characters typed. Thus for example, if you type a and then up-arrow twice, you would get the line of input where we set $aa = 11$.

A few more words about "variables" are in order. Variable names can use up to 19 characters, and must begin with a letter, but after this you can use digits and underscores as well. For example, two valid variable names are `diffusion22time` and `Shock_wave_index`; however, `Final$Amount` would not be an acceptable variable name because of the symbol `$`. Any time that you would like to check on the current status of your variables, just enter the command `who`:

```
>> who
->Your variables are:
A B a aa ans b bb
```

For more detailed information about all of the variables in the workspace (including the size of all of the matrices) use the command `whos`:

```
>> whos

-> name    size    bytes    class
   A       2x2     32 double  array
   B       3x3     72 double  array
   a       1x1      8 double  array
  aa       1x1      8 double  array
  ans      1x1      8 double  array
   b       1x1      8 double  array
  bb       1x1      8 double  array
```

You will notice that MATLAB retains both the number `a` and the matrix `A`. MATLAB is case-sensitive. You will also notice that there is the variable `ans` in the workspace. Whenever you perform an evaluation/calculation in MATLAB, an automatic assignment of the variable `ans` is made to the most recent result (as the output shows). To clear any variable, say `aa`, use the command

```
>> clear aa
```

Do this and check with `who` that `aa` is no longer in the workspace. If you just enter `clear`, all variables are erased from the workspace. More importantly, suppose that you have worked hard on a MATLAB session and would like to retain all of your workspace variables for a future session. To save (just) the workspace variables, say to your floppy `a:` drive, make sure you have your disk inserted and enter:

```
>> save a:/tutvars
```

This will create a file on your floppy called `tutvars.mat` (you could have called it by any other name) with all of your variables. To see how well this system works, go ahead and quit this MATLAB session and start a new one. If you type `who` you will get no output since we have not yet created any variables in this new session. Now (making sure that the floppy with `tutvars` is still inserted) enter the command:

```
>> load a:/tutvars
```

If you enter `who` once again you will notice that all of those old variables are now in our new workspace. You have just made it through your first MATLAB tutorial. End the session now and examine the diary file if you have created one.

If you want more detailed information about any particular MATLAB command, say `who`, you would simply enter:

```
>>help who
```

and MATLAB would respond with some usage information and related commands.

1.4 VECTORS AND AN INTRODUCTION TO MATLAB GRAPHICS

On any line of input of a MATLAB session, if you enter the percent symbol (anything you type after this is ignored by MATLAB's processor and is treated as a comment).⁴ This is useful, in particular, when you are writing a complicated program and would like to enhance it with some comments to make it more understandable (both to yourself at a later reading and to others who will read it). Let us now begin a new MATLAB session.

A vector is a special kind of matrix with only one row or one column. Here are examples of vectors of each type:

$$x = [1 \ 2 \ 3], y = \begin{bmatrix} 2 \\ -3 \\ 5 \end{bmatrix}.$$

```
>> % We create the above two vectors and one more as variables in our MATLAB session.
>> x = [1 2 3], y = [2 ; -3 ; 5], z = [4 -5 6]
>> x = 1 2 3,
y =
```

⁴MATLAB's windows usually conform to certain color standards to make codes easier to look through. For example, when a comment is initiated with `%`, the symbol and everything appearing after it will be shown in green. Also, warning/error messages (as we will soon experience on the next page) appear in red. The default color for input and output is black

```

2
-3      z = 4 -5 6
5

```

```

>> % Next we perform some simple array operations.
>> a = x + z
->a= 5 -3 9
>> b = x + y %MATLAB needs arrays to be the same size to add/subtract
->??? Error using ==> +
Matrix dimensions must agree.
>>c=x.*z %term by term multiplication, notice the dot before the *
->c = 4 -10 18

```

The **transpose** of any matrix A , denoted as A^T or A' , consists of the matrix whose rows are (in order) the columns of A and vice versa. For example the transpose of the 2x3 matrix

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 1 & -2 & 5 \end{bmatrix}$$

is the 3x2 matrix

$$A = \begin{bmatrix} 2 & 1 \\ 4 & -2 \\ 9 & 5 \end{bmatrix}$$

```

>> y %MATLAB uses the prime ' for the transpose operation
-> ans = 2 -3 5
>> b=x+y %cf. with the result for x + y
-*b = 3 -1 8
>> % We next give some other useful ways to create vectors.
>> % To create a (row) vector having 5 elements linearly spaced
>> % between 0 and 10 you could enter
>> linspace(0,10,5) %Do this!
-> ans = 0 2.5000 5.0000 7.5000 10.0000

```

We indicate the general syntax of `linspace` as well as another useful way to create vectors (especially big ones!):

<code>v=linspace (F, L,N)</code> →	If F and L are real numbers and N is a positive integer, this command creates a row vector v with: first entry = F , last entry = L , and having N equally spaced entries
<code>v = F:G:L</code> →	If F and L are real numbers and G is a nonzero real number, this command creates a vector v with: first entry = F , last (possible) entry = L , and gap between entries = G . G is optional with default value 1.

To see an example, enter

```

>> x = 1:.25:2.5 %will overwrite previously stored value of x
->x = 1.0000 1.2500 1.5000 1.7500 2.0000 2.2500 2.5000
>> y = -2:.5:3
-> y = -2.0000 -1.5000 -1.0000 -0.5000 0 0.5000 1.0000 1.5000 2.0000 2.5000 3.0000

```

EXERCISE FOR THE READER 1.1: Use the `linspace` command above to recreate the vector y that we just built.

The basic way to plot a graph in MATLAB is to give it the xy -coordinates (as a vector a) and the corresponding y -coordinates (as a vector b of the same length) and then use the `plot` command.

<code>plot(a,b)</code> →	If a and b are vectors of the same length, this command will create a plot of the line segments connecting (in order) the points in the xy -plane having x -coordinates listed in the vector a and corresponding y -coordinates in the vector b .
--------------------------	---

To demonstrate how this works, we begin with some simple vector plots and work our way up to some more involved function plots. The following commands will produce the plot shown in Figure 1.1.

```

>>x = [1 2 3 4]; y = [1 -3 3 0] ;
>>plot(x,y)

```

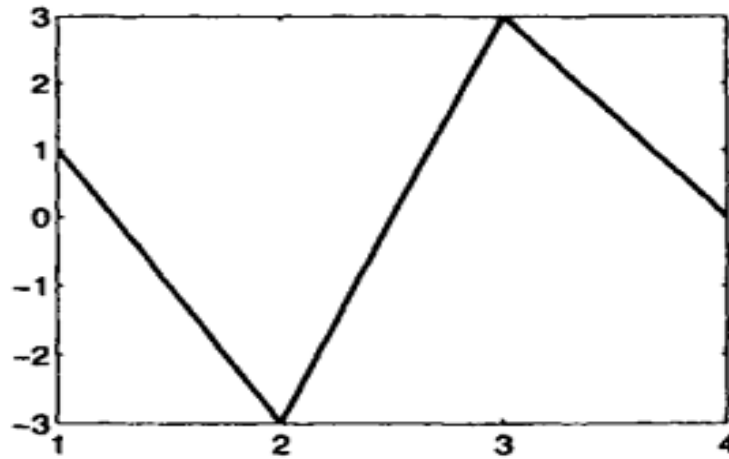


Figure 1.1: A simple plot resulting from the command `plot(x, y)` using the vector $x = [1 \ 2 \ 3 \ 4]$ for x-coordinates and the vector $y = [1 \ -3 \ 3 \ 0]$ for corresponding y-coordinates.⁵

Next, we use the same vector approach to graph the function $y = \cos(x^2)$ on $[0, 5]$. The finer the grid determined by the vectors you use, the greater the resolution. To see this first hand, enter:

```
>> x = linspace(0,5,5);           % I will be supressing a lot of output, you
>>                                % can drop the ';' to see it
>> y = cos(x.^2);
```

Note the dot (.) before the power operator (^). The dot before an operator changes the default matrix operation to a **component-wise operation**. Thus $x.^2$ will create a new vector of the same size as x where each of the entries is just the square of the corresponding entry of x . This is what we want. The command x^2 would ask MATLAB to multiply the matrix (or row vector) x by itself, which (as we will explain later) is not possible and so would produce an error message.

```
>> plot(x,y) % produces our first very rough plot of the function
>> % with only 5 plotting points
```

See Figure 1.2(a) for the resulting plot. Next we do the same plot but using 25 points and then 300 points. The editing techniques of Section 1.2 will be of use as you enter the following commands.

```
>>x = linspace(0,5,25);
>>y = cos(x.^2);
>>plot(x,y) % a better plot with 25 points.
>>x = linspace(0,5,300);
>>y = cos(x.^2);
>> plot(x,y) % the plot is starting to look good with 300 points.
```

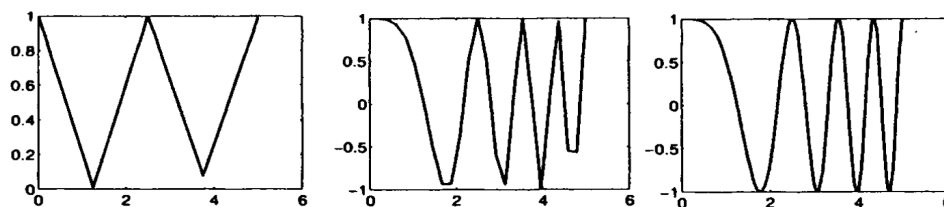


Figure 1.2: Plots of the function $y = \cos(x^2)$ on $[0, 5]$ with increasing resolution: (a) (left) 5 plotting points, (b) (middle) 25 plotting points, and (c) (right) 300 plotting points.

If you want to add more graphs to an existing plot, enter the command:

```
>> hold on %do this!
```

⁵Numerous attributes of a MATLAB plot or other graphic can be modified using the various (very user-friendly) menu options available on the MATLAB graphics window. These include font sizes, line styles, colors, and thicknesses, axis label and tick locations, and numerous other items. To improve readability of this book we will use such features without explicit mention (mostly to make the fonts more readable to accommodate reduced figure sizes).

All future graphs will be added to the existing one until you enter *hold off*. To see how this works, let's go ahead and add the graphs of $y = \cos(2x)$ and $y = \cos^2 x$ to our existing plot of $y = \cos(x^2)$ on $[0, 5]$. To distinguish these plots, we might want to draw the curves in different styles and perhaps even different colors. Table 1.1 is a summary of the codes you can use in a MATLAB plot command to get different plot styles and colors:

Table 1.1: MATLAB codes for plot colors and styles.

Color/Code		Plot Style/Code	
black / k	red / r	solid / -	stars / *
blue / b	white / w	dashed / --	x-marks / x
cyan / c	yellow / y	dotted / :	circles / o
green / g		dash-dot / -.	plus-marks / +
magenta / m		points / .	pentacles / p

Suppose that we want to produce a dashed cyan graph of $y = \cos(2x)$ and a dotted red graph of $y = \cos^2 x$ (to be put in with the original graph). We would enter the following:

```
>>y1 = cos (2*x);
>>plot(x,y1,'c--') %will plot with cyan dashed curve
>>y2 = cos(x).^2; % cos(x)^2 would produce an error
>>plot(x,y2,'r:') %will plot in dotted red style
>>hold off %puts an end to the current graph
```

You should experiment now with a few other options. Note that the last four of the plot styles will put the given object (stars, x-marks, etc.) around each point that is actually plotted. Since we have so many points (300) such plots would look like very thick curves. Thus these last four styles are more appropriate when the density of plot points is small. You can see the colors on your screen, but unless you have a color printer you should make use of the plot styles to distinguish between multiple graphs on printed plots.

Many features can be added to a plot. For example, the steps below show how to label the axes and give your plot a title.

```
>>xlabel('x')
>>ylabel('cos(x.^2), cos(2*x), cos(x).^2')
>>title('Plot created by yourname')
```

Notice at each command how your plot changes; see Figure 1.3 for the final result

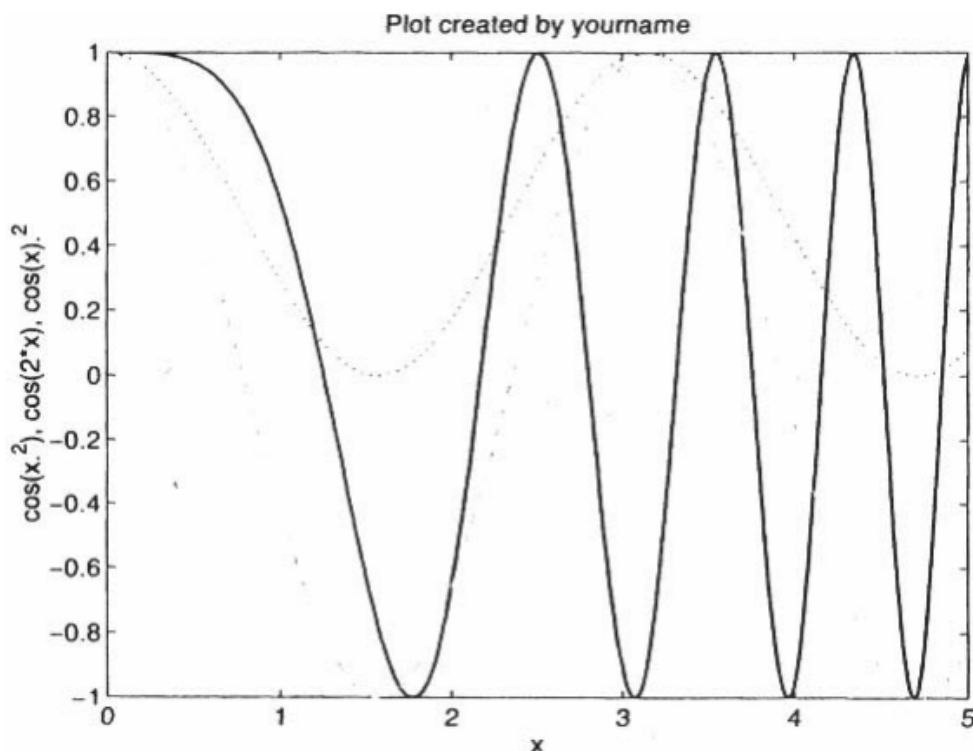


Figure 1.3: Plot of three different trigonometric functions done using different colors and styles

In a MATLAB plot, the points and connecting line segments need not define the graph of a function. For example, to get MATLAB to draw the unit square with vertices (0,0), (1,0), (1,1), (0,1), we could key in the x- and y-coordinates (in an appropriate order so the connecting segments form a square) of these vertices as row vectors. We need to repeat the first vertex at the end so the square gets closed off. Enter

```
>>x=[0 1 1 0 0] ; y=[0 0 1 1 0] ;
>>plot(x,y)
```

Often in mathematics, the variables x and y are given in terms of an auxiliary variable, say t (thought of as time), rather than y simply being given in terms of (i.e., a function of) x . Such equations are called **parametric equations**, and are easily graphed with MATLAB. Thus parametric equations (in the plane) will look like:

$$\begin{cases} x = x(t) \\ y = y(t) \end{cases}$$

These can be used to represent any kind of curve and are thus much more versatile than functions $y = f(x)$ whose graphs must satisfy the vertical line test. MATLAB's plotting format makes plotting parametric equations a simple task. For example, the following parametric equations

$$\begin{cases} x = 2\cos(t) \\ y = 2\sin(t) \end{cases}$$

represent a circle of radius 2 and center (0,0). (Check that they satisfy the equation $x^2 + y^2 = 4$.) To plot the circle, we need only let t run from 0 to $2/\pi$ (since the whole circle gets traced out exactly once as t runs through these values). Enter:

```
>>t = 0:.01:2*pi; % a lot of points for decent resolution , as you
>>% guessed, 'pi' is how MATLAB denotes $\pi$
>>x = 2*cos(t);
>>y = 2*sin(t);
>>plot(x,y)
```

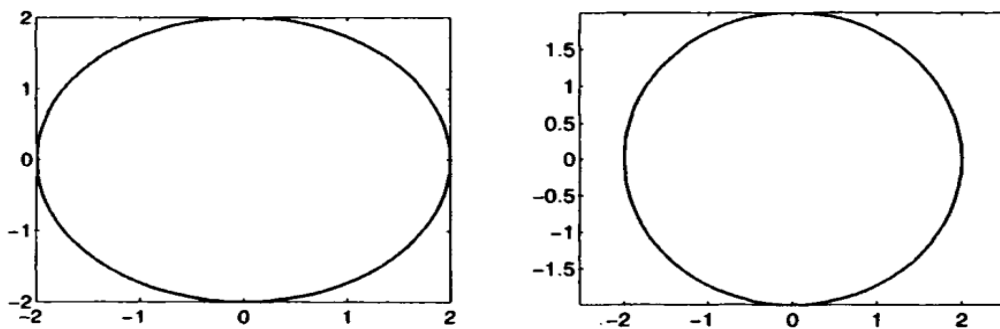


Figure 1.4: Parametric plots of the circle $x^2 + y^2 = 4$, (a) (left) first using MATLAB's default rectangular axis setting, and then (b) (right) after the command `axis('equal')` to put the axes into proper perspective.

You will see an ellipse in the figure window (Figure 1.4(a)). This is because MATLAB uses different scales on the x - and y -axes, unless told otherwise. If you enter: `axis('equal')`, MATLAB will use the same scale on both axes so the circle appears as it should (Figure 1.4(b)). Do this!

EXERCISE FOR THE READER 1.2: In the same fashion use MATLAB to create a plot of the more complicated parametric equations:

$$\begin{cases} x(t) = 5\cos(t/5) + \cos(2t) & \text{for } 0 < t < 10\pi \\ y(t) = 5\sin(t/5) + \sin(3t) \end{cases}$$

Caution: Do not attempt to plot this one by hand!

If you use the `axis('equal')` command in Exercise for the Reader 1.2, you should be getting the plot pictured in Figure 1.5.

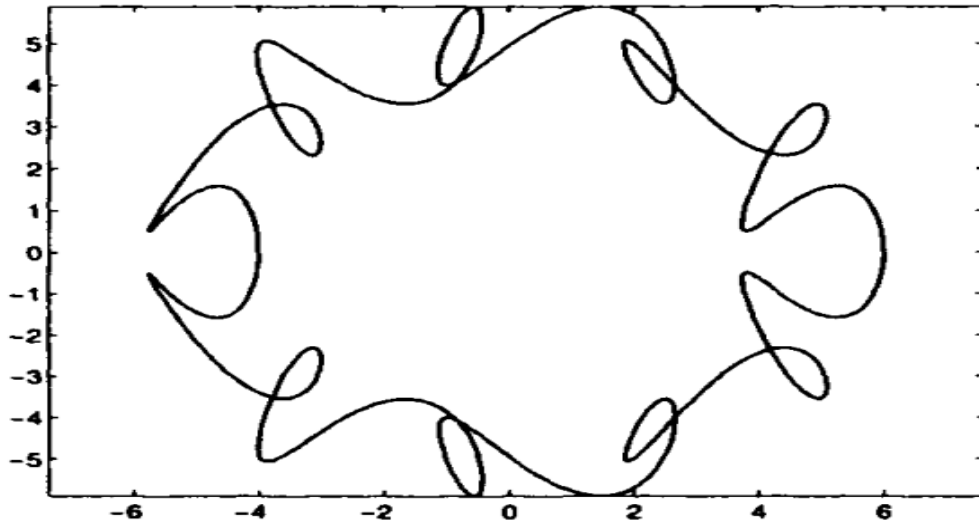


Figure 1.5: A complicated MATLAB parametric plot.

EXERCISES 1.4:

1. Use MATLAB to plot the graph of $\sin(x^4)$ for $0 \leq x \leq 2\pi$, (a) using 200 plotting points, and (b) using 5000 plotting points.
2. Use MATLAB to plot the graph of $y = e^{(-1/x^2)}$ for $-3 \leq x \leq 3$, (a) using 50 plotting points, and (b) using 10,000 plotting points.

<code>axis([xmin xmax ymin ymax])</code> →	Resets the axis range for plots to be: $x_{\min} \leq x \leq x_{\max}$ $y_{\min} \leq y \leq y_{\max}$ Here, the four vector entries can be any real numbers with $x_{\min} < x_{\max}$, and $y_{\min} < y_{\max}$
--	---

3. Use MATLAB to produce a nice plot of the graph of $y = \frac{2-x^2}{x^2+x-6}$ on the interval $[-5, 5]$. Experiment a bit with the axis command as explained in the above note.
4. Use MATLAB to plot the graph of $y = \frac{x^4-16}{x^3+2x^2-6}$ on the interval $[-1, 5]$. Adjust the axes, as explained in the note preceding Exercise 3, so as to get an attractive plot.
5. Use MATLAB to plot the circle of radius 3 and center $(-2, 1)$.
6. Use MATLAB to obtain a plot of the epicycloids that are given by the following parametric

$$\begin{cases} x(t) = (R+r)\cos t - r\cos\left(\frac{R+r}{r}t\right) \\ y(t) = (R+r)\sin t - r\sin\left(\frac{R+r}{r}t\right) \end{cases}, 0 \leq t \leq 2\pi \quad (1.1)$$

equations: using first the parameters $R = 4, r = 1$, and then $R = 12, r = 5$. Use no less than 1000 plotting points.

Note: An epicycloid describes the path that a point on the circumference of a smaller circle (of radius r) makes as it rolls around (without slipping) a larger circle (of radius R).

7. Use MATLAB to plot the parametric equations:

$$\begin{cases} x(t) = e^{-\sqrt{t}} \cos(t) \\ y(t) = e^{-\sqrt{2t}} \sin(t) \end{cases}, 0 \leq t \leq 100.$$

8. Use MATLAB to produce a plot of the linear system (two lines):

$$\begin{cases} 2x + 3y = 13 \\ 2x - y = 1 \end{cases}$$

Include a label for each line as well as a label of the solution (that you can easily find by hand), all produced by MATLAB.

Hints: You will need the hold on command to include so many things in the same graph. To insert the labels, you can use either of the commands below to produce the string of text label at the coordinates (x, y) .

<code>text(x, y, 'label')</code> →	Inserts the text string <code>label</code> in the current graphic window at the location of the specified point (x,y)
<code>gtext('label')</code> →	Inserts the text string <code>label</code> in the current graphic window at the location of exactly where you click your mouse.

9. Use MATLAB to draw a regular octagon (stop-sign shape). This means that all sides have the same length and all interior angles are equal. Scale the axes accordingly.
10. By using the plot command (repeatedly and appropriately), get MATLAB to produce a circle inscribed in a triangle that is in

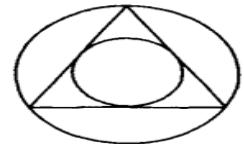


Figure 1.6: Illustration for Exercise 10.

11. By using the plot command (repeatedly and appropriately), get MATLAB to produce something as close as possible to the familiar figure on the right. Do not worry about the line/curve thickness for now, but try to get it so that the eyes (dots) are reasonably visible.



1.5 A TUTORIAL INTRODUCTION TO RECURSION ON MATLAB

Getting a calculator or computer to perform a single task is interesting, but what really makes computers such powerful tools is their ability to perform a long series of related tasks. Such multiple tasks often require a program to tell the computer 1.5: A Tutorial Introduction to Recursion on MATLAB IS what to do. We will get more into this later, but it is helpful to have a basic idea at this point of how this works. We will now work on a rather elementary problem from finance that will actually bring to light many important concepts. There are several programming commands in MATLAB, but this tutorial will focus on just one of them (`while`) that is actually quite versatile.

PROBLEM: To pay off a \$100,000.00 loan, Beverly pays \$1,000.00 at the end of each month after having taken out the loan. The loan charges 8% annual interest ($=8/12\%$ monthly interest) compounded monthly on the unpaid balance. Thus, at the end of the first month, the balance on Beverly's account will be (rounded to two decimals): \$100,000 (prev. balance) + \$666.27 (interest rounded to two decimals) − \$1,000 (payment) = \$99,666.67. This continues until Beverly pays off the balance; her last payment might be less than \$1,000 (since it will need to cover only the final remaining balance and the last month's interest).

- (a) Use MATLAB to draw a plot of Beverly's account balances (on the y -axis) as a function of the number of months (on the x -axis) until the balance is paid off.
- (b) Use MATLAB to draw a plot of the accrued interest (on the y -axis) that Beverly has paid as a function of the number of months (on the x -axis).
- (c) How many years + months will it take for Beverly to completely pay off her loan? What will her final payment be? How much interest will she have paid off throughout the course of the loan?
- (d) Use MATLAB to produce a table of values, with one column being Beverly's outstanding balance given in yearly (12 month) increments, and the second column being her total interest paid, also given in yearly increments. Paste the data you get into your word processor to produce a cleaner table of this data.
- (e) Redo part (c) if Beverly were to increase her monthly payments to \$1,500.

Our strategy will be as follows: We will get MATLAB to create two vectors `B` and `TI` that will stand for Beverly's account balances (after each month) and the total interest accrued. We will set it up so that the last entry in `B` is zero, corresponding to Beverly's account finally being paid off.

There is another way to construct vectors in MATLAB that will suit us well here. We can simply assign the entries of the vector one by one. Let's first try it with the simple example of the vector $x = [1 \ 5 \ -2]$. Start a new MATLAB session and enter:

```
>>x(1) = 1 %specifies the first entry of the vector x, at this point
>> %x will only have one entry
>>x(2) = 5 %you will see from the output x now has the first two of
>>%its three components
>>x(3) = -2
```

The trick will be to use **recursion formulas** to automate such a construction of B and TI. This is possible since a single formula shows how to get the next entry of B or TI if we know the present entry. Such formulas are called recursion formulas and here is what they look like in this case:

$$B(i+1) = B(i) + (.08/12)B(i) - 1000$$

$$TI(i+1) = TI(i) + (.08/12)B(i)$$

In words: The next month's account balance ($B(i+1)$) is the current month's balance ($B(i)$) plus the month's interest on the unpaid balance ($(.08/12)B(i)$) less Beverly's monthly payment. Similarly, the total interest accrued for the next month equals that of the current month plus the current month's interest.

Since these formulas allow us to use the information from any month to get that for the next month, all we really need are the initial values $B(1)$ and $TI(1)$, which are the initial account balance (after zero months) and total interest accrued after zero months. These are of course \$100,000.00 and \$0.00, respectively.

Caution: It is tempting to call these initial values $B(0)$ and $TI(0)$, respectively. However this cannot be done since they are, in MATLAB, vectors (remember, as far as numerical data is concerned: Everything in MATLAB is a matrix [or a vector]!) rather than functions of time, and indices of matrices and vectors must be positive integers ($i = 1, 2, \dots$). This takes some getting used to since i , the index of a vector, often gets mixed up with t , an independent variable, especially by novice MATLAB users.

We begin by initializing the two vectors B and T I as well as the index i .

```
>>B(1)=100000; TI(1)=0; i=1;
```

Next, making use of the recursion formulas, we wish to get MATLAB to figure out all of the other entries of these vectors. This will require a very useful device called a "while loop". We want the while loop to keep using the recursion formulas until the account balance reaches zero. Of course, if we did not stop using the recursion formulas, the balance would keep getting more and more negative and we would get stuck in what is called an **infinite loop**. The format for a while loop is as follows:

```
>>while <condition>
...MATLAB commands
>>end
```

The way it works is that if the <condition> is met, as soon as you enter end, the "...MATLAB commands..." within the loop are executed, one by one, just as if you were typing them in on the command window. After this the <condition> is reevaluated. If it is still met, the "...MATLAB commands..." are again executed in order. If the <condition> is not met, nothing more is done (this is called exiting the loop). The process continues. Either it eventually terminates (exits the loop) or it goes on forever (an infinite loop—a bad program). Let's do a simple example before returning to our problem. Before you enter the following commands, try to guess, based on how we just explained while loops, exactly what MATLAB's output will be. Then check your answer with MATLAB's actual output on your screen. If you get it right you are starting to understand the concept of while loops.

```
>>a=1;
>>while a^2 < 5 * a
a=a+2, a^2
end
```

EXERCISE FOR THE READER 1.3: Analyze and explain each iteration of the above while loop. Note the equation $a = a + 2$ in mathematics makes no sense at all. But remember, in MATLAB the single equal sign means "assignment." So for example, initially $a = 1$. The first run through the while loop the condition is met ($1 = a^2 < 5a = 5$) so a gets reassigned to be $1 + 2 = 3$, and in the same line a^2 is also called to be computed (and listed as output).

Now back to the solution of the problem. We want to continue using the above recursion formulas as long as the balance $B(i)$ remains positive. Since we have already initialized $B(1)$ and $TI(1)$, one possible MATLAB code for creating the rest of these vectors would look like:

```
>>while B(i) > 0
B(i+1)=B(i)+ 8/12/100*B(i)-1000; % This and the next are just our recursion formulas.
TI(i+1)=TI(i)+ 8/12/100*B(i);
    i=i+1; % this bumps the vector index up by one at each iteration.
end
```

Notice that MATLAB does nothing, and the prompt does not reappear again, until the while loop is closed off with an end (and you press enter). Although we have suppressed all output, MATLAB has done quite a lot; it has created the vectors B and TI . Observe also that the final balance of zero should be added on as a final component. There is one subtle point that you should be aware of: The value of i after the termination of the while loop is precisely one more than the number of entries of the vectors B and TI thus far created. Try to convince yourself why this is true! Thus we can add on the final entry of B to be zero as follows:

```
>> n=i; B(n)=0; %We could have just typed 'B(i)=0' but we wanted to
>>% call 'n' the length of the vector B.
```

Another subtle point is that $B(n)$ was already assigned by the while loop (in the final iteration) but was not a positive balance. This is what caused the while loop to end. So what actually will happen at this stage is that Beverly's last monthly payment will be reduced to cover exactly the outstanding balance plus the last month's interest. Also in the final iteration, the total interest was correctly given by the while loop. To do the required plots, we must first create the time vector. Since time is in months, this is almost just the vector formed by the indices of B (and TI), i.e., it is almost the vector $[1\ 2\ 3\ \dots\ n]$. But remember there is one slight subtle twist. Time starts off at zero, but the vector index must start off at 1. Thus the time vector will be $[0\ 1\ 2\ \dots\ n-1]$. We can easily construct it in MATLAB by

```
>>t=0:n-1; %this is shorthand for 't=0:1:n-1', by default the
>>% gap size is one.
```

Since we now have constructed all of the vectors, plotting the needed graphs is a simple matter.

```
>>plot(t,B)
>>xlabel('time in months'), ylabel('unpaid balance in dollars')
>> %we add on some descriptive labels on the horizontal and vertical
>> %axis. Before we go on we copy this figure or save it (it is
>> %displayed below)
>>plot(t, TI)
>> xlabel('time in months'), ylabel('total interest paid in dollars')
```

See Figure 1.7 for the MATLAB graphics outputs.

