AMTH250

Notes on Octave

Gary Bunting

August 22, 2011

Contents

1 Matrices

Most data in Octave is represented as a matrix or, equivalently, a 2 dimensional array of data. This equivalence between arrays of data and matrices is one of great strengths of Octave, but it can also lead to some confusion.

For Octave vectors are just matrices with one row or one column. When working with Octave it is important to distinguish between a row vector

$$\mathbf{a} = \begin{bmatrix} x_1, x_2, \ldots, x_n \end{bmatrix}$$

and the equivalent column vector

$$\mathbf{b} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

because, as matrices, they have different shapes. Since vectors are just a special type of matrix, everything we say about matrices will be taken to include vectors as well.

Often the matrices occurring in Octave are being used simply as tables of data rather than mathematical matrices. This is important, since although some operations, e.g. addition and subtraction, are the same for matrices and data, other operations, e.g. multiplication, are different for matrices and arrays of data. This is the aspect of Octave that many beginners find most difficult.

For two matrices, or equivalently arrays of data, `A` and `B`, the two multiplication operators are matrix multiplication `A*B`, and element-by-element multiplication `A.*B`. Because we are more often using matrices, and especially vectors, as arrays of data, we almost always want to use the dot operator, `A.*B`. Matrix multiplication `A*B`, should only be used when we are working with `A` and `B` as mathematical matrices or vectors.

1.1 Entering Matrices

There are a number of ways matrices can be entered into Octave:

1. As an explicit list of elements.
2. Generated by built-in commands.
3. Created by user defined functions.
4. Loaded as data from external files.

The easiest way to enter small matrices is as explicit lists. The following rules apply:

- Elements of the matrix are separated by spaces or commas.
- Rows of the matrix are separated by semicolons or new lines.
- Start and end the matrix with square brackets, `[` and `]`.

```
octave:> a = [1 2 3; 4 5 6; 7 8 9]
a =
   1   2   3
   4   5   6
   7   8   9
```

The same result could be obtained with:

```
octave:> a = [1 2 3
> 4 5 6
> 7 8 9]
a =
   1   2   3
   4   5   6
   7   8   9
```

1.2 Indexing

The element in row `i` and column `j` of the matrix `a` is is denoted by `a(i,j)`.

```
octave:> a(1,3)
ans =  3
octave:> a(3,1)
ans =  7
```

Indexing can also be used to change elements of a matrix:

```
octave:14> a(2,3) = 100
a =
     1     2     3
     4     5   100
     7     8     9
```

For vectors, of either row or column type, only a single index is needed.

```
octave:15> v = [1 2 3 4]
v =
   1   2   3   4
octave:> v(3)
ans =  3
octave:> v(3) = 30
v =
    1    2   30    4
```

1.3 The Colon Operator

The colon operator : has many uses in constructing and deconstructing vectors and matrices.

The simplest use of the colon operator is to obtain a vector containing all the numbers in some range:

```
octave:> n = 1:10
n =
    1    2    3    4    5    6    7    8    9   10
octave:> n = 10:-2:1
n =
   10    8    6    4    2
```

The colon operator is also be used to pick out selected pieces of a matrix. This is important when matrices are used to store data and we want to extract certain parts of the data.

- `a(i,:)` is the `i`th row of `a`.
- `a(:,j)` is the `j`th column of `a`.
- `a(:,j:k)` is the matrix formed from the `j`th to `k`th columns of `a`, etc.

```
octave:> a = [16 3 2 13; 5 10 11 8; 9 6 7 12; 4 15 14 1]
a =
   16    3    2   13
    5   10   11    8
    9    6    7   12
    4   15   14    1

octave:> a(2,:)                  # the second row
ans =
    5   10   11    8
octave:> a(:,3)                  # the third column
ans =
    2
   11
    7
   14
octave:> a(:,2:3)                # the second to third columns
ans =
    3    2
   10   11
    6    7
   15   14
octave:> a(1:2,3:4)        # the first to second rows and‘
ans =a                         # third to fourth columns
    2   13
   11    8
octave:> a(1,:) = [1 0 0 0]    # set the first row
a =
    1    0    0    0
    5   10   11    8
    9    6    7   12
    4   15   14    1
octave:> a(:,2) = []          # delete the second column
a =

    1    0    0
    5   11    8
    9    7   12
    4   14    1
```

1.4 Concatenation

In expressions such as

```
     a = [1 2; 3 4]
```

the numbers can be replaced by matrices allowing a matrix to be built up from sub-matrices.

