Hartmann Exploratory Data Analysis (EDA) Toolbox
Version 1, December 2004

Please let me know of bugs or places where the coding could be improved.

People who contributed to the toolbox: Christopher Assad (now at JPL), Brian Rasnow (now at Amgen)

Many of the functions are useful because they tell you the syntax for changing particular features of your plots without using the GUI. Avoiding the GUI is particularly useful when writing scripts to automatically generate figures for papers.

Functions I have found particularly useful are listed below. More details on each of the functions are provided further down in this document.

figX	Shortcut identically equal to ***figure(X);*** works for X = 1 to 21
findc	Finds the value in a vector whose value is closest to a given number, even if it is not an exact match
hline	Draws horizontal line(s) on the current plot
vline	Draws vertical line(s) on the current plot
xlog/xlin	Toggles the x axis between logarithmic and linear
ylog/ylin	Toggles the y axis between logarithmic and linear
axx	Rescales the x axis while leaving the y and z axes unchanged
axy	Rescales the y axis while leaving the x and z axes unchanged
rm	Removes an object (e.g., one of the traces on a plot) from the plot
pb	puts back an object onto the plot if you've removed it
lnX	changes the line width of an object to X , where X is 1,2,3,4,5,6,10
[Color]	Change the color of a selected object by typing the name of that color
bpfft	Bandpass filter (no phase distortion)
envelope	Find the envelope of a signal
pfft	Plot the power spectrum of the signal
msplit	Splits a vector up into windows of a given length, with option to omit some data segments
psplit	Finds the power spectrum of all of the rows of a matrix
envelope	Finds the envelope of a signal
plotv	Plots 2-vectors, equivalent to ***plot (data(: , 1), data(: , 2)); or plot (data(1 , :), data(2 , :));***
runningpspec	Finds the "running" power spectrum of a vector
runningxcov	Finds the "running" cross covariance of two vectors

Detailed description of functions

The toolbox contains the following folders:

- Filtering
- PlottingFncts
- Shortcuts
  - Axes
  - Colors
  - DisplayFormats
  - Figures
  - Fonts
  - LnWghts&Styles
- SimpleFunctions
- Statistics

/Filtering

bpfft — **Bandpass filter**

[d] = bpfft(v,rate,fo,f1)

Bandpass filter the vector v between fo and f1. Rate is the sampling rate. No phase distortions. bpfft implements the following three steps:

(1) Converts the signal to the frequency domain
(2) Sets the frequency components you want to remove to zero
(3) Takes the inverse fft to get the signal back to the time domain.

envelope — **Find the envelope of a signal**

[d]=envelope(vector)

Computes the envelope of a signal

mkf — **Make a frequency vector**

[f]=mkf(len,sr)

Given a signal of length n and sampling rate sr, generate appropriate frequency intervals (for example, for a power spectrum plot)

pfft — **Power spectrum of a vector**

[p,f] = pfft(vector,sr)

Plots the square of the absolute value of the fft of the signal (power spectrum), normalized by the length of the signal. Quick and easy – no windowing, no detrending. Do those operations before using pfft if you need to.

pfftm — **Power spectrum of a vector, remove the mean**

[p,f] = pfft(vector,sr)

Identical to pfft except that it removes the mean (DC component) of the signal.

psplit **Power spectrum of the rows of a matrix**

[p]=psplit(matrix)

This function is used most frequently after the function msplit
Finds the power spectrum of each row of matrix p. As for pfft, there is no detrending, no windowing, simply (abs(fft)/len).^2
The matrix should be of the form (i,:) where there are i trials you want fft'ed

Example:

Suppose you have a signal 100,000 points long taken at a sampling rate of 10,000 samples/second.

```
mdata = msplit(data,2500)   % splits the data into 40 windows of 2500 pts
pdata = psplit(mdata)       % finds the power spec of each of the 40 windows
f = mkf(2500,10000)         % inputs are length of run, sample rate
plot(f,mean(pdata));
```

/PlottingFncts

findx, findy, findz **Find x,y,z axis ranges**

[i] = findx OR [i] = findy OR [i] = findz

Echos the min value, max value, and range of the current x [y,z] axis to the screen
The return variable i = minvalue:maxvalue

This function is useful for selecting portions of the data that you have plotted in a figure. So, for example, suppose that you've zoomed in on a particularly interesting feature of the data. Now you want to select out that portion of the data and analyze it. Simply type: testdata=data(round(findx)) and you will have selected the correct points in the data.

