Math 241: Matlab Notes

Dr. Randall Paul

March 5, 2006

Contents

1 Basic Commands

1.1 Introduction

When Matlab is launched a desktop appears which contains several windows. The window we care most about is the *command window*. When you click on the command window a blinking cursor appears after a `>>` prompt. This is the *command line*.

In these notes we will use the convention that anything to be entered into Matlab on the command line (such as the word "input") will appear as:

```
>> input
```

Similarly anything which Matlab prints in response (such as "ans = output") will appear in the same font, but without the `>>`. In general, Matlab prints its response on two lines like so:

```
ans =

    output
```

We suppress this, writing instead:

```
ans =   output
```

These notes contain many numbered exercises. Many ask you to simply type in what is shown and observe the results. Others require some thought and ask you to write some things down. You should work through all the exercises whether they are straight-forward or not. In many cases exercises will not work properly if you have not worked through earlier exercises.

1.2 Matlab as Calculator

The simplest use of Matlab is as a calculator. The normal order of operations applies: expressions within parentheses `( )` first, then exponentiation `^` , then multiplication `*` and division `/`, then addition `+` and subtraction `-`. In most cases all you need worry about is inserting parentheses appropriately.

Exercise 1.2.1: The expression

$$\frac{5.1 + 3^2}{6 - 3 \cdot 3}$$

should be entered into Matlab at the command line as:

```
>> (5.1 + 3^2)/(6 - 3*3)
```

Matlab will respond:

```
ans =  -4.7000
```

Exercise 1.2.2: What should be entered into Matlab so as to evaluate this expression? (your answer should be 17.64)

$\left(\frac{3}{1-\frac{2}{7}}\right)^2$ _____ $= 17.6400$

The familiar division symbol / is called *right division* because you are dividing by the quantity on the right side of the symbol. Matlab also supports *left division* denoted \ where the divisor is on the left side of the symbol.

Exercise 1.2.3:

```
>> 12/3

ans = 4
```

but

```
>> 14\7

ans = 0.5000
```

You don't see left division much since for real numbers `a/b` = `b\a`. Thus you can write all division as right division, and everybody does. This is **not true** for matrices, however, so Matlab provides for both right and left division.

Exercise 1.2.4: Evaluate the following Matlab expression **in your head**, then check using Matlab.

`>> 4 + 8\2 -1` = _____

Matlab supports all standard mathematical functions. Usually the Matlab command for the function is very similar to the normal mathematical notation. For instance:

$\sin(x)$ = `sin( )` $\log_2(x)$ =`log2( )` $\log_{10}(x)$ =`log10( )`

There are some that are not so easy to guess. For instance:

$\sin^{-1}(x)$ = `asin( )` $\ln(x)$ = `log( )` e^x = `exp( )`

Use the *search* feature in *Matlab Help* to find the right command if you have any problems.

Exercise 1.2.5: Use Matlab Help to find the *hyperbolic tangent* of 3.

- Select **Help** from the menu options along the top of the Matlab Desktop.
- Select the **Search** tab, type "hyperbolic tangent", and <enter>.
- Click on different choices until you see "hyperbolic tangent" featured in the window to the right of the search window.
- The correct choice is "tanh(x)". Enter this on the command line.

  ```
  >> tanh(3)

  ans =  0.9951
  ```

Exercise 1.2.6: Evaluate the expression $e^{\sqrt{2}}$

```
>> exp(sqrt(2))

ans =   4.1133
```

Note that the command `exp(2^0.5)` also works.

Exercise 1.2.7: What should be entered into Matlab so as to evaluate this expression? (your answer should be ≈ 1.1719)

$\sqrt[3]{\ln(5)} =$ _____ ≈ 1.1719

1.3 Formatting and Precision

Matlab provides several different ways to display a number using the `format` command. The default is `format short` or four decimal places shown.

Exercise 1.3.1:

```
>> 6\1

ans = 0.1667
```

However,

```
>> format long
>> 6\1

ans = 0.166666666666667
```

The `long` format shows 14 decimal places of precision. One can also have numbers in exponential notation with 4 or 14 decimal places. If a number is too large or small Matlab will automatically shift into exponential notation.

Exercise 1.3.2:

```
>> format short e
>> 6\1

ans = 1.667e-001
```

or 1.667×10^{-1}

There is also `format bank` giving `0.17` (2 decimal places) and `format rat` giving `1/6`. (Fractions, but often this is only an approximation.)

Important note: `format` only effects the number of decimal places **shown**. Unless told otherwise Matlab knows all numbers to 16 significant digits regardless of whether all the digits are shown.

Exercise 1.3.3: Show $\sqrt{1000}$ in all six formats discussed above.

`long` $\Rightarrow$ _____

`short e` $\Rightarrow$ _____

`long e` $\Rightarrow$ _____

`bank` $\Rightarrow$ _____

`rat` $\Rightarrow$ _____

`short` $\Rightarrow$ _____

1.4 Constants and Variables

Matlab knows the value (to 16 significant digits) of various important constants, and has assigned that value a name. If you enter the name into Matlab then Matlab will respond with the value.

Exercise 1.4.1:

```
>> pi
ans = 3.1416
```

Note that Matlab will let you play Indiana state legislature and change the value of `pi`.

Exercise 1.4.2:

```
>> pi = 3
pi = 3
```

As far as Matlab is concerned the value of π is now 3. You can return to reality by using the `clear` command.

```
>> clear pi
>> pi
ans = 3.1416
```

In a doomed effort to satisfy both mathematicians and electrical engineers the names `i` and `j` are both assigned to the imaginary number $\sqrt{-1}$. Matlab can easily evaluate complex expressions.

Exercise 1.4.3: Use Matlab to evaluate the complex expression:

$(1+i)^{10} =$ _____

Matlab also has names for some results that are not numbers.

Exercise 1.4.4:

```
>> 1/0

Warning: Divide by zero
ans = inf
```

or infinity. Similarly,

Exercise 1.4.5:

```
>> 0/0

Warning: Divide by zero
ans = nan
```

or 'not a number'. This is Matlab's way of saying the expression is not defined.

Matlab also has a name for the result of the last calculation: `ans` or 'answer'. Thus you can use the result of the previous calculation in the next.

Exercise 1.4.6:

```
>> 1/6

ans = 0.1667
```

If you then write

```
>> ans - .16

ans = 0.0067
```

Notice that `ans` is not constant—it changes with almost every command you make. Therefore we say `ans` is a *variable*.

In Matlab it's easy to make your own variables. You may assign a name to almost any quantity you like and that name will represent a new variable. A variable name in Matlab must start with a letter, but then may be followed by up to 62 letters, numbers, and underscores (_). In particular, **a variable name cannot contain a space**.

There are also seventeen *reserved names* which cannot be used as variable names. Matlab uses words such as `for` and `function` for other things.

Exercise 1.4.7: Let's give the name `xvalue` to the number `5`.

```
>> xvalue = 5

xvalue =   5
```

Exercise 1.4.8: What would you enter into Matlab to define the variable `e` as the natural base e? (See exercise 1.2.6)

```
>> e =_____________________________
```

As far as Matlab is concerned, typing `xvalue` is exactly the same as typing a `5`, and typing `e` is exactly the same as typing `2.71828182845905`.

