SCHOOL OF PHYSICS, ASTRONOMY & MATHEMATICS

4PAM1008 MATLAB

3 – Creating, Organising & Processing Data

Dr Richard Greenaway

3 Creating, Organising & Processing Data

In this Workshop the matrix type is introduced which is the basic type in MATLAB. You will be shown how to create and manipulate matrices and arithmetic operations will be covered in detail. Before looking at the matrix we will study how to manipulate complex numbers.

3.1 Complex Numbers

As we have already seen, the symbols `i` and `j` by default represent the imaginary number $\sqrt{-1}$, thus we can represent a complex number in MATLAB as follows

```
z = 2 + 5j
```

In keeping with engineering practice we will use `j`. Note however that MATLAB always returns `i` in answers.

Alternatively you can create a complex value using the `complex` function.

```
>> z = complex(2,5)

z =

   2.0000 + 5.0000i

>>
```

You can add, subtract, multiply and divide complex numbers just as you would real numbers.

```
>> z1 = 3 - 4j

z1 =

   3.0000 - 4.0000i

>> z2 = 1 - j

z2 =

   1.0000 - 1.0000i

>> z1 + z2

ans =

   4.0000 - 5.0000i
```

There are various functions available for operating on complex numbers, some of which are listed in Table 3-1.

Function	Description
`real`	Returns the real part of a complex number.
`imag`	Returns the imaginary part of a complex number.
`conj`	Returns the complex conjugate.
`abs`	Returns the magnitude of a complex number.
`angle`	Returns the argument (phase angle) of the complex number.

Table 3-1

© Richard Greenaway 2011

Recall that a complex number, $z = a + ib$, can be represented in polar form as

$$z = Re^{j\theta}$$

where

$$R = \sqrt{a^2 + b^2} \quad \text{and} \quad \theta = \tan^{-1}\left(\frac{b}{a}\right)$$

The magnitude, *R*, can be obtained directly using the `abs` function and likewise θ can be obtained using `angle`.

```
>> z = 4 + 7j

z =

   4.0000 + 7.0000i

>> R = abs(z)

R =

    8.0623

>> theta = angle(z)

theta =

    1.0517

>> z = R*exp(i*theta)

z =

   4.0000 + 7.0000i

>>
```

You could of course calculate the argument using the inverse tangent (`atan` function) but `angle` will always be correct regardless of the quadrant in which the value lies in the complex plane.

Exercise 3-1

1. Given that $u = 7 - 3j$ and $v = 4 + 2j$, calculate the following.
   i) $u + v$
   ii) $4u - v$
   iii) $u^2 - v$
   iv) u/v

2. For $z = 2 - 3j$

   i) Determine z^3

© Richard Greenaway 2011

De Moivre's theorem states that

$$z^n = r^n e^{jn\theta}$$

ii) Convert the value z to exponential form and recalculate z^3 using De Moivre's theorem.

3. A circuit with a resistor and inductor in series has total impedance given by

$$z = R + j2\pi fL$$

Where *R* is the resistor value and *L* is the inductance. If an AC current of frequency *f* Hz is driving the load, the voltage as measured across the two components will be out of phase with the current by an amount θ, equal to the phase angle (argument) of *z*.
i) Determine the phase of the output voltage when

$$R = 5\Omega, \qquad L = 20 \times 10^{-3}\ \text{H}, f = 40\ \text{Hz}$$

ii) Convert the angle to degrees

iii) Confirm the result by calculating the angle directly from the equation,

$$\theta = \tan^{-1}\left(\frac{\mathbb{I}}{\mathbb{R}}\right)$$

where $\mathbb{I}$ is the imaginary part and $\mathbb{R}$ is the real part.

3.2 Matrices

The most basic MATLAB data structure is the matrix: a two-dimensional, rectangularly shaped data structure capable of storing multiple elements of data in an easily accessible format. These data elements can be numbers, characters, logical states of true or false, or even other MATLAB structure types. MATLAB uses these two-dimensional matrices to store single numbers and linear series of numbers as well. In these cases, the dimensions are 1-by-1 and 1-by-n respectively, where n is the length of the numeric series. That is, the simple variables you have encountered so far such as `x = 10` can be considered as a 1 x 1 matrix.

MATLAB also supports data structures that have more than two dimensions. These data structures are referred to as *arrays* in the MATLAB documentation. We will not cover arrays in this course.

We will first discuss how to create, access and manipulate matrices and then cover how operators and functions operate on matrices.

