Introduction to Scientific Programming with MATLAB

Cymra Haskell

September 1, 2010

Preface

My intention in writing this booklet is to provide an introduction to scientific computing using MATLAB. I originally wrote the booklet for students in a second semester calculus class at the University of Southern California, so I assume that the reader has some familiarity with calculus (about what one would learn in a one semester class) but need not have any knowledge of linear algebra or any experience programming.

The booklet consists of six lessons that can each be completed in about an hour (although more time will be needed practicing the concepts in each lesson). Each lesson is followed by a worksheet. The lessons were originally written some years ago based on an earlier version of MATLAB. Later versions of MATLAB may not behave *exactly* as described in this booklet. Also, the default MATLAB settings on the computer on which you are working may be different from the default settings assumed here. This means that there may be minor differences in how MATLAB looks, feels, and functions on your computer and what is described here. For example, some of the error messages may have changed slightly, the command window may be 'docked' or 'undocked', and the icons for changing directories may look different and be located in a different place. However, these are all minor differences. Ask your instructor if you are confused about anything.

By learning scientific computing you are opening up many new possibilities for not only applying mathematics but also doing mathematics. Indeed, you will have unleashed a powerful new tool for mathematical exploration. I wish you the very best in the journey you are about to undertake.

Cymra Haskell

1 Matrices and Vectors

When you think of arithmetic, you probably think of adding, subtracting, multiplying and dividing *numbers*. However, basic arithmetic in MATLAB is concerned with *matrices*. In this lesson, we recall what matrices are and what it means to perform various arithmetic operations with them. This will lay the groundwork for introducing MATLAB in the next lesson.

1.1 General Terminology

A **matrix** is a rectangular array of numbers. It is usually written with brackets enclosing the numbers on either side. The following are all examples of matrices.

a) $A = \begin{pmatrix} 1 & 4 & 1 & 1 \\ -1 & 0 & 6 & 5 \end{pmatrix}$

b) $B = \begin{pmatrix} 2 & -1 & 0 \\ 3 & 5 & 4 \\ 0 & 3 & -7 \end{pmatrix}$

c) $C = \begin{pmatrix} 3.2 & -2.5 & 4.0 & 10.7 & -5.4 \end{pmatrix}$

d) $D = \begin{pmatrix} 0 \\ 1 \\ 2 \end{pmatrix}$

e) $E = \begin{pmatrix} 17 \end{pmatrix}$

The number of rows and columns is called the **dimension** of the matrix. For instance, the matrix A above has 2 rows and 4 columns and is referred to as a 2×4 (read, "two by four") matrix. The matrix B is 3×3, C is 1×5, D is 3×1, and E is 1×1. Notice that the number of rows is always stated first followed by the number of columns. Thus, D is a 3×1 matrix and not a 1×3 matrix. If a matrix has the same number of rows as columns then it is called a **square matrix**. The matrices B and E above are both square matrices. If a matrix has only one row or only one column then it can also be called a **vector**. More specifically, a matrix that has only one row is called a **row vector** and a matrix with only one column is called a **column vector**. For instance, C, D, and E are all vectors; C is a row vector, D is a column vector, and E is both a row vector and a column vector. Notice that a 1×1 matrix such as E above is simply a number. It can be written without brackets surrounding it. It is both a row vector and a column vector, but is most commonly referred to as a **scalar**.

The rows of a matrix are numbered from top to bottom and the columns are numbered from left to right. For example, the first row of A is

$$\begin{pmatrix} 1 & 4 & 1 & 1 \end{pmatrix}$$

and the second column is

$$\begin{pmatrix} 4 \\ 0 \end{pmatrix}.$$

The numbers in a matrix are called **entries**. They can be identified by the row and column that they are in. For example, the entry in the second row and first column of A is -1 and the entry in the second row and third column of B is 4. If M is a matrix, then M_{ij} refers to the entry in its i'th row and j'th column. For example $A_{21} = -1$ and $B_{23} = 4$. Notice that, just as the dimension of a matrix is the number of rows followed by the number of columns, an entry in a matrix is specified by stating first its row number and then its column number. Thus, $A_{21} = -1$ but $A_{12} = 4$. Two matrices are **equal** if and only if they have the same dimension and their corresponding entries are equal to each other.

The **transpose** of a matrix M is the matrix that is obtained when the rows of M are turned into columns (and vice versa). The transpose of M is denoted M^t. For instance, consider the matrix A above. To form A^t we take the first row of A

$$\begin{pmatrix} 1 & 4 & 1 & 1 \end{pmatrix}$$

and write it as a column,

$$\begin{pmatrix} 1 \\ 4 \\ 1 \\ 1 \end{pmatrix}.$$

This is the first column of A^t. Similarly, we take the second row of A and write it as a column obtaining

$$\begin{pmatrix} -1 \\ 0 \\ 6 \\ 5 \end{pmatrix}.$$

This is the second column of A^t. Thus,

$$A^t = \begin{pmatrix} 1 & -1 \\ 4 & 0 \\ 1 & 6 \\ 1 & 5 \end{pmatrix}.$$

Shown below are the transposes of B, C, D and E.

$$B^t = \begin{pmatrix} 2 & 3 & 0 \\ -1 & 5 & 3 \\ 0 & 4 & -7 \end{pmatrix}, \qquad C^t = \begin{pmatrix} 3.2 \\ -2.5 \\ 4.0 \\ 10.7 \\ -5.4 \end{pmatrix}, \qquad D^t = \begin{pmatrix} 0 & 1 & 2 \end{pmatrix}, \qquad E^t = 17.$$

Notice that if M is an $n \times m$ matrix then M^t is an $m \times n$ matrix and that $(M^t)_{ij} = M_{ji}$. For example, $(B^t)_{23} = 3 = B_{32}$ and $(C^t)_{41} = 10.7 = C_{14}$.

1.2 Adding and Subtracting Matrices

If two matrices A and B have the same dimension then we define $A + B$ to be the matrix that is formed when corresponding elements of A and B are added together. Similarly, $A - B$ is the matrix that is formed when each element in B is subtracted from its corresponding element in A. For example

$$\begin{pmatrix} 2 & -5 & 0 \\ -1 & 4 & 5 \end{pmatrix} + \begin{pmatrix} 0 & 3 & 2 \\ 1 & 4 & -2 \end{pmatrix} = \begin{pmatrix} 2 & -2 & 2 \\ 0 & 8 & 3 \end{pmatrix},$$
$$\begin{pmatrix} 2 & -5 & 0 \\ -1 & 4 & 5 \end{pmatrix} - \begin{pmatrix} 0 & 3 & 2 \\ 1 & 4 & -2 \end{pmatrix} = \begin{pmatrix} 2 & -8 & -2 \\ -2 & 0 & 7 \end{pmatrix},$$

but

$$\begin{pmatrix} 2 & -5 & 0 \\ -1 & 4 & 5 \end{pmatrix} + \begin{pmatrix} 0 & 1 \\ 3 & 4 \\ 2 & -2 \end{pmatrix}$$

is not defined. Notice that addition of matrices, like addition of numbers, is both **commutative** and **associative**. In other words, if A, B and C all have the same dimension, then

$$\boxed{\begin{aligned} A + B &= B + A \\ A + (B + C) &= (A + B) + C \end{aligned}}$$

If all of the entries in a matrix are 0 then it is called a **zero matrix**. Here are some examples of zero matrices:

$$\begin{pmatrix} 0 & 0 & 0 \end{pmatrix}, \qquad \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \qquad \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \qquad \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}$$

A zero matrix will often be written $\mathbf{0}$ irrespective of its dimension. Zero matrices play the same role in the arithmetic of matrices as the number 0 does in the arithmetic of numbers. In particular, they are additive identities. This means that given any matrix A and the zero matrix $\mathbf{0}$ that has the same size as A,

$$A + \mathbf{0} = \mathbf{0} + A = A.$$

The **negative** of a matrix A is the matrix that is obtained by multiplying each entry in A by -1. The negative of A is denoted $-A$. For example

$$-\begin{pmatrix} 2 & -5 & 0 \\ -1 & 4 & 5 \end{pmatrix} = \begin{pmatrix} -2 & 5 & 0 \\ 1 & -4 & -5 \end{pmatrix}$$

Notice that $A + (-B)$ is the same as $A - B$. Given any matrix A, the negative of A is the additive inverse of A. In other words,

$$A + (-A) = -A + A = \mathbf{0}.$$

1.3 Multiplying Matrices by a Scalar

Recall that a *scalar* refers to a 1×1 matrix which is just a number. Given any scalar, k, and any matrix A, we define the products kA and Ak to be the matrix that is obtained when each entry in A is multiplied by k. For example

$$5.2\begin{pmatrix} 0 & 3 & 2 \\ 1 & 4 & -2 \end{pmatrix} = \begin{pmatrix} 0 & 3 & 2 \\ 1 & 4 & -2 \end{pmatrix} 5.2 = \begin{pmatrix} 0 & 15.6 & 10.4 \\ 5.2 & 20.8 & -10.4 \end{pmatrix}$$

Mutliplication by a scalar is distributive over addition. In other words, if k is a scalar and A and B are matrices that have the same dimension then

$$\boxed{k(A+B) = kA + kB}$$

Notice that the negative of a matrix is obtained by multiplying it by -1. In other words,

$$(-1)A = -A$$

for any matrix A. Also, if k is a positive integer then kA is the same as adding A to itself k times. For example,

$$3A = A + A + A.$$

1.4 The Product of Two Matrices

If A is an $n \times p$ matrix and B is an $p \times m$ matrix then the product AB is defined and is an $n \times m$ matrix. Notice that the product is only defined when the number of columns in A is equal to the number of rows in B. If this is not the case, then the product is not defined. (There is one exception to this; the product is always defined if either A is 1×1 or B is 1×1 since that matrix is then a scalar.) The definition of the product might seem rather convoluted at first, but you will see later in this lesson why it is defined the way it is.

We define the product AB by defining its ij'th element, $(AB)_{ij}$. To find this element, consider the i'th row of A

$$\begin{pmatrix} A_{i1} & A_{i2} & A_{i3} & \dots & A_{ip} \end{pmatrix}$$

and the j'th column of B

$$\begin{pmatrix} B_{1j} \\ B_{2j} \\ B_{3j} \\ \vdots \\ B_{pj} \end{pmatrix}.$$

Notice that these two vectors have the same number of entries; they both have p entries. Multiply the first entry in the row vector with the first entry in the column

vector, the second entry in the row vector with the second entry in the column vector etc. This yields p numbers each of which is the product of an entry in A and an entry in B. Add these numbers together to obtain $(AB)_{ij}$. In other words,

$$(AB)_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j} + A_{i3}B_{3j} + \ldots A_{ip}B_{pj}.$$

To illustrate this definition, consider the matrices

$$A = \begin{pmatrix} -1 & 3 & 5 \\ 2 & -1 & -2 \end{pmatrix} \qquad \text{and} \qquad B = \begin{pmatrix} 1 & 3 \\ 2 & 0 \\ -1 & 2 \end{pmatrix}.$$

We will find the product AB. Notice that A is 2×3 and B is 3×2, so the number of columns in A is equal to the number of rows in B. This means that the product is defined and is a 2×2 matrix. We will find each of its entries one at a time. To find the entry in the first row and first column, consider the first row of A and the first column of B:

$$\begin{pmatrix} -1 & 3 & 5 \end{pmatrix} \qquad \text{and} \qquad \begin{pmatrix} 1 \\ 2 \\ -1 \end{pmatrix}.$$

Multiply the corresponding entries and add them to obtain

$$(-1)(1) + (3)(2) + (5)(-1) = 0.$$

To find the entry in the first row and second column, consider the first row of A and the second column of B:

$$\begin{pmatrix} -1 & 3 & 5 \end{pmatrix} \qquad \text{and} \qquad \begin{pmatrix} 3 \\ 0 \\ 2 \end{pmatrix}.$$

Multiply the corresponding entries and add them to obtain

$$(-1)(3) + (3)(0) + (5)(2) = 7.$$

To find the entry in the second row and first column, consider the second row of A and the first column of B:

$$\begin{pmatrix} 2 & -1 & -2 \end{pmatrix} \qquad \text{and} \qquad \begin{pmatrix} 1 \\ 2 \\ -1 \end{pmatrix}.$$

Multiply the corresponding entries and add them to obtain

$$(2)(1) + (-1)(2) + (-2)(-1) = 2.$$

To find the entry in the second row and second column, consider the second row of A and the first column of B:

$$\begin{pmatrix} 2 & -1 & -2 \end{pmatrix} \qquad \text{and} \qquad \begin{pmatrix} 3 \\ 0 \\ 2 \end{pmatrix}.$$

Multiply the corresponding entries and add them to obtain

$$(2)(3) + (-1)(0) + (-2)(2) = 2.$$

Thus the product is

$$AB = \begin{pmatrix} 0 & 7 \\ 2 & 2 \end{pmatrix}$$

Here is another example worked out more compactly. Notice that the first matrix is 4×2 and the second matrix is 2×1 so the product is a 4×1 matrix.

$$\begin{pmatrix} -5 & 2 \\ 0 & 1 \\ 2 & 3 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ -2 \end{pmatrix} = \begin{pmatrix} (-5)(1) + (2)(-2) \\ (0)(1) + (1)(-2) \\ (2)(1) + (3)(-2) \\ (0)(1) + (0)(-2) \end{pmatrix} = \begin{pmatrix} -9 \\ -2 \\ -4 \\ 0 \end{pmatrix}$$

Matrix multiplication has some of the properties that multiplication of numbers has but not all of them. Like multiplication of numbers, it is associative and distributive over addition. In other words, if A and B are both $n \times p$, C is $p \times q$, D is $q \times r$, and E and F are both $p \times m$ then

$$\begin{gathered} A(CD) = (AC)D \\ A(E+F) = AE + AF \\ (A+B)E = AE + BE \end{gathered}$$

However, it is not commutative. Indeed, if A is $n \times p$ and B is $p \times m$ and $n \neq m$ then BA is not defined so it certainly cannot be equal to AB. Even when AB and BA are both defined and have the same size, they are not necessarily equal to each other. For instance,

$$\begin{pmatrix} 1 & 2 \\ -1 & -2 \end{pmatrix} \begin{pmatrix} -2 & 4 \\ 1 & -2 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix},$$

$$\text{but} \qquad \begin{pmatrix} -2 & 4 \\ 1 & -2 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ -1 & -2 \end{pmatrix} = \begin{pmatrix} -6 & 4 \\ 3 & -2 \end{pmatrix}$$

In other words,

Matrix multiplication is not commutative.

With real numbers, we know that if $xy = 0$ then either $x = 0$ or $y = 0$. We use this property a lot when solving equations. For example, one way to solve the equation

$$x^2 - 4x + 3 = 0$$

is to factor the left-hand side obtaining

$$(x - 1)(x - 3) = 0.$$

It follows that either $x - 1 = 0$ (i.e $x = 1$) or $x - 3 = 0$ (i.e. $x = 3$). This property is not true with matrix multiplication. As shown in the example above it is possible to have $AB = \mathbf{0}$ without either A or B being equal to $\mathbf{0}$.

$$AB = 0 \not\Rightarrow A = \mathbf{0} \text{ or } B = \mathbf{0}$$

Square matrices that have 1's on the main diagonal and 0's everywhere else are called **identity** matrices. Here are some examples of identity matrices:

$$I_1 = \begin{pmatrix} 1 \end{pmatrix}, \qquad I_2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \qquad I_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \qquad I_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

They play the same role in the multiplication of matrices that 1 plays in the multiplication of numbers. In particular they are multiplicative identities. In other words, if A is an $n \times m$ matrix then

$$AI_m = A \qquad \text{and} \qquad I_n A = A.$$

1.5 Interpreting Matrix Products in Applications

The definition of the product of two matrices seems mysterious to most people the first time they see it. The following examples illustrate why the product is defined the way it is and how it can be interpreted in different contexts.

Example 1: (Adapted from *'Finite Mathematics'* by R. A. Barnett, M. R. Ziegler, and K. E. Byleen)

A company that makes two-person and four-person inflatable boats has two manufacturing plants: one in Massachusetts and the other in Virginia. Each boat is first cut in the Cutting Department, then assembled in the Assembly Department and then packaged in the Packaging Department. The time needed in each department to work on a boat depends on the type of boat and the wages of the workers in each department depends on the plant in which they work. The matrix T below shows the time needed (in hours) for each boat in each department.

$$T = \begin{pmatrix} 1.0 & 0.9 & 0.3 \\ 1.5 & 1.2 & 0.4 \end{pmatrix}$$

The rows of T show the times for two-person and four-person boats in that order, whilst the columns show the times spent in the Cutting Department, the Assembly Department, and the Packaging Department in that order. For instance, notice that $T_{23} = 0.4$. This means that it takes 0.4 hours to package a four-person boat in the Packaging Department. The matrix W below shows the wages (in dollars per hour) of the workers in each department at each plant.

$$W = \begin{pmatrix} 17 & 15 \\ 12 & 10 \\ 11 & 10 \end{pmatrix}$$

The rows of W show the wages for the workers in the Cutting Departments, Assembly Departments, and Packaging Departments in that order, whilst the columns show the wages of the workers at the Massachusetts plant and at the Virginia Plant in that order. For instance, notice that $W_{21} = 12$. This means that a worker in the Assembly Department at the Massachusetts plant earns \$12 per hour.

Notice that T is 2×3 and W is 3×2. This means that the product TW is defined and is a 2×2 matrix. In the calculation of TW below we have put labels on the rows and columns of T and W to remind us of their meaning. We will see why the rows and columns of the product matrix are labeled the way they are when we have understood the meaning of the product matrix TW.

$$TW = \begin{array}{c} \\ 2 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccc} C & A & P \end{array} \\ \begin{pmatrix} 1.0 & 0.9 & 0.3 \\ 1.5 & 1.2 & 0.4 \end{pmatrix} \end{array} \begin{array}{c} \begin{array}{cc} M & V \end{array} \\ \begin{array}{c} C \\ A \\ P \end{array} \begin{pmatrix} 17 & 15 \\ 12 & 10 \\ 11 & 10 \end{pmatrix} \end{array} = \begin{array}{c} \\ 2 \\ 4 \end{array} \begin{array}{c} \begin{array}{cc} M & V \end{array} \\ \begin{pmatrix} 31.1 & 27 \\ 44.3 & 38.5 \end{pmatrix} \end{array}$$

To understand what TW tells us about the manufacture of these boats, consider first the entry in the first row and first column. This entry was obtained using the first row of T and the first column of W:

$$(1.0)(17) + (0.9)(12) + (0.3)(11) = 31.1.$$

Notice that 1.0 is the number of hours it takes to cut a 2-person boat in the Cutting Department and that 17 is the number of dollars per hour workers in the Cutting Department are paid at the Massachusetts plant. In other words, $(1.0)(17)$ is the amount it costs to cut a 2-person boat at the Massachusetts plant. Similarly, $(0.9)(12)$ is the amount it costs to assemble a 2-person boat at the Massachusetts plant and $(0.3)(11)$ is the amount it costs to package a 2-person boat at the Massachusetts plant. Thus, the sum 31.1 is the cost to manufacture (cut, assemble and package) a 2-person boat at the Massachusetts plant. Similarly the entry in, say, the second row and second column, is the cost to manufacture a 4-person boat at the Virginia plant. Thus, the matrix TW indicates the cost of manufacturing each of the different types of boats at each of the different plants.

Notice that the units of the entries in T are \$/hour and the units of the entries in W are hours. So, it makes sense that the units of TW would be \$/hour $\times$ hours = \$.

To multiply two matrices A and B you need the number of columns of A to be equal to the number of rows of B. The dimension of the product is then the number of rows of A and the number of columns of B. The labels work similarly. When we multiplied T and W together we saw that the labels on the columns of T matched the labels on the rows of W. Moreover, the labels on the rows of TW were the labels on the rows of T and the labels on the columns of TW were the labels on the columns of W.

On the other hand, consider the product WT. This is defined and is a 3×3 matrix:

$$WT = \begin{matrix} \\ C \\ A \\ P \end{matrix} \begin{matrix} \begin{matrix} M & V \end{matrix} \\ \begin{pmatrix} 17 & 15 \\ 12 & 10 \\ 11 & 10 \end{pmatrix} \end{matrix} \quad \begin{matrix} \\ 2 \\ 4 \end{matrix} \begin{matrix} \begin{matrix} C & A & P \end{matrix} \\ \begin{pmatrix} 1.0 & 0.9 & 0.3 \\ 1.5 & 1.2 & 0.4 \end{pmatrix} \end{matrix} = \begin{matrix} \\ ? \\ ? \\ ? \end{matrix} \begin{matrix} \begin{matrix} ? & ? & ? \end{matrix} \\ \begin{pmatrix} 39.5 & 33.3 & 11.1 \\ 27.0 & 22.8 & 7.6 \\ 26.0 & 21.9 & 7.3 \end{pmatrix} \end{matrix}.$$

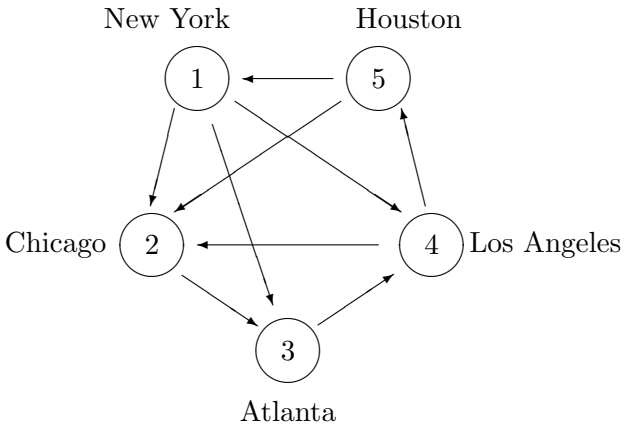
However, the column labels of W don't match the row labels of T and we see, upon reflection, that the matrix doesn't really mean anything. For instance, consider the entry in the first row and second column that was calculated using the first row of W and the second column of T:

$$(17)(0.9) + (15)(1.2) = 33.3.$$

The number 17 in this calculation represents the number of dollars per hour a worker in the Cutting Department receives at the Massachusetts plant. This is multiplied by 0.9 which is the number of hours it takes to assemble a 4-person boat in the Assembly Department. There is no discernible reason to multiply these two numbers together.

Example 2: (Adapted from *'Finite Mathematics'* by R. A. Barnett, M. R. Ziegler, and K. E. Byleen)

A nationwide air freight service has connecting flights between five cities as illustrated in the diagram below.



This diagram can be represented by the incidence matrix A below where A_{ij} is equal to 1 if there is a flight from city i to city j and is equal to 0 otherwise.

$$A = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Notice that the row number indicates the city of departure and the column number indicates the city of arrival.