The following is an example of a common and useful construction:

```
octave:> x = (0:0.5:3)’
x =
   0.00000
   0.50000
   1.00000
   1.50000
   2.00000
   2.50000
   3.00000
octave:> y = [sin(x) cos(x) tan(x)]
y =
    0.00000    1.00000    0.00000
    0.47943    0.87758    0.54630
    0.84147    0.54030    1.55741
    0.99749    0.07074   14.10142
    0.90930   -0.41615   -2.18504
    0.59847   -0.80114   -0.74702
    0.14112   -0.98999   -0.14255
```

In this example x is a column vector of length 7. Then sin(x), cos(x) and tan(x) are column vectors of the same size. The matrix resulting from concatenating the three column vectors is then a 7×3 matrix whose columns are vectors sin(x), cos(x) and tan(x).

1.5 Special Matrices

The following functions are used to construct simple matrices.

zeros	matrix of zeros
ones	matrix of ones
eye	identity matrix
rand	random matrix – uniform(0,1)
randn	random matrix – normal(0,1)
linspace	uniformly spaced vector
logspace	logarithmically spaced vector

```
octave:> rand(3,4)
ans =
   0.097903   0.211107   0.105722   0.912562
   0.945108   0.140032   0.617902   0.532232
   0.425861   0.523031   0.680037   0.077349
octave:> linspace(1,2,5)
ans =
   1.0000   1.2500   1.5000   1.7500   2.0000
```

1.6 Size of a Matrix

The function size returns the size of a matrix, the functions rows and columns give the number of rows and columns, and the function numel gives the total number of elements in the matrix.

```
octave:>  a = randn(3,10);
octave:> size(a)
ans =
    3   10
octave:> numel(a)
ans =  30
```

1.7 Data Analysis

Functions for Data Analysis

`max`	largest element
`min`	smallest element
`mean`	mean value
`median`	median value
`std`	standard deviation
`var`	variance
`sort`	sort in ascending order
`sum`	sum of elements
`prod`	product of elements
`cumsum`	cumulative sum of elements
`cumprod`	cumulative product of elements
`diff`	difference of successive elements

The functions are mostly applied to vectors.

```
octave:> x = [-3 8 -2 0 1]
x =
  -3   8  -2   0   1
octave:> [max(x) min(x)]
ans =
   8  -3
octave:> sum(x)
ans =  4
octave:> mean(x)
ans =  0.80000
octave:17> diff(x)
ans =
   11  -10    2    1
octave:> sort(x)
ans =
  -3  -2   0   1   8
```

To sort in descending order:

```
octave:12> sort(x,'descend')
ans =
   8   1   0  -2  -3
```

Exercise: How do these function deal with `NaN`s?

2 Matrix Algebra

2.1 Transpose

The quote `'` is used to take the transpose of a matrix.

```
octave:> a = [1 2 3; 4 5 6; 7 8 9]
a =
   1   2   3
   4   5   6
   7   8   9
octave:> a'
ans =
   1   4   7
   2   5   8
   3   6   9
```

The distinction between row and column vectors is often important in Octave and the transpose can be used to convert between one and the other:

```
octave:> a = 1:4
a =
   1   2   3   4
octave:> b = a'
b =
   1
   2
   3
   4
octave:> c = b'
c =
   1   2   3   4
```

For complex numbers `z'` is the complex conjugate of `z` and for complex matrices `a'` is the conjugate transpose of `a`.

```
octave:> a = [ 1+2i, 3-4i; i, 1-2i]
a =
   1 + 2i   3 - 4i
   0 + 1i   1 - 2i
octave:> a'
ans =
   1 - 2i   0 - 1i
   3 + 4i   1 + 2i
```

2.2 Adding and Subtracting Matrices

These work as expected:

```
octave:> a = [1 2 3; 4 5 6; 7 8 9]
a =
   1   2   3
   4   5   6
   7   8   9
octave:> a+a
ans =
    2    4    6
    8   10   12
   14   16   18
octave:> a-a
ans =
   0   0   0
   0   0   0
   0   0   0
```

Adding or subtracting a matrix and number results in the number being added to subtracted from *all* the elements of the matrix:

```
octave:> a-10
ans =
  -9  -8  -7
  -6  -5  -4
  -3  -2  -1
```