3.2.1 Creating Matrices

© Richard Greenaway 2011

3.2.1.1 Constructing a Simple Matrix

The simplest way to create a matrix in MATLAB is to use the matrix constructor operator, `[]`. Create a row in the matrix by entering elements (shown as `E` below) within the brackets. Separate each element with a comma or space:

```
row = [E1, E2, ..., Em]      or     row = [E1 E2 ... Em]
```

For example, to create a one row matrix of five elements, type

```
>> A = [12 62 93 -8 22]

A =

    12    62    93    -8    22
```

To start a new row, terminate the current row with a semicolon:

```
A = [row1; row2; ...; rown]
```

This example constructs a 3 row, 5 column (or 3-by-5) matrix of numbers. Note that all rows must have the same number of elements:

```
A = [12 62 93 -8 22; 16 2 87 43 91; -4 17 -72 95 6]
A =
    12    62    93    -8    22
    16     2    87    43    91
    -4    17   -72    95     6
```

Exercise 3-2

Create the following matrix.

$$\begin{bmatrix} -2 & 4 & 3 & 0 \\ 1 & 9 & 296 & 2 \\ -3 & 4 & 34 & 0 \end{bmatrix}$$

3.2.1.2 Generating a Numeric Sequence

You will often have need to create a numeric sequence, a vector of equally spaced values, for instance creating a the x-values for a function you wish to evaluate over a range and plot.

The colon operator (*first*:*last*) generates a 1-by-n matrix (or vector) of sequential numbers from the first value to the last. The default sequence is made up of incremental values, each 1 greater than the previous one:

```
A = 10:15
A =
    10    11    12    13    14    15
```

The numeric sequence does not have to be made up of positive integers. It can include negative numbers and fractional numbers as well:

© Richard Greenaway 2011

```
A = -2.5:2.5
A =
   -2.5000   -1.5000   -0.5000    0.5000    1.5000    2.5000
```

You can change the step value by using the following extended syntax.

`A = first:step:last`

To generate a series of numbers from 10 to 50, incrementing by 5, use

```
A = 10:5:50
A =
    10    15    20    25    30    35    40    45    50
```

You can increment by noninteger values. This example increments by 0.2:

```
A = 3:0.2:3.8
A =
    3.0000    3.2000    3.4000    3.6000    3.8000
```

To create a sequence with a decrementing interval, specify a negative step value:

```
A = 9:-1:1
A =
    9    8    7    6    5    4    3    2    1
```

Exercise 3-3

1. Create a vector between -30 and 50 in steps of 10

2. Create a vector between 0 and 2π in steps of $\pi/10$.

3.2.1.3 Using Specialised Matrix Functions

There are a number of functions provided for generating matrices, some of which are listed in Table 3-3

Function	Description
`zeros`	Create a matrix or array of all zeros.
`ones`	Create a matrix or array of all ones.
`eye`	Create a matrix with ones on the diagonal and zeros elsewhere.
`magic`	Create a square matrix with rows, columns, and diagonals that add up to the same number.
`rand`	Create a matrix or array of uniformly distributed random numbers.

Table 3-2

```
>> A = rand(5)
```


© Richard Greenaway 2011

```
A =

    0.7943    0.2630    0.0838    0.5383    0.9619
    0.3112    0.6541    0.2290    0.9961    0.0046
    0.5285    0.6892    0.9133    0.0782    0.7749
    0.1656    0.7482    0.1524    0.4427    0.8173
    0.6020    0.4505    0.8258    0.1067    0.8687
```

3.2.1.4 Concatenating Matrices

It is simple to concatenate matrices wither row-wise or column-wise using the semi-colon operator or comma operator in the same manner as they are used to construct a matrix. Note the use of square brackets since we are creating a new matrix.

```
>> % Concatenate two matrices row-wise
>> A = ones(1,5)

A =

     1     1     1     1     1

>> B = rand(3,5)

B =

    0.8147    0.9134    0.2785    0.9649    0.9572
    0.9058    0.6324    0.5469    0.1576    0.4854
    0.1270    0.0975    0.9575    0.9706    0.8003

>> C = [A;B]

C =

    1.0000    1.0000    1.0000    1.0000    1.0000
    0.8147    0.9134    0.2785    0.9649    0.9572
    0.9058    0.6324    0.5469    0.1576    0.4854
    0.1270    0.0975    0.9575    0.9706    0.8003


>> % Concatenate two matrices column-wise
>> D = [1; 2; 3; 4;]

D =

     1
     2
     3
     4

>> E = zeros(4,2)

E =

     0     0
     0     0
     0     0
     0     0

>> F = [D,E]

F =

     1     0     0
```


© Richard Greenaway 2011

```
     2     0     0
     3     0     0
     4     0     0

>>
```

3.2.2 Accessing Elements

You can access a single element in an array using the following notation

`A(row,column)`

```
>> A = [2 6 9; 4 2 8; 3 5 1]

A =

     2     6     9
     4     2     8
     3     5     1

>> A(2,3)

ans =

     8
```

To access multiple elements we us the colon operator. Subscript expressions involving colons refer to portions of a matrix. The expression

`A(1:m, n)`

refers to the elements in rows 1 through `m` of column `n` of matrix A.