For instance, you can even use `xvalue` to give `xvalue` a new value.

Exercise 1.4.9:

```
>> xvalue = xvalue + 1

xvalue =   6
```

Matlab reads `xvalue + 1` as `5 + 1` which equals 6. So `xvalue` is now a name for 6.

Let's try a more concrete exercise.

Exercise 1.4.10: Say a student's score in a class is determined by five separate grades (each out of 100): three in-class tests, homework, and a final. The tests are each 20% of your grade, the homework 10%, and the final 30%.
A student in this class gets 86, 91, and 75 on the tests, 80 on the homework, and 65 on the final. What is the student's score for the class (out of 100)?

We may calculate the grade directly,

```
>> .2*(86 + 91 + 75) + .1*80 + .3*65

ans = 77.9000
```

Exercise 1.4.11: A more general way to solve exercise 1.4.10 is to use variables. We first give names to the grades.

```
>> test1 = 86

test1 = 86
```

Matlab now treats the name `test1` exactly like the number 86.

Matlab's response to statements like this is reassuring at first, but gets old after a while. If you don't want Matlab to respond put a `;` at the end of your command.

Exercise 1.4.12: Continuing with exercise 1.4.10.

```
>> test2 = 91;

>> test3 = 75;

>> homework = 80;

>> final = 65;
```

Now we give a name to our class score using the grades.

```
>> score = .2*(test1 + test2 + test3) + .1*homework + .3*final

score = 77.9000
```

(since we wanted Matlab to tell us, we didn't end with a ;)

How would we calculate what the student would have gotten if they'd scored 80 on the final? You might think the following would give it to you, but it doesn't.

Exercise 1.4.13:

```
>> final = 80;

>> score

score = 77.9000
```

`score` is the name for the number 77.9, **not** how it was calculated. To get the new score (using a final of 80) we have to repeat our old calculation, but that doesn't mean typing it in again. Just click on the *up-arrow* until the command appears, or double-click on the command in the *command history window*.

Exercise 1.4.14:

```
>> score = .2*(test1 + test2 + test3) + .1*homework + .3*final

score = 82.4000
```

After you get the hang of this you'll quickly find yourself up to your eyeballs in variables. Matlab provides several commands to manage them. One is `who` which lists all your variables. Another is `clear` which removes variables.

Exercise 1.4.15:

```
>> who
 ans    homework  test1  test3
 final  score     test2
```

The command `whos` has the same effect, but provides more details about the variables.

Exercise 1.4.16:

```
>> clear test1 test2 test3 homework final
```

clears these variables. Thus

```
>> test1

??? Undefined function or variable 'test1'
```

Exercise 1.4.17: Give the name `poly` to the polynomial: $2x^5 - 3x^3 + 17x^2 - 100x + 2331$. (You'll have to define `x` first.) Evaluate this polynomial for

$x = -2$ _____

$x = 15$ _____

$x = e$ _____

1.5 Script M-files

Previously we saw how to repeat a command made earlier in the session without having to re-write it (using the up-arrow or double-clicking in the command history window). Often, though, we'll have many commands that we'll want to repeat many times with small changes. For this we have *script M-files*.

We need to record the commands we want repeated in a *text file*, so first we must call up a *text editor*. From Matlab this is done by clicking on the empty page icon on the far left of the command bar, or by selecting **File- New- M-File**.

An empty window will appear. Enter the commands you want, then **Save as** <name>**.m** (The '**.m**' part tells Matlab that it is a Matlab file.). When you enter <name> at the command line the commands in the file will be executed.

> **Exercise 1.5.1:** We return to exercise 1.4.10.
> Open a text editor and type:
>
> ```
> test1 = 86;
> test2 = 91;
> test3 = 75;
> homework = 80;
> final = 65;
> score = .2*(test1 + test2 + test3) + .1*homework + .3*final
> ```
>
> Now save this file as: `myscore.m`
> (Notice that the new M-file has appeared in the *current directory window*.)
> Return to the command window, but don't close the text editor.
>
> ```
> >> myscore
>
> score = 77.9000
> ```

The effect of writing `myscore` is exactly the same as writing the commands in the file `myscore.m`.

> **Exercise 1.5.2:** Return to the text editor, change the line `final = 65;` to `final = 80;` and save. Go back to the command line.
>
> ```
> >> myscore
>
> score = 82.4000
> ```

> **Exercise 1.5.3:** Make a script M-file called `mypoly.m` for the polynomial evaluation you did in exercise 1.4.17.

It is inconvenient to have to open the M-file every time you want to change the value of one of your variables. One way to avoid this is to put an `input` command into your M-file.

Exercise 1.5.4: Open (or return to) `myscore.m` in the text editor. Replace the line `final = 65` to `final = input('Grade on final? ')`. Save the file and go back to the command line.

```
>> myscore

Grade on final?
```

(The cursor will be after the question mark.) Type `75` and enter.

```
score = 80.9000
```

This is the score if `final = 75`.

Exercise 1.5.5: Edit your file `mypoly.m` so that Matlab asks you for the value of `x` and then evaluates the polynomial. Copy your file below.

As you write larger and larger M-files it will become more and more difficult for me (or anyone else, including you after a while) to see what each command is supposed to be doing. So that M-files can be understood we insert statement that are ignored by Matlab—but are still important for humans. Such statements are called *comments*.

Matlab ignores anything after a `%` sign.

Exercise 1.5.6: Insert text so that `myscore.m` looks like the following:

```
% Script to calculate score from five grades
test1 = 86; % Grades
test2 = 91;
test3 = 75;
homework = 80;
final = input('Grade on final? ')
% 20% on exams, 10% homework, 30% final
score = .2*(test1 + test2 + test3) + .1*homework + .3*final
```

Save and return to the command window.

```
>> myscore

Grade on final?
```

Type `75` and enter.

```
score = 80.9000
```

You see `myscore.m` is completely unaffected by the addition of comments.

> **Exercise 1.5.7:** Insert a comment at the beginning of `mypoly.m` saying what the program does. Type `mypoly` at the command line just to make sure it still works.

2 Matrices

2.1 Matrix Arithmetic by hand

The name 'Matlab' is short for 'Matrix Laboratory' since the fundamental object in Matlab is the matrix. Even the simple numbers we considered in the last section are, for Matlab, just 1×1 matrices.

So what is a matrix? A matrix is a block of numbers (called *elements*), organized into rows and columns. The *dimensions* of the matrix, $n \times m$, are the number of rows (n) and the number of columns (m).

$$\begin{bmatrix} 2 & 8 & 7 & 2 \\ 1 & -5 & 15 & -3 \\ -4 & 11 & 7 & 9 \end{bmatrix}$$

This is a 3×4 matrix.

