Arrays: basics

Arrays are the fundamental data elements of Matlab.

An $m \times n$ array, also known as an $m \times n$ **matrix**, is a rectangular arrangement of values (called **entries** or **elements**) with m rows and n columns; "$m \times n$" is the **size** of the matrix.
A $1 \times n$ array is called a **row vector**; n is the **length** of the vector.
An $m \times 1$ array is a **column vector**; m is the length of the vector.
The entries of a vector are also called **components**.

It is natural for variables in Matlab to be arrays. Usually, vectors are given lowercase letters as variable names, while matrices with m and n both greater than 1 are given captial letters as variable names.
You can set up a matrix in Matlab by entering the values and enclosing them within square brackets. Use spaces or commas to separate entries in the same row, and use semicolons to separate rows.

```
A = [1 2 3 4; 5 6 7 8; 9 10 11 12]
size(A)
```

To extract a particular entry of the matrix, put its "coordinates" (row number, column number) in parentheses after the matrix name:

```
A(3,1)                          % entry in 3rd row, 1st column
A(2,4)                          % entry in 2nd row, 4th column
```

It is easy to assign a new value to a particular entry of the matrix:

```
A(1,4)=-29; A
```

Rows and columns of the matrix can be individually accessed by using the colon operator:

```
A(:,1)                          % selects the first column
A(2,:)                          % selects the second row
```

Vectors are entered similarly:

```
r = [8, -11, 15]                % row vector
c = [8; -11; 15]                % column vector
```

A column vector can also be defined by transposing a row vector:

```
c = [8 -11 15].'                % transpose (dot apostrophe)
A, A.'                          % view the effect on the matrix A
```

Another way of creating vectors is by using the colon operator (we have already used this in constructing for-loops):

```
x=1:7
y=3:10
z=2:3:17, length(z)
w1=-1:0.2:0
w2=-1:0.3:0
u=1.5:-0.3:-0.9
```

The above commands generate row vectors; to get column vectors, apply the transpose operator.

There is another command available for creating a vector with evenly spaced entries:
`linspace(a,b,n)` generates a row vector of n equally spaced points ranging from a to b inclusive; this means that the spacing between consecutive points is $(b-a)/(n-1)$, e.g.

```
v=linspace(-1,1,6)
```

To extract the kth component (entry) of a vector `v`, simply type `v(k)`, e.g.

```
v(3)
```

Just as with matrices, a new value can be assigned to any component of a vector:

```
c, c(3)=pi; c
```

In fact, you can augment a vector just by assigning a value to the additional entry:

```
c(4)=200; c
```

However, remember that every index of an array (row number, column number, or component number of a vector) must be a positive integer, so

```
c(0)=100
```

will return an error.

You can put column vectors side by side (provided they are all of the same length) to create a matrix:

```
a=[2:5].'
M=[a c]
```

Similarly, row vectors (of the same length) can be stacked on top of each other to create a matrix:

```
r
s=[7 6 4]
P=[r; s]
```

Array (elementwise) operations

The basic arithmetic operations and built-in elementary functions such as sqrt, exp, log, sin, cos, abs etc. can all be applied to arrays, **element by element**.

Let's define two matrices of the same size:

```
A=[3 -7; -9 2], B=[5 6; -1 4]
```

Addition or subtraction of a scalar:

```
A-2
```

Addition or subtraction of two arrays (which must be of the same size):

```
A+B
```

Multiplication by a scalar:

```
2*A
```

Elementwise multiplication of two arrays (which must be of the same size):

```
A.*B
```

Division by a scalar:

```
A/(-3)
```

Elementwise division of two arrays (which must be of the same size):

```
A./B, B./A
```

Exponentiation:

```
exp(A)
```

Trigonometric function:

```
cos(B)
```

Absolute value:

```
abs(A)
```

Square root:

```
sqrt(B)
```

Remember to include a **dot** before the usual multiplication, division and exponentiation symbols when the operations are meant to be performed **elementwise**.

Plotting *xy*-graphs

To plot the graph of cos(x) for x between -10 and 10, first set up an array of x-values:

```
x1=-10:0.1:10;
```

Next, calculate an array of corresponding y-values:

```
y1=cos(x1);
```

Then use the plot command:

```
plot(x1,y1)
```

With a different array of x-values, such as

```
x2=[-5 -2.5 sqrt(3) pi 4];
```

we get a different graph:

```
y2=cos(x2);
plot(x2,y2)
plot(x2,y2,'r*')
```

A graph of $y = x\cos(x)$ can be plotted as follows:

```
x3=linspace(-10,10,64);
y3=x3.*cos(x3);
plot(x3,y3)
```

Multiple graphs can be plotted on the same set of axes:

```
figure(2)
plot(x1,y1,x2,y2,'r*',x3,y3)
```

More complicated functions are plotted in the same way. For example:

```
x=linspace(-2,3,100);
y=sin(5*x).*exp(-x/2)./(1+x.^2);
figure(3), plot(x,y)
```

Do you see why we had to use the "dot operations" `.*`, `./` and `.^` in certain places?

The plot command has many options; for instance, you can specify line color, line style (solid, dashed, dotted etc.), the type of marker for data points (crosses, circles, asterisks etc.) and so on. You can also define the x and y ranges for the plot (using `axis`), label the axes (using `xlabel`, `ylabel`), add a title to the figure (using `title`) or, if you're plotting several graphs in the same figure, insert a legend (a key to the different graphs, using `legend`). Look up `plot` and its associated commands (under "See Also") in Matlab's Help to learn more about these options.

Vectorization

Bascially, this means exploiting Matlab's array structures to write (hopefully simpler) programs.
For example, our script for computing Fibonacci numbers can be rewritten so that it uses an array to save the sequence of numbers computed.