Create a vector containing 400 numbers evenly spaced between 0 and 10*pi:
x=linspace(0,10*pi,400);

Plot sin(x) vs. x. The sin function acts on each element of the vector x:
plot(x,sin(x));

Note that to apply operations like * or ^2 elementwise, you precede them with a dot:
plot(x, x.^3 + x.*x - 7*x);

Change the plotting line width:
plot(x,sin(x),"linewidth",5);

Save the plot as a PostScript file:
print -depsc2 "/Users/myname/test.eps"

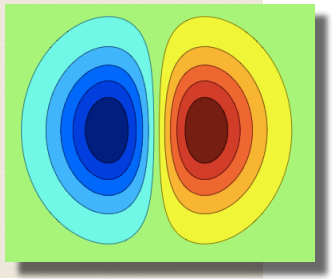
Get more help about the plot function:
help plot

Create a 2D plot by first preparing a "mesh" of x and y values (each will be a matrix):
xrange=linspace(-3,3,200);
yrange=linspace(-3,3,200);
[x,y]=meshgrid(xrange,yrange);

Then plot any function of x and y, like this:
pcolor(xrange,yrange, sin(x.^2+y.^2));

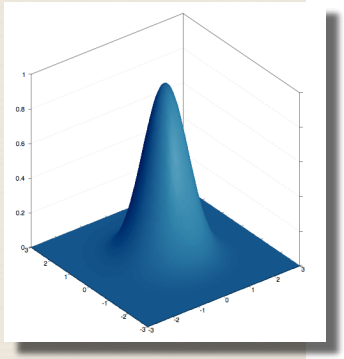
Make the plot nicer by getting rid of the white lines:
shading("flat");

Make a contour plot:
contour(x,y,exp(-x.^2-y.^2).*x);



Make it nicer, with filled colors between the contour lines: Use **contourf** !

Make a 3D surface plot:
surf(x,y,exp(-x.^2-y.^2));
shading("flat");



Use a nicer color scheme:
colormap (ocean (128));

Add axis labels:
xlabel("coordinate x");
ylabel("coordinate y");
zlabel("height");
title("The Gaussian", "FontSize", 20);

Define an empty 3x5 array (3 rows, 5 columns):
M=zeros(3,5);

Set element M_{25} (row 2, column 5):
M(2,5)=42;

Set the column 4 to some vector. Use ":" for the other array dimension (here, the rows):
M(:,4)=[9,8,7];

Calculate a quantum expectation value. Rows of a matrix are separated by semicolon. Psi is defined as a column vector (hence the semicolon!):
sigmax=[0,1 ; 1,0];
Psi=[i;-i]/sqrt(2);
SX = conj(transpose(Psi))*sigmax*Psi

Now we want to solve an equation of motion:
d^2x/dt^2 = -x^3+x+3*sin(t)
This is a driven "Duffing oscillator". As usual, rewrite this into two coupled 1st order equations, via dx/dt=v and dv/dt=…(above). Then call y(1)=x and y(2)=v. Now define a function that returns the time-derivatives, given the current value of y and t:

```
function ydot = f(y,t)
  ydot(1)=y(2);
  ydot(2)=-y(1)^3+y(1)+3*sin(t);
endfunction
```


Solve the diff. equation numerically, after defining the initial value for y0=[x0;v0] as a column vector and the time-points at which we need the solution. The routine needs the time-points as a column vector (not a row vector), so we use "transpose":
y0=[1;3]; t=transpose(linspace(0,40,500));
ysolution=lsode("f",y0,t);

Plot the solution x (component 1 of "y") vs. time:
plot(t,ysolution(:,1));

A phase space plot, by extracting "x" and "v":
plot(ysolution(:,1),ysolution(:,2));

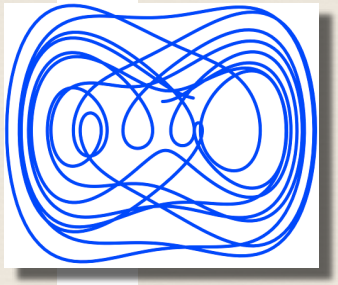
Define a Hamiltonian and get the eigenvalues:
H=[3 , 0.5 ; 0.5 , 1];
lambda=eig(H);

Print them (missing semicolon at end of line!):
lambda

Now we want to sweep one parameter in a 3x3 Hamiltonian, getting the eigenvalues as a function of that parameter. First, prepare the parameter values, and the array "lambdas" (as a N x 3 array) to hold the resulting eigenvalues:
N=200; g=linspace(-4,4,N); lambdas=zeros(N,3);

Loop through all parameter values, and diagonalize H at each value of g. Sort the eigenvalues (needed for nice plots):

```
for j=1:N
  H=[3.0,0.2,0.1 ; 0.2,g(j),0.5 ; 0.1,0.5,0];
  lambdas(j,:)=sort(eig(H));
endfor
```