A colon on its own specifies the whole row or column.

```
>> A(:,2)

ans =

     6
     2
     5
```

The indexing syntax used above to return parts of a matrix can also be used for selectively assigning values to an existing matrix.

For instance.

```
>> A = 1:10

A =

     1     2     3     4     5     6     7     8     9    10

>> A(2:4) = 0

A =
```


© Richard Greenaway 2011

```
     1     0     0     0     5     6     7     8     9    10
```

You can remove a row or column by assigning an 'empty' matrix - `[]`.

```
>> A = rand(4)

A =

    0.1419    0.9595    0.9340    0.3922
    0.4218    0.6557    0.6787    0.6555
    0.9157    0.0357    0.7577    0.1712
    0.7922    0.8491    0.7431    0.7060

>> A(2,:) = []

A =

    0.1419    0.9595    0.9340    0.3922
    0.9157    0.0357    0.7577    0.1712
    0.7922    0.8491    0.7431    0.7060
```

Exercise 3-4

1. Create a 9 x 9 matrix of random numbers.
2. Set the central 3x3 square elements to zero.
3. Create a 9 x 1 vector of ones.
4. Replace the 8th column with this vector.
5. Reduce the matrix to a 7 x 7 by removing the outer columns and rows.

The result should look something like this.

```
    0.7447    0.7757    0.6443    0.5870    0.2277    0.1111    1.0000
    0.1890    0.4868    0.3786    0.2077    0.4357    0.2581    1.0000
    0.6868    0.4359         0         0         0    0.4087    1.0000
    0.1835    0.4468         0         0         0    0.5949    1.0000
    0.3685    0.3063         0         0         0    0.2622    1.0000
    0.6256    0.5085    0.9390    0.8443    0.1848    0.6028    1.0000
    0.7802    0.5108    0.8759    0.1948    0.9049    0.7112    1.0000
```

3.2.3 Other Matrix Operations

Here are some other useful matrix operations.

© Richard Greenaway 2011

3.2.3.1 Transpose

A matrix can be transposed simply using the apostrophe operator.

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

A =

     1     2     3
     4     5     6
     7     8     9

>> B = A'

B =

     1     4     7
     2     5     8
     3     6     9
```

3.2.3.2 Flip a Matrix

The `flipud` and `fliplr` functions can be used to flip a matrix horizontally or vertically.

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

A =

     1     2     3
     4     5     6
     7     8     9

>> B =flipud(A)

B =

     7     8     9
     4     5     6
     1     2     3
```

3.3 Functions and Matrices

One of the powerful features of MATLAB is the way functions handle inputs of vectors and matrices. Consider for example the standard trigonometric functions `sin`, `cos` etc. In MATLAB, as in all generic languages, if you pass a value to the function it will return the associated result of the specified operation. However if you pass a matrix, the function will return a matrix of the same size containing the results calculated for all the input values.

Consider calculating the sine of 3 values. Here is how you would do it in the C programming language.

```
double result [3];

double  vector [3] = { 0.01 , 3.1415926536/2, 3.1415926536 };
for (int i = 0 ; i<3 ;i++)
{
   result[i] = sin(vector[i]);
}
```

In MATLAB it takes just one line …

© Richard Greenaway 2011

```
>> sin([0.001 pi/2 pi])

ans =

    0.0010    1.0000    0.0000
```

Now consider functions which operate on multiple values by default. To calculate the average of a set of values we use the `mean` function and pass a vector containing those values.

```
>> a = rand(1,10)

a =

    0.8147    0.9058    0.1270    0.9134    0.6324    0.0975    0.2785    0.5469    0.9575
0.9649

>> mean(a)

ans =

    0.6239
```

Now, in the case where the input is a matrix the behaviour is different from that of functions like `sin`. `mean` does **not** calculate the average of all the values but treats each *column* as a separate vector.

```
>> A = rand(10,5)

A =

    0.4505    0.1067    0.4314    0.8530    0.4173
    0.0838    0.9619    0.9106    0.6221    0.0497
    0.2290    0.0046    0.1818    0.3510    0.9027
    0.9133    0.7749    0.2638    0.5132    0.9448
    0.1524    0.8173    0.1455    0.4018    0.4909
    0.8258    0.8687    0.1361    0.0760    0.4893
    0.5383    0.0844    0.8693    0.2399    0.3377
    0.9961    0.3998    0.5797    0.1233    0.9001
    0.0782    0.2599    0.5499    0.1839    0.3692
    0.4427    0.8001    0.1450    0.2400    0.1112

>> mean(A)

ans =

    0.4710    0.5078    0.4213    0.3604    0.5013
```

This *columnwise* behaviour occurs frequently. Another example is data plotting, where if you pass a matrix to the `plot` function, it will treat each column in the matrix as a separate data set. Thus if you pass an `n x 3` matrix to plot you will get a graph with 3 lines plotted.