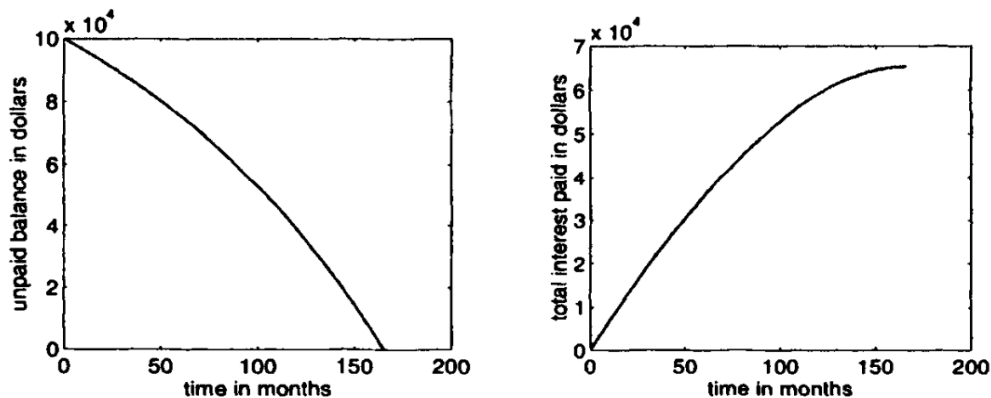


Figure 1.7: (a) (top) Graph of the unpaid balance in dollars, as a function of elapsed months in the loan of \$100,000 that is being analyzed, (b) (bottom) Graph of the total interest paid in dollars, as a function of elapsed months in the loan of \$100,000 that is being analyzed.

We have now taken care of parts (a) and (b). The answer to part (c) is now well within reach. We just have to report the correct components of the appropriate vectors. The time it takes Beverly to pay off her loan is given by the last value of the time vector, i.e.,

```
>>n-1
->166.00 =13 years + 10 months (time of loan period).
```

Her final payment is just the second-to-last component of B, with the final month's interest added to it (that's what Beverly will need to pay to totally clear her account balance to zero):

```
>>format bank % this puts our dollar answers to the nearest cent.
>>B(n-1)*(1+8/12/100)
>>$341.29 (last payment).
```

The total interest paid is just:

```
>>TI(n)
->$65,341.29 (total interest paid)
```

Part (d): Here we simply need to display parts of the two vectors, corresponding to the ends of the first 13 years of the loan and finally the last month (the 10th month after the 13th year). To get MATLAB to generate these two vectors, we could use a while loop as follows:⁶

```
>> k=1; i=1; \%we will use two indices, k will be for the original
>>\% vectors, i will be for the new ones.
>>while k<167
    YB(i)=B(k); YTI(i)=TI(k); %we create the two new "yearly" vectors.
    k=k+12; i=i+1; %at each iteration, the index of the original
    %vectors gets bumped up by 12, but that for
    %the new vectors gets bumped up only by one.
end
```

We next have to add the final component onto each vector (it does not correspond to a year's end). To do this we need to know how many components the yearly vectors already have. If you think about it you will see it is 14, but you could have just as easily asked MATLAB to tell you:

```
>> size(YB) %this command gives the size of any matrix or vector
% (\# of rows, \# of columns).
->ans = 1.00 14.00
>>YB(15)=B(167); YTI(15)=TI(167);
>>YB=YB'; YTI=YTI'; % this command reassigns both yearly vectors to
% be column vectors
```

Before we print them, we would like to print along the left side the column vector of the corresponding years' end. This vector in column form can be created as follows:

```
>>years = 0:14; years = years' %first we create it as a row vector
>>%and then transpose it.
```

We now print out the three columns:

```
>>years, YB, YTI % or better (years, YB, YTI)
```

years=	YB=	YTI=
0	100000.00	0
1.00	95850.02	7850.02
2.00	91355.60	15355.60
3.00	86488.15	22488.15
4.00	81216.69	29216.69
5.00	75507.71	35507.71
6.00	69324.89	41324.89
7.00	62628.90	46628.90
8.00	55377.14	51377.14
9.00	47523.49	55523.49
10.00	39017.99	59017.99
11.00	29806.54	61806.54
12.00	19830.54	63830.54
13.00	9026.54	65026.54
14.00	0.00	65341.29

⁶A slicker way to enter these vectors would be to use MATLAB's special vector-creating construct that we mentioned earlier as follows: YB = B(1:12:167), and similarly for YTI.

Finally, by making use of any decent word processing software, we can embellish this rather raw data display into a more elegant form such as Table 1.2.

Table 1.2: Summary of annual data for the \$100,000 loan that was analyzed in this section.

Years Elapsed:	Account Balance:	Total Interest Paid:
0	\$100000.00	\$ 0
1	95850.02	7850.02
2	91355.60	15355.60
3	86488.15	22488.15
4	81216.69	29216.69
5	75507.71	35507.71
6	69324.89	41324.89
7	62628.90	46628.90
8	55377.14	51377.14
9	47523.49	55523.49
10	39017.99	59017.99
11	29806.54	61806.54
12	19830.54	63830.54
13	9026.54	65026.54
13 + 10 months	0.00	65341.29

Part (e): We can run the same program but we need only modify the line with the recursion formula for the vector B: It now becomes: $B(i+1) = B(i) + I(i+1) - 1500$; . With this done, we arrive at the following data:

```
>>i-1, B(i-1)*(1+8/12/100) , TI(i)
->89 (7 years + 5 months), \ $693.59 (last pmt), \ $32693.59 (total interest paid)
```

EXERCISES 1.5:

1. Use a while loop to add all of the odd numbers up: $1 + 3 + 5 + 7 + \dots$ until the sum exceeds 5 million. What is the actual sum? How many odd numbers were added?
2. Redo Exercise 1 by adding up the even numbers rather than the odd ones.
3. (*Insects from Hell*) An insect population starts off with one pair at year zero. The insects are immortal (i.e., they never die!) and after having lived for two years each pair reproduces another pair (assumed male and female). This continues on indefinitely. So at the end of one year the insect population is still 1 pair (= 2 insects); after two years it is $1 + 1 = 2$ pairs (= 4 insects), since the original pair of insects has reproduced. At the end of the third year it is $1 + 2 = 3$ pairs (the new generation has been alive for only one year, so has not yet reproduced), and after 4 years the population becomes $2 + 3 = 5$ pairs, (a) Find out the insect population (in pairs) at the end of each year from year 1 through year 10. (b) What will the insect population be at the end of 64 years?

HISTORICAL ASIDE: The sequence of populations in this problem: 1,1,1 + 1=2 , 1+ 2 = 3, 2 + 3 = 5,3 + 5 = 8,... was first introduced in the middle ages by the Italian mathematician Leonardo of Pisa (ca. 1180-1250), who is better known by his nickname: Fibonacci (Italian, meaning son of Bonaccio). This sequence has numerous applications and has made Fibonacci quite famous. It comes up, for example, in hereditary effects in incest, growth of pineapple cells, and electrical engineering. There is even a mathematical journal named in Fibonacci's honor (the Fibonacci Quarterly).

4. Continuing Exercise 3, (a) produce a chart of the insect populations at the end of each 10th year until the end of year 100. (b) Use a while loop to find out how many years it takes for the insect population (in pairs) to exceed 1,000,000,000 pairs.
5. (*Another Insect Population Problem*) In this problem, we also start off with a pair of insects, this time mosquitoes. We still assume that after having lived for two years, each pair reproduces another pair. But now, at the end of three years of life, each pair of mosquitoes reproduces one more pair and then immediately dies, (a) Find out the insect population (in pairs) for each year up through year 10. (b) What will the insect population be at the end of 64 years?

6. Continuing Exercise 5, (a) plot the mosquito (pair) population from the beginning through the end of year 500, as a function of time, (b) How many years does it take for the mosquito population (in pairs) to exceed 1,000,000,000 pairs?
7. When their daughter was born, Mr. and Mrs. de la Hoya began saving for her college education by investing \$5,000 in an annuity account paying 10% interest per year. Each year on their daughter's birthday they invest \$2,000 more in the account, (a) Let A_n denote the amount in the account on their daughter's n th birthday. Show that A_n satisfies the following recursion formulas:

$$\begin{aligned} A_0 &= 5000 \\ A_n &= (1.1)A_{n-1} + 2000. \end{aligned} \quad (1.2)$$

(b) Find the amount that will be in the account when the daughter turns 18.

(c) Print (and nicely label) a table containing the values of n and A_n as n runs from 0 to 18.

8. Louise starts an annuity plan at her work, that pays 9% interest compounded monthly. She deposits \$200 each month starting on her 25th birthday. Thus at the end of the first month her account balance is exactly \$200. At the end of the second month, she puts in another \$200, but her first deposit has earned her one month's worth of interest. The 9% means she gets $9\%/12 = 0.75\%$ interest per month. Thus the interest she earns in going from the first to second month is $.75\%$ of \$200 or \$1.50, and so her balance at the end of the second month is 401.50. This continues, so at the end of the 3rd month, her balance is \$401.50 (old balance) + $.75\%$ of this (interest) + \$200 (new deposit) = \$604.51. Louise continues to do this throughout her working career until she retires at age 65.
- (a) Figure out the balances in Louise's account at her birthdays: 26th, 27th, ..., up through her 65th birthday. Tabulate them neatly in a table (either cut and paste by hand or use your word processor—do not just give the raw MATLAB output, but rather put it in a form so that your average citizen could make sense of it).
- (b) At exactly what age (to the nearest month) is Louise when the balance exceeds \$100,000? Note that throughout these 40 years Louise will have deposited a total of \$200/month \times 12 months/yr. \times 40 years = \$96,000.
9. In saving for retirement, Joe, a government worker, starts an annuity that pays 12% interest compounded monthly. He deposits \$200.00 into it at the end of each month. He starts this when he is 25 years old. (a) How long will it take for Joe's annuity to reach a value of \$1 million? (b) Plot Joe's account balance as a function of time.
10. The dot product of two vectors of the same length is defined as follows:

If

$$x = [x(1)x(2)\cdots x(n)], y = [y(1)y(2)\cdots y(n)]$$

then

$$x \cdot y = \sum_{i=1}^n x(i)y(i)$$

The dot product appears and is useful in many areas of math and physics. As an example, check that the dot product of the vectors $[2 \ 0 \ 6]$ and $[1 \ -1 \ 4]$ is 26. In MATLAB, if x and y are stored as row vectors, then you can get the dot product by typing x^*y' (the prime stands for transpose, as in the previous section; Chapter 7 will explain matrix operations in greater detail). Let x and y be the vectors each with 100 components having the forms:

$$\begin{aligned} x &= [1, -1, 1, -1, 1, -1, \dots], \\ y &= [1, 4, 9, 16, 25, 36, \dots]. \end{aligned}$$

Use a while loop in MATLAB to create and store these vectors and then compute their dot product.

Chapter 2

Basic Concepts of Numerical Analysis with Taylor's Theorem

2.1 WHAT IS NUMERICAL ANALYSIS?

Outside the realm of pure mathematics, most practicing scientists and engineers are not concerned with finding exact answers to problems. Indeed, living in a finite universe, we have no way of exactly measuring physical quantities and even if we did, the exact answer would not be of much use. Just a single irrational number, such as

$$\pi = 3.1415926535897932384626433832795028841971693993751058209749\dots,$$

where the digits keep going on forever without repetition or any known pattern, has more information in its digits than all the computers in the world could possibly ever store. To help motivate some terminology, we bring forth a couple of examples.

Suppose that Los Angeles County is interested in finding out the amount of water contained in one of its reserve drinking water reservoirs. It hires a contracting firm to measure this amount. The firm begins with a large-scale pumping device to take care of most of the water, leaving only a few gallons. After this, they bring out a more precise device to measure the remainder and come out with a volume of 12,564,832.42 gallons. To get a second opinion, the county hires a more sophisticated engineering firm (that charges 10 times as much) and that uses more advanced measuring devices. Suppose this latter firm came up with the figure 12,564,832.3182. Was the first estimate incorrect? Maybe not, perhaps some evaporation or spilling took place—so there cannot really be an exact answer to this problem. Was it really worth the extra cost to get this more accurate estimate? Most likely not—even an estimate to the nearest gallon would have served equally well for just about any practical purposes.

Suppose next that the Boeing Corporation, in the design and construction of a new 767 model jumbo jet, needs some wing rivets. The engineers have determined the rivets should have a diameter of 2.75 mm with a tolerance (for error) of .000025 mm. Boeing owns a precise machine that will cut such rivets to be of diameter $2.75 \pm .000006$ mm. But they can purchase a much more expensive machine that will produce rivets of diameters $2.75 \pm .0000001$ mm (60 times as accurate). Is it worth it for Boeing to purchase and use this more expensive machine? The aeronautical engineers have determined that such an improvement in rivets would not result in any significant difference in the safety and reliability of the wing and plane; however, if the error exceeds the given tolerance, the wings may become unstable and a safety hazard.

In mathematics, there are many problems and equations (algebraic, differential, and partial differential) whose exact solutions are known to exist but are difficult, very time consuming, or impossible to solve exactly. But for many practical purposes, as evidenced by the previous examples, an estimate to the exact answer will do just fine, provided that we have a guarantee that the error is not too large. So, here is the basic problem in numerical analysis:

We are interested in a solution x (= **exact answer**) to a problem or equation. We would like to find an estimate x^* (= **approximation**) so that $|x - x^*|$ (= **the actual error**) is no more than the maximum **tolerance** ($= \epsilon$), i.e., $|x - x^*| < \epsilon$. The maximum tolerated error will be specified ahead of time and will depend on the particular problem at hand. What makes this approximation problem very often extremely difficult is that we usually do not know x and thus, even after we get x^* , we will have no way of knowing the actual error. But regardless of this, we still need to be able to guarantee that it is less than ϵ . Often more useful than the actual error is the relative error, which measures the error as a ratio in terms of the magnitude of the actual quantity; i.e., it is defined by

$$\text{relative error} = \frac{|x - x^*|}{|x|}$$

provided, of course, that $x \neq 0$.

EXAMPLE 2.1: In the Los Angeles reservoir measurement problem given earlier, suppose we took the exact answer to be the engineering firm's estimate, $x = 12,564,832.3182$ gallons, and the contractor's estimate as the approximation $x^* = 12,564,832.42$. Then the error of this approximation is $|x - x^*| = 0.1018$ gallons,

EXAMPLE 2.2: In the Boeing Corporation's rivet problem above, the maximum tolerated error is $.000025mm$, which translates to a maximum relative error of (divide by $x = 2.75$) 0.000009 . The machine they currently have would yield a maximum relative error of $0.000006/2.75 = 0.000002$ and the more expensive machine they were considering would guarantee a maximum relative error of no more than $0.0000001/2.75 = 3.6364 \times 10^{-8}$.

For the following reasons, we have chosen Taylor's theorem as a means to launch the reader into the realm of numerical analysis. First, Taylor's theorem is at the foundation of most numerical methods for differential equations, the subject of this book. Second, it covers one of those rare situations in numerical analysis where quality error estimates are readily available and thus errors can be controlled and estimated quite effectively. Finally, most readers should have some familiarity with Taylor's theorem from their calculus courses.

Most mathematical functions are very difficult to compute by just using the basic mathematical operations: $+$, $-$, \times , $*$. How, for example, would we compute $\cos(27^\circ)$ just using these operations? One type of function that is possible to compute in this way is a polynomial. A polynomial in the variable x is a function of the form:

$$p(x) = a_n x^n + \cdots + a_2 x^2 + a_1 x + a_0,$$

where $a_n, \dots, a_2, a_1, a_0$ are any real numbers. If $a_n \neq 0$, then we say that the degree of $p(x)$ equals n . Taylor's theorem from calculus shows how to use polynomials to approximate a great many mathematical functions to any degree of accuracy. In Section 2.2, we will introduce the special kind of polynomial (called Taylor polynomials) that gets used in this theorem and in Section 2.3 we discuss the theorem and its uses.

EXERCISES 2.1:

1. If $x = 2$ is approximated by $x^* = 1.96$, find the actual error and the relative error.
2. If $\pi(= x)$ is approximated by $x^* = 3\frac{1}{8}$ (as was done by the ancient Babylonians, c. 2000BC), find the actual error and the relative error.
3. If $x = 10000$ is approximated by $x^* = 9999.96$, find the actual error and the relative error.
4. If $x = 5280$ feet (one mile) is approximated by $x^\circ = 5281$ feet, find the actual and relative errors.
5. If $x = 0.76$ inches and the relative error of an approximation is known to be 0.05 , find the possible values for x^* .
6. If $x = 186.4$ and the relative error of an approximation is known to be 0.001 , find the possible values for x^* .
7. A civil engineering firm wishes to order thick steel cables for the construction of a span bridge. The cables need to measure 2640 feet in length with a maximum tolerated relative error of 0.005 . Translate this relative error into an actual tolerated maximum discrepancy from the ideal 2640 foot length.

2.2 TAYLOR POLYNOMIALS

Suppose that we have a mathematical function $f(x)$ that we wish to approximate near $x = a$. **The Taylor polynomial of order n , $p_n(x)$** , for this function **centered at (or about) $x = a$** is that polynomial of degree of at most n that has the same values as $f(x)$ and its first n derivatives at $x = a$. The definition requires that $f(x)$ possess n derivatives at $x = a$. Since derivatives measure rates of change of functions, the Taylor polynomials are designed to mimic the behavior of the function near $x = a$. The following example will demonstrate this property.

EXAMPLE 2.3: Find formulas for, and interpret, the order-zero and order-one Taylor polynomials $p_0(x)$ and $p_1(x)$ of a function $f(x)$ (differentiable) at $x = a$.

SOLUTION: The zero-order polynomial $p_0(x)$ has degree at most zero, and so must be a constant function. But by its definition, we must have $p_0(a) = f(a)$. Since $p_0(x)$ is constant this means that $p_0(x) = f(a)$ (a horizontal line function). The first-order polynomial $p_1(x)$ must satisfy two conditions:

$$p_1(a) = f(a) \text{ and } p_1'(a) = f'(a) \tag{1}$$

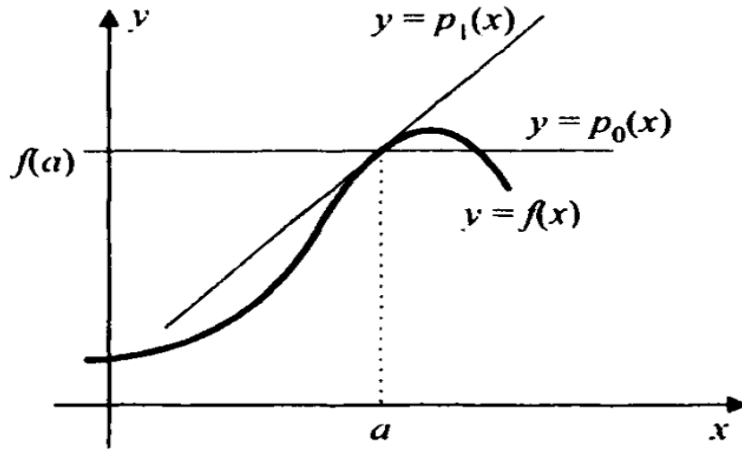


Figure 2.1: Illustration of a graph of a function $y = f(x)$ (heavy black curve) together with its zero-order Taylor polynomial $p_0(x)$ (horizontal line) and first-order Taylor polynomial $p_1(x)$ (slanted tangent line).

Since $p_1(x)$ has degree at most one, we can write $p_1(x) = mx + b$, i.e., $p_1(x)$ is just a line with slope m and y-intercept b . If we differentiate this equation and use the second equation in (1), we get that $m = f'(a)$. We now substitute this in for m , put $x = a$ and use the first equation in (1) to find that $f(a) = p_1(a) = f'(a)a + b$. Solving for b gives $b = f(a) - f'(a)a$. So putting this all together yields that $p_1(x) = mx + b = f'(a)x + f(a) - f'(a)a = f(a) + f'(a)(x - a)$. This is just the tangent line to the graph of $y = f(x)$ at $x = a$. These two polynomials are illustrated in Figure 2.1.

In general, it can be shown that the Taylor polynomial of order n is given by the following formula:

$$p_n(x) = f(a) + f'(a)(x - a) + \frac{1}{2}f''(a)(x - a)^2 + \frac{1}{3!}f'''(a)(x - a)^3 + \dots + \frac{1}{n!}f^{(n)}(a)(x - a)^n, \quad (2)$$

where we recall that the factorial of a positive integer k is given by:

$$k! = \begin{cases} 1, & \text{if } k = 0 \\ 1 \cdot 2 \cdot 3 \cdots (k-1) \cdot k, & \text{if } k = 1, 2, 3, \dots \end{cases}$$

Since $0! = 1! = 1$, we can use Sigma-notation to rewrite this more simply as:

$$p_n(x) = \sum_{k=0}^n \frac{1}{k!} f^{(k)}(a)(x - a)^k. \quad (3)$$

We turn now to some specific examples:

EXAMPLE 2.4:

- For the function $f(x) = \cos(x)$, compute the following Taylor polynomials at $x = 0$: $p_1(x)$, $p_2(x)$, $p_3(x)$, and $p_8(x)$.
- Use MATLAB to find how each of these approximates $\cos(27^\circ)$ and then find the actual error of each of these approximations.
- Find a general formula for $p_n(x)$.
- Using an appropriate MATLAB graph, estimate the length of the largest interval $[-a, a] = \{|x| \leq a\}$ about $x = 0$ that $p_8(x)$ can be used to approximate $f(x)$ with an error always less than or equal to 0.2. What if we want the error to be less than or equal to 0.001?

SOLUTION: Part (a): We see from formula (2) or (3) that each Taylor polynomial is part of any higher-order Taylor polynomial. Since $a = 0$ in this example, formula (2) reduces to:

$$p_n(x) = f(0) + f'(0)x + \frac{1}{2}f''(0)x^2 + \frac{1}{3!}f'''(0)x^3 + \dots + \frac{1}{n!}f^{(n)}(0)x^n = \sum_{k=0}^n \frac{1}{k!}f^{(k)}(0)x^k. \quad (4)$$

A systematic way to calculate these polynomials is by constructing a table for the derivatives:

n	$f^{(n)}(x)$	$f^{(n)}(0)$
0	$\cos(x)$	1
1	$-\sin(x)$	0
2	$-\cos(x)$	-1
3	$\sin(x)$	0
4	$\cos(x)$	1
5	$-\sin(x)$	0

We could continue on, but if one notices the repetitive pattern (when $n = 4$, the derivatives go back to where they began), this can save some time and help with finding a formula for the general Taylor polynomial $p_n(x)$. Using formula (4) in conjunction with the table (and indicated repetition pattern), we conclude that:

$$p_1(x) = 1, p_2(x) = 1 - \frac{x^2}{2} = p_3(x), \text{ and } p_8(x) = 1 - \frac{x^2}{2} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!}.$$

Part (b): To use these polynomials to approximate $\cos(27^\circ)$, we of course need to take x to be in radians, i.e., $x = 27^\circ \left(\frac{\pi}{180^\circ}\right) = .4712389\dots$. Since two of these Taylor polynomials coincide, there are three different approximations at hand for $\cos(27^\circ)$. In order to use MATLAB to make the desired calculations, we introduce a relevant MATLAB function:

To compute $n!$ in MATLAB, use either: `$ factorial (n)`, or `gamma (n+1)`

```
>>x=27*pi/180;
>>format long
>>p1=1; p2=1-x^2/2
-> 0.888966695048774
>> p8=p2+x^4/gamma(5)-x^6/gamma(7)+x^8/gamma(9)
-> 0.89100652433693
>> abs(p1-cos(x)) %abs, as you guessed, stands for absolute value
->0.10899347581163
>>abs(p2-cos(x)) ->0.00203957370062
>>abs(p8-cos(x)) -> 1.485654932409375e-010
```

Thus for example, to get $5! = 120$, we could either type `>> factorial(5)` or `>gamma(6)`. Now we calculate: Transcribing these into usual mathematics notation, we have the approximations for $\cos(27^\circ)$:

$$p_1(27^\circ) = 1, p_2(27^\circ) = p_3(27^\circ) = .888967\dots, p_8(27^\circ) = .89100694\dots,$$

which have the corresponding errors:

$$\begin{aligned} |p_1(27^\circ) - \cos(27^\circ)| &= 0.1089\dots, \\ |p_2(27^\circ) - \cos(27^\circ)| &= |p_3(27^\circ) - \cos(27^\circ)| = 0.002039\dots, \text{ and} \\ |p_8(27^\circ) - \cos(27^\circ)| &= 1.4856 \times 10^{-10}. \end{aligned}$$

This demonstrates quite clearly how nicely these Taylor polynomials serve as approximating tools. As expected, the higher degree Taylor polynomials do a better job approximating but take more work to compute.

Part (c): Finding the general formula for the n th-order Taylor polynomial $p_n(x)$ can be a daunting task, if it is even possible. It will be possible if some pattern can be discovered with the derivatives of the corresponding function at $x = a$. In this case, we have already discovered the pattern in part (b), which is quite simple: We just need a nice way to write it down, as a formula in n . It is best to separate into cases where n is even or odd. If n is odd we see $f^{(n)}(0) = 0$, end of story. When n is even $f^{(n)}(0)$ alternates between $+1$ and -1 . To get a formula, we write an even n as $2k$, where k will alternate between even and odd integers. The trick is to use either $(-1)^k$ or $(-1)^{k+1}$ for $f^{(2k)}(0)$, which both also alternate between $+1$ and -1 . To see which of the two to use, we need only check the starting values at $k = 0$ (corresponding to $n = 0$). Since $f^{(0)}(0) = 1$, we must use $(-1)^k$. Since any odd integer can be written as $n = 2k + 1$, in summary we have arrived at the following formula for $f^{(n)}(0)$:

$$f^{(n)}(0) = \begin{cases} (-1)^k, & \text{if } n = 2k \text{ is even,} \\ 0, & \text{if } n = 2k + 1 \text{ is odd} \end{cases}.$$

Plugging this into equation (4) yields the formulas:

$$\begin{aligned} p_n(x) &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \dots + (-1)^k \frac{x^{2k}}{(2k)!} = \sum_{j=0}^k (-1)^j \frac{x^{2j}}{(2j)!} \\ &\quad (\text{for } n = 2k \text{ or } n = 2k + 1). \end{aligned}$$

$$(\text{for } n = 2k \text{ or } n = 2k + 1).$$

Part (d): In order to get a rough idea of how well $p_8(x)$ approximates $\cos(x)$, we will first need to try out a few plots. Let us first plot these two functions together on the domain: $-10 \leq x \leq 10$. This can be done as follows:

```
>>x=-10:.0001:10;
>> y=cos(x);
>>p8=1-x.^2/2+x.^4/gamma(5)-x.^6/gamma(7)+x.^8/gamma(9);
>> plot(x,y,x,p8,'r-')
```

Notice that we were able to produce both plots (after having constructed the needed vectors) by a single line, without the hold on/hold off method. We have instructed MATLAB to plot the original function $y = \cos(x)$ in the default color and style (blue, solid line) and the approximating function $y = p_8(x)$ as a red plot with the dash/dot style. The resulting plot is the first one shown in Figure 2.2.

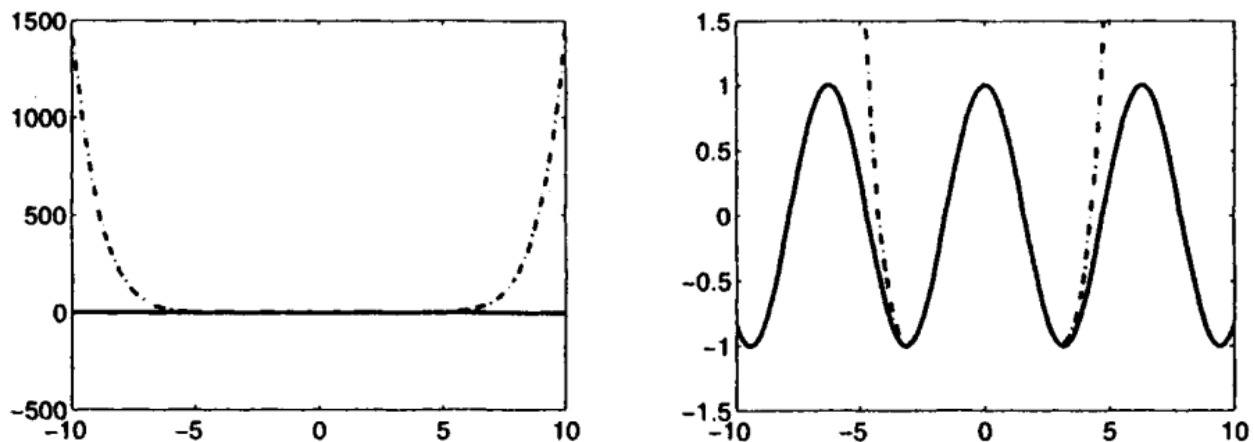


Figure 2.2: Graphs of $y = \cos(x)$ (solid) together with the eighth-order Taylor approximating polynomial $y = p_8(x)$ (dash-dot) shown with two different y-ranges.

To answer (even just) the first question, this plot is not going to help much, owing to the fact that the scale on the y-axis is so large (increments are in 200 units and we need our error to be < 0.2). MATLAB always will choose the scales to accommodate all of the points in any given plot. The eighth-degree polynomial $y = p_8(x)$ gets so large at $x = \pm 10$ that the original function $y = \cos(x)$ is dwarfed in comparison so its graph will appear as a flat line (the x-axis). We could redo the plot trying out different ranges for the x-values and eventually arrive at a more satisfactory illustration.¹ Alternatively and more simply, we can work with the existing plot and get MATLAB to manually change the range of the x - and/or y-axes that appear in the plot. The way to do this is with the command:

<code>axis([xmin xmax ymin ymax]) →</code>	Changes the range of a plot to: $x_{\min} \leq x \leq x_{\max}$ and $y_{\min} \leq y \leq y_{\max}$
--	--

Thus to keep the x-range the same $[-10, 10]$, but to change the y-range to be $[-1.5, 1.5]$, we would enter the command to create the second plot of Figure 2.2.

```
>> axis([-10 10 -1.5 1.5])
```



We can see now from the second plot above that (certainly) for $-3 \leq x \leq 3$ we have $|\cos(x) - p_8(x)| \leq 0.2$. This graph is, however, unsatisfactory in regards to the second question of the determination of an x-interval for which $|\cos(x) - p_8(x)| \leq 0.001$. To answer this latter question and also to get a more satisfactory answer for the first question, we need only look at plots of the actual error $y = |\cos(x) - p_8(x)|$. We do this for two different y-ranges. There is a nice way to get MATLAB to partition its plot window into several (in fact, a matrix of) smaller subwindows.

<code>subplot(m,n,i) →</code>	Causes the plot window to partition into an $m \times n$ matrix of proportionally smaller subwindows, with the next plot going into the i th subwindow (listed in the usual “reading order”—left to right, then top to bottom).
-------------------------------	---

The two error plots in Figure 2.3 were obtained with the following commands:

```
>> subplot(2,1,1)
>> plot(x,abs(y-p8)), axis([-10 10 -.1 .3])
>> subplot(2,1,2)
>> plot(x,abs(y-p8)), axis([-5 5 -.0005 .0015])
```

Notice that the ranges for the axes were appropriately set for each plot so as to make each more suitable to answer each of the corresponding questions.

¹The zoom button  on the graphics window can save some time here. To use it, simply left click on this button with your mouse, then move the mouse to the desired center of the plot at which to zoom and left click (repeatedly). The zoom-out key  works in the analogous fashion.

