Chapter 6

A bit more C

6.1 Vectors & Matrices in C

An N-component vector is represented by a 1-dimensional array with N entries.

Example 3-d vector $\mathbf{x} = (x_1, x_2, x_3)$
is represeneted by

```
float x[3];
```

or

```
#define N 3
.
.
.

main()
{
   float x[N];
}
```


Then the elements of the array are

$$\begin{array}{ccc} \mathrm{x}[0] & \leftrightarrow & x_1 \\ \mathrm{x}[1] & \leftrightarrow & x_2 \\ \mathrm{x}[2] & \leftrightarrow & x_3 \end{array}$$

An N× M matrix is represented by a 2-D array.

Example if $\mathbf{x} = \begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \end{pmatrix}$
is represeneted by

```
float x[2][3];
```

or

```
#define N 3
#define M 3
.
.
.

main()
{
   float x[N][M];
   .
}
```

Then the elements of the array are

$$\begin{array}{ccc} \mathrm{x}[0][0] & \leftrightarrow & x_{11} \\ \mathrm{x}[0][1] & \leftrightarrow & x_{12} \\ \mathrm{x}[0][2] & \leftrightarrow & x_{13} \\ \mathrm{x}[1][0] & \leftrightarrow & x_{21} \\ \mathrm{x}[1][1] & \leftrightarrow & x_{22} \\ \mathrm{x}[1][2] & \leftrightarrow & x_{23} \end{array}$$

The elements of vectors and matrices (ie arrays) can be naturally accessed by for-loops

6.2 Vector Multiplication (and the dot product)

The elements of a vector can be easily accessed with a for-loop.

Example $\mathbf{x} = (x_1, x_2, x_3)$ $\mathbf{y} = (y_1, y_2, y_3)$
$x.y = \mathbf{x} = x_1y_1 + x_2y_2 + x_3y_3 = z$ in C:

```
main()
{
    float x[3],y[3],z;
    int i;
    z=0.0;
    for(i=0;i<3;i++)
    {
    z=z+x[i]*y[i];
    }
}
```

	i=0:	z=0+x[0]*y[0]
trace what happens:	i=1:	z=z+x[1]*y[1]
	i=2:	z=z+x[2]*y[2]

ie for-loop saves you lots of writing especially if say x and y are large.

6.3 Matrix Addition

Say we want to add A[2][2], B[2][2]

Example

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

$$C = A + B = \begin{pmatrix} a_{11} + b_{11} & a_{12} + b_{12} \\ a_{21} + b_{21} & a_{22} + b_{22} \end{pmatrix},$$

in C code we need to access the corresponding elements of A and B and add them. A, B matrices $\Rightarrow$ rows & columns $\Rightarrow$ 2 indices to identify any element.
So in C:

```
main()
{
    float A[2][2],B[2][2],C[2][2];
    int i,j;
    for(i=0;i<2;i++)
    {
        for(j=0;j<2;j++)
        {
        C[i][j]=A[i][j]+B[i][j];
        }
    }
}
```

trace what happens: EXAMPLE

6.4 Passing 1-D arrays to functions

Re-write p03.c with the dot and cross products done in functions.

Dot product:	2 vectors	1 number out
	$\Downarrow$	$\Downarrow$
	2 arguments to the function	function type- double return a number to main

there for we have the function

```
double dotproduct(double A[], double B[]);
```

The [] tells the compiler to expect the inputs to be 1-D arrays.
-see hand out for rest of function.
Cross product: 2 vectors 1 vector out
This is not like the usual functions we have seen where 1 number is returned.
Therefore we cannot use the return statement instead the C syntax is:

```
void crossproduct(double C[],double A[], double B[]);
```

Note from handout... both dotprod and cross prod are called from main kie

```
dotprod=dotproduct(x,y);

crossproduct(z,x,y);
```

ie the inputs are the array names (pointers)
We are really passing the address in memory of each vector.

$$\begin{array}{ccc} x & \leftrightarrow & \&x[0] \\ y & \leftrightarrow & \&y[0] \\ z & \leftrightarrow & \&z[0] \end{array}$$

program will work with

```
dotprod=dotproduct(\&x[0],\&y[0]);
```

or

```
dotprod=dotproduct(x,y);
```


6.5 Functions and Arrays of Dim > 1

An array:

```
int a[3][5];
```

has 2 dimensions (corresponds to a matrix with rows and columns).
or

```
int a[3][1][5];
```

is a 3-dimensional array with 3*1*5 entries.

Passing a 2-dim (or 3-dim ...) array to a function is a little more complicated than the 1 dim case.
Because:
The array name by itself eg a is equivalent to &a[0]
but now we eg

```
int a[3][5];
```

&a[0] is a pointer to an array of 4 integers. ie
a[0][0], a[0][1], a[0][2], a[0][3], a[0][4]. So in this case the base of the array is more correctly given by
&a[0][0] and not just a.
Therefore to pass a multidimensional array using just its name in the main progeam, the function must know the size of all other "columns"
SEE MATRIX ADDITION HANDOUT.

6.6 Matrix Multiplication

This is a bit more complicated then addition as we need to use a third for-loop.

Example

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

$$C = AB = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix},$$

in C:

```
main()
{
    float a[2][2],b[2][2],c[2][2];
    int i,j,k;
    for(i=0;i<2;i++)
    {
         for(j=0;j<2;j++)
         {
            c[i][j]=0;
            for(k=0;k<2;k++)
            {
                  c[i][j]=c[i][j]+a[i][k]*b[k][j];
            }
         }
    }
}
```