Here is an example.

© Richard Greenaway 2011

Example. 3-1

```
>> % Create the values of theta for which we sill calculate sine and cosine
>> x = 0:0.1:2*pi;
>> % Transpose it to make it a column vector
>> x = x';
>> % Calculate sine and cosine for those angles and put the results in an n x 2 matrix
>> result = [sin(x) , cos(x) ];
>> % plot the sine and cosine curves against angle
>> plot(x,result)
```

The plot should look like this …

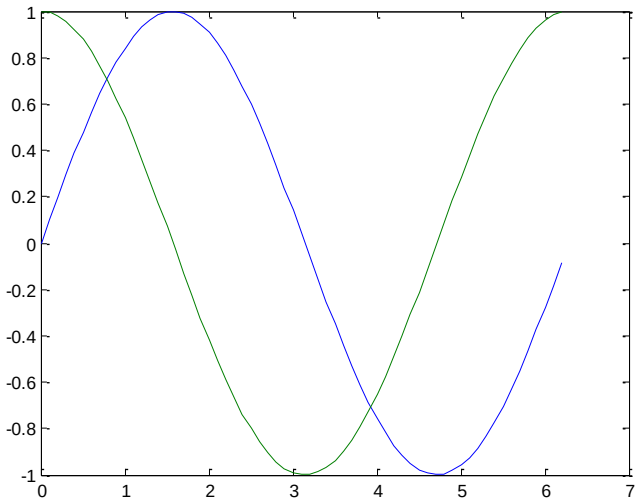


Fig. 1 The result from Example. 3-1

Exercise 3-5

Calculate

$$\sin\theta, \quad \sin(\theta - \pi/2) \;\; and \;\; \sin(\theta - \pi)$$

for $0 \leq \theta \leq 2\pi$ at intervals of 0.1 radians.

Plot the results.

You must use the `sin` and `plot` functions only once.

HINT

After you create the numeric sequence you must convert it from a row of numbers to a column of numbers using the *transpose* operator.

© Richard Greenaway 2011

3.4 Arithmetic Operations

You have already used arithmetic operators when dealing with single values. We will now generalise their use to matrices. You should begin to see how powerful MATLAB is at processing data and how easy it is to operate on complete data sets without the need for traditional loop constructs used in generic programming languages.

3.4.1 Addition and Subtraction

Two matrices of the same size can be added or subtracted. The operation is done on an element by element basis. You can also add a scalar to a matrix.

```
>> A = ones(3)

A =

     1     1     1
     1     1     1
     1     1     1

>> B = A + 4

B =

     5     5     5
     5     5     5
     5     5     5

>> C = eye(3)

C =

     1     0     0
     0     1     0
     0     0     1

>> D = B + C

D =

     6     5     5
     5     6     5
     5     5     6
```

3.4.2 Multiplication

The multiply operator, *, we have seen earlier in fact implements a *matrix multiply* operation. That is, the product of an $m \times p$ matrix $\boldsymbol{A}$ with an $p \times n$ matrix $\boldsymbol{B}$ is an $m \times n$ matrix $\boldsymbol{C}$ whose entries are

$$\boldsymbol{C}_{ij} = \sum_{k=1}^{p} \boldsymbol{A}_{ik}\boldsymbol{B}_{kj}$$

And in general,

$$\boldsymbol{AB} \neq \boldsymbol{BA}$$

So, in MATLAB,

© Richard Greenaway 2011

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

A =

     1     2     3
     4     5     6

>> B = [1 2 ; 3 4 ; 5 6]

B =

     1     2
     3     4
     5     6

>> A*B

ans =

    22    28
    49    64

>> B*A

ans =

     9    12    15
    19    26    33
    29    40    51
```

Remember, the **inner** matrix dimensions must agree. That is, the number of *columns* in the left operand must equal the number of *rows* in the right operand.