2.3 Multiplication of Matrices

Recall that if **A** is a $m_1 \times n_1$ matrix and **B** is a $m_2 \times n_2$ matrix then the matrix product **AB** is defined only if $m_2 = n_1$ and the result is a $m_1 \times n_2$ matrix. Row vectors are $1\times n$ matrices and column vectors are $n\times 1$ matrices.

```
octave:> a = [1 2 3; 4 5 6; 7 8 9]
a =
   1   2   3
   4   5   6
   7   8   9
octave:> b = ones(3,1)
b =
   1
   1
   1
```

```
octave:> a*b
ans =
    6
   15
   24
octave:> b*a
error: operator *: nonconformant arguments (op1 is 3x1, op2 is 3x3)
octave:> b'*a
ans =
   12   15   18
```

Be careful to note the difference between the inner (dot) product and outer product of vectors:

```
octave:> x = [1 2 3]
x =
   1   2   3
octave:> y = [-1; 0; 1]
y =
  -1
   0
   1
octave:> x*y
ans =  2
octave:> y*x
ans =
  -1  -2  -3
   0   0   0
   1   2   3
```

2.4 Powers of Matrices

Matrix powers are only defined for *square* matrices.

```
octave:> a = [1 2 3; 4 5 6; 7 8 9];
octave:> a^3
ans =
    468    576    684
   1062   1305   1548
   1656   2034   2412
```

2.5 The Dot Operator

The dot operator `.` is used in conjunction with the operators `*`, `/` and `^` to perform element by element multiplication, division and powers on vectors and matrices.

```
octave:> v = 1:4
v =
   1   2   3   4
octave:> v.*v
ans =
    1    4    9   16
octave:> v.^3
ans =
    1    8   27   64
octave:> 3.^v
ans =
    3    9   27   81

octave:> a = [1 2 3; 4 5 6; 7 8 9]
a =
   1   2   3
   4   5   6
   7   8   9
octave:> a.*a
ans =
    1    4    9
   16   25   36
   49   64   81
octave:> a.^3
ans =
     1     8    27
    64   125   216
   343   512   729
octave:> a./a
ans =
   1   1   1
   1   1   1
   1   1   1
```

Note in particular:

1. `a*b` is the *matrix product* of `a` with `b`; `a.*b` is the matrix whose components are are formed by multiplying corresponding components of `a` and `b`.

2. `a^3` is the cube of matrix `a`, `a.^3` is the matrix whose components are the cubes of the components of `a`.

3. `a./b` divides each element of `a` by the corresponding element of `b`.

2.6 Mathematical Functions

Octave has all of the common mathematical functions.

Elementary Functions

`sqrt`	square root
`exp`	exponential
`log`	logarithm to base e
`log10`	logarithm to base 10
`log2`	logarithm to base 2
`sin`	sine
`cos`	cosine
`tan`	tangent
`asin`	inverse sine
`acos`	inverse cosine
`atan`	inverse tangent

Like the dot operators, these functions act element by element on vectors and matrices.

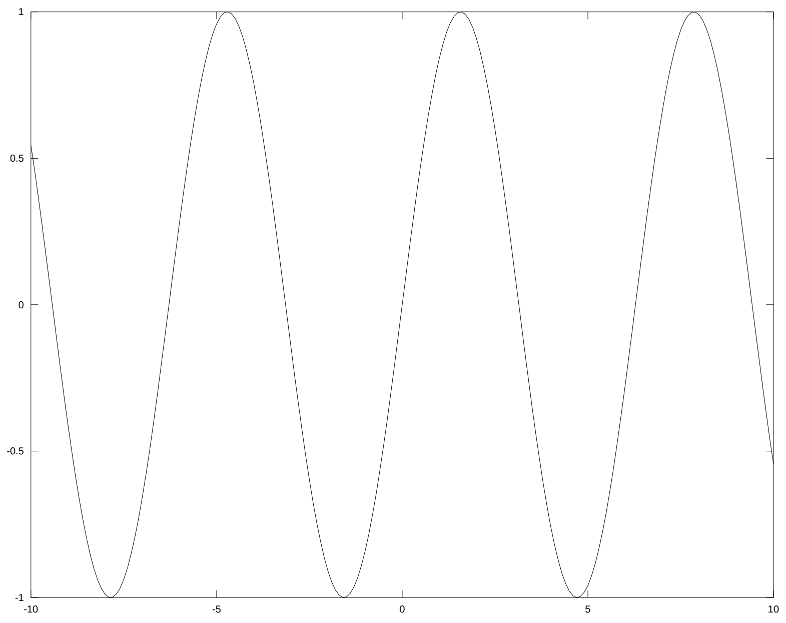
```
octave:> a = [1 2 3; 4 5 6; 7 8 9]
a =
   1   2   3
   4   5   6
   7   8   9
octave:> sqrt(a)
ans =

   1.0000   1.4142   1.7321
   2.0000   2.2361   2.4495
   2.6458   2.8284   3.0000
```