mplot **Plot a matrix with the rows offset**

[void] = mplot([t],[dcoffset],matrix,[offset],[color]);

Plots row matrices with each row offset by an amount specified by the optional argument [offset]. All parameters in square brackets are optional
t is a vector to plot the matrix traces against (usually a time or frequency vector)
dcoffset is the offset for the first trace (default is 0)
offset is the offset *between* traces
(the default calculates offset=mean(max(matrix)'-min(matrix)');)

mplotplot

Not a stand-alone function, Called by mplot.

pb **Put Back an object onto a figure**

pb([n])

Put Back object "n" onto the figure. Default (no input) is current object (n=1)
Note that if you haven't removed the object from the figure to begin with, pb will appear to do nothing; it is putting back an object that is already present.

plotv **Plot a two-vector**

[h]=plotv(twovector,[color]);

Plots 2-vectors, equivalent to ***plot (data(: , 1), data(: , 2)); or***
plot (data(1 , :), data(2 , :));

rect **Draw a rectangle**

[h]=rect(lowerleft,upperright,[color])

Plots the rectangle specified by points at the lowerleft and upperright. Returns a graphics handle to a patch.

Example: h=rect([0,.5], [2,3], 'b')

Draws a blue rectangle whose bottom left corner is at [0,.5] and whose top right corner is at [2,3].

rm **Remove an object from a figure**

rm([n])

Removes object "n" from the figure. Default (no input) is current object (n=1)

TextAtTop **Put multiple lines of text at the top of a figure**

[h]=TextAtTop(cellin)

Adds multiple lines text at the top of a figure with nice spacing.
Example: h= TextAtTop({'morestuff';'lotsmorestuff'});
Returns a handle h to the text.

/Shortcuts/Axes

ax **Set figure axes back to 'auto'**

Shortcut identically equal to ***axis('auto');***

axx, axy, axz **Quickly change figure axes**

Suppose you have a 2d plot and you'd like to change the range of the xaxis but leave the yaxis the same. This means that you have to determine what the y axis range is and then type: axis([newxmin,newxmax,ymin,ymax]). The functions **axx, axy,** and **axz** allow you to do this in a faster way.

axx(xmin, xmax) forces the x axis to range between xmin and xmax, while leaving the y and z axes unchanged
axy(ymin, ymax) forces the y axis to range between xmin and xmax, while leaving the x and z axes unchanged
axz(zmin, zmax) forces the z axis to range between xmin and xmax, while leaving the x and y axes unchanged

cax **Quickly change color axis**

cax(cmin cmax) forces the color axis to range between ***cmin*** and ***cmax*** and then adds a colorbar. You can think of this as a shortcut identically equal to

caxis([cmin, cmax]);
colorbar;

hline **Easily draw horizontal lines**

[h]=hline([startpoint],[endpoint],level,[color]);

Draws a horizontal line from ***startpoint*** to ***endpoint***, at the given vertical ***level***. Bracketed inputs are optional, with the following caveat: you must either specify both ***startpoint*** and ***endpoint***, or neither of the two. Omitting both startpoint and endpoint draws the line the entire extent of the current axes. hline returns a handle h to the line drawn. The default color is black.

Examples:

hline(6) draws a horizontal black line across the entire width of the current axes at the level 6.

h = hline([0:10:100]) will draw 11 lines across the entire width of the current axes at each of the specified levels. The return handle h will be a vector of 11 handles, one to each of the lines.

h = hline(6,10,20,'r') will draw one red horizontal line from 6 to 10 at a y-value of 20

ho **Hold on**

Shortcut identically equal to ***hold on***

rmbox **Remove bounding box**

Removes the bounding box and replaces it with plain black lines

rmticks **Remove ticks from plot**

Removes the ticks from a plot

setxlab, setylab, setzlab Set labels

[h] = setxlab(axisvector)

Changes the labeling for the x, y, and z axis respectively

Often used in conjunction with ***setxtick, setytick, setztick***

Examples:

setxlab([1:2:5])

setxlab(['a';'b';'c';'d';'e';'f'])

Sometimes the interactions between the axis ranges and tick marks get a little buggy. It will be easier for you to play with them than for me to try to explain them here. Also, I think the improved GUI interface may have made this command not so useful as in the past.