```
>> A = [ 1 2 3 4; 5 6 7 8]

A =

     1     2     3     4
     5     6     7     8

>> B = [1 2 ; 3 4 ; 5 6]

B =

     1     2
     3     4
     5     6

>> A * B
??? Error using ==> mtimes
Inner matrix dimensions must agree.
```


© Richard Greenaway 2011

3.4.3 Division (Solving Systems of Linear Equations)

Recall that we can use matrices to solve systems of linear equations.

The following equations

$$x + 3y = 7$$
$$3x - 2y = -12$$

can be represented in matrix form as

$$\begin{bmatrix} 1 & 3 \\ 3 & -2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 7 \\ -12 \end{bmatrix}$$

The general form for a system of linear equations is

$$\boldsymbol{A}u = b$$

which can be solved as follows.

$$\boldsymbol{A}^{-1}\boldsymbol{A}u = \boldsymbol{A}^{-1}b$$
$$\therefore u = \boldsymbol{A}^{-1}b$$

In MATLAB we could solve the equation set above as follows.

```
>> A = [1 3 ; 3 -2]

A =

     1     3
     3    -2

>> b = [7 ; -12]

b =

     7
   -12

>> inv(A)*b

ans =

   -2.0000
    3.0000
```

However, there is a quicker way to do this without explicitly determining the inverse matrix and that is to carry out a *matrix left division* using the *back-slash* operator.

```
>> A = [1 3 ; 3 -2]

A =

     1     3
     3    -2

>> b = [7 ; -12]

b =

     7
   -12
```


© Richard Greenaway 2011

```
>> u = A\b

u =

    -2
     3
```

Functionally, `A\B` corresponds to `inv(A)*b`.

Exercise 3-6

Solve the following equations

1.

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

2.

$$3x - 2y = 7$$
$$6x - 4y = 4$$

Explain this answer

3.

$$2x - 4 = 3y$$
$$6x = 9y + 12$$

Explain this answer

4.

$$2x + 3y + 5z = 9$$
$$x + 2y + 7z = 13$$
$$4x + 11y + 3z = -1$$

5. Repeat question 4 by calculating the inverse explicitly.

3.4.4 Matrix Power

Since, for example

$$\boldsymbol{A}^2 = \boldsymbol{A} \times \boldsymbol{A}$$

The power operator simply does a matrix multiply of a matrix by itself *n* times where *n* is the exponent.

© Richard Greenaway 2011

3.5 Element-wise (Array) Arithmetic Operations

When operating on matrices there are other forms of arithmetic operation which are of importance, in particular operations which act on the elements of a matrix or vector individually, known in MATLAB as array operations.

3.5.1 Array Multiplication (the Hadamard Product)

Array multiplication (known as the *Hadamard Product*) is used extensively in MATLAB. This is the entry-wise product of two matrices of the same size, or of a matrix and a scalar. The Hadamard product of $\boldsymbol{A}$ and $\boldsymbol{B}$ is

$$\boldsymbol{C} = \boldsymbol{A} \circ \boldsymbol{B}$$

where,

$$\boldsymbol{C}_{ij} = \boldsymbol{A}_{ij}\boldsymbol{B}_{ij}$$

In MATLAB we use the *dot-star* operator : '**.***' to compute this product.

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

A =

     1     2
     3     4

>> A.*A

ans =

     1     4
     9    16
```

This operator is particularly useful in MATLAB because it allows us to do an element-by-element multiply of data in a single command. That is we can generate multiple data points for complex functions which are the product of two or more other functions in one line of code.

3.5.2 Array Division

The division equivalent of array multiplication is exactly what you would expect. If matrix $\boldsymbol{A}$ is divided by matrix $\boldsymbol{B}$, the result is a matrix of the same size where each element is the result of dividing the corresponding elements of $\boldsymbol{A}$ by $\boldsymbol{B}$.

```
>> a = ones(1,5)

a =

     1     1     1     1     1

>> b = 1:5

b =

     1     2     3     4     5

>> a./b
```


© Richard Greenaway 2011

```
ans =

    1.0000    0.5000    0.3333    0.2500    0.2000
```

3.5.3 Array Power

Finally, array power can be used to raise individual elements of a matrix to powers. The exponent can be a scalar in which case all elements of the matrix are raised to the same power or a matrix of the same size as the operand in which case each element of the matrix is raised to the power of the value in the corrrsponding element of the exponent matrix.

```
>> a = ones(1,4)*2

a =

     2     2     2     2

>> b = 1:4

b =

     1     2     3     4

>> a.^b

ans =

     2     4     8    16
```

Now try the following exercises.

Exercise 3-7

Consider the following function.

$$y(\theta) = \sin\left(\frac{\theta}{N}\right)\sin^2(\theta)$$
$$0 \le \theta \le N\pi$$

Generate values for the given function at intervals of $\frac{\pi}{10}$ between 0 and 12π and plot the resulting data following the steps below.