Let's consider the product matrix $AA = A^2$:

$$A^2 = \begin{array}{c} \\ Depart \end{array} \overset{\begin{array}{c} Arrive \end{array}}{\begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix}} \quad \begin{array}{c} \\ Depart \end{array} \overset{\begin{array}{c} Arrive \end{array}}{\begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix}}$$

$$= \begin{array}{c} \\ Depart \end{array} \overset{\begin{array}{c} Arrive \end{array}}{\begin{pmatrix} 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 2 & 1 & 0 \end{pmatrix}}.$$

It looks at first as if the column label of the first matrix doesn't match up with the row label of the second matrix. However, in a sense it does, since any city that you arrive at can then be departed from. This is how these labels should be interpreted in order to make sense of the product matrix. To understand the full product, notice that $A_{ij}A_{jk}$ can take on two possible values: 0 or 1. When it is possible to fly from City i to City j and from City j to City k both $A_{ij} = 1$ and $A_{jk} = 1$, so the product is 1. When it is not possible to do this because one of these flights doesn't exist, then at least one of A_{ij} and A_{jk} will be 0 which will make the product equal to 0. Thus the value of $A_{ij}A_{jk}$ tells us whether or not it is possible to fly $i \to j \to k$, and

$$(A^2)_{ik} = A_{i1}A_{1k} + A_{i2}A_{2k} + A_{i3}A_{3k} + A_{i4}A_{4k} + A_{i5}A_{5k}$$

tells us how many ways there are to get from City i to City k if we insist on taking exactly 2 flights to get there. Thus, each entry in A^2 tells us how many ways there are of getting from the city determined by the row number to the city determined by the column number by taking exactly two flights. It follows that $A + A^2$ shows the number of ways of flying from any city to any other by taking at most 2 flights.

Similarly, $AAA = A^3$ shows the number of ways to get between each pair of cities taking exactly three flights and $A + A^2 + A^3$ shows the number of ways of flying from any city to any other by taking at most 3 flights.

1.6 The Length of a Vector and the Distance Between Vectors

Most of the matrices you will deal with this semester when you use Matlab will be vectors. A vector is simply a list of numbers. We think of a vector as small if all of its entries are close to 0 and large if any of its entries are large (in absolute value). More precisely, we define the **length** of a vector $x = \begin{pmatrix} x_1 & x_2 & \dots & x_n \end{pmatrix}$ or $x = \begin{pmatrix} x_1 & x_2 & \dots & x_n \end{pmatrix}^t$ to be the square root of the sum of the squares of all of its entries and denote the length by $||x||$. In other words

$$||x|| = \sqrt{x_1^2 + x_2^2 + \dots x_n^2}.$$

If x is a row vector then x^t is a column vector and xx^t is a scalar and is equal to the sum of the squares of all of the entries in x. Thus

$$||x|| = \sqrt{xx^t}.$$

Similarly, if x is a column vector then

$$||x|| = \sqrt{x^t x}.$$

For example, if $x = \begin{pmatrix} -3 & 0 & 4 \end{pmatrix}$ then

$$||x|| = \sqrt{\begin{pmatrix} -3 & 0 & 4 \end{pmatrix} \begin{pmatrix} -3 \\ 0 \\ 4 \end{pmatrix}} = \sqrt{(-3)^2 + 0^2 + 4^2} = 5.$$

Two vectors x and y that have the same dimension are close to each other if all of their corresponding entries are close to each other. In other words, they are close if $x - y$ is small. We think of $||x - y||$ as the **distance** between x and y. For instance, suppose $x = \begin{pmatrix} -2 & 5 & 4 \end{pmatrix}$ and $y = \begin{pmatrix} -2.1 & 30 & 4.01 \end{pmatrix}$. Even though the first and third entries in x and y are close to each other, the second entry is not and we see that the distance of x from y,

$$||x - y|| = || \begin{pmatrix} 0.1 & -25 & -0.01 \end{pmatrix} || = \sqrt{0.1^2 + (-25)^2 + (-0.01)^2} = 25.000002$$

is not particularly small.

1.7 Further Study

Linear algebra is the study of matrices and vectors. If you pursue your studies in science and/or engineering you will undoubtedly see that matrices and vectors are ubiquitous. For instance, they are used extensively in computer graphics, operations research, graph theory, differential equations, and geometry. The introduction to matrices given here is very rudimentary. The goal in this lesson was to cover them sufficiently to allow you to use Matlab and other high-level scientific programming languages effectively. In particular, you should notice that we haven't discussed what it means to divide two matrices.

Worksheet 1

1. Consider the following matrices.

$$A = \begin{pmatrix} 4 & 1 & 3 & 2 \\ 1 & 4 & 3 & 2 \end{pmatrix} \qquad B = \begin{pmatrix} 1 & 2 \\ 9 & 0 \end{pmatrix} \qquad C = \begin{pmatrix} 5 & 0 & 4 \end{pmatrix}$$

$$D = \begin{pmatrix} -8 & 2 & 0 & 4 \\ -1 & 2 & 3 & 7 \\ 1 & 6 & 6 & 5 \\ 5 & 9 & 7 & -3 \end{pmatrix} \qquad E = \begin{pmatrix} -3.4 \end{pmatrix} \qquad F = \begin{pmatrix} 2 \\ -1 \\ 3 \\ 5 \\ 6 \end{pmatrix}$$

   a) What is the dimension of A?
   b) What is the dimension of C?
   c) Which of these matrices are vectors?
   d) Which of these matrices are scalars?
   e) Which of these matrices are square matrices?
   f) Find A_{23}.
   g) Find B_{22}.
   h) Find D_{32}.
   i) Find A^t.
   j) Find F^t.
   k) Find $(A^t)_{32}$.
   l) Find $(B^t)_{12}$.
   m) Find $(D^t)_{43}$.

2. Consider the following matrices.

$$A = \begin{pmatrix} 4 & 1 \\ 3 & -2 \end{pmatrix} \qquad B = \begin{pmatrix} 0 & 2 \\ 1 & 4 \end{pmatrix} \qquad C = \begin{pmatrix} 5 & 0 \\ 1 & 8 \\ 2 & 3 \end{pmatrix} \qquad D = \begin{pmatrix} -1 & 2 & 0 \\ 1 & 0 & 7 \end{pmatrix}$$

$$E = \begin{pmatrix} 10 & 4 & -1 \end{pmatrix} \qquad F = \begin{pmatrix} 2 \\ -1 \\ 3 \end{pmatrix} \qquad G = \begin{pmatrix} 1 & -1 & 0 \\ 0 & 2 & -1 \\ 0 & 1 & 1 \end{pmatrix}$$

   Find the sums and products below. If the result is undefined write 'undefined.'

   a) $A - B$

b) $B + D$

c) $C + D^t$

d) $5A$

e) AB

f) BD

g) EC

h) EF

i) FE

j) GD

k) GF

3. (Adapted from an exercise in *'Finite Mathematics and its Applications,'* by Goldstein, Schneider, and Siegel.)

   A bakery makes three types of cookies: I, II, and III. Each type of cookie is made using four ingredients: A, B, C, and D. The number of units of each ingredient used in each type of cookie is given by the matrix M below. The cost per unit of each of the four ingredients (in cents) is given by the matrix C. The selling price of each cookie (in cents) is given by the matrix S. An order is received. The number of each type of cookie that is ordered is given by the matrix R.

$$M = \begin{pmatrix} 1 & 0 & 2 & 4 \\ 3 & 2 & 1 & 1 \\ 2 & 5 & 3 & 1 \end{pmatrix} \qquad C = \begin{pmatrix} 10 \\ 20 \\ 15 \\ 17 \end{pmatrix}$$

$$S = \begin{pmatrix} 175 \\ 150 \\ 225 \end{pmatrix} \qquad R = \begin{pmatrix} 10 & 20 & 15 \end{pmatrix}$$

   For each matrix described below,

   i) state the dimension of the matrix; and

   ii) explain what the entries in the matrix tell you about the bakery;

   (Hint: It would probably be very helpful to label the rows and columns of M, C, S and R above.)

   a) RM

   b) MC

   c) RS

d) $R(S - MC)$

4. (Adapted from an exercise in *'Finite Mathematics and its Applications,'* by Goldstein, Schneider, and Siegel.)

   Three professors teaching the same course have entirely different grading policies. The fraction of A's, B's, C's, D's, and F's that each professor gives to the students in his/her class is shown in the matrix M below. The number of points a grade of A, B, C, D, and F is worth is shown in the matrix P below. The number of students that each professor has is shown in the matrix N below.

$$M = \begin{pmatrix} .25 & .35 & .3 & .1 & 0 \\ .1 & .2 & .4 & .2 & .1 \\ .05 & .1 & .2 & .4 & .25 \end{pmatrix}$$
$$P = \begin{pmatrix} 4 & 3 & 2 & 1 & 0 \end{pmatrix}$$
$$N = \begin{pmatrix} 240 & 120 & 40 \end{pmatrix}$$

   a) Consider the matrix whose entries show the total number of A's, B's, C's, D's and F's that are awarded. Write this matrix as the product of two of the matrices M, M^t, P, P^t, N, and N^t.
   b) Consider the matrix whose entries show the average grade (in points) given by each professor. Write this matrix as the product of two of the matrices M, M^t, P, P^t, N, and N^t.

5. Alice and Betty are doing an experiment to determine how fast a culture of bacteria grows. Initially, the culture contains one hundred cells, and one day later it contains two hundred cells. They plan to take measurements every day for 5 days in a row (giving 6 measurements in all). Alice thinks the culture will grow linearly, so she predicts that the measurements (in hundreds of cells) that they get will be

$$a = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix}.$$

   Betty thinks that the culture will grow exponentially, so she predicts that the measurements will be

$$b = \begin{pmatrix} 1 & 2 & 4 & 8 & 16 & 32 \end{pmatrix}.$$

   The actual measurements turned out to be

$$d = \begin{pmatrix} 1 & 2 & 3 & 6 & 10 & 20 \end{pmatrix}.$$

   Clearly, neither Alice nor Betty was exactly correct. However, who was closer? Explain.

2 Using Matlab like a Graphing Calculator

MATLAB is a high level programming language specifically designed for scientific computation. The advantages of using MATLAB over lower level programming languages is that it can be used effectively without having a deep understanding of the inner workings of the computer and it is relatively easy to do sophisticated graphics.

The easiest way to learn how to use a piece of software is to use it! In these lessons, please do type in everything as instructed, but don't feel limited by what is in the lesson. If you find yourself wondering what MATLAB might do if you typed in such-and-such, find out by typing it in!

2.1 Goals of this lesson

In this lesson you will learn how to

- use MATLAB interactively to perform calculations as you would on a scientific calculator;
- create variables in the workspace that can be scalars, vectors, or matrices;
- create patterned vectors and matrices;
- access and change entries in a vector or matrix;
- perform the usual arithmetic operations with matrices;
- perform element-wise operations with matrices; and
- sketch the graph of a function.

2.2 Using MATLAB as a Scientfic Calculator

When you first open MATLAB, the Command Window appears. In this window you can give commands to MATLAB. The commands are typed at the prompt

```
>>
```

and are executed by MATLAB when you type return. For instance, if you want MATLAB to calculate the value of $0.43^{-1.5}$ you can type

```
>> 0.43^(-1.5)
```

followed by the return key, just as you would on a calculator. MATLAB responds with

```
ans =
    3.5465
```

In these lessons, whatever appears after the prompt `>>` you should type (DO IT!). This should always be followed by the return key. After showing you what to type, we display the output that is produced by MATLAB and that you should see in

the command window. For instance, here is how you can use MATLAB to calculate $5.65(4.3 + 7 - 2)/5.4$:

```
>> 5.64*(4.3 + 7 - 2)/5.4
ans =
    9.7306
```

The number π can be accessed in MATLAB by typing `pi` and the number e is accessed by typing `exp(1)`:

```
>> pi
ans =
    3.1416
>> exp(1)
ans =
    2.7183
```

All the standard functions on a calculator are accessed in a predictable way in MATLAB. Here is a list of functions that are commonly used in calculus and their MATLAB equivalents.

Name	Math notation	Matlab
sine	sin	`sin`
cosine	cos	`cos`
tangent	tan	`tan`
cosecant	csc	`csc`
secant	sec	`sec`
cotangent	cot	`cot`
log base e	ln	`log`
log base 10	$\log_{10}$	`log10`
log base 2	$\log_2$	`log2`
exponential		`exp`
square root	$\sqrt{\ }$	`sqrt`
absolute value	$\lvert\ \rvert$	`abs`

When you use these functions, the input should be enclosed in round brackets. Here are some examples.

```
>> sin(pi/7)
ans =
    0.4339
>> log10(0.001)
ans =
    -3
>> log(exp(-10.3))
ans =
    -10.3000
>> exp(sqrt(9) - 3)
ans =
```

```
    1
```

You will find that you often want to repeat a command or slightly modify a command. To save you typing the command out again, you can use the up and down arrow keys to access previous commands and the right and left arrow keys to edit them. For instance, if you type the up arrow key, then the last command that you entered will appear at the prompt:

```
>> exp(sqrt(9) - 3)
```

Instead of typing return, type the up arrow key again. This time the command prior to that one appears:

```
>> log(exp(-10.3))
```

Now, let's edit this command. Using the left arrow key, move the cursor so that it is between the g of log and the parenthesis, (. (You can also click there with the mouse if you prefer.) Delete the word log and replace it with sin and then type return.

```
>> sin(exp(-10.3))
ans =
    3.3633e-05
```

2.3 Variables, Equals Sign as Assignment, and Suppressing Output

Notice that when MATLAB performs a calculation it stores the result in a variable called `ans`. This is a variable in the workspace that can be accessed and used in the next calculation. You probably have a button on your calculator that accesses the result of the previous calculation; typing `ans` has the same effect in MATLAB. For instance, the following code calculates the value of $1/(5\sin(\pi/7))$ in three steps:

```
>> sin(pi/7)
ans =
    0.4339
>> 5*ans
ans =
    2.1694
>> 1/ans
ans =
    0.4610
```

You can also create your own variables. For instance, when you type

```
>> x = 4
x =
    4
```

MATLAB creates a variable in the workspace called x and stores the value 4 in this variable. A 'variable' for MATLAB is really a location in memory. In other words, MATLAB has located a block of memory that it refers to as x. Whenever `x` is typed now in the workspace, MATLAB will look in that location and read off the value that is stored there. Right now, the value is 4, so if we type

```
>> x^2 + x
```

```
ans =
    20
```

MATLAB calculates the value of $x^2 + x = 4^2 + 4$. We can easily change the value of x. For instance, suppose we want x to have the value -3. We can do this by simply typing:

```
>> x = -3
x =
    -3
```

Now when we calculate $x^2 + x$ we are calculating the value of $(-3)^2 + (-3)$:

```
>> x^2 + x
ans =
    6
```

Variables in MATLAB are case-sensitive. In other words, x and X are different. Since we haven't introduced a variable X in this lesson MATLAB complains when we ask it to do a calculation using X:

```
>> X^2 + X
???  Undefined function or variable 'X'.
```

It can't perform the calculation because it doesn't know what X is.

When we use an equals sign in ordinary mathematics it can mean different things depending on the context. For instance, when we write

$$(x + 3)(x - 2) = 0,$$

the equals sign indicates that we want to find all those values of x that make the value of the expression on the left the same as the value of the expression on the right. In this case, these values are -3 and 2. On the other hand, when we write

$$(x + 3)(x - 2) = x^2 + x - 6,$$

the equals sign here indicates that, whatever the value of x, the value of the expression on the left is the same as the value of the expression on the right. In general, computers aren't very good at dealing with ambiguity and in MATLAB an equals sign can only be used in a very specific way and it has a very specific meaning. In particular it is an *assignment* command; whenever you type an equals sign, it should be preceded, on the left, by the name of a variable and it should be followed, on the right, by an expression that can be evaluated. MATLAB responds by evaluating the expression on the right and storing the result in the variable on the left. (If the variable on the left doesn't already exist in the workspace then it creates it, if it does already exist in the workspace then its old value is lost and is replaced with the value of the expression.) This meaning of the equals sign allows us to type some expressions that look very weird on paper. For instance, consider what happens when we type `x = x + 3` in the Command Window. On paper, this is nonsense. To MATLAB however, it makes perfect sense. It calculates the expression on the right and gets $x + 3 = -3 + 3 = 0$ (since the value of x in this lesson is presently -3) and stores this number in the variable x. So now x has the value 0:

```
>> x = x + 3
x =
    0
```

When we use a variable to calculate a quantity and then store the value obtained back in that same variable, we will say that we have 'overwritten' the old value with the new value. Thus, in the example above, we overwrote x with $x + 3$. You will see that it is very common to do this in scientific computing.

If we type a semicolon after a command but before typing return, MATLAB will execute the command but will suppress the output. In other words, it won't print the output to the screen. For example, type:

```
>> y = 2*x + 5;
>>
```

It looks like MATLAB hasn't done anything, but in fact it has. It calculated the quantity $2x + 5 = 2(0) + 5 = 5$ and stored the result in a variable called y. Thus, if you ask MATLAB what the value of y is or tell it to perform a calculation with y it doesn't complain the way it did with X above:

```
>> y
y =
    5
>> y^2 - 10
ans =
    15
```

Although right now it may seem strange to suppress the output (why would you want MATLAB to calculate something if you didn't want to see the result?), you will see shortly that it is very useful to be able to do this. You will often have MATLAB perform hundreds or thousands of calculations, and you wouldn't want to see hundreds or thousands of numbers being printed out to the screen.

When we create variables in calculus, we often give them suggestive names. For instance, we might call the radius of a circle r, or the distance travelled d. Similarly, when you create variables in MATLAB you will want to give them suggestive names. A name doesn't have to be a single letter only. It can actually be a name. For instance, if a variable represents the radius of a circle you might call it `radius`. However, it's a good idea to avoid using names that are English names. This is because many English names already have a meaning in MATLAB. For instance, if a variable represented a length you might be tempted to call it `length`. However, this is a bad idea, since `length` is a MATLAB command (see below) that calculates the length of a vector. So, it's a good idea to misspell or abbreviate names when creating variables. For instance, you might call the variable `len` instead.

2.4 Creating Variables that are Matrices

The basic variable in MATLAB is a matrix rather than a number. For instance, the variable x that we introduced above is considered by MATLAB to be a 1×1 matrix.

Higher dimensional matrices can be constructed in MATLAB in several ways. The simplest way is to type in the entries one by one; square brackets surround all of the entries, the entries in each row are separated either by a space or by a comma, and the rows are separated by a semi-colon. For instance, to create the variables

$$A = \begin{pmatrix} 0 & 1 \\ 4 & 3 \\ -2 & 0 \end{pmatrix} \qquad B = \begin{pmatrix} 1 & 3 & 0 & -1 \end{pmatrix}$$

$$C = \begin{pmatrix} 2 & -1 & 0 \\ 1 & 3 & 2 \end{pmatrix} \qquad D = \begin{pmatrix} 4 \\ -1 \end{pmatrix}$$

we can type:

```
>> A = [0 1; 4 3; -2 0]
A =
     0   1
     4   3
    -2   0
>> B = [1 3 0 -1]
B =
     1   3   0  -1
>> C = [2, -1, 0; 1, 3, 2]
C =
     2  -1   0
     1   3   2
>> D = [4; -1]
D =
     4
    -1
```

2.5 Creating Patterned Vectors and Matrices

As you will see when we discuss how to graph a function using MATLAB, we often want to create vectors that are very long (it would not be unusual to have vectors that have hundreds or thousands of entries) and whose entries start at one number and increase incrementally until they reach another number. You wouldn't want to have to type out all these numbers, so MATLAB has a way of creating such vectors; if you type $a{:}\,b$ (where a and b are numbers) MATLAB interprets this to be the row vector whose first entry is a and whose entries increase by 1 unit until they get to b. Here are some examples:

```
>> 0:5
ans =
     0   1   2   3   4   5
>> -2.3:0.7
ans =
    -2.3000   -1.3000   -0.3000    0.7000
```

```
>> 4.1:6.7
ans =
    4.1000   5.1000   6.1000
```

If you want a different step size then you can type $a:c:b$. This is a row vector whose first entry is a and whose entries increase by c units until they get to b, as illustrated in the following example:

```
>> 0:.2:1.5
ans =
    0   0.2000   0.4000   0.6000   0.8000   1.0000   1.2000   1.4000
```

The commands `ones` and `zeros` produce a matrix whose entries are all 1's or all 0's respectively. When you use the command you indicate how many rows and columns the matrix should have as shown in the examples below.

```
>> ones(2, 3)
ans =
    1   1   1
    1   1   1
>> zeros(2, 1)
ans =
    0
    0
```

The command `eye` produces an identity matrix. Since identity matrices are always square, you only have to specify the number of rows you want the matrix to have as shown in the example below.

```
>> eye(3)
ans =
    1   0   0
    0   1   0
    0   0   1
```

When you create a large matrix (for instance when you create a vector using the colon notation) it can be difficult to work out exactly how many entries it has. You can do this with the `size` command (or `length` command in the case of a vector). The `size` command tells you the number of rows and columns that a matrix has, the `length` command tells you how many entries a vector has. Here are some examples using these.

```
>> size(A)
ans =
    3   2
>> ones(size(A))
ans =
    1   1
    1   1
    1   1
>> length(B)
ans =
```

```
    4
>> length(3.2:.01:5.6)
ans =
    241
```

If x is a vector then `sum(x)` is the sum of all its entries. If x is a matrix, then `sum(x)` is a row vector consisting of the sum of the entries in each column. Here are some examples:

```
>> sum(B)
ans =
    3
>> sum(A)
ans =
    2    4
```

2.6 Accessing and Changing Entries in a Matrix

If M is a matrix variable in MATLAB then `M(1, 2)` refers to the entry in the first row and second column, in other words it's M_{12}. Similarly, `M(3, 1)` refers to the entry M_{31} etc. To illustrate, consider the matrices A, B and D that you have in your workspace. They should be:

$$A = \begin{pmatrix} 0 & 1 \\ 4 & 3 \\ -2 & 0 \end{pmatrix}, \qquad B = \begin{pmatrix} 1 & 3 & 0 & -1 \end{pmatrix}, \qquad D = \begin{pmatrix} 4 \\ -1 \end{pmatrix}$$

```
>> A(3, 1)
ans =
    -2
>> B(2, 1)
???  Index exceeds matrix dimensions.
```

Notice that, since B has only one row, there is no element B_{21}, so MATLAB complained when we tried to ascertain its value. Notice also that there is no zero'th row or column, as illustrated in the following example.

```
>> A(0,1)
???  Subscript indices must either be real positive integers or
logicals.
```

If M is a row vector then you don't need to specify which row the entry is in, since it must be in the one and only row. In other words you can use `M(2)` as a short-hand for `M(1, 2)`. Similarly, if M is a column vector, `M(3)` means the same thing as `M(3, 1)`. Here are two examples:

```
>> B(3)
ans =
    0
>> D(1)
```

```
ans =
    4
```

We can easily change an individual entry in a matrix by assigning it a new value. In the first example below, we change A_{11} so that it has the value 7, in the second example, we change A_{21} so that it is the square of its previous value.

```
>> A(1, 1) = 7
A =
     7   1
     4   3
    -2   0
>> A(2, 1) = A(2, 1)^2
A =
     7   1
    16   3
    -2   0
```

We can access a whole row or column of a matrix M by using a colon. For instance `M(1, :)` is the row vector consisting of all those entries that lie in the first row and in any column of M. In other words, it is the first row of M. Similarly, `M(:, 2)` is the column vector consisting of all those entries that lie in any row and in the second column of M. In other words, it is the second column of M. Here are two examples:

```
>> A(3, :)
ans =
    -2   0
>> A(:, 1)
ans =
     7
     4
    -2
```