3 Graphs

3.1 Simple Graphs

The simplest graph takes two vectors and plots one against the other:

```
octave:> x = (-10:0.1:10)';
octave:> y = sin(x);
octave:> plot(x,y)
```

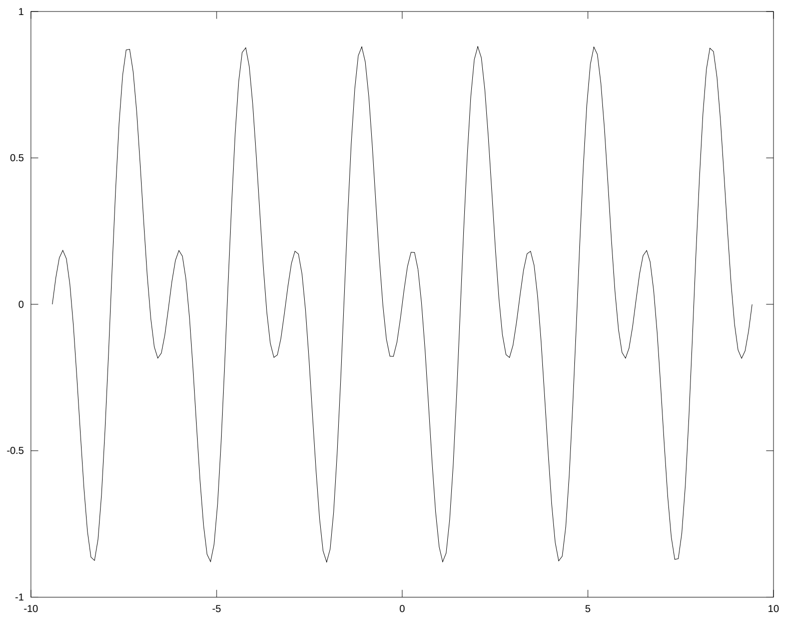


Octave graphs joins points by straight lines which sometimes gives the graph a slight polygonal look. If you want a smooth looking graph you need to take a fairly dense of points, 1000 will usually do, for the x-coordinates.

Graphs and Dot Operators

Plotting graphs is another place where you usually want to use a dot operator (see notes to Assignment 4). For example to graph $y = \sin x \cos 3x$:

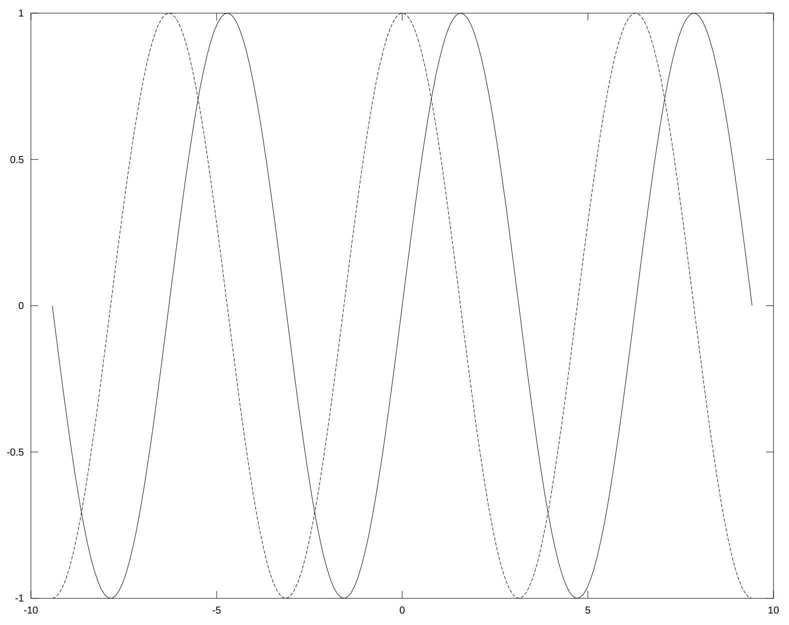
```
octave:> x = linspace(-3*pi, 3*pi, 200);
octave:> y =sin(x).*cos(3*x);
octave:> plot(x,y)
```



3.2 Multiple Curves

We can plot multiple curves, one on top of the other, by the command `hold on` and then plotting them successively. If in the command `plot2d(x, y)` `y` is a matrix then, assuming `x` column vector, each of the *columns* of the matrix is plotted as a separate curve.

```
octave:> x = linspace(-3*pi, 3*pi, 200)';
octave:> plot(x, [sin(x) cos(x)])
```