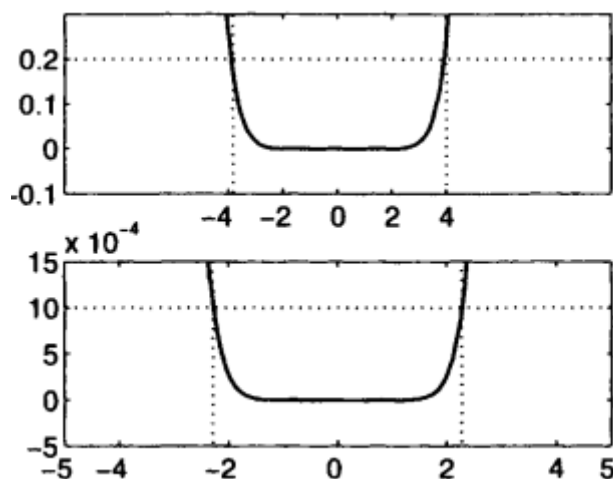


Figure 2.3: Plots of the error $y = |\cos(x) - p_8(x)|$ on two different y-ranges. Reference lines were added to help answer the question in part (d) of Example 2.4.

From Figure 2.3, we can deduce that if we want to guarantee an error of at most 0.2, then we can use $p_8(x)$ to approximate $\cos(x)$ anywhere on the interval $[-3.8, 3.8]$, while if we would like the maximum error to be only 0.001, we must shrink the interval of approximation to about $[-2.2, 2.2]$. In Figure 2.4 we give a MATLAB-generated plot of the function $y = \cos(x)$ along with several of its Taylor polynomials.

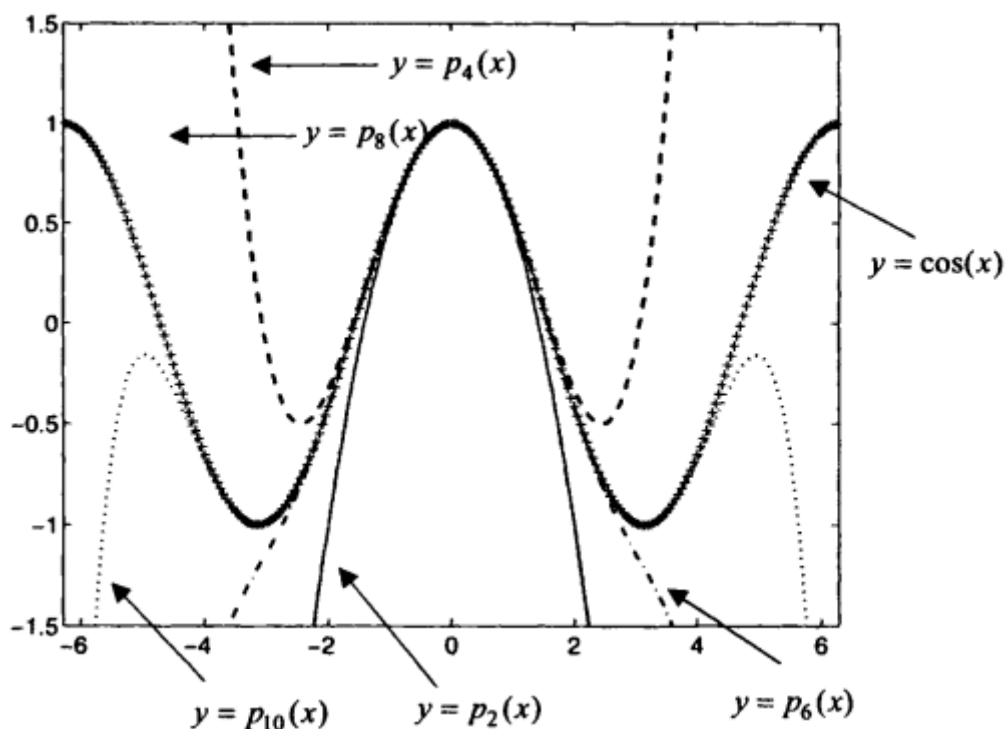


Figure 2.4: Some Taylor polynomials for $y = \cos(x)$.

EXERCISE FOR THE READER 2.1: Use MATLAB to produce the plot in Figure 2.4 (without the arrow labels).

It is a rare situation indeed in numerical analysis where we can actually compute the exact errors explicitly. In the next section, we will give Taylor's theorem, which gives us usable estimates for the error even in cases where it cannot be explicitly computed.

EXERCISES 2.2:

- Find the second- and third-order Taylor polynomials $p_2(x)$ and $p_3(x)$, centered at $x = a$, for each of the following functions.

(a) $f(x) = \sin(x), a = 0$	(b) $f(x) = \tan(x), a = 0$
(c) $f(x) = e^x, a = 1$	(d) $f(x) = x^{1/3}, a = 8$

2. Repeat Exercise 1 for each of the following:
 - (a) $f(x) = \cos(x), a = \pi/2$
 - (b) $f(x) = \arctan(x), a = 0$
 - (c) $f(x) = \ln x, a = 1$
 - (d) $f(x) = \cos(x^2), a = 0$
3. (a) Approximate $\sqrt{65}$ by using the first-order Taylor polynomial of $f(x) = \sqrt{x}$ centered at $x = 64$ (this is tangent line approximation discussed in first-semester calculus) and find the error and the relative error of this approximation.
 (b) Repeat part (a) using instead the second-order Taylor polynomial to do the approximation.
 (c) Repeat part (a) once again, now using the fourth-order Taylor polynomial.
4. (a) Approximate $\sin(92^\circ)$ by using the first-order Taylor polynomial of $f(x) = \sin(x)$ centered at $x = \pi/2$ (tangent line approximation) and find the error and the relative error of this approximation
 (b) Repeat part (a) using instead the second-order Taylor polynomial to do the approximation.
 (c) Repeat part (a) using the fourth-order Taylor polynomial.
5. Find a general formula for the order n Taylor polynomial $p_n(x)$ centered at $x = 0$ for each of the following functions:
 - (a) $y = \sin(x)$
 - (b) $y = \ln(1+x)$
 - (c) $y = e^x$
 - (d) $y = \sqrt{x+1}$
6. Find a general formula for the order n Taylor polynomial $p_n(x)$ centered at $x = 0$ for each of the following functions:
 - (a) $y = \tan(x)$
 - (b) $y = 1/(1+x)$
 - (c) $y = \arctan(x)$
 - (d) $y = x \sin(x)$
7. (a) Compute the following Taylor polynomials, centered at $x = 0$, of $y = \cos(x^2)$:

$$p_1(x), p_2(x), p_6(x), p_{10}(x)$$

(b) Next, use the general formula obtained in Example 2.4 for the general Taylor polynomials of $y = \cos(x)$ to write down the order 0, 1, 3, and 5 Taylor polynomials. Replace x with x^2 in each of these polynomials. Compare these with the Taylor polynomials in part (a).

8. Consider the function $f(x) = \sin(3x)$. All of the plots in this problem are to be done with MATLAB on the interval $[-3, 3]$. The Taylor polynomials refer to those of $f(x)$ centered at $x = 0$. Each graph of $f(x)$ should be done with the usual plot settings, while each graph of a Taylor polynomial should be done with the dot style.

The simultaneous graphs of $f(x)$ along with the 1st-order Taylor polynomial (= tangent line)	A graph of the error $ f(x) - p_1(x) $
The simultaneous graphs of $f(x)$ along with the 3rd-order Taylor polynomial	A graph of the error $ f(x) - p_3(x) $
The simultaneous graphs of $f(x)$ along with the 9th-order Taylor polynomial	A graph of the error $ f(x) - p_9(x) $

9. (a) Let $f(x) = \ln(1+x^2)$. Find formulas for the following Taylor polynomials of $f(x)$ centered at $x = 0$: $p_2(x), p_3(x), p_6(x)$. Next, using the subplot command, create a graphic window split in two sides (left and right). On the left, plot (together) the four functions $f(x), p_2(x), p_3(x), p_6(x)$. In the right-side subwindow, plot (together) the corresponding graphs of the three errors: $|f(x) - p_2(x)|, |f(x) - p_3(x)|$, and $|f(x) - p_6(x)|$. For the error plot adjust the y-range so as to make it simple to answer the question in part (b). Use different styles/colors to code different functions in a given plot.
10. (a) Let $f(x) = x^2 \sin(x)$. Find formulas for the following Taylor polynomials of $f(x)$ centered at $x = 0$: $p_1(x), p_4(x), p_9(x)$. Next, using the subplot command, get MATLAB to create a graphic window split in two sides (left, and right). On the left, plot (together) the four functions $f(x), p_1(x), p_4(x), p_9(x)$. In the right-side subwindow, plot (together) the corresponding graphs of the three errors: $|f(x) - p_1(x)|, |f(x) - p_4(x)|$, and $|f(x) - p_9(x)|$. For the error plot adjust the y-range so as to make it simple to answer the question in part (b). Use different styles/colors to code different functions in a given plot.
11. In Example 2.3, we derived the general formula (2) for the zero- and first-order Taylor polynomial.
 - (a) Do the same for the second-order Taylor polynomial, i.e., use the definition of the Taylor polynomial $p_2(x)$ to show that (2) is valid when $n = 2$.
 - (b) Prove that formula (4) for the Taylor polynomials centered at $x = 0$ is valid for any n .
 - (c) Prove that formula (2) is valid for all n .

Suggestion: For part (c), consider the function $g(x) = f(x+a)$, and apply the result of part (b) to this function. How are the Taylor polynomials of $g(x)$ at $x = 0$ related to those of $f(x)$ at $x = a$?

12. (*Another Kind of Polynomial Interpolation*) In this problem we compare the fourth-order Taylor polynomial $p_3(x)$ of $y = \cos(x)$ at $x = 0$ (which is actually $p_4(x)$) with the third-order polynomial $p(x) = a_0 + a_1x + a_2x^2 + a_3x^3$, which has the same values and derivative as $\cos(x)$ at the points $x = 0$ and $x = \pi$. This means that $p(x)$ satisfies these four conditions:

$$p(0) = 1 \quad p'(0) = 0 \quad p(\pi) = -1 \quad p'(\pi) = 0$$

Find the coefficients: a_0, a_1, a_2 , and a_3 of $p(x)$, and then plot all three functions together. Discuss the errors of the two different approximating polynomials.

2.3 TAYLOR'S THEOREM



Figure 2.5: Brook Taylor (1685-1731), English mathematician²

In the examples and problems of previous section we introduced Taylor polynomials $p_n(x)$ of a function $y = f(x)$ (appropriately differentiable) at $x = a$, and we saw that they appear to often serve as great tools for approximating the function near $x = a$. We also have seen that as the order n of the Taylor polynomial increases, so does its effectiveness in approximating $f(x)$. This of course needs to be reconciled with the fact that for larger values of n it is more work to form and compute $p_n(x)$. Additionally, the approximations seemed to improve, in general, when x gets closer to a . This latter observation seems plausible since $p_n(x)$ was constructed using only information about $f(x)$ at provides precise quantitative estimates for the error $x = a$. Taylor's theorem provides precise quantitative estimates for the error $|f(x) - p_n(x)|$, which can be very useful in choosing an appropriate order n so that $p_n(x)$ will give an approximation within the desired error bounds. We now present Taylor's theorem. For its proof we refer the reader to any decent calculus textbook.

THEOREM 2.1: (*Taylor's Theorem*) Suppose that for a positive integer n , a function $f(x)$ has the property that its $(n+1)$ st derivative is continuous on some interval I on the x -axis that contains the value $x = a$. Then the n th-order remainder $R_n(x) \equiv f(x) - p_n(x)$ resulting from approximating $f(x)$ by $p_n(x)$ is given by

$$R_n(x) = \frac{f^{(n+1)}(c)}{(n+1)!} (x-a)^{n+1} \quad (x \in I), \quad (5)$$

for some number c , lying between a and x (inclusive); see Figure 2.6.

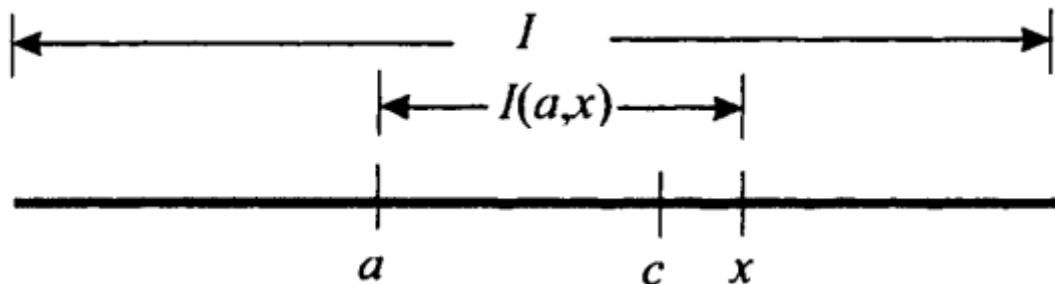


Figure 2.6: One possible arrangement of the three points relevant to Taylor's theorem.

REMARKS: (1) Note that like many such theorems in calculus, Taylor's theorem asserts the existence of the number c , but it does not tell us what it is.

²Taylor was born in Middlesex, England, and his parents were quite well-rounded people of society. His father was rather strict but instilled in Taylor a love for music and painting. His parents had him educated at home by private tutors until he entered St. John's College in Cambridge when he was 18. He graduated in 1709 after having written his first important mathematical paper a year earlier (this paper was published in 1714). He was elected rather early in his life (1712) to the Royal Society, the election being based more on his potential and perceived mathematical powers rather than on his published works, and two years later he was appointed to the prestigious post of Secretary of the Royal Society. In this same year he was appointed to an important committee that was to settle the issue of "who invented calculus" since both Newton and Leibniz claimed to be the founders. Between 1712 and 1724 Taylor published 13 important mathematical papers on a wide range of subjects including magnetism, logarithms, and capillary action. Taylor suffered some tragic personal events. His father objected to his marriage (claiming the bride's family was not a "good" one) and after the marriage Taylor and his father cut off all contact until 1723, when his wife died giving birth to what would have been Taylor's first child. Two years later he remarried (this time with his father's blessings) but the following year his new wife also died during childbirth, although this time his daughter survived.

(2) By its definition, (the absolute value of) $R_n(x)$ is just the actual error of the approximation of $f(x)$ by its n th-order Taylor polynomial $p_n(x)$, i.e.,

$$\text{error} = |f(x) - p_n(x)| = |R_n(x)|.$$

Since we do not know the exact value of c , we will not be able to calculate the error precisely; however, since we know that c must lie somewhere between a and x on I , call this interval $I(a, x)$, we can estimate that the unknown quantity $|f^{(n+1)}(c)|$ that appears in $R_n(x)$, can be no more than the maximum value of this $(n+1)$ st derivative function $|f^{(n+1)}(z)|$ as z runs through the interval $I(a, x)$. In mathematical symbols, this is expressed as:

$$|f^{(n+1)}(c)| \leq \max \left\{ |f^{(n+1)}(z)| : z \in I(a, x) \right\}. \quad (6)$$

EXAMPLE 2.5: Suppose that we wish to use Taylor polynomials (at $x = 0$) to approximate $e^{0.7}$ with an error less than 0.0001.

(a) Apply Taylor's theorem to find out what order n of a Taylor polynomial we could use for the approximation to guarantee the desired accuracy.

(b) Perform this approximation and then check that the actual error is less than maximum tolerated error.

SOLUTION: Part (a): Here $f(x) = e^x$, so, since $f(x)$ is its own derivative, we have $f^{(n)}(x) = e^x$ for any n , and so $f^{(n)}(0) = e^0 = 1$. From (4) (or (2) or (3) with $a = 0$), we can therefore write the general Taylor polynomial for $f(x)$ centered at $x = 0$ as

$$p_n(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!} = \sum_{k=0}^n \frac{x^k}{k!},$$

and from (5) (again with $a = 0$), $R_n(0.7) = \frac{e^c}{(n+1)!} (0.7)^{n+1}$.

How big can e^c be? For $f(x) = e^x$, this is just the question of finding out the right side of (6). In this case the answer is easy: Since c lies between 0 and 0.7, and e^x is an increasing function, the largest value that e^c can be is $e^{0.7}$. To honestly use Taylor's theorem here (since "we do not know" what $e^{0.7}$ is—that's what we are trying to approximate), let's use the conservative upper bound: $e^c \leq e^{0.7} \leq e^1 = e < 3$.

Now Taylor's theorem tells us that

$$\text{error} = |e^{0.7} - p_n(0.7)| = |R_n(0.7)| = \frac{e^c}{(n+1)!} (0.7)^{n+1}.$$

(Since all numbers on the right side are nonnegative, we are able to drop absolute value signs.) As was seen above, we can replace e^c with 3 in the right side above, to get

$$\text{something larger than the error} = \frac{3}{(n+1)!} (0.7)^{n+1}.$$

The rest of the plan is simple: We find an n large enough to make the "something larger than actual error" to be less than the desired accuracy 0.0001. Then it will certainly follow that the actual error will also be less than 0.0001. We can continue to compute $3(0.7)^{n+1}/(n+1)!$ until it gets smaller than 0.0001. Better yet, let's use a while loop to get MATLAB to do the work for us; this will also provide us with a good occasion to introduce the remaining relational operators that can be used in any while loops (or subsequent programming). (See Table 2.1.)

Table 2.1: Dictionary of MATLAB's relational operators.

Mathematical Relation	MATLAB Code
$>, <$	$>, <$
\geq, \leq	$>=, <=$
$= \neq$	$==, \sim =$

We have already used one of the first pair. For the last one, we reiterate again that the single equal sign in MATLAB is reserved for "assignment." Since it gets used much less often (in MATLAB codes), the ordinary equals in mathematics got stuck with the more cumbersome MATLAB notation.

Now, back to our problem. A simple way to figure out that smallest feasible value of n would be to run the following code:

```
>> n=1;
>>while 3*(0.7)^(n+1)/gamma(n+2) >= 0.0001
n=n+1;
end
```

This code has no output, but what it does is to keep making n bigger, one-by-one, until that "something larger than the actual error" gets less than 0.0001. The magic value of n that will work is now (by the way the while loop was constructed) simply the last stored value of n :

```
>> n → 6
```

Part (b): The desired approximation is now: $e^{0.7} \approx p_6(0.7) = \sum_{k=0}^6 \frac{(0.7)^k}{k!} \Big|_{x=0.7}$.

We can do the rest on MATLAB:

```

>>x=0.7;
>> n=0;
>>p6=0; % we initialize the sum for the Taylor polynomial p6
>>while n<=6;
p6=p6+x^n/gamma(n+1);
n=n+1;
end
>>p6
->2.0137 (approximation)
>> abs(p6-exp(0.7)) %we now check the actual error
->> 1.7889e-005 (this is less than 0.0001, as we knew from Taylor's theorem.)

```

EXERCISE FOR THE READER 2.2: If we use Taylor polynomials of $f(x) = \sqrt{x}$ centered at $x = 16$ to approximate $\sqrt{17} = f(16+1)$, what order Taylor polynomial should be used to ensure that the error of the approximation is less than 10^{-10} ? Perform this approximation and then look at the actual error. For any function $f(x)$, which has infinitely many derivatives at $x = a$, we can form the **Taylor series (centered) at $x = a$** :

$$f(a) + f'(a)(x-a) + \frac{f''(a)}{2}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \cdots + \frac{f^{(n)}(a)}{n!}(x-a)^n + \cdots = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(x-a)^k.$$

Comparing this with (2) and (3), the formulas for the n th Taylor polynomial $p_n(x)$ at $x = a$, we see that the Taylor series is just the infinite series whose first n terms are exactly the same as those of $p_n(x)$. The Taylor series may or may not converge, but if Taylor's theorem shows that the errors $|p_n(x) - f(x)|$ go to zero, then it will follow that the Taylor series above converges to $f(x)$. When $a = 0$ (the most common situation), the series is called the Maclaurin series (Figure 2.7).

It is useful to have some Maclaurin series for reference. Anytime we are able to figure out a formula for the general Taylor polynomial at $x = 0$, we can write down the corresponding Maclaurin series. The previous examples we have done yield the Maclaurin series for $\cos(x)$ and e^x . We list these in Table 2.2, as well as a few other examples whose derivations will be left to the exercises.

Table 2.1: Some Maclaurin series expansions.

Function	Maclaurin Series	
e^x	$1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^k}{k!} + \cdots$	(8)
$\cos(x)$	$1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \cdots + \frac{(-1)^k x^{2k}}{(2k)!} + \cdots$	(9)
$\sin(x)$	$x - \frac{x^3}{3!} + \frac{x^5}{5!} \cdots + \frac{(-1)^k x^{2k+1}}{(2k+1)!} + \cdots$	(10)
$\arctan(x)$	$x - \frac{x^3}{3} + \frac{x^5}{5} - \cdots + \frac{(-1)^{2k+1} x^{2k+1}}{2k+1} + \cdots$	(11)
$\frac{1}{1-x}$	$1 + x + x^2 + x^3 + \cdots + x^k + \cdots$	(12)

One very useful aspect of Maclaurin and Taylor series is that they can be formally combined to obtain new Maclaurin series by all of the algebraic operations (addition, subtraction, multiplication, and division) as well as with substitutions, derivatives, and integrations. These informally obtained expansions are proved to be legitimate in calculus books. The word "formal" here means that all of the above operations on an infinite series should be done as if it were a finite sum. This method is illustrated in the next example.

EXAMPLE 2.6: Using formal manipulations, obtain the Maclaurin series of the functions (a) $x \sin(x^2)$ and (b) $\ln(1+x^2)$.

SOLUTION: Part (a): In (10) simply replace x with x^2 and formally multiply by x (we use the symbol \sim to mean "has the Maclaurin series"):

$$\begin{aligned}
 x \sin(x^2) &\sim \\
 x \left(x^2 - \frac{(x^2)^3}{3!} + \frac{(x^2)^5}{5!} \cdots + \frac{(-1)^k (x^2)^{2k+1}}{(2k+1)!} \right) \\
 &= x^3 - \frac{x^7}{3!} + \frac{x^{11}}{5!} \cdots + \frac{(-1)^k x^{4k+3}}{(2k+1)!} + \cdots
 \end{aligned}$$

NOTE: This would have been a lot more work to do by using the definition and looking for patterns.

Part (b): We first formally integrate (12): $-\ln(1-x)$



Figure 2.7: Brook Taylor (1685-1731), English mathematician ³ $\sim \int (1 + x + x^2 + x^3 + \cdots + x^n + \cdots) dx = C + x + \frac{x^2}{2} + \frac{x^3}{3} + \cdots + \frac{x^{n+1}}{n+1} + \cdots$

By making $x = 0$, we get that the integration constant C equals zero. Now negate both sides and substitute x with $-x^2$ to obtain:

$$\begin{aligned}\ln(1+x^2) &\sim x^2 - \frac{(-x^2)^2}{2} - \frac{(-x^2)^3}{3} - \dots - \frac{(-x^2)^{n+1}}{n+1} - \dots \\ &\sim x^2 - \frac{x^4}{2} + \frac{x^6}{3} - \dots + \frac{(-1)^{k+1}x^{2k}}{k} + \dots\end{aligned}$$

Our next example involves another approximation using Taylor's theorem. Unlike the preceding approximation examples, this one involves an integral where it is impossible to find the antiderivative.

EXAMPLE 2.7: Use Taylor's theorem to evaluate the integral $\int_0^1 \sin(t^2) dt$ with an error $< 10^{-7}$.

SOLUTION: Let us denote the n th-order Taylor polynomial of $\sin(x)$ centered at $x = 0$ by $p_n(x)$. The formulas for each $p_n(x)$ are easily obtained from the Maclaurin series (10). We will estimate $\int_0^1 \sin(t^2) dt$ by $\int_0^1 p_n(t^2) dt$ for an appropriately large n . We can easily obtain an upper bound for the error of this approximation:

$$\text{error} = \left| \int_0^1 \sin(t^2) dt - \int_0^1 p_n(t^2) dt \right| \leq \int_0^1 |\sin(t^2) - p_n(t^2)| dt \leq \int_0^1 |R_n(t^2)| dt,$$

where $R_n(x)$ denotes Taylor's remainder. Since any derivative of $f(x) = \sin(x)$ is one of $\pm \sin(x)$ or $\pm \cos(x)$, it follows that for any $x, 0 \leq x \leq 1$, we have

$$|R_n(x)| = \left| \frac{f^{(n+1)}(c)x^{n+1}}{(n+1)!} \right| \leq \frac{1}{(n+1)!}.$$

Since in the above integrals, t is running from $t = 0$ to $t = 1$, we can substitute $x = t^2$ in this estimate for $|R_n(x)|$. We can use this and continue with the error estimate for the integral to arrive at:

$$\text{error} = \left| \int_0^1 \sin(t^2) dt - \int_0^1 p_n(t^2) dt \right| \leq \int_0^1 |R_n(t^2)| dt \leq \int_0^1 \frac{dt}{(n+1)!} = \frac{1}{(n+1)!}.$$

As in the previous example, let's now get MATLAB to do the rest of the work for us. We first need to determine how large n must be so that the right side above (and hence the actual error) will be less than 10^{-7} .

```
>>n = 1;
>>while 1/gamma(n+2) >= 10^(-7)
n=n+1;
end
>>n -> 10
```

So it will be enough to replace $\sin(t^2)$ 10 th-order Taylor polynomial evaluated at $t^2, p_n(t^2)$. The simplest way to see the general form of this polynomial (and its integral) will be to replace x with t^2 in the Maclaurin series (10) and then formally integrate it (this will result in the Maclaurin series for $\int_0^x \sin(t^2) dt$):

If we substitute $x = 0$, we see that the constant of integration C equals zero. Thus,

$$\int_0^x \sin(t^2) dt \sim \frac{x^3}{3} - \frac{x^7}{7 \cdot 3!} + \frac{x^{11}}{11 \cdot 5!} + \dots + \frac{(-1)^k x^{4k+3}}{(4k+3) \cdot (2k+1)!} + \dots$$

We point out that the formal manipulation here is not really necessary since we could have obtained from (10) an explicit formula for $p_{10}(t^2)$ and then directly integrated this function. In either case, integrating this function from $t = 0$ to $t = 1$ gives the partial sum of the last Maclaurin expansion (for $\int_0^x \sin(t^2) dt$) gotten by going up to the $k = 4$ term, since this corresponds to integrating up through the terms of $p_{10}(t^2)$.

³Maclaurin was born in a small village on the river Ruel in Scotland. His father, who was the village minister, died when Colin was only six weeks old. His mother wanted Colin and his brother John to have good education so she moved the family to Dumbarton, which had reputable schools. Colin's mother died when he was only nine years old and he subsequently was cared for by his uncle, also a minister. Colin began studies at Glasgow University when he was 11 years old (it was more common during these times in Scotland for bright youngsters to begin their university studies early—in fact universities competed for them). He graduated at age 14 when he defended an impressive thesis extending Sir Isaac Newton's theory on gravity. He then went on to divinity school with the intention of becoming a minister, but he soon ended this career path and became a chaired mathematics professor at the University of Aberdeen in 1717 at age 19. Two years later, Maclaurin met the illustrious Sir Isaac Newton and they became good friends. The latter was instrumental in Maclaurin's appointment in this same year as a Fellow of the Royal Society (the highest honor awarded to English academicians) and subsequently in 1725 being appointed to the faculty of the prestigious University of Edinburgh, where he remained for the rest of his career. Maclaurin wrote several important mathematical works, one of which was a joint work with the very famous Leonhard Euler and Daniel Bernoulli on the theory of tides, which was published in 1740 and won the three a coveted prize from the Académie des Sciences in Paris. Maclaurin was also known as an excellent and caring teacher. He married in 1733 and had seven children. He was also known for his kindness and had many friends, including members of the royalty. He was instrumental in his work at the Royal Society of Edinburgh, having it transformed from a purely medical society to a more comprehensive scientific society. During the 1745 invasion of the Jacobite army, Maclaurin was engaged in hard physical labor in the defense of Edinburgh. This work, coupled with an injury from falling off his horse, weakened him to such an extent that he died the following year.

```

>>p10=0 ;
>>k=0;
>>while k<=4
p10=p10+(-1)^k/(4*k+3)/gamma(2*k+2) ;
k = k+1;
end
>>format long
>> p10S
->p10 = 0.31026830280668

```

In summary, we have proved the approximation

$$\int_0^1 \sin(t^2) dt \approx 0.31026830280668.$$

Taylor's theorem has guaranteed that this is accurate with an error less than 10^{-7} .

EXERCISE FOR THE READER 2.3: Using formal manipulations, find the 10th order Taylor polynomial centered at $x = 0$ for each of the following functions: (a) $\sin(x^2) - \cos(x^3)$, (b) $\sin^2(x^2)$.

$$\begin{aligned} \sin(t^2) &\sim t^2 - \frac{t^6}{3!} + \frac{t^{10}}{5!} \cdots + \frac{(-1)^k t^{4k+2}}{(2k+1)!} + \cdots \Rightarrow \\ \int_0^x \sin(t^2) dt &\sim \int_0^x \left(t^2 - \frac{t^6}{3!} + \frac{t^{10}}{5!} + \cdots + \frac{(-1)^k t^{4k+2}}{(2k+1)!} + \cdots \right) dt \\ &\sim C + \frac{x^3}{3} - \frac{x^7}{7 \cdot 3!} + \frac{x^{11}}{11 \cdot 5!} + \cdots + \frac{(-1)^k x^{4k+3}}{(4k+3) \cdot (2k+1)!} + \cdots. \end{aligned}$$

EXERCISES 2.3:

- In each part below, we give a function $f(x)$, a center a to use for Taylor polynomials, a value for x , and a positive number ε that will stand for the error. The task will be to (carefully) use Taylor's theorem to find a value of the order n of a Taylor polynomial so that the error of the approximation $f(x) \approx p_n(x)$ is less than ε . Afterward, perform this approximation and check that the actual error is really less than what it was desired to be.
 - $f(x) = \sin(x)$, $a = 0$, $x = 0.2\text{rad}$, $\varepsilon = 0.0001$
 - $f(x) = \tan(x)$, $a = 0$, $x = 5^\circ$, $\varepsilon = 0.0001$
 - $f(x) = e^x$, $a = 0$, $x = -0.4$, $\varepsilon = 0.00001$
 - $f(x) = x^{1/3}$, $a = 27$, $x = 28$, $\varepsilon = 10^{-6}$
- Follow the directions in Exercise 1 for the following:
 - $f(x) = \cos(x)$, $a = \pi/2$, $x = 88^\circ$, $\varepsilon = 0.0001$
 - $f(x) = \arctan(x)$, $a = 0$, $x = 1/239$, $\varepsilon = 10^{-8}$
 - $f(x) = \ln x$, $a = 1$, $x = 3$, $\varepsilon = 0.00001$
 - $f(x) = \cos(x^2)$, $a = 0$, $x = 2.2$, $\varepsilon = 10^{-6}$
- Using only the Maclaurin series developed in this section, along with formal manipulations, obtain Maclaurin series for the following functions:
 - $x^2 \arctan(x)$
 - $\ln(1+x)$
 - $\frac{x^2+3x}{1-x}$
 - $\int_0^x \frac{1}{1-t^3} dt$
- Using only the Maclaurin series developed in this section, along with formal manipulations, obtain Maclaurin series for the following functions:
 - $\ln(1+x)$
 - $1/(1+x^2)$
 - $\arctan(x^2) - \sin(x)$
 - $\int_0^x \cos(t^5) dt$
- Find the Maclaurin series for $f(x) = \sqrt{1+x}$.
- Find the Maclaurin series for $f(x) = (1+x)^{1/3}$.
- (a) Use Taylor's theorem to approximate the integral $\int_0^1 \cos(t^5) dt$ with an error less than 10^{-8} . (First find a large enough order n for a Taylor polynomial that can be used from the theorem, then actually perform the approximation.)
 (b) How large would n have to be if we wanted the error to be less than 10^{-30} ?