In the examples below we use this notation to change a whole row or column of a matrix. In the first example we return the matrix A to its original value by reconstructing its first column. In the second example we try to change the first row of A so that it is equal to the first column of A. However, since the length of the row is different from the length of the column, MATLAB complains.

```
>> A(:,1) = [0; 4; -2]
A =
     0   1
     4   3
    -2   0
>> A(1,:)  = A(:,1)
???  Subscripted assignment dimension mismatch.
```

You can also add a whole row or column to a matrix. However, the row or column you are adding has to be the correct size so that the result is still a matrix. In the

next example we extend D making it into the 3×2 matrix

$$\begin{pmatrix} 4 & 2 \\ -1 & 2 \\ 2 & 2 \end{pmatrix}$$

Notice, however, that the first way we try to do this it complains because the result would not be a matrix.

```
>> D(:, 2) = 2*ones(3, 1)
???  Subscripted assignment dimension mismatch.
>> D(:, 2) = 2*ones(2, 1)
D =
     4   2
    -1   2
>> D(3, :)  = 2*ones(1, 2)
D =
     4   2
    -1   2
     2   2
```

2.7 Performing Arithmetic Operations with Matries

MATLAB interprets the operations `+`, `-`, `*` and `^` to mean matrix arithmetic. To illustrate, consider the matrices A and C that you have in the workspace. Their values should be:

$$A = \begin{pmatrix} 0 & 1 \\ 4 & 3 \\ -2 & 0 \end{pmatrix} \qquad \text{and} \qquad C = \begin{pmatrix} 2 & -1 & 0 \\ 1 & 3 & 2 \end{pmatrix}.$$

For instance, since A is 3×2 and C is 2×3, AC is defined and is a 3×3 matrix. We can get MATLAB to find its value by typing:

```
>> A*C
ans =
     1   3   2
    11   5   6
    -4   2   0
```

On the other hand, since A and C have different sizes, $A + C$ is not defined and MATLAB complains when we try to evaluate it:

```
>> A + C
???  Error using ==> plus
Matrix dimensions must agree.
```

Similarly, A^2 means A multiplied by itself, which is not defined, since you can't multiply a 3×2 matrix by a 3×2 matrix. We see that MATLAB complains when we try to evaluate it:

```
>> A^2
```

```
??? Error using ==> mpower
Matrix must be square.
```

There is an exception to this. On paper, the expression $A + 5$ is undefined because you can't add a 3×2 matrix to a 1×1 matrix (or scalar). However, MATLAB allows you to add a scalar to a matrix. It interprets the sum to be the matrix that is obtained when the scalar is added to each entry in the matrix:

```
>> A + 5
ans =
    5   6
    9   8
    3   5
```

An apostrophe is used to indicate the transpose of a matrix:

```
>> A' - C
ans =
   -2   5  -2
    0   0  -2
```

2.8 Performing Operations on Each Entry in a Matrix Individually

Although it's wonderful how easy it is to do matrix arithmetic in MATLAB, you will often find that you don't want to do matrix arithmetic, but instead want to perform some kind of arithmetic operation to the individual entries of a matrix or vector. As a general rule, if you precede an operation with a period, then that tells MATLAB to perform that operation on each entry of the matrix individually rather than to perform matrix arithmetic. The examples in this section illustrate this. They use the matrices A, C and D in the workspace whose values are:

$$A = \begin{pmatrix} 0 & 1 \\ 4 & 3 \\ -2 & 0 \end{pmatrix}, \qquad C = \begin{pmatrix} 2 & -1 & 0 \\ 1 & 3 & 2 \end{pmatrix}, \qquad \text{and} \qquad D = \begin{pmatrix} 4 & 2 \\ -1 & 2 \\ 2 & 2 \end{pmatrix}.$$

The expression `A*D` means the matrix product of A and D. This is undefined since the number of columns of A is not equal to the number of rows of D. However, if we precede the `*` with a period and type `A.*D` then each entry in A is multiplied by the corresponding entry in D as shown below:

```
>> A*D
??? Error using ==> mtimes
Inner matrix dimensions must agree.
>> A.*D
ans =
    0   2
   -4   6
   -4   0
```

The operation $.*$ is only defined if the dimensions of the two matrices are the same:

```
>> C*A
ans =
```

```
    −4   −1
     8    10
>> C.*A
???  Error using ==> times
Matrix dimensions must agree.
```

Similarly, `./` means divide corresponding entries and `.^`n means raise each entry to the power of n. The following examples illustrate this. Notice that when we try to divide by zero MATLAB returns `Inf` as the answer, but alerts us with a warning that we may be doing something we weren't expecting to do.

```
>> A./D
ans =
       0       0.5000
   −4.0000    1.5000
   −1.0000         0
>> A./C'
Warning:  divide by zero.
ans =
     0     1
    −4     1
   −Inf    0
>> A.^2
ans =
     0    1
    16    9
     4    0
```

You will never need to type `.+` or `.-` since the matrix interpretation of `+` and `-` already is to add (respectively subtract) coresonding entries.

As a general rule, MATLAB functions such as `sin` or `exp` will accept a vector and not just a number as their input. When the input is a vector, MATLAB evaluates the function at each entry in the vector and returns this whole vector of values. As you will see in the next section, this is very useful for sketching graphs of functions.

```
>> x = 0:(pi/6):pi
x =
    0   0.5236   1.0472   1.5708   2.0944   2.6180   3.1416
>> sin(x)
ans =
    0   0.5000   0.8660   1.0000   0.8660   0.5000   0.0000
```

2.9 Sketching the Graph of a Function

MATLAB sketches functions the way you learnt to do so many years ago: by plotting points. To illustrate this, let's consider the function

$$f(x) = \frac{x}{x^2 + 1}.$$

If you were to plot this function by plotting points you would start by setting up a table similar to the one below.

x	y
-2	-0.4
-1	-0.5
0	0
1	0.5
2	0.4

You would then plot these points and connect the dots. To do this in MATLAB we first create a vector with all of the x-coordinates. You can call this vector whatever you want. We'll call it `x` since that seems like a reasonable choice in the context.

```
>> x = -2:2
x =
    −2   −1   0   1   2
```

Now we need to create the vector that has all of the corresponding y coordinates. Let's do this one step at a time. First notice that

```
>> x.^2
ans =
    4   1   0   1   4
```

is a vector consisting of each coordinate squared. This means that

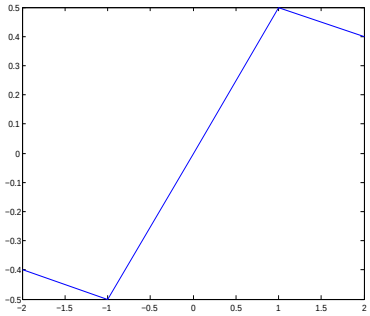
```
>> x.^2 + 1
ans =
    5   2   1   2   5
```

is a vector consisting of each coordinate squared plus one. To get the vector of y-values, we need to take each value in x and divide by its corresponding value in the vector above. In other words we need to dot-divide each entry in x by each entry in the vector above. Thus

```
>> y = x./(x.^2 + 1)
y =
    −0.4000   −0.5000   0   0.5000   0.4000
```

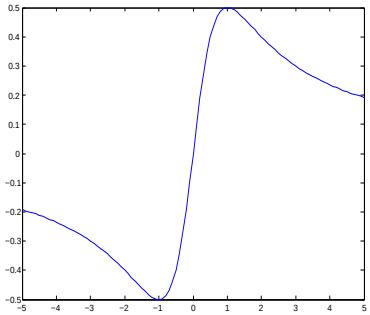
is the vector of y-values. Notice that this looks like the formula for the function where all of the operations have a dot preceding them. We can then use the `plot` command to plot the points:

```
>> plot(x, y)
```



Look for the plot on your screen; it will be in a different window. It's not a very detailed plot of the function and it looks a little funny because we only used a few points. To get a better plot we should use more points that cover a wider range and are closer together:

```
>> x = -5:.1:5;
>> y = x./(x.^2 + 1);
>> plot(x, y)
```



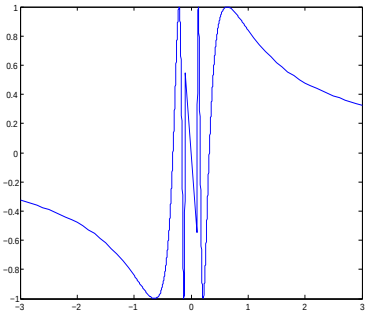
Notice that we used the semicolon to suppress the output when creating the vectors x and y; there was no reason for us to see all of those numbers.

The hardest part of plotting a graph on a computer or calculator is deciding what x-values to use in your plot. There are basically two issues; what range of x values should be used, and how large the spaces should be between the x-values. There is no easy answer to either question. Calculators often use $-10 \leq x \leq 10$ as a default range. This may be a good place to start if you don't otherwise have any idea of what range to use. To determine the spaces between the x-values, remember that the smaller the spaces are, the more realistic the graph will look. A good rule of thumb is to choose the spacing depending on the range, so that there are about 500 points being plotted. In the example below, we sketch the graph of

$$g(x) = \sin(\pi/x).$$

Recall that this graph oscillates infinitely often as x approaches 0. Notice that near 0 we use more points because we want to see these oscillations on the graph, (we'll never see all of them though; we'd have to use infinitely many points for that!), and we don't attempt to sketch the graph too near to $x = 0$.

```
>> x1 = -3:.1:-1;
>> x2 = -1:.001:-.1;
>> x3 = .1:.001:1;
>> x4 = 1:.1:3;
>> x = [x1, x2, x3, x4];
>> y = sin(1./x);
>> plot(x, y)
```

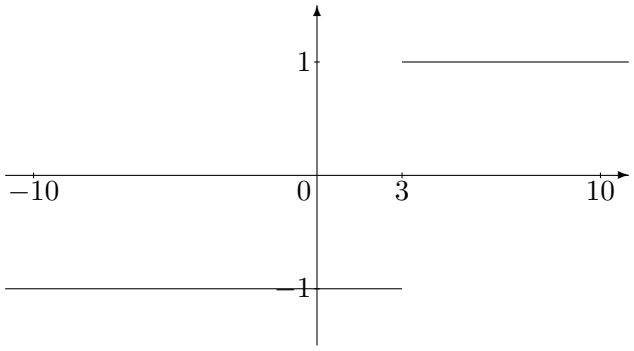


Notice that the graph should be oscillating between -1 and +1 but sometimes it seems to turn around before -1 or before +1. This is because, the x-value where it achieves the -1 or the $+1$ is not included in the x-values being plotted. So, the graph as shown is misleading and doesn't accurately depict the complete graph of the function.

Notice that, when you plot a function using Matlab, it connects the dots. This means that a jump discontinuity in a function will look like a vertical line (or a nearly vertical line, depending on how much space there is between x values in your plot) and if a function has a vertical asymptote, then Matlab will connect the pieces on either side of the asymptote. For instance, consider the function

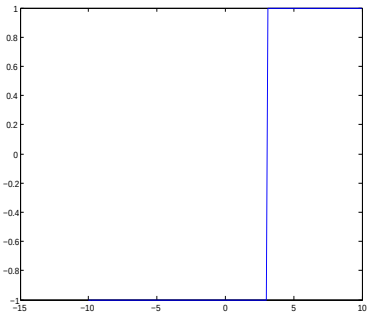
$$h(x) = \frac{|x-3|}{x-3} = \begin{cases} -1 & x < 3 \\ 1 & x > 3 \end{cases}$$

The graph of this function consists of two horizontal half-lines:



However, when we sketch it in Matlab, these two lines are connected by a nearly vertical line:

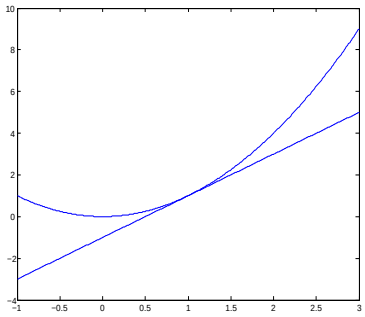
```
>> x = -10.01:.1:10.01;
>> y = abs(x - 3)./(x - 3);
>> plot(x, y)
```



Notice the function h isn't defined at $x = 3$, so to avoid having 3 as one of the entries in our x vector, we started the vector at -10.01 instead of at -10. If you did have 3 as one of the entries in your vector it wouldn't be a big deal though; MATLAB would simply give you a warning message saying that it was being asked to divide by 0 when calculating y and omit that point when it does the plot. Indeed, in this case the plot looks the way it ought to, since MATLAB doesn't connect the two horizontal lines!

Every time you create a new plot, the old plot is lost. If you want to keep the old plot and put the new plot in a new window, type `figure` before creating the new plot. This opens up a new figure window and makes it active. When you then create a new plot, it will appear in this window. Alternatively, if you want the new plot to appear in the same window as the old plot without losing the old plot, type `hold on`. This tells Matlab to superimpose the new plot on the old plot. In the following example, we plot the graph of $y = x^2$ along with its tangent line at $x = 1$. Notice that $y = 2x - 1$ is the equation of this tangent line.

```
>> x = -1:.01:3;
>> y = x.^2;
>> plot(x, y)
>> ytan = 2*x - 1;
>> hold on
>> plot(x, ytan)
```

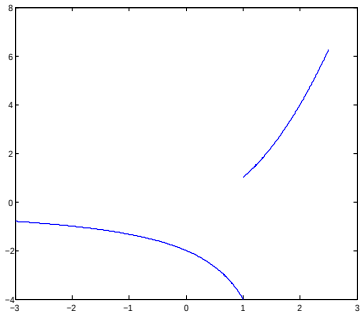


When you use `hold on` the new plot does not have to use the same x values as the old plot. If new x-values are used, Matlab will adjust the window to accommodate the new values. In the next example, we use `hold on` to plot a graph of the piecewise function

$$k(x) = \begin{cases} 4/(x-2) & x \leq 1 \\ x^2 & x > 1 \end{cases}$$

The first command `hold off` undoes the `hold on` command. The effect is that the first plot is created in the active plot window, erasing what was previously there.

```
>> hold off
>> x = -3:.01:1;
>> y = 4./(x - 2);
>> plot(x, y)
>> hold on
>> x = 1:.01:2.5;
>> y = x.^2;
>> plot(x, y)
```



2.10 Miscellany

You may type more than one command on one line. The commands should be separated by either commas or semicolons. Any command that ends with a semicolon will have its output suppressed.

```
>> x = 2, y = cos(.3), z = 3*x*y
x =
    2
y =
    0.9553
z =
    5.7320
>> x = 5; y = cos(.5); z = x*y^2
z =
    3.8508
```

Although MATLAB performs calculations to a very high precision, by default it only prints out 4 decimal places to the screen. If you want to see the results to a higher degree of precision you can type

```
>> format long
```

To return to 4 decimal places, you can type

```
>> format short
```

MATLAB generally leaves a blank line between the commands you type and its response. If you want to omit this line, you can type

```
>> format compact
```

To reinstate the blank lines, type

```
>> format loose
```

MATLB provides online help. One way to get help is to click on the Help menu in the command window. If you know the name of the command you want to use, but are having trouble remembering exactly how to use it or exactly what it does, you can type `help` followed by the name of the command.

When working with MATLAB you will see that you create many variables and may not be able to keep track of all of them. To see what variables you have created in the workspace type

```
>> who
Your variables are:
ans    x    y
```

For more information about these variables you can type

```
>> whos
 Name  Size  Bytes  Class
 ans   1x1       8  double array
 x     1x1       8  double array
 y     1x1       8  double array
Grand total is 2 elements using 16 bytes
```

If you want to clear one or more of these variables from the workspace, you can use the `clear` command. If you type `clear` followed by the name of the variable(s) you would like to remove then MATLAB will remove only those variables. If you type `clear` followed by nothing, then MATLAB will clear all of the variables from the workspace.

```
>> clear x
>> who
Your variables are:
ans    y
>> clear
>> who
```

MATLAB doesn't appear to respond because there are no variables in the workspace to report.

Worksheet 2

1. Use Matlab to calculate the following quantities.

   a) $\dfrac{\pi^2}{4^{1.6} - 3\ln(3.5)}$

   b) $e^{\sin(13\pi/7)}$

   c) $\sqrt{\log_2(58)}$

   d) $(A+B)^t C$ where A, B and C are the matrices

$$A = \begin{pmatrix} 1.23 & -5.41 & 0.05 \\ -10.02 & 3.14 & 5.34 \end{pmatrix}$$
$$B = \begin{pmatrix} 0.54 & -32.16 & 4.34 \\ 4.12 & -32.10 & 6.12 \end{pmatrix}$$
$$C = \begin{pmatrix} 5.42 & 33.11 \\ -10.20 & 6.14 \end{pmatrix}$$

   e) AB where

$$A = \begin{pmatrix} 1 & 2 & 3 & \dots & 100 \\ 1 & 3 & 5 & \dots & 199 \\ 1.1 & 4.1 & 7.1 & \dots & 298.1 \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} 1 & 100 \\ 1 & 99 \\ 1 & 98 \\ \vdots & \vdots \\ 1 & 1 \end{pmatrix}$$

   f) The number of entries in the vector `-345:pi:1035`.

2. Try to answer each of the following questions first without using MATLAB. Then, check your answers using MATLAB.

   a) Suppose you type the following code in the command window.

```
>> x = 1 - 5*2 + 13;
>> x + 5;
```

      A) What is the value of `ans`? Circle your answer.

         i) -60

         ii) -55

iii) 4
iv) 9
v) There's not enough information to determine its value.

B) What is the value of x? Circle your answer.
i) -60
ii) -55
iii) 4
iv) 9
v) There's not enough information to determine its value.

b) Suppose you type the following code in the command window.

```
>> x = 5/4 + 1;
>> x = x + 4;
```

A) What is the value of `ans`? Circle your answer.
i) 2.25
ii) 6.25
iii) 1
iv) 5
v) There's not enough information to determine its value.

B) What is the value of x? Circle your answer.
i) 2.25
ii) 6.25
iii) 1
iv) 5
v) There's not enough information to determine its value.

c) Suppose you type the following code in the command window.

```
>> x^2 - 1 = 0
```

How will MATLAB respond? Circle your answer.

i)
```
x =
     1
```

ii)
```
x =
    -1     1
```

iii)
```
???  x^2 - 1 = 0
Error:  The expression to the left of the equals sign is not
a valid target for an assignment.
```

d) Suppose a and b are variables in the workspace and you type the following code in the command window.

```
>> temp = a; a = b; b = temp;
```

Describe, in words, what effect this has on the values of a and b.

3. Try to answer each of the following questions first without using MATLAB. Then, check your answers using MATLAB.

a) Suppose $x = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \end{pmatrix}$ is a row vector in the workspace. Consider the following commands that could be typed in the command window.

A) `>> x*x`

B) `>> x*x'`

C) `>> x'*x`

D) `>> x.*x`

In each case, choose which of the following would be Matlab's response.

i)
```
ans =
     55
```

ii)
```
ans =
     1   4   9   16   25
```

iii)
```
ans =
     1   2   3   4   5
     2   4   6   8   10
     3   6   9   12  15
     4   8   12  16  20
     5   10  15  20  25
```

iv)
```
???  Error using ==> times
Matrix dimensions must agree.
```

v)
```
???  Error using ==> mtimes
Inner matrix dimensions must agree.
```

b) Suppose x is a column vector in the workspace. Which of the following calculates the value of $||x||$? Circle your answer(s).

i) `>> sqrt(x*x)`

ii) `>> sqrt(x*x')`

iii) `>> sqrt(x'*x)`

iv) `>> sqrt(sum(x.*x))`

c) Suppose

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

is a matrix in the workspace. Find the value of B in each case.

i) `>> B = A*A`
ii) `>> B = A.*A`
iii) `>> B = A(2, 1)`
iv) `>> B = A(:, 2)`
v) `>> B = A(1, :)`

4. Use MATLAB to help you understand what the graph of each function looks like. Once you have understood what the graph looks like, sketch it by hand in the space provided, showing all of its important features: intervals where it's increasing, intervals where it's decreasing, any horizontal and vertical asymptotes, any holes, and any jump discontinuities. Find all of the quantities indicated in parts i) through v) in each case. If the function has none, simply say none.

   i) $\lim_{x\to\infty} f(x)$
   ii) $\lim_{x\to-\infty} f(x)$
   iii) The location of any vertical asymptotes.
   iv) The location of any jump discontinuities.
   v) The location of any holes.

   a) $f(x) = \dfrac{\sin x}{|x|}$

   b) $f(x) = \dfrac{9^x - 7^x}{10^x - 1}$

   c) $f(x) = \dfrac{x}{\sqrt{x^2+1}}$

   d) $f(x) = \dfrac{\tan x - x}{x^3}$

3 Programming in Matlab - Scripts and For Loops

3.1 Goals of this lesson

In this lesson you will learn how to:

- change folders in MATLAB so that you can save files in an appropriate place and organize your work,
- write m-file scripts, and
- write for loops.

You will also further develop your familiarity with MATLAB and start getting a feel for how you can get MATLAB to do things that you might not at first think that you could do.

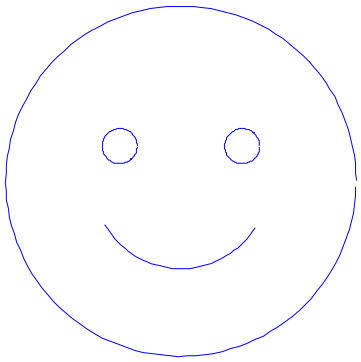
3.2 Changing Folders in MATLAB

In this lesson you will start programming with MATLAB. This means that you won't just be using MATLAB interactively, but will be creating files that need to be saved. Most of you probably don't have MATLAB on your own personal computer, so when you are using MATLAB you'll be working on a public machine. Rather than save your files on the hard drive of a machine that is not your own, we strongly suggest that you purchase a little flash drive (if you don't already have one) that will plug into a USB port on the machine you are working on. You can then save your files on the flash drive and you will always have them with you.

To get MATLAB to save files to your flash drive, first put the flash drive in the USB port and create a folder (or directory) in it called, say, learnmatlab and a folder inside that called Lesson3. Then open up MATLAB. In the top right hand corner of the command window you will see a little square with 3 dots in it. Click on that square and a browser window will open up. In the browser window, navigate to your flash drive and then to the folder in it called learnmatlab and then to the folder in that called Lesson3. Select this folder. You should see its path name written out in the top of the command window. This is now your working folder in MATLAB. Any file that you create in MATLAB will be saved here, and you will also be able to access any of the files that are here.

3.3 Writing an M-file Script

Let's see how we can get MATLAB to draw the smiley shown below.



Notice that the smiley is essentially one circle for the face, a circle for each eye and an arc of a circle for the mouth. To plot all of these circles and the arc on the same plot we'll use the `hold on` command. However, if we make a mistake, say, in plotting the third circle, we'll have to go back and start all over with plotting the first circle again. This could involve a lot of typing and be frustrating. For this reason, it's useful to be able to edit a whole sequence of commands without executing them. You can do this by creating what is known as an **m-file script**. An m-file script is simply a sequence of commands that are saved in a file. (The file is saved with a .m extension. That is why it's called an m-file.) When you type the name of the file at the prompt in the command window, MATLAB looks for a file with that name and, if it finds one, executes the commands in sequence. So to get MATLAB to draw the smiley above, we'll create an m-file script that does this.