Matrices of the same dimension can be added simply by adding the corresponding entries;

$$\begin{bmatrix} 9 & 5 & 1 \\ 11 & -7 & 8 \end{bmatrix} + \begin{bmatrix} -2 & 5 & 7 \\ 12 & 6 & -8 \end{bmatrix} = \begin{bmatrix} 7 & 10 & 8 \\ 23 & -1 & 0 \end{bmatrix}$$

but matrices of different dimensions cannot be added.

$$\begin{bmatrix} 4 & -9 & 3 \end{bmatrix} + \begin{bmatrix} 5 \\ -8 \\ 3 \end{bmatrix} = \text{Not defined}$$

> **Exercise 2.1.1:** Add the matrices below, and give the dimension of the result.
>
> $$\begin{bmatrix} 9 & 5 \\ 10 & 8 \\ -6 & -2 \end{bmatrix} + \begin{bmatrix} -7 & 5 \\ 3 & -8 \\ 5 & 4 \end{bmatrix} =$$
>
> Dimension = _____

Matrices of the correct dimensions may also be multiplied, but the process is much more complicated. The number of columns in the left matrix must be equal to the number of rows in the right matrix. The resulting matrix will have the same number of rows as the left matrix, and the same number of columns as the right matrix.

For instance, multiplying a 4×2 matrix by a 2×3 matrix is defined ($2 = 2$). The result is a 4×3 matrix. However the product in the reverse order (2×3 times 4×2) is **not defined** since $3 \neq 4$.

In a product of two matrices, to find the number in the n-th row and m-th column take the n-th row of the left matrix and the m-th column of the right matrix, multiply corresponding elements, and then sum these products.

Let's find the entry in the first row, first column for the product of the two matrices below.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix}$$

First note that we're multiplying a 2×3 by a 3×2, so the multiplication is defined. The resulting matrix will be 2×2.

The first row on the left is $1 \quad 2 \quad 3$, the first column on the right is $7 \quad 9 \quad 11$. Multiplying and adding gives us:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 1 \cdot 7 + 2 \cdot 9 + 3 \cdot 11 & \\ & \end{bmatrix} = \begin{bmatrix} 58 & \\ & \end{bmatrix}$$

Exercise 2.1.2: Calculate the other three entries of the product matrix above.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 & \\ & \end{bmatrix}$$

Exercise 2.1.3: Calculate the product of the matrices below.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \begin{bmatrix} 10 \\ 11 \\ 12 \end{bmatrix} =$$

2.2 Matrix Arithmetic in Matlab

The easiest way to enter a relatively small, simple matrix into Matlab is to type in the entries between `[` and `]` signs. The entries should be separated by spaces, and the rows by `;` 's.

Exercise 2.2.1: Give the name `A` to the matrix

$$\begin{bmatrix} 2 & 8 & 7 & 2 \\ 1 & -5 & 15 & -3 \\ -4 & 11 & 7 & 9 \end{bmatrix}$$

```
>> A = [2 8 7 2; 1 -5 15 -3; -4 11 7 9]
A =  2   8   7   2
     1  -5  15  -3
    -4  11   7   9
```

The Matlab command `size( )` returns the dimensions of the matrix to which it is applied.

Exercise 2.2.2:

```
>> size(A)
ans =   3   4
```

since A is a 3×4 dimensional matrix.

Exercise 2.2.3: What would you write in Matlab to give the name `B` to the following matrix?

$$\begin{bmatrix} 9 & 5 & 1 \\ 11 & -7 & 8 \end{bmatrix}$$

>> _____

Exercise 2.2.4: What would you write in Matlab to give the name `C` to the following matrix?

$$\begin{bmatrix} 10 \\ 11 \\ 12 \end{bmatrix}$$

>> _____

It is also possible to make bigger matrices from smaller ones using the same notation, but entering matrices instead of numbers. The only restriction is that the result must be a rectangular block of numbers (that is, each row must have the same number of columns and vice versa).

Exercise 2.2.5:

```
>> [A C]

ans =  2   8   7   2  10
       1  -5  15  -3  11
      -4  11   7   9  12
```

Exercise 2.2.6: What would you write in Matlab to use the matrices `B` and $\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$ to make the matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 9 & 5 & 1 \\ 11 & -7 & 8 \end{bmatrix}$$

```
>> _____________________________
```

As with numbers, Matlab makes it easy to do matrix arithmetic—as long as the operation is defined!

Exercise 2.2.7:

```
>> B*A

 ans = 19   58  145   12
      -17  211   28  115
```

Exercise 2.2.8:

```
>> A + B

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

Exercise 2.2.9: Of the expressions below decide **in your head** which quantities are defined and circle them. For those that are defined use Matlab to find the result.

```
A*B   A*C     C*A     B*C
C*B   A + C   B + C   C + C
```

Results:

Another important operation that can be performed on matrices is *transposition*, or exchanging the rows and columns of a matrix. In Matlab this is accomplished with the apostrophe `'`.

Exercise 2.2.10:

```
>> B'

ans = 9  11
      5  -7
      1   8
```

Exercise 2.2.11: Find the transpose of the matrix `C`, and write the result below.

In Matlab the apostrophe is also used for *complex conjugation*, so if you apply it to a matrix that has complex elements you will get the *transpose-conjugate* matrix.

Exercise 2.2.12:

```
>> D = [1+i 2+3*i]

D = 1.0000 + 1.0000i   2.0000 + 3.0000i

>> D'

ans = 1.0000 - 1.0000i
      2.0000 - 3.0000i
```

2.3 Arrays

A $1 \times n$ dimensional matrix is often called a list or an *array*. Arrays are very important in Matlab, so there are special methods for generating them. One method is to use the form:

[<begin>:<step>:<end>]

This generates an array whose first element is <begin>, whose second is <begin> + <step>, third is <begin> + 2<step>, etcetera until <end> is reached or exceeded.

Exercise 2.3.1:

```
>> K = [3 : 0.5 : 5]

K = 3.0000   3.5000   4.0000  4.5000   5.0000
```

Exercise 2.3.2:

```
>> L = [6 : -2 : -3]

L = 6   4   2   0  -2
```

Exercise 2.3.3: How many elements are in the array?

[15 20 25 ... 95 100]

```
>> M = [15 : 5 : 100];
>> size(M)
ans =   1  18
```

There are 18 elements in the array.

If the middle quantity <step> is omitted, then Matlab assumes it is +1.

Exercise 2.3.4:

```
>> [3:7]
ans =   3   4   5   6   7
```

A second method of generating arrays has the form:

`linspace(` <begin>, <end>, <length> `)`

This generates an array of length <length> whose first element is <begin>, and whose **last** is <end>. The intermediate elements are spaced evenly between <begin> and <end>.

Exercise 2.3.5:

```
>> K = linspace(3, 5, 5)
K = 3.0000   3.5000   4.0000  4.5000   5.0000
```

Exercise 2.3.6: How would you use `linspace` to define `L` (as in exercise 2.3.2)?

```
>> L = linspace(       ,       ,      )
```

Exercise 2.3.7: What array would the command `linspace(3, 9, 4)` produce? (Try to figure it out in your head, then check using Matlab.)

```
ans =
```

2.4 Elements and Element-wise operations

Up until now we have referred to matrices as single objects, but what about the elements? How do you deal with the numbers that make up the matrix on an individual basis?

If you want to refer to or change one element in a matrix you put the matrix name, then the row and column of the element in parentheses. If the matrix is one-dimensional (only one row or column) you need only put one number in parentheses.

Exercise 2.4.1: Using matrix `A` from exercise 2.2.1.

```
>> A(2,3)

A(2,3) = 15
```

Exercise 2.4.2: Using matrix `C` from exercise 2.2.4.

