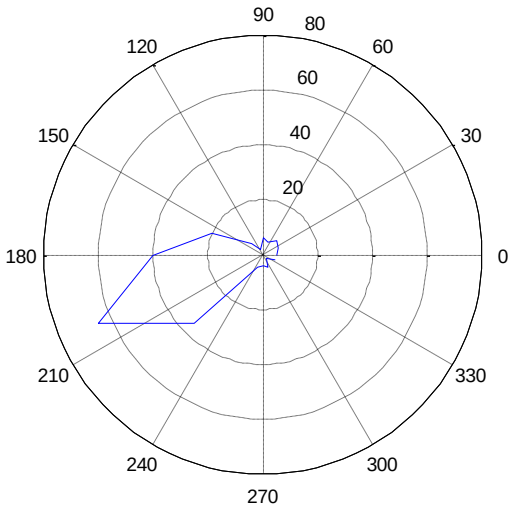
More Graphics Tools

1.1.1. polar

A simple extension to the Cartesian plots of the plot function is the `polar` function. This function takes a vector of angles (in radians) and radii as its argument and returns a polar plot of these data. Generalizations to matrices are the same as for the plot function and line style options can also be specified as an optional third argument. Say you determined the mean firing rate for a neuron responding to visual motion in 16 directions spread evenly between 0 and 360 degrees.

```
direction = (0:22.5:360-22.5)
response = [5 6 7 5 6 2 6 20 40 65 35  5 4 5 2 5] ;
polar(direction*pi/180,response)
xlabel 'Direction Tuning'
```



From this plot it is immediately clear that the neuron responded most vigorously when the motion was down and to the left.

1.1.2. plotyy

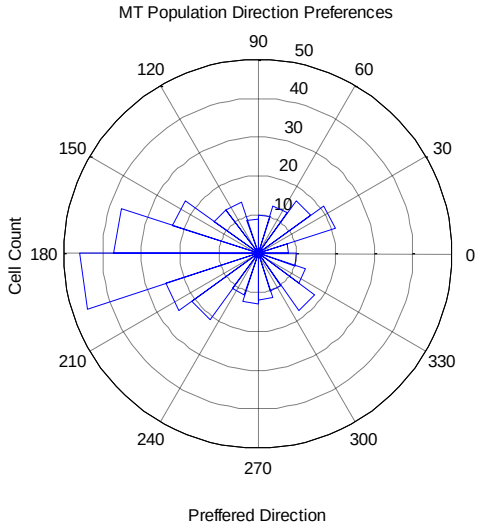
Sometimes you may wish to show two curves on the same x-axis, but on different y-axis. This could, for instance, be useful to show how two measures with very different units (say to show how the weight and the volume of a brain changes with age). The plotyy function is useful for this; it creates one y-axis on the left side of the x-axis and another on the right side.

1.1.3. rose

Many quantities (and some arguments) in Neuroscience are circular (e.g. direction, orientation, phase). For such data a polar histogram is usually better than a Cartesian one. The rose function implements this. As an example, consider a population of motion selective neurons recorded from the middle temporal area. Each one has a preferred direction and the list of preferred directions, in radians, is stored in the variable `preferredDirection`.

```
rose(preferredDirections)
xlabel 'Preffered Direction'
ylabel 'Cell Count'
```

```
title 'MT Population Direction Preferences'
```



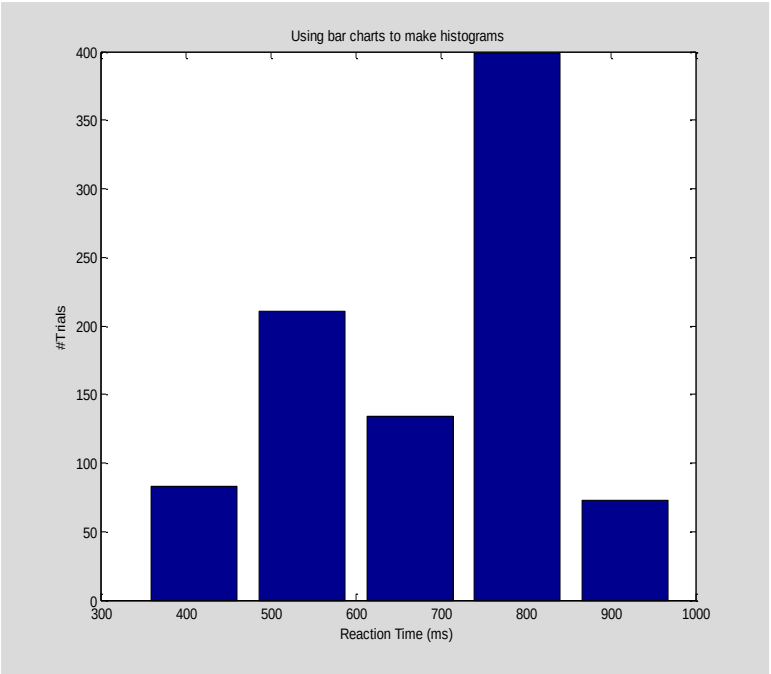
Which shows that more cells preferred leftward motion than any of the other directions of motion. Don't forget to transform your dataset to radians before applying the `rose` function. The figure also shows that automatic labeling is not perfect in this case (the 'Cell Count' label is a bit confusing; it should be along a radial axis somewhere). This illustrates that some hand tuning would be needed for publication quality graphics.

