The Mastering MATLAB Toolbox

The *Mastering MATLAB Toolbox* is a collection of approximately 140 function M-files written by the authors of *Mastering MATLAB 5*. The functions in the *Toolbox* range from commonly used utilities, to numerical analysis and optimization functions, to powerful high level plotting routines, to polished graphical user interface functions for setting axes, line, surface, paper, and other properties. If you downloaded or otherwise received a copy of the *Mastering MATLAB Toolbox* that was associated with the first edition of this text, you have a general idea of what is available in the toolbox. However, since then a lot has changed. The changes include:

1) The size of the *Toolbox* has nearly doubled, with many of the new functions providing significant new capabilities.
2) Many of the original functions were modified to improve functionality.
3) All original functions have been reviewed and revised to make them MATLAB version 5 compatible and to make use of new features in version 5.
4) Some function names have been changed to improve naming consistency.

This edition of the *Mastering MATLAB Toolbox* is freely available from

```
http://www.eece.maine.edu/mm
```

After decompressing the `Toolbox` file on your platform, you will find a directory or folder named `MM5` containing the files: `Register`, `Contents.m`, `Readme.m`, and numerous `.p` files, which are the toolbox files in *p-code* format. To install the *Mastering MATLAB Toolbox*, simply copy the `MM5` directory and its contents into the `Toolbox` directory (within the `MATLAB` directory) and add the `MM5` directory to the MATLAB search path (using the tools discussed in Chapter 5.)

Since the *Toolbox* is distributed in *p-code* format, no on-line help is available for the *Toolbox*. That is, » `help` *`mmname`* or » `helpwin` *`mmname`* displays no help text for the function *`mmname`*. In addition, it is not possible to see the MATLAB code that implements the functions. If you wish to have access to the *Toolbox* M-files and their help files, you must register the *Toolbox*. This is most easily accomplished by going to

```
http://order.kagi.com/?4D8
```

and following the directions posted there. Alternatively, you can run the `Register` program on PC and Macintosh platforms, or fill out the `Register` text file on Unix

platforms.

All net proceeds of the registration fee for unlocking the *Mastering MATLAB Toolbox* M-files are contributed to an endowment fund for undergraduate student scholarships in the Department of Electrical and Computer Engineering at the University of Maine.

Upon payment of the registration fee, you will receive instructions by e-mail for creating the *Mastering MATLAB Toolbox* M-files. These M-files contain complete help text documenting all features of each *Toolbox* function. In addition they provide valuable examples that demonstrate efficient MATLAB programming.

Questions and comments regarding Mastering MATLAB 5 and the *Mastering MATLAB Toolbox* can be sent via e-mail to:

```
mm@eece.maine.edu
```

Mastering MATLAB Toolbox Function Tutorial

The primary features of the *Mastering MATLAB Toolbox* are demonstrated in the rest of this document. To facilitate the discussion, the functions are discussed in an order that coincides with the chapter order of this text. The section names used in this chapter correspond to the associated chapter title in the text.

Chapter 2: Basic Features

The *Mastering MATLAB Toolbox* functions associated with the material in the *Basic Features* chapter are shown in the table below.

Function	Description
mmdigit	Round to given significant places in specified base.
mmlimit	Limit values between extremes.
mmlog10	Dissect decimal floating point numbers.
mmquant	Quantize input values.

These utility functions perform basic operations on arrays of numbers. **mmdigit** allows one to round numbers to precision less than the standard double precision used in MATLAB. This function is useful for studying the effects of finite precision on parameter values in areas such as filtering and control. For example,

```
» mmdigit(pi,2,10) % round pi to 2 digits in base 10
ans =
          3.1
» ps = mmdigit(pi,23,2) % round pi to 23 digits (bits) in base 2
ps =
       3.1416
» pi - ps % compare single to double precision
ans =
     1.51e-07
» r = rand(1,5).^randn(1,5)
r =
      0.97547       1.0074       1.1479      0.39809        0.807
» rr = mmdigit(r,3,10)
rr =
        0.975         1.01         1.15        0.398        0.807
```

As shown above mmdigit allows one to round numbers to any realizable number of digits in any base. In general, it returns an array having the same size as that input.

The function **mmlimit** constrains array elements to lie between lower and upper bounds with those elements that violate the bounds being set equal to the respective bound. For example,

```
» x = linspace(-pi,pi,6) % create data
x =
      -3.1416       -1.885     -0.62832      0.62832        1.885       3.1416
» y = mmlimit(x,0,4) % limit to range 0 to 4
```

```
y =
            0            0            0      0.62832        1.885       3.1416

» [y,i] = mmlimit(x,0,4) % return which limit was hit
y =
            0            0            0      0.62832        1.885       3.1416
i =
    -1    -1    -1     0     0     0
```

In the above example, the scalar lower and upper limits were applied to all elements of the input vector x. A second output argument returns an array containing the elements –1, 0, and +1, where –1 is returned for those elements that exceed the lower limit, 0 is returned for those elements that do not exceed either limit, and +1 is returned for those elements that exceed the upper limit. In addition to scalar lower and upper limits, arrays the same size as the input argument to be limited can be used. In this case, the limits apply on an element by element basis. For example,

```
» x % review data
x =
      -3.1416       -1.885     -0.62832      0.62832        1.885       3.1416
» low = 1 - rand(1,6) % create lower limit
low =
     0.083096      0.58973      0.10635      0.94211      0.64713      0.18683
» high = 2 + low % create upper limit
high =
       2.0831       2.5897       2.1064       2.9421       2.6471       2.1868
» [z,i] = mmlimit(x,low,high) % now do element by element limiting
z =
     0.083096      0.58973      0.10635      0.94211        1.885       2.1868
i =
    -1    -1    -1    -1     0     1
```

The function **mmlog10** dissects numbers into a mantissa whose magnitude lies between one and ten and an integer power-of-ten exponent, i.e., it splits numbers into their scientific notation components. For example,

```
» x = rand(1,6).^(10*randn(1,6)) % create some data
x =
   3.0401e+10      0.12981   3.9542e+05   1.6788e-05   1.5174e+05   2.5384e-06
» [m,e] = mmlog10(x) % dissect it
m =
       3.0401       1.2981       3.9542       1.6788       1.5174       2.5384
e =
    10    -1     5    -5     5    -6
```

In this example, `m` is the mantissa and `e` is the exponent. As with `mmdigit` and `mmlimit`, `mmlog10` returns output arrays the same size as the input array.

The function **`mmquant`** quantizes the values in an input array. For example,

```
» x = linspace(0,pi,7);
» y = sin(x)  % create data to quantize
y =
         0    0.5000    0.8660    1.0000    0.8660    0.5000         0

» yq = mmquant(y,9,-1,1)
yq =
         0    0.5000    0.7500    1.0000    0.7500    0.5000         0
```

Here, the array `y` is quantized to 9 levels between –1 and 1. Input array values that exceed the specified lower or upper bounds are quantized to the bounds.

Chapter 5: File and Directory Management

The *Mastering MATLAB Toolbox* functions associated with the material in the *File and Directory Management* chapter are shown in the table below.

Function	Description
`mmcd`	Change working directory using a GUI.
`mmload`	Load matrix and string header from an ASCII
`mmsave`	Save matrix and string header to an ASCII file.

The function **`mmcd`** is a GUI that performs the same function as the command `cd`. That is, `mmcd` allows one to use the mouse to graphically change directories. A screen shot of the `mmcd` window is shown below.

This GUI has an editable text box at the top showing the current directory. If desired, a new directory can be typed into this box. In the middle of the GUI is a listbox showing subdirectories within the current directory, with `..` representing the parent

directory. Double-clicking on any of the directories listed in the listbox makes it the current directory, thereby changing the subdirectory list in the listbox. Three of the four buttons at the bottom of the GUI take you to specific directories. The **Matlab** button goes to the directory specified by the command `matlabroot`. The **Toolbox** button goes to the `Toolbox` directory. Pressing the **Restore** button restores the current directory that existed when `mmcd` was called. The **Close** button accepts the new current directory and closes the GUI.

The functions **`mmload`** and **`mmsave`** provide convenient tools for loading and saving ASCII data files that have character string headers. File names and directory placement are chosen using the dialog boxes `uigetfile` and `uiputfile` respectively. In doing so, it is easy to load or save a file anywhere on your file system. To demonstrate the use of `mmload` and `mmsave`, consider the following example data:

```
» x = linspace(0,2*pi,10)';
» y = [sin(x) cos(x) tan(x)];
» h = '       sin(x)      cos(x)        tan(x)';
» disp(h), disp(y)
       sin(x)       cos(x)         tan(x)
            0            1            0
      0.64279      0.76604       0.8391
      0.98481      0.17365       5.6713
      0.86603         -0.5      -1.7321
      0.34202     -0.93969     -0.36397
     -0.34202     -0.93969      0.36397
     -0.86603         -0.5       1.7321
     -0.98481      0.17365      -5.6713
     -0.64279      0.76604      -0.8391
            0            1            0
```

Issuing `mmsave(y,h)` saves the data in a format that looks like that shown above. First the header text `h` is placed in the file, then the data contained in the variable `y` is saved. The file name and directory are chosen using the `uiputfile` dialog box. After doing so, the file contents are:

```
   sin(x)      cos(x)       tan(x)
     0       1        0
  0.64279    0.76604     0.8391
  0.98481    0.17365     5.6713
  0.86603      -0.5     -1.7321
  0.34202    -0.93969    -0.36397
 -0.34202    -0.93969     0.36397
 -0.86603      -0.5      1.7321
 -0.98481     0.17365    -5.6713
 -0.64279     0.76604    -0.8391
    0       1       0
```

The function **mmload** performs the inverse operation. It loads a file containing data such as that above. After opening the file, mmload returns a string matrix containing all lines of the header text and assigns the data to a numerical variable. Applying mmload to the data file just created produces:

```
» [Y,S] = mmload
Y =
            0            1            0
      0.64279      0.76604       0.8391
      0.98481      0.17365       5.6713
      0.86603         -0.5      -1.7321
      0.34202     -0.93969     -0.36397
     -0.34202     -0.93969      0.36397
     -0.86603         -0.5       1.7321
     -0.98481      0.17365      -5.6713
     -0.64279      0.76604      -0.8391
            0            1            0
S =
       sin(x)      cos(x)        tan(x)
```

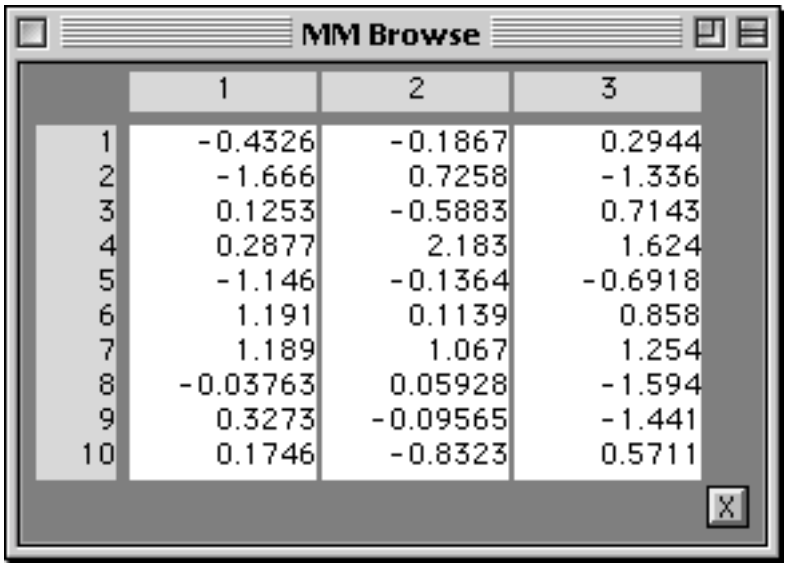
Chapter 6: Arrays and Array Operations

The *Mastering MATLAB Toolbox* functions associated with the material in the *Arrays and Array Operations* chapter are shown in the table below.

Function	Description
`mmbrowse`	Graphical matrix browser.
`mmdeal`	Deal array or string matrix into individual arguments.
`mmrand`	Uniformly distributed random arrays.
`mmrandn`	Normally distributed random arrays.
`mmshiftd`	Shift or circularly shift matrix rows.
`mmshiftr`	Shift or circularly shift matrix columns.

These functions provide useful utilities for arrays. **`mmbrowse`** is a GUI that allows one to graphically view the contents of a one- or two-dimensional array. For example, passing the following data to `mmbrowse` produces the window shown below.

```
» d = randn(10,3);
» mmbrowse(d)
```



When the data array is too large to show in its entirety, scroll bars appear allowing the user to scroll to any part of the array. Clicking on a scroll bar arrow increments by one row or column. Clicking on a scroll bar trough increments the array by one page in either the row or column direction. The array cannot be edited using this

GUI; it is provided for viewing only. The GUI is closed by clicking the standard window close box or by pressing the X button in the lower right corner.