8. The error function is given by the formula: $\operatorname{erf}(x) = (2/\sqrt{\pi}) \int_0^x e^{-t^2} dt$. It is used extensively in probability theory, but unfortunately the integral cannot be evaluated exactly. Use Taylor's theorem to approximate $\operatorname{erf}(2)$ with an error less than 10^{-6} .
9. Since $\tan(\pi/4) = 1$ we obtain $\pi = 4 \arctan(1)$. Using the Taylor series for the arctangent, this gives us a scheme to approximate π .
- (a) Using Taylor polynomials of $\arctan(x)$ centered at $x = 0$, how large an order n Taylor polynomial would we need to use in order for $4p_n(1)$ to approximate π with an error less than 10^{-12} ?
- (b) Perform this approximation.
- (c) How large an order n would we need for Taylor's theorem to guarantee that $4p_n(1)$ approximates π with an error less than 10^{-50} ?
- (d) There are more efficient ways to compute π . One of these dates back to the early 1700 s, when Scottish mathematician John Machin (1680-1751) developed the inverse trig identity:

$$\frac{\pi}{4} = \arctan\left(\frac{1}{5}\right) - \arctan\left(\frac{1}{239}\right) \quad (13)$$

to calculate the first 100 decimal places of π . There were no computers back then, so his work was all done by hand and it was important to do it in way where not so many terms needed to be computed. He did it by using Taylor polynomials to approximate each of the two arctangents on the right side of (13). What order Taylor polynomials would Machin have needed to use (according to Taylor's theorem) to attain his desired accuracy?

(e) Prove identity (13).

Suggestion: For part (d), use the trig identity:

$$\tan(A \pm B) = \frac{\tan A \pm \tan B}{1 \mp \tan A \tan B}$$

to calculate first $\tan(2 \tan^{-1} \frac{1}{5})$, then $\tan(4 \tan^{-1} \frac{1}{5})$, and finally $\tan(4 \tan^{-1} \frac{1}{5} - \tan^{-1} \frac{1}{239})$.

HISTORICAL ASIDE: Since antiquity, the problem of figuring out π to more and more decimals has challenged the mathematical world, and in more recent times the computer world as well. Such tasks can test the powers of computers as well as the methods used to compute them. Even in the 1970 s π had been calculated to over 1 million places, and this breakthrough was accomplished using an identity quite similar to (13). See [Bec-71] for an enlightening account of this very interesting history.

10. (Numerical Differentiation) (a) Use Taylor's theorem to establish the following *forward difference formula*:

$$f'(a) = \frac{f(a+h) - f(a)}{h} - \frac{h}{2} f''(c),$$

for some number c between a and $a+h$, provided that $f'(x)$ is continuous on $[a, a+h]$. This formula is often used as a means of numerically approximating the derivative $f'(a)$ by the simple difference quotient on the right; in this case the error of the approximation would be $|hf''(c)/2|$ and could be made arbitrarily small if we take h sufficiently small.

(b) With $f(x) = \sinh(x)$, and $a = 0$, how small would we need to take h for the approximation in part (a) to have error less than 10^{-5} ? Do this by first estimating the error, and then (using your value of h) check the actual error using MATLAB. Repeat with an error goal of 10^{-10} .

(c) Use Taylor's theorem to establish the following *central difference formula*:

$$f''(a) = \frac{f(a+h) - 2f(a) + f(a-h)}{h^2} - \frac{h^2}{12} f^{(4)}(c)$$

for some number c between $a-h$ and $a+h$, provided that $f^{(4)}(x)$ is continuous on $[a-h, a+h]$. This formula is often used as a means of numerically approximating the derivative $f''(a)$ by the simple difference quotient on the right; in this case the error of the approximation would be $|h^2 f^{(4)}(c)/12|$ and could be made arbitrarily small if we take h sufficiently small.

(d) Repeat part (b) for the approximation of part (c). Why do the approximations of part (c) seem more efficient, in that they do not require as small an h to achieve the same accuracy?

(e) Can you derive (and prove using Taylor's theorem) an approximation for $f'(x)$ whose error is proportional to h^2 ?

⁴Of course, since MATLAB's compiler keeps track of only about 15 digits, such an accurate approximation could not be done without the help of the Symbolic Toolbox (see Appendix A).

Chapter 3

Introduction to M-Files

3.1 WHAT ARE M-FILES?

Up to now, all of our interactions with MATLAB have been directly through the command window. As we begin to work with more complicated algorithms, it will be preferable to develop standalone programs that can be separately stored and saved into files that can always be called on (by their name) in any MATLAB session. The vehicle for storing such a program in MATLAB is the so-called Mfile. M-files are programs that are plain-text (ASCII) files written with any word processing program (e.g., Notepad or MS Word) and are called M-files because they will always be stored with the extension `< filename > .m`¹ As you begin to use MATLAB more seriously, you will start to amass your own library of M-files (some of these you will have written and others you may have gotten from other sources such as the Internet) and you will need to store them in various places (e.g., certain folders on your own computer, or also on your portable disk for when you do work on another computer). If you wish to make use of (i.e., "call on") some of your M-files during a particular MATLAB session from the command window, you will need to make sure that MATLAB knows where to look for your M-files. This is done by including all possible directories where you have stored M-files in MATLAB's path.²

EXAMPLE 3.1: Here is a simple script which assumes that numbers x_0, y_0 , and $r > 0$ have been stored in the workspace (before the script is invoked) and that will graph the circle with center (x_0, y_0) and radius r .

```
t= 0:.001:2*pi;
x = x0 + r * cos(t);
y = y0 + r *sin(t);
plot(x,y)
axis('equal')
```

If the above lines are simply typed as is into a text file and saved as, say, `circdrw.m` into some directory in the path, then at any time later on, if we wish to get MATLAB to draw a circle of radius 2 and center $(5, -2)$, we could simply enter the following in the command window:

```
>> r=2; x0=5; y0= -2;
>> circdrw
```

and voilà! the graphic window pops up with the circle we desired. Please remember that any variables created in a script are global variables, i.e., they will enter the current workspace when the script is invoked in the command window. One must be careful of this since the script may have been written a long time ago and when it is run the lines of the script are not displayed (only executed).

Function M-files are stored in the same way as script M-files but are quite different in the way they work. Function M-files accept any number of input variables and can output any number of output variables (or none at all). The variables introduced in a function M-file are local variables, meaning that they do not remain in the workspace after a function M-file is called in a MATLAB session. Also, the first line of a function M-file must be in the following format:

```
function [<output variables>] = <function_name>(<input variables>)
```

¹It is recommended that you use the default MATLAB M-file editor gotten from the "File" menu (on the top left of the command window) and selecting "New" -> "M-File." This editor is designed precisely for writing M-files and contains many features that are helpful in formatting and debugging. Some popular word processing programs (notably MS Word) will automatically attach a certain extension (e.g., ".doc") at the end of any filename you save a document as and it can be difficult to prevent such things. On a Windows/DOS-based PC, one way to change an M-file that you have created in this way to have the needed ".m" extension is to open the DOS command window, change to the directory you have stored your M-file in, and rename the file using the DOS command `ren < filename > .m.doc < filename > .m` (the format is: `ren < oldfilename > .oldextension < newfilename > .newextension`).

²Upon installation, MATLAB sets up the path to include a folder "Work" in its directory, which is the default location for storing M-files. To add other directories to your path, simply select "Set Path" from the "File Menu" and add on the desired path. If you are using a networked computer, you may need to consult with the system administrator on this.

Another important format issue is that the `< function_name >` (which you are free to choose) should coincide exactly with the filename under which you save the function M-file.

EXAMPLE 3.2: We create a function M-file that will do essentially the same thing as the script in the preceding example. There will be three input variables: We will make the first two be the coordinates of the center (x_0, y_0) of the circle that we wish MATLAB to draw, and the third be the radius. Since there will be no output variables here (only a graphic), our function M-file will look like this:

```
function [ ] = circdrwf(x0,y0,r)
    t=0:.001:2*pi;
    x=x0+r*cos(t);
    y=y0+r*sin(t);
    plot(x,y)
    axis('equal')
```

In particular, the word *function* must be in lowercase. We then save this Mfile as *circdrwf.m* in an appropriate directory in the path. Notice we gave this M-file a different name than the one in Example 3.1 (so they may lead a peaceful coexistence if we save them to the same directory). Once this function M-file has been stored we can call on it in any MATLAB session to draw the circle of center $(5, -2)$ and radius 2 by simply entering

```
>> circdrwf(5, -2, 2)
```

We reiterate that, unlike with the script of the first example, after we use a function M-file, none of the variables created in the file will remain in the workspace. As you gain more experience with MATLAB you will be writing a lot of function M-files (and probably very soon find them more useful than script Mfiles). They are analogous to "functions" in the C-language, "procedures" in PASCAL, and "programs" or "subroutines" in FORTRAN. The `< filenames >` of M-files can be up to 19 characters long (older versions of MATLAB accepted only length up to 8 characters), and the first character must be a letter. The remaining characters can be letters, digits, or underscore (`_`).

EXERCISE FOR THE READER 3.1: Write a MATLAB script, call it *listp2*, that assumes that a positive integer has been stored as n and that will find and output all powers of 2 that are less than or equal to n . Store this script as an Mfile *listp2.m* somewhere in MATLAB's path and then run the script for each of these values of n : $n = 5$, $n = 264$, and $n = 2917$.

EXERCISE FOR THE READER 3.2: Write a function M-file, call it *fact*, having one input variable—a nonnegative integer n , and the output will be the factorial of n : $n!$. Write this program from scratch (using a while loop) without using a built-in function like *gamma*. Store this M-file and then run the following evaluations: *fact*(4), *fact*(10), *fact*(0).

Since MATLAB has numerous built-in functions it is often advisable to check first if a proposed M-file name that you are contemplating is already in use. Let's say you are thinking of naming a function M-file you have just written with the name *det.m*. To check first with MATLAB to see if the name is already in use you can type:

```
>>exist ('det') %possible outputs are 0, 1, 2, 3, 4, 5, 6, 7, 8
->5
```

The output 5 means (as does any positive integer) *det* is already in use. Let's try again (with a trick often seen on vanity license plates):

```
>>exist ('det1')
->0
```

The output zero means the filename *det1* is not yet spoken for so we can safely assign this filename to our new M-file.

EXERCISES 3.1:

- (a) Write a MATLAB function M-file, call it *rectdrw* $f(1, w)$, that has two input variables: 1, the length and w , the width, and no output variables, but will produce a graphic of a rectangle with horizontal length = 1 and vertical width = w . Arrange it so that the rectangle sits well inside the graphic window and so that the axes are equally scaled.
- (a) Write a function M-file, call it *segdrwf* (x, y) , that has two input vectors $x = [x_1 \ x_2 \ \cdots \ x_n]$ and $y = [y_1 \ y_2 \ \cdots \ y_n]$ of the same size and no output variables, but will produce the graphic gotten by connecting the points $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$. You might wish to make use of the MATLAB built-in function *size* (A) that, for an input matrix A, will output its size.
 - Run this program with the inputs $x = [1 \ 3 \ 5 \ 7 \ 9 \ 1]$ and $y = [1 \ 4 \ 1 \ 4 \ 8 \ 1]$.
 - Determine two vectors x and y so that *segdrwf* (x, y) will produce an equilateral triangle.

3. Redo Exercise 1, creating a script M-file called rectdrw rather than a function M-file.
4. Redo Exercise 2, creating a script M-file called segdrw rather than a function M-file.
5. (*Finance*) Write a function M-file, call it compint $f(r, P, F)$, that has three input variables: r , the annual interest rate, P , the principal, and F , the future goal. Here is what the function should do: It assumes that we deposit P dollars (assumed positive) into an interest-bearing account that pays 100% interest per year compounded annually. The investment goal is F dollars (F is assumed larger than P , otherwise the goal is reached automatically as soon as the account is opened). The output will be one variable consisting of the number of years it takes for the account balance to first equal or exceed F . Store this M-file and run the following: comintf (0.06, 1000, 100000), comintf (0.085, 1000, 100000), comintf(0.10, 1000, 1000000), and comintf (0.05, 100, 1000000).

Note: The formula for the account balance after t years is P dollars is invested at 100% compounded annually is $P(1 + r)^t$.

6. (*Finance*) Write a function M-file, call it loanperf (r, L, PMT), that has three input variables: r , the annual interest rate, L , the loan amount, and PMT , the monthly payment. There will be one output variable n , the number of months needed to pay off a loan of L dollars made at an annual interest rate of 100% (on the unpaid balance) where at the end of each month a payment of PMT dollars is made. (Of course L and PMT are assumed positive.) You will need to use a while loop construction as in Example 1.I (Sec. 1.3). After storing this M-file, run the following: loanperf (.0799, 15000, 500), loanperf (.019, 15000, 500), loan (0.99, 22000, 450). What could cause the program loanperf (r, L, PMT) to go into an infinite loop? In the next chapter we will show, among other things, ways to safeguard programs from getting into infinite loops.
7. Redo Exercise 6 writing a script M-file (which assumes the relevant input variables have been assigned) rather than a function M-file.
8. Write a function M-file, call it oddfact (n), that inputs a positive integer n and will output the product of all odd positive integers that are less than or equal to n . So, for example, oddfact (8) will be $1 \cdot 3 \cdot 5 \cdot 7 = 105$. Store it and then run this function for the following values: oddfact (5), oddfact (22), oddfact (29). Use MATLAB to find the first value of n for which oddfact (n) exceeds or equals 1 million, and then 5 trillion.
9. Redo Exercise 5 writing a script M-file (which assumes the relevant input variables have been assigned) rather than a function M-file.
10. Write a function M-file, call it evenfact (n), that inputs a positive integer n and will output the product of all even positive integers that are less than or equal to n . So, for example, evenfact (8) will be $2 \cdot 4 \cdot 6 \cdot 8 = 384$. Store it and then run this function for the following values: evenfact (5), evenfact (22), evenfact (29). Get MATLAB to find the first value of n for which even fact(n) exceeds or equals 1 million, and then 5 trillion. Can you write this M-file without using a while loop, using instead some of MATLAB's built-in functions?
11. Use the error estimate from Example 2.5 (Sec. 2.3) to write a function M-file called expcal (x , err) that does the following: The input variable x is any real number and the other input variable err is any positive number. The output will be an approximation of e^x by a Taylor polynomial $p_n(x)$ based at $x = 0$, where n is the first nonnegative integer such that the error estimate based on Taylor's theorem that was obtained in Example 2.5 gives a guaranteed error less than err . There should be two output variables, n , the order of the Taylor polynomial used, and $y = p_n(x)$ = the approximation. Run this function with the following input data: (2, 0.001), $(-6, 10^{-12})$, (15, 0.000001), $(-30, 10^{-25})$. For each of these y -outputs, check with MATLAB's built-in function exp to see if the actual errors are as desired. Is it possible for this program to ever enter into an infinite loop?
12. Write a function M-file, called coscal(x, err), that does exactly what the function in Exercise 11 does except now for the function $y = \cos(x)$. You will need to obtain a Taylor's theorem estimate for the (actual) error $|\cos(x) - p_n(x)|$. Run this function with the following input data: (0.5, 0.0000001), $(-2, 0.0001)$, $(20^\circ, 10^{-9})$, and $(360020^\circ, 10^{-9})$ (for the last two you will need to convert the inputs to radians). For each of these y -outputs, check with MATLAB's built-in function cos(x) to see if the actual errors are as desired. Is it possible for this program to ever enter into an infinite loop? Although $\cos(360020^\circ) = \cos(20^\circ)$, the outputs you get will be different; explain this discrepancy.

3.2 CREATING AN M-FILE FOR A MATHEMATICAL FUNCTION

Function M-files can be easily created to store (complicated) mathematical functions that need to be used repeatedly. Another way to store mathematical functions without formally saving them as M-files is to create them as "in-line objects." Unlike M-ftles, in-line objects are stored only as variables in the current workspace. The following example will illustrate such an M-file construction; inline objects will be introduced in Chapter 6.

EXAMPLE 3.3: Write a function M-file, with filename `bumpy.m`, that will store the function given by the following formula:

$$y = \frac{1}{4\pi} \left[\frac{1}{(x-2)^2 + 1} + \frac{1}{(x+0.5)^4 + 32} + \frac{1}{(x+1)^2 + 2} \right].$$

Once this is done, call on this newly created M-file to evaluate y at $x = 3$ and to sketch a graph of the function from $x = -3$ to $x = 3$.

SOLUTION: After the first "function definition line," there will be only one other line required: the definition of the function above written in MATLAB's language. Just like in a command window, anything we type after the percent symbol (%) is considered as a comment and will be ignored by MATLAB's processor. Comment lines that are put in immediately following the function definition line are, however, somewhat more special. Once a function has been stored (somewhere in MATLAB's path), and you type `help <function_name>` on the command window, MATLAB displays any comments that you inserted in the M-file after the function definition line. Here is one possibility of a function M-file for the above function:

```
function y = bumpy(x)
% our first function M-file
% x could be a vector
% created by <yourname> on <date>
y=1/(4*pi) * (1/((x-2).^2+1) + 1./((x+0.5).^4+32) + 1./((x+1).^2+2));
```

Some comments are in order. First, notice that there is only one output variable, y , but we have not enclosed it in square brackets. This is possible when there is only one output variable (it would still have been okay to type `function [y] = bumpy(x)`). In the last line where we typed the definition of y (the output variable), notice that we put a semicolon at the end. This will suppress any duplicate outputs since a function M-file is automatically set up to print the output variable when evaluated. Also, please look carefully at the placement of parentheses and (especially) the dots when we wrote down the formula. If x is just a number, the dots are not needed, but often we will need to create plots of functions and x will need to be a vector. The placement of dots was explained in Chapter 1.

The above function M-file should now be saved with the name `bumpy.m` with the same filename appearing (without the extension) in the function definition line into some directory contained in MATLAB's path (as explained in the previous section). This being done, we can use it just like any of MATLAB's built-in functions, like `cos`. We now proceed to perform the indicated tasks.

```
>>bumpy(3)
-> 0.0446
>>y % Remember all the variables in a MATLAB M-file are local only.
-> Undefined function or variable y.
>>x=-3:.01:3 ;
>> plot(x,bumpy(x))
```

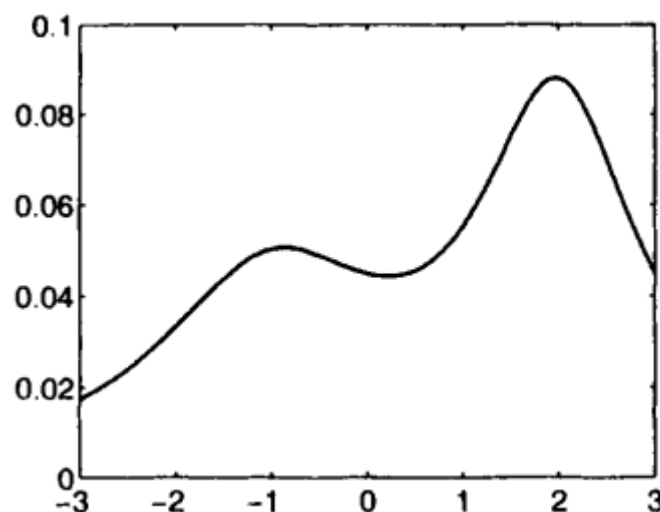


Figure 3.1: A graph of the function $y = \text{bumpy}(x)$ of Example 3.3.

EXAMPLE 3.4: For the function $\text{bumpy}(x)$ of the previous example, find "good" approximations to the following:

(a) $\int_{-3}^3 \text{bumpy}(x) dx$

(b) The maximum and minimum values of $y = \text{bumpy}(x)$ on the interval $-1.2 \leq x \leq 1$ (i.e., the height of the left peak and that of the valley) and the corresponding x -coordinates of where these extreme values occur.

(c) A solution of the equation $\text{bumpy}(x) = 0.08$ on the interval $0 \leq x \leq 2$ (which can be seen to exist by examination of bumpy 's graph in Figure 3.1).

SOLUTION: Part (a): The relevant built-in function in MATLAB for doing definite integrals is `quad`, which is an abbreviation for quadrature, a synonym for integration.³ The syntax is as follows:

<code>quad('function', a, b, tol) →</code>	Approximates the integral $\int_a^b \text{function}(x) dx$ with the goal of the error being less than <code>tol</code> .
--	--

The function must be stored as an M-file in MATLAB's path or the exact name of a built-in function, and the name must be enclosed in 'single quotes'.⁴ If the whole last argument `tol` is omitted (along with the comma that precedes it), a maximum error goal of 10^{-3} is assumed. If this command is run and you just an answer (without any additional warnings or error messages), you can safely assume that the approximation is accurate within the sought-after tolerance.

```
>> quad('bumpy', -3, 3)    %\rightarrow$ 0.3061
>> format long
>> ans    -> 0.30608471060690
```

As explained above, this answer should be accurate to at least three decimals. It is actually better than this since if we redo the calculation with a goal of six digits of accuracy, we obtain:

```
>> quad('bumpy', -3, 3, 10^(-6))
-> 0.30608514875582
```

This latter answer, which is accurate to at least six digits, agrees with the first answer (after rounding) to six digits, so the first answer is already quite accurate. There are limits to how accurate an answer we can get in this way. First of all, MATLAB works with only about 15 or so digits, so we cannot hope for an answer more accurate than this. But roundoff and other errors can occur in large-scale calculations and these can put even further restrictions on the possible accuracy, depending on the problem. We will address this issue in more detail in later depending on the problem. We will address this issue in more detail in later chapters. For many practical purposes and applications the `quad` function and the there will be no need to write new MATLAB M-files to perform such tasks.

Part (b): To (approximately) solve the calculus problem of finding the minimum value of a function on a specified interval, the relevant MATLAB built-in function is `fminbnd` and the syntax is as follows:

<code>fminbnd('function', a, b, optimset('TolX', tol)) →</code>	Approximates the x -coordinate of the minimum value of $\text{function}(x)$ on $[a, b]$ with a goal of the error being $< \text{tol}$.
---	---

The usage and syntax comments for `quad` apply here as well. In particular, if the whole last argument `optimset('TolX', tol)` is omitted (along with the comma that precedes it), a maximum error goal of 10^{-3} is assumed. Note the syntax for changing the default tolerance goal is a bit different than for `quad`. This is due to the fact that `fminbnd` has more options and is capable of doing a lot more than we will have occasion to use it for in this text. For more information, enter `help optimset`.

```
>> xmin=fminbnd('bumpy', -1.2, 1)    %We first find the x-coordinate of
>> % the valley with a three digit accuracy (at least)
>> 0.21142776202687
>> xmin=fminbnd('bumpy', -1.2, 1, optimset('TolX', 1e-6))
>> %Next let's go for 6 digits of accuracy.
>> 0.21143721018793 (=x-coordinate of valley)
```

The corresponding y -coordinate (height of the valley) is now gotten by evaluating `bumpy(x)` at `xmin`.

```
>> ymin = bumpy(xmin) -> 0.04436776267211 (= y-coordinate of valley)
```

³MATLAB has another integrator, `quadl`, that gives more accurate results for well-behaved integrands. For most purposes, though, `quad` is quite satisfactory and versatile as a general quadrature tool.

⁴An alternative syntax (that avoids the single quotes) for this and other functions that call on M-files is `quad(@function, a, b, tol)`. Another way to create mathematical functions is to create them as so-called inline functions which are stored only in the workspace (as opposed to M-files) and get deleted when the MATLAB session is ended. Inline functions will be introduced in Chapter 6

Since we know that the x -coordinate is accurate to six decimals, a natural question is how accurate is the corresponding y -coordinate that we just obtained? One obvious thing to try would be to estimate this error by plotting $\text{bumpy}(x)$ on the interval $x_{\min} - 10^{-6} \leq x \leq x_{\min} + 10^{-6}$ and then seeing how much the y -coordinates vary on this plot. This maximum variation will be an upper bound for the difference of y_{\max} and the actual value of the y -coordinate for the valley. When we try and plot $\text{bumpy}(x)$ on this interval as follows we get the following warning message:

```
>>x=(xmin-10^(-6)): 10^(-9):(xmin +10^(-6)) ;
>> plot(x,bumpy(x))
->Warning: Requested axes limit range too small; rendering with minimum range allowed by
machine precision.
```

Also, the corresponding plot (which we do not bother reproducing) looks like that of a horizontal line, but the y -tick marks are all marked with the same number (.0444) and similarly for the x -tick marks. This shows that MATLAB's plotting precision works only up to a rather small number of significant digits. Instead we can look at the vector $\text{bumpy}(x)$ with x still stored as the vector above, and look at the difference of the maximum less the minimum.

$\max(v), \min(v) \rightarrow$	For a vector v , these MATLAB commands will return the maximum entry and the minimum entry; e.g.: If $v = [28 \ 50]$ then $\max(v) \rightarrow 50$, and $\min(v) \rightarrow 28$.
--------------------------------	--

```
>> max(bumpy(x)) - min(bumpy(x)) -> 1.785377401475330e-014
```

What this means is that, although x_{\min} was guaranteed only to be accurate to six decimals, the corresponding y -coordinate seems to be accurate to MATLAB precision, which is about 15 digits!

EXERCISE FOR THE READER 3.3: Explain the possibility of such a huge discrepancy between the guaranteed accuracy of x_{\min} (to the actual x -value of where the bottom of the valley occurs) being 10^{-6} and the incredibly smaller value 10^{-14} of the apparent accuracy of the corresponding $y_{\min} = \text{bumpy}(x_{\min})$. Make sure to use some calculus in your explanation.

EXERCISE FOR THE READER 3.4: Explain why the above vector argument does not necessarily guarantee that the error of y_{\min} as an approximation to the actual y -coordinate of the valley is less than $1.8 \times 10^{-14} \dots$

We turn now to the sought-after maximum. Since there is no built-in MATLAB function (analogous to `fminbnd`) for finding maximums, we must make do with what we have. We can use `fminbnd` to locate maximums of functions as soon as we make the following observation: The maximum of a function $f(x)$ on an interval I , if it exists, will occur at the same x -value as the minimum value of the negative function $-f(x)$ on I . This is easy to see; just note that the graph of $y = -f(x)$ is obtained by the graph of $y = f(x)$ by turning the latter graph upside-down (more precisely, reflect it over the x -axis), and when a graph is turned upside-down, its peaks become valleys and its valleys become peaks. Let's initially go for six digits of accuracy:

```
>> xmax = fminbnd ('-bumpy(x)', -1.2, 1, optimset('TolX',le-6))
-> -0.86141835836638 (= x-coordinate of left peak)
```

The corresponding y -coordinate is now:

```
>> bumpy(xmax) -> 0.05055706241866 (= y-coordinate of left peak)
```

Part (c): One way to start would be to simultaneously graph $y = \text{bumpy}(x)$ together with the constant function $y = 0.08$ and continue to zoom in on the intersection point. As explained above, though, this graphical approach will limit the attainable accuracy to three or four digits. We must find the root (less than 2) of the equation $\text{bumpy}(x) = 0.08$. This is equivalent to finding a zero of the standard form equation $\text{bumpy}(x) - 0.08 = 0$. The relevant MATLAB function is `fzero` and its usage is as follows:

$\text{fzero}(\text{function}', a) \rightarrow$	Finds a zero of function (x) near the value $x = a$ (if one exists). Goal is machine precision (about 15 digits).
---	---

```
>>fzero('bumpy(x)-0.08', 1.5)
>>Zero found in the interval: [1.38,1.62].
>>1.61904252091472 (=desired solution)
>> bumpy(ans) %as a check, let's see if this value of x does what we
>>% want it to do.
-> 0.080000000000000 %Not bad!
```

EXERCISE FOR THE READER 3.5: Write a function M-file with filename wiggly.m that will store the following function:

$$y = \sin \left(\exp \left[\frac{1}{(x^2 + .5)^2} \right] \right) \sin(x).$$

- Plot this function from $x = -2$ through $x = 2$.
- Integrate this function from $x = 0$ to $x = 2$ (use 10^{-5} as your accuracy goal).
- Approximate the x -coordinates of both the smallest positive local minimum (valley) and the smallest positive local maximum (peak) from $x = -2$ through $x = 2$.
- Approximate the smallest positive solution of wiggly ($x = x/2$) (use 10^{-5} as your accuracy goal).

EXERCISES 3.2:

- Create a MATLAB function M-file for the function $y = f(x) = \exp(\sin[\pi/(x+0.001)^2]) + (x-1)^2$ and then plot this function on the interval $0 \leq x \leq 3$. Do it first using 10 plotting points and then using 50 plotting points and finally using 500 points.
 - Compute the corresponding integral $\int_1^3 f(x)dx$.
 - What is the minimum value (y -coordinate) of $f(x)$ on the interval $[1, 10]$? Make sure your answer is accurate to within the nearest $1/10,000$ th.
- Create a MATLAB function M-file for the function $y = f(x) = \frac{1}{x} \sin(x^2) + \frac{x^2}{50}$ and then plot this function on the interval $0 \leq x \leq 10$. Do it first using 200 plotting points and then using 5000 plotting points.
 - Compute the corresponding integral $\int_1^{10} f(x)dx$.
 - What is the minimum value (y -coordinate) of $f(x)$ on the interval $[1, 10]$? Make sure your answer is accurate to within the nearest $1/10,000$ th. Find also the corresponding x -coordinate with the same accuracy.
- Evaluate the integral $\int_0^1 \sin(t^2) dt$ (with an accuracy goal of 10^{-7}) and compare this with the answer obtained in Example 2.7 (Sec. 2.3).
- Find the smallest positive solution of the equation $\tan(x) = x$ using an accuracy goal of 10^{-12} .
 - Using calculus, obtain a bound for the actual error.
- Find all zeros of the polynomial $x^3 + 6x^2 - 14x + 5$.

NOTE: We remind the reader about some facts on polynomials. A polynomial $p(x)$ of degree n can have at most n roots (that are the x -intercepts of the graph $y = p(x)$). If $x = r$ is a root and if the derivative $p'(r)$ is not zero, then we say $x = r$ is a root of multiplicity 1. If $p(r) = p'(r) = 0$ but $p''(r) \neq 0$, then we say the root $x = r$ has multiplicity 2. In general we say $x = r$ is a root of $p(x)$ of multiplicity a , if all of the first $a - 1$ derivatives equal zero:

$$p(r) = p'(r) = p''(r) = \dots p^{(a-1)}(r) = 0 \text{ but } p^{(a)}(r) \neq 0$$

Algebraically $x = r$ is a root of multiplicity a means that we can factor $p(x)$ as $(x - r)^a g(x)$ where $g(x)$ is a polynomial of degree $n - a$. It follows that if we add up all of the multiplicities of all of the roots of a polynomial, we get the degree of the polynomial. This information is useful in finding all roots of a polynomial.

- Find all zeros of the polynomial $2x^4 - 16x^3 - 2x^2 + 25$. For each one, attempt to ascertain its multiplicity.
- Find all zeros of the polynomial

$$x^6 - \frac{25}{4}x^3 + \frac{4369}{64}x^4 + \frac{8325}{32}x^3 + \frac{13655}{8}x^2 - \frac{325}{32}x + \frac{21125}{8}.$$

For each one, attempt to ascertain its multiplicity.

- Find all zeros of the polynomial

$$x^8 + \frac{136}{5}x^7 + 210x^6 - \frac{165}{5}x^5 - 4094x^4 + \frac{4528}{5}x^3 + 17232x^2 + 320x + 5600$$

For each one, attempt to ascertain its multiplicity.