1. Generate a vector of theta values at the specified interval and over the specified range for *N=6*. (section 3.2.1.2)

2. Determine *y* for all values of theta.

   Since y is the product of two functions, $y = f(\theta)g(\theta)$. To calculate $y(\theta_i)$ we must evaluate $f(\theta_i)$ and $g(\theta_i)$ and then take the product and do this for each value of θ. Thus you use the array product. Similarly, you must use the array power operator to calculate $\sin^2\theta$

3. Now plot the function using the `plot` command. Type `help plot` to see how to use it.

© Richard Greenaway 2011

3.6 Relational Operations and Logical Vectors

Relational operators compare operands quantitatively, using operators like "less than" and "not equal to." The following table provides a summary.

<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
~=	Not equal to

Table 3-3

Logical operators return a logical *true* or *false* represented as 1 and 0 respectively as illustrated in the example below.

```
>> 3 > 6

ans =

     0

>> 3 < 6

ans =

     1
```

The answer generated is not the integer value 1 but the *logical* value 1. A value of type *logical* can only have two states, either **1** (*true*) or **0** (*false*). When applied to a matrix or vector a relational operator works on an element-by-element basis.

```
>> a = 1:5

a =

     1     2     3     4     5

>> b = ones(1,5)*3

b =

     3     3     3     3     3

>> a > b

ans =

     0     0     0     1     1
```

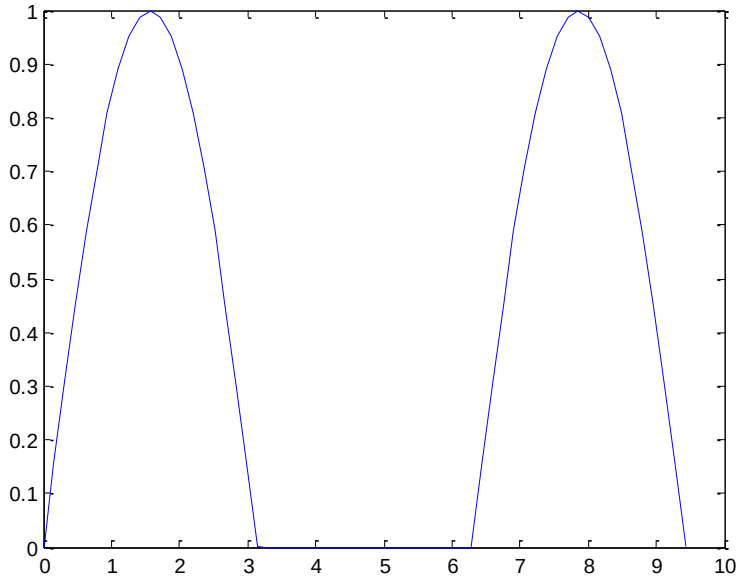

© Richard Greenaway 2011

That is, `a > b` is *false* for the first three elements and *true* for the last two.

What makes this type of operation so powerful is that logical vectors can be used to index elements in a matrix, thus we can find data within a data set (a vector or matrix) which conform to some relational test. An example should make this clear.

```
>> x = 0 : pi/20 : 3*pi;      % Generate theta values
>> y = sin(x);
>> neg = y < 0;     % find all values of sin(x) less than 0
>> y(neg) = 0;      % set all those values to 0
>> plot(x,y);
```

The result looks like this.



The operation `y<0` returns a logical vector with **1**s in elements corresponding to values in the `y` vector less than zero and **0**s in all the other elements. This vector is stored as the variable `neg`. When `neg` is used to index `y` (as in `y(neg)`), it indexes all elements in `y` less than zero, and so `y(neg)= 0` replaces all negative values with zero.

We can reduce the number of steps required to do the above exercise to 4 as follows.

Exercise 3-8

Carry out the following steps.

```
>> x = 0 : pi/20 : 3*pi;
>> y = sin(x);
>> y = y .* (y > 0);
>> plot(x,y)
```


© Richard Greenaway 2011

Can you see how that works? Do it yourself and look at the output from each step. You can break down the operations in any way you want. For instance, look at what y>0 does by typing it on its own and then do the array multiply .

Here is an exercise to try.