The function **mmdeal** is a utility for extracting or dealing an array into individual elements. For example,

```
» x = eye(3); % example data
» [a,b,c] = mmdeal(x,1) % deal rows
a =
     1     0     0
b =
     0     1     0
c =
     0     0     1
» [d,e,f] = mmdeal(x,2) % deal columns
d =
     1
     0
     0
e =
     0
     1
     0
f =
     0
     0
     1
» [a,b,c,d] = mmdeal(rand(2)) % deal individual elements
a =
      0.95013
b =
      0.23114
c =
      0.60684
d =
      0.48598
» [e,f] = mmdeal(x,'dup') % simply duplicate x
e =
     1     0     0
     0     1     0
     0     0     1
f =
     1     0     0
     0     1     0
     0     0     1
```

In addition, mmdeal deals out the contents of a cell array just as the function deal

does:

```
» g = {eye(2) ones(2)}
g =
    [2x2 double]    [2x2 double]
» [a,b] = mmdeal(g)
a =
     1     0
     0     1
b =
     1     1
     1     1
```

`mmdeal` has other properties as well that will be discussed in the character string section of this chapter.

The functions **`mmrand`** and **`mmrandn`** provide a shell over the functions `rand` and `randn` respectively that automate the procedure of generating random array having statistical properties that differ from the defaults used by `rand` and `randn`. For example,

```
» avg = 5; % desired average value
» tol = 2; % desired +/- tolerance
» mmrand(avg,tol,2,5) % create 2-by-5 array
ans =
       3.2316       6.2527       3.5556       3.7949       4.0888
       4.4115       3.0394       3.8111       5.4152       3.7953
» var = 3; % desired variance
» mmrandn(avg,var,3,4)
ans =
       3.8003       7.1357       8.5725       4.5298
         7.07       8.8707       1.3926      0.18774
       7.4469       7.0058       4.9406       5.7719
```

`mmrand` allows one to specify the average value and spread of the uniformly distributed array. In the above example, `mmrand(avg,tol,2,5)` produced a 2-by-5 array having an average value of 5, uniformly distributed between `avg-tol` and `avg+tol`. `mmrandn` allows one to specify the average value and variance of the normally distributed array. In the above example, `mmrandn(avg,var,3,4)` produced a 3-by-4 array having and average value of 5 and a variance of 3.

The functions **`mmshiftd`** and **`mmshiftr`** shift two-dimensional arrays down and to the right respectively. For example,

```
» x = [1:5;6:10;11:15] % example data
```

```
x =
     1     2     3     4     5
     6     7     8     9    10
    11    12    13    14    15
» mmshiftd(x,1) % shift down one row
ans =
     0     0     0     0     0
     1     2     3     4     5
     6     7     8     9    10
» mmshiftd(x,-2) % shift up two rows
ans =
    11    12    13    14    15
     0     0     0     0     0
     0     0     0     0     0
» mmshiftd(x,-2,0) % same as above
ans =
    11    12    13    14    15
     0     0     0     0     0
     0     0     0     0     0

» mmshiftd(x,1,1) % circularly shift down one row
ans =
    11    12    13    14    15
     1     2     3     4     5
     6     7     8     9    10
» mmshiftd(x,-2,1) % circularly shift up two rows
ans =
    11    12    13    14    15
     1     2     3     4     5
     6     7     8     9    10
» mmshiftr(x,3) % shift to the right three columns
ans =
     0     0     0     1     2
     0     0     0     6     7
     0     0     0    11    12
» mmshiftr(x,-1,1) % circularly shift left one column
ans =
     2     3     4     5     1
     7     8     9    10     6
    12    13    14    15    11
```

As shown above, negative shifts move up and to the left. The presence of a logical True third argument signifies a circular shift rather than replacement with zeros.

Chapter 8: Relational and Logical Operations

The *Mastering MATLAB Toolbox* functions associated with the material in the *Relational and Logical Operations* chapter are shown in the table below.

Function	Description
mmempty	Substitute value if empty.
mmisv5	True for MATLAB version 5.
mmono	Test for monotonic vector.

The function **mmempty** provides simple substitution for empty first input arguments. For example,

```
» mmempty(eye(3),rand(3))
ans =
     1     0     0
     0     1     0
     0     0     1
» mmempty([],rand(3))
ans =
      0.67214      0.68128      0.50281
      0.83812      0.37948      0.70947
      0.01964      0.83180      0.42889
```

This function is useful for substituting default values for empty parameters and for avoiding problems with the find function when an input may be empty:

```
» a = []; % empty variable
» find(a~=0)
Warning: Future versions will return empty for empty ~= scalar comparisons.
ans =
     1
» find(mmempty(a,0)~=0) % no problem here!
ans =
     []
```

The function **mmisv5** provides a logical test for the presence of MATLAB version 5 that is useful when creating M-files that must work under both MATLAB version 4 and 5. For example,

```
» v5 = mmisv5
v5 =
     1
```

```
» islogical(v5)
ans =
     1
```

In addition, `mmisv5` can provide logical results for subversions of MATLAB as well. For example,

```
» mmisv5(5.1) % yes, this is version 5.1
ans =
     1

» mmisv5(5.0) % no, it's not version 5.0
ans =
     0
```

The function **`mmono`** returns an integer signifying the monotonicity of its input vector. For more information on this function see on-line help.

Chapter 9: Set, Bit, and Base Operations

The *Mastering MATLAB Toolbox* function associated with the material in the *Set, Bit, and Base Operations* chapter is shown in the table below.

Function	Description
mmismem	True for set members.

The function **mmismem** is a competitor to the ismember function in MATLAB. mmismem is orders of magnitude faster than ismember for large sets and uses much less memory for temporary variables. For example,

```
» set = rand(100); % a set of 10,000 elements
» a = rand(50); % a set of 2,500 elements to find in set

» tic, mmismem(a,set); toc % Mastering MATLAB function
elapsed_time =
      0.41658
» tic, ismember(a,set); toc % MATLAB function
elapsed_time =
       53.968

» a = linspace(0,1,100); % try smaller sets
» set = linspace(-1,1,200);
» tic, mmismem(a,set); toc
elapsed_time =
     0.015127
» tic, ismember(a,set); toc
elapsed_time =
     0.029379
```

Chapter 10: Character Strings

The *Mastering MATLAB Toolbox* functions associated with the material in the *Character Strings* chapter are shown in the table below.

Function	Description
mmdeal	Deal array or string matrix into individual arguments.
mmonoff	String ON/OFF to/from logical conversion.
mmprintf	Data array to string matrix conversion.

Earlier the function **mmdeal** was shown to deal a numerical array into individual variables. When called with a string matrix or cell array of strings input, the output arguments are individual deblanked strings. For example,

```
» lawyers = char('Cochran','Shapiro','Clark','Darden'); % string matrix
» [a,b,c,d] = mmdeal(lawyers) % deal them out deblanked
a =
Cochran
b =
Shapiro
c =
Clark
d =
Darden
» lawcells = cellstr(lawyers) % convert to cell array
lawcells =
    'Cochran'
    'Shapiro'
    'Clark'
    'Darden'
» [a,b,c,d] = mmdeal(lawcells) % deal out cell contents deblanked
a =
Cochran
b =
Shapiro
c =
Clark
d =
Darden
```

The function **mmonoff** provides a utility for converting `'on'` and `'off'` property values of many Handle Graphics property names to the logical values of 1 and 0 respectively. The reverse conversion also works. For example,

```
» yn = get(0,'ShowHiddenHandles')
yn =
off
» mmonoff(yn)
ans =
     0
» yn = get(0,'AutomaticFileUpdates')
yn =
on
» mmonoff(yn)
ans =
     1
» mmonoff(1) % go back
ans =
on
» mmonoff(0) % go back
ans =
off
```

The function **mmprintf** is similar to the MATLAB function num2str. The function num2str always maintains the dimensions of the input array, whereas mmprintf forms a string matrix with the i^{th} row being the string representation of the i^{th} element of the input. For example,

```
» x = linspace(0,2*pi,6)
x =
            0       1.2566       2.5133       3.7699       5.0265       6.2832
» num2str(x)
ans =
0     1.2566     2.5133     3.7699     5.0265     6.2832
» size(ans)
ans =
     1    56
» mmprintf(x)
ans =
0
1.2566
2.5133
3.7699
5.0265
6.2832
» size(ans)
ans =
     6     6
» num2str(x') % do what mmprintf does
ans =
     0
```

```
1.2566
2.5133
3.7699
5.0265
6.2832
» size(ans)
ans =
     6     6
```

Because the function `mmprintf` is simpler, it is also significantly faster than `num2str`:

```
» tic, for i=1:100, mmprintf(x); end, toc
elapsed_time =
      0.59949
» x = x'; convert to column
» tic, for i=1:100, num2str(x); end, toc
elapsed_time =
       2.0586
```

Finally, a format string of the user's choice can be applied to the data by calling `mmprintf` as:

```
» mmprintf('%10.2f',x)
ans =
      0.00
      1.26
      2.51
      3.77
      5.03
      6.28
```

Chapter 11: Time Functions

The *Mastering MATLAB Toolbox* function associated with the material in the *Time Functions* chapter is shown in the table below.

Function	Description
`mmpause`	Pause for a specified (fractional) time.

The MATLAB command `pause` allows one to pause operations for integer values of seconds. In those cases, where fractional pause amounts are desired, the function `mmpause(x)` pauses operations for $0 \leq x \leq 10$ seconds.

Chapter 16: Numerical Linear Algebra

The *Mastering MATLAB Toolbox* function associated with the material in the *Numerical Linear Algebra* chapter is shown in the table below.

Function	Description
`mmrwls`	Recursive weighted least squares.

In addition to the least squares solutions provided by the forward and backward slashes / and \, `lscov`, and `nnls`, the *Mastering MATLAB Toolbox* provides the function **`mmrwls`**. This function allows one to apply diagonal weighting to the least squares problem, which is equivalent to multiplying each row by some weight constant. The greater the weight given to a row, the less the error that is tolerated in that row when finding the solution.

In addition to equation weighting, `mmrwls` also supports recursive least squares. That is, if new data becomes available, there is no need to recompute the least squares solution. One need only update it to reflect the new data and its weight. This recursive solution is beneficial when the data is gathered experimentally over an extended time, in which case one can watch the solution change as the experiment proceeds. For example consider fitting a second order polynomial to cos(x) over the range $-\pi \le x \le \pi$:

```
» x = linspace(-pi,pi,11);
» y = cos(x);  % create data for fitting
» [p,P] = mmrwls([1 1 1],1e-6)  % initialize polynomial p and matrix P
p =
     1
     1
     1
P =
     1000000           0           0
           0     1000000           0
           0           0     1000000
» for i = 1:11 % add data one point at a time
    [p,P]=mmrwls(y(i),[x(i)^2 x(i) 1],1,p,P);
    error_norm=norm(y-polyval(p,x)) % display error
end
error_norm =
    0.7899
error_norm =
    0.7883
error_norm =
    0.7891
error_norm =
```

```
    0.7894
error_norm =
    0.7882
error_norm =
    0.7882
error_norm =
    0.7891
error_norm =
    0.7886
error_norm =
    0.7883
error_norm =
    0.7901
error_norm =
    0.7878
```

For complete details of the features of `mmrwls`, see on-line help.

Chapter 17: Data Analysis

The *Mastering MATLAB Toolbox* functions associated with the material in the *Data Analysis* chapter are shown in the table below.

Function	Description
mmax	Matrix maximum value.
mmin	Matrix minimum value.
mmpeaks	Find indices of relative extremes.