- Check that the value $x = 2$ is a zero of both of these polynomials:

$$P(x) = x^8 - 2x^7 + 6x^5 - 12x^4 + 2x^2 - 8$$

$$Q(x) = x^8 - 8x^7 + 28x^6 - 61x^5 + 95x^4 - 112x^3 + 136x^2 - 176x + 112$$

Next, use `fzero` to seek out this root for each polynomial using $a = 1$ (as a number near the root) and with accuracy goal 10^{-12} . Compare the outputs and try to explain why the approximation seemed to go better for one of these polynomials than for the other one.

Chapter 4

Programming in MATLAB

4.1 SOME BASIC LOGIC

Computers and their programs are designed to function very logically so that they always proceed by a well-defined set of rules. In order to write effective programs, we must first learn these rules so we can understand what a computer or MATLAB will do in different situations that may arise throughout the course of executing a program. The rules are set forth in the formal science of logic. Logic is actually an entire discipline that is considered to be part of both of the larger subjects of philosophy and mathematics. Thus there are whole courses (and even doctoral programs) in logic and any student who wishes to become adept in programming would do well to learn as much as possible about logic. Here in this introduction, we will touch only the surface of this subject, with the hope of supplying enough elements to give the student a working knowledge that will be useful in understanding and writing programs.

The basic element in logic is a **statement**, which is any declarative sentence or mathematical equation, inequality, etc. that has a **truth value** of either **true** or **false**.

EXAMPLE 4.1: For each of the English or mathematical expressions below, indicate which are statements, and for those that are, decide (if possible) the truth value.

(a) Al Gore was Bill Clinton's Vice President

(b) $3 < 2$

(c) $x + 3 = 5$

(d) If $x = 6$ then $x^2 > 4x$.

SOLUTION: All but (c) are statements. In (c), depending on the value of the variable x , the equation could be either true (if $x = 2$) or false (if $x =$ any other number). The truth values of the other statements are as follows: (a) true, (b) false, and (d) true.

If you enter any mathematical relation (with one of the relational operators from Table 2.1), MATLAB will tell you if the statement is true or false in the following fashion:

Truth Value	MATLAB Code
True	1 (as output) Any nonzero number (as input)
False	0 (as input and output)

```
>>3<2
-> 0 (MATLAB is telling us the statement is false.)
>> x=6; x^2>4*x$
->1 (MATLAB is telling us the statement is true.)
```

Logical statements can be combined into more complicated compound statements using **logical operators**. We introduce the four basic logical operators in Table 4.1, along with their approximate English translations, MATLAB code symbols, and precise meanings.

TABLE 4.1: The basic logical operators. In the meaning explanation, it is assumed the p and q represent statements whose truth values are known.

Name operator	English Approximation	MATLAB Code	Meaning
Negation	not p	$\sim p$	$\sim p$ is true if p is false, and false if p is true
Conjunction	p and q	$p \& q$	$p \& q$ is true if both p and q are true, otherwise it's false.
Disjunction	p or q	$p q$	$p q$ is true in all cases except if p and q are both false,
Exclusive Disjunction	p or q (but not both) ¹	$\text{xor}(p, q)$	$\text{xor}(p, q)$ is true if exactly one of p or q is true. If p and q are both true or both false then $\text{xor}(p, q)$ is false.

EXAMPLE 4.2: Determine the truth value of each of the following compound statements.

- (a) San Francisco is the capital of California and Egypt is in Africa.
- (b) San Francisco is the capital of California or Egypt is in Africa.
- (c) San Francisco is not the capital of California.
- (d) not $(2 > -4)$
- (e) letting $x = 2, z = 6$, and $y = -4$: $x^2 + y^2 > z^2/2$ or $zy < x$
- (f) letting $x = 2, z = 6$, and $y = -4$: $x^2 + y^2 > z^2/2$ or $zy < x$ (but not both)

SOLUTION: To abbreviate parts (a) through (c) we introduce the symbols:

p = San Francisco is the capital of California.

q = Egypt is in Africa.

From basic geography, Sacramento is California's capital so p is false, and q is certainly true. Statements (a) through (c) can be written as: p and q , p or q , not p , respectively. From what was summarized in Table 4.1 we now see (a) is false, (b) is true, and (c) is true.

For part (d), since $2 > -4$ is true, the negation not $(2 > -4)$ must be false.

For parts (e) and (f), we note that substituting the values of x, y , and z the statements become:

(e) $20 > 18$ or $-24 < 2$ i.e., true or true, so true

(f) $20 > 18$ or $-24 < 2$ (but not both), i.e., true or true (but not both), so false.

MATLAB does not know geography but it could have certainly helped us with the mathematical questions (d) through (f) above. Here is how one could do these on MATLAB:

```
>> ~(2>-4)
-> 0 (=false)
>> x=2; z = 6; y=-4; (x^2+y^2 > z^2/2) | (z*y < x)
->1 (=true)
>>x=2; z=6; y=-4; xor(x^2+y^2 > z^2/2, z*x<x)
-> 0 (=false)
```

EXERCISES 4.1:

- For each of the English or mathematical expressions below, indicate which are statements, and for those that are statements, decide (if possible) the truth value.
 - (a) Ulysses Grant served as president of the United States.
 - (b) Who was Charlie Chaplin?
 - (c) With $x = 2$ and $y = 3$ we have $\sqrt{x^2 + y^2} = x + y$.
 - (d) What is the population of the United States?
- For each of the English or mathematical statements below, determine the truth value.
 - (a) George Harrison was a member of the Rolling Stones.
 - (b) Canada borders the United States or France is in the Pacific Ocean.
 - (c) With $x = 2$ and $y = 3$ we have $x^x > y$ or $x^y > y^x$.
 - (d) With $x = 2$ and $y = 3$ we have $x^x > y$ or $x^y > y^x$ (but not both).
- Assume that we are in a MATLAB session in which the following variables have been stored: $x = 6, y = 12, z = -4$. What outputs would the following MATLAB commands produce? Of course, you should try to figure out these answers on your own and afterward use MATLAB to check your answers.
 - (a) `>> x + y >= z`
 - (b) `>> x xor(z, x - 2 * y)`
 - (c) `>> (x = -2 * z) | (x^2 > 50 & y^2 > 100)`
 - (d) `>> (x = 2 * z) | (x^2 > 50 & y^2 > 100)`

¹Although most everyone understands the meaning of "and," in spoken English the word "or" is often ambiguous. Sometimes it is intended as the disjunction but other times as the exclusive disjunction. For example, if on a long airplane flight the flight attendant asks you, "Would you like chicken or beef?" Certainly here the exclusive disjunction is intended—indeed, if you were hungry and tried to ask for both, you would probably wind up with only one plus an unfriendly flight attendant. On the other hand, if you were to ask a friend about his/her plans for the coming weekend, he/she might reply, "Oh, I might go play some tennis or I may go to Janice's party on Saturday night." In this case the ordinary disjunction is intended. You would not be at all surprised if your friend wound up doing both activities. In logic (and mathematics and computer programming) there is no room for such ambiguity, so that is why we have two very precise versions of "or."

4. The following while loops were separately entered in different MATLAB sessions. What will the resulting outputs be? Do this one carefully by hand and then use MATLAB to check your answers.

<pre>(a) >> i = 1; x=-3; >> while (i<3) & (x<35) x=-x*(i+1) end (c) >> i = 1; x=-3; >> while xor(i<3, x<35) x=-x*(i+1) end</pre>	<pre>(b) >> i = 1; x=-3; >> while (i<3) (x<35) x=-x*(i+1) end</pre>
--	---

5. The following while loops were separately entered in different MATLAB sessions. What will the resulting outputs be? Do this one carefully by hand and then use MATLAB to check your answers.

```
>>i = 1; x = 2; y = 3;
>> while (i<5) | (x==y)
x = x*2, y = y+x, i=i+1;
end
```

4.2 LOGICAL CONTROL FLOW IN MATLAB

Up to this point, the reader has been given a reasonable amount of exposure to while loops. The *while loop* is quite universal and is particularly useful in those situations where it is not initially known how many iterations must be run in the loop. If we know ahead of time how many iterations we want to run through a certain recursion, it is more convenient to use a for loop. For loops are used to repeat (iterate) a statement or group of statements a specified number of times. The format is as follows:

```
>>for n=(start):(gap):(end )
...MATLAB commands...
end
```

The **counter** n (which could be any variable of your choice) gets automatically bumped up at each iteration by the "gap." At each iteration, the "...MATLAB commands..." are all executed in order (just like they would be if they were to be entered manually again and again). This continues until the counter meets or exceeds the "end" number.

```
>> for n=1:5 \% if 'gap' is omitted it is assumed to be 1.
>> x(n) = n^3; \% we will be creating a vector of cubes of successive integers.
    end \% all output has been suppressed, but a vector x has been created
>> x \% let's display x now
-> x = 1 8 27 64 125
```

Note that since a comma in MATLAB signifies a new line, we could also have written the above for loop in a single line. We do this in the next loop below:

```
>> for n=1:2:10, x(k)=2; end
>> x \% we display x again. Try to guess what it now looks like.
-> x = 2 8 2 64 2 0 2 0 2
```

Observe that there are now nine entries in the vector x . This loop overwrote some of the five entries in the previous vector x (which still remained in MATLAB's workspace). Let us carefully go through each iteration of this loop, explaining exactly what went on at each stage: $k = 1$ (start) \rightarrow we redefine $x(1)$ to be 2 (from its original value of 1). $k = 1 + 2 = 3$ (augment k by gap = 2) \rightarrow redefine $x(3)$ to be 2 ($x(2)$ was left to its original value of 8). $k = 3 + 2 = 5 \rightarrow$ redefine $x(5)$ to be 2. $k = 5 + 2 = 7 \rightarrow$ defines $x(7)$ to be 2 (previously x was a length 5 vector, now it has 7 components), the skipped component $x(6)$ is by default defined to be 0. $k = 7 + 2 = 9 \rightarrow$ defines $x(9)$ to be 2 and the skipped $x(8)$ to be 0. $k = 9 + 2 = 11$ (exceeds end = 10 so for loop is exited and thus completed).

The gap in a for loop can even be a negative number, as in the following example that creates a vector in backwards order. The semicolon is omitted to help the reader convince himself or herself how the loop progresses.

```
>> for i=3:-1: 1, y(i)=i, end
```

```
→ y = 0  0  3
   → y = -y  2  3
   → y = 1  2  3
```

A very useful tool in programming is the if-branch. In its basic form the syntax is as follows:

```
>> if <conditions>
...MATLAB commands...
end
```

The way such an if-branch works is that if the listed <condition >(which can be any MATLAB statement) is true (i.e., has a nonzero value), then all of the "...MATLAB commands..." listed are executed in order and upon completion the if-branch is then exited. If the <condition> is false then the "...MATLAB commands..." are ignored (i.e., they are by-passed) and the if-branch is immediately exited. As with loops in MATLAB, if-branches may be inserted within loops (or branches) to deal with particular situations that arise. Such loops/branches are said to be nested. Sometimes if-branches are used to "raise a flag" if a certain condition arises. The following MATLAB command is often useful for such tasks:

```
fprintf('<any English / text phrase>') -> Causes MATLAB to print: <any English phrase>.
```

Thus the output of the command `fprintf('Have a nice day!')` will simply be `→ Have a nice day!` This command has a useful feature that allows one to print the values of variables that are currently stored within a text phrase. Here is how such a command would work: We assume that (previously in a MATLAB session) the values $w = 2$ and $h = 9$ have been calculated and stored and we enter:

```
>> fprintf('the width of the rectangle is %d,the length is %d.', w, h)
→ the width of the rectangle is 2, the length is 9.»
```

Note that within the "text" each occurrence of % was replaced, in order, by the (current) values of the variables listed at the end. They were printed as integers (without decimals); if we wanted them printed as floating point numbers, we would use of in place of ofd. Also note that MATLAB unfortunately put the prompt » at the end of the output rather than on the next line as it usually does. To prevent this, simply add (at the end of your text but before the single right quote) \r which stands for "carriage return." This carriage return is also useful for splitting up longer groups of text within an fprintf.

Sometimes in a nested loop we will want to exit from within an inner loop and at other times we will want exit from the mother loop (which is the outermost loop inside of which all the other loops/branches are a part of) and thus halt all operations relating to the mother loop. These two exits can be accomplished using the following useful MATLAB commands:

break (anywhere within a loop) →	Causes immediate exit only from the single loop in which break was typed.
return (anywhere within a nested loop) →	Causes immediate exit from the mother loop, or within a function M-file, immediate exit from M-file (whether or not output has been assigned).

The next example illustrates some of the preceding concepts and commands.

EXAMPLE 4.4: Carefully analyze the following two nested loops, decide what exactly they cause MATLAB to do, and then predict the exact output. After you do this, read on (or use MATLAB) to confirm your predictions.

(a)

```
for n=1:5
for k=1:3
a=n+k
if a>=4, break, end
end
end
```

NOTE: We have inserted tabs to make the nested loops easier to distinguish. Always make certain that each loop/branch is paired with its own end.

(b)

```
for n=1:5
for k=1:3
a=n+k
if a>=4
```

```
fprintf('We stop since a has reached the value %d \r', a)
return
end
end
end
```

SOLUTION: Part (a): Both nested loops consist of two loops. The mother loop in each is, as usual, the outermost loop (with counter n). The first loop begins with the mother loop setting the counter n to be 1, then immediately moves to the second loop and sets k to be 1; now in the second loop a is assigned to be the value of $n + k = 1 + 1 = 2$ and this is printed (since there is no semicolon). Since $a = 2$ now, the "if-condition" is not satisfied so the if-branch is bypassed and we now iterate the k -loop by bumping up k by 1 (= default gap). Note that the mother loop's n will not get bumped up again until the secondary k -loop runs its course. So now with $k = 2$, a is reassigned as $a = n + k = 1 + 2 = 3$ and printed, the ifbranch is again bypassed and k gets bumped up to 3 (its ending value), a is now reassigned as $a = n + k = 1 + 3 = 4$ (and printed). The if-branch condition is now met, so the commands within it (in this case only a single "break" command) are run. So we will break out of the k -loop (which is actually redundant at this point since k was at its ending value and the k -loop was about to end anyway). But we are still progressing within the mother n -loop. So now n gets bumped up by 1 to be 2 and we start afresh the k -loop again with $k = 1$. The variable a is now assigned as $a = n + k = 2 + 1 = 3$ (and printed), the if-branch is bypassed since the condition is not met and k gets bumped up to be 2. Next a gets reassigned as $a = n + k = 2 + 2 = 4$, and printed. Now the if-branch condition is met so we exit the k -loop (this time prematurely) and n now gets bumped up to be 3. Next entering the k -loop with $k = 1$, a gets set to be $n + k = 3 + 1 = 4$, and printed, the "ifbranch condition" is immediately satisfied, and we exit the k -loop and n now gets bumped up to be 4. As in the last iteration, the k -loop will just reassign a to be 5 and print this, break and n will go to 5 (its final value). In the final stage, a gets assigned as 6, the if-branch breaks us out of the k -loop, and since n is at its ending value, the mother loop exits. The actual output for part (a) is thus:

```
-> a=2 a=3 a=4 a=3 a=4 a=4 a=5 a=6
```

Part (b): Apart from the fprintf command, the main difference here is the replacement of the break command with the return command. As soon as the if-branch condition is satisfied, the conditions within will be executed and the return will cause the whole nested loop to stop in its tracks. The output will be as follows:

```
-> a=2 a=3 a=4 We stop since a has reached the value 4
```

EXERCISE FOR THE READER 4.1: Below are two nested loops. Carefully analyze each of them and try to predict resulting outputs and then use MATLAB to verify your predictions. For the second loop the output should be given in the default format short.

(a)

```
>>for i=1:5
i
if i>2, fprintf('test'), end
end
```

(b)

```
>>for i=1:8, x(i)=0; end %initialize vector
>>for i=6:-2:2
for j=1:i
x(i)-x(i)+1/j;
end
end
>>x
```

The basic form of the if-branch as explained above allows one to have MATLAB perform a list of commands in the event that one certain condition is fulfilled. In its more advanced form, if-branches can be set up to perform different sets of commands depending on which situation might arise. In the fullest possible form, the syntax of an if-branch is as follows:

```
>> if <condition_1>
...MATLAB commands 1...
elseif <condition_2>
...MATLAB commands _2...
elseif <condition_n> ,
...MATLAB
else
...MATLAB
end
```

There can be any number of else if cases (with subsequent MATLAB commands) and the final else is optional. Here is how such an if-branch would function: The first thing to happen is that $\langle \text{condition}_1 \rangle$ gets tested. If it tests true (nonzero), then the "...MATLAB commands_1..." are all executed in order, after which the if-branch is exited. If $\langle \text{condition}_1 \rangle$ tests false (zero), then the ifbranch moves on to the next $\langle \text{condition}_2 \rangle$ (associated with the first elseif). If this condition tests true, then "...MATLAB commands_2.." are executed and the if-branch is exited, otherwise it moves on to test the next $\langle \text{condition}_3 \rangle$, and so on. If the final else is not present, then once the loop goes through testing all of the conditions, and if none were satisfied, the if-branch would exit without performing any tasks. If the else is present, then in such a situation the "...MATLAB commands..." after the else would be performed as a catch-all to all remaining situations not covered by the conditions listed above.

Our next example will illustrate the use of such an extended if-branch and will also bring forth a rather subtle but important point.

EXAMPLE 4.5: Create a function M-file for the mathematical function defined by the formula:

$$y = \begin{cases} -x^2 - 4x - 2, & \text{if } x < -1 \\ |x|, & \text{if } |x| \leq 1, \\ 2 - e^{\sqrt{x-1}}, & \text{if } x > 1 \end{cases}$$

then store this M-file and get MATLAB to plot this function on the interval $-4 \leq x \leq 4$.

SOLUTION: The M-file can be easily written using an if-branch. If we use the filename ex4_5, here is one possible M-file:

```
function y = ex4_5(x)
if x<-1
y = -x.^2-4*x-2;
elseif x>1
y = 2-exp(sqrt(x-1));
else
y=abs(x);
end
end
```

It is tempting to now obtain the desired plot using the usual command sequence:

```
>>x=-4:.001:4 ; y=ex4_5(x); plot(x,y)
```

There is a subtle problem here, though. If we were to enter these commands, we would obtain the graph of (the last function in the formula) $y = |x|$ on the whole interval $[-4, 4]$. Before we explain this and show how to resolve this problem, it would behoove the reader to try to decide what is causing this to happen and to figure out a way to fix this problem.

If we carefully go on to see what went wrong with the last attempt at a plot, we observe that since x is a vector, the first condition $x < -1$ in the if-branch now becomes a bit ambiguous. When asked about such a vector inequality, MATLAB will return a vector of the same size as x that is made up of zeros and ones. In each slot, the vector is 1 if the corresponding component of x satisfies the inequality ($x < -1$) and 0 if it does not satisfy this inequality. Here is an example:

```
>>[2 -5 3 -2 -1] < -1 %causes MATLAB to test each of the 5 inequalities as true (1)
% or false (0)
->0 1 0 1 0
```

Here is what MATLAB did in the above attempt to plot our function. Since x is a (large) vector, the first condition $x < -1$ produced another vector as the same size as x made up of (both) 0's and 1's. Since the vector was not all true (1's), the condition as a whole was not satisfied so it moved on to the next condition $x > 1$, which for the same reason was also not satisfied and so bypassed. This left us with the catch-all command $y = \text{abs}(x)$ for the whole vector x , which, needless to say, is not what we had intended.

So now how can we fix this problem? One simple fix would be to use a for loop to construct the y-coordinate vector using ex4_5(x) with only scalar values for x .

Here is one such way to construct y :

```
>> size(x) %first we find out
» % the size of x
-> 1 8001
» for n=1:8001 0
y(n)=ex4_5(x(n) ) ;
end
>> plot(x,y) %now we can
>>% get the desired plot
```

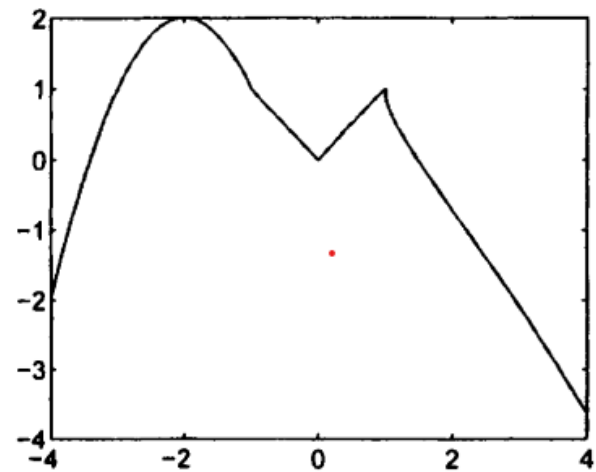


Figure 4.1: The plot of the function of Example 4.5

A more satisfying solution would be to rebuild the M-file in such a way that when vectors are inputted for x , the if-branch testing is done separately for each component. The following program will do the job:

```
function y = ex4_5v2(x)
for i = 1:length(x)
if x(i)<-1
y(i) = -x(i).A
2-4*x(i)-2;
elseif x(i)>1
y(i) = 2-exp(sqrt(x(i)-1));
else
y(i)=abs(x(i) ) ;
end
```

With this M-file stored in our path the following commands would then indeed produce the desired plot of Figure 4.1:

```
>> x=-4:.001:4 ; y=ex4_5v2(x); plot(x,y )
```

In dealing with questions involving integers, MATLAB has several numbertheoretic functions available. We mention three here. They will be useful in the following exercise for the reader as well as in the exercises of this section.

floor(x) →	Gives the greatest integer that is $\leq x$ (the floor of x).
ceil(x) →	Gives the least integer that is $\geq x$ (the ceiling of x).
round(x) →	Gives the nearest integer to x .

For example, $\text{floor}(2.5) = 2$, $\text{ceil}(2.5) = 3$, $\text{ceil}(-2.5) = -1$, and $\text{round}(-2.2) = -2$. Observe that a real number x is an integer exactly when it equals its floor (or ceiling, or $\text{round}(x) = x$). EXERCISE FOR THE READER 4.2: (a) Write a MATLAB function M-file, call it `sum2sq`, that will take as input a positive integer n and will produce the following output:

(i) In case n cannot be written as a sum of squares (i.e., if it is not possible to write $n = a^2 + b^2$ for some nonnegative integers a and b) then the output should be the statement: "the integer $\langle n \rangle$ cannot be written as a sum of squares" (where $\langle n \rangle$ will print as an actual numerical value).

(ii) If n can be written as a sum of squares (i.e., $n = a^2 + b^2$ can be solved for nonnegative integers a and b) then the output should be "the integer $\langle n \rangle$ can be written as the sum of the squares of $\langle a \rangle$ and $\langle b \rangle$ " (here again, $\langle n \rangle$ and also $\langle a \rangle$ and $\langle b \rangle$ will print as a actual numerical values) where a and b are actual solutions of the equation.

(b) Run your program with the following inputs: $n = 5, n = 25, n = 12, 233, n = 100, 000$.

(c) Write a MATLAB program that will determine the largest integer $< 100,000$ that cannot be written as a sum of squares. What is this integer?

(d) Write a MATLAB program that will determine the first integer > 1000 that cannot be written as a sum of squares. What is this integer?

(e) How many integers are there (strictly) between 1000 and 100,000 that cannot be expressed as a sum of the squares of two integers?

A useful MATLAB command syntax for writing interactive script M-files is the following:

$x = \text{input}(' \langle \text{Enter input phrase} \rangle: ') \rightarrow$	When a script with this command is run, you will be prompted in command window by the same to enter an input for script after which your input will be stored as variable x and the script will be executed.
--	--

The command can, of course, also be invoked in a function M-file, or at any time in the MATLAB command window. The next example presents a way to use this command in a nice mathematical experiment.

EXAMPLE 4.6: (Number Theory: The Collatz Problem) Suppose we start with any positive integer a_1 , and perform the following recursion to define the rest of the sequence a_1, a_2, a_3, \dots :

$$a_{n+1} = \begin{cases} a_n/2, & \text{if } a_n \text{ is even} \\ 3a_n + 1, & \text{if } a_n \text{ is odd} \end{cases}.$$

We note that if a term a_n in this sequence ever reaches 1, then from this point on the sequence will cycle through the values 1, 4, 2. For example, if we start with $a_1 = 5$, the recursion formula gives $a_2 = 3 \cdot 5 + 1 = 16$, and then $a_3 = 16/2 = 8, a_4 = 8/2 = 4, a_5 = 4/2 = 2, a_6 = 2/2 = 1$, and so on (4, 2, 1, 4, 2, 1, ...). Back in 1937, German mathematician Lothar Collatz conjectured that no matter what positive integer we start with for a_1 , the above recursively defined sequence will always reach the 1, 4, 2 cycle. Collatz is an example of a mathematician who is more famous for a question he asked than for problems he solved or theorems he proved (although he did significant research in numerical differential equations). The Collatz conjecture remains an open problem to this day.² Our next example will give a MATLAB script that is useful in examining the Collatz conjecture. Some of the exercises will outline other ways to use MATLAB to run some illuminating experiments on the Collatz conjecture.

EXAMPLE 4.7: We write a script (and save it as `collatz`) that does the following. It will ask for an input for a positive integer to be the initial value $a(1)$ of a Collatz experiment. The program will then run through the Collatz iteration scheme until the sequence reaches the value 1, and so begins to cycle (if ever). The script should output a sentence telling how many iterations were used for this Collatz experiment, and also give the sequence of numbers that were run through until reaching the value of 1.

```
%Collatz script
a(i) = input('Enter a positive integer: ');
n=1;
while a(n) ~= 1
    if ceil(a(n)/2)==a(n)/2 %tests if a(n) is even
        a(n+1)=a(n)/2;
    else
        a(n+1)=3*a(n)+1;
    end
    n=n+1;
end
fprintf('\n Collatz iteration with initial value a(1)= %d \n', a(1))
fprintf(' took %d iterations before reaching the value 1 and ', n-1)
fprintf(' beginning \n to cycle. The resulting pre-cycling')
fprintf('* sequence is as follows:')
a
clear a %lets us start with a fresh vector a on each run
```

Enter a positive integer: 5 (MATLAB gives the first message, we only enter 5, and enter to get all of the informative output below.)

```
->Collatz iteration with initial value a(1) = 5 took
5 iterations before reaching the value 1 and beginning
to cycle. The resulting pre-cycling sequence is as follows:
a =5 16 8 4 2 1
```

EXERCISE FOR THE READER 4.3: (a) Try to understand this script, enter it, and run it with these values: $a(1) = 6, 9, 1, 12, 19, 88, 764$. Explain the purpose of the last command in the above script that cleared the vector `a`.

EXERCISES 4.2:

²The Collatz problem has an interesting history; see, for example [Lag-85] for some details. Many mathematicians have proved interesting results that strongly support the truth of the conjecture. For example, in 1972, the famous Princeton number-theorist J. H. Conway [Con-72] proved that if a Collatz iteration enters into a cycle other than (1, 4, 2), the cycle must be of length at least 400 (i.e., the cycle itself must consist of at least 400 different numbers). Subsequently, J. C. Lagarias (in [Lag-85]) extended Conway's bound from 400 to 275,000! Recent high-speed computer experiments (in 1999, by T. Oliveira e Silva [OeS-99]) have shown the Collatz conjecture to be true for all initial values of the sequence less than about 2.7×10^{16} . Despite all of these breakthroughs, the problem remains unsolved. P. Erdős, who was undoubtedly one of the most powerful problem-solving mathematicians of the twentieth century, was quoted once as saying "Mathematics is not yet ready for such problems," when talking about the Collatz conjecture. In 1996 a prize reward of 1,000 (approx. \$2,000) was offered for settling the Collatz conjecture. Other math problems have (much) higher bounties. For example the Clay Foundation (URL: www.claymath.org/prizeproblems/statement.htm) has listed seven math problems and offered a prize of \$1 million for each one Enter a positive integer: 5 (MATLAB gives the first message, we only enter 5, and enter to then get all of the informative output below.

1. Write a MATLAB function M-file, called `sumodsq (n)`, that does the following: The input is a positive integer n . Your function should compute the sum of the squares of all odd integers that do not exceed n :

$$1^2 + 3^2 + 5^2 + \cdots + k^2$$

where k is the largest odd integer that does not exceed n . If this sum is less than 1 million, the output will be the actual sum (a number); if this sum is greater than or equal to 1 million, the output will be the statement " $< n >$ is too big" where " $< n >$ " will appear as the actual number that was inputted.

2. Write a function M-file, call it `sevenpow (n)`, that inputs a positive integer n and that figures out how many factors of 7 n has (call this number k) and outputs the statement: "The largest power of 7 which $\langle n \rangle$ contains as a factor is $\langle k \rangle$." So for example, if you run `sevenpow(98)` your output should be the sentence "The largest power of 7 which 98 contains as a factor is 2." Run the commands: `sevenpow(36067)`, `sevenpow(671151153)`, and `sevenpow(3080641535629)`.
3. (a) Write a function M-file, call it `sumsq (n)`, that will input a positive integer n and output the sum of the squares of all positive integers that are less than or equal to n ($\text{sumsq}(n) = 1^2 + 2^2 + 3^2 + \cdots + n^2$). Check and debug this program with the results $\text{sumsq}(1) = 1$, $\text{sumsq}(3) = 14$.
4. (a) Write a MATLAB function M-file, call it `sum 2s(n)`, that will take as input a positive integer n and will produce for the output either of the following:
 - (i) The sum of all of the positive powers of 2 ($2 + 4 + 8 + 16 + \cdots$) that do not exceed n , provided this sum is less than 50 million.
 - (ii) In case the sum in (i) is greater than or equal to 50 million, the output should simply be "overflow"
 (b) Run your function with the following inputs: $n = 1, n = 10, n = 265, n = 75,000, n = 65,000,000$.
 (c) Write a short MATLAB code that will determine the largest integer n for which this program does not produce "overflow."
5. (a) Write a MATLAB function M-file, called `bigpro (x)`, that does the following: The input is a real number x . The only output of your function should be the real number formed by the product $x(2x)(3x)(4x) \cdots (nx)$, where n is the first positive integer such that either nx is an integer or $|nx|$ exceeds x^2 (whichever comes first).
 (b) Of course, after you write the program you need to debug it. What values should result if we were to use the (correctly created) program to find: `bigpro(4)`, `bigpro(2.5)`, `bigpro(12.7)`? Run your program for these values as well as for the values $x = -3677/9, x = 233.6461$, and $x = 125,456.789$.
6. (*Probability: The Birthday Problem*) This famous problem in probability goes as follows: If there is a room with a party of people and everyone announces his or her birthday, how many people (at least) would there need to be in the room so that there is more than a 50% chance that at least two people have the same birthday? To solve this problem, we let $P(n)$ = the probability of a common birthday if there are n people in the room. Of course $P(1) = 0$ (no chance of two people having the same birthday if there is only one person in the room), and $P(n) = 1$ when $n > 365$ (there is a 100% chance, i.e., guaranteed, two people will have the same birthday if there are more people in the room than days of the year; we ignore leap-year birthdays). We can get an expression for $P(n)$ by calculating the complementary probability (i.e., the probability that there will not be a common birthday among n different people). This must be

$$\frac{365}{365} \cdot \frac{364}{365} \cdot \frac{363}{365} \cdots \frac{366-n}{365}$$

This can be seen as follows: The first person can have any of the 365 possible birthdays, the second person can have only 364 possibilities (since he/she cannot have the same birthday as the first person), the third person is now restricted to only 363 possible birthdays, and so on. We multiply the individual probabilities (fractions) to get the combined probability of no common birthday. Now this is the complementary probability of what we want (i.e., it must add to $P(n)$ to give $1 = 100\%$ since it is guaranteed that either there is a common birthday or not). Thus

$$P(n) = 1 - \frac{365}{365} \cdot \frac{364}{365} \cdot \frac{363}{365} \cdots \frac{366-n}{365}$$

- (a) Write a MATLAB function M-file for this function $P(n)$. Call the M-file `bprob (n)` and set it up so that it does the following: If n is a nonnegative integer, the function `bprob (n)` will output the sentence: "If a room contains $< n >$ people then the probability of a common birthday is $< P(n) >$ " where " $< n >$ " and " $< P(n) >$ " should be the actual numerical values. If n is any other type of number (e.g., a negative number or 2.6) the output should be "Input $\langle n \rangle$ is not a natural number so the probability is undefined." Save your M-file and then run it for the following values: $n = 3, n = 6, n = 15, n = 90, n = 110.5$, and $n = 180$
- (b) Write a MATLAB code that uses your function in part (a) to solve the birthday problem, i.e., determine the smallest n for which $P(n) > .5$. More precisely, create a for loop whose only output will be: n = the minimum number needed (for $P(n)$ to be $> .5$) and the associated probability $P(n)$.