In the command window, click on **File** → **New** → **M-file**. This opens up a new window that looks rather like the command window. However, it's not a command window. Actually, it's just an edit window. You can type in the window in the same way you would in a simple word processor: inserting, deleting, moving up and down, cutting and pasting, etc.

To draw the smiley let's start by drawing the face. This is simply a circle. We know that the command `plot(x, y)`, where $x = (x_1, x_2, \ldots x_n)$ and $y = (y_1, y_2, \ldots y_n)$ are vectors, will plot the point (x_1, y_1) and connect it with a straight line to (x_2, y_2) and connect that with a straight line to (x_3, y_3) etc. So, to plot a circle, we want the points (x_i, y_i), $i = 1, 2, \ldots n$ to lie on a circle and rotate around it. The size or location of the circle is irrelevant since MATLAB will scale everything to fit in the figure window, so we'll just draw the unit circle centered at the origin. In other words we should have $(x_1, y_1) = (1, 0)$, $(x_2, y_2) = (\cos\theta, \sin\theta)$, $(x_3, y_3) = (\cos(2\theta), \sin(2\theta))$ etc. where θ is a small angle. We can produce this with the following code. Type this code in the edit window. If you make a mistake typing it, don't worry, simply go back and edit it as you would in a word processor.

```
t = 0:.05:2*pi;
x = cos(t);
y = sin(t);
plot(x, y)
```

Notice that the commands aren't executed when they are typed in. Consciously make a mistake while typing them in and go back and edit it. To check that this code does indeed draw a circle we need to *execute* the code in the file. To do this we first need to save the file. Do this by clicking on **File** → **Save As** and save the file as `smiley.m`. It will be saved in your working folder (learnmatlab/lesson3). Depending on how MATLAB is set up, you will probably see it appear in a section of the command window that lists all the files in your current folder. Now type

```
>> smiley
```

at the prompt in the command window. Matlab searches for the file smiley.m. When it finds it, it executes the sequence of commands that are in it and produces a circle. If you typed the four lines in with no mistakes then Matlab should open up a figure window and plot the circle. However, it doesn't look quite like a circle. It looks like an ellipse. This is because the default scales that MATLAB uses on the x and y axes are not the same. To correct this, go back to the edit window and add the line

```
axis equal
```

at the end. This tells MATLAB to use the same scale for the x and y axes. Save the file again, by clicking on **File** → **Save** and type

```
>> smiley
```

again at the prompt in the command window. Now the figure should be replaced by a figure that looks like a circle. Close the figure window.

Next, let's add the eyes. Since we'll be adding more plots to the same plot, the first thing we need to do is tell Matlab to put the new plots on the same axes as the old plot. Recall that we do this with the command `hold on`. Add the line

```
hold on
```

as the next line in your m-file. To make the eyes we'll draw little circles of radius .1 centered at $(\pm.35, .2)$. To do the right eye we multiply x and y above by .1 and then shift x by .35 and shift y by .2. The following code does this. Add these lines at the bottom of the m-file.

```
x = .1*x + .35;
y = .1*y + .2;
plot(x, y)
```

Save the file again and type

```
>> smiley
```

again at the prompt in the command window. You should see the face with one eye (the right one). Close this figure window before proceeding.

To make the left eye we need to undo the shift we just made to x and then shift it another -.35. In other words, we need to shift x by $-.7$. We don't need to change y. Add the following lines at the bottom of the m-file:

```
x = x - .7;
plot(x, y)
```

Save the file again and type

```
>> smiley
```

again at the prompt in the command window. You should see the face with two eyes now. Close the figure window before proceeding.

We'll make the mouth by drawing an arc of the circle that has radius .5 and whose center is at the origin. Add the following lines at the bottom of the m-file.

```
t = (7*pi/6):.05:(11*pi/6);
x = .5*cos(t);
y = .5*sin(t);
plot(x, y)
```

Save the file again and type

```
>> smiley
```

again at the prompt in the command window. You should see the face with two eyes and a mouth. Close the figure window before proceeding. One final touch will make the smiley even cuter; the command `axis off` removes the axes from the plot. Add this command at the end of your m-file and save the file again. The complete file should now look like:

```
t = 0:.05:2*pi;
x = cos(t);
y = sin(t);
plot(x, y)
axis equal
hold on
x = .1*x + .35;
y = .1*y + .2;
plot(x, y)
x = x - .7;
plot(x, y)
t = (7*pi/6):.05:(11*pi/6);
x = .5*cos(t);
y = .5*sin(t);
plot(x, y)
axis off
```

Save the file and execute it as before by typing `smiley` at the prompt in the command window. You should see a smiley face just like the one above.

3.4 Adding Comments to Your M-files

Anything on a line that appears after a % sign (be it in an m-file or in the command window) is called a comment and is ignored by MATLAB. For instance, when you type

```
>> x = 3 % This assigns the value 3 to the variable x.
x =
    3
```

then MATLAB ignores everything that appears after the % sign and just 'sees' the command `x = 3` so it executes this command assigning the value of 3 to the variable x.

Comments are used in m-files to explain how the code works. Computer code can be very difficult to read; you would be surprised how you can write code one day and not be able to decipher it the next day. So, it's a really good idea to get into the habit of inserting comments that explain the purpose of each piece of code. MATLAB will also ignore blank lines in m-files, so you can use blank lines to group pieces of code together. Go to the window that contains the smiley file. Here is one way that you might insert comments and use blank lines to make the code more readable and to explain how it works:

```
% This file creates a smiley face.

% The following code produces the outline of the face.  The
% outline is the unit circle centered at the origin.
t = 0:.05:2*pi;
x = cos(t); % x is the x-coordinate of each point to be plotted
y = sin(t); % y is the y-coordinate of each point to be plotted
plot(x, y)

% The next command makes the circle look like a circle instead
% of an ellipse.
axis equal

% The next command ensures that the subsequent features are added
% to the picture instead of replacing it.
hold on

% Next we plot the right eye.  The right eye is a circle of radius
% .1 and center (.35, .2)
x = .1*x + .35;
y = .1*y + .2;
plot(x, y)

% Next we plot the left eye.  The left eye is a circle of radius
% .1 and center (-.35, .2)
x = x - .7;
plot(x, y)

% Next we plot the smile.  This is an arc of the circle that has
% radius .5 and whose center is at the origin.
t = (7*pi/6):.05:(11*pi/6);
x = .5*cos(t);
y = .5*sin(t);
plot(x, y)

axis off % this removes the axes from the picture
```

Notice that MATLAB has a very nice editor that recognizes comments and colors them in green so that they stand out. After typing in the comments, save the file and type

```
>> smiley
```

at the prompt in the command window. The file should work exactly as before producing a frowning smiley face.

It's painful to write comments in your files because it's hard to describe what you're doing. Also, in the beginning, you'll probably find that your comments aren't very helpful because you don't understand them when you try to read the code later. However, with practice you'll get much better at it, and it's so important to put comments in your code that you must start practicing it now.

3.5 Introduction to For Loops

One of the great strengths of computers is that they can do repetitive tasks very quickly. One of the easiest ways to tell MATLAB to repeat a task a fixed, predetermined, number of times, is to use a for loop. The structure of a for loop is as follows:

`for` *variable* `=` *expression*
  statements
`end`

where *variable* is the name of a variable, *expression* is usually a vector, and the *statements* are one or more MATLAB commands. MATLAB executes the loop by first setting the variable equal to the first entry in the vector. It then executes the commands. When it comes to `end` it returns to the `for` line and sets the variable equal to the next entry in the vector. It then executes the commands again. It continues until the variable is equal to the last entry in the vector after which it will proceed with any commands that appear after the `end` or will wait for another command. In addition, a for loop is often preceded by one or more commands that initialize the quantities in the loop.

Although this may seem confusing right now, it will hopefully become clear after a few examples.

Example 1: As a first example, consider the following code that consists of an initialization statement, `som = 1`, followed by a for loop, followed by a request for the value of the variable `som`.

```
>> som = 1;
>> for i = 2:100
som = som + 1/i;
end
>> som
som =
    5.1874
```

Let's look closely at how this code works. As you know, the first command,

`som` $= 1$;

creates a variable called `som` and sets it equal to 1. This *initializes* the loop. The `for` statement in the next command tells MATLAB that it is entering a loop (a sequence of commands that will be performed over and over again). It creates a variable called i and sets it equal to the first entry in the vector `2:100`, namely 2. It then performs the commands that occur between the `for` statement and the `end` line. In this case there is only one, namely `som = som + 1/i;`. This is an assignment command; it evaluates the expression on the right and sets it equal to the variable on the left. Since the value of `som` at this point is 1 and the value of i is 2, the expression on the right has the value $1 + 1/2 = 1.5$, so the variable `som` is assigned the value 1.5. When MATLAB comes to the `end` statement, it returns to the `for` line, and sets i equal to the next entry in the vector `2:100`, namely 3. It then performs the command `som = som + 1/i`. This time, `som` has the value 1.5 and i has the value 3, so the variable `som` is assigned the value $1.5 + 1/3 = 1.8333$. MATLAB then goes back to the `for` statement and sets i equal to the next entry in the vector `2:100`, namely 4. This time, when it comes to the command `som = som + 1/i`, `som` has the value 1.8333 and i has the value 4, so the variable `som` is assigned the value $1.8333+1/4 = 2.0833$. MATLAB continues looping and updating the value of `som` each time until $i = 100$. After performing the command `som = som + 1/i` when $i = 100$ the loop ends, since there are no more values left for i, and MATLAB waits for another command. The following table summarizes this process:

Initialization	`som` $= 1$
$i = 2$	`som` $= 1 + 1/2$
$i = 3$	`som` $= (1 + 1/2) + 1/3$
$i = 4$	`som` $= (1 + 1/2 + 1/3) + 1/4$
$i = 5$	`som` $= (1 + 1/2 + 1/3 + 1/4) + 1/5$
$\vdots$	$\vdots$
$i = 100$	`som` $= (1 + 1/2 + 1/3 + \ldots + 1/99) + 1/100$

We see that when the code has been executed the value of `som` is the sum of the reciprocals of the first 100 integers. In other words

$$1 + 1/2 + 1/3 + \ldots + 1/100 = 5.1874.$$

When you write for loops in an m-file, it is a good idea to write them as we did above, with the `for` statement on the first line, followed by the commands, followed by `end` on a line on its own. Moreover, it's really good to respect the indentation that MATLAB suggests to you as you type it in. However, if you are writing a for loop in the command window, you can type it all on one line if you wish. In this

case, a comma follows the `for` statement, the commands are separated as usual with either a comma or a semicolon (depending on whether or not you want the output to be suppressed) and `end` is typed at the end. For instance, we could have typed the example above as:

```
>> som = 1;
>> for i = 2:100, som = som + 1/i; end
>> som
som =
   5.1874
```

Notice that the variable i used in the code above, could have had any name at all. It is a dummy variable, in the sense that it doesn't really have any intrinsic meaning in and of itself; it is only used to determine what the next calculation should be. This variable is commonly called i, j, k, ii, jj, kk, n or m. Other choices are also fine. However, be careful not to call it something that already has some other meaning (for example, it would have been a really bad idea to call it `som` in the example above). Also notice that there are many different ways to write a for loop for the same calculation. In particular, there are different ways to initialize the calculation and there are many choices for the vector, *expression* . The code below illustrates another valid way to get MATLAB to calculate the sum of the reciprocals of the first 100 integers. In this code the loop is initialized with `som` $= 0$ and the dummy loop variable is called n instead of i:

```
>> som = 0;
>> for n = 1:100, som = som + 1/n; end
>> som
som =
   5.1874
```

Here is the summary table showing how the loop progresses in this version of the code.

Initialization	`som` $= 0$
$n = 1$	`som` $= 0 + 1/1 = 1$
$n = 2$	`som` $= 1 + 1/2$
$n = 3$	`som` $= (1 + 1/2) + 1/3$
$n = 4$	`som` $= (1 + 1/2 + 1/3) + 1/4$
$\vdots$	$\vdots$
$n = 100$	`som` $= (1 + 1/2 + 1/3 + \ldots + 1/99) + 1/100$

Example 2: In the next example, consider the following code. Type this code in an m-file and save the file as rx.m in the folder learnmatlab/lesson3 on your flash drive. Notice that the code consists of two initialization statements followed by a for loop.

```
len = length(x);
r = ones(1, len - 1);
```

```
for i = 1:(len - 1)
  r(i) = x(i+1)/x(i);
end
```

When you type

```
>> rx
```

at the prompt in the command window the first thing MATLAB is asked to do is find the length of the vector `x` and assign that length to the variable `len`. It will return an error message if there is no variable `x` in the workspace or if `x` is not a vector. So, to understand the code, let's create a vector `x` in the workspace:

```
>> x = 1:6;
```

Now let's see what happens when we execute the m-file:

```
>> rx
>> r
r =
    2.0000    1.5000    1.3333    1.2500    1.2000
```

We can see that `r` is a vector of length five and we see what its entries are, but how is it related to `x`? Let's work through the commands in the m-file to find out. In the first line the variable `len` is set equal to the length of `x` which in this case is 6:

```
>> len
len =
    6
```

In the second line, `r` is set equal to a $1 \times (6-1)$ matrix (in other words a row vector of length 5) whose entries are all equal to 1. So at this point in the execution, the variable `r` has the value $(1, 1, 1, 1, 1)$. In the next line MATLAB enters a for loop. The dummy loop variable is called `i` and it will take on the values 1, 2, 3, 4, 5 as MATLAB progresses through the loop. The first time through the loop, the value of `i` is 1. The next command is an assignment command. The value of the expression on the right-hand side is `x(2)/x(1)` which is $2/1 = 2$ and this is assigned to `r(1)`. So the vector `r` now has the value $(2, 1, 1, 1, 1)$. The next line is `end`, so MATLAB returns to the `for` line and sets the dummy variable `i` equal to the next value in the vector, namely, 2. It then executes the assignment command assigning the value `x(3)/x(2)` $= 3/2 = 1.5$ to `r(2)`. So the vector `r` now has the value $(2, 1.5, 1, 1, 1)$. The next time through the loop, `r(3)` gets assigned the value `x(4)/x(3)` $= 4/3 = 1.333$. The next time through the loop, `r(4)` gets assigned the value `x(5)/x(4)` $= 5/4 = 1.25$, and the last time through the loop, `r(5)` is assigned the value `x(6)/x(5)` $= 6/5 = 1.2$. Thus we see that the entries in `r` are the ratios of the successive entries in `x`.

In summary, what the file rx.m does, is create a vector `r` whose entries consist of the ratios of successive elements in the vector `x`. Let's check that this is correct by starting with a different vector `x`:

```
>> x = 10:-2:2
x =
   10    8    6    4    2
>> rx
```

```
>> r
r =
    0.8000    0.7500    0.6667    0.5000
```

Looks correct!

3.6 Guidelines for Writing Your Own For Loops

As we have seen, when you want MATLAB to repeat a task a fixed, predetermined number of times it might be appropriate to use a for loop. When you write a for loop, you have to choose:

1. how to initialize the loop (if necessary), and
2. what the dummy vector should be.

When choosing the dummy vector you need to pay attention to:

1. how the values in the dummy vector are related to the calculation you want to be performed at each iteration in the loop, and
2. what the first and last values in the dummy vector should be.

We illustrate these ideas with the following two examples.

Example 3: In this example we will see how to write a for loop that will calculate the following sum:

$$(1 + 1/2)(1 + 1/4)(1 + 1/6)(1 + 1/8)(1 + 1/10) \dots (1 + 1/100).$$

Notice that to calculate this by hand we would start by calculating $1+1/2 = 1.5$. Then, we'd multiply this number $1 + 1/4 = 1.25$ to get $1.5 \times 1.25 = 1.875$. Then we'd multiply this number by $1 + 1/6 = 1.1667$ to get $1.875 \times 1.1667 = 2.1875$. This process continues until we're done. This is a repetitive process that will be performed a fixed number of times. This tells us that the way to get MATLAB to do this calculation is with a for loop.

What we are calculating is a product so let's call it `prod` for a lack of better name. Our first 'estimate' of `prod` is $(1 + 1/2)$. This could be our initialization outside of the loop. Then, the first time through the loop, the value of `prod` will be updated by multiplying it by $(1 + 1/4)$, so that it's new value becomes $(1 + 1/2)(1 + 1/4)$. The next time through the loop its value will be updated again by multiplying it by $1 + 1/6$ so that it becomes $(1 + 1/2)(1 + 1/4)(1 + 1/6)$, etc. Let's summarize this in a table.

Initialization	`prod` $= 1 + 1/2$
1st time	`prod` $=$ `prod` $* (1 + 1/4) = (1 + 1/2)(1 + 1/4)$
2nd time	`prod` $=$ `prod` $* (1 + 1/6) = (1 + 1/2)(1 + 1/4)(1 + 1/6)$
3rd time	`prod` $=$ `prod` $* (1 + 1/8) = (1 + 1/2)(1 + 1/4)(1 + 1/6)(1 + 1/8)$
$\vdots$	$\vdots$
last time	`prod` $=$ `prod` $* (1 + 1/100) = (1 + 1/2)(1 + 1/4)(1 + 1/6) \ldots (1 + 1/100)$

At this point we have chosen how to initialize the loop, so what we need to do next is choose what the dummy vector is going to be. Each time through the loop one command is executed, namely, 'replace `prod` by `prod` times an appropriate quantity.' In other words, the code will look as follows:

```
>> prod = (1 + 1/2);
>> for k = expression
prod = prod*quantity
end
```

where *expression* is the dummy vector that we are trying to determine, and *quantity* changes each time MATLAB goes through the loop, so it needs to be expressed in terms of the dummy vector. The first time through the loop *quantity* should be $1 + 1/4$. The second time it should be $1 + 1/6$, the third time by $1 + 1/8$ etc. So, a simple choice for the dummy vector would be $k = (4, 6, 8, \ldots 100)$. The value of *quantity* would then simply be `1 + 1/k`. With this choice the code becomes:

```
>> prod = (1 + 1/2); % the initialization
>> for k = 4:2:100, % the dummy vector (4, 6, 8, ... 100)
prod = prod*(1 + 1/k); % the calculation
end
>> prod
prod =
    8.0385
```

It looks like the product is equal to 8.0385.

How can we be sure that MATLAB is doing the correct calculation? There are a number of ways we can check our loop. One way is to do the same loop with fewer terms, then compare the result with those few terms explicitly calculated by hand:

```
>> prod = (1 + 1/2);
>> for k = 4:2:8, prod = prod*(1 + 1/k); end
>> prod
prod =
    2.4609
>> (1 + 1/2)*(1 + 1/4)*(1 + 1/6)*(1 + 1/8)
ans =
    2.4609
```

Looks like that's okay. Another way is to remove the semi-colon from the calculation of `prod` so that you can see the result of the calculation each time through the loop. You can check if these are correct.

```
>> prod = (1 + 1/2);
>> for k = 4:2:100, prod = prod*(1 + 1/k), end
prod =
    1.8750
prod =
    2.1875
prod =
    2.4609
prod =
    2.7070
     ⋮
prod =
    8.0385
```

Notice that the value of `prod` the first time through the loop is 1.8750 and that $(1 + 1/2)(1 + 1/4) = 1.8750$ so this is correct. Similarly, the $(1 + 1/2)(1 + 1/3)(1 + 1/4) = 2.1875$ which is the value of `prod` the second time through the loop. So, it looks like the program is doing the correct thing. Be careful doing this if your loop contains many terms; your screen will be overrun with numbers! Finally, to check your code you can write out a summary table of the loop as below.

Initialization	`prod` $= 1 + 1/2$
$k = 4$	`prod` $= (1 + 1/2)(1 + 1/4)$
$k = 6$	`prod` $= (1 + 1/2)(1 + 1/4)(1 + 1/6)$
$k = 8$	`prod` $= (1 + 1/2)(1 + 1/4)(1 + 1/6)(1 + 1/8)$
⋮	⋮
$k = 100$	`prod` $= (1 + 1/2)(1 + 1/4)(1 + 1/6) \ldots (1 + 1/100)$

It looks like it is doing what we want and we have confidence that the complete product is indeed 8.0385.

Remember that the code we wrote above isn't the only way to get MATLAB to do this calculation. Indeed, when a loop is supposed to calculate the product of many terms it is typical to initialize the value of the product to be 1 outside of the loop. Also, the dummy vector is often taken to be a vector that starts at 0 or 1 and increases by 1 each time, so that it is either $(0, 1, 2, \ldots)$ or $(1, 2, 3, \ldots)$. In this case, you can think of the value of the dummy variable as the number of times the loop has been iterated. So, let's see what the code would look like if we initialized `prod` to be 1 outside of the loop and had $k = (1, 2, 3, \ldots)$. The table showing how the loop should progress would look as follows:

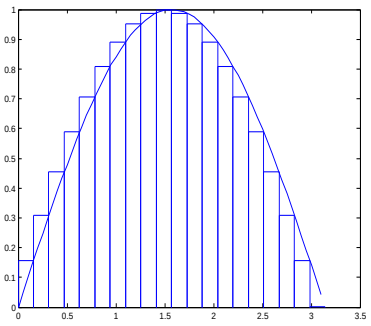
Initialization	prod $= 1$
$k = 1$	prod $=$ prod $* (1 + 1/2) = (1 + 1/2)$
$k = 2$	prod $=$ prod $* (1 + 1/4) = (1 + 1/2)(1 + 1/4)$
$k = 3$	prod $=$ prod $* (1 + 1/6) = (1 + 1/2)(1 + 1/4)(1 + 1/6)$
$\vdots$	$\vdots$
$k =?$	prod $=$ prod $* (1 + 1/100) = (1 + 1/2)(1 + 1/4)(1 + 1/6) \ldots (1 + 1/100)$

Notice that here we need to work out how the multiplicative factors are related to the value of `k` and what the final value of `k` should be. When $k = 1$ the factor is $(1 + 1/2)$, when $k = 2$ it is $(1 + 1/4)$, when $k = 3$ it is $(1 + 1/6)$ etc. It looks like the multiplicative factor is always $(1 + 1/(2k))$. At the last iteration the factor is $(1 + 1/100)$, so on this iteration $2k = 100$ so $k = 50$. Thus, we get the following code:

```
>> prod = 1;
>> for k = 1:50, prod = prod*(1 + 1/(2*k)); end
>> prod
prod =
    8.0385
```

Example 4: In this next example we will use a for loop to get MATLAB to draw the picture shown below that illustrates the right hand rule with 20 intervals to evaluate

$$\int_0^{\pi} \sin x \, dx$$



We'll create this code in an m-file where we can test it piece by piece as we develop it. Recall that to do this you need to click on **File** → **New** → **M-file** in the command window. This opens up a new edit window.

First we need to draw the graph of $\sin x$ from $x = 0$ to $x = \pi$. We do this by creating a vector of x-values, the corresponding vector of y=values and then using the plot command. Type the following commands in your edit window.

```
x = 0:.05:pi;
y = sin(x);
```

```
plot(x, y)
```

Click on **File** → **Save As** and save the file as `rhr.m`. Check that the file does indeed do what you want it to do by typing

```
>> rhr
```

at the prompt in the command window. You should see the graph of $\sin x$ from $x = 0$ to $x = \pi$.