setxlabelpos, setylabelpos Set label positions

[h]=setxlabelpos(choice);

sets the position of the x [y] label

Input argument 'choice' is a string equal to:

top | cap | middle | baseline | bottom

setxtick, setytick, setztick Set axes' tick marks

[h] = setxtick(axisvector)

Changes the tickmarks for the x, y, and z axis respectively

These are used most frequently in conjunction with ***setxlab, setylab, setzlab***

unlike setxlab, the input argument axisvector must be numeric

Interactions between the axis ranges and tick marks can get a little buggy. It will be easier for you to play with these than for me to try to explain them here. Also, I think the improved GUI interface may have made this command not so useful as in the past.

ticksin Turn ticks inwards

Turns the ticks on a plot inwards

Shortcut identically equal to ***set(gca,'TickDir','in');***

ticksout Turn ticks outwards

Turns the ticks on a plot outwards

Shortcut identically equal to ***set(gca,'TickDir','out');***

vline Quickly draw vertical lines on a plot

[h]=vline([startpoint],[endpoint],level,[color]);

Identical to hline except that it draws vertical lines instead of horizontal ones.

xlin/xlog Toggle x-axis between lin and log

Toggles the x-axis back and forth between linear and logarithmic axes

ylin/ylog Toggle y-axis between lin and log

Toggles the y axis back and forth between linear and logarithmic axes

/Shortcuts/Colors

Matlab's default color choices for yellow, green and cyan are difficult to see on a white background. Typing any of the following commands will turn the current object to that color. This means that on a white background, there are now 13 distinguishable colors: black, blue, lblue, dblue, cyan, green, dgreen, lgreen, yellow, orange, red, purple, magenta/maroon. Magenta and maroon are identical.

black
white
blue (altered from default blue)
lblue
dblue
cyan (altered from default cyan)
green (altered from default green)
dgreen
lgreen
yellow (altered from default yellow)
orange
red (same as default red)
purple
magenta (same as default magenta)
maroon (identical to magenta, because I can never remember what the name actually is)

LookAtColors

Displays the full range of Matlab colors at once. Matlab colors are defined by the three vector [i, j, k]. Useful in the design of custom colors for figures. This function takes a couple seconds to run – be patient.

ChooseColorMap

Cycle through the color maps so you can see what your picture looks like in different colors as you are procrastinating putting the final touches on your paper

/Shortcuts/DisplayFormats

bank
Shortcut identically equal to ***format bank***
compact
Shortcut identically equal to ***format compact***
long
Shortcut identically equal to ***format long***
loose
Shortcut identically equal to ***format loose***
short
Shortcut identically equal to ***format short***

/Shortcuts/Figures

fig
Shortcut identically equal to ***figure***, but default window is smaller.

figX
Shortcut identically equal to ***figure(X)***

ca
Shortcut identically equal to ***close all***

closeall
Shortcut identically equal to ***close all***

WidePaper
Resize the figure so it prints in landscape orientation and uses the full page

TallPaper
Resize the figure so it still prints in portrait orientation but uses the full page.

EmailSize
Resize the figure so it fits nicely in a typical email window without rescaling

The next 6 functions will need to be adapted for your individual screens. Set the figure to the size that you want it, and then type get(gcf, 'Position') to find out what values to choose.

bigfig
Open a "big" figure window

biggen
Make the current figure window "big"

smallfig
Open a "small" figure window

smallen
Make the current figure window "small"

longfig
Open a figure window across the length of the top of your screen

longen
Make the current figure window run across the top of your screen

/Shortcuts/Fonts

font10
font10bold
font12
font12bold
font14
font14bold
font16
font16bold
font18
font18bold

Sets the fontsize of object "n" to X [and bold if stated]
Default (no input) is current object (n=1)
Note that of course if your most recent object is a line, and not a text object, then nothing will happen.