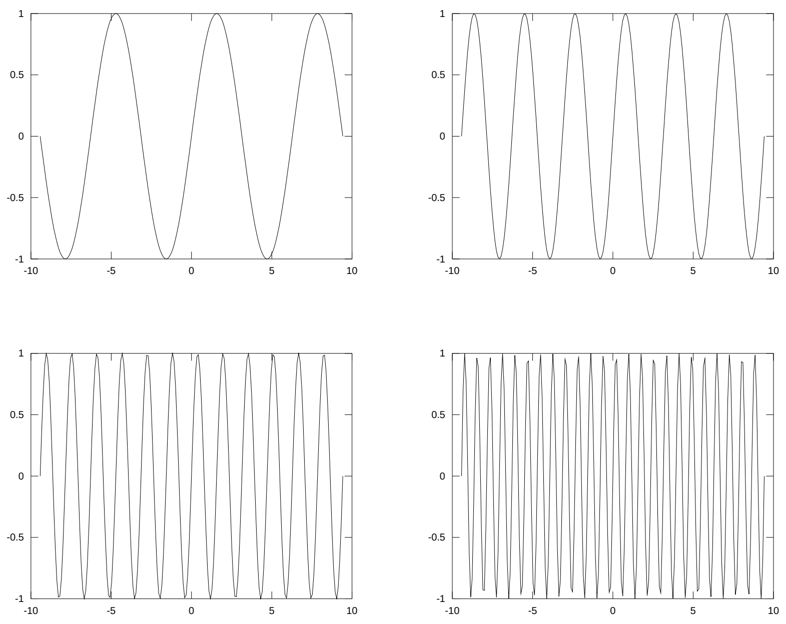


3.3 Multiple Plots

Multiple graphs can included in one figure using the `subplot` command.

```
octave:> x = linspace(-3*pi, 3*pi, 200);
octave:> subplot(2,2,1)
octave:> plot(x, sin(x))
octave:> subplot(2,2,2)
octave:> plot(x, sin(2*x))
octave:> subplot(2,2,3)
octave:> plot(x, sin(4*x))
octave:> subplot(2,2,4)
octave:> plot(x, sin(8*x))
octave:> subplot(1,1,1)
```

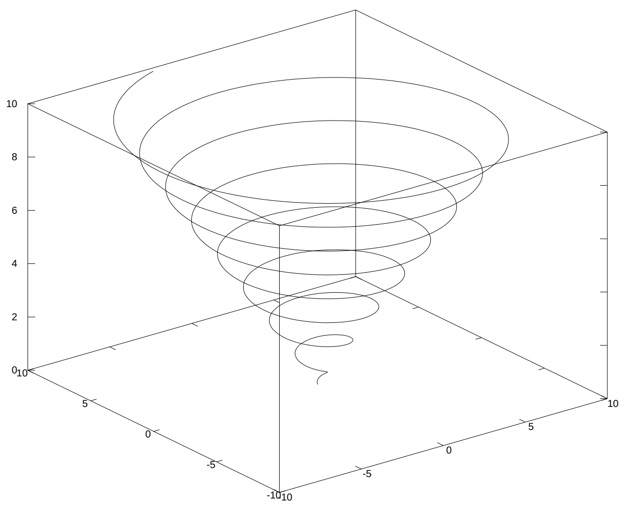
The last command resets to a single plot.



3.4 3D Curves

Curves in 3 dimensional space can be plotted using `plot3`. It takes three vectors containing the values the x, y and z coordinates of the points on the curve. By holding down the left mouse key and moving the mouse you can alter the orientation of the graph.