The functions **mmax** and **mmin** find the maximum and minimum values in a matrix respectively. Both functions optionally returns the indices of all maxima or minima. For example,

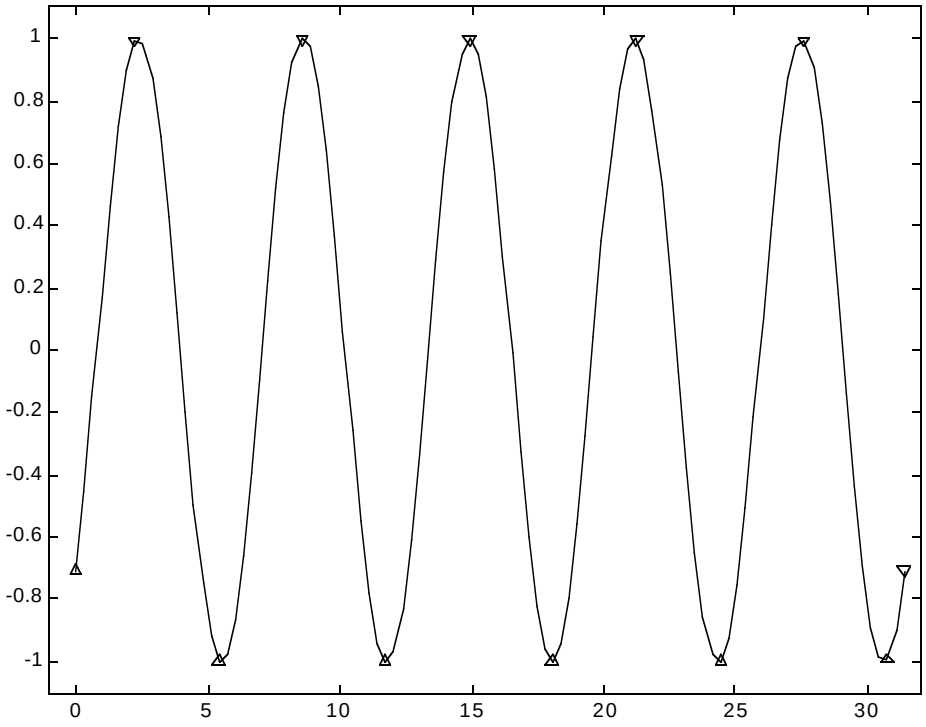
```
» x = randn(4) % test data
x =
      -1.0565      0.21932      -1.0106      0.59128
       1.4151      -0.9219      0.61446      -0.6436
     -0.80509      -2.1707      0.50774      0.38034
      0.52874    -0.059188       1.6924      -1.0091
» mmax(x) % find maximum
ans =
       1.6924
» [mx,r,c] = mmax(x) % find max and indices
mx =
       1.6924
r =
     4
c =
     3
» mmin(x) % find minimum
ans =
      -2.1707
» [mn,r,c] = mmin(x) % find min and indices
mn =
      -2.1707
r =
     3
c =
     2
» [mx,r,c] = mmax(eye(3)) % find max and all indices
mx =
     1
r =
     1
```

```
     2
     3
c =
     1
     2
     3
```

The function **mmpeaks** searches through an array and finds the indices of ***relative*** maxima, minima, or both. For example,

```
» x = linspace(0,10*pi);
» y = sin(x-pi/4);
» imax = mmpeaks(y,'max')
imax =
     8    28    48    68    88   100
» imin = mmpeaks(y,'min')
imin =
     1    18    38    58    78    98
» plot(x,y,x(imax),y(imax),'v',x(imin),y(imin),'^')
» axis([-1 32 -1.1 1.1])
» title('Figure MM.1: MMPEAKS Example.')
```

Figure MM.1: MMPEAKS Example.



Chapter 18: Polynomials

The *Mastering MATLAB Toolbox* functions associated with the material in the *Polynomials* chapter are shown in the table below.

Function	Description
mmp2str	Polynomial vector to string conversion.
mmpadd	Polynomial addition.
mmpintrp	Inverse interpolate polynomial.
mmpoly	Real polynomial construction.
mmpscale	Scale polynomial.
mmpshift	Shift polynomial.
mmpsim	Polynomial simplification, strip leading zero terms.
mmrwpfit	Recursive weighted polynomial curve fitting.
mm2dpfit	Two-dimensional polynomial curve fitting.
mm2dpval	Two-dimensional polynomial evaluation.
mm2dpstr	Two-dimensional polynomial vector to string
mmp2pm	Polynomial to polynomial matrix conversion.
mmpm2p	Polynomial matrix to polynomial conversion.
mmpmder	Polynomial matrix derivative.
mmpmfit	Polynomial matrix curve fitting.
mmpmint	Polynomial matrix integration.
mmpmsel	Select subset of a Polynomial matrix.
mmpmeval	Polynomial matrix evaluation.

The functions in this table appear in three groups. The first seven functions operate on conventional polynomial vectors. The three in the middle of the table apply to two-dimensional polynomials, and the last eight functions apply to polynomial matrices, **which in this context are simply matrices with each row containing a conventional polynomial vector**.

The function **mmp2str** converts a polynomial vector into a character string:

```
» p = [2 -3 0 sqrt(2)]
p =
            2           -3            0       1.4142
» mmp2str(p) % standard conversion
ans =
2x^3 - 3x^2 + 1.414
```

```
» mmp2str(p,'s') % use variable s instead of x
ans =
2s^3 - 3s^2 + 1.414
» mmp2str(p,1) % pull out nonunity first term
ans =
2*(x^3 - 1.5x^2 + 0.7071)
» mmp2str(p,'z',1) % use z and pull out term
ans =
2*(z^3 - 1.5z^2 + 0.7071)
```

The function **mmpadd** performs polynomial addition as demonstrated in the chapter on polynomials.

The function **mmpintrp** inverse interpolates a polynomial. That is if *y=p(x)* describes a polynomial *p*, `mmpintrp(p,yi)` returns those values of *x* where *yi=p(x)*. For example,

```
» p % recall p
p =
           2           -3            0       1.4142
» mmpintrp(p,0) % where p(x) crosses y=0
ans =
     -0.58268
» mmpintrp(p,1) % it crosses y=1 in three places
ans =
       1.3933
      0.44256
     -0.33587
```

The function **mmpoly** makes it easy to construct polynomials with real-valued coefficients. For example,

```
» mmpoly(-1,-2,-3)      % (x+1)(x+2)(x+3)
ans =
     1     6    11     6
» mmpoly(-1,[-2;-3;-4]) % (x+1)(x+2)(x+3)(x+4)
ans =
     1    10    35    50    24
» mmpoly(0,-1,[1 2 2])  % (x)(x+1)(x^2 + 2x + 2)
ans =
     1     3     4     2     0
» mmpoly(0,-1+2j)       % (x)(x+1+2j)(x+1-2j)
ans =
     1     2     5     0
» roots(ans)
ans =
           0
```

```
          -1 +           2i
          -1 -           2i
» mmpoly(-1,'+',[1 2 2],'-',-1,-2) % (x+1)+(x^2+2x+2)-(x+1)(x+2)
ans =
     1
```

Scalar arguments and column vectors are considered root locations and row vectors are considered polynomials to `mmpoly`. If only one component of a complex root is provided, its complex conjugate is created. Simple addition and subtraction of terms can also be done as shown in the last example.

The function **`mmpscale`** performs simple polynomial scaling, i.e., given *y=p(x)*, `mmpscale(p,b)` creates the polynomial *p(x/b)*. This scaling scales the roots of the polynomial *p* by *b*. For example,

```
» p % recall earlier polynomial
p =
            2           -3            0       1.4142
» roots(p)
ans =
      1.0413 +     0.35937i
      1.0413 -     0.35937i
    -0.58268
» ps = mmpscale(p,2) % scale by two
ps =
         0.25        -0.75            0       1.4142
» roots(ps) % roots are scaled by two
ans =
      2.0827 +     0.71875i
      2.0827 -     0.71875i
     -1.1654
```

The function **`mmpshift`** shifts a polynomial. That is, given *y=p(x)*, `mmpshift(p,a)` creates the polynomial *p(x+a)*. This shifting subtracts *a* from the roots of *p(x)*. For example,

```
» p % recall p(x)
p =
            2           -3            0       1.4142
» roots(p)
ans =
      1.0413 +     0.35937i
      1.0413 -     0.35937i
    -0.58268
» ps = mmpshift(p,1) % shift it by one
ps =
```

```
            2            3            0      0.41421
» roots(ps) % shift subtract one from roots
ans =
     -1.5827
    0.041341 +     0.35937i
    0.041341 -     0.35937i
```

The function **mmpsim** is a utility function used by other polynomial functions in the *Mastering MATLAB Toolbox* to perform polynomial simplification, e.g.,

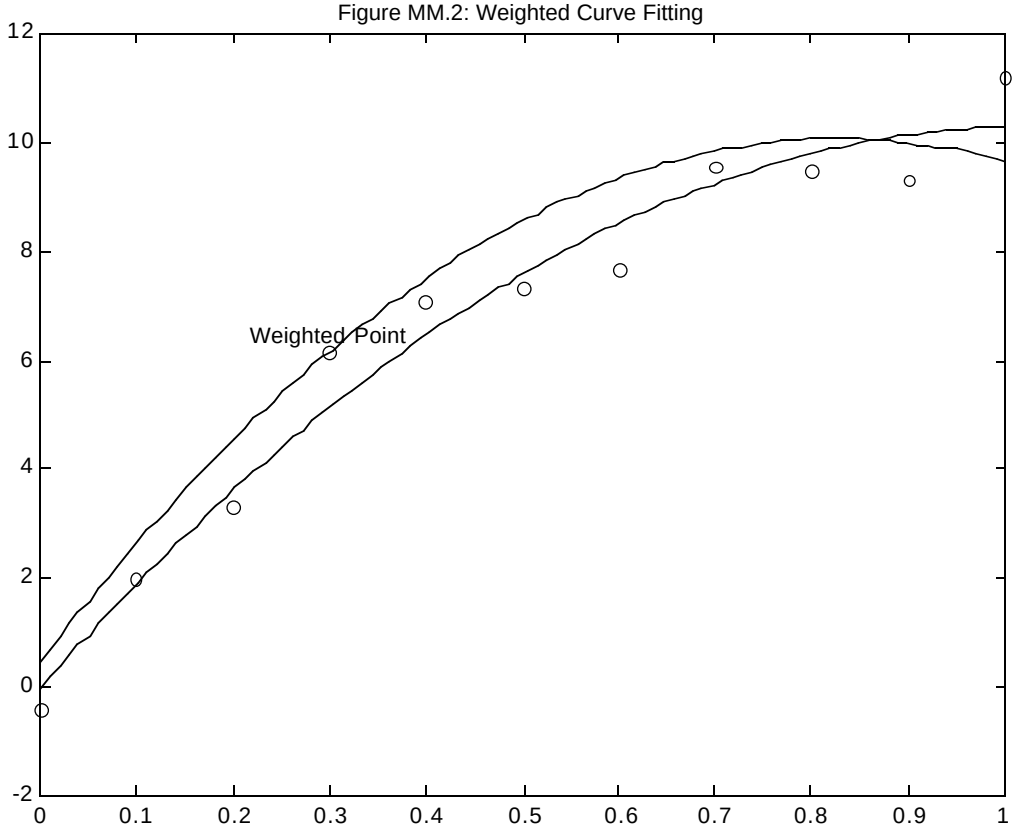
```
» a = [eps 1 2 -eps 4] % example polynomial
a =
   2.2204e-16            1            2  -2.2204e-16            4
» mmpsim(a)
ans =
     1     2     0     4
```

As shown above, `mmpsim` deletes zero or negligible leading coefficients and changes insignificant interior terms to zero. `mmpsim(a,tol)` allows the user to specify the tolerance used.

The function `polyfit` in MATLAB fits a polynomial to a data set assuming that the error at each point has equal weight. Moreover, it takes the entire data set at a time. The *Mastering MATLAB Toolbox* function **mmrwpfit**, on the other hand, allows one to weight the data points differently and to update a fit when new data is available. For example,

```
» x = [0 .1 .2 .3 .4 .5 .6 .7 .8 .9 1]; % data from Polynomial chapter
» y = [-.447 1.978 3.28 6.16 7.08 7.34 7.66 9.56 9.48 9.30 11.2];

» p = polyfit(x,y,2); % standard fit

» w = ones(size(x)); % default weights
» w(4) = 100; % give 4th point lots of weight (small error)
» pw = mmrwpfit(x,y,2,w); % weighted fit

» xi = linspace(0,1); % evaluate both fits and plot

» yi = polyval(p,xi);
» yw = polyval(pw,xi);
» plot(x,y,'o',xi,yi,xi,yw)
» text(x(4)-.08,y(4)+.3,'Weighted Point')
» title('Figure MM.2: Weighted Curve Fitting')
```

Figure MM.2: Weighted Curve Fitting



Note in the figure how the weighted fit appears to pass through the point with the highest weight. The weighting vector must be the same size as the data vectors and contain only positive values. For assistance using the recursive features of `mmrwpfit`, see on-line help.

The next three functions in the above table, **`mm2dpfit`**, **`mm2dpval`**, and **`mm2dpstr`**, consider two-dimensional polynomials of the form

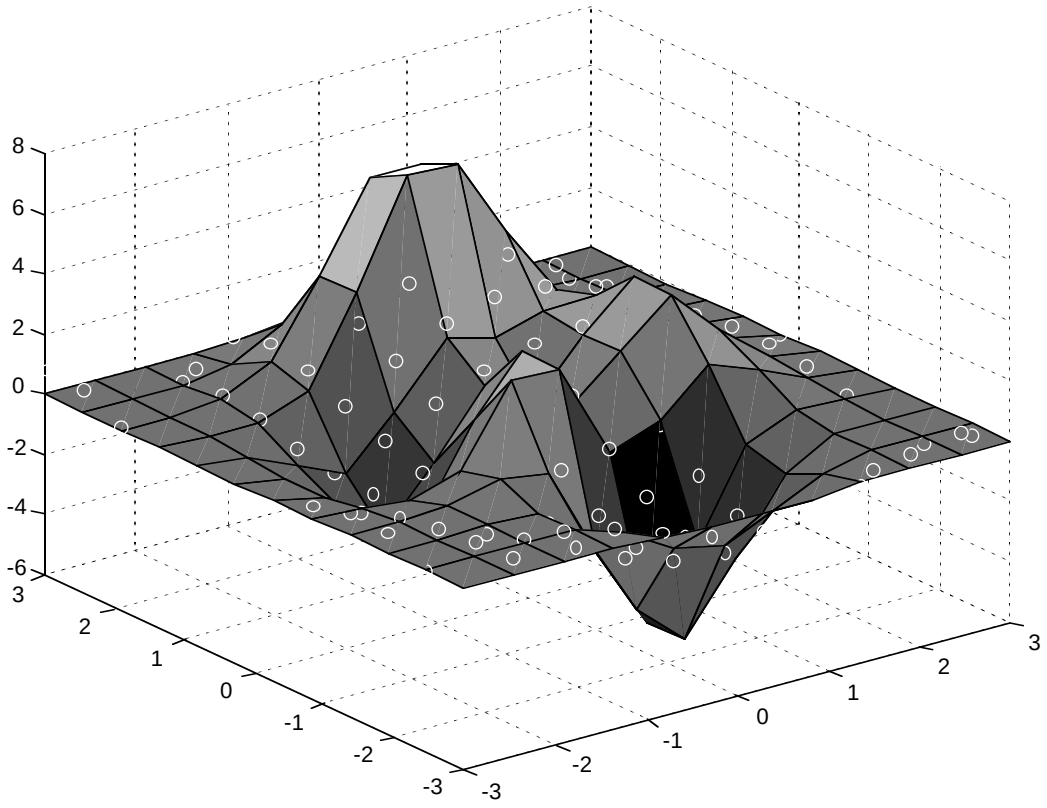
$$P(x,y) = p_{0,0} + p_{1,0}x + p_{0,1}y + p_{1,1}xy + p_{2,0}x^2 + p_{0,2}y^2 + \cdots$$

That is, the two-dimensional polynomial contains terms that appear in the Taylor series expansion of a function of two variables. These two-dimensional polynomial functions can handle different polynomial orders in the two dimensions. If N_X and N_Y are the x- and y-orders respectively, then the resulting polynomial contains all terms $p_{i,j} \cdot x^i y^j$ for $i \le N_X$, $j \le N_Y$, and $i+j \le \max(N_X, N_Y)$.