Now we need to draw the 20 rectangles. First notice that the rectangles will be added on top of the graph, so before drawing them we need to add the command

```
hold on
```

as the next line in the m-file. Notice that drawing the rectangles is a highly repetitive task; first we draw the first rectangle, then we draw the second rectangle, then we draw the third rectangle etc. Thus, it would be appropriate to code this with a for loop.

Each rectangle consists of three lines: a vertical line to the left, a vertical line to the right and a horizontal line across the top. The first rectangle stretches from $x = 0$ to $x = \pi/20$ and has height $\sin(\pi/20)$. The second rectangle stretches from $x = \pi/20$ to $x = 2\pi/20$ and has height $\sin(2\pi/20)$. The third rectangle stretches from $x = 2\pi/20$ to $x = 3\pi/20$ and has height $\sin(3\pi/20)$ etc. The last rectangle stretches from $x = 19\pi/20$ to $x = \pi$ and has height $\sin\pi = 0$. Each iterate of the loop will draw a rectangle. Let's summarize this in the table below:

1st time	Draw the vertical line from $(0, 0)$ to $(0, \sin(\pi/20))$ Draw the horizontal line from $(0, \sin(\pi/20))$ to $(\pi/20, \sin(\pi/20))$ Draw the vertical line from $(\pi/20, 0)$ to $(\pi/20, \sin(\pi/20))$
2nd time	Draw the vertical line from $(\pi/20, 0)$ to $(\pi/20, \sin(2\pi/20))$ Draw the horizontal line from $(\pi/20, \sin(2\pi/20))$ to $(2\pi/20, \sin(2\pi/20))$ Draw the vertical line from $(2\pi/20, 0)$ to $(2\pi/20, \sin(2\pi/20))$
3rd time	Draw the vertical line from $(2\pi/20, 0)$ to $(2\pi/20, \sin(3\pi/20))$ Draw the horizontal line from $(2\pi/20, \sin(3\pi/20))$ to $(3\pi/20, \sin(3\pi/20))$ Draw the vertical line from $(3\pi/20, 0)$ to $(3\pi/20, \sin(3\pi/20))$
$\vdots$	$\vdots$
last time	Draw the vertical line from $(19\pi/20, 0)$ to $(19\pi/20, \sin(20\pi/20))$ Draw the horizontal line from $(19\pi/20, \sin(20\pi/20))$ to $(20\pi/20, \sin(20\pi/20))$ Draw the vertical line from $(20\pi/20, 0)$ to $(20\pi/20, \sin(20\pi/20))$

To choose the dummy vector, look at the pattern of x and y coordinates at each iteration. In the first iterate the x-coordinates being used are 0 and $\pi/20$ and the y coordinates are 0 and $\sin(\pi/20)$. In the second iterate the x-coordinates being used are $\pi/20$ and $2\pi/20$ and y coordinates are 0 and $\sin(2\pi/20$, etc. So let's take the dummy vector to be $xr = (\pi/20, 2\pi/20, 3\pi/30 \ldots 20\pi/20)$. Then at each iterate the x-coordinates being used will be $xr - \pi//20$ and xr and the y coordinates will be 0 and $\sin(xr)$. Thus the loop will look like the following. Type it in your m-file.

```
for xr = (pi/20):(pi/20):pi
  xl = xr - pi/20;
  y = sin(xr);
  plot([xl, xl], [0, y])
  plot([xl, xr], [y, y])
  plot([xr, xr], [0, y])
end
```

Save the file and type

```
>> rhr
```

at the prompt in the command window. You should see the full picture shown above.

3.7 Nesting For Loops

You can nest two (or more) for loops inside of each other. For instance, suppose we want to construct the matrix

$$A = \begin{pmatrix} 2 & 2^{1/2} & 2^{1/3} & 2^{1/4} & \dots & 2^{1/50} \\ 3 & 3^{1/2} & 3^{1/3} & 3^{1/4} & \dots & 3^{1/50} \\ 4 & 4^{1/2} & 4^{1/3} & 4^{1/4} & \dots & 4^{1/50} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 20 & 20^{1/2} & 20^{1/3} & 20^{1/4} & \dots & 20^{1/50} \end{pmatrix}$$

Notice that A is a 19×50 matrix whose ij'th entry is $A_{ij} = (i+1)^{1/j}$. We will construct A by assigning the value $i^{1/j}$ to each entry A_{ij} for each value of $i = 1, 2, \dots 19$ and $j = 1, 2, \dots 50$. To do this, we need A to already be a matrix of the correct size; it doesn't matter what its entries are, since we will reassign all of those, we just need it to be the correct size. So we start by typing

```
>> A = zeros(19, 50);
```

to set A to be a 19×50 matrix. (It would have been equally good to type `A = ones(19, 50)`.) Now, if we were to assign the values individually we could type:

```
>> A(1,1) = 2;
>> A(1,2) = 2^(1/2);
>> A(1,3) = 2^(1/3);
```

etc. until we got to

```
>> A(1,50) = 2^(1/50);
```

Then we would continue with the second row and type:

```
>> A(2,1) = 3;
>> A(2,2) = 3^(1/2);
>> A(2,3) = 3^(1/3);
```

etc. until we got to

```
>> A(2,50) = 3^(1/50);
```

Then we would continue with the third row etc. Notice that in the first set of commands we would assign the values of A_{1j} for $j = 1, 2 \dots 50$. In the second set of

commands we would assign the values of A_{2j} for $j = 1, 2, \ldots 50$. In the third set of commands we would assign the values of A_{3j} for $j = 1, 2, \ldots 50$. This continues until the last set of commands where we would assign the values of $A_{20,j}$ for $j = 1, 2, \ldots 50$. Thus, we would be repeating an action 19 times, so instead of this we code this with a loop where the dummy variable goes from 1 to 19 in steps of size 1:

```
>> for i = 1:19
statements;
end
```

where *statements* are the commands that assign the correct values to A_{ij} for $j = 1, 2, \ldots 50$. We could type these commands out one by one. They would be:

```
A(i,1) = (i+1);
A(i,2) = (i+1)^(1/2);
A(i,3) = (i+1)^(1/3);
```

etc. until

```
A(i,50) = (i+1)^(1/50);
```

However, again we see that this is essentially an action that is repeated 50 times, so it could be compactly coded with a for loop:

```
for j = 1:50
A(i,j) = (i+1)^(1/j);
end
```

Thus, the complete code consists of two nested for loops as shown below.

```
>> A = zeros(19,50);
>> for i = 1:19
for j = 1:50
A(i,j) = (i+1)^(1/j);
end
end
```

The summary table below shows how MATLAB cycles through the nested loops.

$i = 1$	$j = 1$	$A_{11} = 2^{1/1} = 2$
	$j = 2$	$A_{12} = 2^{1/2}$
	$j = 3$	$A_{13} = 2^{1/3}$
	$\vdots$	$\vdots$
	$j = 50$	$A_{1,50} = 2^{1/50}$
$i = 2$	$j = 1$	$A_{21} = 3^{1/1} = 3$
	$j = 2$	$A_{22} = 3^{1/2}$
	$j = 3$	$A_{23} = 3^{1/3}$
	$\vdots$	$\vdots$
	$j = 50$	$A_{2,50} = 3^{1/50}$
$\vdots$	$\vdots$	$\vdots$
$i = 19$	$j = 1$	$A_{19,1} = 20^{1/1} = 3$
	$j = 2$	$A_{19,2} = 20^{1/2}$
	$j = 3$	$A_{19,3} = 20^{1/3}$
	$\vdots$	$\vdots$
	$j = 50$	$A_{19,50} = 20^{1/50}$

Worksheet 3

1. First try to answer these questions without using MATLAB. Then use MATLAB to check your answers.

   a) Consider the code below.

   ```
   s = 0;
   for n = 0:100, s = s + 1/2^n; end
   ```

   i) Fill in the blanks in the summary table of the loop below.

Initial	$s = 0$
$n = 0$	$s =$
$n = 1$	$s =$
$n = 2$	$s =$
$n = 3$	$s =$
$\vdots$	$\vdots$
$n = 100$	$s =$

   ii) Which of the following is the value of s after the loop has been completed? Circle your answer.

   I) $1/100^2$

   II) $1 + 1/4 + 1/9 + 1/16 + \ldots 1/10,000$

   III) $1/2^{100}$

   IV) $1 + 1/2 + 1/4 + \ldots + 1/2^{100}$

   V) $1/2 + 1/4 + 1/8 + \ldots + 1/2^{100}$

   b) Suppose $x = (x_1, x_2, \ldots x_n)$ is a vector in the workspace. Consider the code below.

   ```
   n = length(x);
   s = 0;
   for i = 1:(n - 1), s = s + abs(x(i) - x(i+1)); end
   ```

   i) Fill in the blanks in the summary table of the loop.

Initial	$s = 0$
$i = 1$	$s =$
$i = 2$	$s =$
$i = 3$	$s =$
$i = 4$	$s =$
$\vdots$	$\vdots$
$i = n - 1$	$s =$

ii) What would the final value of s be if $x = (1, 3, 5, 7, 9, 11)$?

iii) What would the final value of s be if $x = (2, 8, 4, 1, 7)$?

2. First try to do these problems without using MATLAB. Then, use MATLAB to check your answers.

a) Consider the codes shown in i) to iv) below. In which one(s) will the variable `s` finish with the value

$$(1)(2) + (3)(4) + (5)(6) + \ldots (299)(300)?$$

Circle all those that apply.

i)
```
s = 0;
for i = 1:300, s = s + i*(i + 1); end
```

ii)
```
s = 0;
for dv = 1:2:299, s = s + dv*(dv + 1); end
```

iii)
```
s = 0;
for n = 2:300, s = s + (n-1)*n; end
```

iv)
```
s = 0;
for k = 1:150, s = s + (2*k - 1)*2*k; end
```

v)
```
s = 2;
for j = 1:150, s = s + (2*j + 1)*2*(j+1); end
```

b) Suppose x is a row vector in the workspace. Which of the following codes will result in the variable `norm` taking on the value $||x||$? Circle all those that apply. *Hint: Recall that if* $x = (x_1, x_2, \ldots x_n)$ *then*

$$||x|| = \sqrt{x_1^2 + x_2^2 + \ldots x_n^2}.$$

i)
```
norm = 0;
for i = x, norm = norm + i^2; end
norm = sqrt(norm);
```

ii)
```
norm = 0;
for i = 1:length(x), norm = x(i)^2; end
norm = sqrt(norm);
```

iii)
```
norm = x(1)^2;
for i = 1:(length(x) - 1), norm = norm + x(i)^2; end
norm = sqrt(norm);
```

iv)
```
norm = x(1)^2;
for i = x(2:length(x)), norm = norm + x(i)^2; end
norm = sqrt(norm);
```

v)
```
norm = sqrt(x*x');
```

c) Suppose A is a 10×20 matrix in the workspace. Which of the following codes will assign the value

$$\begin{pmatrix} 1 & 2 & 3 & 4 & \ldots & 20 \\ 2 & 4 & 6 & 8 & \ldots & 40 \\ 3 & 6 & 9 & 12 & \ldots & 60 \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 10 & 20 & 30 & 40 & \ldots & 200 \end{pmatrix}$$

to A? Circle all that apply.

i)
```
for i = 1:10, for j = 1:20, A(i,j) = i*j; end, end
```

ii)
```
for i = 1:10, A(i,:)  = i:i:(20*i); end
```

iii)
```
for n = 1:20, A(:,n) = n:n:(10*n); end
```

iv)
```
A = 0;
for n = 1:20, for m = 1:10, A = A + n*m; end, end
```

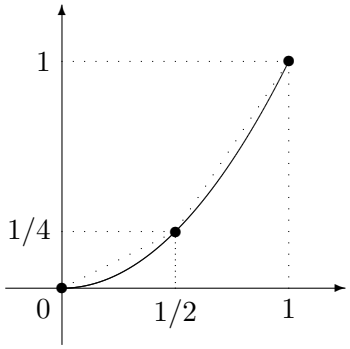
v)
```
x = 1:20;
for i = 1:10, A(i,:)  = i*x; end
```

3. a) Write a script m-file to estimate the value of

$$\int_0^1 e^{-x^2}\, dx$$

using the trapezoidal rule with $n = 20$. What is the estimate of the value of the integral?

b) Consider the curve $y = x^2$ between $x = 0$ and $x = 1$ shown below. In this problem we want to estimate the length of this curve.



Notice first that the length of this curve is greater than $\sqrt{2} = 1.4142$ (the length of the straight line connecting $(0, 0)$ and $(1, 1)$) and is less than 2 (the sum of the horizontal distance between $(0, 0)$ and $(1, 0)$ and the vertical distance between $(1, 0)$ and $1, 1)$). To find the length of the curve more precisely, we could divide the curve into two pieces; the piece from $(0, 0)$ to $(1/2, 1/4)$ and the piece from $(1/2, 1/4)$ to $(1, 1)$ (see the picture above). The length of the first piece is approximately the length of the straight line between $(0, 0)$ and $(1/2, 1/4)$:

$$\sqrt{(1/2 - 0)^2 + (1/4 - 0)^2} = 0.5590.$$

Similarly, the length of the second piece is approximately

$$\sqrt{(1 - 1/2)^2 + (1 - 1/4)^2} = 0.9014.$$

Thus, the sum of these two lengths,

$$0.5590 + 0.9014 = 1.4604,$$

is an estimate of the length of the curve. We would expect the true length of the curve to be slightly larger than this. To get a better estimate we could divide the curve into more pieces.

i) Imagine dividing the curve up into 10 pieces given by the points $(0, 0)$, $(.1, .01)$, $(.2, .04)$, ..., $(1, 1)$. Write a script m-file that finds the sum of the straight line distances between the successive points:

$$\begin{aligned}
&\sqrt{(0 - .1)^2 + (0 - .01)^2} \\
&+ \sqrt{(.1 - .2)^2 + (.01 - .04)^2} \\
&+ \ldots \\
&+ \sqrt{(.9 - 1)^2 + (.81 - 1)^2}
\end{aligned}$$

What estimate does this give you for the length of the curve?

ii) Now imagine dividing the curve up into 100 pieces given by the points $(0, 0)$, $(.01, .0001)$, $(.02, .0004)$, ..., $(1, 1)$. Modify your m-file in part a) to find the sum of the straight line distances between these successive points:

$$\begin{aligned}&\sqrt{(0-.01)^2+(0-.0001)^2}\\&+\sqrt{(.01-.02)^2+(.0001-.0004)^2}\\&+\ldots\\&+\sqrt{(.99-1)^2+(.9801-1)^2}\end{aligned}$$

What estimate does this give you for the length of the curve?

4 Function M-files

4.1 Goals of this lesson

In this lesson you will learn

- what a function m-file is;
- how to write function m-files;
- how variables are handled in function m-files;
- how to include help lines in your function m-files;
- how to make your function m-files work when inputs are vectors; and
- techniques for debugging and testing your code.

In this lesson you will be creating files that you will need to save somewhere. So, insert your flash drive into the USB port and inside the learnmatlab folder create a folder called Lesson4. In MATLAB, navigate, as shown in the last lesson, so that this is your working folder.

4.2 Introduction to Function M-files

In the last lesson you learnt how to write script m-files. If you recall, a script m-file is simply a file that contains a list of MATLAB commands that are executed in the order that they appear when the name of the file is typed at the prompt in the command window. In this lesson you will learn about another type of m-file called a function m-file. Like a script m-file, a function m-file is simply a list of MATLAB commands. What is different about script and function m-files is the way they treat variables. Script m-files have full access to all of the variables in the workspace, and the commands that are executed can use these variables and modify them. Function m-files are called *functions* because they have inputs and outputs. In this sense they are just like the functions you work with in calculus. The only way a function m-file can access or modify variables in the workspace is through its inputs and outputs. If this doesn't make much sense yet, hang in there! All of these ideas will be explained further in the rest of the lesson.

Example 1: An m-file for a rational function

Let's start by creating a simple function m-file of the rational function

$$y = \frac{10x}{x^2 + 1}.$$

In the MATLAB command window click on **File** → **New** → **M-file** just as you would if you were going to create a script m-file. In the window that pops up type the following:

```
function y = myf(x)
y = 10*x/(x^2 + 1);
```

Save the file by clicking on **File** → **Save**. MATLAB suggests that you should save the file as myf.m. Go ahead and do that.

The first line of this file is called the function definition line. It is telling MATLAB that the file called myf (this is shorthand for 'my function') is a function m-file as opposed to a script m-file and that it has one input that will be referred to, in the file, as x and one output that will be referred to, in the file, as y. The code inside the file shows how the value of y is obtained from the value of x.

Before we see how to use this file, recall that when we type an expression at the prompt in the command window, we are telling MATLAB to evaluate that expression and assign the value to the variable `ans`. For example, if we type:

```
>> 5^2 - 2
ans =
    23
```

MATLAB evaluates the expression $5^2 - 2$ and assigns this value to the variable `ans`. Having recalled this, let's see how we can use our function m-file myf. Type:

```
>> myf(2)
ans =
    4
```

Notice that `myf(2)` is an expression that has a value. By typing it at the prompt we are telling MATLAB to find its value and assign that value to the variable `ans`. To find the value of `myf(2)` MATLAB looks for a file called myf. This file is a function m-file that has one input which is referred to in the file as x and one output which is referred to in the file as y. It sets the value of x to 2 and executes the code in the file. Whatever value y has when all of the code has been executed becomes the value of the expression `myf(2)`. In this case the value of y is

$$10(2)/(2^2 + 1) = 4$$

so this becomes the value of `ans`. As another example of how to use myf, type

```
>> temp = myf(3);
```

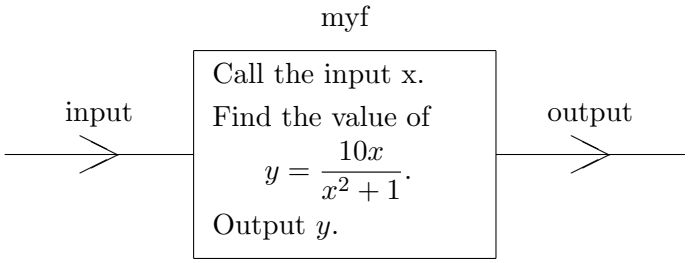
In this case we are telling MATLAB to find the value of `myf(3)` and assign that value to a variable called temp. To find the value of `myf(3)` MATLAB sets the variable x in the myf function file to 3 and executes the code in the file. The code in the file assigns the value $10(3)/(3^2+1) = 3$ to the variable y. Since y is the name of the output variable, this is the value of `myf(3)`. This value is assigned to the variable `temp`:

```
>> temp
temp =
    3
```

Notice that the file myf is acting just like the function defined by the formula:

$$\text{myf}(x) = \frac{10x}{x^2 + 1}.$$

A box picture of this function is:



The file myf plays the role of the box. It takes an input and creates an output from it. The commands in the file tell MATLAB how to create the output from the input.

The functions you study in beginning calculus have only one input and one output. However, more general functions can have more than one input and output. In the next example, the function m-file has one input and two outputs.

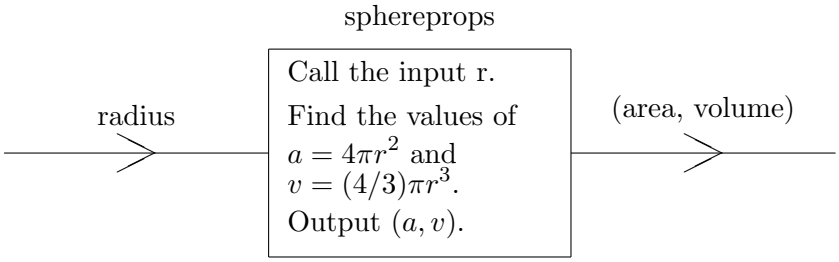
Example 2: Properties of a Sphere

In the command window pull down **File** → **New** → **M-file**. In the window that pops up type the following:

```
function [a, v] = sphereprops(r)

a = 4*pi*r^2;
v = (4/3)*pi*r^3;
```

Save the file by clicking on **File** → **Save**. MATLAB suggests that you should save the file as sphereprops.m. Go ahead and do that.

Notice that the function definition line in this file tells MATLAB that sphereprops is a function m-file that has one input that is referred to as r in the file and two outputs that are referred to as a and v respectively in the file. The code inside the file shows how the values of a and v are obtained from the value of r. We recognize from the formulae that a and v are the surface area and volume of a sphere that has radius r. A box picture of this function is:



Suppose we want to find the surface area and volume of a sphere of radius 2. We can use `sphereprops` to do this by typing at the prompt in the command window

```
>> [area, volume] = sphereprops(2);
```

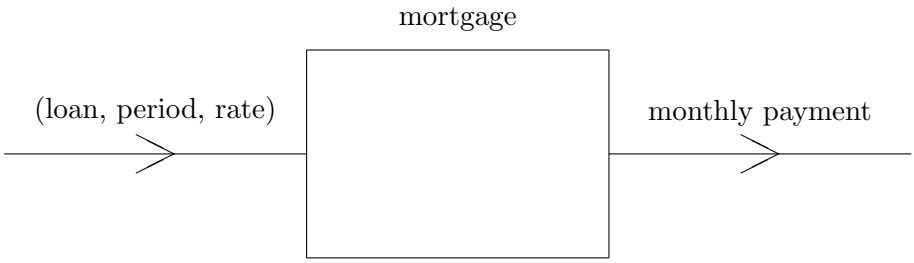
This command instructs MATLAB to find the value of the expression on the right and assign that value to the variables on the left. To find the value of the expression on the right, MATLAB looks for a file called sphereprops. It finds it, and sees that it is a function m-file that has one input which is referred to in the file as r and two outputs that are referred to in the file as a and v respectively. It sets the value of r to 2 and executes the code in the file. In this case, the code assigns a the value $4\pi(2^2) = 50.2655$ and assigns v the value $(4/3)\pi(2^3) = 33.5103$. Thus the value of `sphereprops(2)` is the pair of numbers $(50.2655, 33.5103)$. These numbers are assigned to the variables `area` and `volume` respectively. To check this type:

```
>> area
area =
    50.2655
>> volume
volume =
    33.5103
```

4.3 Function M-files as Black Boxes

The function myf in example 1 above was a generic rational function. We constructed it with no particular application in mind, so our focus was on the formula by which the output was obtained from the input. However, in example 2, the input and the outputs had meaning; the input was the radius of a sphere and the outputs were the corresponding volume and surface area. The person that wrote the m-file had to know the formulae for the surface area and volume of a sphere, but once the m-file is written it can be used by anybody without their having to know these formulae. This is why m-files are so important; once they are written you only have to remember the meanings of the input and output variables; you don't need to remember the details of how the output is calculated from the input.

To emphasize this, we have created an m-file that is a mortgage calculator; given the loan amount, the number of years over which the loan will be payed back, and the annual interest rate, the mortgage calculator calculates the monthly payments. Your instructor should have made the file available to you. Access the file and save it in your working MATLAB folder (probably the learnmatlab/Lesson4 folder on your flash drive). The file is called mortgage.p. The reason it has a .p extension instead of a .m extension is that it has been saved in a pre-compiled form so that you can't see the code inside it. This was done on purpose to emphasize that you don't need to see the code in order to use the file. All you need to know is what the inputs and outputs represent. A box picture of `mortgage` is shown below:



Notice that, in this case, we haven't filled in what's going on inside the box because we don't know. This won't prevent us from using the function though; to find out what your monthly payments would be if you borrowed $200,000, at an annual rate of 5% and paid it back over 30 years type

```
>> mortgage(200000, 30, 5)
ans =
    1.0727e+3
```

The monthly payments would be $1,073.

