# Using MATLAB for Systems Calculations   1. Basics

Eric W. Hansen
rev. CRS 6/03

## Introduction

MATLAB (MATrix LABoratory) is a software package designed for efficient, reliable numerical computing. Using MATLAB greatly simplifies the number crunching associated with problems in systems, control, and signal processing. MATLAB is not a piece of instructional software that you will never see again after graduation. Rather, it is widely used in industry for practical design calculations, as well as in academic research. The purpose of this handout is to illustrate, by example, how MATLAB may be used in this course. You can use this handout as a cookbook, to look up a particular calculation when you need it, but you will do well to spend the time necessary to actually figure out how MATLAB works.

In this handout, I will move quickly through the basics. For more information, the MATLAB systems is the first place to turn. Throughout, computer code is represented by a different font, `like this`. Furthermore, what you type is **`bold`**, and what the computer returns is `plain`. The MATLAB prompt looks like this: `».` Comments within MATLAB code are marked off with the percent sign, like this: `% This is a comment.`

## MATLAB as a calculator

MATLAB can be used as a simple calculator. For example, to add two numbers you can type:

```
» 12 + 15

ans =

     27
```

That's not too exciting, but one of the great things about MATLAB is that you can handle much more complex calculations almost that simply. But you can first learn many of its features through using it just as a calculator.

One handy feature is that you can easily modify any command you typed without having to retype the whole thing. The up-arrow key will display the previous command on the command line. You can then edit that line and hit return again to redo the modified calculation. Hitting the up arrow repeatedly lets you go back to any previous command.

Another way to avoid retyping things is to use variables. For example, we could type:

```
» a = 10 + 5

a =

     15

» 12 + a

ans =

     27
```

If you didn't think to put a result into a variable, you can still reuse it, because MATLAB puts it into a default variable 'ans', so you can then type:

```
» ans/2

ans =

     13.5000
```

If you don't want to see the result you can type a semicolon at the end of the line to suppress the output for that command.

```
» a = 17;
»
```

MATLAB of course has all the mathematical functions you might expect, like square root (sqrt), sine (sin), and absolute value (abs). To quickly get the details on any function, type help and the name of the function on the command line:

```
» help sin

SIN    Sine.
    SIN(X) is the sine of the elements of X.
```

Note that the capitalization is used for emphasis; to use the function you do not need caps. Also note that the argument of trigonometric functions in MATLAB is always in radians.

```
» sin(pi/2)


ans =

     1
```

Above we used simple commands for simple calculations. But matlab can do much more complex calculations with commands almost this simple. This is because....

## The basic data type is a complex-valued matrix

This is the basic paradigm for computation with MATLAB, and the key to its power. Here's an example:

```
» A = [1 2; 3 4; 5 6]

A =
     1     2
     3     4
     5     6
```

You use a space (or a comma) to separate elements in a row, and semicolons (or a linefeed) to separate rows. MATLAB always types back the result of your command. Again, if you don't want to see the result you can type a semicolon at the end of the line to suppress the output for that command.

```
» A = [1 2; 3 4; 5 6];
»
```

A vector is a special case of a matrix — a single row or a single column.

```
» b = [1;2;3]

b =
     1
     2
```

```
          3

    » b = [1
    2
    3]

    b =
         1
         2
         3

    » c = [4 5 6]

    c =
         4     5     6

    » c = [4, 5, 6]

    c =
         4     5     6
```

A scalar is also a special case of a matrix (1 row by 1 column)

```
    » d = 2

    d =
         2
```

Complex numbers are written using *i* or *j* — MATLAB predefines `i` and `j` to be equal to $\sqrt{1}$ (be careful not to define them to be something else!)

```
    » z = 1+2i

    z =
       1.0000 + 2.0000i
```

A matrix (or vector) is transposed by putting a prime on it. If `x` is complex-valued, `x'` is the conjugate transpose, and `x.'` is just the transpose without conjugation. If `x` is real, of course, there is no difference between `'` and `.'` .

```
    » [1+i 2-i]'

    ans =
       1.0000 - 1.0000i
       2.0000 + 1.0000i

    » [1+i 2-i].'

    ans =
       1.0000 + 1.0000i
       2.0000 - 1.0000i
```

To find the size of a matrix,

```
    » size(A)

    ans =
         3     2
```

that is, 3 rows × 2 columns. The statement `s=size(A)` puts the size into a 1×2 vector, which can then be used in other calculations. `s(1)` is the number of rows, and `s(2)` is the number of columns.