To illustrate these functions consider the following example:

```
» [x,y,z] = peaks(12);
» P = mm2dpfit(x,y,z,4,5); % fit the polynomial

» mm2dpstr(P) % show polynomial as a string
ans =
1.219 +0.8012x^1 -0.3119x^2 -0.07956x^3 +0.01848x^4
 +2.078y^1 +0.1509x^1y^1 -0.5662x^2y^1 -0.01175x^3y^1 +0.02905x^4y^1
 -0.1282y^2 -0.2246x^1y^2 +0.01735x^2y^2 +0.01375x^3y^2
 -0.1179y^3 -0.009357x^1y^3 +0.03376x^2y^3
 +0.0006174y^4 +0.01201x^1y^4
 -0.009318y^5

» zz = mm2dpval(P,x,y); % evaluate fit at original points

» surf(x,y,z) % show plot
» hold on
» plot3(x,y,zz,'ow') % plot 'o' at interpolated points
» hold off
» title('Figure MM.3: 2-D Polynomial Fitting')
```

Figure MM.3: 2-D Polynomial Fitting



In the above example, $N_X = 4$ and $N_Y = 5$. In addition, the output of `mm2dpfit` is encoded with the polynomial structure and therefore is best viewed using `mm2dpstr`. As shown, the two-dimensional polynomial functions work well with the *plaid* data arrays created by `meshgrid` and with standard three-dimensional plotting routines.

The remaining functions considered in this section apply to polynomial matrices. ***As described earlier, a polynomial matrix contains polynomial vectors in each of its rows.*** The benefit of creating a polynomial matrix is that all polynomials can be evaluated simultaneously using matrix algebra. Perhaps more importantly, when one wants to fit polynomials to multiple data sets all defined over the same range, all polynomials can be found simultaneously using a single MATLAB statement. These features dramatically speed execution by eliminating repeated calls to the MATLAB functions `polyval` and `polyfit`.

Consider the following example that uses polynomial matrix functions:

```
» p1 = [1 2 3 4]; % create sample polynomials
» p2 = fliplr(p1);
» p3 = 3;
» p4 = mmpoly(-1,-2,-3,-4);
» PM = mmp2pm(p1,p2,p3,p4) % create polynomial matrix
PM =
     0     1     2     3     4
     0     4     3     2     1
     0     0     0     0     3
     1    10    35    50    24
» [p1,p2,p3,p4] = mmpm2p(PM) % extract polynomials back out
p1 =
     1     2     3     4
p2 =
     4     3     2     1
p3 =
     3
p4 =
     1    10    35    50    24
» [p5,p6] = mmpm2p(PM,[4;2]) % extract only 4th and 2nd
p5 =
     1    10    35    50    24
p6 =
     4     3     2     1
```

Differentiation and integration are possible:

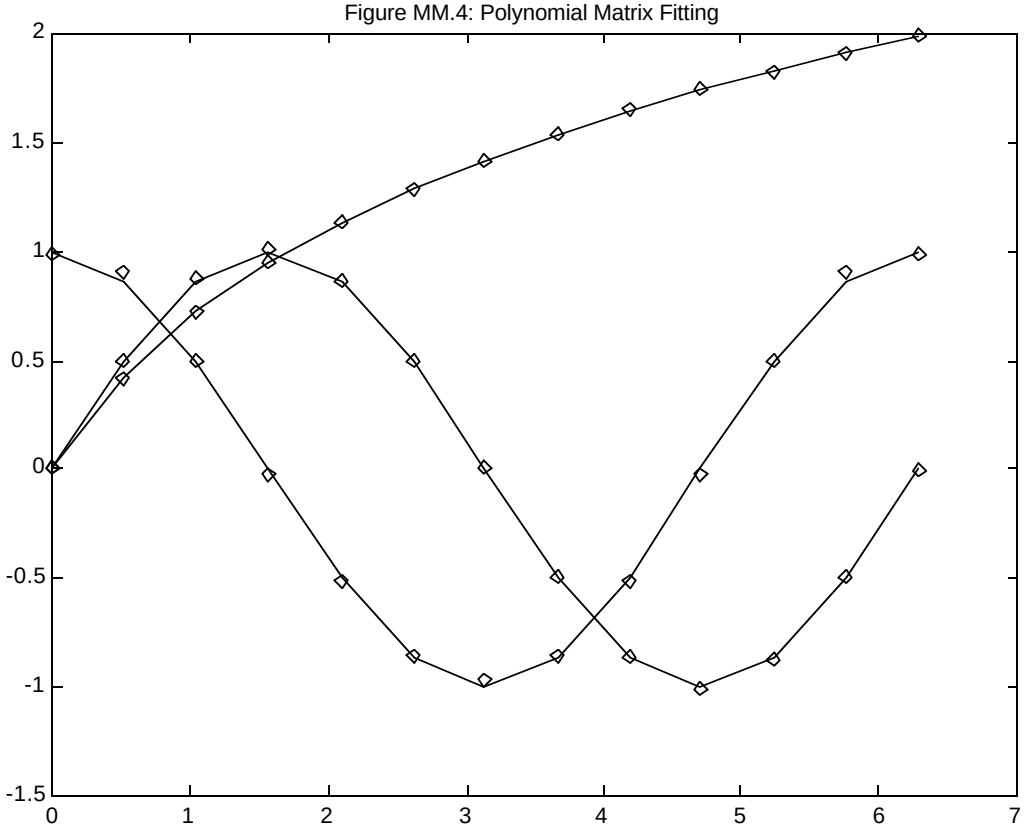
```
» PMD = mmpmder(PM) % compute derivative of all polys
```

```
PMD =
     0     3     4     3
     0    12     6     2
     0     0     0     0
     4    30    70    50
» PMI = mmpmint(PMD) % integrate the derivative to get PM back?
PMI =
     0     1     2     3     0
     0     4     3     2     0
     0     0     0     0     0
     1    10    35    50     0
» PM % note constant terms are zero above by default
PM =
     0     1     2     3     4
     0     4     3     2     1
     0     0     0     0     3
     1    10    35    50    24
» PMI = mmpmint(PMD,[4 1 3 24]) % do it again, provide constant terms
PMI =
     0     1     2     3     4
     0     4     3     2     1
     0     0     0     0     3
     1    10    35    50    24
» p7 = mmpmint(PM,sqrt(2),1) % integrate only 1st poly
p7 =
         0.25      0.66667          1.5            4       1.4142
» p8 = mmpmint(p1,sqrt(2)) % works also for plain polynomial
p8 =
         0.25      0.66667          1.5            4       1.4142
```

Now try curve fitting and evaluation:

```
» x = linspace(0,2*pi,13)'; % use transpose to make column
» Y = [sin(x) cos(x) log(x+1)]; % column oriented data

» P = mmpmfit(x,Y,5); % fit all three columns to 5th order
» Yf = mmpmeval(P,x); % evaluate all three columns
» plot(x,Y,x,Yf,'d')
» title('Figure MM.4: Polynomial Matrix Fitting')
```

Figure MM.4: Polynomial Matrix Fitting



Chapter 19: Interpolation

The *Mastering MATLAB Toolbox* functions associated with the material in the *Interpolation* chapter are shown in the table below.

Function	Description
mmgetpt	Get graphical point with interpolation.
mminterp	1-D monotonic linear interpolation.
mmsearch	1-D NON-monotonic linear interpolation.

The function **mmgetpt** linearly interpolates a data set at a single point and computes the slope at the point by fitting a quadratic to the nearest three points and evaluating the quadratic at the selected point. The primary purpose for mmgetpt is to interpolate graphical data dynamically with the mouse. Typical usage of mmgetpt is:

```
xy = get(gca,'CurrentPoint'); % get cursor position
xdata = get(Hl,'Xdata'); % Hl is handle to plotted line
ydata = get(H,'Ydata'); % line data

[x,y,yp] = mmgetpt(xy(1,1:2),xdata,ydata)
```

Here, x and y are the linearly interpolated points and yp is the estimated slope at (x,y). By default, the interpolation is done with respect to the *x*-axis data. mmgetpt(xy(1,1:2),xdata,ydata,0) interpolates with respect to the *y*-axis data.

The function mmgetpt is used in the functions mmprobe and mmginput.

In MATLAB version 4, the function table1 was used to perform linear interpolation of a data set. table1 was so terribly slow that the function **mminterp** was developed. mminterp is typically an order of magnitude faster than table1. Now in MATLAB version 5, the function interp1 uses a fast algorithm for interpolation. It is so much better than the old table1 that mminterp is no longer faster than what MATLAB provides in interp1. Nevertheless, mminterp is retained in the *Mastering MATLAB Toolbox* because it beats interp1 by up to 50% in the case where the independent variable is not linearly spaced. When the independent variable is linearly spaced interp1 beats mminterp by up to 50%. In addition, mminterp is much faster than interp1 when there is a single interpolant. The primary calling syntax for mminterp matches that for interp1. Since interp1 was covered in the chapter on interpolation, there is no need to rehash the material here using mminterp. If you compare interp1 to mminterp you will see that they are based on two entirely different algorithms.

One of the weaknesses of the interpolation tools provided in MATLAB is that they require the independent variable to be monotonic. When the independent variable

is not monotonic, the interpolation algorithms simply don't know where to look for interpolated values. When the point to be found is limited to a single point, the function **mmsearch** finds all linear interpolates when the independent variable is not monotonic. For example,

```
» x = linspace(0,5*pi); % create data
» y = sin(x);           % y is not monotonic!
» mmsearch(y,x,0.5)     % interpolate y to find x when y=0.5
ans =
      0.52521
       2.6162
       6.8074
       8.9006
       13.092
       15.183
```

Chapter 20: Cubic Splines

The *Mastering MATLAB Toolbox* functions associated with the material in the *Cubic Splines* chapter are shown in the table below.

Function	Description
mmsparea	Cubic spline area.
mmspcut	Extract or cut out part of a cubic spline.
mmspdata	Cubic spline data extraction.
mmspder	Cubic spline derivative interpolation.
mmspii	Inverse interpolate cubic spline.
mmspint	Cubic spline integral interpolation.
mmspline	Cubic spline construction with method choice.
mmspxtrm	Cubic spline extremes.

The cubic spline functions provided in the *Mastering MATLAB Toolbox* augment those already available in MATLAB. As described in the chapter on cubic splines, the functions `mmspint` and `mmspder` find the cubic spline representations of the integral and derivative of a data set. In addition to these functions, the functions `mmsparea`, `mmspii`, and `mmspxtrm` provide analysis functions, `mmspcut` and `mmspdata` are utility functions, and `mmspline` is an alternative to the function `spline` with many added features.

The function **`mmsparea`** is the spline equivalent to `quad` and `quad8`. `mmsparea` computes the area under a spline between two endpoints. For example,

```
» x = (0:.1:1)*2*pi;   % create rough data
» y = sin(x);

» mmsparea(x,y,0,2*pi) % area from 0 to 2*pi
ans =
   2.7756e-17

» pp = spline(x,y);    % find pp form first
» mmsparea(pp,0,2*pi)  % compute area using pp form
ans =
   2.7756e-17
```

The function **`mmspcut`** allows one to extract a spline out of another by specifying a new restricted range for the independent variable *x*:

```
» pp2 = mmspcut(pp,pi,2*pi); % extract last half
```

The function **mmspdata** returns the original data used to construct a spline from its piecewise polynomial vector:

```
» [x,y] = mmspdata(pp2) % get data from short spline
x =
       3.1416       3.7699       4.3982       5.0265       5.6549       6.2832
y =
            0     -0.58779     -0.95106     -0.95106     -0.58779            0
```

The function **mmspii** performs inverse interpolation. That is, given a single *y* value it finds all *x* values such that *y=s(x)* where *s(x)* is the piecewise polynomial. For example,

```
» xi = mmspii(pp,0.5) % points where spline of sine is equal to 0.5
xi =
      0.52256        2.618
» sin(xi) % how far off is spline representation?
ans =
       0.4991      0.49999
```

The function **mmspxtrm** finds the zero slope points of a spline representation. For example,

```
» [xi,yi,PN] = mmspxtrm(pp)
xi =
       1.5704       4.7127
yi =
      0.99974     -0.99974
PN =
     1    -1
```

Here the vector `PN` denotes –1 for minimum points and +1 for maximum points.