```
>> C(2) = 7;
>> C

C = 10
     7
    12
```

Exercise 2.4.3: What would you write in Matlab to change the element in the first column, second row of `B` to zero? (the matrix `B` is from exercise 2.2.3.)

```
>> _____________________________
ans =   9   5   1
        0  -7   8
```

The colon operator (`:`) may be used as 'the entire row' or 'the entire column'.

Exercise 2.4.4:

```
>> A(2,:)

ans = 1  -5  15  -3
```

Exercise 2.4.5: What would you write in Matlab to produce the second column in `B`?

```
>> _____________________________
ans =  5
      -7
```

You can also use an array as indexes as long as the elements are positive integers.

Exercise 2.4.6:

```
>> A(1:2,2:3)

ans =   8   7
       -5  15
```

This is a square matrix composed of the elements in the first and second rows, and second and third columns of the matrix `A` from exercise 2.2.1. (See below.)

$$\begin{bmatrix} 2 & 8 & 7 & 2 \\ 1 & -5 & 15 & -3 \\ -4 & 11 & 7 & 9 \end{bmatrix}$$

Exercise 2.4.7: What would you write in Matlab to produce the following sub-matrix of `A`?

```
>> A(     ,     )

ans =   1  -5  15
       -4  11   7
```

We have seen how matrix addition and subtraction simply add or subtract the corresponding elements of two matrices of the same dimension. What if we wanted to multiply corresponding elements? Recall that matrix multiplication **does not** do this. However element-wise multiplication will.

A period (`.`) followed by any operation makes the operation *element-wise*.

Exercise 2.4.8: The following matrix operation is not defined since you cannot multiply a 3×1 matrix by another 3×1 matrix. (Recall `C` from exercise 2.2.4)

```
>> C*C

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

However with element-wise multiplication,

Exercise 2.4.9:

```
>> C.*C

ans = 100
       49
      144
```

Similarly `C^2` is undefined since `C^2` = `C*C`, but

Exercise 2.4.10:

```
>> C.^2

ans = 100
       49
      144
```

Exercise 2.4.11:

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

E =  1   2
     3   4

>> F = [5 6; 12 8]

F =  5   6
    12   8

>> F./E

ans = 5   3
      4   2
```

but notice that

```
>> F/E

ans =  -1   2
      -12   8
```

hence *element-wise right division* is different from *matrix right division* (which we haven't yet discussed).

Exercise 2.4.12: What will `E.*F` be? What will `E./F` be?
(Try to do them in your head first, then check with Matlab.)

Let's consider left division. Element-wise it's the same as right division.

Exercise 2.4.13:

```
>> E.\F

ans = 5   3
      4   2
```

since you're still dividing the elements of `F` by the elements of `E`. However for matrix left division

Exercise 2.4.14:

```
>> E\F

ans =  2.0000  -4.0000
       1.5000   5.0000
```

the result is different. We'll discuss this more in section 2.5.

Like operations, some functions treat a matrix as a matrix while others act element-wise. In general those functions which don't deal with a matrix property directly act element-wise. This includes all the familiar functions of one variable like `sin`, `log`, etc.

Exercise 2.4.15:

```
>> G = pi.*[0: 0.5: 2]

G = 0   1.5708   3.1416   4.7124   6.2832

>> sin(G)

ans = 0   1.0000   0.0000  -1.0000  -0.0000
```

We will use this property a great deal when we get to graphing functions (section 3.3).

2.5 Matrix Division

Matrices have a wild variety of applications, but certainly they are most useful for *solving systems of linear equations*. The first step in doing so is to put such a system in the form of a matrix equation.

Exercise 2.5.1: The following system of three linear equations and three unknowns can be put into matrix form.

$$\begin{aligned} 4x - 3y + 2z &= 7 \\ 2x + 5y + z &= -1 \\ -x + 7y - 6z &= 4 \end{aligned}$$

First we consider it as an equation of two 3×1 matrices,

$$\begin{bmatrix} 4x - 3y + 2z \\ 2x + 5y + z \\ -x + 7y - 6z \end{bmatrix} = \begin{bmatrix} 7 \\ -1 \\ 4 \end{bmatrix}$$

then we write the left side as a product of a 3×3 and a 3×1

$$\begin{bmatrix} 4 & -3 & 2 \\ 2 & 5 & 1 \\ -1 & 7 & -6 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 7 \\ -1 \\ 4 \end{bmatrix}$$

Exercise 2.5.2: Put the following system of four linear equations and four unknowns into matrix form.

$$\begin{aligned} x_1 + 5x_2 - 2x_3 + 4x_4 &= 1 \\ -3x_1 + 6x_2 + 9x_3 - 7x_4 &= 2 \\ 4x_1 + 7x_2 + 3x_3 + 3x_4 &= -2 \\ 5x_1 - 6x_2 + x_3 + x_4 &= -1 \end{aligned}$$

How then do we solve such systems? First we put the system into the form: $AX = B$ where A is an $n \times n$ matrix that we know, B is an $n \times 1$ matrix that we know, and X is an $n \times 1$ matrix that we don't know. To solve for X we need only *matrix left divide* both sides by A. Thus,

$$AX = B \quad \Rightarrow \quad A\backslash AX = A\backslash B \quad \Rightarrow \quad X = A\backslash B$$

Exercise 2.5.3: Solve the system

$$\begin{array}{r} 4x - 3y + 2z = 7 \\ 2x + 5y + z = -1 \\ -x + 7y - 6z = 4 \end{array} \quad \Longleftrightarrow \quad \begin{bmatrix} 4 & -3 & 2 \\ 2 & 5 & 1 \\ -1 & 7 & -6 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 7 \\ -1 \\ 4 \end{bmatrix}$$

```
>> A = [4 -3 2; 2 5 1; -1 7 -6];
>> B = [7; -1; 4];
>> X = A\B
```

```
X =   2.1469
     -0.6923
     -1.8322
```

Therefore $x \approx 2.1469$, $y \approx -0.6923$, and $z \approx -1.8322$. It's easy enough to check our answer, just recall that $x =$ `X(1)`, $y =$ `X(2)`, and $z =$ `X(3)`.

Exercise 2.5.4: Check the first equation, $4x - 3y + 2z = 7$

```
>> 4*X(1) - 3*X(2) +2*X(3)
```

```
ans =   7.0000
```

or check the whole system.

```
>> A*X
```

```
ans =   7.0000
       -1.0000
        4.0000
```

which is the matrix `B`.

Exercise 2.5.5: Solve the system:

$$\begin{array}{r} x_1 + 5x_2 - 2x_3 + 4x_4 = 1 \\ -3x_1 + 6x_2 + 9x_3 - 7x_4 = 2 \\ 4x_1 + 7x_2 + 3x_3 + 3x_4 = -2 \\ 5x_1 - 6x_2 + x_3 + x_4 = -1 \end{array}$$

$x_1 =$ _____
$x_2 =$_____
$x_3 =$ _____
$x_4 =$ _____

Warning: You can divide by any number except zero, but matrices are trickier. First, you may only divide by square matrices. Second, you may only divide by matrices whose *determinant* is not zero. Moreover if the determinant of a matrix is very small then division by that matrix may lead to large numerical errors.

Exercise 2.5.6: Matlab can easily calculate determinants.