You have actually used many MATLAB functions already without realizing it. For instance when you type

```
>> sin(pi/6)
ans =
    0.5000
```

you are using a MATLAB function called sin. This function has one input that is interpreted as the radian measure of an angle and one output that is the sine of that angle. We don't know the details of how MATLAB calculates the output from the input, but we don't need to know them (provided we trust that MATLAB is doing it correctly). Some other MATLAB functions that you have used are `size`, `length`, `abs` etc. Thus, in essence, when you write your own function m-file, you are extending MATLAB's language to include another function. You can give your m-file to other people and they can use it without having to understand the details of how the output is obtained from the input.

4.4 Writing a Function M-File

As you saw in Examples 1 and 2 above, to create a function m-file you click on **File** → **New** → **M-file** in the command window just as you would if you were creating a script m-file. MATLAB distinguishes a function m-file from a script m-file by the very first line in the file which is called the function definition line. This line tells MATLAB that this file is a function m-file instead of a script m-file and it shows how many inputs and outputs the function has and what they are called inside the file. This line must have the form

`function [`*output1*`,` *output2*`, ...] =` *name*`(`*input1*`,` *input2*`, ...)`

where *output1*, *output2*, etc. are the names used inside the file for the output variables, *input1*, *input2* are the names used inside the file for the input variables,

and *name* is the name of the function. After this line you type the code that, when executed, will use the values of the input variables to find the values of the output variables.

Examples 1 and 2 illustrate how function m-files can be written. Here are two more examples.

Example 3: Number of roots of a quadratic function

In this example we will write a function m-file that calculates how many (real) roots the quadratic function $f(x) = ax^2 + bx + c$ has. Let's call the function `quadroots`. Click on **File** → **New** → **M-file** in the command window. The inputs to our function will be the coefficients a, b, and c and there will be one output whose value is the number of roots that the function $f(x) = ax^2 + bx + c$ has. Let's call the inputs a, b and c and let's call the output `numroots`. This means that the first line of the file should be:

```
function numroots = quadroots(a, b, c)
```

The code following this line must use the values of a, b and c to find the number of roots of the function. We know that the number of roots depends on the discriminant $\Delta = b^2 - 4ac$. If $\Delta > 0$ then the function has two roots, if $\Delta = 0$ then it has one root, and if $\Delta < 0$ then it has no roots. So, the first line of code finds the value of Δ:

```
delta = b^2 - 4*a*c;
```

We would like to know the sign of Δ. MATLAB has a function that will calculate this; it is called `sign`. You can get details of how to use this function by typing `help sign` at the prompt in the command window:

```
>> help sign
SIGN Signum function.
  For each element of X, SIGN(X) returns 1 if the element
  is greater than zero, 0 if it equals zero and -1 if it is
  less than zero.  For the nonzero elements of complex X,
  SIGN(X) = X ./ ABS(X).
  See also abs.
  Reference page in Help browser
    doc sign
```

Notice that if x is a number then `sign(x)` is 1 if $x > 0$, it's 0 if $x = 0$ and it's -1 if $x < 0$. Thus, to get the number of roots of the function we want to find `sign(delta)` and add one to it. Thus, the second (and final) line of our code is:

```
numroots = sign(delta) + 1;
```

The complete m-file should look like the following:

```
function numroots = quadroots(a, b, c)
delta = b^2 - 4*a*c;
numroots = sign(delta) + 1;
```

Save the file. To check the file, consider the quadratic function $f(x) = (x-1)(x+4) = x^2 + 3x - 4$, which has two roots, 1 and -4. To test our `quadroots` function, type, in the command window:

```
>> quadroots(1, 3, -4)
ans =
    2
```

We see that `quadroots` produced the correct answer. Now we'll try it on the function $f(x) = x^2 + 1$ that has no real roots. Type

```
>> quadroots(1, 0, 1)
ans =
    0
```

Again, we got the correct answer. As a last test, we'll try it on the function $f(x) = 3(x-1)^2 = 3x^2 - 6x + 3$ which has one real root. Type

```
>> quadroots(3, -6, 3)
ans =
    1
```

Again, it produced the correct answer.

Example 4: Finding the sum of the squares of the first n odd integers

In the next example we write a function m-file that finds the sum of the squares of the first n odd integers where n is an input to the file. Click on **File** → **New** → **M-file** in the command window. Let's call the function `sumofsqrs`. It will have one input that we'll call n and one output that we'll call s, so the first line should read:

```
function s = sumofsqrs(n)
```

Given, n, the output should be the value of

$$1^2 + 3^2 + 5^2 + \ldots + m^2$$

where m is chosen so that there are n terms in the sum. Notice that to get 1 term in the sum, m should be 1, to get 3 terms in the sum m should be 3, to get 3 terms in the sum m should be 5 etc. Summarizing this in a table we get

n	1	2	3	...
m	1	3	5	...

Observing the pattern (write in more terms if you need to) we see that $m = 2n - 1$. Having found m there are a number of ways that we could find the sum. One way would be to write a for loop as follows:

```
m = 2*n - 1;
s = 0;
for i = 1:2:m
s = s + i^2;
end
```

Another way would be:

```
n = 2*n - 1;
x = 1:2:n;
s = sum(x.^2);
```

Yet another way would be:

```
n = 2*n - 1;
x = 1:2:n;
s = x*x';
```

Using the last way (which is probably the quickest since MATLAB is written to work fastest on matrix operations) the complete file looks like:

```
function s = sumofsqrs(n)
n = 2*n - 1;
x = 1:2:n;
s = x*x';
```

Save the file. To test it, we'll use it to find, $1^2 + 3^2 + 5^2 + 7^2 = 84$. In the command window type:

```
>> sumofsqrs(4)
ans =
    84
```

It worked!

4.5 Variables in Function M-files

As we have seen, function m-files can be used without the user knowing or understanding the code that was used to write them. All the user needs to know is what the inputs and outputs represent. It is important, therefore, that the m-file doesn't change the values of variables in the workspace since the user might not even know that this is happening.

For example, consider the function `sumofsqrs` that we wrote above. When the function is called, the command `x = 1:2:n;` is executed. This command creates a variable x (if it doesn't already exist) and assigns it the value $(1, 3, 5, \ldots n)$. If you executed this command in the command window and you already had a variable called x in the workspace then its value would be lost and replaced by $(1, 3, 5, \ldots n)$. If the m-file worked the same way this could be highly problematic. In particular, suppose you were using the m-file without having looked at the code inside of it. Then you wouldn't even know that your value of x had been written over, and all future calculations that you did using x would be wrong!

The way MATLAB (and other programming languages) get around this problem is they give a function its own workspace. The only way a function can access the value of a variable in the main workspace is if that value is passed as an input to the function. The only way it can change the value of a variable in the main workspace is if one of its output variables is assigned to that variable. Any other variables it creates or uses as it executes the code in the file are in the function's own workspace. Thus, when the function `sumofsqrs` is called and the command `x = 1:2:n` is executed, it won't matter if there is a variable x in the main workspace or not. Only the variable x in the function's workspace will be modified and assigned the value $(1, 3, 5, \ldots n)$.

To check that you understand this, recall the function `sphereprops` that we created in Example 2. For easy reference here's what the m-file looks like:

```
function [a, v] = sphereprops(r)
a = 4*pi*r^2;
v = (4/3)*pi*r^3;
```

Type

```
>> clear
```

in the command window to clear the main workspace of all variables. Now, type the following line in the command window

```
>> [area, volume] = sphereprops(2);
```

There are now two variables in the main workspace `area` and `volume`. You can see this by typing

```
>> who
Your variables are:
area    volume
```

Their values are the area and volume of a sphere of radius 2 respectively:

```
>> area
area =
    50.2655
>> volume
volume =
    33.5103
```

Notice that the variables a and v that appear in the m-file are not listed as variables in the main workspace. This is because these variables are *local to the function*. To illustrate with another example let's clear the workspace again:

```
>> clear
```

Now type the following lines of code:

```
>> a = 5;
>> v = 7;
>> [a, volume] = sphereprops(2);
>> a
a =
    50.2655
>> volume
volume =
    33.5103
>> v =
v = 7
```

Notice that in the first two lines MATLAB creates variables a and v in the main workspace and assigns them the values 5 and 7 respectively. Then, the function `sphereprops` is called which has two outputs; the surface area of a sphere of radius 2 and the volume of a sphere of radius 2. The first is assigned to the variable a in the workspace overwriting its previous value and the second is assigned to a variable `volume` that is created in the workspace. Notice that the variable v in the

function `sphereprops` is *local to the function* and does not change the value of v in the workspace. As a final example let's return to the function `sumofsqrs`. First, clear the workspace:

```
>> clear
```

Now type in the following code.

```
>> n = 4;
>> sumofsqrs(n);
ans =
    84
```

Notice that when sumofsqrs is called, the variable n in the function's workspace is set equal to the value of n in the main workspace, namely 4. In the first line of code in the m-file, the value of the local variable n is changed to $2 * 4 - 1 = 7$. However, this only changed the variable n that was *local to the function*; the variable n in the workspace remained unchanged. In particular, if we ask MATLAB what the value of n is:

```
>> n
n =
    4
```

we see that it still has the value 4.

4.6 Comments and Help Lines in a Function M-file

As with script m-files it is always a good idea to fully comment your function m-files explaining what each piece of code does. (This really isn't necessary in our four examples because the files are so short, but you should try to make a habit of it anyway.) In addition, function m-files can and should be commented with what are known as help lines. These are comment lines that appear directly after the function definition line and before the first line of code. In these lines you should describe what the function does and what its inputs and outputs are. When the user types, at the prompt in the command window,

```
>> help functionname
```

where *functionname* is the name of the function, these lines will appear, showing the user what the function does and how it should be used. For example, consider the function `sphereprops` in Example 2. Click on the function in the edit window and add the comment lines shown below.

```
function [a, v] = sphereprops(r)
% The function [a, v] = sphereprops(r) calculates the surface area
% and volume of a sphere from its radius.
% Inputs:
% r (number):  The radius of the sphere.
% Outputs:
% a (number):  The surface area of the sphere.
% v (number):  The volume of the sphere.
```

```
a = 4*pi*r^2;
v = (4/3)*pi*r^3;
```

Save the file. Now, in the command window at the prompt type

```
>> help sphereprops
The function [a, v] = sphereprops(r) calculates the surface area
and volume of a sphere from its radius.
Inputs:
r (number):  The radius of the sphere.
Outputs:
a (number):  The surface area of the sphere.
v (number):  The volume of the sphere.
```

This tells the user exactly how to use the file without them having to understand or even see the code that was used to write it. As another example, here are good help lines to include in the m-file `quadroots` of Example 3.

```
function numroots = quadroots(a, b, c)
%
% The function numroots = quadroots(a, b, c) finds how many real
% roots the quadratic function f(x) = ax^2 + bx + c has.
% Inputs:
% a (number):  the coefficient of x^2 in the quadratic
% b (number):  the coefficient of x in the quadratic
% c (number):  the constant term in the quadratic
% Output:
% numroots (number):  the number of real roots (either 0, 1, or 2)

delta = b^2 - 4*a*c;
numroots = sign(delta) + 1;
```

4.7 Vectorizing Your Code

Recall that to sketch the graph of, say, $\sin(x)$ in MATLAB, we first create a vector of x-values, then create the corresponding vector of y-values, and then plot the two vectors against each other:

```
>> x = -10:.1:10;
>> y = sin(x);
>> plot(x, y)
```

Notice what happened when we typed `y = sin(x)`. The vector $x = (-10, -9.9, \ldots 10)$ was fed into MATLAB's `sin` function. The way the function worked was that it found the sine of each entry in x. In other words `sin(x)` was the vector

$$(\sin(-10), \sin(-9.9), \ldots \sin(10)).$$

Suppose now that we wanted to sketch the graph of the function `myf` in Example 1. If we did the same thing as above MATLAB would complain:

```
x = -10:.1:10;
```

```
>> y = myf(x);
???  Error using ==> mpower
Matrix must be square.

Error in ==> myf at 3
y = 10*x/(x^2 + 1);
```

MATLAB didn't know how to work out what x^2 was because x was a vector, so it couldn't be multiplied by itself with regular matrix multiplication. What we wanted y to be was the vector of values

$$\left(\frac{-100}{10^2+1}, \frac{-99}{9.9^2+1}, \frac{-98}{9.8^2+1}, \ldots, \frac{-100}{10^2+1}\right).$$

In other words, we wanted MATLAB to do pointwise operations `./` and `.^` instead of the regular matrix operations `/` and `^`. Go back to the myf m-file and replace the matrix operations with point-wise operations:

```
function y = myf(x)
y = 10*x./(x.^2 + 1);
```

Save the file. Now, let's try again to sketch the graph of the function:

```
x = -10:.1:10;
>> y = myf(x);
>> plot(x, y)
```

It works!

Since variables in MATLAB are so often vectors and matrices, you should always think about how you would like your function to behave should the inputs be vectors or matrices and you should write the code in such a way that it behaves like that. As another example, consider the function sphereprops that we constructed in Example 2. If the input $r = (r_1, r_2, \ldots r_n)$ were a vector then probably a good way for the function to behave would be for it to find the area and volume of each of the spheres of radius $(r_1, r_2, \ldots r_n)$. Thus, it would be good if both a and v in the file were vectors. We can achieve this easily by replacing each operation in the file with its corresponding dot-operation. Notice that we also changed the help lines to reflect this.

```
function [a, v] = sphereprops(r)
% The function [a, v] = sphereprops(r) calculates the surface area
% and volume of a sphere from its radius.
% Inputs:
% r (vector):  The radii of spheres.
% Outputs:
% a (vector):  The surface areas of the spheres with radii given
% in r.
% v (vector):  The volumes of the spheres with radii given in r.

a = 4*pi*r.^2;
v = (4/3)*pi*r.^3;
```

4.8 Calling one M-file From Another M-file

One m-file can use another m-file. As an example, we'll create a function m-file that plots the graph of the myf function. We'll call the function plotmyf. Its inputs will be the least and greatest values of x used in the plot as well as the step size between adjacent points. It will have no output. Click on **File** → **New** → **M-file** in the command window and type the following in the edit window that opens up:

```
function plotmyf(xmin, xmax, step)
x = xmin:step:xmax;
y = myf(x);
plot(x, y)
```

Save the file as plotmyf.m. Notice that the code in plotmyf.m calls the function myf. To use plotmyf to plot the function myf, type

```
>> plotmyf(-10, 10, .1)
```

at the prompt in the command window. It worked.

4.9 Testing and Debugging Your Code

Remember when you write m-files that you should write one little piece at a time, save the file and run the file to check that it's doing what you expected. Then, build on it and write another little piece, save and test it again. If at any point you are getting something different from what you expected, think about what is happening at each stage. To see what's happening try removing the semicolons from the ends of the lines so that MATLAB will print what it's doing to the command window. This can really help you decipher what's going on. Once you've worked out what's going on you can put the semicolons back in again, since you don't normally want to see the guts of an m-file.

Worksheet 4

1. *(Using function m-files when you don't know the code inside them.)*

   a) i) Using the function `mortgage.p` that you accessed when working through Lesson 4, plot a graph of monthly payment as a function of the amount borrowed when the mortgage is paid over 30 years with an annual interest rate of 5%. Sketch your graph by hand here.

   ii) From your graph determine a formula for the monthly payment as a function of the amount borrowed under these conditions.

   b) The concentration of drug in a patient's bloodstream depends on when the drug was administered, how much was administered, and on the patient's weight. Your instructor should have made the file `happacine.p` available to you. Save the file in your working MATLAB folder (probably learnmatlab/Lesson4 on your flash drive). This function determines the concentration of the drug Happacine in the blood of a patient. The function has three inputs. The first input is the weight of the patient in pounds. The second input is a vector of length two that indicates when and how much drug was administered to the patient. The first entry of this vector is the size of the doses (in mg) and the second entry is the number of hours between doses. The third input is the number of hours that have passed since the first dose was administered. The output of the function is the concentration of the drug in the bloodstream of the patient at this point in time, measured in mg/liter. For example, if you type

```
>> happacine(125, [5, 8], 60)
ans =
    0.7588
```

   then you see that when a person who weighs 125 lb is administered 5 mg of the drug every 8 hours, the concentration of the drug in the bloodstream after 60 hours (or 2.5 days) is 0.7588 mg/liter. The third input may also be a vector. In this case, the function will give the concentrations in the bloodstream at each of the times listed in the vector. For example, if you type

```
>> happacine(125, [5, 8], [1, 9, 17])
ans =
    0.1446     0.2732     0.3873
```

   then you see that when a person who weighs 125 lb is administered 5 mg of the drug every 8 hours, the concentration of the drug in the bloodstream after 1 hour is 0.1446 mg/liter, after 9 hours is 0.2732 mg/liter and after 17 hours is 0.3873 mg/liter.

Use MATLAB to draw a graph of the concentration of drug in the bloodstream as a function of time when a person who weighs 125 lb is administered 5 mg of the drug every 8 hours. Describe the graph and explain why it looks the way it does. In particular, describe

i) the overall general shape of the graph,
ii) why and when the graph has points of discontinuity,
iii) what happens to the concentration in the long run, and
iv) how long it takes for the concentration to reach a 'steady state.'

2. *(Writing your own function m-files.)*

a) Write a function m-file for the function $y = x^2 + 3$. The input should be a number, x, and the output should be the corresponding number y. Name the function `myquad`. Test that your m-file works by typing

```
>> myquad(5)
```

at the command line. MATLAB should respond with

```
ans =
    28
```

Make sure that your m-file works when the input is a vector. Test that it works by typing

```
>> myquad(1:4)
```

at the command line. MATLAB should respond with

```
ans =
    4     7     12     19
```

b) Write a function m-file whose inputs are the length, width, and height of a rectangular solid block, and whose outputs are the volume and surface area of the block. Name the function `solidprops`. Test that your m-file works by typing

```
>> [volume, surfacearea] = solidprops(2, 5, 6)
```

at the command line. MATLAB should respond with

```
volume =
    60
surfacearea =
    104
```

c) Recall the script m-file you wrote for Worksheet 3 to estimate the value of

$$\int_0^1 e^{-x^2}\,dx$$

using the trapezoidal rule with $n = 20$. Modify this m-file so that it is a function m-file that has one input and one output. The input should be

the number of intervals used in the trapezoidal approximation and the output should be the estimated value of the integral.

d) Write a function m-file that calculates the value of

$$(1)(2) + (2)(3) + (3)(4) + \ldots (n-1)(n)$$

where n is an input to the function. Name the function `sumprod`. Test your function by noticing that `sumprod(2)` should be equal to $(1)(2) = 2$ and `sumprod(4)` should be equal to $(1)(2) + (2)(3) + (3)(4) = 20$.

3. *(Variables in function m-files.)* Try to answer each of the following questions first without using MATLAB. Then, check your answers using MATLAB.

a) Suppose you have the following two function m-files saved in your working directory.

```
function y = myf1(x)
x = x^2;
y = 5*x + 3;
```

Suppose, moreover, that you type the following commands at the prompt in the command window:

```
>> clear, x = 6; z = myf1(x);
```

In each case, determine which response MATLAB will give when the variable shown is typed at the prompt in the command window. (Blank lines have been ignored in the responses.)

i) `>> z`
   I) `z = 33`
   II) `z = 183`
   III) `??? Undefined function or variable 'z'.`

ii) `>> x`
   I) `x = 6`
   II) `x = 36`
   III) `??? Undefined function or variable 'x'.`

iii) `>> y`
   I) `y = 33`
   II) `y = 183`
   III) `??? Undefined function or variable 'y'.`

b) Suppose you have the following function m-file saved in your working directory.

```
function y = myf2(x)
```

```
x = x^2
y = 5*x + 3
```

Notice that `myf2` is almost the same as `myf1`. The only difference is that the semicolons after the commands are left out in `myf2`. Suppose, as above, that you type the following commands at the prompt in the command window:

```
>> clear, x = 6; z = myf2(x);
```

Notice that in this case, MATLAB responds with

```
x =
    36
y =
    183
```

In each case below, determine which response MATLAB will give when the variable shown is typed at the prompt in the command window. (Blank lines have been ignored in the responses.)

i) `>> z`
   I) `z = 33`
   II) `z = 183`
   III) `??? Undefined function or variable 'z'.`

ii) `>> x`
   I) `x = 6`
   II) `x = 36`
   III) `??? Undefined function or variable 'x'.`

iii) `>> y`
   I) `y = 33`
   II) `y = 183`
   III) `??? Undefined function or variable 'y'.`

5 Boolean Expressions and While Loops

5.1 Goals of this lesson:

In this lesson you will learn

- what a Boolean expression is,
- why we want MATLAB to be able to determine if a Boolean expression is true or false,
- what relational operators are,
- the difference between `==` and `=`,
- what the Boolean operators & and | do,
- what a while loop is and when to use one, and
- techniques for writing a while loop.

In this lesson you will be creating files that you will need to save somewhere. So, insert your flash drive into the USB port and inside the learnmatlab folder create a folder called Lesson5. In MATLAB, navigate so that this is your working folder.

5.2 Introduction to Boolean Expressions

A statement that may be true or false is called a *Boolean expression*. For instance, the statements, "$5 > 3$" and "An apple is not a fruit," are Boolean expressions. The first statement is true and the second is false. (These examples taken from Wikipedia.) Boolean expressions can contain a variable. In this case they may be true for some values of the variable and false for other values. For instance, the statement "$x > 3$" is true when x is equal to 5 but is false when x is equal to 1.

Why do we want MATLAB to determine whether a statement is true or false? This is because often what we do next depends on whether a statement is true or false. For instance, suppose we want to evaluate

$$\lim_{x \to 0^+} \frac{2^x - 1}{\sin x}.$$

correct to 5 decimal places. As you know, we can do this numerically by creating a table of values using values of x that get closer and closer to 0 as shown below:

x	$(2^x - 1)/\sin x$
1	1.188395
.1	0.718932
.01	0.695567
.001	0.693388
.0001	0.693171
⋮	⋮

If we want to know the limit to 2 decimal places, we look at the table above and see that the last two entries haven't changed when rounded to 2 decimal places, so we conclude that the value of the limit is 0.69 when rounded to 2 decimal places. If we want to know the limit to 5 decimal places, then we haven't gone far enough; we have to fill in the table with more values of x.

It is easy enough to get MATLAB to evaluate the expression at progressively smaller values of x, but how can MATLAB know when to stop without having a human intervene and say, "stop now?" For MATLAB to know when to stop it needs to be able to ascertain at each stage whether or not it is true that the values have stopped changing in the fifth decimal place. For this reason, it is important that MATLAB be able to determine if a statement is true or false.

After we have discussed and understood Boolean expressions a little better we will return to this example to see explicitly how we could implement it in MATLAB.