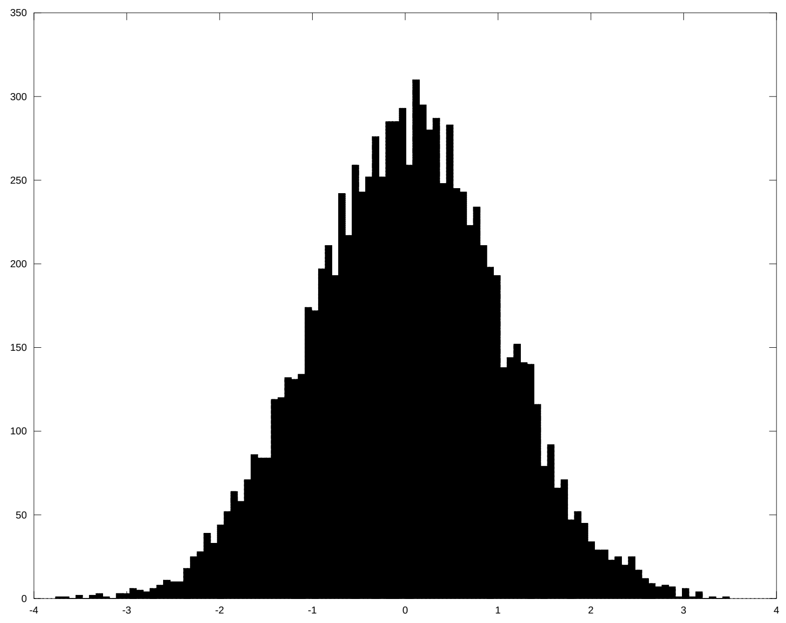
```
octave:> z = (0:.01:10);
octave:> plot3(z.*sin(5*z), z.*cos(5*z), z)
```



3.5 Histograms

Histograms can be plotted with the `histplot(data, n)` command. Here `n` is the number of bins in the histogram and `data` is the vector of data for which we want to draw the histogram. You usually need to experiment with number of bins. The following example draws a histogram of a vector of normally distributed random numbers.

```
octave:> rr = randn(1, 10000);
octave:> hist(rr,100)
```



4 Programming in Octave

4.1 `FOR` loops

In Octave `for` loops are used to iterate over a set of values. Here is a simple example of a `for` loop which should be easy to understand:

```
octave:> v = zeros(1,10);
octave:> for i = 1:10
> v(i) = i;
> end
octave:> v
v =
    1     2     3     4     5     6     7     8     9    10
```

Two points:

1. The vector `v` was initialized to vector of zeros before performing the loop. This is not strictly necessary, but is good programming practice. If `v` wasn't initialized, then on each pass through the loop the size of `v` would have to increase. `for` loops can be pretty slow and continually having to reallocate memory could slow things down even more.

2. `for` loops are one place you almost always want to terminate statements with semicolons; otherwise the a result would be printed on every pass through the loop.

Here is an another example producing the 5 by 5 identity matrix:

```
octave:> ident = zeros(5, 5);
octave:> for i = 1:5
> ident(i,i) = 1;
> end
octave:> ident
ident =

   1   0   0   0   0
   0   1   0   0   0
   0   0   1   0   0
   0   0   0   1   0
   0   0   0   0   1
```

It is possible to have `for` loops within `for` loops. These are called *nested loops* and are useful in constructing matrices conforming to some pattern.

The **Hilbert matrix** is the $n \times n$ matrix

$$\mathbf{H} = \begin{bmatrix} 1 & \frac{1}{2} & \cdots & \frac{1}{n} \\ \frac{1}{2} & \frac{1}{3} & \cdots & \frac{1}{n+1} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{1}{n} & \frac{1}{n+1} & \cdots & \frac{1}{2n-1} \end{bmatrix}$$

Here is how to produce the 4×4 Hilbert matrix:

```
octave:> for i = 1:4
> for j = 1:4
>   h(i,j) = 1/(i+j-1);
> end
> end
octave:> h
h =
   1.00000   0.50000   0.33333   0.25000
   0.50000   0.33333   0.25000   0.20000
   0.33333   0.25000   0.20000   0.16667
   0.25000   0.20000   0.16667   0.14286
```

Note in particular that each `for` statement is matched with an `end` statement. Indenting `for` loops makes it clear which `end` statement matches which `for` statement.

4.2 Functions

To produce the Hilbert matrix $\mathbf{H}_n$ for *any* value of n it is best to define a `function` to perform the task:

```
octave:> function h = hilbert(n)
> for i = 1:n
>   for j = 1:n
>     h(i,j) = 1/(i+j-1);
>   end
> end
> endfunction
octave:> hilbert(3)
ans =

   1.00000   0.50000   0.33333
   0.50000   0.33333   0.25000
   0.33333   0.25000   0.20000
```

In this example

1. `hilbert` is the name of the function.
2. `n` is the argument to the function. Functions can any number of arguments.
3. `h` is the value returned by the function. The actual value returned is the value of `h` immediately before the `endfunction` statement is reached.

Functions are another place you almost always want to terminate statements with semicolons; otherwise all intermediate steps in the calculation would be printed each time the function is called.