```
>> det(A)

ans = -143
```

So you may (and did) divide by the matrix `A`.

```
>> C = [1 2; 2 4];
>> det(C)

ans = 0
```

So you **cannot** divide by the matrix `C`. If you try,

```
>> D = [5 7];
>> D/C

Warning: Matrix is singular to working precision.
ans = -inf   inf
```

2.6 Standard Matrices

Matlab has functions which generate certain standard matrices in whatever shape is wanted.

Exercise 2.6.1:

```
>> zeros(2,3)

ans =   0   0   0
        0   0   0

>> ones(3,5)

ans =   1   1   1   1   1
        1   1   1   1   1
        1   1   1   1   1

>> I = eye(3)

I =   1   0   0
      0   1   0
      0   0   1
```

This last is called the *identity matrix*. It's called this because it is the *multiplicative identity* for matrix multiplication. That is, it's like multiplying by 1.

Exercise 2.6.2:

```
>> I*A

ans = 4  -3   2
      2   5   1
     -1   7  -6
```

So $I * A = A$. Note that this is different from the *element-wise identity*, `ones`.

Exercise 2.6.3:

```
>> I.*A

ans = 4   0   0
      0   5   0
      0   0  -6
```

which is **not** `A`.

```
>> id = ones(3)

id =   1   1   1
       1   1   1
       1   1   1

>> id.*A

ans = 4  -3   2
      2   5   1
     -1   7  -6
```

which **is** `A`.

Note as well that `zeros`, `ones`, `eye`, generate a square matrix if given only one dimension.

Exercise 2.6.4: What would you write in Matlab to give the name `E` to the following matrix?

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

```
>> _____________________
```

Exercise 2.6.5: What would you write in Matlab to give the name `F` to the following matrix? (See exercise 2.2.6)

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

```
>> _____________________
```

Application: Splines

A *spline* is a polynomial (or a piece-wise combination of polynomials) used to approximate numerical data.

Part I: In this part we will be looking for the 3rd degree polynomial, p, satisfying the conditions: $p(1) = 5$, $p(3) = 7$, $p'(1) = -1$, and $p'(3) = 0.5$.
We know a 3rd degree polynomial has the form:

$$p(x) = a_1x^3 + a_2x^2 + a_3x + a_4$$

We need to find the constants $a_1,\ a_2,\ a_3,$ and a_4.

Begin by making each of the conditions into a linear equation with unknowns $a_1 \ldots a_4$.

$$p(1) = \quad 5 \quad \Rightarrow$$

$$p(3) = \quad 7 \quad \Rightarrow$$

$$p'(1) = -1 \quad \Rightarrow$$

$$p'(3) = 0.5 \quad \Rightarrow$$

Now solve the linear system.

$a_1 =$ _____
$a_2 =$ _____
$a_3 =$ _____
$a_4 =$ _____

Part II: In this part we will be looking for the 3rd degree polynomial, q, satisfying the following conditions:

$$q(x_1) = \quad y_1$$

$$q(x_2) = \quad y_2$$

$$q'(x_1) = \quad d_1$$

$$q'(x_2) = \quad d_2$$

Write a script M-file called `oitspline.m` which gets the values $\{x_i\}$, $\{y_i\}$, and $\{d_i\}$ from `input` statements, then calculates and prints out the constants $\{a_i\}$. Insert comments and e-mail the script M-file to me.

3 Output Methods

3.1 Strings

A *character string* is a special kind of array. It contains number codes which correspond to symbols on the keyboard. We've already seen strings used with the `input` command introduced in exercise 1.5.4. For instance for `input('x value= ')`, the group of characters between the apostrophes, "`x value= `", is the string.

Giving a name to a string is like giving a name to a matrix, but you enclose the string in apostrophes (`'`) rather than square brackets (`[ ]`).

Exercise 3.1.1: Give the name `saying` to the character string "Thanks for your support. "

```
>> saying = 'Thanks for your support. '

saying =   Thanks for your support.
```

It's important to remember that a string is a kind of array, with all the rules and commands that go with arrays.

Exercise 3.1.2: Find the length of the string `saying`.

```
>> size(saying)

ans =   1  25
```

The array `saying` contains 25 characters in one row. (Yours might be only 24 if you didn't include the last space.)

For instance you can select just part of the string by using the `:` operator.

Exercise 3.1.3: Extract the word "`Thanks`" from the string `saying`.

```
>> saying(1:6)

ans =   Thanks
```

Exercise 3.1.4: What would you write in Matlab to extract the word "`support`" from the string `saying`?

```
>> _____________________

ans =   support
```

Exercise 3.1.5: What would you write in Matlab to extract the letters "`Tak o orspot`" from the string `saying`? (see exercise 2.3.1)

```
>> _____________________

ans =   Tak o orspot
```

Again, like matrices, strings can be glued together or *concatenated*.

Exercise 3.1.6:

```
>> fruits = ['oranges' 'apples' 'bananas']

fruits =   orangesapplesbananas
```

Making a vertical list is a little tricky, though. Each row must have the same number of columns, but "apples" only has six letters while the others have seven.

Exercise 3.1.7: Make a vertical listing of the fruits.

```
>> fruits = ['oranges'; 'apples'; 'bananas']

??? Error using ==> vertcat
All rows in the bracketed expression must have the same
number of columns.
```

but if you add a space at the end of "apples" then they are all seven characters long.

Exercise 3.1.8: Make a vertical listing of the fruits.

```
>> fruits = ['oranges'; 'apples '; 'bananas']

fruits =   oranges
           apples
           bananas
```

The function `char` does this automatically.

Exercise 3.1.9: Make a vertical listing of the generals.

```
>> generals = char('Krum','Heraclius','Vercingetorix')

generals =   Krum
             Heraclius
             Vercingetorix
```

3.2 Displaying Strings

Strings are the main means by which a person interacts with a program. The `input` command uses a string to ask for information. Other commands use strings to answer questions and output data.

The simplest way to provide information is to just display a string. When you want to display just the content of the string, without the clunky "name =" portion, use the display command, `disp( )`.

Exercise 3.2.1:

```
>> disp(saying)

Thanks for your support.
```

More complicated output requires a more complicated function. If the information you are displaying includes the content of one or more numerical variables use the command, `sprintf( )`.

Exercise 3.2.2: Calculate and display the area of a circle of radius 3.

```
>> radius = 3;
>> area = pi*radius^2;
>> sprintf('The area of a circle of radius %.5g is %.5g', radius, area)


ans =   The area of a circle of radius 3 is 28.274
```

Notice that the values of the variables `radius` and `area` are inserted into the string where there appears a `%.5g`. The `%.` part tells `sprintf( )` that a number is to be inserted. The `5` tells `sprintf( )` how many digits to display, and the `g` means "general format". There are other formats, but this one will be sufficient for our purposes.

Exercise 3.2.3: What would you write in Matlab to calculate and display the volume of a sphere of radius 4.2? (The formula for the volume of a sphere is $V = \frac{4\pi}{3}r^3$.)