## Arithmetic is performed according to matrix-vector rules

1. **Addition and subtraction**

   In an ordinary computer language (like C), you add two variables by writing z = x+y. In MATLAB, you do the same thing, but MATLAB performs the operation according to the rules for matrix-vector arithmetic. Here are some examples

```
» b1 = [1 2 3]

b1 =
     1     2     3

» b2 = [4 5 6]

b2 =
     4     5     6

» b1 + b2

ans =
     5     7     9

» b1 - b2

ans =
     -3    -3    -3
```

   If the dimensions of the vectors don't agree, you get an error

```
» b = [1; 2; 3];
» c = [4 5 6];
» b+c

??? Error using ==> +
Matrix dimensions must agree.
```

   The only exception to this rule is for adding a scalar to a vector or matrix. There are many times when you want to increase each element of a vector by the same amount, and MATLAB conveniently lets you do this as follows:

```
» 1+c

ans =
    5     6     7
```

2. **Multiplication**

   Multiplication comes in two flavors: "dot" product and "direct" product. The dot product follows the usual linear algebra rule: $\vec{u} \cdot \vec{v} = \mathbf{u}\mathbf{v}^T = u_1 v_1 + u_2 v_2 + u_3 v_3 + \ldots + u_N v_N$.

```
» b1

b1 =
     1     2     3

» b2

b2 =
     4     5     6

» b1*b2

??? Error using ==> *
```

```
Inner matrix dimensions must agree.

» b2’

ans =
      4
      5
      6

» b1*b2’

ans =
      32
```

Matrix-vector or matrix-matrix multiplication is just many dot products put together. Be careful that your matrix dimensions agree, or you get an error. The rule for checking matrix dimensions is this: If you have two matrices, with dimensions $r_1 \times c_1$ and $r_2 \times c_2$, you can multiply them if $c_1 = r_2$ — the “inner dimensions agree”.

```
» A = [1 2; 3 4; 5 6]

A =
     1     2
     3     4
     5     6

» b = [1;2;3]

b =
     1
     2
     3

» A*b

??? Error using ==> *
Inner matrix dimensions must agree.

» A'*b

ans =
    22
    28
```

The other kind of multiplication, the direct product, is denoted by “.*” rather than “*”. It simply multiplies element-by-element

```
» b1.*b2

ans =
     4    10    18
```

Multiplication by a scalar behaves the way you’d like it to:

```
» 2*c

ans =
     8    10    12
```

**3. Division**

Division is tricky. The element-by-element operation is the one you’ll use most frequently for systems calculations:

```
» b1./b2

ans =
    0.2500    0.4000    0.5000
```

When used with a scalar, the plain `/` behaves the way you'd expect:

```
» c/2

ans =
     2.0000    2.5000    3.0000
```

But when both operands are matrices, things are different. The two division operations, denoted by `\` and `/`, solve systems of simultaneous linear algebraic equations. Consult a manual for details.

4. **Powers**
   Exponents can be applied element-by-element:

```
» b.^2

ans =
     1
     4
     9
```

Without the dot, **`A^P`** the attempts to perform the matrix operation $A^p$. This only works for a square matrix.

```
» M = [1, 2; 3, 4];

» M.^2

ans =
      1     4
      9     16

» M^2

ans =
      7     10
      15    22
```

The result of **`M^2`** is the matrix `M*M`.

5. **Other math functions**
   In MATLAB, all the usual math functions are applied element-by-element. For example, `sin(x)` returns a vector of sine values.

## Shortcuts for matrix manipulation

1. To create a vector of regularly spaced values, *e.g.*, a time axis for a simulation:

```
% The colon indicates a range of values
» t = 0:10

t =
     0     1     2     3     4     5     6     7     8     9    10

% You can specify the step size
» t = 0:.1:1
```

```
t =
  Columns 1 through 7
         0    0.1000    0.2000    0.3000    0.4000    0.5000    0.6000

  Columns 8 through 11
    0.7000    0.8000    0.9000    1.0000

% linspace(first, last, N) gives N equally spaced points between first and last.
» t = linspace(0,1,6)

t =
         0    0.2000    0.4000    0.6000    0.8000    1.0000
```

**2.** Matrices of all ones or all zeros. `ones(rows, cols)` and `zeros(rows, cols)`:

```
» N = ones(2,3)

N =
     1     1     1
     1     1     1

» P = zeros(1,2)

P =
     0     0
```