Here is another example, the factorial function:

$$n! = 1 \times 2 \times \ldots \times n$$

```
octave:> function fact = factorial(n)
> fact = 1;
> for k = 1:n
>   fact = k*fact;
> end
> endfunction
octave:> factorial(5)
ans =  120
octave:> factorial(100)
ans =  9.3326e+157
```

Functions and Dot Operators

We have seen that built-in functions like `sin` can take vectors or matrices as arguments and then act element-by-element on that argument. We usually want the same thing to happen when we define mathematical functions in Octave, for example, when we want to graph the function. This requires careful attention to the use of *dot* operators.

Consider, for example, the function

$$f(x) = \sin x \cos x$$

Since Octave functions like `sin` and `cos` act element by element, we would expect the same of their product. Thus we write their product as

```
  sin(x).*cos(x)
```

Here is how we would write it as a Octave function:

```
octave:> function y = f(x)
> y = sin(x).*cos(x);
> endfunction
```

4.3 Returning Multiple Values

The mechanism for writing functions returning multiple values in Octave is quite simple. The following example should make it clear:

```
octave:> function [y1,y2] = ff(x1, x2)
> y1 = x1 + x2;
> y2 = x1 - x2;
> endfunction
octave:> ff(10, 20)
ans =  30
octave:> [y1,y2]=ff(10, 20)
y1 =  30
y2 = -10
```

Unless you assign two values the result, only the first value is returned.

4.4 Comparison and Logical Operators

The comparison operators are used to compare values:

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

These result in the values 0 or 1, 0 for false, 1 for true:

```
octave:> 13 >= 28
ans = 0
```

They are typically used in `while` and `if` statements (see below).

By combining these with the logical operators:

`&`	and
`|`	or
`~`	not

more complicated conditions can be expressed

```
octave:> (1 >= 3) | (6 < 7)
ans =  1
```

Note that “or” is used in inclusive sense — “a or b” means a is true or b is true or both are true.

4.5 `WHILE` Loops

The main use of `for` loops is to repeat a series of statements a fixed number of times. In contrast `while` loops repeat a series of statements until a given condition is satisfied.

The following example illustrates how to find $\varepsilon_{\text{mach}}$ without knowing the precision of arithmetic we are using. Recall that $\varepsilon_{\text{mach}}$ is the smallest floating point number such that

$$1 + \varepsilon_{\text{mach}} \neq 1$$

Start with $\texttt{eps} = 1$ and repeatedly halve it until $1 + \texttt{eps} = 1$. Then the value of `eps` we finish up with is twice $\varepsilon_{\text{mach}}$, since the previous value `eps` must have been the last value which satisfied $1 + \texttt{eps} \neq 1$. (We are assuming here that we have a binary computer.)

```
octave:> eps = 1;
octave:> while (1 + eps ~= 1)
> eps = eps/2;
> end
octave:> 2*eps
ans =  2.2204e-16
```

4.6 `IF` Statements

`if` statements allow us to perform alternative actions depending on the result of a test. The general form of the `if` statement is

```
if (test1)
  statements
elseif (test2)
  statements
.
.
else
  statements
end
```

You can have any number of `elseif` clauses. On the other other hand, you don't have to have an `elseif` clause nor, indeed, an `else` clause.

Here is a function which returns the sign of a number:

```
octave:> function s = signum(x)
> if (x > 0)
>   s = 1;
> elseif (x < 0)
>   s = -1;
> else
>   s = 0;
> end
> endfunction
octave:> signum(-12345)
ans = -1
```

5 Working with Files in Octave

To work with external files in Octave, both you and Octave need to know where these files are. This is particularly important when creating and editing script and function files (see below).

The `pwd` command tells you the folder which Octave is working from, for example

```
octave:> pwd
ans = /home/gbunting/amth250/Octave
```

If you are familiar with working with paths, then you can control where Octave looks for files. The `path` command gives the path uses to look for files.

5.1 Function Files

Functions can be typed directly into Octave as in the examples in §4.2. Function files contain function definitions like the functions `hilbert` and `factorial` in §4.2. It is common to put functions in files rather than enter them directly into Octave since (a) they are then saved away for further use, and (b) it is easy to correct or modify a function by editing the file.