```
>> _____

>> _____

>> _____
```

Exercise 3.2.4: Write a script M-file called `area_cyl` which asks the user for a radius and height, then calculates and displays the resulting surface area. After you've tested it, write your script M-file below. The formula for the surface area of a cylinder is $S = 2\pi r(h + r)$.

To make a % actually appear in a string displayed using `sprintf( )` write the % twice.

Exercise 3.2.5: Calculate and display what percentage 31 is of 83. Show only one decimal place.

```
>> top = 31;
>> bottom = 83;
>> percentage = 31/83*100;
>> sprintf('%.2g is %.3g%% of %.2g', top, percentage, bottom)
ans =   31 is 37.3% of 83
```

We use the same method to display apostrophes ' or left division \. If you want to include an apostrophe or left division in a string write it twice. Other keystrokes can be inserted using backslash-character codes. For instance, `\n` = <enter> or 'new line'.

Exercise 3.2.6: What would you write in Matlab to generate the following output?

```
>> _____________________________________________
ans =  My son's score was:
       31 out of 83!
```

Exercise 3.2.7: Edit your file `oitspline.m` so that it displays the actual third degree polynomial, rather than just the coefficients $a_1 \ldots a_4$. For example using the values in part I of the application, the output should be:

```
ans = -0.6250 x^3 + 4.1250 x^2 + -7.3750 x + 8.8750
```

Write the lines you added below.

3.3 2-D Plotting

One of the most useful things that Matlab does is plot the graphs of functions. There are an enormous number of different kinds of graphs (polar, semi-log, scatter, etc), and Matlab does them all. However we will concentrate on just a few.

The simplest plotting command in Matlab is `plot`. This command takes two arguments, an array `x` which contains all the x values, and another array `y` which contains the y values. The `plot` function adds axes and connects the points with lines.

Exercise 3.3.1: The following commands produce a blocky-looking parabola between -2 and 2, plotted using only five points.

```
>> x = linspace(-2,2,5)
x =  -2  -1  0  1  2
>> y = x.^2
y =   4   1  0  1  4
>> plot(x,y)
```

A *figure window* titled `Figure 1` should appear containing a rather sad-looking parabola. If we want it to look smooth (like a real parabola) we'll need to use more than five points.

Exercise 3.3.2: The following commands produce a much smoother parabola, plotted using forty-one points.

```
>> x = linspace(-2,2,41);
>> y = x.^2;
>> plot(x,y)
```

Now `Figure 1` holds a much nicer-looking parabola.

The x and y axes may be labelled using the commands `xlabel` and `ylabel`. Likewise the graph can be given a title with the `title` command.

Exercise 3.3.3:

```
>> xlabel('x values');
>> ylabel('y values');
>> title('Parabola');
```

Note that the labels and title have been added to `Figure 1`.

Exercise 3.3.4: Plot the graph of $y = \sin(x)$ from $-\pi$ to 2π using 50 points. Label the x-axis "radians", and give it the title "Sinusoid". Write the commands you used below.

```
>> x =_____________________________

>> y =_____________________________

>> plot(_____________________________

>> _____________________________

>> _____________________________
```

Notice that our old parabola graph (exercise 3.3.2) is gone. If we want to put two or more graphs on the same axes we use the `hold on` command. When we're ready to do an entirely new graph or set of graphs we use `hold off`.

Multiple graphs can be set off from one another by using color. You can force the use of a color by putting certain letters between single quotes in the `plot` function. (`'b'` is blue, `'g'` green, `'r'` red, `'c'` cyan, `'m'` magenta, `'y'` yellow, and `'k'` black.)

Exercise 3.3.5: Graph the polynomial $y = x - \frac{1}{6}x^3$ on the same axes as the sine graph above (exercise 3.3.4). Let this graph be red.

```
>> hold on
>> y = x - x.^3/6;
>> plot(x,y,'r')
```

Sometimes the axes that Matlab chooses automatically are not the best. You can impose your own limits on the x and y axes with the `axis` command. The form of the command is:

`axis( [` <min x>, <max x>, <min y>, <max y> `] )`

Exercise 3.3.6: Force the x axis to run from $-\pi$ to 2π, and the y axis from -2 to 2.

```
>> axis([-pi, 2*pi, -2, 2]);
```

Exercise 3.3.7: Graph $y = \cos(x)$ and $y = 1 - \frac{1}{2}x^2 + \frac{1}{24}x^4$ on the same axes, the x-axis from -2π to 2π, the y-axis from -2 to 2. Use fifty points, let the graph of the polynomial be magenta, and write the commands below.
(The graphs of $y = \sin(x)$ and $y = x - \frac{1}{6}x^3$ should **not** be present.)

Exercise 3.3.8: Edit your file `oitspline.m` so that the polynomial is plotted between `x1` and `x2` using 50 points. Write the lines you added below.

3.4 3-D Plotting

The graphs we studied in section 3.3 had the form $y = f(x)$. This leads to a two dimensional graph, with variables x and y. Graphs in this section will have the form $z = f(x, y)$, and so are three dimensional—involving variables x, y, and z.

Using Matlab to do a three dimensional graph requires some intermediate steps. First you define the range on the x and y axes, and these will be simple arrays. The z, however will be a two-dimensional array—a matrix. To calculate z you need to turn x and y into square matrices as well. For this we use the `meshgrid` command.

Exercise 3.4.1: Define x and y ranges going from -2 to 2. Make square matrices `X` and `Y` with these ranges.

```
>> x = [-2:2];
>> y = [-2:2];
>> [X Y] = meshgrid(x,y)

X =   -2 -1  0  1  2
      -2 -1  0  1  2
      -2 -1  0  1  2
      -2 -1  0  1  2
      -2 -1  0  1  2

Y =    2  2  2  2  2
       1  1  1  1  1
       0  0  0  0  0
      -1 -1 -1 -1 -1
      -2 -2 -2 -2 -2
```

We use the matrices `X` and `Y` to generate `Z` according to a function of two variables.

Exercise 3.4.2: Calculate `Z` for the function, $f(x, y) = x^2 + y^2$.

```
>> Z = X.^2 + Y.^2

Z =    8  5  4  5  8
       5  2  1  2  5
       4  1  0  1  4
       5  2  1  2  5
       8  5  4  5  8
```

Finally we use the command `mesh` to graph the paraboloid. (Make sure the `hold` is `off` before using this command.)

Exercise 3.4.3: Graph the function $z = x^2 + y^2$.

```
>> mesh(X,Y,Z)
```

If you click on the *Rotate 3D* button on the figure window (next to the hand), you can then click and hold onto the figure. This allows you to pull the figure around, viewing it from different perspectives.

Again to get a nice smooth paraboloid you need more than 25 points. We repeat the process with $20 \times 20 = 400$ points.

Exercise 3.4.4: Graph the function $f(x, y) = x^2 + y^2$ for x and y between -2 and 2.