**3.** To extract parts of a matrix:

```
» A = [1 2; 3 4; 5 6]

A =
     1     2
     3     4
     5     6

» A(1,1)

ans =
     1

» A(2,3)

???  Index exceeds matrix dimensions.

» A(3,2)

ans =
     6

% Again, the colon specifies a range
» A(1:2, 1)

ans =
     1
     3

» A(2:3, 1:2)

ans =
     3     4
     5     6

% The colon, used by itself, means “all rows” or “all columns”
» A(2:3, :)
```

```
ans =
     3     4
     5     6

» A(:, 1)

ans =
     1
     3
     5
```

**4.** To manipulate parts of a matrix

```
» A(1,1)=0

A =
     0     2
     3     4
     5     6

» b(1:2) = [-1; -2]

b =
    -1
    -2
     3
```

**5.** Build larger matrices and vectors out of smaller ones.

```
» b = [1; 2; 3]

b =
     1
     2
     3

» b2 = [b; 4]

b2 =
     1
     2
     3
     4

» A2 = [A, b]

A2 =
     0     2     1
     3     4     2
     5     6     3
```

**6.** The index 'end' signifies the last element of a vector.

```
» b = [1; 2; 3];

» b(end)

ans =
3
```

## Strings

Strings are just arrays of characters, rather than numbers.

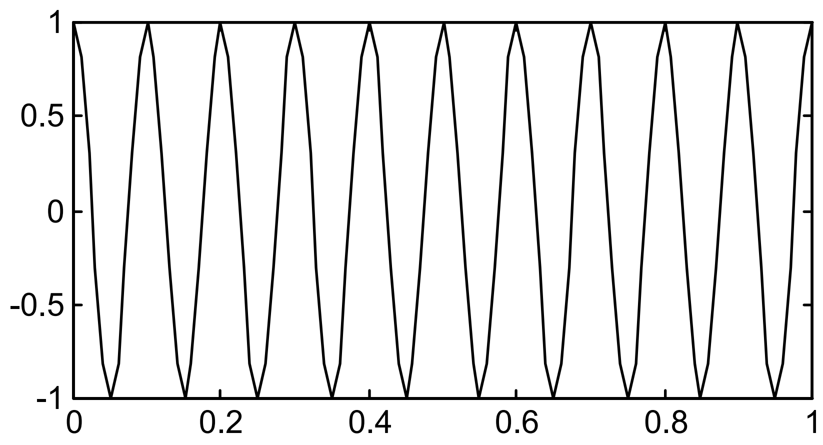
```
      »s1 = 'Welcome to Engs 22.'

      s1 =
      Welcome to Engs 22.

      »s1(1:4)

      ans =
      Welc

      »s1(end-4:end)

      ans =
      s 22.

      »s2 = 'Hope you have a good term.'

      s2 =
      Hope you have a good term.

      »s = [s1, s2]

      s =
      Welcome to Engs 22.Hope you have a good term.
```

Note the lack of a space between the two strings. How would you fix that?

## Calculating and plotting functions

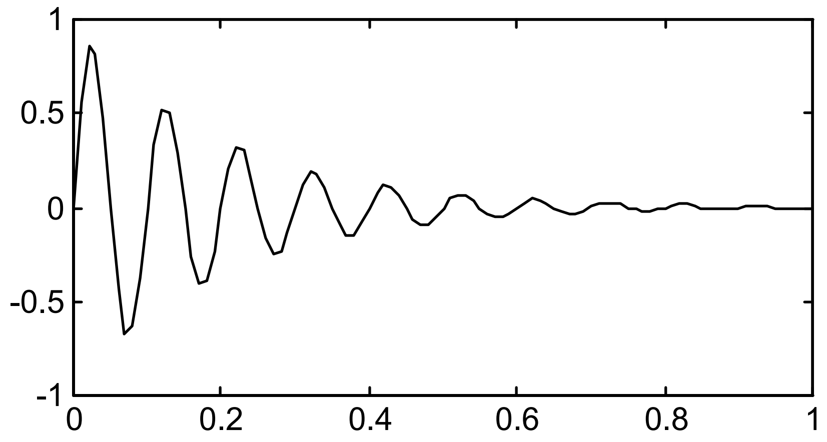
To make and graph a sine wave:

```
      » f = 10;         % Hz
      » dt = 0.01;      % sec
      » t = 0:dt:1;     % time axis vector
      » x = cos(2*pi*f*t);
      » plot(t,x)
```