The function **mmspline** is a generalization of the function spline. `mmspline` offers a variety of features as described by a portion of its help text:

```
MMSPLINE Cubic Spline Construction with Method Choice. (MM)
  MMSPLINE(X,Y,METHOD,P) computes the cubic spline interpolant
  from the data in X and Y, using the method in METHOD and optional
  parameter vector P. MMSPLINE returns the piecewise polynomial
  pp-form to be evaluated with PPVAL.

   METHOD     Description
  'clamped'   P is vector of slopes y' at the two endpoints
  'natural'   no P, curvature y'' is zero at the two endpoints
  'extrap'    no P, extrapolated spline, same as function SPLINE
  'parabolic' no P, first and last splines are parabolic not cubic
  'curvature' P is vector of curvatures y'' at the two endpoints
  'periodic'  no P, y' and y'' match at the two endpoints
  'aperiodic' no P, y' opposite at endpoints, y'' equal at endpoints
```

As is apparent from the above help text `mmspline` allows one to specify a variety of end conditions for the spline to meet. Although not documented above, one can also specify interior break points where the slope is not continuous. This function always returns a piecewise polynomial representation that matches that used by `spline`. As a result, it can be manipulated and evaluated with the same set of spline functions. In particular, `ppval` is used to evaluate the spline returned by `mmspline`. The function `mmspline` is interesting to tinker around with to see how various end point conditions affect the shape of the spline.

Chapter 21: Fourier Analysis

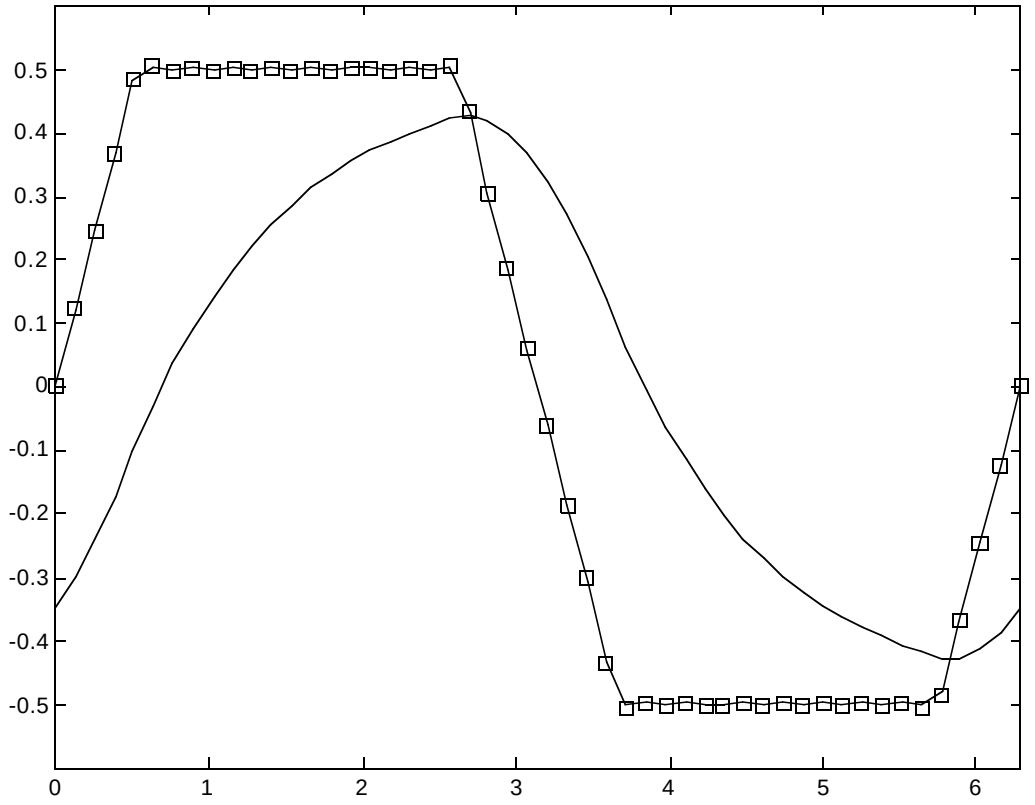
The *Mastering MATLAB Toolbox* functions associated with the material in the *Fourier Analysis* chapter are shown in the table below.

Function	Description
fshelp	Help for Fourier series.
fsangle	Angle between two Fourier series.
fsderiv	Fourier series of time derivative.
fsdelay	Add time delay to a Fourier series.
fseval	Fourier series function evaluation.
ffind	Find Fourier series approximation.
fsform	Fourier series format conversion.
fsharm	Fourier series harmonic component
fsintgrl	Fourier series of time integral.
fsmsv	Fourier series mean square value.
fspeak	Fourier series peak time value.
fspf	Fourier series power factor computation.
fsprod	Fourier series of a product of time
fsresize	Resize a Fourier series.
fsresp	Fourier series linear system response.
fsround	Round Fourier series coefficients.
fssize	Highest harmonic in a Fourier series.
fssum	Fourier series of a sum of time functions.
fssym	Enforce symmetry constraints.
fstable	Generate common Fourier series.
fsthd	Total harmonic distortion of a Fourier
mmfftbin	FFT bin frequencies.
mmfftpfc	FFT positive frequency components.
mmftfind	Find Fourier transform approximation.
mmwindow	Generate window functions.

The above functions form a fairly comprehensive suite of functions for the manipulation of Fourier series. They all follow from the basic analysis demonstrated in the chapter on Fourier series. Given the large number of functions available, only a few are demonstrated in the example below.

```
» kn = fstable('trapezoid',21,2/3); % pick F.S. from function table
» t = [0 30 150 210 330 360]*pi/360; % build the trapezoid as well
» f = [0 1   1  -1  -1   0]/2;

» fn = fsfind(t,f,21); % find its F.S.
» max(abs(kn-fn)) % maximum error
ans =
   0.00032216
» yn = fsresp(1,[1 1],kn); % pass F.S. through H(s)=1/(s+1)

» ti = linspace(0,2*pi,50); % let's look at everything!
» ki = fseval(kn,ti);
» fi = fseval(fn,ti);
» yi = fseval(yn,ti);
» plot(ti,ki,ti,fi,'s',ti,yi)
» axis([0,2*pi,-.6,.6])
» title('Figure MM.5: Fourier Series Tools')
```

Figure MM.5: Fourier Series Tools



The last four functions in the table above are signal processing utilities. `mmftfind` encapsulates the procedure for approximating the Fourier transform of a continuous time function as demonstrated in the chapter on Fourier analysis. The function `mmwindow` generates twelve common window functions that are useful for Fourier series smoothing and general signal processing applications.

Chapter 22: Optimization

The *Mastering MATLAB Toolbox* functions associated with the material in the *Optimization* chapter are shown in the table below.

Function	Description
`mmfminc`	Minimization with inequality constraints.
`mmfminu`	Minimize a function of several variables.
`mmfsolve`	Solve a set of nonlinear equations.
`mmlceval`	Evaluate a linear combination of functions.
`mmlcfit`	Curve fit to a linear combination of functions.
`mmnlfit`	Nonlinear curve fitting.
`mmnlfit2`	2-D nonlinear curve fitting.
`mmsneval`	Simple nonlinear curve fit evaluation.
`mmsnfit`	Simple nonlinear curve fit by transformation.

The functions in the above table provide optimization tools for a variety of optimization tasks. The functions may not be as rigorous or powerful as those found in the *Optimization Toolbox*, but they are well suited for common curve fitting and optimization tasks. Of the functions in the above table, `mmfminu` and `mmfsolve` are rigorous optimization functions roughly on par with those in the *Optimization Toolbox*.

The functions **`mmsnfit`** and **`mmsneval`** handle simple nonlinear curvefitting problems. They follow the approach taken by typical numerical analysis texts whereby the input data is transformed in such a way that the problem becomes a linear regression problem that is easily solved with `polyfit(x,y,1)`. Afterwards the polynomial coefficients are transformed into those required for the nonlinear fit. To demonstrate these functions, consider the function

$$y = Ax^b$$

where A and b are unknown parameters. If we define $Y = \log(y)$ and $X = \log(x)$, then this equation becomes

$$Y = p_1 X + p_2$$

where $p_1 = b$ and $p_2 = \log(A)$. The modified equation is now a first order polynomial in X. So to fit data to the nonlinear expression, compute X and Y, find the polynomial coefficients $[p_1\ p_2]$ that produces the least squared error in the data, then

return $b = p_1$ and $A = \exp(p_2)$.

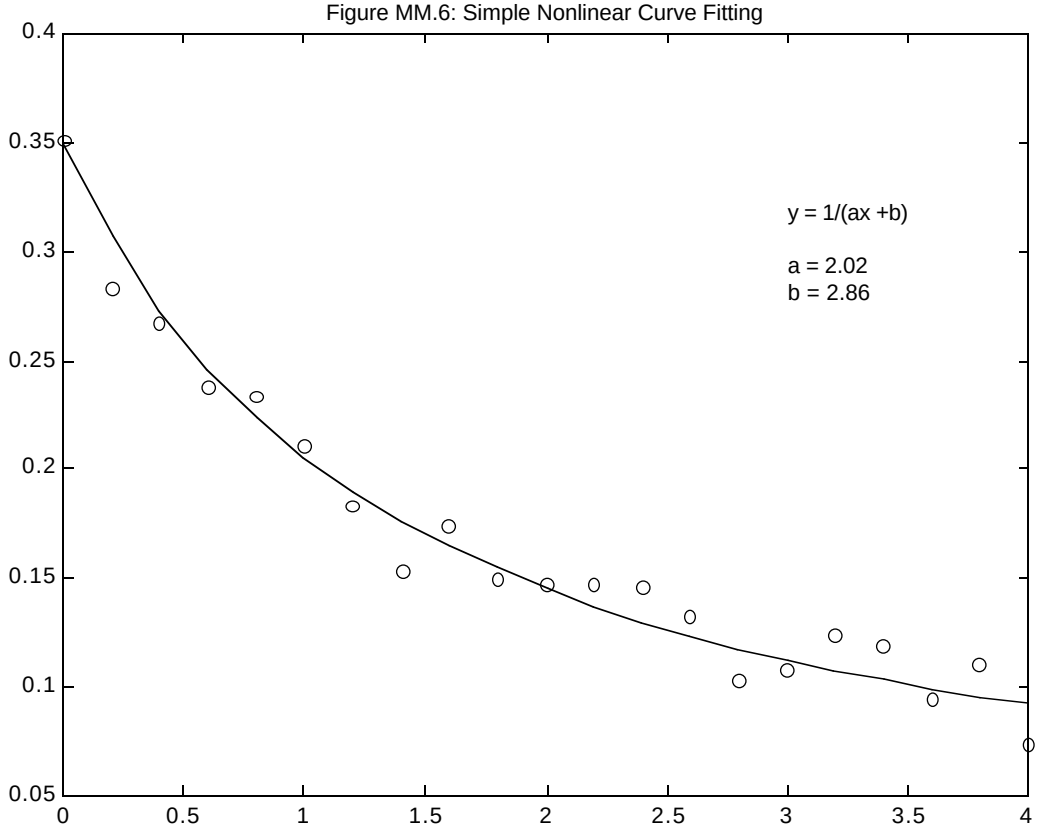
As described by its help text, the function `mmsnfit` offers nine different functions to fit including simple linear regression.

```
» help mmsnfit
 MMSNFIT Simple Nonlinear Curve Fit by Transformation. (MM)
  [A,B]=MMSNFIT(x,y,N) performs least squares curve fitting by
  linearizing the data, fitting it to a straight line,
  then inverting the linearization. A and B are the desired
  curve fit coefficients.
  N identifies the function to be fit:

   N | function     N | function        N | function
   --+-----------   ------------------  ---------------
   0 | y= A*x+B     3 | y= 1/(A*x+B)    6 | y= A*x^B
   1 | y= (A/x)+B   4 | y= 1/(A*x+B)^2  7 | y= A*log(x)+B
   2 | y= A/(x+B)   5 | y= x/(A*x+B)    8 | y= A*exp(B*x)
                                       9 | y= A*x*exp(B*x)
  See also MMSNEVAL.
```

The function `mmsneval` simplifies the function evaluation process. To illustrate `mmsnfit` and `mmsneval`, consider the following example:

```
» x = 0:.2:4; % create some data
» y = 1./(2*x+3) + mmrand(0,.02,size(x));
» [a,b] = mmsnfit(x,y,3); % curve fit
» yf = mmsneval(x,a,b,3); % evaluate

» plot(x,y,'o',x,yf) % plot
» text(3,.3,sprintf('y = 1/(ax +b)\n\na = %.3g\nb = %.3g',a,b))
» title('Figure MM.6: Simple Nonlinear Curve Fitting')
```

Figure MM.6: Simple Nonlinear Curve Fitting



The functions **mmlcfit** and **mmlceval** apply to a different problem. In the polynomial curve fitting problem, the unknown coefficients appear linearly, thereby allowing standard least squares to be used to find the coefficients. Rather than using the power series basis functions, x^0, x^1, x^2, ... as is done in polynomial curve fitting, the function mmlcfit allows the user to specify the basis functions. To illustrate the use of mmlcfit and its companion evaluation function mmlceval, consider the following example:

```
» x = linspace(0,2*pi); % example data
» y = 2*sin(x+pi/4);

» FUN = ['sin(x)';'cos(x)'] % functions to be fit
» p = mmlcfit(x,y,FUN)      % curve fit
p =
       1.4142       1.4142
» yf = mmlceval(x,p,FUN); % evaluate the fit functions
» max(abs(y-yf)) % maximum error
ans =
```

```
   2.2204e-15
```

This example demonstrates the identity . While the above example used a string matrix to define the functions to be fit, it is also possible to use a cell array of strings or a function M-file to define the functions to be fit.