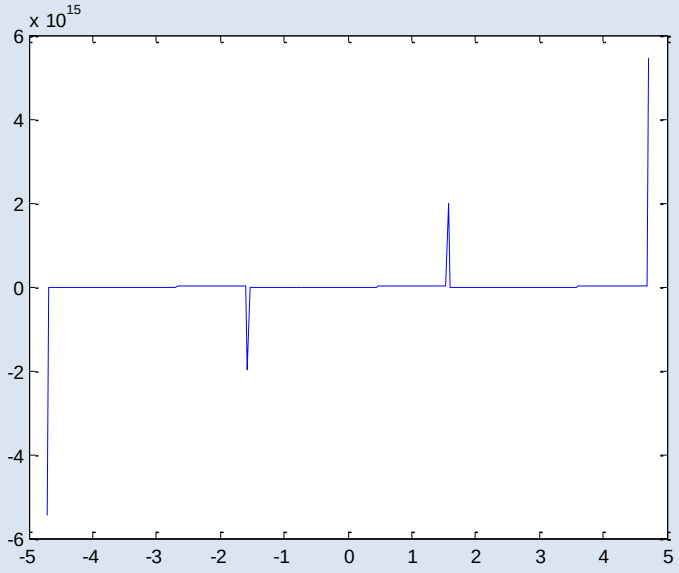
Exercise 3-9

1. Plot the graph of $\tan\theta$ between $-\frac{3}{2}\pi$ and $\frac{3}{2}\pi$.

   Define angles x at intervals of $\pi/100$.

```
theta  = -3/2*pi : pi/100 : 3/2*pi;
y = tan(theta);
plot(theta,y);
```

   If you've done it correctly you should get a plot that looks like this.



It doesn't look much like a tan θ curve unfortunately, why is that? Well as θ approaches odd-multiples of π/2, tan θ approaches ±∞. Look at the y-axis and you'll see that the peak values are huge. The structure of the curve is lost.
To solve this problem we need to locate the values of tan θ that are very large (negative and positive) and replace them with zeros.

2. Replace the values of y which are greater than some arbitrary large number (it's not crucial, but look at the plot above) by zero and then re-plot the graph.

   We can do this as follows. Locate all values less than our 'arbitrary' large number.

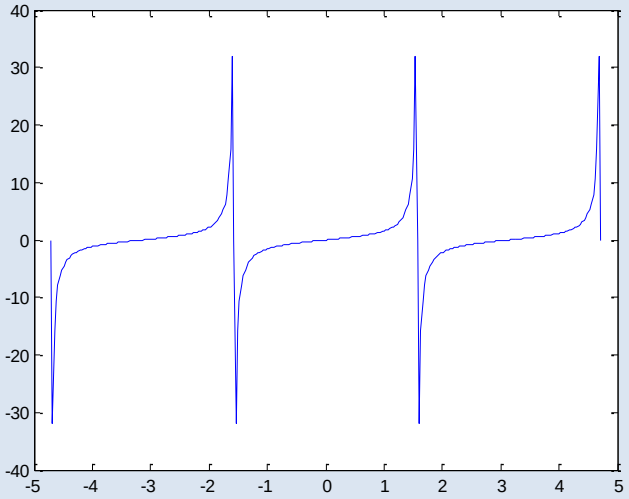
© Richard Greenaway 2011

```
result = (abs(y) < 1e10);
```

The vector `result` will have *ones* where the (absolute) value of `y` is less than 10^{10} and *zeros* elsewhere. So if we multiply y by this vector, that should get rid of the large values.

You don't have to split the task, you can remove the large values from `y` in a single line.

Try that now and then re-plot the data. If you get it right it should look like this.



© Richard Greenaway 2011

Exercise 3-10

We are going to plot a function known as the *sinc* function which has the following form.

$$y = \frac{\sin\theta}{\theta}$$

1. Define theta values between -3π/2 and 3π/2 in steps of π/10.

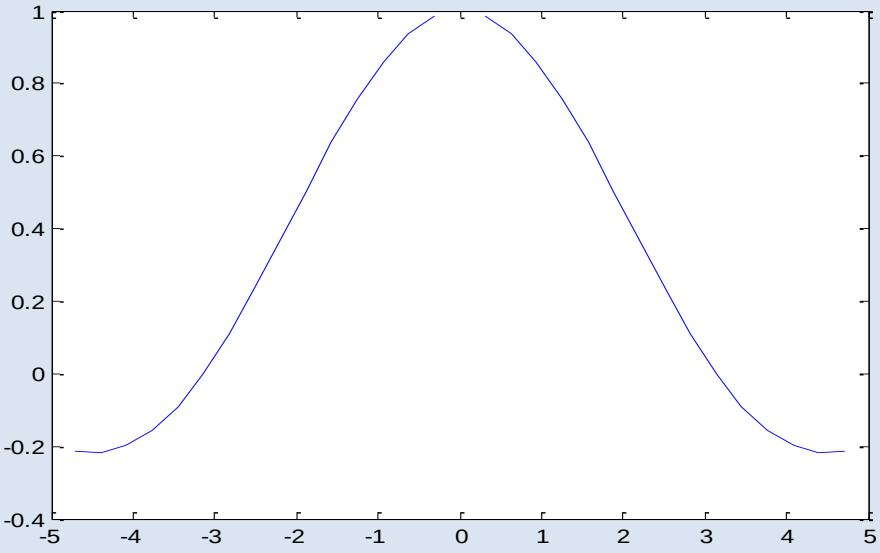
```
theta = -3/2*pi : pi/10 : 3/2*pi;
```