/Shortcuts/LnWghts&Styles

ln1
ln2
ln3
ln4
ln5
ln6
ln10

[void]=lnX(n)
Sets the line width of object "n" to X.
Default (no input) is current object (n=1)

solid
dash
dashdot

[void]=solid or dash or dashdot (n)
Sets the line style of object "n" to the appropriate style.
Default (no input) is current object (n=1)

/SimpleFunctions

brotate **Rotate an n by 2 matrix**

[xprime] = rotate(x,theta,[origin])

Applies the rotation matrix [cos(theta) -sin(theta); sin(theta) cos(theta)] to the matrix x. Default origin is (0,0)

cdiff **Central difference approximation**

[d] =cdiff(vec)

Calculate the derivative of the vector vec using the central difference approximation. This has the advantage that there are the same number of points in the vector and its derivative, instead of one fewer as is the case when you use the Matlab function diff.

chist **Color Histogram**

[handle,no,xo] = chist(y,x,[color])

This function is identical to Matlab's HIST function except that it allows you to plot histograms in color, and returns a handle to the graphic so you can change facecolor and edgecolor if desired.

distptline **Distance between a point and a line**

[d,xd,yd]=distptline(x1,y1,m,b);

Finds the distance d from point (x1,y1) to the line y=mx+b.
The distance from a point (x1,y1) from a line having the equation ax+by+c =0 is given by abs(ax1+by1+c)/sqrt(a*a+b*b)

distptpt **Distance between points**

[d,xd,yd]=distptpt(x1,y1,x2,y2)

Finds the distance between two points (x1,y1), (x2,y2)
xd=(x2-x1)
yd=(y2-y1)
d=sqrt((xd.*xd)+(yd.*yd))

endmatch **Match up last and first points of the data**

[longvec] = endmatch(vec);

Endmatch is generally used in conjunction with bpfft or another filter.

Suppose you have a signal whose first point has a value of 8 and whose last point has a value of 100. You want to filter the signal (using bpfft, for example) to look at a particular frequency range. Because the fft assumes infinitely repeating data, the discontinuity between the last and first point will give you bad ringing.

One way to solve this problem is to "detrend" the data, but this only works if the best straight-line linear fit really does succeed in matching up the first and last points. Detrending also changes the mean of the data.

An alternative approach is to simply add a bunch of points onto the end of your vector that slowly connect up the endpoint with a point at the same value as your start point. After these two points are forced to match up, you can filter and get no ringing, and then remove all of the extra points you added in.

It doesn't really matter how long it takes for the points to match up as long as they eventually do meet. EndMatch therefore matches up the two ends of the vector with a line the length of the vector. The output argument longvec is of length 2*length(vec)

Example:

```
longdata=endmatch(data);
filtered_data = bpfft(data,10000,0,100);
filtered_data = filtered_data(1:length(data));
```

findc **Find closest value in vector**

[index,value_at_index]=findc(vector_in,value_desired,[quiet]);

Determines the value in vector_in closest to value_desired. This is useful because the Matlab function "find" does not work unless there is an exact match out to the umpteenth decimal place.

The output argument "value_at_index" is the value in vector_in closest to value_desired. The output argument "index" is the index location of that value in vector_in

Important : If vector_in contains more than one closest match to value_desired, then the output argument index is set to the first location of the closest match

By default, FINDC echos the results to the screen in the form of a matrix that includes the 4 values that surround desired_value, as follows:

The top row of the matrix is idx-2:idx+2
The middle row of the matrix is vector_in(idx-2:idx+2);
The bottom row of the matrix is vector_in(idx-2:idx+2) - desired_value

quiet is an optional argument that can either be a string or a number. If quiet is set to anything other than the number 0, it suppresses the screen echo.

Example:

```
x=[3.4, 6.3, 6.2, 7.5, 9.4, 5.2, 1.1, 7.5, 3];
 a = findc(x,1)
```

Returns idx = 7, valatidx = 1.1
And echoes this matrix to the screen:

```
5.0000   6.0000   7.0000   8.0000   9.0000
9.4000   5.2000   1.1000   7.5000   3.0000
8.4000   4.2000   0.1000   6.5000   2.0000
```

gaussian **Create a Gaussian of given mean and std.**

[t,g]=gaussian(xbar,sigma,[color]);

OR

[t,g]=gaussian(vector,[color]);

Creates a gaussian with mean x and std sigm

OR

Creates a gaussian with mean x=mean(vector) and std=std(vector)

g = [1/(sigm*sqrt(2*pi))] * [exp ((-(t-xbar).^2) / (2*sigm*sigm))]
t is a vector that ranges from -5 sigma to +5 sigma against which to plot g.

how **Display variables that start with a given string**

how is a script, not a function.

type the command how, and it will prompt you for input
how displays all variables starting with the string you input