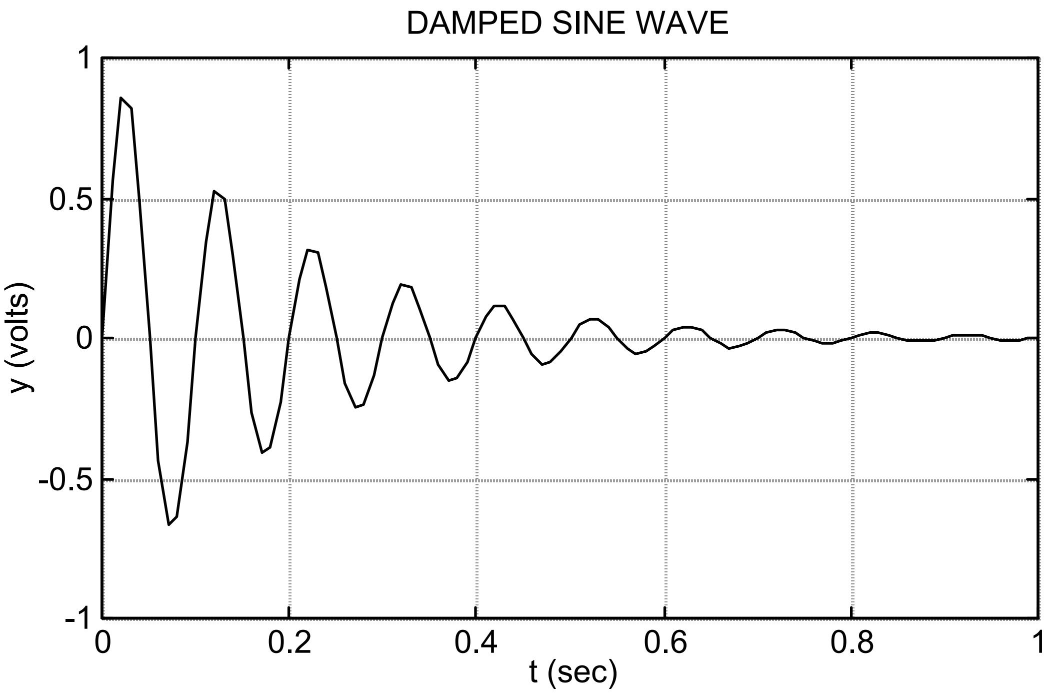
Here's a damped sine wave. Note the use of the ".*" operation.

```
      » tau = 0.2;      % sec
      » y = exp(-t/tau) .* sin(2*pi*f*t);
      » plot(t,y)
```



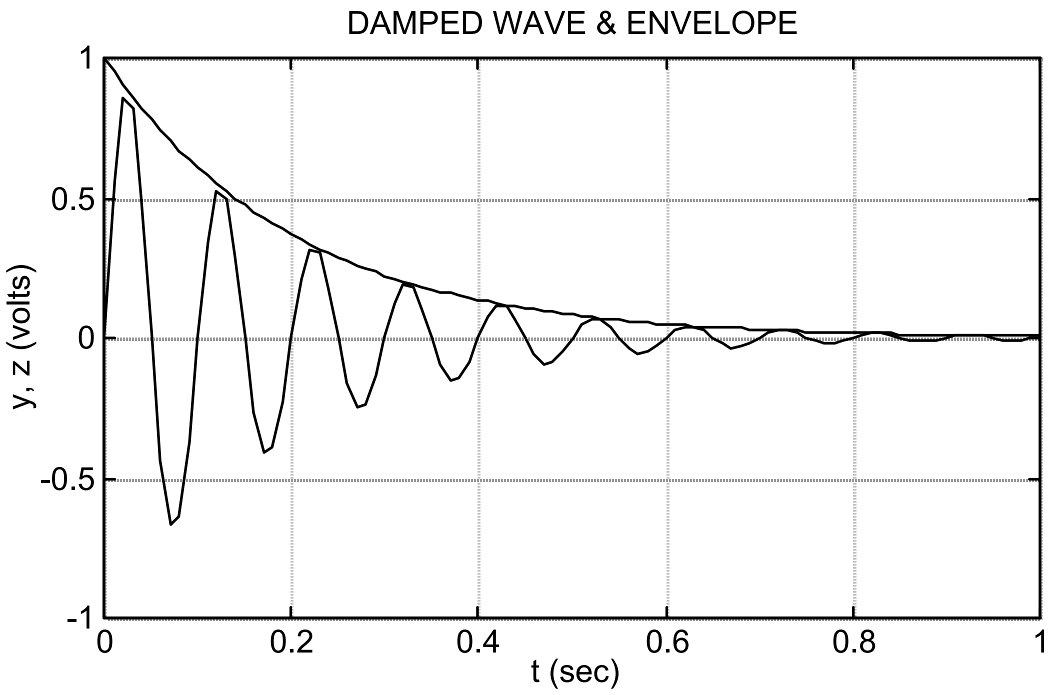
It’s easy to add labels to a plot.

```
» xlabel(‘t (sec)’), ylabel(‘y (volts)’)
» title(‘DAMPED SINE WAVE’)
» grid      % Add a grid
```