2. Now create the *sinc* function. Remember you have use array division.

```
y = sin(theta)./theta;
```

3. Now plot the result using the `plot` function.

```
plot(theta,y)
```



Notice that the curve has no value at *theta* = 0. That's because.

$$\frac{\sin 0}{0} = \frac{0}{0}$$

which, as far as MATLAB is concerned, is undefined[1].
MATLAB handles it very well, in such a situation it returns the special value **NaN** (*not-a-number*) which is ignored when plotting, hence the gap in the plot. We nevertheless have a missing value, so how can we deal in general with problems like this?

[1] It is evident from the curve of course that $\frac{\sin 0}{0}$ *is* defined which can be proven from L'Hôpital's rule which states that if $\lim_{x\to c} f(x) = \lim_{x\to c} g(x) = 0$ then $\lim_{x\to c} \frac{f(x)}{g(x)} = \lim_{x\to c} \frac{f'(x)}{g'(x)}$ and therefore $\left.\frac{\sin\theta}{\theta}\right|_{\theta=0} = \left.\frac{d\,(sin\theta)/d\theta}{d\theta/d\theta}\right|_{\theta=0} = 1$

© Richard Greenaway 2011

If we replace the angle 0 with a very small number (ie an approximation to zero) then our function will work. MATLAB provides such a number, **eps**. `eps` returns the 'distance' between contiguous floating point numbers as they are represented in MATLAB, the 'resolution' of floating point numbers if you like.

4. Type `eps` in the command window to see what value it is.

5. Now see what happens when you put this angle into the equation. Calculate, $\sin(eps)/eps$.

So now all we have to do is find θ = 0 in our vector of θ values and replace it with *eps* and recalculate the sinc function. Again, we can do it with logical vectors.

6. In the theta vector replace all occurrences of **0** with **eps**.

```
theta2 = theta + (theta == 0)*eps;
```

   Compare the values in `theta2` with those in `theta`. How does the line above work.

7. Recalculate the sinc function and re-plot it.

3.7 Operator Precedence

You can build expressions that use any combination of arithmetic, relational, and logical operators. Precedence levels determine the order in which MATLAB evaluates an expression. Within each precedence level, operators have equal precedence and are evaluated from left to right. The precedence rules for MATLAB operators are shown in this list, ordered from highest precedence level to lowest precedence level:

1. Parentheses `()`
2. Transpose `(.')`, power `(.^)`, complex conjugate transpose `(')`, matrix power `(^)`
3. Unary plus `(+)`, unary minus `(-)`, logical negation `(~)`
4. Multiplication `(.*)`, right division `(./)`, left division `(.\)`, matrix multiplication `(*)`, matrix right division `(/)`, matrix left division `(\)`
5. Addition `(+)`, subtraction `(-)`
6. Colon operator `(:)`
7. Less than `(<)`, less than or equal to `(<=)`, greater than `(>)`, greater than or equal to `(>=)`, equal to `(==)`, not equal to `(~=)`
8. Element-wise AND `(&)`
9. Element-wise OR `(|)`
10. Short-circuit AND `(&&)`
11. Short-circuit OR `(||)`

© Richard Greenaway 2011

3.8 Saving and Reading Data

3.8.1 Saving Data

Data in the MATLAB workspace can be saved to special files with the extension **mat**. Typing **save** on the command line will save all the data to the default file **matlab.mat**. Alternatively you can save selected variables to a file name of your choice.

The following command will save two matrices, `A` and `c` to the file **test.mat**.

```
>> save 'test' A c
```

When you save data to a file it will overwrite any existing file of the same name so you will lose whatever data had been previously saved. If you want to add data to an existing file then use the -append option as follows.

```
>> save 'test' B -append
```

The file **test.mat** will now contain 3 variables, `A`, `B` and `c`.

There are various other ways to export data from MATLAB to formats such as csv, excel files, text files etc.

3.8.2 Reading Data

Data previously saved to **mat** files can be read back in to the MATLAB workspace using the **load** function.

If the data was saved to the default **matlab.mat** file then simply type load and all the variables will be restored to the workspace. For other files you must specify the filename.

```
>> load 'test'
```


© Richard Greenaway 2011