Examples: a1 = 3; a2 = 4; b1 = 5; c1 = 6; asdf = 4.7

how

Display all variables starting with: [I choose to input a]

returns:

a1 = 3
a2 = 4
asdf = 4.7000

how

Display all variables starting with: [I choose to input b]

returns:

b1 = 5

inflections **Find inflection points**

[infsup,infsdown] =inflections(f)

Finds inflection points (zero crossings of the second derivative)
Should be self-explanatory.
This function could be improved perhaps by using the central difference approximation (cdiff) instead of the built-in Matlab function diff.

int2str2 **Convert integer to string with extra zeros**

[j]=int2str2(i, [maxval=100])

INT2STR2 is useful in the case that you have to load files of the form filename_001, filename_002 … filename_013. In other words, you need to convert 1 to the string 001, 13 to the string 013 and so on. The optional value maxval gives the maximum number of zeros to add on. Maxval defaults to 100.

Example 1: Your files range between 1 and 23.
INT2STR2(i,23) produces 01, 02, ... 10, 11, 12 ... 23
Setting maxval to any number between 10 and 99 produces the same results

Example 2: Your files range between 1 and 3790.
INT2STR2(i,3790) produces 0001, 0002, ... 0010, 0011 ... 0099, 0100 0101, 0102 ... 0999, 1000, 1001, 1002 ... 3790
Setting maxval to any number between 1000 and 9999 produces the same results

iseven **Determine whether a number is even**

[ret] = iseven(num)

returns 1 if num is even, 0 otherwise.

ismono **Determine if a vector changes monotonically**

[ret] =ismono(vec);

returns 1 if values in vec are monotonically increasing or decreasing, otherwise returns 0

isodd **Determine whether a number is odd**

[ret] = isodd(num)

Returns 1 if num is odd, 0 otherwise.

linfit **Linear fit to xy data, graphical option and r^2**

[m,b,normres,r2,stringout]=linfit(x,y,plotflag,echoflag)

Finds the equation y=m*x + b that "best fits" the data x and y.

linfit calls the Matlab function ***polyfit***, so the "best fit" is in the same least-squares sense as ***polyfit***

plotflag defaults to 'n' (no plot). Set plotflag to 'y' to plot the linear fit.

echoflag defaults to 'y' (echos result to the screen). Set echoflag to 'n' to avoid screen echo of results.

normres is the norm of the residuals. Note that normres depends on how your y values are scaled. If your y values range from 1 to 10, normres will be 100 times smaller than if they range from 100 to 1000.

r2 is a measure that does not depend on how the data is scaled. r2 is calculated by normalizing the covariance of x and y by their standard deviations, that is r2 = cov(x,y)/(std(x)*std(y)). Remember that std is the square root of the variance. Note that the r2 value will be identical to the value obtained using corrcoef(x,y);

stringout is a useful string you can put at the top of your graph using the function ***TextAtTop***

loadimage **Load a tif image file**

[a]=loadimage(filename);

This function is obsolete and should be replaced with loadtif

Allows you to load an image file without remembering where to put all the apostrophes in your eval statement. Especially useful when loading up a bunch of files sequentially in a loop. The input argument *filename* must be a string. This function is identically equal to:

eval(['a=imread(''' filename '.tif'' ,''tif'');']);

Example: a = loadimage('Picture13') loads the file Picture13.tif

loadjpg **Load a jpg image file**

[a]=loadjpg(filename);

Allows you to load an image file without remembering where to put all the apostrophes in your eval statement. Especially useful when loading up a bunch of files sequentially in a loop. The input argument *filename* must be a string. This function is identically equal to:

eval(['a=imread(''' filename '.jpg'' ,''jpg'');']);

Example: a = loadjpg('Picture01') loads the file Picture01.jpg

loadtif **Load a tif image file**

[a]=loadtif(filename);

Allows you to load an image file without remembering where to put all the apostrophes in your eval statement. Especially useful when loading up a bunch of files sequentially in a loop. The input argument *filename* must be a string. This function is identically equal to:

eval(['a=imread(''' filename '.tif'' ,''tif'');']);

Example: a = loadtif('Picture13') loads the file Picture13.tif

mksin **Make a sine wave**

[d]=mksin(frequency, samplerate, length, [phase]);

Creates a sine wave of desired length, given frequency and sampling rate Optional argument *phase* defaults to zero.

mkt **Make a time vector against which to plot data**

[t] = mkt(N,[SR=10000],['m'])