You can put two functions on the same graph.

```
» z = exp(-t/tau);
» plot(t,y, t,z), grid
» xlabel(‘t (sec)’), ylabel(‘y,z (volts)’)
» title(‘DAMPED WAVE & ENVELOPE’)
```



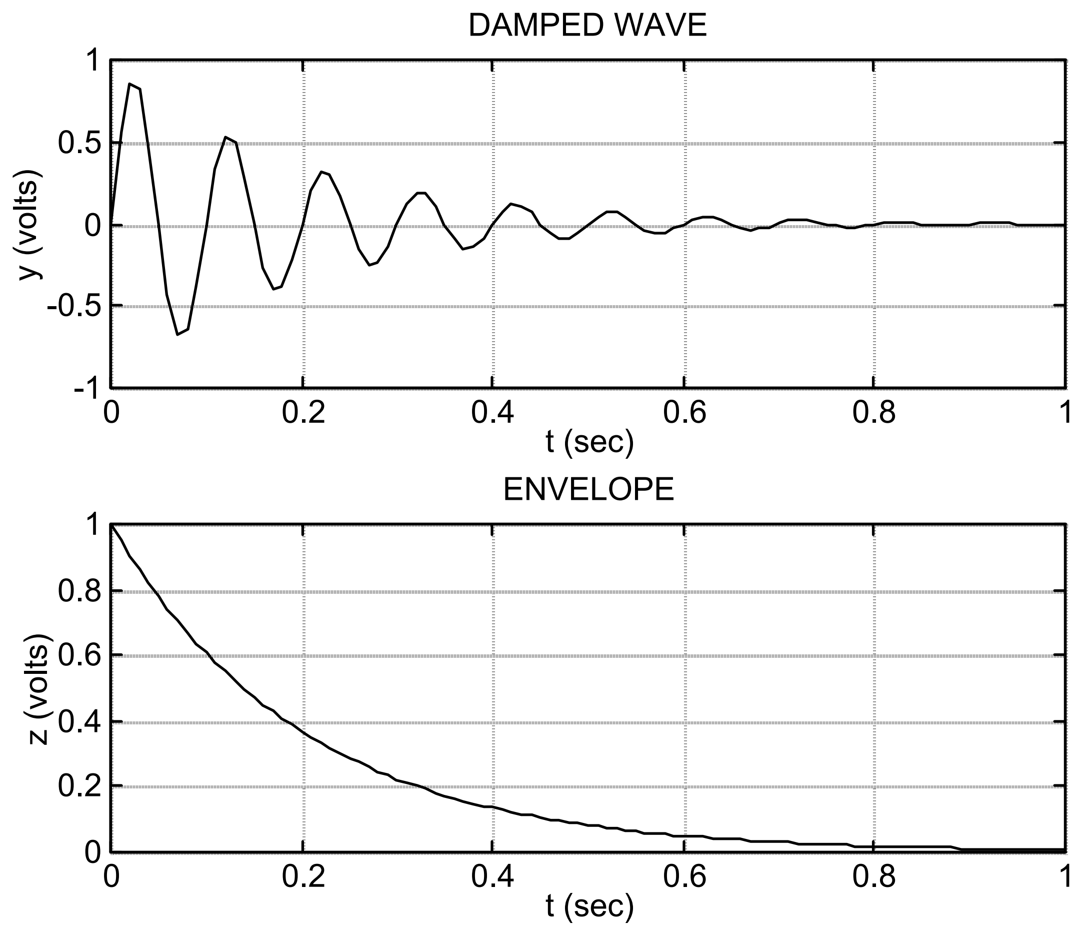
The `subplot` command enables you to put two graphs in the same figure window. It divides the window into an array of "panes", which are numbered consecutively across, then down, from the upper left. For example, `subplot(2,3,1)` divides the window into six panes, two rows × three columns, and directs the subsequent plot commands to the upper-left, or first, pane. `subplot(2,3,3)` plots in the upper right, and `subplot(2,3,4)` in the lower left, etc.