```
>> x = linspace(-2,2,20);
>> y = linspace(-2,2,20);
>> [X,Y] = meshgrid(x,y);
>> Z = X.^2 + Y.^2;
>> mesh(X,Y,Z)
```

Exercise 3.4.5: Graph the function below for x and y between -4 and 4. Use $50 \times 50 = 2500$ points, and write the commands below.

$$f(x, y) = \frac{1}{1 + x^2 + 4y^2}$$

```
>> _____

>> _____

>> _____

>> _____

>> _____
```

There are other commands besides `mesh` which can be used to make the graph after `X`, `Y`, and `Z` have been calculated. `surf` generates a *surface plot*.

Surface plots can also be given *shading*. Shading can be *flat* or *interp*.

Exercise 3.4.6:

```
>> surf(X,Y,Z)
>> shading interp
```

4 Programming

4.1 Logical Type and Operations

We've talked about several complicated data types; now we'll talk about the simplest. *Logical* type take only two values: 1 or 0. The value 1 should be interpreted as "true", and the value 0 as "false".

We don't define a variable as the logical type. Logical variables are almost always the result of some *relational* or *logical* operation.

Exercise 4.1.1:

```
>> 2 > 5

ans =  0
```

The relational operator `>` works just like + or * in that it takes two quantities and returns a value. The value, though, is logical. In this case, since 2 is not bigger than 5, the value is 0 (meaning the statement is **false**.)

Relational operators can also work with arrays.

Exercise 4.1.2:

```
>> y = [1:6]
y =   1 2 3 4 5 6
>> y > 2
ans =  0 0 1 1 1 1
```

The first two values in the array `y` are not greater than 2, so they return "false" or 0. The next four are greater than 2, so they return 1's.

Matlab's relational operators are listed below.

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

Notice that relational equal `==` is different from assignment `=` which **sets** one thing equal to another.

Exercise 4.1.3: How will Matlab respond to the following commands? (Guess and then check.)

```
>> y == 4
```

```
>> y = 4
```

Logical operators are different from relational operators in that they take two logical values and return another logical value. The main logical operators are "AND" (`&`) and "OR" (`|`).

<statement1> `&` <statement2>= 1 (true) if both <statement1> AND <statement2> are true. Otherwise it equals 0 (false).

<statement1> `|` <statement2>= 1 if either <statement1> OR <statement2> is true. It equals 0 only if they're both false.

Exercise 4.1.4:

```
>> y = [1:6]
y =   1 2 3 4 5 6
>> true_false = (y>=2)&(y<=4)
true_false =   0 1 1 1 0 0
```

You get a 1 (true) for 2, 3, and 4 since those numbers are both greater than or equal to 2 AND less than or equal to 4.

Exercise 4.1.5: How will Matlab respond to the following command using OR? (Guess and then check.)

```
>> (y>4)|(y<=2)

ans =   _____________________________________
```

It's often a good idea to put parentheses `( )` around expressions involving relational or logical operators. The reason is that Matlab generally performs these operations last.

4.2 Conditional Statements

Sometimes we only want a command to be given if some condition is satisfied. For instance maybe we would like to divide by some variable, but only if the variable is not equal to zero. Matlab supports several commands that execute "conditionally". The simplest is the construction:

```
if <logical>
   <command>
end
```

Here <command> will only be executed if <logical> is 1 (true).

`if-end` constructions are different from the Matlab commands we have seen before because you can type <enter> without any command being executed. After you type `if` Matlab will not execute commands until you type `end`.

If you make a mistake in one of these types of statements, you need to terminate the command before you can start over. For this press <Control-C>. This should give you back a command line prompt (`>>`), which means you are ready to start again.

Exercise 4.2.1:

```
>> top = 20;
>> bottom = 4;
>> if bottom~=0
     top/bottom
   end

ans =   5
```

since the logical statement `bottom~=0` (`bottom` is not equal to 0) is true, 20 is divided by 4. However,

```
>> bottom = 0;
>> if bottom~=0
     top/bottom
   end
```

yields nothing since the logical statement is false. (`bottom` is NOT not equal to 0)

For the most part the <enter>s and the indents are not necessary for making a conditional statement work. We insert them to make the commands more readable. However, `end` (and `else` and `elseif` defined below) must have their own lines or there will be an error.

A slightly more complicated construction has the form:

```
if <logical>
   <command1>
else
   <command2>
end
```

Here <command1> is executed only if <logical> is true, and <command2> will be executed only if <logical> is false.

Exercise 4.2.2:

```
>> if bottom~=0
     top/bottom
   else
     disp('Can''t divide by zero!')
   end

Can't divide by zero!
```

Most of the commands discussed in this lesson are most effective when used in a script M-file, hence many of the exercises will ask you to make one. Be sure to test you script M-file to make sure it does what it's supposed to do.

Exercise 4.2.3: How will Matlab respond to the commands?

```
>> a = [1:5];
>> a = a.^2;
>> if (a(3)>5)
     sprintf('Wow! It''s %.2g!',a(3))
   else
     sprintf('Oh, it''s only %.2g.',a(3))
   end

ans = _____
```

Exercise 4.2.4: Make a script M-file called `up1back10.m` which takes a variable `x` and returns `x+1` if `x` is less than 10, and `x-10` otherwise. Write the script M-file below.

Our last kind of conditional statement has the form:

```
if <logical1>
   <command1>
elseif <logical2>
   <command2>
else
   <command3>
end
```

If <logical1> is true then <command1> is executed. If <logical1> is false and <logical2> is true then <command2> is executed. If both are false then <command3> is executed.

Exercise 4.2.5:

```
>> firmness = 4;
>> if (firmness > 5)
     disp('Too hard!')
   elseif (firmness < 5)
     disp('Too soft!')
   else
     disp('Just right!')
   end

Too soft!
```

4.3 Loops

In mathematical programming it is common to perform very many, very similar calculations. These sequences of repeated commands are called *loops*. Matlab supports the two most common types of loops: `for` and `while` loops. As with conditional statements, a loop statement doesn't execute until you enter `end`.

The form of a `for` loop is:

```
for <variable>=<array>
   <statements>
end
```

Matlab will execute <statements> once for every column in <array>. During the first execution <variable> equals the first column in <array>. During the second execution <variable> equals the second column, etc.

Exercise 4.3.1:

```
>> for k = [1:3]
     disp(sprintf('%.1g times through the loop.', k))
   end

1 times through the loop.
2 times through the loop.
3 times through the loop.
```

The array `1 2 3` has three columns so the `disp` command is executed three times. The first time `k` equals the first column, 1. The second time `k` is 2, and the third 3.

Exercise 4.3.2: How will Matlab respond to the following commands?

```
>> for k = [99:-1:96]
     disp(sprintf('%.2g bottles of beer on the wall.', k))
   end
```

An application for loops is to produce lists of numbers called *sequences*. One example of a sequence is the *geometric sequence:* $1, 2, 4, 8, 16 \ldots$.

How might we use Matlab to generate the geometric sequence? In Matlab a sequence is just an array, which we can call `A`.

Exercise 4.3.3: To begin, set the first number in the array to be 1.

```
>> A(1) = 1;
```

Then set the second number, which is twice the first number.

```
>> A(2) = 2*A(1)

A = 1 2
```

`A(2)` is two times `A(1)`, so `A(2)`$= 2 * 1 = 2$. Likewise the third number is twice the second number, the fourth number is twice the third, etc.

```
>> A(3) = 2*A(2);
>> A(4) = 2*A(3)

A = 1 2 4 8
```

We can perform this process much faster by using a loop.

Exercise 4.3.4: Use a loop to find the first six numbers in the geometric sequence.