When `mmsnfit` and `mmlsfit` do not work, the functions **`mmnlfit`** and **`mmnlfit2`** provides tools for general curve fitting. These functions use the MATLAB function `fmins` and are therefore limited by the capabilities of `fmins`. As you may recall, `fmins` implements the Nelder-Mead simplex algorithm. As such, it does not require or approximate a gradient and therefore often requires a large number of iterations to converge. At the same time, this algorithm is fairly robust since it is not easily fooled by functions that are not smooth. To illustrate the use of these functions consider the `mmsnfit` example yet again. In this case the function to be fit must be written as `y=fun(x,p)` where `p` is the unknown parameter vector:

```
function y=testfun1(x,p)
% TESTFUN1(X,P) test function for MMNLFIT

y=1./(p(1)*x+p(2));
```

Then we're ready to create the data and call `mmnlfit`:

```
» x = 0:.2:4; % create some data
» y = 1./(2*x+3) + mmrand(0,.02,size(x));

» [p,e] = mmnlfit(x,y,'testfun1',[1 1])
p =
       2.0393       2.9429
e =
    0.0031948
```

Using `mmnlfit` produced better results because the nonlinear transformation used in `mmsnfit` nonlinearly weights the data. This produces a nonlinearly weighted least squares estimate, that in this case is not as good as that returned by `mmnlfit`.

The function **`mmfminc`** utilizes `fmins` as well. In this case, minimization with inequality constraints is implemented by a simple penalty function approach. That is, the problem

$$\min_{\boldsymbol{x}} f(\boldsymbol{x}) \quad \text{subject to} \quad \boldsymbol{g}(\boldsymbol{x}) \le 0$$

is handled by solving the unconstrained optimization problem

$$\min_{\boldsymbol{x}} \left\{ f(\boldsymbol{x}) + K \sum_i \left(g_i(\boldsymbol{x}) > 0\right) g_i^2(\boldsymbol{x}) \right\}$$

where K is a positive constant, and $(g_i(\boldsymbol{x}) > 0)$ is a logical statement that is zero when False and one when True. Written in this way, the summation term is a penalty function that increases as the degree of constraint violation increases.

In `mmfminc`, the parameter K takes on values from an increasing sequence, the minimization problem is solved for each value of K, and the result from each run is used as the initial condition for the next run.

To illustrate the use of `mmfminc` consider the following example where the function and constraints is defined by the M-file:

```
function [f,G]=testfun2(x)
% [f,G]=TESTFUN2(X) test function for MMFMINC
%
% f is a scalar function of vector x.
% G is a vector function whose (i)th component is the (i)th constraint

f=x(1)^2 + 5*x(2)^2 + abs(prod(x));

G(1)=x(1)+1;
G(2)=x(1)+x(2)+3;
```

Using the above function M-file, we call `mmfminc` as follows:

```
» k = [1 10 100 1e5 1e8];           % increasing K terms
» tol = [1e-3 1e-4 1e-5 1e-6 1e-8]; % tolerances as K increases
» xo = [-1; -1];
» x = mmfminc('testfun2',xo,k,tol)
x =
      -2.6964
     -0.30362
» [f,G] = testfun2(x)
f =
       8.5501
G =
      -1.6964   2.9066e-08
```

In this case the first constraint `x(1)<1` is met but not active. The second constraint is active and met within a factor of two of the tightest tolerance, `1e-8`. These results are typical of what can be expected with `mmfminc`. `mmfminc` cannot be used blindly. The K values, tolerances, number of runs to try, and the initial conditions all influence the

convergence and efficiency of `mmfminc`. Never be satisfied with first results. Try other `k`, `tol`, `xo` values before you accept a solution. Beware that there may more than one solution to your problem.

While `fmins` is robust, it evaluates the function to be minimized many times. For problems where function evaluation is expensive, the function `fminu` in the *Optimization Toolbox* is appropriate. If you do not have access to this *Toolbox*, **`mmfminu`** in the *Mastering MATLAB Toolbox* implements the same BFGS QuasiNewton method. While not illustrated below, `mmfminu` offers a number of usersettable parameters that determine tolerances, analytic or numerical gradient calculations, and initial Hessian matrix values. For further information regarding these optional parameters, see on-line help.

To illustrate use of `mmfminu`, reconsider the banana function:

```
function f=banana(x)
% Rosenbrock's banana function

f=100*(x(2)-x(1)^2)^2 + (1-x(1))^2;
```

Comparing `fmins`, `fminu`, and `mmfminu` using the banana function leads to:

```
» xo = [-1.9;2]; % initial conditions

» tic, [xs,opts] = fmins('banana',xo); toc
elapsed_time =
      0.16972
» opts(10) % number of function evaluations
ans =
   212
» banana(xs) % function value at solution
ans =
   1.0314e-09

» tic, [xu,optu] = fminu('banana',xo); toc % In Optimization Toolbox!
elapsed_time =
      0.27602
» optu(10) % number of function evaluations
ans =
   159
» banana(xu) % function value at solution
ans =
   2.0213e-09

» global MMFMINU_ITER % MMFMINU internal function counter
```

```
» tic, xm = mmfminu('banana',xo); toc
elapsed_time =
      0.13673
» MMFMINU_ITER % number of function evaluations
MMFMINU_ITER =
   158
» banana(xm) % function value at solution
ans =
   2.0055e-11
```

For this simple example, all three functions worked well, with the *Optimization Toolbox* function being the slowest of the three for this example.

The MATLAB function `fzero` finds a zero of a function of a scalar variable. In *n*dimensional space, the *Optimization Toolbox* function `fsolve` does the same thing. As an alternative to this function, the Mastering MATLAB Toolbox offers **`mmfsolve`**. `mmfsolve` implements the same numerical algorithm used in the original `fsolve` that appeared in MATLAB version 3.5. This original algorithm was replaced in MATLAB version 4 and later by an entirely different `fsolve` function in the *Optimization Toolbox*.

To illustrate the use of `mmfsolve` consider the following example based on the equations described in the M-file:

```
function f=testfun3(x)
%TESTFUN3(X) test function for MMFSOLVE

f(1,1) = sin(x(1)) + cos(x(1)) - 1; % f MUST be a column
f(2,1) = 3*x(1)*(x(2)-.5);
```

```
» [xf,tc] = mmfsolve('testfun3',[1;1])
xf =
       1.5708
          0.5
tc =
     1
» testfun3(xf)
ans =
  -2.7247e-08
   2.8502e-08
```

In addition to the default calling syntax shown above, a number of optional parameters including tolerances and Jacobian calculation approaches can be specified. See on-line help for more information.

Chapter 23: Integration and Differentiation

The *Mastering MATLAB Toolbox* functions associated with the material in the *Integration and Differentiation* chapter are shown in the table below.

Function	Description
mmcurve	Length along a plane curve.
mmderiv	Approximate derivative using weighted central differences.
mmintgrl	Cumulative integral using Simpson's rule.
mmvolume	Cumulative volume integral using trapezoidal rule.

Formerly, the function **mmintgrl** duplicated the functionality of cumtrapz for vector and matrix inputs. However, in this edition of the *Mastering MATLAB Toolbox*, mmintgrl uses Simpson's rule rather than the trapezoidal rule implemented by cumtrapz. As a result of the higher order fit provided by Simpson's rule, mmintgrl is usually much more accurate than cumtrapz. For example,

```
» t = linspace(0,pi,11); % create simple data to integrate
» y = sin(t);

» yt = cumtrapz(t,y); % MATLAB's trapezoidal rule
» ys = mmintgrl(t,y); % Mastering MATLAB's Simpson's rule
» yi = 1 - cos(t);    % Actual cumulative integral

» trap_error = norm(yt-yi) % norm of trapezoidal rule error
trap_error =
     0.033967

» simp_error = norm(ys-yi) % norm of Simpson's rule error
simp_error =
   0.00094122
```

Notice the nearly two orders of magnitude decrease in error between cumtrapz and mmintgrl.

The use of the function diff was demonstrated for finding approximate forward- and backward-difference derivative approximations. If was also shown that central differences were much more accurate. Because of this, the function **mmderiv** was written for the *Mastering MATLAB Toolbox*. This function utilizes central differences with quadratic polynomial fitting at the endpoints. Consider the following simple example:

```
» x = linspace(0,2*pi,31); % create data
» y = sin(x);
» yd = mmderiv(x,y);  % approximate derivative
» max(abs(yd-cos(x))) % worst error
ans =
     0.014398
```

In addition to working on vectors, `mmderiv`, `mmintgrl`, and `cumtrapz` work on column-oriented data arrays as well where `x` is a vector and `y` is an array having as many rows as `length(x)`. In addition, `mmderiv` and `mmintgrl` also work when the independent variable is NOT linearly spaced.

The function **`mmvolume`** is the two-dimensional equivalent to `cumtrapz`. It computes the cumulative integral under a surface using the trapezoidal rule and data in standard `meshgrid` format:

```
» [x,y,z] = peaks;
» V = mmvolume(x,y,z);
```

In the above, the integral is assumed to be zero along `x=min(x)` and `y=min(y)`. See on-line help for more information.

Chapter 24: Ordinary Differential Equations

The *Mastering MATLAB Toolbox* functions associated with the material in the *Ordinary Differential Equations* chapter are shown in the table below.

Function	Description
mmlsim	Linear system simulation using modess.
mmode45	ODE solution using modess.
mmode45p	ODE plotted solution using modess.
mmodechi	ODE cubic Hermite interpolation.
mmodeini	Initialize ODE parameters for modess.
mmodess	Single step ODE solution, 4-5 order Runge-Kutta.

Given the powerful MATLAB ODE suite, you'd think that there couldn't be any ordinary differential equation functions in the *Mastering MATLAB Toolbox*. Well there are. The primary function provided is mmodess, which is a modern, 4–5th order, Runge-Kutta algorithm. The reason for including this function is that it only takes a single time step each time it is called. In other words by using mmodess, you get closer access to the integration process than you can get with ode45 and the other ODE suite functions. The other functions mmlsim, mmode45, and mmode45p use mmodess in simple ways to create useful outer drivers. Given the relative simplicity of mmodess, and the examples demonstrated by the drivers provided, you can write your own driver functions that implement exactly what you want. As such, mmodess is particularly useful when the incremental solution of a set of differential equation is a part of a larger problem that needs to be solved. The help text for mmodess is

```
» help mmodess
 MMODESS Single Step ODE Solution, 4-5th Order. (MM)
  [T,Y,YP,STATS]=MMODESS('yprime',t,y,yp) integrates the
  system of first order differential equations computed in the
  M-file yprime(t,y) from the time point t where y = y(t)
  and yp = yprime(t,y). yprime(t,y) must return a column vector.

  [T,Y,YP,STATS] are the results at the integrator chosen time T>t,
  where Y=Y(T), YP=yprime(T,Y), and STATS is a vector of statistics:
  STATS=[IERR FAIL ORDER] where IERR identifies the variable Y(IERR)
  which dominated the error in the step, FAIL is the number of failed steps
  encountered in this integration step and ORDER is the order
  of the accepted solution. Order is 2, 3, or 5.
  T is scalar, Y and YP are ROW vectors.

  Integration parameters must be initialized by MMODEINI.
  Typical usage:
```

```
mmodeini default
t=0;           % initial time
y=[y1;y2;...]; % initial condition column vector
yp=feval('yprime',t,y); % initial derivatives
while test
    [t,y,yp]=mmodess('yprime',t,y,yp);
    % process data
end
```

Because `mmodess` takes only a single step, it gets its parameters through the global variables it shares with `mmodeini`. This latter function provides a gateway for setting parameters and communicating with `mmodess` during the integration process. The function `mmodechi` provides the ability to interpolate the ODE solution between output points. While this cubic Hermite polynomial approach is good, it is not as accurate as the continuous extension methods used in the ODE suite. At the same time, interpolating with `mmodechi` is more accurate than any of the algorithms in `interp1` because it makes explicit use of the known slopes at the integration output points.