5.3 Boolean Expressions and Relational Operators

Boolean expressions in MATLAB are characterized by the six relational operators:

`==`	equal
`~=`	not equal
`<`	less than
`>`	greater than
`<=`	less than or equal
`>=`	greater than or equal

For instance, the expression `3 > 5` is a Boolean expression that is false. When a Boolean expression is typed at the prompt in the command window, MATLAB *evaluates* the expression. In other words, it determines whether the expression is true or false. If it is true then the value of the expression is 1 and if it is false then the value is 0. For instance type the following at the prompt in the command window:

```
>> 3 > 5
ans =
    0
```

When a Boolean expression contains the value of a variable, MATLAB will use the current value of the variable to determine whether the expression is true or false. For instance type the following at the prompt in the command window.

```
>> x = 5;
>> x < 7
ans =
    1
```

Since x was assigned the value 5 in the first command, it was true that x was less than 7, so the Boolean expression `x < 7` had the value 1 (True). Now type the following to check your understanding of what each relational operator means:

```
>> x == 3
ans =
    0
>> x ~= 3
ans =
    1
>> x^2 - 16 < 9
ans =
    0
>> x^2 - 16 <= 9
ans =
    1
>> 2*x + 6 > 3*x
ans =
    1
>> 2*x + 6 >= 3*x
ans =
    1
```

5.4 The Difference Between = and ==

Recall that a single equals sign = is an *assignment* command. For instance, when you type

```
>> x = 7;
```

at the prompt in the command window, the expression on the right is evaluated (its value is 7) and its value is assigned to the variable on the left. In other words, the variable x is assigned the value 7. In contrast, a double equals sign == is a *relational operator*. For instance, when you type

```
>> x == 5;
```

at the prompt in the command window, you are asking MATLAB to find the value of this Boolean expression (and assign the value to the variable `ans`). In other words you are asking MATLAB to determine if the statement is true or false. In this case, the right hand side has value 5 while the left hand side is x which has the value 7, so the statement is not true. Thus, when MATLAB evaluates the expression it gets 0 (false) and this value is assigned to the variable `ans`:

```
>> ans
ans =
    0
```

5.5 The Boolean Operators & and |

Sometimes we want to know if two things are both true or if at least one of them is true. We can tell MATLAB to ascertain this by combining the two expressions with

either an ampersand sign & (signifying and) or a vertical line | (signifying or). In particular, if A and B are both Boolean expressions then $A\&B$ is another Boolean expression that is true if both A and B are true and otherwise is false. On the other hand, $A \mid B$ is a Boolean expression that is true if either one or both of A and B are true and is false only in the case when both A and B are false. Here is a table that summarizes this:

A	B	$A\&B$	$A \mid B$
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F

For instance, type the following at the prompt in the command window:

```
>> x = 3; y = 5;
>> x + y > 0
ans =
    1
>> x - y > 0
ans =
    0
```

Since $x = 3$ and $y = 5$, notice that $x + y = 8$ which is greater than 0, so the first Boolean expression is true and evaluated to 1. On the other hand, $x - y = -2$ which is not greater than 0 so the second Boolean expression is false and evaluated to 0. Now type:

```
>> (x + y > 0) & (x - y > 0)
ans =
    0
```

This Boolean expression is false and evaluated to 0 because it is not true that both $x + y > 0$ and $x - y > 0$. On the other hand, if we type:

```
>> (x + y > 0) | (x - y > 0)
ans =
    1
```

then this Boolean expression is true since it is true that $x + y > 0$, so the expression evaluated to 1.

5.6 While Loops

Computers are particularly well-suited to perform repetitive tasks. We have seen that one way to get MATLAB to perform a repetitive task is to use a for loop. Recall that the structure of a for loop is:

```
for variable = expression
statements
end
```

where *expression* is a vector. When MATLAB executes the loop, it sets the variable equal to each element in the vector successively, and executes the commands in *statements* each time. Thus, the number of times the commands in *statements* are executed is known beforehand and is equal to the length of the vector. For this reason a for loop is only useful if you know beforehand how many times the task is to be repeated. However, many times you don't know this; you only know when to stop when you see the results of the calculations that are being performed. In a situation like this you can use a while loop. A while loop in MATLAB has the form:

`while` *expression*
statements
`end`

where *expression* is a Boolean expression that is either true or false. Typically *expression* has a variable in it and it is true for some values of the variable but not for others. When MATLAB comes upon the `while` statement, it determines whether *expression* is true or false. If it is false, it skips the loop altogether and continues with any commands that appear after `end` or it stops if there are no such commands. On the other hand, if *expression* is true, then it executes the code in *statements*. After executing that code, it returns to *expression* to determine if it is still true. If it is not true, then the loop is finished and MATLAB continues with any commands that appear after `end` or stops if there are no such commands. On the other hand, if *expression* is still true, then it executes the code in *statements* for a second time and then returns again to *expression* to determine if it is still true. It continues like this until finally *expression* is false at which point the loop has been completed.

The examples below illustrate how while loops work.

Example 1: Find the sum of the reciprocals of the first 99 integers.

Click on **File** → **New** → **M-file** to create a new m-file. Type the following code in the m-file.

```
i = 1;
s = 0;
while i < 100
s = s + 1/i;
i = i + 1;
end
s
```

Save the file as `ex1.m`. Notice that `ex1.m` is a script m-file. To understand what the commands do when `ex1` is typed at the prompt in the command window, let's go through them step by step. In the first two lines of code, two variables i and s are created and they are assigned the values 1 and 0 respectively. When MATLAB comes upon the while statement, i has the value 1 so the expression $i < 100$ is true. So, the statements in the body of the while loop are executed. The first statement in the body of the loop is an assignment statement. The value of the expression on the right hand side is $s + 1/i = 0 + 1/1 = 1$ and this is assigned to the variable s.

So s now has the value 1. The second statement is also an assignment statement. The value of the expression on the right hand side is $i + 1 = 1 + 1 = 2$ and this is assigned to the variable i. So i now has the value 2. MATLAB then returns to the while statement. At this point i has the value 2 so the expresion $i < 100$ is still true so the statements in the body of the while loop are executed again. In the first statement s is assigned the value $1 + 1/2$ and in the second statement i is assigned the value $2 + 1 = 3$. MATLAB then returns to the while statement. This time $i = 3$ so the expression $i < 100$ is true again. So, the statements in the body of the loop are executed once again resulting in s being assigned the value $(1 + 1/2) + 1/3$ and i being assigned the value $3 + 1 = 4$. The loop continues in this way until i is assigned the value 99. When $i = 99$ the expression $i < 100$ is still true so the statements in the body of the loop are executed resulting in s being assigned the value $(1 + 1/2 + \ldots 1/98) + 1/99$ and i being assigned the value $99 + 1 = 100$. This time when MATLAB returns to the while statement the expression $i < 100$ is false, so MATLAB skips to the statement that appears after `end` and prints out the value of s. This procedure is summarized in the table below.

$i < 100$	s	i
	0	1
True	$0 + 1/1 = 1$	$1 + 1 = 2$
True	$1 + 1/2$	$2 + 1 = 3$
True	$(1 + 1/2) + 1/3$	$3 + 1 = 4$
True	$(1 + 1/2 + 1/3) + 1/4$	$4 + 1 = 5$
$\vdots$	$\vdots$	$\vdots$
True	$(1 + 1/2 + 1/3 + \ldots 1/98) + 1/99$	$99 + 1 = 100$
False		

Thus, the value of s that is printed out in the command window is the sum of the reciprocals of the integers from 1 to 99:

$$s = 1 + 1/2 + 1/3 + \ldots 1/99.$$

Type

```
>> ex1
s =
    5.1774
```

in the command window to execute the code in `ex1`. We see that

$$1 + 1/2 + 1/3 + \ldots 1/99 = 5.1774.$$

Notice that another way to find this sum would be to use a for loop as in the code below:

```
>> s = 0;
>> for i = 1:99, s = s + 1/i; end
```

```
>> s
s =
    5.1774
```

Yet another way wouldn't use a loop at all as in the code below:

```
>> s = sum(1./(1:99))
s =
    5.1774
```

Example 2: How many times do you have to roll a die before you roll a 6?

How many times do you have to roll a die before you roll a 6? Well, it varies. You might roll a 6 the first time, or you might have to roll the die 50 times (or more) before you roll a 6! In this example we'll use a while loop to have MATLAB simulate the process of rolling a die until a 6 is obtained and count the number of times the die is rolled.

The code uses two MATLAB functions that we haven't seen yet, so let's explore these first. The function `rand` can be used to generate a random number chosen from the interval $[0, 1]$. Type

```
>> help rand
```

at the prompt to see how this function works. To explore it, type

```
>> rand(1)
ans =
    0.4565
>> rand(1)
ans =
    0.0185
>> rand(1)
ans =
    0.8214
```

Notice that every time you type it you get something different. This is because the number you get is random. The other function we'll use is `ceil`.

```
>> help ceil
```

at the prompt in the command window to see how this function works. This function rounds its input to the next highest integer. To test it out, type the following at the prompt in the command window:

```
>> ceil(3.42)
ans =
    4
>> ceil(3)
ans =
    3
>> ceil (-2.65)
ans =
    -2
```

Now let's create the m-file that simulates throwing a die until a 6 is obtained and counting how many times it is thrown. Click on **File** → **New** → **M-file** to create a new m-file. Type the following in the m-file.

```
d = 0;
count = 0;
while d ~= 6
d = ceil(6*rand(1))
count = count + 1;
end
count
```

Save the file as `die.m`. To understand how the code works when `>> die` is typed at the prompt in the command window, let's run through it step by step. Initially variables d and `count` are created whose values are both 0. When MATLAB comes upon the while statement, it evaluates the Boolean expression `d ~= 6` and finds that it is true since $d = 0 \neq 6$. So it executes the statements in the body of the loop. The first statement is an assignment statement. To find the value on the right it generates a random number between 0 and 1 and multiplies it by 6. This gives a random number between 0 and 6. It then rounds this number up to the nearest integer obtaining either 1, 2, 3, 4, 5, or 6. Each integer is as likely as any other integer, so this has simulated a roll of the die. The value obtained is assigned to d and the value of d is printed out to the command window. Thus, the value of d is what is obtained on the first roll of the die. The next statement is also an assignment statement. The value on the left is $0 + 1 = 1$ and this is assigned to the variable `count`. MATLAB then returns to the while statement. If a 6 was rolled then the Boolean expression `d ~= 6` is false and MATLAB will go the statement after end and print out the value of `count` which is 1. Otherwise, if a 6 was not rolled, then the Boolean expression is true and the statements in the body of the loop will be executed once again. The first statement simulates rolling the die again and assigns the value rolled to d and prints this out to the command window. The second statement assigns the value $1 + 1 = 2$ to `count`. MATLAB then returns to the while statement. If a 6 was just rolled then the Boolean expression is false and MATLAB goes to the statement after end and prints out the value of `count` which is 2. Otherwise it executes the statements again. This continues until finally a 6 is rolled. Notice that when MATLAB finally stops, the value of d will be 6 and the value of `count` that is printed out to the command window is the number of times the die was rolled to get a 6. Type `>> die` a few times at the prompt in the command window in order to see the program work.

```
>> die
d =
    3
d =
    4
d =
    5
```

```
d =
    6
count =
    4
>> die
d =
    3
d =
    6
count =
    2
```

Notice that every time you run the program it's different. This is because it's random! If you were to do this experiment by actually rolling a die it would be different every time too. Let's go back and edit the file to find the distribution of the number of times we need to roll a die in order to obtain a 6. To do this we want to run the program many times and keep track of the value of `count` each time. Let's run it fifty thousand times. Then we'll draw a histogram of the values obtained. Edit your file so it looks like the file below. Notice that we've embedded the while loop inside a for loop that performs the while loop fifty thousand times. Each time it is performed we store the value of `count`, so `count` is now a vector of length fifty thousand. Notice also that we've put a colon where d is assigned its new value and we've removed the statement that prints the value of `count` to the command window; we don't want to see tens of thousands of values being printed out to the command window!

```
count = zeros(1, 50000);
for i = 1:(50000)
d = 0;
while d ~= 6
d = ceil(6*rand(1));
count(i) = count(i) + 1;
end
end
```

Save the file and type

```
>> die
```

at the prompt in the command window to run it. It will take a moment; you are asking MATLAB to do a lot of calculations! When the prompt appears again you will know that it's done. Now, to draw a histogram of the number of rolls needed type

```
>> bar(histc(count, .5:70.5))
```

Notice that the most commonly occurring number of times that you have to roll the die is 1 and that this decreases as time goes on. Type

```
>> die
>> figure
>> bar(histc(count, .5:70.5))
```

again to get another histogram based on 50000 rolls of the die. Notice that the two histograms look very similar. Although every time you repeat the experiment it will look different, when you roll the die 50000 times the histogram of the number of rolls needed to get a 6 looks very similar.

5.7 Techniques for Writing Your Own While Loops

When you want MATLAB to repeat a task over and over again, but you don't know a-priori how many times it will need to repeat it, it is probably appropriate to use a while loop. When you write the while loop you need to think about four things.

1. You need to determine what exactly is being repeated. The commands for this will go in the body of the loop.
2. Under what conditions should the loop stop. This will be expressed using a Boolean expression that goes in the while statement.
3. What needs to be initialized before entering the loop?
4. What exactly do you want the value of everything to be when you exit the loop.

To illustrate the process of writing a while loop let's consider the example in the first section of this lesson in which we want MATLAB to evaluate

$$\lim_{x \to 0^+} \frac{2^x - 1}{\sin x}$$

correct to 5 decimal places. We'll write an m-file script to do this.

We know that to do this numerically we can plug in values of x that get closer and closer to 0 and evaluate the function at each value. This is a repetitive action, but we don't know a-priori how many times we will need to repeat it. This indicates that a while loop is an appropriate way to code this. We want to stop when the present value and the previous value are the same to 5 decimal places; in other words, when the absolute value of the difference between the present value and the previous value is less than 10^{-6}. So, our m-file script should consist of commands that tells MATLAB to do the following:

1. Choose an initial value of x (we used 1 in the table above).
2. Evaluate the function at this value.
3. Choose another value of x that's closer to 0 (we used .1 in the table above).
4. Evaluate the function at this new value of x.
5. Find the absolute value of the difference between the two values of the function. If it is less than 10^{-6} then use this new value as the value of the limit and stop.

6. Otherwise, if the difference is greater than 10^{-6}, choose another value of x that's closer to 0 (we used .01 in the table above).

7. Evaluate the function at this new value of x.

8. Find the absolute value of the difference between this new value of the function and the previous value. If the difference is less than 10^{-6} then use this new value as the value of the limit and stop.

9. Otherwise, if the difference is greater than 10^{-6}, choose another value of x that's closer to 0 (we usesd .001 in the table above).

10. Evaluate the function at this new value of x.

    etc.

Notice that steps 5, 6, and 7 are the same as steps 8, 9 and 10 which are the same as steps 11, 12 and 13 etc. So, these are the steps that will be inside the while loop. The criterion for continuing with the loop is that the difference between the new function value and the old function value be greater than 10^{-6}. When the loop is finished, the value of the limit should be set equal to the newest value of the function. Thus, our m-file script should be a translation of the following commands into MATLAB code:

1. Choose an initial value of x (we used 1 in the table above).

2. Evaluate the function at this value.

3. Choose another value of x that's closer to 0 (we used .1 in the table above).

4. Evaluate the function at this new value of x.

5. While the absolute value of the difference between the new value and the old value is greater than 10^{-6}

   5a Choose a new value of x (we used one tenth of the previous value in the table above).

   5b Evaluate the function at this new value.

6. The limit is equal to the newest value of the function.

Now, to translate this into code is relatively simple. The only thing we have to worry about is what the 'new value of x' and the 'old value of x' are because these keep changing; what was the new value becomes the old value as soon as a yet newer value of x is chosen. So, we'll have to insert commands telling MATLAB to change them exactly like that. We'll do a similar thing for the 'new' and 'old' values of the function. So, now we're ready to create the m-file. Click on **File** → **New** → **M-file** to create a new m-file. Type in the following code. The comments are there for

you to see that the code is really a direct translation of the steps listed above into MATLAB .

```
xold = 1; % Step 1
yold = (2^xold - 1)/sin(xold); % Step 2
xnew = 0.1*xold; % Step 3
ynew = (2^xnew - 1)/sin(xnew); % Step 4
while abs(ynew - yold) > 10^(-6) % Step 5
      % In the next two commands the new values of x and y
      % become the old values respectively.
      xold = xnew;
      yold = ynew;
      xnew = 0.1*xold; % Step 5a
      ynew = (2^xnew - 1)/sin(xnew); % Step 5b
end % end of Step 5
lim = ynew; % Step 6
```

Save the file as `limit1.m`. Test it by typing

```
>> limit1
```

at the prompt in the command window. It looks like MATLAB didn't do anything. This is because we put semi-colons after each line in the code. To see what the value of the limit is, type

```
>> lim
lim =
   0.69314720407832
```

at the prompt in the command window. Well, we got a number for the limit, 0.69315, but how do we know it's right? To check that MATLAB is doing what you expect it to be doing, you can go back to the m-file and remove some appropriately chosen semi-colons. That way you can see those calculations as they are performed to check that MATLAB is doing the calculations you expect. So, let's go back to the m-file and remove all those semi-colons where a new value of x or a new value of y is calculated. The m-file (without the comments) should now look like:

```
xold = 1
yold = (2^xold - 1)/sin(xold)
xnew = 0.1*xold
ynew = (2^xnew - 1)/sin(xnew)
while abs(ynew - yold) > 10^(-6)
      xold = xnew;
      yold = ynew;
      xnew = 0.1*xold
      ynew = (2^xnew - 1)/sin(xnew)
end
lim = ynew
```

Save the file and then type `>> limit1` again at the prompt in the command window. Here is what comes out (in a more compressed form).

```
>> limit1
```

```
xold = 1
yold = 1.18839510577812
xnew = 0.10000000000000
ynew = 0.71893224680670
xnew = 0.01000000000000
ynew = 0.69556659839056
xnew = 1.000000000000000e-03
ynew = 0.69338757814533
xnew = 1.000000000000000e-04
ynew = 0.69317120492028
xnew = 1.000000000000000e-05
ynew = 0.69314958283152
xnew = 1.000000000000000e-06
ynew = 0.69314742079396
xnew = 1.000000000000000e-07
ynew = 0.69314720407832
lim = 0.69314720407832
```

Sure enough, it looks like the values of x are decreasing as expected: 1, 0.1, 0.01, 0.001, ... Similarly the y-values look good. In particular, the first few of them match the values in the table given at the beginning of the lesson. Also, the value of `lim` was correctly assigned to the last value of the function.

Let's look at another way to code the same process. When we constructed the code above in `limit1.m` we did it by noticing that steps 5, 6 and 7 were the same as steps 8, 9 and 10 etc. However, looking at it from a different point of view we see that, in essence, steps 3, 4 and 5 are the same as 6, 7 and 8 which are the same as 9, 10 and 11 etc. In this case we would have the following pseudo-code:

1. Choose an initial value of x (we'll use 1).
2. Evaluate the function at this value.
3. While the absolute value of the new value and the old value is greater than 10^{-6}
   - 3a Choose a new value of x (we used one tenth of the previous value in the table above).
   - 3b Evaluate the function at this new value.
4. The limit is equal to the newest value of the function.

The only problem with this pseudocode is that when MATLAB comes upon step 3 the first time, it won't be able to test whether or not the absolute value of the difference between the new value and the old value is greater than 10^{-6} because there isn't a new value and an old value; the function has only been evaluated once. This will produce an error message. So, what you need to do is create a variable whose value will be this difference, and initialize the variable before step 3 to any

number that is bigger than 10^{-6} so that the loop will definitely be executed at least once. Here is code that works. Click on **File** → **New** → **M-file** to create a new m-file and type in the code.

```
xold = 1; % Step 1
yold = (2^xold - 1)/sin(xold); % Step 2
diff = 1;
while diff > 10^(-6) % Step 3
      xnew = 0.1*xold; % Step 3a
      ynew = (2^xnew - 1)/sin(xnew); % Step 3b
      diff = abs(xnew - xold);
      % In the next two commands the new values of x and y
      % become the old values respectively.
      xold = xnew;
      yold = ynew;
end % end of Step 3
lim = yold; % Step 4
```

Save the file as `limit2` and type

```
>> limit2
```

at the prompt to see that it works.

5.8 Quitting a Program that is Caught in an Infinite Loop

When you write m-files that contain while loops, sooner or later you will run a program and it will never stop. This can happen because the Boolean expression in the loop never becomes false. This is inconvenient (to say the least). To stop a program in midstream type Ctrl+C on Windows machines and Command+. (the command key with the period key) on Macs.

Worksheet 5

1. *First try to answer this question without using MATLAB. Then, use MATLAB to check your answers.*

   Consider the following code.

```
>> s = 0; n = 1;
>> while n <= 5
s = s + n^2;
n = n + 2;
end
```

   a) What is the value of `s` immediately after this code has been executed? Circle your answer.

   i) $s = 1^2 + 2^2 + 3^2$
   ii) $s = 1^2 + 2^2 + 3^2 + 4^2$
   iii) $s = 1^2 + 2^2 + 3^2 + 4^2 + 5^2$
   iv) $s = 1^2 + 3^2$
   v) $s = 1^2 + 3^2 + 5^2$
   vi) $s = 1^2 + 3^2 + 5^2 + 7^2$

   b) What is the value of `n` immediately after this code has been executed? Circle your answer.

$$n = 4 \qquad n = 5 \qquad n = 6 \qquad n = 7 \qquad n = 8 \qquad n = 9$$

2. Albert wanted to calculate the value of

$$\lim_{x \to 0^+} \frac{2 - 2^{\cos x}}{\sin^2 x}$$

   correct to 6 decimal places and to that end wrote the following m-file.

```
xold = 1;
yold = (2 - 2^cos(xold))/sin(xold)^2;
diff = 1;
while diff > 10^(-7)
  x = 0.1*xold;
  y = (2 - 2^cos(x))/sin(x)^2;
  diff = abs(y - yold);
end
```

a) When Albert runs the program, however, MATLAB doesn't appear to do anything. Why? Circle your answer.

   i) There is an error in the m-file, so MATLAB can't execute the code.

   ii) MATLAB is executing the code but it never stops because the criterion `diff > 10^(-7)` is always satisfied.

   iii) The code is okay and MATLAB is executing the code. It just takes a while because there are a lot of calculations to do.

b) If the code needs to be fixed indicate how it should be fixed.

c) Use the corrected m-file to find the value of the limit. Write your answer correct to 6 decimal places. *(You can use the command* `format long` *to get MATLAB to print out more than 4 decimal places to the command window.)*

3. Recall the m-file script you wrote in Worksheet 3 and the function m-file you wrote in Worksheet 4 to find the value of

$$\int_0^1 e^{-x^2}\,dx$$

using the Trapezoidal rule. Modify your function m-file so that its input is the accuracy to which you want the integral to be calculated and the output is the value of the integral calculated to that accuracy. Your m-file should start by estimating the value of the integral using 2 sub-intervals, then it should estimate the value of the integral using 4 sub-intervals. If the two estimates are the same to the desired accuracy then it should return this value as the output. Otherwise, it should estimate the value of the integral using 8 sub-intervals, etc. Attach a copy of the m-file when you hand in this worksheet. What is the value of the integral correct to 6 decimal places? Write your answer here.

6 Function Handles and If Statements

6.1 Goals of this lesson:

In this lesson you will learn

- what a function handle is;
- how to use a function handle to pass a function as an argument to a function m-file;
- how to use a function handle to create a function in-line; and
- how to write an if statement.