Generate a time vector (in units of seconds or milliseconds) of length N samples, while assuming a sample rate of SR. SR defaults to 10000 Hz. By default, t is generated in units of seconds. Set the last optional argument to 'm' to generate the t vector in milliseconds

mktxcor **Make a time vector against which to plot xcorr of data**

[t]=mktxcor(N,[SR=10000],['m', 'p', or 's'])

Generate a time vector (in units of seconds or milliseconds) of length N samples, while assuming a sample rate of SR. SR defaults to 10000 Hz. By default, t is generated in units of seconds. Set the last optional argument to 'm' to generate the t vector in milliseconds. Set the last optional argument to 'p' to generate the vector in samples.

msplit **Split a vector into segments**

Mreturn=msplit(run,len,[omitvec]);

MSPLIT splits the input argument 'run' into a matrix. 'run' is divided into N segments of length len each (where N obviously depends on the length of the run). Thus Mreturn is a matrix of size N by len. Split does not zero pad, it simply truncates if there are not enough points at then end of run.

The optional argument omitvec is a sequence of startpoints and endpoints to let the user select out some noisy parts of the data. In this case MSPLIT divides the run up to omitstart (still no zero padding) and then starts the next trial with the point omitend+1

pow2n **Find power of 2 nearest to a value**

[d]= pow2n(run)

If the input argument run is a number, POW2N finds the power of two that is nearest to, but smaller than, that number. If the input argument run is a vector, POW2N finds the power fo two that is nearest to, but smaller than, the length of that vector. This is useful when you want to shorten a vector to the nearest power of 2 to do an fft.

Examples: pow2n(31) = 16
pow2n(33) = 32
If data is a vector of length 70000
pow2n(data) = 65536

runningpspec **Find the running power spectrum of a vector**

[t, f, x] = runningpspec (vector, winsize, shiftwin, SR);

Input arguments:

vector is the signal you want to find the power spctrum of
winsize is the window size (an integer, usually about 1/10 to 1/50 of the length of your data)
shiftwin is how many points you want to shift the window over by (an integer, usually about 1/4 to 1/2 of your window size)
SR is the sampling rate of the signal

Output arguments:

t is simply a time vector and f is a frequency vector
x is a matrix that contains the running power spectrum
The easiest way to plot the output will be something like:
pcolor(t,f,x);shading('flat')

The easiest way to explain this function is to walk you through a few iterations of the main loop.

(1) The first window of data is vector(1:winsize)
(2) Find the power spectrum of that first window of data.
(3) Chop off all frequencies higher than Nyquist, take the log of the data, and put that result into the first column of matrix x.
(4) Now shift the window over by shiftwin number of points. This means that the next window of data will be vec(shiftwin:shiftwin+windowsize-1)
(5) Find the power spectrum of that second window of data.
(6) Chop off all frequencies higher than Nyquist, take the log of the data, and put that result into the *second* column of matrix x.
… And so forth...

runningxcov Finds the running cross-covariance between two vectors

[t, xt, x] = runningxcov(vec1,vec2,winsize, shiftwin);

vec1 and vec2 are the signals you want to find the xcov of
winsize is the window size (an integer, usually about 1/10 to 1/50 of the length of your data)
shiftwin is how many points you want to shift the window over by (an integer, usually about 1/4 to 1/2 of your window size)

t is simply a time vector and xt the time lag
x is a matrix that contains the running cross covariance
The easiest way to plot the output will be something like:
pcolor(t,xt,x);shading('flat')

The easiest way to explain this function is to walk you through a few iterations of the main loop.

(1) The first windows of data are: vec1(1:winsize) and vec2(1:winsize)
(2) Find the cross-covariance of the two vectors for that first window of data and put that result into the first column of matrix x.
(3) Now shift the window over by shiftwin number of points. This means that the next window of data will be vec(shiftwin:shiftwin+windowsize-1)
(4) Find the cross covariance of the two vectors over that second window of data. and put the result in the *second* column of x.
And so forth...

scale Scales the data to specified values

[scaled_vector, scalefactor]=scale(vector,[lower=0],[upper=1]);
OR
[scaled_vector, scalefactor]=scale(vector,vector2);

Scales the input argument 'vector' between the values of 'lower' and 'upper.' If 'lower' and 'upper' are omitted, the function SCALE scales 'vector' between 0 and 1, and it alerts you that this is what it is doing.