Create a file, `hilbert.m` (it is mandatory to end function files with the suffix `.m`), containing the function above:

```
function h = hilbert(n)
% The  n x n Hilbert matrix
  h = zeros(n,n);
  for i = 1:n
    for j = 1:n
      h(i,j) = 1/(i + j - 1);
    end
  end
endfunction
```

Once this done the function `hilbert` is available to Octave provided, of course, that the function file is in a place that Octave will look.

```
octave:> hilbert(6)
ans =
   1.000000   0.500000   0.333333   0.250000   0.200000   0.166667
   0.500000   0.333333   0.250000   0.200000   0.166667   0.142857
   0.333333   0.250000   0.200000   0.166667   0.142857   0.125000
   0.250000   0.200000   0.166667   0.142857   0.125000   0.111111
   0.200000   0.166667   0.142857   0.125000   0.111111   0.100000
   0.166667   0.142857   0.125000   0.111111   0.100000   0.090909
```

5.2 Script Files

Script files are *plain text* files that they contain any sort of Octave commands. There are two main uses for script files:

1. To repeat a series of commands, often to perform a numerical experiment.
2. To enter largish problems into Octave. Here the use of script files allows us to correct errors by editing a file rather than retyping the whole problem into Octave.

A script file, called say `plotsin.m`, to graph $\sin x$ might contain

```
% plot sin(x) over the interval[-10,10]
x = linspace(-10, 10, 200);
y = x.*sin(x);
plot(x,y)
```

Then, at the Octave prompt, the command

```
octave:> plotsin
```

will execute the commands in the script and plot the graph. The suffix `.m` is mandatory if the script file is executed by simply typing its name. Files with the suffix `.m` can be either function files or script files – the generic name is *m-files.*

5.3 Collections of Functions

One limitation of function files is that they can only contain a single function. When you want to make a collection of functions available to Octave the easiest approach is to put them in a file, called say `myfuncs.oct` (don't use the suffix `.m`), which may look like

```
function y = func1(...)
...
endfunction

function y = func2(...)
...
endfunction
   .
   .
   .
```

Then the command

```
octave:> source('myfuncs.oct')
```

will load the functions into Octave.

5.4 Exporting Data

The `save` command is used to write Octave data to external files, which can then be used by Octave or other programs. Here is an example:

```
octave:> x = (0:0.5:3)';
octave:> y =[x sin(x) cos(x)]
y =
   0.00000   0.00000   1.00000
   0.50000   0.47943   0.87758
   1.00000   0.84147   0.54030
   1.50000   0.99749   0.07074
   2.00000   0.90930  -0.41615
   2.50000   0.59847  -0.80114
   3.00000   0.14112  -0.98999

octave:> save mydata y
octave:> save -ascii mydata.txt y
```

The file `mydata` looks like:

```
# Created by Octave 3.2.3, Wed Aug 17 22:32:33 2011 EST <gbunting@gbunting-laptop>
# name: y
# type: matrix
# rows: 7
# columns: 3
 0 0 1
 0.5 0.479425538604203 0.8775825618903728
 1 0.8414709848078965 0.5403023058681398
 1.5 0.9974949866040544 0.07073720166770291
 2 0.9092974268256817 -0.4161468365471424
 2.5 0.5984721441039565 -0.8011436155469337
 3 0.1411200080598672 -0.9899924966004454
```

This is form can read back into Octave using the `load` command.

The file `mydata` looks like

```
 0.00000000e+00 0.00000000e+00 1.00000000e+00
 5.00000000e-01 4.79425539e-01 8.77582562e-01
 1.00000000e+00 8.41470985e-01 5.40302306e-01
 1.50000000e+00 9.97494987e-01 7.07372017e-02
 2.00000000e+00 9.09297427e-01 -4.16146837e-01
 2.50000000e+00 5.98472144e-01 -8.01143616e-01
 3.00000000e+00 1.41120008e-01 -9.89992497e-01
```

This form is suitable if you want the data to read by some other program[1].

[1]It is a bit of a worry that data is not saved to full precision. To save data to full precision save without the `-ascii` option and edit out the Octave comment lines if necessary.

5.5 Importing Data

The `load` command is used to read data in a variety of formats. If the data was saved from Octave, as in the file `mydata` in the previous section, then

```
octave:> load mydata
octave:> y
y =
   0.00000   0.00000   1.00000
   0.50000   0.47943   0.87758
   1.00000   0.84147   0.54030
   1.50000   0.99749   0.07074
   2.00000   0.90930  -0.41615
   2.50000   0.59847  -0.80114
   3.00000   0.14112  -0.98999
```

recovers the variable(s) saved to file.

If the data was saved from Octave, as in the file `mydata.txt` in the previous section, then

```
octave:> load mydata.txt
octave:> mydata
mydata =
   0.00000   0.00000   1.00000
   0.50000   0.47943   0.87758
   1.00000   0.84147   0.54030
   1.50000   0.99749   0.07074
   2.00000   0.90930  -0.41615
   2.50000   0.59847  -0.80114
   3.00000   0.14112  -0.98999
```

reads the data into a matrix whose name is derived from the filename.