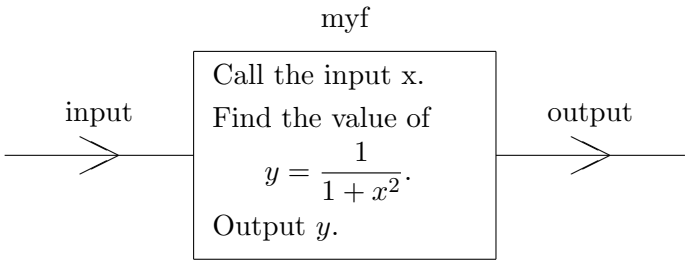
In this lesson you will be creating files that you will need to save somewhere. So, insert your flash drive into the USB port and inside the learnmatlab folder create a folder called Lesson6. In MATLAB, navigate, so that this is your working folder.

6.2 Review of how Function M-files Work

To understand function handles we need to recall how function m-files work. So, let's create a simple function m-file of the function $y = 1/(1 + x^2)$. We'll call this function `myf`. By this time you know how to create such a file; click on **File** → **New** → **M-file**, and type the following in the window that appears.

```
function y = myf(x)
y = 1./(1 + x.^2);
```

Remember to save the file before proceeding. Here is a box picture of this function.



If we type, say,

```
>> temp = myf(2)
temp =
    0.2000
```

at the prompt in the command window, then MATLAB looks for an m-file called `myf`. When it finds it, it sees that it has one input x. Before executing any commands in the m-file it assigns the value 2 to the variable x. In other words, it executes the statement

```
x = 2;
```

in the function workspace. It then executes the commands in the m-file. In this case there is only one and this command tells MATLAB to assign the value $1/(1+2^2) = 0.2$ to the variable y in the function workspace. Since y is the output variable, this value is assigned to the variable `temp` in the main workspace. If, on the other hand, we simply typed

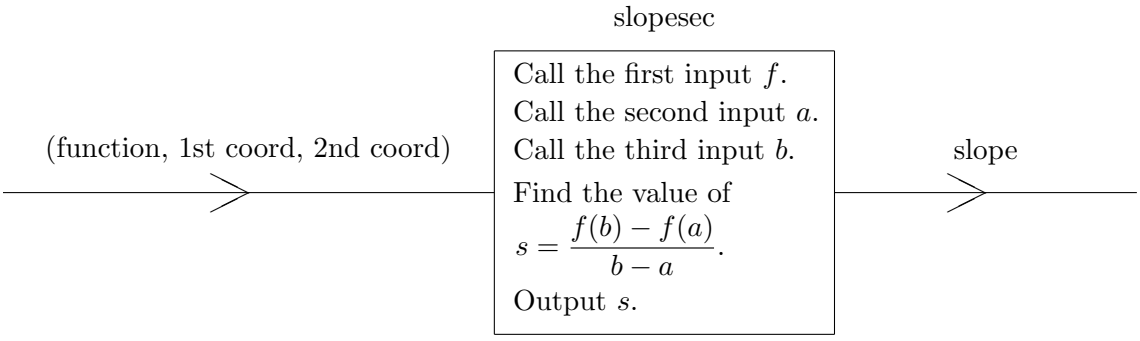
```
>> temp = myf
???  Input argument "x" is undefined.

Error in ==> myf at 3
y = 1./(1 + x.^2);
```

MATLAB complains because it is supposed to assign the output of `myf` to the variable `temp`, but it hasn't been given an input, so it can't produce an output.

6.3 Why We Need Function Handles

We perform many operations on functions. For instance, we take derivatives and integrals of functions, we sketch graphs of functions, and we find zeros of functions. If we want to write an m-file that performs one of these actions and that can be used without modification on any function, then we need the m-file to be a function m-file that has an input that is itself a function.

For instance, suppose we want to write a function m-file `slopesec` that calculates the slope of a secant line. The inputs to the m-file will be the function whose secant line we are interested in and the x-coordinates of the two points on the graph of the function that define the secant line. A box picture of this function m-file would look like:



It is easy enough to write this m-file. Click on **File** → **New** → **M-file** and type the following in the window that appears:

```
function s = slopesec(f, a, b)
s = (f(b) - f(a))/(b - a);
```

Remember to save the file. Now, let's try to use this file to find the slope of the function $f(x) = 1/(1 + x^2)$, that we coded in the m-file `myf`, between $x = 2$ and $x = 3$. It might seem, at first, that we should be able to do this by typing

```
>> slopesec(myf,2,3)
???  Input argument "x" is undefined.

Error in ==> myf at 3
y = 1./(1 + x.^2);
```

but MATLAB complains. Let's carefully understand why it complains. When `slopesec(myf,2,3)` is typed at the prompt in the command window, MATLAB looks for the m-file `slopesec`. When it finds it, it sees that it has three inputs, f, a and b. The first thing it does, before executing any of the commands in the m-file, is attempt to assign `myf` to the variable f, 2 to the variable a, and 3 to the variable b. In other words, it attempts to execute the commands `f = myf`, `a = 2`, and `b = 3`. The last two, `a = 2` and `b = 3` are no problem, but the first one `f = myf` is different because `myf` is a function m-file and not a variable in the workspace with a value assigned to it. MATLAB interprets the command `f = myf` as, ' assign the output of `myf` to the variable f.' As we saw above, it cannot do this because there's no input to create this output. This is why it complains.

What we want MATLAB to do when it comes across the command `f = myf` is to make f into a function that is the same as the function `myf`. We can get this effect by using what is called a *function handle*.

6.4 Introduction to Function Handles

All the variables that we have seen so far in these lessons have had values that were either numbers or, more generally, matrices. A *function handle* is a variable (i.e. a location in memory) whose value is a *function*. Before we look at how to create function handles, let's see how to use them. First, clear your workspace by typing

```
>> clear
```

at the prompt in the command window. Your instructor should have made the file lesson6vars.mat available to you. Save this file in the learnmatlab/Lesson6 folder on your flash drive (or in whatever folder you are using in MATLAB). This file contains MATLAB variables. To access these variables you need to load the file into your workspace. You can do this by typing

```
>> load lesson6vars
```

at the prompt in the command window. To see the name of the variables that have been loaded, type

```
>> whos
 Name        Size  Bytes  Class

 myhandle1   1x1    16    function-handle array
 myfhandle   1x1    16    function-handle array

Grand total is 2 elements using 16 bytes
```

Notice that there are two variables called `myhandle1` and `myfhandle`. They are function handles as indicated in the Class column. In other words, they are variables whose value is a function. As we shall see later, the value of `myhandle1` is the

function $f(x) = \sin(x^2)$ and the value of `myfhandle` is the function defined by the function m-file `myf`, namely $f(x) = 1/(1+x^2)$. We can use the variables to evaluate these functions in the same way that we would if they were m-files. For instance, if we type

```
>> myhandle1(2)
ans =
    -0.7568
```

then MATLAB calculates the value of $\sin(2^2)$. Similarly, we can plot the graph of $f(x) = \sin(x^2)$ by typing

```
>> plot(-5:.01:5, myhandle1(-5:.01:5))
```

Although in these ways `myhandle1` and `myf` behave just as if they were function m-files, they are, in fact, variables. In particular, if you type

```
>> temp = myhandle1;
```

then MATLAB doesn't complain. It simply finds the value of the variable `myhandle1`, which in this case is the function $f(x) = \sin(x^2)$, and assigns this value to the variable `temp`. This means that `temp` is also a function handle. To see this type:

```
>> whos
 Name        Size   Bytes  Class

 myhandle1   1x1    16     function-handle array
 myfhandle   1x1    16     function-handle array
 temp        1x1    16     function-handle array

Grand total is 3 elements using 48 bytes
```

To see that the value of `temp` is the same as the value of `myhandle1` type the following

```
>> myhandle1(4) == temp(4)
ans =
    1
>> figure
>> plot(-5:.01:5, temp(-5:.01:5))
```

Now, let's see how we can use a function handle to solve our problem of passing the function `myf` as an input argument to the function `slopesec`. Instead of typing `slopesec(myf, 2, 3)` at the prompt in the command window, we should type `slopesec(myfhandle, 2, 3)`. Since `myfhandle` is a function handle whose value is the function `myf`, when MATLAB executes the command `f = myfhandle` this makes sense; the variable f becomes a function handle whose value is the same as the value of `myfhandle`, namely `myf`.

```
>> slopesec(myfhandle, 2, 3)
ans =
    -0.1000
```

Notice that this is the correct value since

$$\frac{\frac{1}{1+3^2} - \frac{1}{1+2^2}}{3-2} = -0.1.$$

6.5 Creating Function Handles

It is easy to create function handles. First, let's consider the case when we have a function m-file and we want to create a function handle whose value is that file. We can do this simply by preceding the name of the file with an @ sign. For instance, `@myf` is a function handle whose value is the function m-file `myf`. Thus, if we type

```
>> @myf
ans =
    @myf
```

at the prompt in the command window, then the variable `ans` becomes a function handle whose value is the m-file `myf`. So, for example, it makes sense to type

```
>> ans(2)
ans =
    0.2000
```

Having typed this the variable `ans` has now become a regular variable whose value is a number so now MATLAB complains when we type `ans(2)` again:

```
>> ans(2)
???  Index exceeds matrix dimensions.
```

More usually, when we create a function handle, we will want to give it a name rather than assign it to the variable `ans`. We do this in the same way as any other variable. For instance, the function handle `myfhandle` was created by typing:

```
>> myfhandle = @myf;
```

Notice that, typing `myfhandle(2)` has exactly the same effect as typing `myf(2)`,

```
>> myfhandle(2)
ans =
    0.2000
>> myf(2)
ans =
    0.2000
```

but `myfhandle` is a variable and `myf` is an m-file, so `temp = myfhandle` makes sense whereas `temp = myf` does not:

```
>> temp = myfhandle;
>> temp = myf;
???  Input argument "x" is undefined.

Error in ==> myf at 3
y = 1./(1 + x.^2);
```

You can create a function handle for any MATLAB function in exactly the same way; by preceding the name of the function with an @ sign. For instance

```
>> temp = @sin;
```

assigns the function $f(x) = \sin(x)$ to the function handle `temp`. So, typing

```
>> plot(-3*pi:.01:3*pi, temp(-3*pi:.01:3*p))
```

has the same effect as typing

```
>> plot(-3*pi:.01:3*pi, sin(-3*pi:.01:3*p))
```

If a function is not too complicated, you don't have to have an m-file to create a function handle whose value is that function. You can create it in-line instead. For instance, if you type

```
>> temp = @(x) x.^2 - 4*x;
```

then `temp` becomes a function handle whose value is the function $f(x) = x^2 - 4x$. Since a function handle behaves just like an m-file this is a very convenient way to define functions in MATLAB without having to write an m-file for them. Such a function is called an *anonymous function* since it doesn't have a name. Notice that the x in this expression is a dummy variable; the @(x) that precedes the expression tells MATLAB that the expression $x^2 - 4x$ should be treated as a function of x. For instance

```
>> slopesec(@(x) x.^2 - 4*x, 4, 1)
ans =
    1
```

finds the slope of the secant line of the function $f(x) = x^2 - 4x$ through the points whose x-coordinates are 1 and 4. Similarly,

```
>> slopesec(@(t) t.^2 - 4*t, 4, 1)
ans =
    1
```

does the same thing.

Often we have a function that has more than one input, and we are interested in the function of one variable that is obtained by holding all of the inputs constant except for one of them. We can easily create this function of one variable in-line. For example, consider the function m-file `mortgage` that we used in Lesson 4. Copy this m-file from your flash drive's learnmatlab/lesson4 folder to the folder you are working in now. If you type

```
>> m = @mortgage
```

then `m` is a function handle whose value is the function m-file `mortgage`. This function has three inputs; the amount of money that is borrowed, the number of years over which it is paid back, and the annual interest rate. The function handle `m` cannot produce an output value unless it is given values for all three of the inputs:

```
>> m(200000, 30, 5)
ans =
    1.0727e+03
```

Now, consider the function that is obtained when the number of years over which the mortgage is paid and the annual interest rate are held constant at 30 and 5% respectively. If we type

```
>> m = @(p) mortgage(p, 30, 5)
```

then `m` is a function handle whose value is this function of one variable. If you recall, you looked at this function in Worksheet 4 and you found that it was linear. We could now use our m-file `slopesec` to find its slope (since the slope of a straight line is equal to the slope of all of its secant lines):

```
>> slopesec(@(p) m(p, 30, 5), 100000, 200000)
```

```
ans =
    0.0042
```

When you type the name of a variable at the prompt in the command window, MATLAB finds the value of that variable and prints it out in the command window. So, to see that the function handles `myhandle1` and `myfhandle` do indeed have the values $f(x) = \sin(x^2)$ and `myf` respectively as we claimed above, we can simply type the name of these variables at the prompt in the command window:

```
>> myhandle1
myhandle1 =
    @(x) sin(x.^2)
>> myfhandle
myfhandle =
    @myf
```

6.6 Comment on Writing Function M-files in which One of the Inputs is a Function

As we saw when we wrote the m-file `slopesec` we didn't have to make any special provisions because one of the inputs was a function. We simply gave that variable a name (we called it f in that case but any name would do) and treated it, when writing the file, as if it were a function. It was when we *used* the function `slopesec` that we had to make sure that the value of that variable was a function handle instead of, say, the name of an m-file.

6.7 If Statements

We have looked at two different kinds of *flow control* structures, namely for loops and while loops. In this section we will look at yet another, if statements. An if statement has the form

```
if expression1
    statements1
elseif expression2
    statements2
elseif expression3
    statements3
    ⋮
else
    statements4
end
```

The expressions are all Boolean expressions that may be true or false. The statements following each expression are MATLAB commands and are only executed if the expression preceding them is true. There may be any number of `elseif`'s including none at all. The statements after the `else` are only executed if none of

the previous expressions are true. It is not necessary for `else` to appear in an if statement.

If statements are relatively intuitive. We use them when what we want MATLAB to do depends on the results of previous calculations. As an example, create the following function that finds the number of real roots of the quadratic function $f(x) = ax^2 + bx + c$. If you recall, we created such a function in Lesson 4 by using MATLAB's `sign` function. Here we'll do it by using an if statement. Click on **File** → **New** → **M-file** and type the following in the window that appears.

```
function numroots = quadroots2(a, b, c)
%
% The function numroots = quadroots(a, b, c) finds how many real
% roots the quadratic function f(x) = ax^2 + bx + c has.
% Inputs:
% a (number):  the coefficient of x^2 in the quadratic
% b (number):  the coefficient of x in the quadratic
% c (number):  the constant term in the quadratic
% Output:
% numroots (number):  the number of real roots (either 0, 1, or 2)

delta = b^2 - 4*a*c;
if delta > 0
   numroots = 2;
elseif delta == 0
   numroots = 1;
else
   numroots = 0;
end
```

Remember to save the file. Notice that $f(x) = x^2 + 1$ has no real roots, $f(x) = x^2 - 4x + 4 = (x-2)^2$ has one real root, and $f(x) = x^2 + x - 6 = (x-2)(x+3)$ has two real roots. We can test `numroots2` on these functions by typing:

```
>> numroots2(1,0,1)
ans =
   0
>> numroots2(1,-4,4)
ans =
   1
>> numroots2(1,1,-6)
ans =
   2
```

6.8 An M-File for Finding a Root of a Function Using the Bisection Method

The following function m-file implements the method of bisection to find a root of a function. Recall that in the method of bisection you start with two initial guesses of the root, one where the value of the function is positive and the other where the value of the function is negative. If the function is continuous then this guarantees that there is a root somewhere between these two numbers. You then look at the value of x in between these two numbers. This is the next approximation. You evaluate the function there to determine if it is positive or negative. This gives you an interval that is half the size of the original in which you know there is a root. The method continues until you have the accuracy you desire.

The m-file below implements this algorithm. The function whose root is to be found is an input to the function as are the two initial guesses and the accuracy to which the root is to be found. The implementation illustrates if statements and writing an m-file in which one of the inputs is a function. Although the program looks long, most of it is comments.

Click on **File** → **New** → **M-file** and type the following in the window that appears.

```
function root = bisection(f, x0, x1, tol)
% bisection(f, x0, x1) finds a root of the function f using the
% method of bisection.
%
% Output:
% A number which is the value of the root to the desired accuracy.
%
% Inputs:
% f (function handle):  The function whose root is to be found.
% x0 (number):  A number in the domain of f.
% x1 (number):  A number in the domain of f.
% tol (positive number):  The accuracy to which the root will be
% found.  An actual root of f will lie within +/- tol of the
% value returned by the function.
%
% Notes:
% * f(x0) and f(x1) must have opposite signs (ie one of them must
% be positive and the other negative).  The algorithm finds a
% root that lies between x0 and x1.
% * The function f must be continuous to produce a reliable
% estimate of a root.


% The algorithm only works if f f(x0) and f(x1) have opposite
```

```
% signs.  If they do not, then the following if statement will
% quit the program printing an error message.
if f(x0)*f(x1) > 0
   error('f(x0) and f(x1) must have opposite signs.')
end


% The following if statement ensures that f is negative at x=a
% and positive at x=b.
if f(x0) < 0
   a = x0;
   b = x1;
else
   a = x1;
   b = x0;
end


% If either f(a) or f(b) is equal to 0 then the root has been
% found.  The following if statement sets both a and b equal
% to the root in this case.
if f(a) == 0
   b = a;
elseif f(b) == 0
   a = b;
end


% The root will be the value of (a+b)/2.  It will be determined
% to the desired accuracy when |b-a| < 2*tol.
intervalsize = 2*tol;


% The following loop performs the iterations until the desired
% accuracy is achieved.  If a = b then the loop is skipped
% because the root has already been found.
while abs(b-a) >= intervalsize
   temp = (a+b)/2; % the new value to be tested.
   if f(temp) == 0 % root has been found:  both a and b take on
   this value.
      a = temp;
      b = temp;
   elseif f(temp) < 0 % a becomes the new value.
      a = temp;
   else % (f(temp) > 0) b becomes the new value.
```

```
        b = temp;
    end
end


root = (a+b)/2;
```

Remember to save the file when you're done. Remember that when we use the file to find a root of a function, we'll have to enter the function as a function handle. Test the file by typing in the following commands. Notice that the function $x^2 - 4$ has roots at ± 2, the function $x^2 \sin x$ has roots at all the multiples of π, the function $1/(1 + x^2)$ implemented in `myf` is positive everywhere and doesn't have any roots, and the function $\sin(x^2)$ that is assigned to the function handle `myhandle1` has roots at the square roots of all the multiples of π.

```
>> bisection(@(x) x^2 - 4, 1, 3, 0.0001)
>> bisection(@(x) x^2 - 4, 1.1, 2.04, 0.001)
>> bisection(@(x) x^2 - 4, 1.1, 10, 10∧(-12))
>> bisection(@(x) x^2 - 4, 3, 4, 0.5)
>> bisection(@(x) x^2 - 4, -3, 4, 0.0001)
>> bisection(@(x) (x^2)*sin(x), 0.5, 4, 10∧(-6))
>> bisection(@(x) (x^2)*sin(x), 28.7, 40, 10∧(-6))
>> bisection(@myf, -5, 8, 10^(-6))
>> bisection(myhandle1, 0.2, 1.3, 0.0001)
```

Worksheet 6

1. *(Function handles.)* First try to answer these questions without using MATLAB. Then, check your answers using MATLAB.

   a) Suppose you type

   ```
   >> clear, f = sin; f(2)
   ```

   at the prompt in the command window. How does MATLAB respond? Circle your answer.

   i) It complains because it needs an input to `sin` in order to execute the command `f = sin`.

   ii) It complains because `f` is not a function so it cannot find the value of `f(2)`.

   iii) `ans = 0.9083`

   b) Suppose you type

   ```
   >> clear, myf = x.^2 - 5*x; g = myf; g(3)
   ```

   at the prompt in the command window. How does MATLAB respond? Circle your answer.

   i) It complains because `x` is not a variable in the workspace so it can't execute the command `myf = x.^2 - 5*x`.

   ii) It complains because `myf` is a function and not a variable, so it can't execute the command `g = myf`.

   iii) It complains because `g` is not a function so it can't find the value of `g(3)`.

   iv) `ans = -6`

   c) Suppose you type

   ```
   >> clear, myf = @(x) x.^2 - 5*x; g = myf; g(3)
   ```

   at the prompt in the command window. How does MATLAB respond? Circle your answer.

   i) It complains because `x` is not a variable in the workspace so it can't execute the command `myf = @(x) x.^2 - 5*x`.

   ii) It complains because `myf` is a function and not a variable, so it can't execute the command `g = myf`.

   iii) It complains because `g` is not a function so it can't find the value of `g(3)`.

   iv) `ans = -6`

d) Suppose you type

```
>> clear, x = 1; y = 7; myf = @(x, y) x*y - y^2./x; myf(2, 7)
```

at the prompt in the command window. How does MATLAB respond? Circle your answer.

i) It complains because x and y both have particular values, so it cannot execute the command `myf = @(x, y) x*y - y^2./x`.

ii) It complains because function handles can only be functions of one variable, so it cannot execute the command `myf = @(x, y) x*y - y^2./x`.

iii) `ans = -10.5`

iv) `ans = -42`

e) Suppose you type

```
>> clear, y = 7; myf = @(x) x*y - y^2./x; myf(2)
```

at the prompt in the command window. How does MATLAB respond? Circle your answer.

i) It complains because `x` is not a variable in the workspace, so it cannot execute the command `myf = @(x) x*y - y^2./x`.

ii) It complains because `y` has a particular value, so it cannot execute the command `myf = @(x) x*y = y^2./x`.

iii) It complains because `myf` should have two inputs x and y, so `myf(2)` doesn't make sense.

iv) `ans = -10.5000`

2. *(Using an m-file that has an input that is a function handle.)*

a) Use the m-file `bisection` from lesson 6 to find the root of $f(x) = x - \tan x$ that lies between $\pi/2$ and $3\pi/2$. Find the root with a tolerance of $\pm 10^{-6}$.

b) Use the m-file `bisection` from lesson 6 to find all solutions to the equation $x^3 = 5x - 3$. Find each solution with a tolerance of $\pm 10^{-9}$. *(Note:* `bisection` *will only find one root each time it is used, so you will have to use it more than once to find all of the roots.)*

3. *(Writing an m-file that has a function handle as one of its inputs.)*

Recall the m-files you wrote in Worksheets 3, 4 and 5 to evaluate

$$\int_0^1 e^{-x^2}\, dx$$

using the Trapezoidal Rule. Modify the m-file you wrote for Worksheet 5 so that it finds the value of

$$\int_a^b f(x)\,dx$$

for any function f and any limits a and b. Your m-file should have four inputs; the function whose integral is to be found (f), the lower and upper limits of integration (a and b respectively), and the accuracy to which the integral should be found. Use your m-file to find the value of

$$\int_0^\pi \sin(x^2)\,dx$$

accurate to 4 decimal places.

4. *(If statements.)*

The following code appears in an m-file in which x and y are variables.

```
if x > y
  temp = x;
  x = y;
  y = temp;
end
```

a) If $x = 3$ and $y = 7$ prior to this code being executed, what are the values of x and y after the code is executed?

b) If $x = 7$ and $y = 3$ prior to this code being executed, what are the values of x and y after the code is executed?

c) Describe in words what the code does.