With two input arguments (both of which must be vectors), SCALE uses the minimum and maximum of 'vector2' as 'lower' and 'upper' to scale 'vector,' and it alerts you that this is what it is doing.

ScaleToUnitArea Scales a vector to unit area

[retvec] = ScaleToUnitArea(vec);

Should be self-explanatory

shorten **Shorten a vector to a specified length**

[shortvector]=shorten(vector, [length] or ['nXXX'])

where nXXX stands for nearest XXX, where XXX is whatever power of 10 you want it rounded closest to.

Example: shortvec = shorten(data, 'n100')
shortens the vector data to the nearest 100.

With only one input argument, SHORTEN will do one of three things:
(1) Shorten to the nearest 1000,
(2) Shorten to the nearest pow2n (see function POW2N)
(3) If the vector is very long (>100000 points), shorten to pow2n(vec) + pow2n(vec)/2

SHORTEN decides which of these three things to do depending on which value is closest to the orginal length of "vec". The point here is to remove as few points as possible, but still have a reasonable vector to work with so that ffts don't take forever.

tdif **Find horizontal distances between user-input points**

[arrayout]=tdif;

Gets the horizontal difference between input points that the user clicks on. Ignores vertical distances. Computes mean, median, and std of the horizontal distances and displays them to the screen. Note that this function is identically equal to xdif

trajectory **Animates the trajectory of vec1 vs. vec2.**

[void]=trajectory(vec1,vec2,[numpts],[erasemode ('xor','none')],[speed]);

Default is numpts =3, erasemode 'none', speed = .05;
Animates the trajectory of vec1 vs. vec2.

vdif **Find vertical distances between user-input points**

[arrayout]=vdif;

Gets the vertical difference between input points that the user clicks on. Ignores horizontal distances. Computes mean, median, and std of the vertical distances and displays them to the screen. Note that this function is identically equal to ydif

why **Answers provided to members of the LIMS, Fish, and Rat labs**

xdif **Find horizontal distances between user-input points**

[arrayout] = xdif;

Identically equal to the function tdif

ydif **Find vertical distances between user-input points**

[arrayout] = ydif;

Identically equal to the function vdif

zeropad **Pad a vector with zeros**

[retvec] =zeropad(vec,desiredlength,'option');

If desiredlength is less than the length of the vector, the program assumes you want to add on that many zeros. Otherwise it assumes the final vector should end up being the desired length

The input argument 'option' determines where the zeros are added, as follows:

option = 's' or 'start' adds the zeros at the start of the vector
option = 'e' or 'end' adds the zeros at the end of the vector
option = 'se' or 'both' (default, adds zeros symmetrically to start and end)

/Statistics

combination **Find random combinations of integers**

[n] = combination(N, k)

Returns vector n which is a random combination of k integers whose values range between 1 and N. Example: n = comination(100,4) might return [1,4,20,99]

comparevecs **Look for common elements between two vectors**

[inAnotB, inBnotA, inAandB]=comparevecs(A, B);

Compares the values in vectors A and B and looks for values in common. The output variable names for this function should make it self-explanatory.

invunc **Calculate inverse uncertainty**

[c, d]=invunc(a,b)

Suppose you have some quantity with an error, such as 10 +- 2.
You want to find 1/(10+-2).
This function returns c=1/a and d= the appropriate uncertainty.
The trick is that whatever percentage of a b is, you want
d to be that same percentage of c.
Hartmann EDA Toolbox v1, Dec 2004

msd **Display statistics of a vector on the screen**

[void]=msd(vec);

Display mean, median, standard deviation, and mean + 1, 2, 3, and 4 times the std on the screen

nmean **Calculate mean and std, ignoring NaNs**

[m, s]=nmean(X,[echoflag =0])

For vectors known to contain NaNs, NMEAN(X) returns the mean value m and standard deviation s of the elements in X, ignoring the NaNs. For matrices known to contain NaNs, NMEAN(X) returns m and s as row vectors that contain (respectively) the mean and standard deviation of each column of the matrix, again ignoring the NaNs.

The Matlab functions mean and std return a NaN if any element is a NaN. In contrast, NMEAN ignores the NaNs and finds the mean and std of all of the other elements.

echoflag defaults to 0 (do not display the results on the screen). Set echoflag to any string or numeric value to turn echo on.

removedups **Removes any duplicate values from the vector vec**

[shortenedvec] = removedups(vec)

Should be self-explanatory