Plot all 3 eigenvalues, versus the values of "g". This is easy, as the plot routine automatically plots several curves if the "lambdas" is an array:
plot(g,lambdas);

Now we want to solve the time-dependent Schrödinger equation. First, define the Hamiltonian and its dimensions as global variables, so they can be accessed inside the function that defines the right-hand-side of the differential equation:

```
global H;
global dimH;
H=[1.0, 0.4i, 0.7  ;   -0.4i, 0, 0.5  ;   0.7, 0.5, 0 ];
dimH=3;
```

There is a little challenge: The Octave differential equation solver only works for real-valued differential equations. Our solution is to store the real parts of the wave function amplitudes in the first dimH elements of the vector “psi”, and the imaginary parts in the second half. So we need to switch back and forth between this representation and the complex numbers (in which the SEQ itself is written most naturally):

```
function psidot=SEQ(psi,t)
  global H;
  global dimH;
  psiC=psi(1:dimH)+i*psi(dimH+1:2*dimH);
  psiCdot=-i*H*psiC;
  psidot(1:dimH)=real(psiCdot);
  psidot(dimH+1:2*dimH)=imag(psiCdot);
endfunction
```

Initialize psi0 (at time 0), using similar tricks:

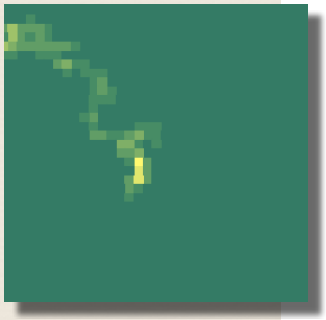
```
PsiComplex0=[1.0,0,0];
psi0=zeros(1,2*dimH);
psi0(1:dimH)=real(PsiComplex0);
psi0(dimH+1:2*dimH)=imag(PsiComplex0);
```

Now solve and plot the result (here: the probability to be in level 3, versus time):

```
t=transpose(linspace(0,20,400));
psis=lsode("SEQ",psi0,t);
PsiComplex=psis(:,1:dimH)+i*psis(:,dimH
+1:2*dimH);
plot(t,abs(PsiComplex(:,3)).^2 );
```

Animation! We implement a random walker, by drawing a random number and deciding which way to go, in each step. “scatter” makes a plot of a set of points (here it is only one point, our particle), and the “pause” waits for 0.1 seconds. We also keep the axis limits fixed, otherwise the plot would be rescaled each time. “nx++” is equivalent to “nx=nx+1”, just as in C.

```
N=100; M=20; nx=0; ny=0;
for j=1:N
  r=rand();
  if r<0.25
    nx++;
  else
    if r<0.5
      nx--;
    else
      if r<0.75
        ny++;
      else
        ny--;
      endif
    endif
  endif
  scatter([nx],[ny],20,"filled");
  axis([-M M -M M], "square");
  pause(.1);
endfor
```



We might want to record how often the particle visits any site. In the beginning, before the loop, we would create the array:

```
counts=zeros(2*M,2*M);
```

Inside the loop, before the pause, we count:

```
 if nx<=M && nx>-M && ny<=M && ny>-M
    counts(nx+M,ny+M)++;
  endif
```

Finally, to plot it, after the loop:

```
pcolor(transpose(counts));
axis([1 2*M 1 2*M], "square");
shading("flat"); colormap(summer);
```


Integration! Naive version of a Riemann sum:

```
x=linspace(0,3,100); dx=x(2)-x(1);
result=sum( exp(-x.^2) ) * dx
```

Use Octave’s built-in routines by first defining the function to be integrated:

```
function res = F ( x )
  res=exp(-x.^2);
endfunction
result=quadgk( "F", 0, 3 )
```

Fourier transform! Use fft to calculate the Fourier transform of a function. It is most efficient to define it on a grid that is a power of two:

```
N=2^8; t=linspace(-30,30,N);
Fourier=fft(sin(2*t-0.5));
```

When trying to plot this, we have to calculate the corresponding frequency range. Also, the FFT produces frequency 0 sitting at index 1. It is nicer to shift everything by half the array length:

```
dt=t(2)-t(1); domega=2*pi/(dt*N);
omega = [ -(ceil((N-1)/2):-1:1)*domega 0
(1:floor((N-1)/2))*domega ];
plot(omega,fftshift(abs(Fourier)));
```

[Note: The FFT of y is normalized such that sum(abs(fft(y)).^2)/sum(abs(y).^2)=N]

WHERE TO GET OCTAVE

You can download Octave for Linux, Windows or Mac OS, for free, both with a graphical user interface (built-in editor) or just to run from a text interface (which still enables plotting, though):
https://www.gnu.org/software/octave/

You can either enter commands directly (get the result immediately), or write commands into a file that you save under “myprogram.m” and which you run from the octave command line simply by typing its name (you have to be in the same directory or use “addpath” to extend the search path):

```
myprogram
```