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |

With y and z defined as above,

```
» subplot(2,1,1)
»   plot(t,y), grid
»   xlabel('t (sec)'), ylabel('y (volts)'), title('DAMPED WAVE')
» subplot(2,1,2)
»   plot(t,z), grid
»   xlabel('t (sec)'), ylabel('z (volts)'), title('ENVELOPE')
```

(the indenting is optional, just to make it clear which plot commands belong to which subplot), produces the following graph:



## MATLAB Programming (M-files)

One of the nice things about MATLAB is that sequences of MATLAB commands can be combined into **M-files**, which enables you to create your own commands or MATLAB programs. Many of the MATLAB commands you use are themselves M-files composed of more primitive MATLAB operations. When I am doing a quick calculation or just trying something out, I usually work just in the command window, but for a major calculation, I use M-files a lot.

M-files come in two flavors. A **script** is simply a set of commands stored in a file, foo.m, say. (Without the ".m" extension, MATLAB can't find it). When you type `foo` in the MATLAB command window, the commands in the script are executed in sequence. For example, suppose foo.m contains the following

```
y = cos(x);

plot(x, y);
```

The M-file is created with a text editor and stored in a directory where MATLAB can find it (see the `path` command). In your command window, you create the vector x, then run foo:

```
» x = 2*pi*linspace(0,1,100);       % 100 points between 0 and 2pi

» foo
```

The result will be a plot of a cosine curve. If you then change x and run foo again, you'll get a different result.

Scripts are simply shorthand for things you would type in the command window. They aren't true functions like you learned to write in Engs 20. The variables in the script are global in scope, just like the ones in the command window. So, if you have a variable y already defined in your command window, foo will change it. This might not be what you want to do.

In most applications of M-files, you'll want to create a **function**. The variables in a function have local scope, and inputs and outputs are specified in the function call, as in C++ or Java. A function M-file looks like a script, except for the first line. Here's the foo script rewritten as a function:

```
function y = foo(theta)
y = cos(theta);
plot(theta, y);
```

and here are the MATLAB commands which invoke foo:

```
» x = 2*pi*linspace(0,1,100);       % 100 points between 0 and 2pi
» foo(x);
```

If you actually wanted to use the cosine values in a subsequent calculation, you could take the output of foo and put it into a vector:

```
» z = foo(x);
```

Note that the output vector doesn't have to be called y, even though it's called y in the M-file. The variable y in foo.m is local to the function, and loses its identity when its values are passed back to the command window, where it was called. Likewise, the input variable can be called theta in the function, but you can use any different name when you call the function.

M-files can become quite sophisticated, with if-then-else flow control, multiple inputs and outputs, and dialog boxes, but these basic facts should be enough to get you started.

## Printing from MATLAB

1. **Figures** can be printed from the file menu in the figure window.

2. **m-files** can't be printed from MATLAB on our linux or UNIX systems. You must save the file, and then print it using the 'lpr' print command from UNIX. Open a UNIX shell, and make sure you are in the same directory as the file you want to print. Then, to print an m-file called "filename," type (in the UNIX window NOT the MATLAB window)

   ```
   %lpr filename.m
   ```

   To specify the printer it goes to, use –P and then the name of the printer, e.g.,

   ```
   %lpr –Plw225-1 filename.m
   ```

3. **Saving a plot to a file for inclusion in a word processing document**

   From the figure window "File" menu, choose "export." You have a choice of formats; EPS (encapsulated postscript) is a good choice, because it saves "vector" graphics—information about where lines are drawn, what text there is and what font it is in, etc. This results in the highest quality graphics on whatever printer you are using, without using as much memory as "bitmap" formats that simply record the color of each pixel. You should particularly avoid using JPG (JPEG) format for line art such as graphs, because it is designed for photographs and will make simple line art look terrible.