(c) Get MATLAB to draw a neat plot of $P(n)$ vs. n (for all n between 1 and 365), and on the same plot, include the plots of the two horizontal lines with y -intercepts .5 and .9. Interpret the intersections.

7. Write a function *M*-file, call it `pythag (n)`, that inputs a positive integer n and determines whether n is the hypotenuse of a right triangle with sides of integer lengths. Thus your program will determine whether the equation $n^2 = a^2 + b^2$ has a solution with a and b both being positive integers.

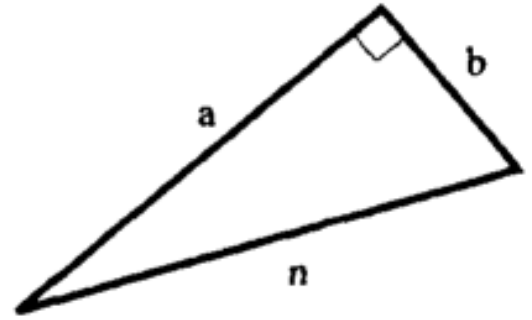


Figure 4.2: Pythagorean triples.

Such triples n, a, b are called Pythagorean triples (Figure 4.2). In case there is no solution (as, for example, if $n = 4$), your program should output the statement: "There are no Pythagorean triples with hypotenuse $\langle n \rangle$." But if there is a solution your output should be a statement that actually gives a specific Pythagorean triple for your value of n . For example, if you type `pythag (5)`, your output should be something like: "There are Pythagorean triples having 5 as the hypotenuse, for example: 3, 4, 5 is one such triple." Run this for several different values of n . Can you find a value of n larger than 1000 that has a Pythagorean triple? Can you find an n that has two different Pythagorean triples associated with it (of course not just by switching a and b)?

HISTORICAL NOTE: Since the ancient times of Pythagoras, mathematicians have tried long and hard to find integer triple solutions of the corresponding equation with exponent 3: $n^3 = a^3 + b^3$. No one has ever succeeded. In the 1700s the amateur French mathematician Pierre Fermat conjectured that no such triples can exist. He claimed to have a truly remarkable proof of this but there was not enough space in the margin of his notes to include it. There has been an incredible amount of research trying to come up with this proof. Just recently, more than 300 years since Fermat stated his conjecture, Princeton mathematician Andrew Wiles came up with a proof. He was subsequently awarded the Fields medal, the most prestigious award in mathematics.

8. (*Plane Geometry*) For an integer n that is at least equal to 3, a regular n -gon in the plane is the interior of a set whose boundary consists of n flat edges (sides) each having the same length (and such that the interior angles made by adjacent edges are all equal). When $n = 3$ we get an equilateral triangle, when $n = 4$ we get a square, and when $n = 8$ we get a regular octagon, which is the familiar stop-sign shape. There are regular n -gons for any such value of n ; some are pictured in Figure 4.3.

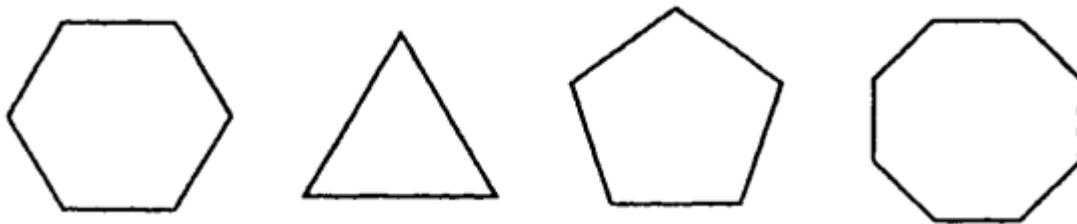


Figure 4.3: Some regular polygons

(a) Write a MATLAB function *M*-file, `ngonper 1(n,dia)`, that has two input variables, n = the number of sides of the regular n -gon, and dia = the diameter of the regular n -gons. The diameter of an n -gon is the length of the longest possible segment that can be drawn connecting two points on the boundary. When n is even, the diameter segment cuts the n -gons into two congruent (equal) pieces. Assuming that n is an even integer greater than 3 and dia is any positive number, your function should have a single output variable that equals the perimeter of the regular n -gons with diameter = dia . Your solution should include a handwritten mathematical derivation of the formula for this perimeter. This will be the hard part of this exercise, and it should be done, of course, before you write the program. Run your program for the following sets of input data: (i) $n = 4$, $dia = \sqrt{4}$, (ii) $n = 12$, $dia = 12$, (iii) $n = 1000$, $dia = 5000$.

(b) Remove the restriction that n is even from your program in part (a). The new function (call it now `ngonper (n, dia)`) will now do everything that the one you constructed in part (a) did but it will be able to input and deal with any integer n greater than or equal to 3. Again, include with your solution a mathematical derivation of the perimeter

formula you are using in your program. Run your program for these sets of values: (i) $n = 3$, $\text{dia} = 2$, (ii) $n = 5$, $\text{dia} = 4$, (iii) $n = 999$, $\text{dia} = 500$.

(c) For which values of n (if any) will your function in part (b) continue to give the correct perimeter of an n -gon that is no longer regular? An irregular n -gon is the interior of a set in the plane whose boundary consists of n flat edges whose interior angles are not all equal. Examples of irregular n -gons include any nonequilateral triangle ($n = 3$), any quadrilateral that is not a square ($n = 4$). For those n 's for which you say things still work, a (handwritten mathematical) proof should be included, and for those n 's for which you say things no longer continue to work, a (handwritten) counterexample should be included.

9. (*Plane Geometry*) This exercise consists of doing what is asked for in Exercise 8 (a)(b)(c) but with changing all occurrences of the word "perimeter" to "area." In parts (a) and (b) use the M-file names `ngonar 1(n, dia)` and `ngonarea (n, dia)`.
10. (*Finance: Compound Interest*) Write a script file called `compints` that will compute (as output) the future value A in a savings account after prompting the user for the following inputs: the principal P (= amount deposited), the annual interest rate r (as a decimal), the number k of compoundings per year (so quarterly compounding means $k = 4$, monthly means $k = 12$, daily means $k = 365$, etc.), and the time t that the money is invested (measured in years). The relevant formula from finance is $A = P(1 + r/k)^{kt}$. Run the script using the following sets of inputs: $P = \$10,000$, $r = 8\%(.08)$, $k = 4$, and $t = 10$, then changing t to 20, then also changing r to 11%.

Suggestion: You probably want to have four separate "input" lines in your script file, the first asking for the principal, etc. Also, to get the printout to look nice, you should switch to format bank inside the script and then (at the very end) switch back to format short.

11. (*Finance: Compound Interest*) Write a script file called `comings` that takes the same inputs as in the previous exercise, but instead of producing the output of the future account balance, it should produce a graph of the future value A as a function of time as the time t ranges from zero (day money was invested) until the end of the time period that was entered. Run the script for the three sets of data in the previous problem.
12. (*Finance: Future Value Annuities*) Write a script file called `fvanns` that will compute (as output) the future value FV in an annuity after prompting the user for the following inputs: the periodic payment PMT (= amount deposited in account per period), the annual interest rate r (as a decimal), the number k of periods per year, that is, the number of compoundings per year (so quarterly compoundings/deposits means $k = 4$, monthly means $k = 12$, bimonthly means $k = 24$, etc.), and the time t that the money is invested (measured in years). The relevant formula from finance is $FV = PMT \left((1 + r/k)^{kt} - 1 \right) / (r/k)$. Run the script using the following sets of inputs: $PMT = 200$, $r = 7\%(.07)$, $k = 12$, and $t = 30$, then changing t to 40, then also changing r to 9%. Next change PMT to 400 on each of these three sets of inputs. Note, the first set of inputs could correspond to a worker who starts a supplemental retirement plan (say a 401(k)), deposits \$200 each month starting at age 35, and continues until he/she plans to retire at age 65 ($t = 30$ years later). The FV will be his/her retirement nest egg at time of retirement. The next set of data could correspond to the same retirement plan but started at age 25 (10 years more time). In each case compare the future value with the total amount of contributions. To encourage such supplemental retirement plans, the federal government allows such contributions (with limits) to be done before taxation.

Suggestion: You probably want to have four separate "input" lines in your script file, the first asking for the principal, etc. Also, to get the printout to look nice, you should switch to format bank inside the script and then (at the very end) switch back to format short.

13. (*Finance: Future Value Annuities*) In this exercise you will be writing a script file that will take the same inputs as in the previous exercise (interactively), but instead of just giving the future value at the end of the time period, this script will produce a graph of the growth of the annuity's value as a function of time.
 - (a) Base your script on the formula given in the preceding exercise for future value annuities. Call this script `fvanns`. Run the script for the same sets of inputs that were given in the previous exercise.
 - (b) Rewrite the script file, this time constructing the vector of future values using a recursion formula rather than directly (as was asked in part (a)). Call this script `fvann2s`. Run the script for the same sets of inputs that were given in the previous exercise.
14. (*Number Theory: The Collatz Problem*) Write a function M-file, call it `collctr`, that takes as input, a positive integer n (the first element for a Collatz experiment), and has as output the positive integer n , which equals the number of iterations required for the Collatz iteration to reach the value of 1. What is the first positive integer n for which this number of iterations exceeds 100? 200? 300?

4.3 WRITING GOOD PROGRAMS

Up to this point we have introduced the two ways that programs can be written and stored for MATLAB to use (function M-files and script M-files) and we have also introduced the basic elements of control flow and a few very useful

built-in MATLAB functions. To write a good program for a specified task, we will need to put all of our skills together to come up with an M-file that, above all, does what it is supposed to do, is efficient, and is as eloquent as possible. In this section we present some detailed suggestions on how to systematically arrive at such programs. Programming is an art and the reader should not expect to master it easily or in a short time.

STEP 1: Understand the problem, do some special cases by hand, and draw an outline. Before you begin to actually type out a program, you should have a firm understanding of what the problem is (that the program will try to solve) and know how to solve it by hand (in theory, at least). Computers are not creative. They can do very well what they are told, but you will need to tell them exactly what to do, so you had better understand how to do what needs to be done. You should do several cases by hand and record the answers. This data will be useful later when you test your program and debug it if necessary. Draw pictures (a flowchart), and write in plain English an explanation of the program, trying to be efficient and avoiding unnecessary tasks that will use up computer time.

STEP 2: Break up larger programs into smaller module programs. Larger programs can usually be split up into smaller independent programs. In this way the main program can be considerably reduced in size since it can call on the smaller module programs to perform secondary tasks. Such a strategy has numerous advantages. Smaller programs are easier to write (and debug) than larger ones and they may be used to create other large or improved programs later on down the road.

STEP 3: Test and debug every program. This is not an option. You should always test your programs with a variety of inputs (that you have collected output data for in Step 1) to make sure all of the branches and loops function appropriately. Novice and experienced programmers alike are often shocked at how rarely a program works after it is first written. It may take many attempts and changes to finally arrive at a fully functional program, but a lot of valuable experience can be gained in this step. It is one thing to look at a nice program and think one understands it well, but the true test of understanding programming is to be able to create and write good programs. Before saving your program for the first time, always make sure that every "for", "while", or "if" has a matching "end". One useful scheme when debugging is to temporarily remove all semicolons from the code, perhaps add in some auxiliary output to display, and then run your program on the special cases that you went through by hand in Step 1. You can see first-hand if things are proceeding along the lines that you intended.

STEP 4: After it finally works, try to make the program as efficient and easy to read as possible. Look carefully for redundant calculations. Also, try to find ways to perform certain required tasks that use minimal amounts of MATLAB's time. Put in plenty of comments that explain various elements of the program. While writing a complicated program, your mind becomes full of the crucial and delicate details. If you read the same program a few months (or years) later (say, to help you to write a program for a related task), you might find it very difficult to understand without a very time-consuming analysis. Comments you inserted at the time of writing can make such tasks easier and less time consuming. The same applies even more so for other individuals who may need to read and understand your program.

The efficiency mentioned in Step 4 will become a serious issue with certain problems whose programs (even good ones) will push the computer to its limits. We will come up with many examples of such problems this book. We mention here two useful tools in testing efficiency of programs or particular tasks. A flop (abbreviation for floating point operation) is roughly equivalent to a single addition, subtraction, multiplication, or division of two numbers in full MATLAB precision (rather than a faster addition of two single-digit integers, say). Counting flops is a common way of comparing and evaluating efficiency of various programs and parts thereof. MATLAB has convenient ways of counting flops⁴ or elapsed time (tic/toc) :

flops(0) ...MATLAB commands... flops	The flops (0) resets the flop counter at zero. The flops tells the number of flops used to execute the "MATLAB commands" in between. (Not available since Version 5, see Footnote 3.)
tic ...MATLAB commands... toc	This tic resets the stopwatch to zero. The toe will tell the elapsed time used The toe will tell the elapsed time used to execute the "MATLAB commands.")

The results of *tic/toc* depend not just on the MATLAB program but on the speed of the computer being used, as well as other factors, such as the number of other tasks concurrently being executed on the same computer. Thus the same MATLAB routines will take varying amounts of time on different computers (or even on the same computer under different circumstances). So, unlike flop comparisons, *tic/toc* comparisons cannot be absolute.

EXAMPLE 4.8: Use the *tic/toc* commands to compare two different ways of creating the following large vector: [123...10,000]. First use the nonloop construction and then use a for loop. The results will be quite shocking, and since we will need to work with such large single vectors quite often, there will be an important lesson to be learned from this example. When creating large vectors in MATLAB, avoid, if possible, using "loops."

⁴The flop commands in MATLAB are actually no longer available since Version 5 (until further notice). This is due to the fact that, starting with Version 6, the core programs in MATLAB got substantially revised to be much more efficient in performing matrix operations. It was unfortunate that the flop counting features could no longer be made to perform in this newer platform (collateral damage). Nonetheless, we will, on occasion, use this function in cases where flop counts will help to illustrate important points. Readers that do not have access to older versions of MATLAB will not be able to mimic these calculations.

SOLUTION:

```
>>tic , for n=1:10000 , x(n)=n ; end, toc
->elapsed_time =8.9530 (time is measured in seconds)
>> tic , y=1:10000; toc
->elapsed_time = 0
```

This gives some basis for comparison. We see that the non-loop technique built a vector 10 times as large in about 1/1000 of the time that it took the loop construction to build the smaller vector! The flop-counting comparison method would not apply here since no flops were done in these constructions.

Our next example will deal with a concept from linear algebra called the determinant of a square matrix, which is a certain important number associated with the matrix. We now give the definition of the determinant of a square $n \times n$ matrix⁴

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix}.$$

If $n = 1$, so $A = [a_{11}]$, then the determinant of A is simply the number a_{11} . If $n = 2$, so $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$, the determinant of A is defined to be the number $a_{11}a_{22} - a_{12}a_{21}$ that is just the product of the main diagonal entries (top left to bottom right) less the product of the off diagonal entries (top right to bottom left).

For $n = 3$, $A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$ and the determinant can be defined using the $n = 2$ definition by the so-called cofactor expansion (on the first row). For any entry a_{ij} of the 3×3 matrix A , we define the corresponding submatrix A_{ij} to be the 2×2 matrix obtained from A by deleting the row and column of A that contain the entry a_{ij} . Thus, for example,

$$A_{13} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix}.$$

Abbreviating the determinant of the matrix A by $\det(A)$, the determinant of the 3×3 matrix A is given by the following formula:

$$\det(A) = a_{11} \det(A_{11}) - a_{12} \det(A_{12}) + a_{13} \det(A_{13}).$$

Since we have already shown how to compute the determinant of a 2×2 matrix, the right side can be thus computed. For a general $n \times n$ matrix A , we can compute it with a similar formula in terms of some of its $(n-1) \times (n-1)$ submatrices:

Below are two MATLAB commands that are used to work with entries and submatrices of a given general matrix A .

$A(i,j) \rightarrow$	Represents the entry a_{ij} located in the i th row and the j th column of the matrix A .
$A([i1 \ i2 \ \dots \ imax])$ $A([j1 \ j2 \ \dots \ jmax]) \rightarrow$	Represents the submatrix of the matrix A formed using the rows $i1, i2, \dots, imax$ and columns $j1, j2, \dots, jmax$.
$A([i1 \ i2 \ \dots \ imax], :) \rightarrow$	Represents the submatrix of the matrix A formed using the rows $i1, i2, \dots, imax$ and all columns

EXAMPLE 4.9: (a) Write a MATLAB function file, called `mydet2 (A)`, that calculates the determinant of a 2×2 matrix A .

(b) Using your function `mydet2` of part (a), build a new MATLAB function file called `mydet3(A)` that computes the determinant of a 3×3 matrix A (by performing cofactor expansion along the first row).

(c) Write a program `mydet (A)` that will compute the determinant of a square matrix of any size. Test it on the matrices shown below. MATLAB has a built-in function `det` for computing determinants. Compare the results, flop counts (if available), and times using your function `mydet` versus MATLAB's program `det`. Perform this comparison also for a randomly generated 8×8 matrix. Use the following command to generate random matrices:

⁴The way we define the determinant here is different from what is usually presented as the definition. One can find the formal definition in books on linear algebra such as [HoKu-71]. What we use as our definition is often called *cofactor expansion on the first row*. See [HoKu-71] for a proof that this is equivalent to the formal definition. The latter is actually more complicated and harder to compute and this is why we chose cofactor expansion

<code>rand(n,m) →</code> NOTE: <code>rand (n)</code> is equivalent to <code>rand (n, n)</code> , and <code>rand</code> to <code>rand (1)</code> .	Generates an $n \times m$ matrix whose entries are randomly selected from 0^5
---	--

SOLUTION: The programs in parts (a) and (b) are quite straightforward:

```
function y = mydet2(A)
y=A(1,1)*A(2,2)-A(1,2)*A(2,1);
function y = mydet3(A)
y=A(1,1)*mydet2(A(2:3,2:3))-A(1,2)*mydet2(A(2:3,[1...
3]))+A(1,3)*mydet2(A(2:3,1:2));
```

NOTE: The three dots (...) at the end of a line within the second function indicate (in MATLAB) a continuation of the command. This prevents the carriage return from executing a command that did not fit on a single line.

The program for part (c) is not quite so obvious. The reader is strongly urged to try and write one now before reading on.

Without having the `mydet` program call on itself, the code would have to be an extremely inelegant and long jumble. Since MATLAB allows its functions to (recursively) call on themselves, the program can be elegantly accomplished as follows:

```
function y = mydet(A)
y=0; %initialize y
[n, n] = size(A); %record the size of the square matrix A
if n ==2
y=mydet2(A) ;
return
end
for i=1:n
y=y+(-1)^(i + 1)*A(1,i)*mydet(A(2:n, [1:(i-1) (i+1):n]));
end
```

Let's now run this program side by side with MATLAB's `det` to compute the requested determinants.

```
>>A=[2 7 8 10; 0 -1 4 -9 ; 0 0 3 6; 0 0 0 5] ;
>> A1=[1 2 -1 -2 1 2;0 3 0 2 0 1;1 0 2 0 3 0;1 1 1 1 1 1; . . .
-2-10123 ; 123123] ;

>>flops(0) , tic , mydet(A), toc , flops
->ans = -30(=determinant), elapsed_time = 0.0600, ans = 182 (=flop count)

>> flops(0) , tic , mydet(A1), toc , flops
->ans = 324, elapsed_time = 0.1600, ans =5226

>> flops(0) , tic , det(A), toc , flops
->ans =-30, elapsed_time = 0, ans =52

>> flops(0) , tic , det(A1) , toc , flops
->ans =324, elapsed_time = 0, ans = 117
```

So far we can see that MATLAB's built-in `det` works quicker and with a lot less flops than our `mydet` does. Although `mydet` still performs reasonably well, check out the flop-count ratios and how they increased as we went from the 4×4 matrix `A` to the 6×6 matrix `A1`. The ratio of flops `mydet` used to the number that `det` used rose from about a factor of 3.5 to a factor of nearly 50. For larger matrices, the situation quickly gets even more extreme and it becomes no longer practical to use `mydet`. This is evidenced by our next computation with an 8×8 matrix.

```
>>Atest = rand(8) ; %we suppress output here .
>>flops(0) , tic , det(Atest) , toc , flops
->ans = -0.0033, elapsed_time = 0, ans = 326
>> flops(0) , tic , mydet(Atest) , toc , flops
->ans = -0.0033, elapsed_time =8.8400, ans = 292178
```

MATLAB's `det` still works with lightning speed (elapsed time was still undetectable) but now `mydet` took a molasses-slow nearly 9 seconds, and the ratio of the flop count went up to nearly 900! If we were to go to a 20×20 matrix, at this pace, our `mydet` would take over 24 years to do! (See Exercise 5 below.) Surprisingly though, MATLAB's `det` can find the

determinant of such a matrix in less than 1/100 of a second (on the author's computer) with a flop count of only about 5000. This shows that there are more practical ways of computing (large matrix determinants) than by the definition or by cofactor expansion. In Chapter 7 such a method will be presented.

Each time when the `rand` command is invoked, MATLAB uses a program to generate a random numbers so that in any given MATLAB session, the sequence of "random numbers" generated will always be the same. Random numbers are crucial in the important subject of simulation, where trials of certain events that depend on chance (like flipping a coin) need to be tested. In order to assure that the random sequences are different at each start of a MATLAB session, the following command should be issued before starting to use `rand`:

<pre>rand('state',sum(100*clock)) →</pre>	<p>This sets the "state" of MATLAB's random number generator in a way that depends in a complicated fashion on the current computer time. It will be different each time MATLAB is started.</p>
---	---

Our next exercise for the reader will require the ability to store strings of text into rows of numerical matrices, and then later retrieve them. The following basic example will illustrate how such data transformations can be accomplished: We first create text string `T` and a numerical vector `v`:

```
>> T = 'Test' , v = [1 2 3 4 5 6]
->T = Test, v= 1 2 3 4 5 6
```

If we examine how MATLAB has stored each of these two objects, we learn that both are "arrays," but `T` is a "character array" and `v` is a "double array" (meaning a matrix of numbers):

```
>> whos T v
->Name    Size    Bytes    Class
   T      1x4      8      char array
   v      1x6     48     double char
```

If we redefine the first four entries of the vector `v` to be the vector `T`, we will see that the characters in `T` get transformed into numbers:

```
>> v(1:4)=T
->v = 84 101 115 116 5 6
```

MATLAB does this with an internal dictionary that translates all letters, numbers, and symbols on the keyboard into integers (between 1 and 256, in a case-sensitive fashion). To transform back to the original characters, we use the `char` command, as follows:

```
>> U=char(v(1:4))
->U =Test
```

Finally, to call on a stored character (string) array within an *fprintf* statement, the symbol `%s` is used as below:

```
>> fprintf('The %s has been performed.' , U)
->The Test has been performed.
```

EXERCISE FOR THE READER 4.4: (Electronic Raffle Drawing Program)

(a) Create a script M-file, *raffledraw*, that will randomly choose the winner of a raffle as follows: When run, the first thing the program will do is prompt the user to enter the number of players (this can be any positive integer). Next it will successively ask the user to input the names of the players (in single quotes, as text strings are usually inputted) along with the corresponding weight of each player. The weight of a player can be any positive integer and corresponds to the number of raffle tickets that the player is holding. Then the program will randomly select one of these tickets as the winner and output a phrase indicating the name of the winner.

(b) Run your M-file with the following data on four players: Alfredo has four tickets, Denise has two tickets, Sylvester has two tickets, and Laurie has four tickets. Run it again with the same data.

EXERCISES 4.3:

1. Write a MATLAB function M-file, call it `sum3sq(n)`, that takes as input a positive integer n and as output will do the following. If n can be expressed as a sum of three squares (of positive integers), i.e., if the equation:

$$n = a^2 + b^2 + c^2$$

has a solution with a, b, c all positive integers, then the program should output the sentence, "The number $\langle n \rangle$ can be written as the sum of the squares of the three positive integers $\langle a \rangle$, $\langle b \rangle$, and $\langle c \rangle$." Each of the numbers in brackets must be actual integers that solve the equation. In case the equation has no solution (for a, b, c), the output should be the sentence: "The number $\langle n \rangle$ cannot be expressed as a sum of the squares of three positive integers." Run your program with the numbers $n = 3, n = 7, n = 43, n = 167, n = 994, n = 2783, n = 25,261$. Do you see a pattern for those integers n for which the equation does/does not have a solution?

2. Repeat Exercise 1 with "three squares" being replaced by "four squares," so the equation becomes:

$$n = a^2 + b^2 + c^2 + d^2$$

Call your function `sum 4 sq`. In each of these problems feel free to run your programs for a larger set of inputs so as to better understand any patterns that you may perceive.

3. (*Number Theory: Perfect Numbers*) (a) Write a MATLAB function M-file, call it `divsum (n)`, that takes as input a positive integer n and gives as output the sum of all of the proper divisors of n . For example, the proper divisors of 10 are 1, 2, and 5 so the output of `divsum (10)` should be 8 ($= 1 + 2 + 5$). Similarly, `divsum (6)` should equal 6 since the proper divisors of 6 are 1, 2, and 3. Run your program for the following values of n : $n = 10, n = 224, n = 1410$ (and give the outputs).

(b) In number theory, a perfect number is a positive integer n that equals the sum of its proper divisors, i.e., $n = \text{divsum}(n)$. Thus from above we see that 6 is a perfect number but 10 is not. In ancient times perfect numbers were thought to carry special magical properties. Write a program that uses your function in part (a) to get MATLAB to find and print all of the perfect numbers that are less than 1000. Many questions about perfect numbers still remain perfect mysteries even today. For example, it is not known if the list of perfect numbers goes on forever.

4. (*Number Theory: Prime Numbers*) Recall that a positive integer n is called a prime number if the only positive integers that divide evenly into n are 1 and itself. Thus 4 is not a prime since it factors as 2×2 . The first few primes are as follows: 2, 3, 5, 7, 11, 13, 17, 19, 23, ... (1 is not considered a prime for some technical reasons). There has been a tremendous amount of research done on primes, and there still remain many unanswered questions about them that are the subject of contemporary research. One of the first questions that comes up about primes is whether there are infinitely many of them (i.e., does our list go on forever?). This was answered by an ancient Greek mathematician, Euclid, who proved that there are infinitely many primes. It is a very time-consuming task to determine if a given (large) number is prime or not (unless it is even or ends in 5).

(a) Write a MATLAB function M-file, call it `primeck (n)`, that will input a positive integer $n > 1$, and will output either the statement: "the number $\langle n \rangle$ is prime," if indeed, n is prime, or the statement "the number $\langle n \rangle$ is not prime, its smallest prime factor is $< k >$," if n is not prime, and here k will be the actual smallest prime factor of n .

Test (and debug) your program for effectiveness with the following inputs for n :

$$n = 51, n = 53, n = 827, n = 829.$$

Next test your program for efficiency with the following inputs (depending on how you wrote your program and also how much memory your computer has, it may take a very long time or not finish with these tasks)

$$n = 8237, n = 38877, n = 92173, n = 1,875,247, n = 2038074747, n = 22801763489, \\ n = 1689243484681, n = 7563374525281.$$

In your solution, make sure to give exactly what the MATLAB printout was; also, next to each of these larger numbers, write down how much time it took MATLAB to perform the calculation.

(b) Given enough time (and assuming you are working on a computer that will not run out of memory) will this MATLAB program always work correctly no matter how large n is? Recall that MATLAB has an accuracy of about 15 significant digits.

5. We saw in Example 4.7 that by calculating a determinant by using cofactor expansion, the number of flops (additions, subtractions, multiplications, and divisions) increases dramatically. For a 2×2 matrix, the number (worst-case scenario, assuming no zero entries) is 3; for a 3×3 matrix it is 14. What would this number be for a 5×5 matrix? For a 9×9 matrix? Can you determine a general formula for an $n \times n$ matrix?
6. (*Probability and Statistics*) Write a program called `cointoss (n)` that will have one input variable $n =$ a positive integer and will simulate n coin tosses, by (internally) generating a sequence of n random numbers (in the range $0 \leq x \leq 1$) and will count each such number that is less than 0.5 as a "HEAD" and each such number that is greater than 0.5 as a "TAIL." If a number in the generated sequence turns out to be exactly $= 0.5$, another simulated coin toss should be made (perhaps repeatedly) until a "HEAD" or a "TAIL" comes up. There will be only one

output variable: P = the ratio of the total number of "HEADS" divided by n . But the program should also cause the following sentence to be printed: "In a trial of $< n >$ coin tosses, we had $< H >$ flips resulting in 'HEAD' and $< T >$ flips resulting in 'TAIL,' so 'HEADS' came up $< 100P >$ % of the time." Here, $< H >$ and $< T >$ are to denote the actual numbers of "HEAD" and "TAIL" results. Run your program for the following values of n : 2, 4, 6, 10, 50, 100, 1000, 5000, 50,000. Is it possible for this program to enter into an infinite loop? Explain!