As an example, consider `mmode45`, which calls `mmodess` internally. `mmode45` is called with the same syntax as `ode45`, provided the parameters for `mmodess` are initialized by a call to `mmodeini`:

```
» tspan = [0 20];  % set parameters
» yo = [2;0];
» [t,y] = ode45('vdpol',tspan,yo); % call MATLAB ode45

» mmodeini default % set defaults for mmode45/mmodess
» [t,y] = mmode45('vdpol',tspan,yo); % integrate
```

For more complete information regarding the ODE tools in the Mastering MATLAB Toolbox, see on-line help. In particular, see `mmodeini` and `mmodess`.

Chapter 26: 2-D Graphics

The *Mastering MATLAB Toolbox* functions associated with the material in the *2-D Graphics* chapter are shown in the table below.

Function	Description
`mmarrow`	Plot moveable arrows on current axes.
`mmfill`	Fill plot of area between two curves.
`mmpolar`	Linear or logarithmic polar coordinate plot.
`mmplot2`	Plot two Y arrays vs. one X with right side axis label.
`mmplotc`	2-D plot with an ASCII character marker at data points.
`mmploti`	Incremental 2-D line plotting.
`mmplotz`	Plot with axes drawn through zero.
`mmprobe`	Probe data on 2-D axis using mouse.
`mmginput`	Graphical input using mouse.
`mmgui`	Double-click activation of plotting GUIs.
`mmsaxes`	Set axes specifications using a GUI.
`mmscolor`	Set RGB specifications using a GUI.
`mmsfont`	Set font characteristics using a GUI.
`mmsline`	Set line specifications using a GUI.
`mmedit`	Edit axes text using mouse.
`mmtext`	Place and drag text using mouse.
`mmtile`	Tile figure windows on screen.
`mmzoom`	Simple 2-D zoom-in function.

Of all the functions in the *Mastering MATLAB Toolbox*, the functions provided for this chapter and the next are the most powerful and elaborate.

The function **`mmarrow`** places arrow(s) on the current graph that can be dragged, rotated, and stretched. During this process, numerical feedback about arrow position and shape is provided dynamically on the plot at the mouse pointer. Clearly this is difficult to show on the printed page of this book. To check out `mmarrow`, type `mmarrow(3)` after creating a linear plot. Three arrows of default size are created in the center of the current axes. Clicking on the arrow tip and dragging, drags the arrow under the mouse pointer, giving dynamic feedback on arrow tip position as you move. Clicking on an arrow tail with the right mouse button (Shift-click on the Macintosh) holds the arrow tip down and rotates and stretches the tail under the mouse pointer, with dynamic feedback provided about arrow length and angle. In

addition to default characteristics, arrows can be created with numerous setable properties. See on-line help for more information.

The function **mmfill** duplicates many of the features of the MATLAB function area. In addition, it allows one to fill an area between two curves. See on-line help for further information.

The function **mmpolar** represents a significant improvement over the polar function in MATLAB. mmpolar offers a substantial set of parameters that can be queried and set as well as a dynamic cursor and polar grid for exploring graphical data. Basic functionality of mmpolar is provided by the following passage taken from its help text.

```
MMPOLAR Linear or Logarithmic Polar Coordinate Plot. (MM)
  Hl=MMPOLAR(THETA,RHO,S1,S2,...) generates a polar coordinate plot using
  the angle vector THETA in radians and radius array RHO and optional
  linestyle specifications S1,S2, etc. THETA must be a vector, but RHO can
  be either a vector the same size as THETA or a matrix having length(THETA)
  rows. If RHO is a vector only S1 is valid. If RHO is a matrix the (i)th
  column is plotted using Si. Empty Si applies the default linestyle to the
  (i)th column of RHO. If present, Hl contains handles to the plotted lines.

  Hl=MMPOLAR(THETA,RHO,'log',S1,S2,...) generates a polar plot with a
  log10 RHO axis.

  MMPOLAR on {off} enables {disables} feedback of mouse position over
  the current plot with the mouse button down. Clicking in the figure window
  outside the axis rectangle also disables mouse feedback.
```

mmpolar offers an extensive set of user settable parameters that are described in its complete help text. See on-line help for more information.

The functions **mmplot2** represents a significant improvement over the MATLAB function plotyy. mmplot2 creates a single axes object so axis, zoom, legend, subplot, and mmzoom work. On the other hand, plotyy creates two axes, one on top of the other, making axes modification commands fail or work erratically. As described by the initial portion of its help text, mmplot2 plots two data sets, one with a left hand side y-axis and the other with a right hand side y-axis:

```
MMPLOT2 Plot 2 Data Arrays on a Common X Axis. (MM)
  MMPLOT2(X,Y,S1,Ylim,R,S2,Rlim)
  plots   Y vs. X with Y labeled on the left hand side, and
  plots   R vs. X with R labeled on the right hand side.
  X must be a vector, whereas Y and R can be matrices.

  Ylim = [Ymin Ymax] and Rlim = [Rmin Rmax] are optional row
```

```
  vectors specifying Y and R axis limits.
  S1 and S2 are optional color and line/marker style for Y and R data.
  Default: S1 = '-' (solid lines) S2 = ':' (dotted lines).
```

`mmplot2` offers other features that are described in its complete help text.

The function **`mmploti`** is an incremental plotting routine. That is, it adds line segments to an existing plot as data is made available. This routine is demonstrated in the function `mmode45p`, where it is used to show the ODE solution as it is being generated. `mmploti` is useful in any application where graphical feedback is desired as a solution evolves. The help text for `mmploti` is:

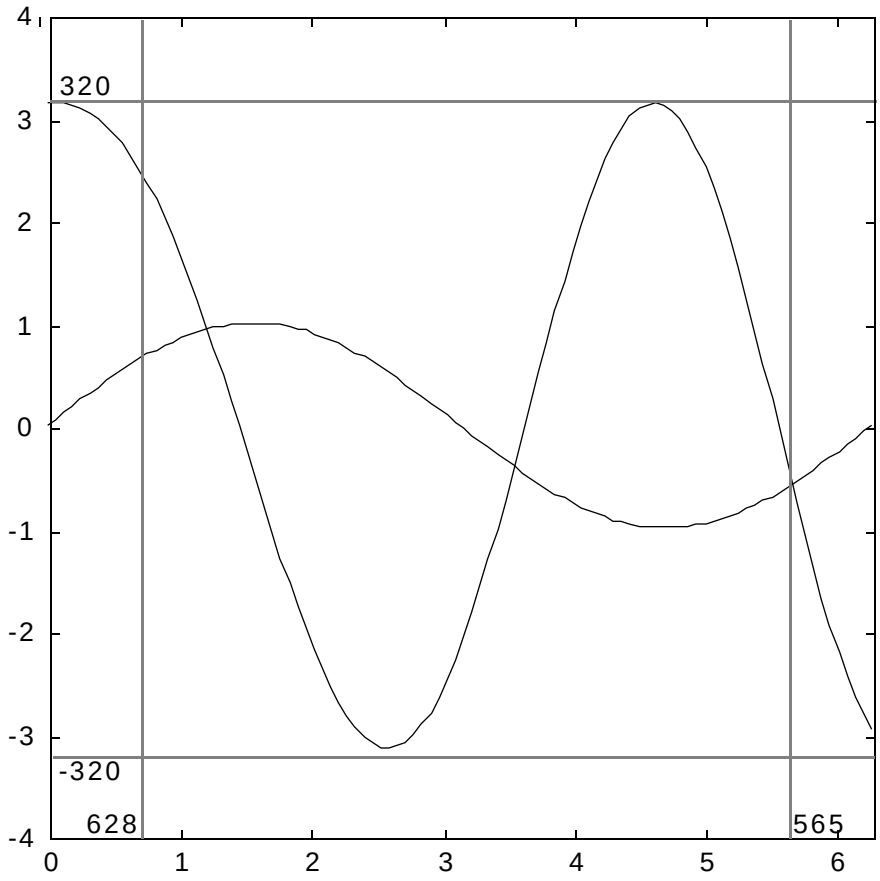
```
» help mmploti
 MMPLOTI Incremental 2D Line Plotting. (MM)
  Ha=MMPLOTI(V) initializes a 2D plot for future plotting
  by using the axis limits given in V=[Xmin Xmax Ymin Ymax].
  Ha is a handle to the created axes.

  Hl=MMPLOTI(x,Y) plots Y versus x, appending data to any
  prior calls. x is a vector and Y is a vector or column
  oriented data matrix having as many rows as length(x).
  Hl contains handles to the created or appended lines.

  MMPLOTI('done') or MMPLOTI done  refreshes the final plot,
  turns hold off and auto scales the axis.
```

There are times when it is beneficial to move the axis lines and tick marks to the center of the plot rather than their standard left and bottom locations. The function **`mmplotz`** performs this feat. `mmplotz` is called using the same syntax and features as the standard `plot` function.

Maybe you've used a digital oscilloscope that allows you to probe and analyze data on the screen using marker lines and a cursor. The function **`mmprobe`** provides this feature for 2-D line plots. `mmprobe` places two vertical and two horizontal marker lines on the current axes. These lines can be dragged around the screen to measure attributes of the underlying data. In addition, clicking and dragging on a data line reveals important details about the plotted data itself and pressing several keys on the keyboard automates a number of common measurement actions.

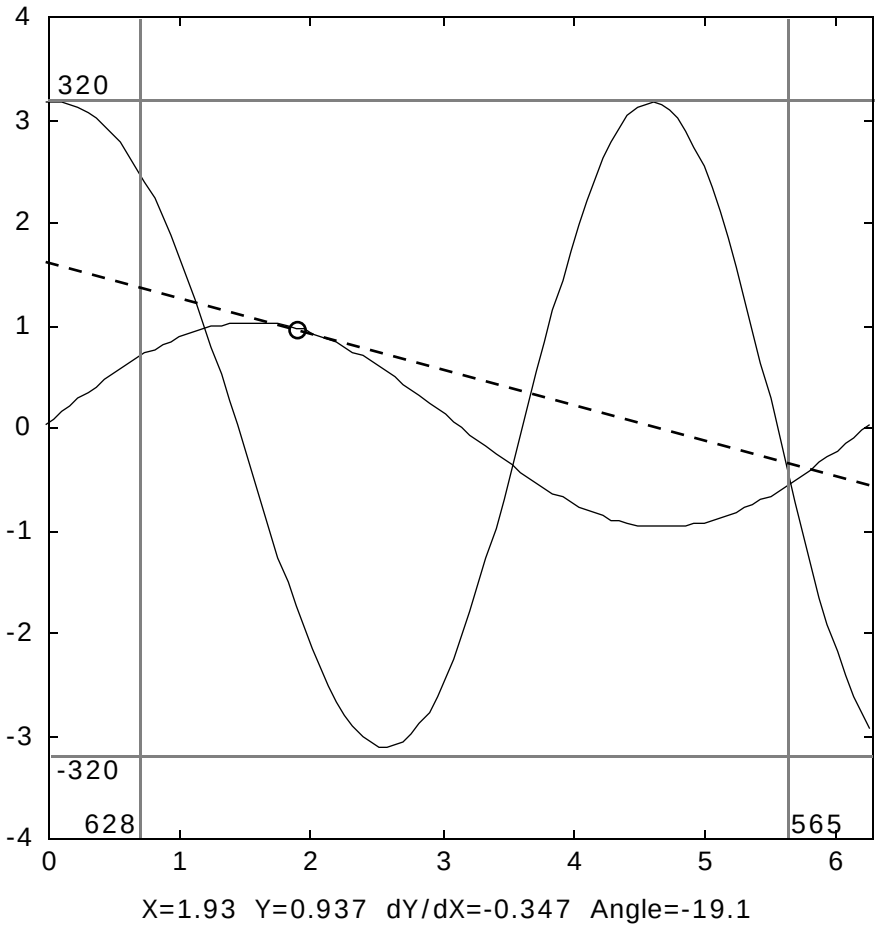
The following plots demonstrate the utility of `mmprobe`. In Figure MM.7, information about the markers is shown in the `xlabel` location. In Figure MM.8, the tangent line at the mouse-selected data point is shown as is information about the data point in the `xlabel` location. In the figures, the mouse can be used to move the markers and select different data points for analysis. `mmprobe` offers many other features for analyzing plotted data. See on-line help for further information.