```
>> clear A
>> A(1) = 1;
>> for k = [2:6]
     A(k) = 2*A(k-1);
   end
>> A

A = 1  2  4  8  16  32
```

Exercise 4.3.5: How will Matlab respond to the following commands? (It's important that you guess first, then check.)

```
>> clear A
>> A(1) = 2;
>> for k = [2:5]
     A(k) = A(k-1) + 4;
   end
>> A
```

Exercise 4.3.6: In the sequence $\{3, 7, 15, 31, \ldots\}$ the first number is 3 and all the others are found by multiplying the previous number by 2 and then adding 1. Write a script M-file called `sequence.m` which produces the first 10 numbers in this sequence. Write `sequence.m` below.

Exercise 4.3.7: The *Fibonacci sequence* is a sequence of numbers with the properties that the first two numbers are 1's, and all the others are the sum of the previous two. Hence the first six numbers in the Fibonacci sequence are:

$$1, 1, 2, 3, 5, 8, \ldots \qquad (1+1=2, \quad 1+2=3, \quad 2+3=5, \quad 3+5=8\ldots)$$

a) What is the next number in the sequence? _____

b) Write a script M-file called `fibon.m` which takes a variable `n` and returns the first n numbers in the Fibonacci sequence. Write `fibon.m` below.

The next topic is really a combination of the previous two, a *conditional loop*. When using a `for` loop one knows exactly how many times the loop will be repeated—the number of columns in an array. A `while` loop repeats for as long as a given condition remains true. The format is:

```
while <logical>
   <statements>
end
```

So <statements> will be repeated until <logical> is 0 (false).

Exercise 4.3.8:

```
>> x = 3;
>> while (x>0)
     x = x-1
   end
x =   2
x =   1
x =   0
```

Once `x` is no longer greater than zero the loop ends.

Exercise 4.3.9: How will Matlab respond to the following commands?

```
>> x = 1;
>> while (x<30)
     x = 3*x
   end
```

4.4 Functions

As we have seen, Matlab has many functions defined for your use (`sin( )`, `det( )`, `size( )`, ...). You may, however, define your own functions. The process is very much like making a script M-file, but there are important differences.

Exercise 4.4.1: Open the text editor just as if you were getting ready to make an script M-file. Write the following in the editor and save as `multi.m`.

```
function out=multi(x,y)
%returns the product of the arguments
out=x*y;
```

Now we have a function called `multi` which takes two numbers and multiplies them together. (Ok, it's not the most useful function, but you have to walk before you run.)

Returning to the command line:

Exercise 4.4.2:

```
>> multi(-3,5)

ans =  -15
```

Inside the function, `x` takes on the value -3 and `y` takes the value 5. The variable `out` is assigned the product, -15. Notice that `out` appears in the first line. It is our *output variable*. When the last command in `multi.m` is executed, our function returns the value of the output variable, in this case -15.

A big difference between script M-files and functions is that the variables inside a function are *local*. That means that outside the function, they don't exist. So for instance, you don't have to `clear` them.

Exercise 4.4.3:

```
>> x

x =   81
```

This value of `x` is left over from exercise 4.3.9. Notice that it is **not** -3, the value of `x` inside the function.

```
>> out

??? Undefined function or variable 'out'.
```

The variable `out` doesn't exist outside `multi`.

The comment after the `function` line is also important. You can see it if you write `help` and the function name.

Exercise 4.4.4:

```
>> help multi

returns the product of the arguments
```

The general form of a function file is:

`function` <output variable>=<function name> (<arguments>)
<comments>
<commands>

There may be many arguments separated by commas, or none at all. The function file should be saved as <function name>`.m`

Exercise 4.4.5: Given the function file `mod7.m` below,

```
function out = mod7(x)
% calculates x modulo 7
while (x<0)
    x = x + 7;
end
while (x>=7)
    x = x - 7;
end
out = x;
```

how will Matlab respond to the commands?

```
>> mod7(-4)

ans = ______

>> mod7(16)

ans = ______
```

Exercise 4.4.6: Edit your file `fibon.m` so that it is a function taking `n` as an argument, returning an array containing the first n numbers in the Fibonacci sequence. Write the new file below.

Exercise 4.4.7: Edit your file `oitspline.m` (see exercise 3.3.8) so that it is a function taking six arguments (`x1, x2, y1, y2, d1, d2`) and returning a string representing the cubic polynomial and its graph.
So, for example:

```
>> oitspline(1,3,5,7,-1,0.5)

ans =  -0.625x^3 + 4.125x^2 + -7.375x + 8.875
```

Write your new file below.

Application: Euler's Method

Euler's method for finding the solution to a differential equation was invented by the eighteenth century Swiss mathematician Leonard Euler (pronounced "oil-er"). While there are now much better methods, Euler's method is simple and effective.

An *initial value problem* is a differential equation together with a starting point. The solution is a function, $y = f(t)$, which passes through the starting point and whose slope y' agrees with the differential equation for each time t.

Consider the following initial value problem.

$$y(0) = 3 \qquad y' = y^2 - 4y$$

Euler's method produces a sequence $\{y_k\}$ according to the rule,

$$y_1 = 3, \qquad y_k = y_{k-1} + \Delta t(y_{k-1}^2 - 4y_{k-1})$$

where Δt is a small number called the *time step*. The smaller Δt is, the more accurate Euler's method will be. For this application we let $\Delta t = 0.1$.

1. Write a script M-file called `euler.m` that finds the first 51 numbers in the Euler's method sequence, `Y(k)`, for the initial value problem given above.

   Then let `T = [0: .1: 5]` and plot `Y` verses `T`.

   (Set both the x and y axes to run from 0 to 5.)

2. Edit your script M-file so that it is a function called `euler.m` which takes as its argument y at $t = 0$.

   (So, `euler(3)` should give us the same graph as part 1 above.)

   Email `euler.m` to me.

Application: The Sieve of Eratosthenes

Among the many accomplishments of the third century B.C.E. scientist Eratosthenes is a method for calculating all the *prime numbers* between 2 and an arbitrary integer n.

An integer is prime if the only positive integers that divide it evenly are 1 and itself. For instance 7 is prime, but 9 is not (since 3 divides 9 evenly).

The Sieve works like this. List all the numbers from 2 to n. Start at 2. Now throw away all the multiples of 2: $4, 6, 8 \ldots$. Go to the next number you haven't thrown away which is 3. Throw away all the multiples of 3. The next number is 5 (since you threw away 4). Throw away all the multiples of 5. Continue until you get to the square root of n.

The set of numbers you didn't throw away are all the primes between 2 and n.

Part I: Apply the Sieve by hand to the integers from 2 to 100 below.

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41
42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61
62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81
82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	

primes =

Part II: Write a Matlab function called `sieve.m` which takes n as its argument and returns an array with all the primes between 2 and n. E-mail me the function file.

Suggestions:

- Define an array called `list` that is composed of n ones. `list(k)==1` means that k is on the list. `list(k)==0` means k has been thrown off the list.
- Define a loop that makes an array called `primes` which has all the k's for which `list(k)==1`.

  So if `list = 0 1 1 0 1 0` then `primes = 2 3 5`.
- Define a loop that given a number `p` throws out all the multiples of `p`.
- Finally define a loop for the `p`'s, starting from 2 and going to `sqrt(n)` that throws out all the multiples of `p` if `list(p)==1`.