7. (*Probability and Statistics*) Write a program similar to the one in the previous exercise except that it will not print the sentence, and it will have three output variables: P (as before), H = the number of heads, and T = the number of tails. Set up a loop to run this program with $n = 1000$ fixed for $k = 100$ times. Collect the outcomes of the variable H as a vector: $[h_0, h_1, h_2, \dots, h_{n+1}]$ (with $n + 1 = 1001$ entries) where each h_i denotes the number of times that the experiment resulted in having exactly h_i heads (so $H = h_i$) and then plot the graph of this vector (on the x -axis n runs from 0 to 1001 and on the y -axis we have the h_i -values). Repeat this exercise for $k = 200$ and then $k = 500$ times.
8. (*Probability: Random Integers*) Write a MATLAB function M-file, `randint(n,k)`, that has two input variables n and k being positive integers. There will be one output variable R , a vector with k components $R = [r_1, r_2, \dots, r_k]$, each of whose entries is a positive integer randomly selected from the list $\{1, 2, \dots, n\}$. (Each integer in this list has an equal chance of being generated at any time.)
9. (*Probability: Random Walks*) Create a MATLAB M-file called `ran2walk(n)` that simulates a random walk in the plane. The input n is the number of steps in the walk. The starting point of the walk is at the origin $(0, 0)$. At each step, random numbers are chosen (with uniform distribution) in the interval $[-1/2, 1/2]$ and are added to the present x - and y -coordinates to get the next x - and y -coordinates. The MATLAB command `rand` generates a random number in the interval $[0, 1]$, so we must subtract 0.5 from these to get the desired distributions. There will be no output variables, but MATLAB will produce a plot of the generated random walk.
Run this function for the values $n = 8, 25, 75, 250$ and (using the subplot option) put them all into a single figure. Repeat once again with the same values. In three dimensions, these random walks simulate the chaotic motion of a dust particle that makes many microscopic collisions and produces such strange motions. This is because the microscopic particles that collide with our particle are also in constant motion. We could easily modify our program by adding a third z -coordinate (and using `plot3(x,y,z)` instead of `plot(x,y)`) to make a program to simulate such three-dimensional random walks. Interestingly, each time you run the `ran2walk` function for a fixed value of n , the paths will be different. Try it out a few times. Do you notice any sort of qualitative properties about this motion? What are the chances (for a fixed n) that the path generated will cross itself? How about in three dimensions? Does the motion tend to move the particle away from where it started as n gets large? For these latter questions do not worry about proofs, but try to do enough experiments to lead you to make some educated hypotheses.
10. (*Probability Estimates by Simulation*) In each part, run a large number of simulations of the following experiments and take averages to estimate the indicated quantities.

(a) Continue to generate random numbers in $(0, 1)$ using `rand` until the accumulated sum exceeds 1. Let N denote the number of such random numbers that get added up when this sum first exceeds 1. Estimate the expected value of N , which can be thought of as the theoretical (long-run) average value of N if the experiment gets repeated indefinitely.

(b) Number a set of cards from 1 to 20, and shuffle them. Turn the cards over one by one and record the number of times K that card number i ($1 \leq i \leq 20$) occurs at (exactly) the i th draw. Estimate the expected value of K .

Note: Simulation is a very useful tool for obtaining estimates for quantities that can be impossible to estimate analytically; see [Ros-02] for a well-written introduction to this interesting subject. In it the reader can also find a rigorous definition of the expectation of a random variable associated with a (random) experiment. The quantities K and N above are examples of random variables. Their outcomes are numerical quantities associated with the outcomes of (random) experiments. Although the outcomes of random variables are somewhat unpredictable, their long-term averages do exhibit patterns that can be nicely characterized.

For the above two problems, the exact expectations are obtainable using methods of probability; they are $N = e$ and $K = 1$.

The next four exercises will revisit the Collatz conjecture that was introduced in the preceding section.

11. (*Number Theory: The Collatz Problem*) Write a function M-file, call it `collsz`, that takes as input a positive integer n (the first element for a Collatz experiment), and has as output a positive integer size equaling the size of the largest number in the Collatz iteration sequence before it reaches the value of 1. What is the first positive integer n for which this maximum size exceeds the value 100 ? 1000 ? 100,000 ? 1,000,000?
12. (*Number Theory: The Collatz Problem*) Modify the script file, `collat`, of Example 4.7 in the text to a new one, `collatzg`, that will interactively take the same input and internally construct the same vector a , but instead of producing output on the command window, it should produce a graphic of the vector a 's values versus the index of the vector. Arrange the plot to be done using blue pentagrams connected with lines. Run the script using the following inputs: 7, 15, 27, 137, 444, 657.

Note: The syntax for this plot style would be

```
plot (index, a, bp-).
```

13. (Number Theory: The Collatz Problem) If a Collatz experiment is started using a negative integer for $a(1)$, all experiments so far done by researchers have shown that the sequence will eventually cycle. However, in this case, there is more than one possible cycle. Write a script, *collatz2*, that will take an input for $a(1)$ in the same way as the script *collatz* in Example 4.7 did, and the script will continue to do the Collatz iteration until it detects a cycle. The output should include the number of iterations done before detecting a cycle as well as the actual cycle vector. Run your script using the following inputs: $-2, -4, -8, -10, -56, -88, -129$.

Suggestion: A cycle can be detected as soon as the same number $a(n)$ has appeared previously in the sequence. So your script will need to store the whole Collatz sequence. For example, each time it has constructed a new sequence element, say $a(20)$, the script should compare with the previous vector elements $a(1), a(20), \dots, a(19)$ to see if this new element has previously appeared. If not, the iteration goes on, but if there is a duplication, say, $a(20) = a(15)$, then there will be a cycle and the cycle vector would be $(a(15), a(16), a(17), a(18), a(19))$

14. (Number Theory: The Collatz Problem) Read first the preceding exercise. We consider two cycles as equivalent in a Collatz experiment if they contain the same numbers (but not necessarily in the same order). Thus the cycle $(1, 4, 2)$ has the equivalent forms $(4, 2, 1)$, and $(2, 1, 4)$. The program in the previous exercise, if encountering a certain cycle, may output any of the possible equivalent forms, depending on the first duplication encountered. We say that two cycles are essentially different if they are not equivalent cycles. In this exercise, you are to use MATLAB to help you figure out the number of essentially different Collatz cycles that come up from using negative integers for $a(1)$ ranging from -1 to $-20,000$.

Note: The Collatz conjecture can be paraphrased as saying that all Collatz iterations starting with a positive integer must eventually cycle and the resulting cycles are all equivalent to $(4, 2, 1)$. The essentially different Collatz cycles for negative integer inputs in this problem will cover all that are known to this date. It is also conjectured that there are no more.

Chapter 5

Floating Point Arithmetic and Error Analysis

5.1 FLOATING POINT NUMBERS

We have already mentioned that the data contained in just a single irrational real number such as π has more information in its digits than all the computers in the world could possibly ever store. Then again, it would probably take all the scientific surveyors in the world to look for and not be able to find any scientist who vitally needed, say, the 534th digit of this number. What is usually required in scientific work is to maintain accuracy with a certain number of so-called **significant digits**, which constitutes the portion of a numerical answer to a problem that is trusted to be correct. For example, if we want π to three significant digits, we could use 3.14. A computer can only work with a finite set of numbers; these computer numbers for a given system are usually called floating point numbers. Since there are infinitely many real numbers, what has to happen is that big (infinite) sets of real numbers must get identified with single computer numbers. Floating point numbers are best understood by their relations with numbers in scientific notation, such as 3.14159×10^0 , although they need not be written in this form.

A **floating point number system** is determined by a **base** β (any positive integer greater than one), a precision s (any positive integer; this will be the number of significant digits), and two integers m (negative) and M (positive), that determine the exponent range. In such a system, a floating point number can always be expressed in the form:

$$\pm d_1 d_2 \cdots d_s \times \beta^e,$$

where,

$$d_i = 0, 1, 2, \dots, \text{ or } \beta - 1 \text{ but } d_1 \neq 0 \text{ and } m \leq e \leq M.$$

The number zero is represented as $.00 \cdots 0 \times \beta^{-m}$. In a computer, each of the three parts (the sign \pm , mantissa $d_1 d_2 \cdots d_s$, and the exponent β) of a floating point number is stored in its own separate fixed-width field. Most contemporary computers and software on the market today (MATLAB included) use **binary arithmetic** ($\beta = 2$). Hand-held calculators use decimal base $\beta = 10$. In the past, other computers have used different bases that were usually powers of two, such as $\beta = 16$ (hexadecimal arithmetic). Of course, such arithmetic (different from base 10) is done in internal calculations only. When the number is displayed, it is converted to decimal form. An important quantity for determining the precision of a given computing system is known as the **unit roundoff** u (or the **umachine epsilon**¹), which is the maximum relative error that can occur when a real number is approximated by a floating point number.² For example, one Texas Instruments graphing calculator uses the floating point parameters $\beta = 10, s = 12, m = -99$, and $M = 99$, which means that this calculator can effectively handle numbers whose absolute values lie (approximately) between 10^{-99} and 10^{99} , and the unit roundoff is $u = 10^{-12}$. MATLAB's arithmetic uses the parameters: $\beta = 2, s = 53, m = -1074$, and $M = 1023$, which conforms to the **IEEE double precision standard**.³ This means that MATLAB can effectively handle numbers with absolute values from $2^{-1074} \approx 10^{-324}$ to $2^{1023} \approx 10^{308}$; also the unit roundoff is $u = 2^{-53} \approx 10^{-16}$.

5.2 FLOATING POINT ARITHMETIC: THE BASICS

Many students have gotten accustomed to the reliability and logical precision of exact mathematical arithmetic. When we get the computer to perform calculations for us, we must be aware that floating point arithmetic compounded with

¹The rationale for this terminology is that the Greek letter epsilon (ϵ) is usually used in mathematical analysis to represent a very small number.

²There is another characterization of the unit roundoff as the gap between the floating point number 1 and the next floating point number to the right. These two definitions are close, but not quite equivalent; see Example 5.4 and Exercise for the Reader 5.3 for more details on how these two quantities are related, as well as explicit formulas for the unit roundoff.

³The IEEE (I-triple-E) is a nonprofit, technical professional association of more than 350,000 individual members in 150 countries. The full name is the Institute of Electrical and Electronics Engineers, Inc. The IEEE single-precision (SP) and double-precision (DP) standards have become the international standard for computers and numerical software. The standards were carefully developed to help avoid some problems and incompatibilities with previous floating point systems. In our notation, the IEEE SP standard specifies $\beta = 2, s = 24, m = -126, M = 127$ and the IEEE DP standard has $\beta = 2, s = 53, m = -1022, M = 1023$.

roundoff errors can lead to unexpected and undesirable results. Most large-scale numerical algorithms are not exact algorithms, and when such methods are used, attention must be paid to the error estimates. We saw this at a basic level in Chapter 2, and it will reappear again later on several occasions. Here we will talk about different sorts of errors, namely, those that arise and are compounded by the computer's floating point arithmetic. We stress the distinction with the first type of errors. Even if an algorithm is mathematically guaranteed to work, floating point errors may arise and spoil its success. All of our illustrations below will be in base 10 floating point arithmetic, since all of the concepts can be covered and better understood in this familiar setting; changing to a different base is merely a technical issue. To get a feel for the structure of a floating point number system, we begin with an example of a very small system.

EXAMPLE 5.1: Find all floating point numbers in the system with $\beta = 10, s = 1, m = -1$, and $M = 1$.

SOLUTION: In this case, it is a simple matter to write down all of the floating point numbers in the system:

$$\begin{array}{lll} \pm.1 \times 10^{-1} = \pm.01 & \pm.1 \times 10^0 = \pm.1 & \pm.1 \times 10^1 = \pm 1 \\ \pm.2 \times 10^{-1} = \pm.02 & \pm.2 \times 10^0 = \pm.2 & \pm.2 \times 10^1 = \pm 2 \\ \vdots & \vdots & \vdots \\ \pm.9 \times 10^{-1} = \pm.09 & \pm.9 \times 10^0 = \pm.9 & \pm.9 \times 10^1 = \pm 9. \end{array}$$

Apart from these, there is only $0 = .0 \times 10^{-1}$. Of these 55 numbers, the nonnegative ones are pictured on the number line in Figure 5.1. We stress that the gaps between adjacent floating point numbers are not always the same; in general, these gaps become smaller near zero and more spread out far away from zero (larger numbers).

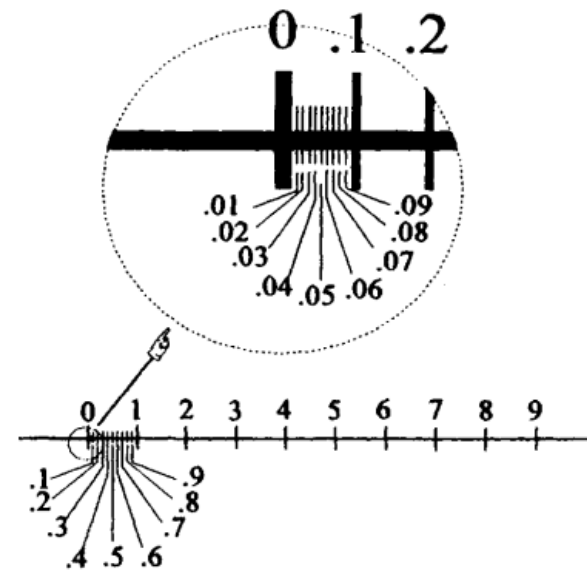


Figure 5.1: The nonnegative floating point numbers of Example 5.1. The omitted negative floating point numbers are just (in any floating point number system) the opposites of the positive numbers shown. The situation is typical in that as we approach zero, the density of floating point numbers increases.

Let us now talk about how real numbers get converted to floating point numbers. Any real number x can be expressed in the form

$$x = \pm d_1 d_2 \cdots d_s d_{s+1} \cdots \times 10^e$$

where there are infinitely many digits (this is the only difference from the floating point representation (1) with $\beta = 10$) and there is no restriction on the exponent's range. The part $d_1 d_2 \cdots d_s d_{s+1} \cdots$ is called the mantissa of x . If x has a finite decimal expansion, we can trail it with an infinite string of zeros to conform with (2). In fact, the representation (2) is unique for any real number (i.e., there is only one way to represent any x in this way) provided that we adopt the convention that an infinite string of 9's not be allowed; such expansions should just be rounded up. (For example, the real number .3799999999... is the same as .38.) At first glance, it may seem straightforward how to represent a real number x in form (2) by a floating point number of form (1) with $\beta = 10$; simply either chop off or round off any of the digits past the allowed number. But there are serious problems that may arise, stemming from the fact that the exponent e of the real number may be outside the permitted range. Firstly, if $e > M$, this means that x is (probably much) larger than any floating point number and so cannot be represented by one. If such a number were to come up in a computation, the computation is said to have overflowed. For example, in the simple setting of Example 5.1, any number $x \geq 10$ would overflow this simple floating point system. Depending on the computer system, overflows will usually result in termination of calculation or a warning. For example, most graphing calculators, when asked to evaluate an expression like e^{5000} , will either produce an error message like "OVERFLOW", and freeze up or perhaps give ∞ as an output. MATLAB behaves similarly to the latter pattern for overflows:

```
>>exp(5000)
->ans = Inf %MATLAB's abbreviation for infinity.
```

Inf (or inf) is MATLAB's way of saying the number is too large to continue to do any more number crunching, except for calculations where the answer will be either Inf or - Inf (a very large negative number). Here are some examples:

```
>> exp(5000)
->ans = Inf % MATLAB tells us we have a very big positive number here
>> 2*exp(5000)
->ans = Inf %No new information
>> exp(5000)/-5
->ans = -Inf %OK now we have a very big negative number.
>> 2*exp(5000)-exp(5000)
->ans = NaN % "NaN" stands for "not a number"
```

The last calculation is more interesting. Obviously, the expression being evaluated is just e^{5000} , which, when evaluated separately, is outputted as Inf. What happens is that MATLAB tries instead to do inf-inf, which is undefined (once large numbers are converted to inf, their relative sizes are lost and it is no longer possible for MATLAB to compare them).⁴

A very different situation occurs if the exponent e of the real number x in (2) is less than m (too small). This means that the real number x has absolute value (usually much) smaller than that of any nonzero floating point number. In this case, a computation is said to have underflowed. Most systems will represent an underflow by zero without any warning, and this is what MATLAB does.

Underflows, although less serious than overflows, can be a great source of problems in large-scale numerical calculations. Here is a simple example. We know from basic rules of exponents that $e^p e^{-p} = e^{p-p} = e^0 = 1$, but consider the following calculation:

```
>> exp(-5000)
->ans = 0 %this very small number has underflowed to zero
>> exp(5000)*exp(-5000)
->ans = NaN
```

The latter calculation had both underflows and overflows and resulted in $0 \star \text{Inf}$ ($= 0 \cdot \infty$), which is undefined. We will give another example shortly of some of the tragedies that can occur as the result of underflows. But now we show two simple ways to convert a real number to a floating point number in case there is no overflow or underflow. So we assume the real number x in (2) has exponent e satisfying $m \leq e \leq M$. The two methods for converting the real number x to its floating point representative $\text{fl}(x)$ are as follows:

(i) Chopped (or Truncated) Arithmetic: With this system we simply drop all digits after d_s :

$$\text{fl}(x) = \text{fl}(\pm .d_1 d_2 \cdots d_s d_{s+1} \cdots \times 10^e) = \pm d_1 d_2 \cdots d_s \times 10^e$$

(ii) Rounded Arithmetic: Here we do the usual rounding scheme for the first s significant digits. If $d_{s+1} < 5$, we simply chop as in method (i), but if $d_{s+1} \geq 5$, we need to round up. This may change several digits depending on if there is a string of 9's or not. For example, with $s = 4$, $\dots 2456823 \dots$ would round to $.2457$ (onedigit changed), but $.2999823$ would round to $.3000$ (four digits changed). So a nice formula as in (i) is not possible. There is, however, an elegant way to describe rounded arithmetic in terms of chopped arithmetic using two steps.

Step 1: Add $5 \times 10^{-(s+1)}$ to the mantissa $d_1 d_2 \cdots d_s d_{s+1} \cdots$ of x .

Step 2: Now chop as in (i) and retain the sign of x .

EXAMPLE 5.2: The following example parallels some calculations in exact arithmetic with the same calculations in 3-digit floating point arithmetic with $m = -8$ and $M = 8$. The reader is encouraged to go through both sets of calculations, using either MATLAB or a calculator. Note that at each point in a floating point calculation, the numbers need to be chopped accordingly before any math operations can be done.

Exact Arithmetic	Floating Point Arithmetic
$x = \sqrt{3}$	$\text{fl}(x) = 1.73 \ (\equiv .173 \times 10^1)$
$x^2 = 3$	$\text{fl}(x)^2 = 2.99$

Thus, in floating point arithmetic, we get that $\sqrt{3}^2 = 2.99$. This error is small but understandable.

The same calculation with larger numbers, of course, results in a larger error; but relatively it is not much different. A series of small errors can pile up and amount to more catastrophic results, as the next calculations show.

⁴We mention that the optional "Symbolic Toolbox" for MATLAB allows, among other things, the symbolic manipulation of such expressions. The Symbolic Toolbox does come with the student version of MATLAB. Some of its features are explained in Appendix A.

Exact Arithmetic	Floating Point Arithmetic
$x = 1000$	$fl(x) = 1000$
$y = 1/x = .001$	$fl(y) = .001$
$z = 1 + y = 1.001$	$fl(z) = 1$
$w = (z - 1) \cdot x^2$	$fl(w) = (1 - 1) \cdot 1000^2$
$= y \cdot x^2$	$= 0 \cdot 1000^2$
$= \frac{1}{x} \cdot x^2$	$= 0$
$= x = 1000$	

The floating point answer of 0 is a ridiculous approximation to the exact answer of 1000 ! The reason for this tragedy was the conversion of an underflow to zero. By themselves, such conversions are rather innocuous, but when coupled with a sequence of other operations, problematic results can sometimes occur.

The floating point answer of 0 is a ridiculous approximation to the exact answer of 1000 ! The reason for this tragedy was the conversion of an underflow to zero. By themselves, such conversions are rather innocuous, but when coupled with a sequence of other operations, problematic results can sometimes occur.

When we do not make explicit mention of the exponent range $m \leq e \leq M$, we assume that the numbers that come up have their exponents in the appropriate range and so there will be no underflows or overflows.

EXERCISE FOR THE READER 5.1: Perform the following calculations in twodigit rounded arithmetic, and compare with the exact answers.

- (a) $(.15)^2$
- (b) $365,346 \times .4516$
- (c) $8001 \div 123$

Our next example should alert the reader that one needs to be cautious on many different fronts when using floating point arithmetic. Many arithmetic rules that we have become accustomed to take for granted sometimes need to be paid careful attention when using floating point arithmetic.

EXAMPLE 5.3: Working in three-digit chopped floating point arithmetic with the exponent e restricted to the range $-8 \leq e \leq 8$, perform the following tasks: (a) Compute the infinite series: $\sum_{n=1}^{\infty} \frac{1}{n^2} = 1 + \frac{1}{4} + \frac{1}{9} + \dots$

(b) In each part below an equation is given and your task will be to decide how many solutions it will have in this floating point arithmetic. For each part you should give one of these four answers: **NO SOLUTION, EXACTLY ONE SOLUTION, BETWEEN 2 AND 10 SOLUTIONS, or MORE THAN 10 SOLUTIONS**, (Work here only with real numbers; take all underflows as zero.)

- (i) $3x = 5$
- (ii) $x^3 = 0$

SOLUTION: Part (a): Unlike with exact arithmetic, when we sum this infinite series in floating point arithmetic, it is really going to be a finite summation since eventually the terms will be getting too small to have any effect on the accumulated sum. We use the notation $S_N = \sum_{n=1}^N \frac{1}{n^2} = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \dots + \frac{1}{N^2}$ for the partial sum (a finite sum). To find the infinite sum, we need to calculate (in order) in floating point arithmetic S_1, S_2, S_3, \dots and continue until these partial sums no longer change. Here are the step-by-step details:

$$\begin{aligned}
 S_1 &= 1 \\
 S_2 &= S_1 + 1/4 = 1 + .25 = 1.25 \\
 S_3 &= S_2 + 1/9 = 1.25 + .111 = 1.36 \\
 S_4 &= S_3 + 1/16 = 1.36 + .0625 = 1.42 \\
 S_5 &= S_4 + 1/25 = 1.42 + .040 = 1.46 \\
 S_6 &= S_5 + 1/36 = 1.46 + .0277 = 1.48 \\
 S_7 &= S_6 + 1/49 = 1.48 + .0204 = 1.50 \\
 S_8 &= S_7 + 1/64 = 1.50 + .0156 = 1.51 \\
 S_9 &= S_8 + 1/81 = 1.51 + .0123 = 1.52 \\
 S_{10} &= S_9 + 1/100 = 1.52 + .010 = 1.53 \\
 S_{11} &= S_{10} + 1/121 = 1.53 + .00826 = 1.53
 \end{aligned}$$

We can now stop this infinite process since the terms being added are small enough that when added to the existing partial sum 1.53, their contributions will just get chopped. Thus in the floating point arithmetic of this example, we have computed $\sum_{n=1}^{\infty} \frac{1}{n^2} = 1.53$, or more correctly we should write $fl\left(\sum_{n=1}^{\infty} \frac{1}{n^2}\right) = 1.53$. Compare this result with the result from exact arithmetic $\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6} = 1.64\dots$. Thus in this calculation we were left with only one significant digit of accuracy!

Part (b): (i) The equation $3x = 5$ has, in exact arithmetic, only one solution, $x = 5/3 = 1.666\dots$. Let's look at the candidates for floating point arithmetic solutions that are in our system. This exact solution has floating point representative 1.66. Checking this in the equation (now working in floating point arithmetic) leads to: $3 \cdot 1.66 = 4.98 \neq 5$. So this will not be a floating point solution. Let's try making the number a bit bigger to 1.67 (this would be the smallest possible jump

to the next floating point number in our system). We have (in floating point arithmetic) $3 \cdot 1.67 = 5.01 \neq 5$, so here $3x$ is too large. If these two numbers do not work, no other floating point numbers can (since for other floating point numbers $3x$ would be either less than or equal to 4.98 or greater than or equal to 5.01). Thus we have "NO SOLUTION" to this equation in floating point arithmetic! ⁵

(ii) As in (i), the equation $x^3 = 0$ has exactly one real number solution, namely $x = 0$. This solution is also a floating point solution. But there are many, many others. The lower bound on the exponent range $-8 \leq e$ is relevant here. Indeed, take any floating point number whose magnitude is less than 10^{-3} , for example, $x = .0006385$. Then $x^3 = (.0006385)^3 = 2.60305 \dots \times 10^{-10} = .260305 \times 10^{-9}$ (in exact arithmetic). In floating point arithmetic, this computation would underflow and hence produce the result $x^3 = 0$. We conclude that in floating point arithmetic, this equation has "MORE THAN 10 SOLUTIONS" (see also Exercise 10 of this section).

EXERCISE FOR THE READER 5.2: Working two-digit rounded floating point arithmetic with the exponent e restricted to the range $-8 \leq e \leq 8$, perform the following tasks:

(a) Compute the infinite series: $\sum_{n=1}^{\infty} \frac{1}{n} = 1 + \frac{1}{2} + \frac{1}{3} + \dots$

(b) In each part below an equation is given and your task will be to decide how many solutions it will have in this floating point arithmetic. For each part you should give one of these four answers: NO SOLUTION, EXACTLY ONE SOLUTION, BETWEEN 2 AND 10 SOLUTIONS, or MORE THAN 10 SOLUTIONS. (Work here only with real numbers; take all underflows as zero.)

(i) $x^2 = 100$

(ii) $8x^2 = x^5$

EXERCISES 5.2:

- In three-digit chopped floating point arithmetic, perform the following operations with these numbers: $a = 10000$, $b = .05$, and $c = 1/3$.
 - Write c as a floating point number, i.e., find $\text{fl}(c)$.
 - Find $a + b$.
 - Solve the equation $ax = c$ for x .
- In three-digit rounded floating point arithmetic, perform the following tasks:
 - Find $1.23 + .456$
 - Find $110,000 - 999$
 - Find $(.055)^2$
- In three-digit chopped floating point arithmetic, perform the following tasks:
 - Solve the equation $5x + 8 = 0$.
 - Use the quadratic formula to solve $1.12x^2 + 88x + 1 = 0$.
 - Compute $\sum_{n=1}^{\infty} \frac{1}{n^4} = \frac{1}{1^4} + \frac{1}{2^4} + \frac{1}{3^4} + \dots$.
- In three-digit rounded floating point arithmetic, perform the following tasks:
 - Solve the equation $5x + 4 = 17$.
 - Use the quadratic formula to solve $x^2 - 2.2x + 3 = 0$.
 - Compute $\sum_{n=1}^{\infty} \frac{(-1)^n \cdot 10}{n^4 + 2} = \frac{-10}{1^4 + 2} + \frac{10}{2^4 + 2} - \frac{10}{3^4 + 2} + \dots$.
- In each part below an equation is given and your task will be to decide how many solutions it will have in 3-digit chopped floating point arithmetic. For each part you should give one of these four answers: NO SOLUTION, EXACTLY ONE SOLUTION, BETWEEN 2 AND 10 SOLUTIONS, or MORE THAN 10 SOLUTIONS. (Work here only with real numbers with exponent e restricted to the range $-8 \leq e \leq 8$, and take all underflows as zero.)
 - $2x + 7 = 16$
 - $(x + 5)^2(x + 1/3) = 0$
 - $2^x = 20$
- Repeat the directions of Exercise 5, for the following equations, this time using 3-digit rounded floating point arithmetic with exponent e restricted to the range $-8 \leq e \leq 8$.
 - $2x + 7 = 16$
 - $x^2 - x = 6$
 - $\sin(x^2) = 0$
- Using three-digit chopped floating point arithmetic (in base 10), do the following:
 - Compute the sum: $1 + 8 + 27 + 64 + 125 + 216 + 343 + 512 + 729 + 1000 + 1331$, then find the relative error of this floating point answer with the exact arithmetic answer.

⁵We point out that when asked to (numerically) solve this equation in floating point arithmetic, we would simply use the usual (pure) mathematical method but work in floating point arithmetic, i.e., divide both sides by 3. The question of how many solutions there are in floating point arithmetic is a more academic one to help highlight the differences between exact and floating point arithmetic. Indeed, any time one uses a calculator or any floating point arithmetic software to solve any sort of mathematical problem with an exact mathematical method we should be mindful of the fact that the calculation will be done in floating point arithmetic.

- (b) Compute the sum in part (a) in the reverse order, and again find the relative answer of this floating point answer with the exact arithmetic answer.
 (c) If you got different answers in parts (a) and (b), can you explain the discrepancy?

8. Working in two-digit chopped floating point arithmetic, compute the infinite series $\sum_{n=1}^{\infty} \frac{1}{n}$.
 9. Working in two-digit rounded floating point arithmetic, compute the infinite series

$$\sum_{n=2}^{\infty} \frac{1}{n^{3/2} \ln n}.$$

10. In the setting of Example 5.3(b) (ii), exactly how many floating point solutions are there for the equation $x^3 = 0$?
 11. (a) Write a MATLAB function M-file $z = rfloatadd(x, y, s)$ that has inputs x and y being any two real numbers, a positive integer s , and the output z will be the sum $x + y$ using s -digit rounded floating point arithmetic. The integer s should not be more than 14 so as not to transcend MATLAB's default floating point accuracy
 (b) Use this program (perhaps in conjunction with loops) to redo Exercise for the Reader 5.2, and Exercise 9.
 12. (a) Write a MATLAB function M-file $z = cfloatadd(x, y, s)$ that has inputs x and y being any two real numbers, a positive integer s , and the output z will be the sum $x + y$ using s -digit chopped floating point arithmetic. The integer s should not be more than 14 so as not to transcend MATLAB's default floating point accuracy.
 (b) Use this program (perhaps in conjunction with loops) to redo Example 5.3(a), and Exercise 7 .
 13. (a) How many floating point numbers are there in the system with $\beta = 10, s = 2, m = -2, M = 2$? What is the smallest real number that would cause an overflow in this system?
 (b) How many floating point numbers are there in the system with $\beta = 10, s = 3, m = -3, M = 3$? What is the smallest real number that would cause an overflow in this system?
 (c) Find a formula that depends on s, m , and M that gives the number of floating point numbers in a general base 10 floating point number system ($\beta = 10$). What is the smallest real number that would cause an overflow in this system?

NOTE: In the same fashion as we had with base 10 , for any base $\beta > 1$, any nonzero real number x can be expressed in the form:

$$x = \pm d_1 d_2 \cdots d_s d_{s+1} \cdots \times \beta^e$$

where there are infinitely many digits $d_i = 0, 1, \dots, \beta - 1$, and $d_1 \neq 0$. This notation means the following infinite series:

$$x = \pm (d_1 \times \beta^{-1} + d_2 \times \beta^{-2} + \cdots + d_s \times \beta^{-s} + d_{s+1} \times \beta^{-s-1} + \cdots) \times \beta^e.$$

To represent any nonzero real number with its **base β** expansion, we first would determine the exponent e so that the inequality $1/\beta \leq |x|/\beta^e < 1$ is valid. Next we construct the "digits" in order to be as large as possible so that the cumulative sum multiplied by β^e does not exceed $|x|$. As an example, we show here how to get the binary expansions ($\beta = 2$) of each of the numbers $x = 3$ and $x = 1/3$. For $x = 3$, we get first the exponent $e = 2$, since $1/2 \leq |3|/2^2 < 1$. Since $(1 \times 2^{-1}) \times 2^2 = 2 < 3$, the first digit d_1 is 1 (in binary arithmetic, the digits can only be zeros or ones).

The second digit d_2 is also 1 since the cumulative sum is now

$$(1 \times 2^{-1} + 1 \times 2^{-2}) \times 2^2 = 2 + 1 = 3.$$

Since the cumulative sum has now reached $x = 3$, all remaining digits are zero, and we have the binary expansion of $x = 3$:

$$3 = .1100 \cdots 00 \cdots \times 2^2$$

Proceeding in the same fashion for $x = 1/3$, we first determine the exponent e to be -1 (since $1/3/2^{-1} = 2/3$ lies in $[1/2, 1)$). We then find the first digit $d_1 = 1$, and cumulative sum is $(1 \times 2^{-1}) \times 2^{-1} = 1/4 < 1/3$. Since $(1 \times 2^{-1} + 1 \times 2^{-2}) \times 2^{-1} = 3/8 > 1/3$, we see that the second digit $d_2 = 0$. Moving along, we get that $d_3 = 1$ and the cumulative sum is

$$(1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}) \times 2^{-1} = 5/16 < 1/3$$

Continuing in this fashion, we will find that $d_4 = d_6 = \cdots = d_{2n} = 0$, and $d_5 = d_7 = \cdots = d_{2n+1} = 1$ and so we obtain the binary expansion:

$$1/3 = .101010 \cdots 1010 \cdots \times 2^{-1}$$

If we require that there be no infinite string of ones (the construction process given above will guarantee this), then these expansions are unique. Exercises 14-19 deal with such representations in nondecimal bases ($\beta \neq 10$).