Figure MM.7: Marker Use in MMPROBE

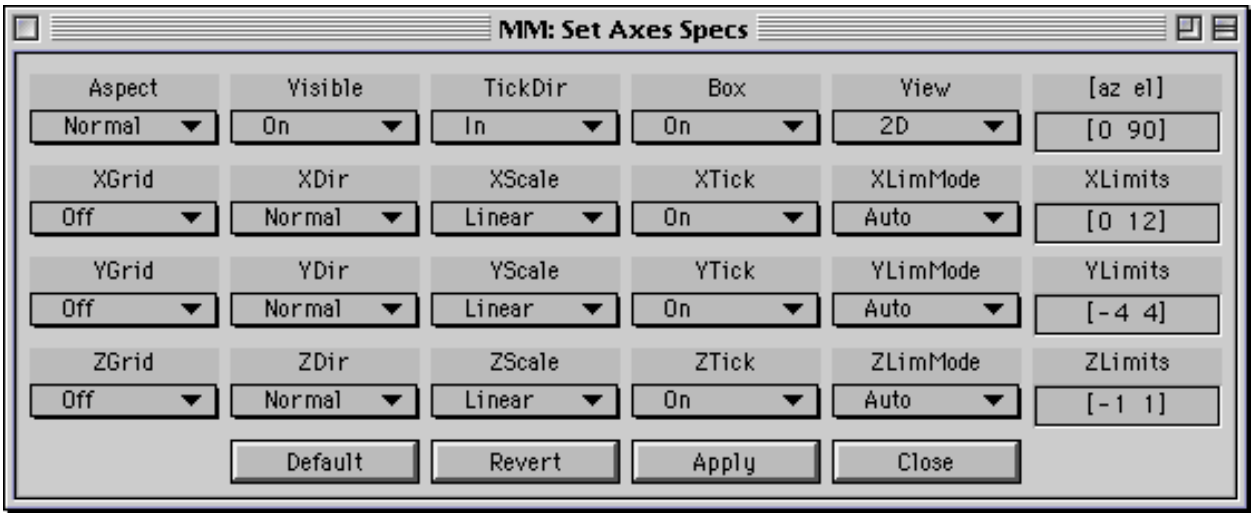


dY=6.4 dX=5.03 dY/dX=1.27 1/dX=0.199 Diag=8.14 Angle=51.9

Figure MM. 8: Data Probing in MMPROBE



Of the remaining functions listed in the table at the beginning of this chapter, the two most important are **`mmsaxes`** and **`mmsline`**. These two functions are GUIs for setting *axes* and *line* characteristics respectively. Each provides a GUI for interactive modification of the current plot. As can be seen from their screen images below, they allow the user to choose all important aspects of *axes* and *lines* using popup menus and editable text boxes.


MM: Set Axes Specs
Aspect
Normal
Visible
On
TickDir
In
Box
On
View
2D
[az el]
[0 90]
XGrid
Off
XDir
Normal
XScale
Linear
XTick
On
XLimMode
Auto
XLimits
[0 12]
YGrid
Off
YDir
Normal
YScale
Linear
YTick
On
YLimMode
Auto
YLimits
[-4 4]
ZGrid
Off
ZDir
Normal
ZScale
Linear
ZTick
On
ZLimMode
Auto
ZLimits
[-1 1]
Default
Revert
Apply
Close


MM: Set Line Specs
Visible
Style
Width
Marker
Size
Color
RGB
1
0.5
none
6
b
[0.0 0.0 1.0]
2
0.5
none
6
RGB->
[0.0 0.5 0.0]
Revert
Apply
Close

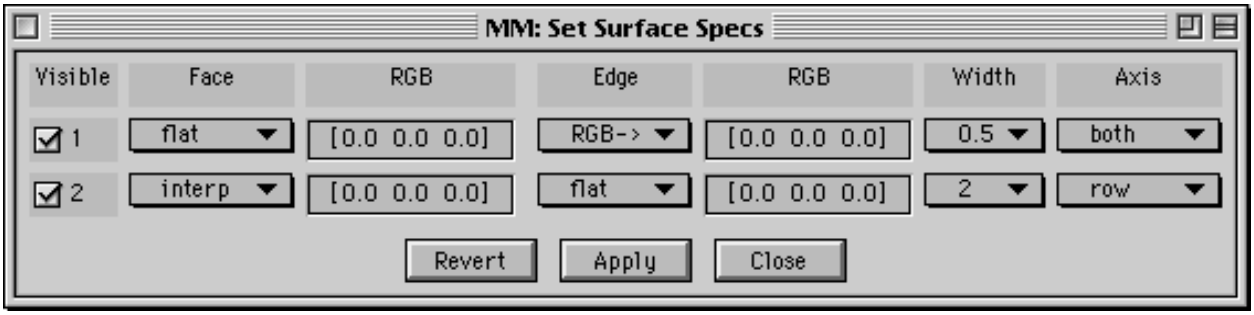
Chapter 27: 3-D Graphics

The *Mastering MATLAB Toolbox* functions associated with the material in the *3-D Graphics* chapter are shown in the table below.

Function	Description
`mmsaxes`	Set axes specifications using a GUI.
`mmhole`	Create hole in 3D graphics data.
`mminxy`	Minima of 3D data along X and Y axes.
`mmxtract`	Extract subset of 3D graphics data.
`mmssurf`	Set surface specifications using a GUI.
`mmsview`	Set azimuth and elevation using a GUI.
`mmzoom3`	Simple 3D X-Y plane zoom-in function.

The functions in the table above manipulate three-dimensional graphics. The function `mmsaxes`, which provides a GUI for setting axes characteristics, was shown in the last section. The functions `mmhole` and `mmxtract` are utilities for manipulating three-dimensional graphics data. `mmhole` automates the process illustrated in the 3-D Graphics chapter where the data in a rectangular region were set to `NaN`s to create a hole in a `mesh` or `surf` plot. `mmxtract` extracts a subset of graphics data so that `mesh` and `surf` plots can be zoomed.

The most powerful function in the above table is **`mmssurf`**, which is a GUI for setting surface properties. This function allows you to convert a `mesh` plot into a `surf` plot or vise-versa, or set any characteristics in between the two. A screen image of the GUI is shown below.

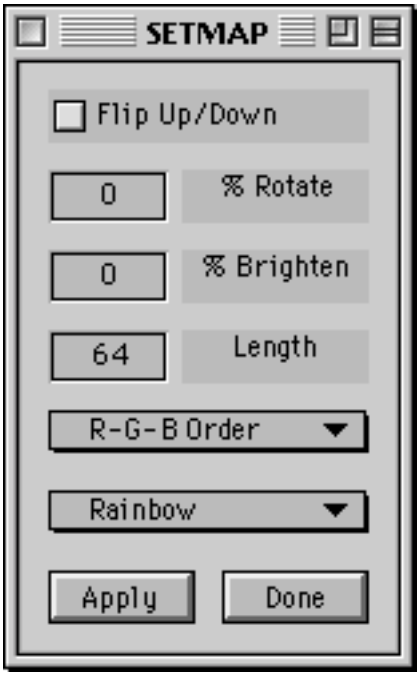


Chapter 28: Using Color and LIght

The *Mastering MATLAB Toolbox* functions associated with the material in the *Using Color and Light* chapter are shown in the table below.

Function	Description
mmap	Single color colormap.
mmsmap	Set figure colormap using a GUI.
rainbow	Colormap variant to hsv.

The three functions in this table are related to color maps. mmap creates a color map that contains one color varying from dark to light. The color map rainbow, is exactly what its name implies: a color map that mimics the colors in the rainbow. The function mmsmap is yet another GUI for setting and manipulating figure color maps interactively. A screen image of mmsmap is shown below. The lowest popup menu contains a list of twenty or so colormaps to choose from. The other popup menu allows you to rearrange the columns of the color map. That is, you can exchange red with blue, etc. The other features of mmsmap are self explanatory.

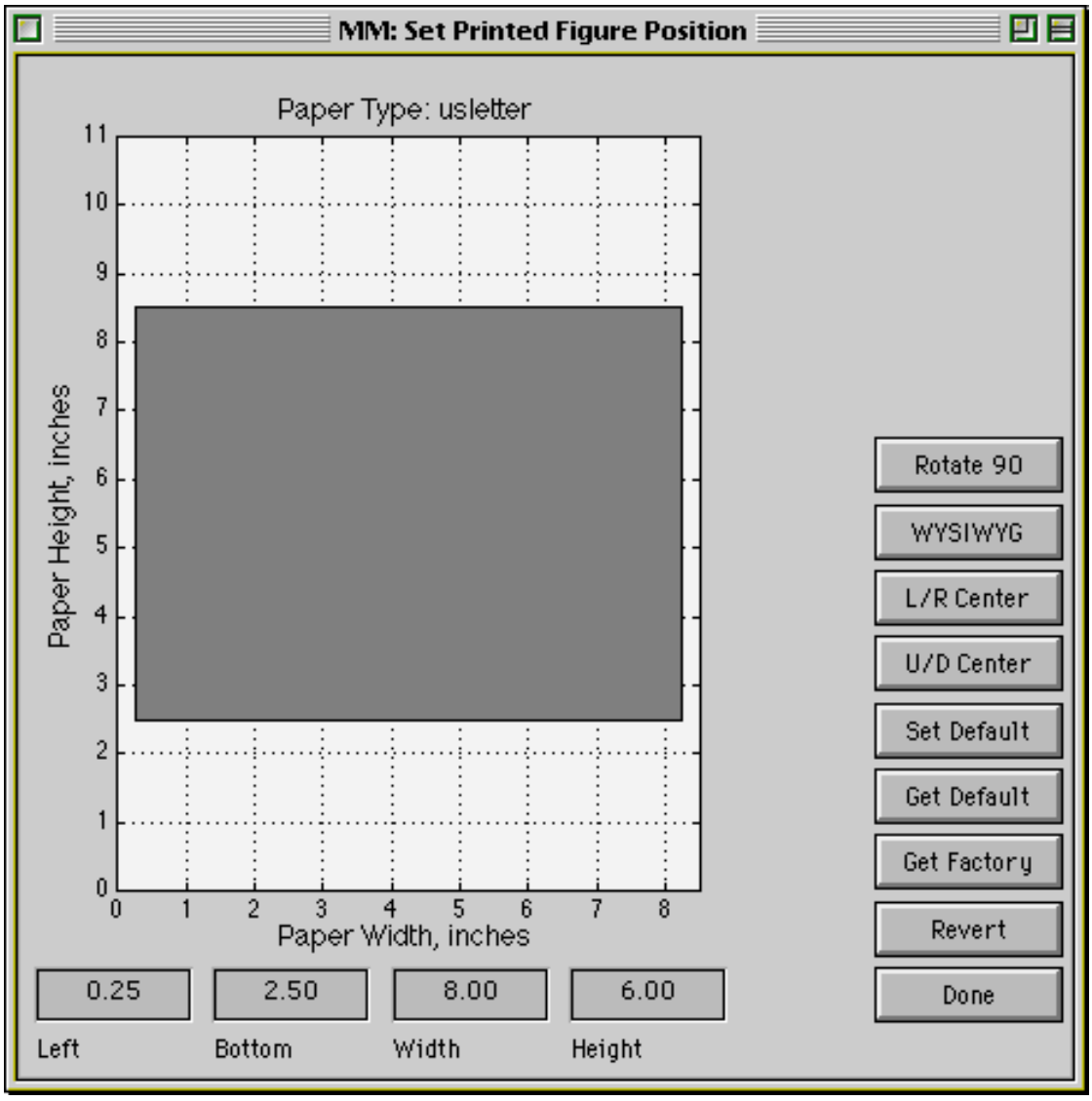


Chapter 30: Printing and Exporting Graphics

The *Mastering MATLAB Toolbox* functions associated with the material in the *Printing and Exporting Graphics* chapter are shown in the table below.

Function	Description
`mmspage`	Set figure paper position using a GUI.
`mmpaper`	Set default paper properties.

These two functions set printing characteristics. `mmpaper` is a simple *Command* window function for setting paper size and orientation. `mmspage` is a GUI similar to the MATLAB function `pagedlg` for controlling graphic placement on the printed page:



Chapter 31: Handle Graphics

The *Mastering MATLAB Toolbox* functions associated with the material in the *Handle Graphics* chapter are shown in the table below.

Function	Description
mmbox	Get position vector of a rubberband box.
mmedit	Edit axes text using mouse.
mmgca	Get current axes if it exists.
mmgcf	Get current figure if it exists.
mmget	Get multiple object properties.
mmgetpos	Get object position vector in specified units.
mmfitpos	Fit position within another object.
mmsetpos	Set position relative to another object.
mmsetptr	Set mouse pointer location over object or at axes data point.
mmgetsiz	Get font size in specified units.
mminrect	True when point is inside position rectangle.
mmpaper	Set default paper properties.
mmrgb	Color specification conversion and substitution.
mmtext	Place and drag text with mouse.
mmzap	Delete graphics object using mouse.

The above functions are primarily utilities that are useful within other functions that manipulate handle graphics properties. For example, `mmgetpos`, `mmfitpos`, and `mmsetpos` automate the process of getting, fitting, and setting position properties in specified units. The function `mmget` was originally written in MATLAB version 4 to facilitate getting multiple object properties. While the function `get` in version 5 has this capability, `mmget` is still useful because it effectively combines `get` with the function `deal` so that its output is individual variables rather than a cell array or structure. For example

```
[xlim,ylim,np] = mmget(gca,'Xlim','Ylim','NextPlot')
```

returns the corresponding property values for the listed property names.

Summary

This completes the tutorial description of the *Mastering MATLAB Toolbox*. For more complete information, register your copy of the Toolbox and receive the complete text of all M-files.