14. (a) Find the binary expansions of the following real numbers: $x = 1000, x = -2, x = 2.5$.
 (b) Find the binary expansions of the following real numbers: $x = 5/32, x = 2/3, x = 1/5, x = -0.3, x = 1/7$.
 (c) Find the exponent e and the first 5 digits of the binary expansion of π .
 (d) Find the real numbers with the following (terminating) binary expansions: $1010 \cdots 00 \cdots \times 2^8, .1110 \cdots 00 \cdots \times 2^{-3}$.
15. (a) Use geometric series to verify the binary expansion of $1/3$ that was obtained in the previous note.
 (b) Use geometric series to find the real numbers having the following binary expansions:
 $.10010101 \cdots \times 2^1, .11011011 \cdots \times 2^0, .1100011011011 \cdots \times 2^1$.
- (c) What sort of real numbers will have binary expansions that either end in a sequence of zeros, or repeat, like the one for $1/3$ obtained in the note preceding Exercise 14?
16. (a) Write down all floating point numbers in the system with $\beta = 2, s = 1, m = -1, M = 1$. What is the smallest real number that would cause an overflow in this system?
 (b) Write down all floating point numbers in the system with $\beta = 2, s = 2, m = -1, M = 1$. What is the smallest real number that would cause an overflow in this system?
 (c) Write down all floating point numbers in the system with $\beta = 3, s = 2, m = -1, M = 1$. What is the smallest real number that would cause an overflow in this system?
17. (a) How many floating point numbers are there in the system with $\beta = 2, s = 3, m = -2, M = 2$? What is the smallest real number that would cause an overflow in this system?
 (b) How many floating point numbers are there in the system with $\beta = 2, s = 2, m = -3, M = 3$? What is the smallest real number that would cause an overflow in this system?
 (c) Find a formula that depends on s, m , and M that gives the number of floating point numbers in a general binary floating point number system ($\beta = 2$). What is the smallest real number that would cause an overflow in this system?
18. Repeat each part of Exercise 17, this time using base $\beta = 3$.
19. Chopped arithmetic is defined in arbitrary bases exactly the same as was explained for decimal bases in the text. Real numbers must first be converted to their expansion in base β . For rounded floating point arithmetic using s -digits with base β , we simply add $\beta^{-s}/2$ to the mantissa and then chop. Perform the following floating point additions by first converting the numbers to floating point numbers in base $\beta = 2$, doing the operation in two-digit chopped arithmetic, and then converting back to real numbers. Note that your real numbers may have more digits in them than the number s used in base 2 arithmetic, after conversion.
 (a) $2 + 6$
 (b) $22 + 7$
 (c) $120 + 66$

5.3 FLOATING POINT ARITHMETIC: FURTHER EXAMPLES AND DETAILS

In order to facilitate further discussion on the differences in floating point and exact arithmetic, we introduce the following notation for operations in floating point arithmetic:

$$x \oplus y \equiv \text{fl}(x + y)$$

$$x \ominus y \equiv \text{fl}(x - y)$$

$$x \otimes y \equiv \text{fl}(x \cdot y)$$

$$x \oslash y \equiv \text{fl}(x \div y)$$

(i.e., we put circles around the standard arithmetic operators to represent the corresponding floating point operations). To better illustrate concepts and subtleties of floating point arithmetic without getting into technicalities with different bases, we continue to work only in base $\beta = 10$.

In general, as we have seen, floating point operations can lead to different answers than exact arithmetic operations. In order to track and predict such errors, we first look, in the next example, at the relative error introduced when a real number is approximated by its floating point number representative.

EXAMPLE 5.4: Show that in s -digit chopped floating point arithmetic, the unit roundoff u is 10^{1-s} , and that this number equals the distance from one to the next (larger) floating point number. We recall that the unit roundoff is defined to be the maximum relative error that can occur when a real number is approximated by a floating point number.

SOLUTION: Since $\text{fl}(0) = 0$, we may assume that $x \neq 0$. Using the representations (1) and (2) for the floating point and exact numbers, we can estimate the relative error as follows:

$$\begin{aligned} \left| \frac{x - \text{fl}(x)}{x} \right| &= \left| \frac{d_1 d_2 \cdots d_s d_{s+1} \cdots \times 10^e - d_1 d_2 \cdots d_s \times 10^e}{d_1 d_2 \cdots d_s d_{s+1} \cdots \times 10^e} \right| \quad (s) \text{ slot} \\ &= \left| \frac{00 \cdots 0 d_{s+1} d_{s+2} \cdots \times 10^e}{d_1 d_2 \cdots d_{s+1} d_{s+2} \cdots \times 10^e} \right| \leq \frac{.00 \cdots 099 \cdots}{.10 \cdots 000 \cdots} \\ &= \frac{.00 \cdots 100 \cdots}{10 \cdots 000 \cdots} = \frac{10^{-s}}{10^{-1}} = 10^{1-s} \end{aligned}$$

Since equality can occur, this proves that the number on the right side is the unit roundoff. To see that this number u coincides with the gap between the floating point number 1 and the next (larger) floating point number on the right, we write the number 1 in the form (1):

$$1 = .10 \cdots 00 \times 10^1,$$

(note there are s digits total on the right, $d_1 = 1$, and $d_2 = d_3 = \cdots = d_s = 0$); we see that the next larger floating point number of this form will be:

$$1 + \text{gap} = .10 \cdots 01 \times 10^1.$$

Subtracting gives us the unit roundoff:

$$\text{gap} = .00 \cdots 01 \times 10^1 = 10^{-s} \times 10^1 = 10^{1-s},$$

as was claimed.

EXERCISE FOR THE READER 5.3: (a) Show that in s -digit rounded floating point arithmetic the unit roundoff is $u = \frac{1}{2} 10^{1-s}$, but that the gap from 1 to the next floating point number is still 10^{1-s} .

(b) Show also that in any floating point arithmetic system and for any real number x , we can write

$$\text{fl}(x) = x(1 + \delta), \text{ where } |\delta| \leq u$$

In relation to (4), we also assume that for any single floating point arithmetic operation: $x \odot y$, with \odot representing any of the floating point arithmetic operations from (3), we can write

$$x \odot y = (x \circ y)(1 + \delta), \text{ where } |\delta| \leq u$$

and where \circ denotes the exact arithmetic operation corresponding to \odot . This assumption turns out to be valid for IEEE (and hence, MATLAB's) arithmetic but for other computing environments may require that the bound on δ be replaced by a small multiple of u . We point out that IEEE standards require that $x \odot y = \text{fl}(x \circ y)$.

In scientific computing, we will often need to do a large number of calculations and the resulting roundoff (or floating point) errors can accumulate. Before we can trust the outputs we get from a computer on an extensive computation, we have to have some confidence of its accuracy to the true answer. There are two major types of errors that can arise: round-off errors and algorithmic errors. The first results from propagation of floating point errors, and the second arises from mathematical errors in the model used to approximate the true answer to a problem. To decrease mathematical errors, we will need to do more computations, but more computations will increase computer time and roundoff errors. This is a major dilemma of scientific computing! The best strategy and ultimate goal is to try to find efficient algorithms; this point will be applied and reemphasized frequently in the sequel.

To illustrate roundoff errors, we first look at the problem of numerically adding up a set of positive numbers. Our next example will illustrate the following general principle:

A General Principle of Floating Point Arithmetic: When numerically computing a large sum of positive numbers, it is best to start with the smallest number and add in increasing order of magnitude.

Roughly, the reason the principle is valid is that if we start adding the large numbers first, we could build up a rather large cumulative sum. Thus, when we get to adding to this sum some of the smaller numbers, there are much better chances that all or parts of these smaller numbers will have decimals beyond the number of significant digits supported and hence will be lost or corrupted.

EXAMPLE 5.5: (a) In exact mathematics, addition is associative: $(x + y) + z = x + (y + z)$. Show that in floating point arithmetic, addition is no longer associative.

REMARK: Formula (6), although quite complicated, can be seen to demonstrate the above principle. Remember that u is an extremely small number so that the error on the right will normally be small as long as there are not an inordinate number of terms being added. In any case, the formula makes it clear that the relative contribution of the first term being added is the largest since the error estimate (right side of (6)) is a sum of terms corresponding to each a_i multiplied by the proportionality factor $(N - i)u$. Thus if we are adding $N = 1,000,001$ terms then this proportionality factor is $1,000,000u$.

(worst) for a_1 but only u (best) for a_N , and these factors decrease linearly for intermediate terms. Thus it is clear that we should start adding the smaller terms first, and save the larger ones for the end.

SOLUTION: Part (a): We need to find (in some floating point arithmetic system) three numbers x, y , and z such that $(x \oplus y) \oplus z \neq x \oplus (y \oplus z)$. Here is a simple example that also demonstrates the above principle: We use 2 -digit chopped arithmetic with $x = 1, y = z = .05$. Then,

$$(x \oplus y) \oplus z = (1 \oplus .05) \oplus .05 = 1 \oplus .05 = 1,$$

but

$$x \oplus (y \oplus z) = 1 \oplus (.05 \oplus .05) = 1 \oplus .1 = 1.1..$$

This not only provides a counterexample, but since the latter computation (gotten by adding the smaller numbers first) gave the correct answer it also demonstrates the above principle.

Part (b): We continue to use the notation for partial sums that was employed in Example 5.3 (i.e., $S_1 = a_1, S_2 = a_1 + a_2, S_3 = a_1 + a_2 + a_3$, etc.). By using identity (5) repeatedly, we have:

$$\text{fl}(S_2) = a_1 \oplus a_2 = (a_1 + a_2)(1 + \delta_2) = S_2 + (a_1 + a_2)\delta_2, \text{ where } |\delta_2| \leq u, \text{ and so}$$

$$\begin{aligned} \text{fl}(S_3) &= \text{fl}(S_2) \oplus a_3 = (\text{fl}(S_2) + a_3)(1 + \delta_3) \text{ where } |\delta_3| \leq u \\ &= (S_2 + (a_1 + a_2)\delta_2 + a_3)(1 + \delta_3) \\ &= S_3 + (a_1 + a_2)\delta_2 + (a_1 + a_2 + a_3)\delta_3 + (a_1 + a_2)\delta_2\delta_3 \\ &\approx S_3 + (a_1 + a_2)\delta_2 + (a_1 + a_2 + a_3)\delta_3. \end{aligned}$$

To get to the last estimate, we ignored the higher-order (last) term of the second-to-last expression. Continuing to estimate in the same fashion leads us to

$$\text{fl}(S_N) \approx S_N + u \begin{cases} a_1(\delta_2 + \delta_3 + \delta_4 + \cdots + \delta_N) \\ + a_2(\delta_2 + \delta_3 + \delta_4 + \cdots + \delta_N) \\ + a_3(\delta_3 + \delta_4 + \cdots + \delta_N) \\ + a_4(\delta_4 + \cdots + \delta_N) \\ \vdots \\ + a_N(\delta_N) \end{cases}$$

where each of the δ_i 's arise from application of (5) and thus satisfy $|\delta_i| \leq u$. Bounding each of the $|\delta_i|$'s above with u and using the triangle inequality produces the asserted error bound in (6).

EXERCISE FOR THE READER 5.4: From estimate (6) deduce the following (cleaner but weaker) error estimates for the roundoff error in performing a finite summation of positive numbers $S_N = a_1 + a_2 + \cdots + a_N$ in floating point arithmetic:

$$(a) \text{ Error} = |\text{fl}(S_N) - S_N| \leq Nu \sum_{n=1}^N a_n.$$

Next, we give some specific examples that will compare these estimates against the actual roundoff errors. Recall from calculus that an (infinite) p -series $\sum_{n=1}^{\infty} \frac{1}{n^p} = 1 + \frac{1}{2^p} + \frac{1}{3^p} + \cdots$ converges (i.e., adds up to a finite number) exactly when $p > 1$, otherwise it diverges (i.e., adds up to infinity). If we ask MATLAB (or any floating point computing system) to add up the terms in any p -series (or any series with positive terms that decrease to zero), eventually the terms will get too small to make any difference when they are added to the accumulated sum, so it will eventually appear that the series has converged (if the computer is given enough time to finish its task).⁶ Thus, it is not possible to detect divergence of such a series by asking the computer to perform the summation. Once a series is determined to converge, however, it is possible to get MATLAB to help us estimate the actual sum of the infinite series. The key question in such a problem is to determine how many terms need to be summed in order for the partial sums to approximate the actual sum within the desired tolerance for error. We begin with an example for which the actual sum of the infinite series is known. This will allow us to verify if our accuracy goal is met.

EXAMPLE 5.6: Consider the infinite p -series $\sum_{n=1}^{\infty} \frac{1}{n^2} = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \cdots$. Since $p = 2 > 1$, the series converges to a finite sum S

(a) How many terms N of this infinite sum would we need to sum up to so that the corresponding partial sum $\sum_{n=1}^N \frac{1}{n^2} = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \cdots + \frac{1}{N^2}$ is within an error of 10^{-7} of the actual sum S (i.e., $\text{Error} = |S - S_N| \leq 10^{-7}$)?

(b) Use MATLAB to perform this summation and compare the result with the exact answer $S = \pi^2/6$ to see if the error goal has been met. Discuss roundoff errors as well.

SOLUTION: Part (a): The mathematical error analysis needed here involves a nice geometric estimation method for the error that is usually taught in calculus courses under the name of the integral test. In estimating the infinite sum

⁶We point out, however, that many symbolic calculus rules, and in particular, abilities to detect convergence of infinite series, are features available in MATLAB's Symbolic Toolbox (included in the Student Version). See Appendix A.

$S = \sum_{n=1}^{\infty} \frac{1}{n^2}$ with the finite partial sum $S_N = \sum_{n=1}^N \frac{1}{n^2}$, the error is simply the tail of the series:

$$\text{Error} = \left| \sum_{n=1}^{\infty} \frac{1}{n^2} - \sum_{n=1}^N \frac{1}{n^2} \right| = \sum_{n=N+1}^{\infty} \frac{1}{n^2} = \frac{1}{(N+1)^2} + \frac{1}{(N+2)^2} + \frac{1}{(N+3)^2} + \cdots.$$

The problem is to find out how large N must be so that this error is $\leq 10^{-7}$; of course, as is usually the case, we have no way of figuring out this error exactly (if we could, then we could determine S exactly). But we can estimate this "Error" with something larger, let's call it ErrorCap, that we can compute. Each term in the "Error" is represented by an area of a shaded rectangle (with base = 1) in Figure 5.2. Since the totality of the shaded rectangles lies under the graph of $y = 1/x^2$, from $x = N$ to $x = \infty$, we have

$$\text{Error} < \text{Error Cap} \equiv \int_N^{\infty} \frac{dx}{x^2} = \left. \frac{x^{-1}}{-1} \right|_{x=N}^{x=\infty} = 1/N;$$

and we conclude that our error will be less than 10^{-7} , provided that $\text{Error Cap} \leq 10^{-7}$, or $1/N \leq 10^{-7}$, or $N \geq 10^7$.

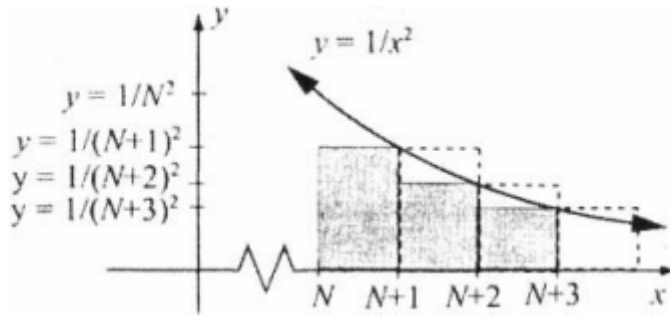


Figure 5.2: The areas of the shaded rectangles (that continue on indefinitely to the right) add up to precisely the Error $= \sum_{n=N+1}^{\infty} \frac{1}{n^2}$ of Example 5.6. But since they lie directly under the curve $y = 1/x^2$ from $x = N$ to $x = \infty$, we have $\text{Error} < \text{Error Cap} \equiv \int_N^{\infty} \frac{dx}{x^2}$.

Let us now use MATLAB to perform this summation, following the principle of adding up the numbers in order of increasing size.

```
>> Sum=0; %initialize sum
>>for n=10000000:-1:1
Sum=Sum +1/n^2;
end
>> Sum ->Sum = 1.64493396684823
```

This took about 30 seconds on the author's PC. Since we know that the exact infinite sum is $\pi^2/6$, we can look now at the actual error.

```
>> pi/s2/6-Sum
->ans = 9.999999495136080e-008 %This is indeed (just a wee bit) less than the desired
tolerance 10^-7.
```

Let us look briefly at (6) (with $a_n = 1/n^2$) to see what kind of a bound on roundoff errors it gives for this computation. At each step, we are adding to the accumulated sum a quotient. Each of these quotients ($1/n^2$) gives a roundoff error (by (5)) at most $a_n u$. Combining this estimate with (6) gives the bound

$$\begin{aligned} |\text{fl}(S_N) - S_N| &\leq 2u \left\{ \left[\frac{9999999}{(10000000)^2} \right] + \left[\frac{9999999}{(9999999)^2} \right] + \left[\frac{9999998}{(9999998)^2} \right] + \right. \\ &\quad \left. \cdots + \left[\frac{2}{(2)^2} \right] + \left[\frac{1}{(1)^2} \right] \right\} \\ &\leq 2u \left\{ \frac{1}{10000000} + \frac{1}{9999999} + \frac{1}{9999998} + \cdots + \frac{1}{2} + 1 \right\}. \end{aligned}$$

The sum in braces is $\sum_{n=1}^{10^7} \frac{1}{n}$. By a picture similar to the one of Figure 5.2, we can estimate this sum as follows: $\sum_{n=1}^{10^7} \frac{1}{n} \leq 1 + \int_1^{10^7} \ln(x) dx = 1 + \ln(10^7)$. Since the unit roundoff for MATLAB is 2^{-53} , we can conclude the following upper bound on the roundoff error of the preceding computation:

$$|\text{fl}(S_N) - S_N| \leq 2u (1 + \ln(10^7)) \approx 3.8 \times 10^{-15}.$$

We thus have confirmation that roundoff error did not play a large role in this computation. Let's now see what happens if we perform the same summation in the opposite order.

```

» Sum=0;
» for n=1:10000000
Sum=Sum +1/n^2;
end
>> pi^2/6-Sum ->ans = 1.000009668405966e-007

```

Now the actual error is a bit worse than the desired tolerance of 10^{-7} . The error estimate (6) would also be a lot larger if we reversed the order of summation. Actually for both of these floating point sums the roundoff errors were not very significant; such problems are called well-conditioned. The reason for this is that the numbers being added were getting small very fast. In the exercises and later in this book, we will encounter problems that are ill-conditioned, in that the roundoff errors can get quite large relative to the amount of arithmetic being done. The main difficulty in the last example was not roundoff error, but the computing time. If we instead wanted our accuracy to be 10^{-10} , the corresponding calculation would take over 8 hours on the same computer! A careful look at the strategy we used, however, will allow us to modify it slightly to get a much more efficient method. A Better Approach: Referring again to Figure 5.2, we see that by sliding all of the shaded rectangles one unit to the right, the resulting set of rectangles will completely cover the region under the graph of $y = 1/x^2$ from $x = N + 1$ to $x = \infty$. This gives the inequality:

$$\text{Error} > \int_{N+1}^{\infty} \frac{dx}{x^2} = \left[\frac{x^{-1}}{-1} \right]_{x=N}^{x=\infty} = 1/(N+1).$$

In conjunction with the previous inequality, we have in summary that:

$$\frac{1}{N+1} < \text{Error} < \frac{1}{N}.$$

If we add to our approximation S_N the average value of these two upper and lower bounds, we will obtain the following much better approximation for S :

$$\tilde{S}_N \equiv S_N + \frac{1}{2} \left[\frac{1}{N} + \frac{1}{N+1} \right].$$

The new error will be at most one-half the length of the interval from $1/(N+1)$ to $1/N$:

$$|S - \tilde{S}_N| \equiv \text{New Error} \leq \frac{1}{2} \left[\frac{1}{N} - \frac{1}{N+1} \right] = \frac{1}{2N(N+1)} < \frac{1}{2N^2}.$$

(The elementary last inequality was written so as to make the new error bound easier to use, as we will now see.) With this new scheme much less work will be required to attain the same degree of accuracy. Indeed, if we wanted the error to be less than 10^{-7} , this new scheme would require that the number of terms N needed to sum should satisfy $1/2N^2 \leq 10^{-7}$ or $N \geq \sqrt{10^7/2} = 2236.07\dots$, a far cry less than the 10 million terms needed with the original method! By the same token, to get an error less than 10^{-10} , we would need only take $N = 70,711$! Let us now verify, on MATLAB, this 10-significant-digit approximation:

Now the actual error is a bit worse than the desired tolerance of 10^{-7} . The error estimate (6) would also be a lot larger if we reversed the order of summation. Actually for both of these floating point sums the roundoff errors were not very significant; such problems are called well-conditioned. The reason for this is that the numbers being added were getting small very fast. In the exercises and later in this book, we will encounter problems that are ill-conditioned, in that the roundoff errors can get quite large relative to the amount of arithmetic being done. The main difficulty in the last example was not roundoff error, but the computing time. If we instead wanted our accuracy to be 10^{-10} , the corresponding calculation would take over 8 hours on the same computer! A careful look at the strategy we used, however, will allow us to modify it slightly to get a much more efficient method.

A Better Approach: Referring again to Figure 5.2, we see that by sliding all of the shaded rectangles one unit to the right, the resulting set of rectangles will completely cover the region under the graph of $y = 1/x^2$ from $x = N + 1$ to $x = \infty$. This gives the inequality

$$\text{Error} > \int_{N+1}^{\infty} \frac{dx}{x^2} = \left[\frac{x^{-1}}{-1} \right]_{x=N}^{x=\infty} = 1/(N+1).$$

In conjunction with the previous inequality, we have in summary that:

$$\frac{1}{N+1} < \text{Error} < \frac{1}{N}.$$

If we add to our approximation S_N the average value of these two upper and lower bounds, we will obtain the following much better approximation for S :

$$\tilde{S}_N \equiv S_N + \frac{1}{2} \left[\frac{1}{N} + \frac{1}{N+1} \right].$$

The new error will be at most one-half the length of the interval from $1/(N+1)$ to $1/N$:

$$|S - \tilde{S}_N| \equiv \text{New Error} \leq \frac{1}{2} \left[\frac{1}{N} - \frac{1}{N+1} \right] = \frac{1}{2N(N+1)} < \frac{1}{2N^2}.$$

(The elementary last inequality was written so as to make the new error bound easier to use, as we will now see.) With this new scheme much less work will be required to attain the same degree of accuracy. Indeed, if we wanted the error to be less than 10^{-7} , this new scheme would require that the number of terms N needed to sum should satisfy $1/2N^2 \leq 10^{-7}$ or $N \geq \sqrt{10^7/2} = 2236.07\dots$, a far cry less than the 10 million terms needed with the original method! By the same token, to get an error less than 10^{-10} , we would need only take $N = 70,711$! Let us now verify, on MATLAB, this 10-significant-digit approximation:

```

» Sum=0; N=70711;
» for n=N:-1:1
Sum=Sum +1/n^2;
end
>> format long
>> Sum=Sum+(1/N + 1 / (N+1) ) /2          ->Sum = 1.64493406684823
>>abs(Sum-pi^2/6) ->ans =8.881784197001252e-016

```

The actual error here is even better than expected; our approximation is actually as good as machine precision! This is a good example where the (worst-case) error guarantee (New Error) is actually a lot larger than the true error. A careful examination of Figure 5.2 once again should help to make this plausible.

We close this section with an exercise for the reader that deals with the approximation of an alternating series. This one is rather special in that it can be used to approximate the number π and, in particular, we will be able to check the accuracy of our numerical calculations against the theory. We recall from calculus that an alternating series is an infinite series of the form $\sum (-1)^n a_n$, where $a_n > 0$ for each n . Leibniz's Theorem states that if the a_n 's decrease, $a_n \geq a_{n+1}$ for each n (sufficiently large), and converge to zero, $a_n \rightarrow 0$ as $n \rightarrow \infty$, then the infinite series converges to a sum S . It furthermore states that if $S_N = \sum^N (-1)^n a_n$ denotes a partial sum (the lower index is left out since it can be any integer), then we have the error estimate: $|S - S_N| \leq a_{N+1}$.

EXERCISE FOR THE READER 5.5: Use the infinite series expansion:

$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots = \frac{\pi}{4},$$

to estimate π with an error less than 10^{-7} .

EXERCISES 5.3:

NOTE: Unless otherwise specified, assume that all floating point arithmetic in these exercises is done in base 10.

- (a) In chopped floating point arithmetic with s digits and exponent range $m \leq e \leq M$, write down (in terms of these parameters s, m , and M) the largest positive floating point number and the smallest positive floating point number.
(b) Would these answers change if we were to use rounded arithmetic instead?
- Recall that for two real numbers x and y , the average value $(x+y)/2$ of x and y lies between the values of x and y .
(a) Working in a chopped floating point arithmetic system, find an example where $\text{fl}((x+y)/2)$ is strictly less than $\text{nl}(x)$ and $\text{fl}(y)$.
(b) Repeat part (a) in rounded arithmetic.
- (a) In chopped floating point arithmetic with base β , with s digits and exponent range $m \leq e \leq M$, write down (in terms of these parameters β, s, m , and M) the largest positive floating point number and the smallest positive floating point number.
(b) Would these answers change if we were to use rounded arithmetic instead?
- For each of the following arithmetic properties, either explain why the analog will be true in floating point arithmetic, or give an example where it fails. If possible, provide counterexamples that do not involve overflows, but take underflows to be zero.
(a) (*Commutativity of Addition*) $x + y = y + x$
(b) (*Commutativity of Multiplication*) $x \cdot y = y \cdot x$
(c) (*Associativity of Addition*) $x \cdot (y \cdot z) = (x \cdot y) \cdot z$
(d) (*Distributive Law*) $x \cdot (y + z) = x \cdot y + x \cdot z$
(c) (*Zero Divisors*) $x \cdot y = 0 \Rightarrow x = 0$ or $y = 0$

5. Consider the infinite series: $1 + \frac{1}{8} + \frac{1}{27} + \frac{1}{64} + \cdots \frac{1}{n^3} + \cdots$.
 - (a) Does it converge? If it does not, stop here; otherwise continue.
 - (b) How many terms would we have to sum to get an approximation to the infinite sum with an absolute error $< 10^{-7}$?
 - (c) Obtain such an approximation.
 - (d) Using an approach similar to what was done after Example 5.6, add an extra term to the partial sums so as to obtain an improved approximation for the infinite series. How many terms would be required with this improved scheme? Perform this approximation and compare the answer with that obtained in part (c).
6. Consider the infinite series: $\sum_{n=1}^{\infty} \frac{1}{n\sqrt{n}}$.
 - (a) Does it converge? If it does not, stop here; otherwise continue.
 - (b) How many terms would we have to sum to get an approximation to the infinite sum with an absolute error $1/500$?
 - (c) Obtain such an approximation.
 - (d) Using an approach similar to what was done after Example 5.6, add an extra term to the partial sums so as to obtain an improved approximation for the infinite series. How many terms would be required with this improved scheme? Perform this approximation and compare the answer with that obtained in part (c).
7. Consider the infinite series: $\sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n} = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \cdots$.
 - (a) Show that this series satisfies the hypothesis of Leibniz's theorem (for n sufficiently large) so that from the theorem, we know the series will converge to a sum S .
 - (b) Use Leibniz's theorem to find an integer N so that summing up to the first N terms only will give approximation to the sum with an error less than .0001.
 - (c) Obtain such an approximation.
8. Repeat all parts of Exercise 7 for the series: $\sum_{n=1}^{\infty} (-1)^n \frac{\ln n}{n}$.
9. (a) In an analogous fashion to what was done in Example 5.5, establish the following estimate for floating point multiplications of a set of N positive real numbers, $P_N = a_1 \cdot a_2 \cdots a_N$:

$$|f(P_N) - P_N| \leq Nu$$

We have, as before, ignored higher-order error terms. Thus, as far as minimizing errors is concerned, unlike for addition, the roundoff errors do not depend substantially on the order of multiplication.

- (b) In forming a product of positive real numbers, is there a good order to multiply so as to minimize the chance of encountering overflows or underflows? Explain your answer with some examples.
10. (a) Using an argument similar to that employed in Example 5.4, show that in base β chopped floating point arithmetic the unit roundoff is given by $u = \beta^{1-s}$. (b) Show that in rounded arithmetic, the unit roundoff is given by $u = \beta^{1-s}/2$.
11. Compare and contrast a two-digit ($s = 2$) floating point number system in base $\beta = 4$ and a fourdigit ($s = 4$) binary ($\beta = 2$) floating point number system.
12. In Exercise for the Reader 5.5, we used the following infinite series to approximate π :

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} \cdots \Rightarrow \pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} \cdots$$

This alternating series was not a very efficient way to compute π , since it converges very slowly. Even to get an approximation with an accuracy of 10^{-7} , we would need to sum about 20 million terms. In this problem we will give a much more efficient (faster-converging) series for computing π by using Machin's identity ((13) of Chapter 2):

$$\frac{\pi}{4} = 4 \tan^{-1} \left(\frac{1}{5} \right) - \tan^{-1} \left(\frac{1}{239} \right)$$

- (a) Use this identity along with the arctangent's MacLaurin series (see equation (11) of Chapter 2) to express π either as a difference of two alternating series, or as a single alternating series. Write your series both in explicit notation (as above) and in sigma notation.
- (b) Perform an error analysis to see how many terms you would need to sum in the series (or difference of series) to get an approximation to π with error $< 10^{-7}$. Get MATLAB to perform this summation (in a "good" order) to thus obtain an approximation to π .
- (c) How many terms would we need to sum so that the (exact mathematical) error would be less than 10^{-30} ? Of course, MATLAB only uses 16-digit floating point arithmetic so we could not directly use it to get such an approximation to π (Unless we used the symbolic toolbox; see Appendix A).

13. Here is π , accurate to 30 decimal places:

$$\pi = 3.141592653589793238462643383279\dots$$

Can you figure out a way to get MATLAB to compute π to 30 decimals of accuracy without using its symbolic capabilities?

Suggestions: What is required here is to build some MATLAB functions that will perform certain mathematical operations with more significant digits (over 30) than what MATLAB usually guarantees (about 15). In order to use the series of Exercise 12, you will need to build functions that will add/subtract, and multiply and divide (at least). Here is an example of a possible syntax for such new function: $z = \text{highaccuracyadd}(x, y)$ where x and y are vectors containing the mantissa (with over 30 digits) as well as the exponent and sign (can be stored as 1 for plus, 2 for negative in one of the slots). The output z will be another vector of the same size that represents a 30+-significant-digit approximation to the sum $x + y$. This is actually quite a difficult problem, but it is fun to try.