Sometimes when you make a histogram you don't just want the picture, but you need the number of elements in each bin as well. The `hist` and `rose` functions return arguments for this purpose:

```
[nrPerBin,binCenter]= hist(reactionTimes(:),5)
nrPerBin =
    83   211   134   399    73
binCenter =
  410.3779  537.0183  663.6587  790.2991  916.9396
```

Note that calling the `hist` function with output arguments does not generate a graph. From the output of the `hist` function, however, you can generate the histogram chart with the `bar` function:

```
bar(binCenter,nrPerBin)
title 'Using bar charts to make histograms'
xlabel 'Reaction Time (ms)'
ylabel '#Trials'
```

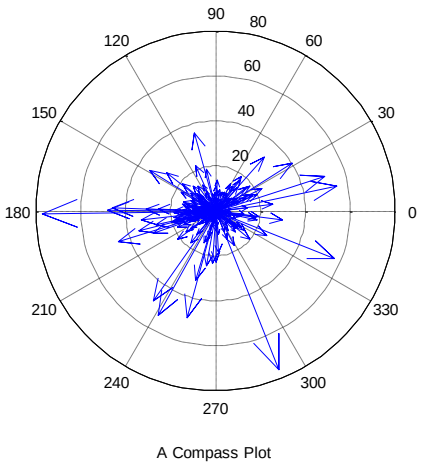


The function `rose` can also return bin centers and numbers per bin, but to plot these data on a polar grid, you will have to use the `polar` function.

1.1.4. compass, feather, quiver

Directions, velocities and orientations are most easily depicted with arrows. Matlab provides the compass function which plots arrows emanating from the origin of a polar coordinate system, the feather function which shows arrows emanating from a horizontal line and the quiver function with which you can plot arrows at arbitrary positions in two or even three dimensional space. Strangely, all these functions require Cartesian coordinates as input. To make their use somewhat easier, use the functions that convert between polar and Cartesian coordinates: `pol2cart` and `cart2pol`. As an example, consider the population of direction tuned neurons discussed in the example of the `rose` plot above. Suppose that apart from their preferred directions, we also know the firing rate of these neurons when stimulated in their preferred direction (stored in `peakFiringRate`).

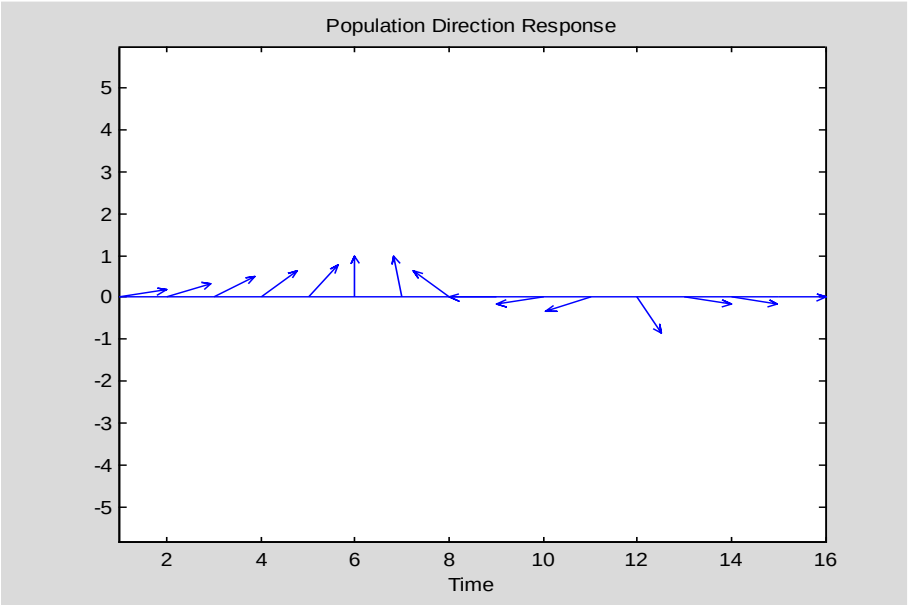
```
[cartDirections,cartRates] = pol2cart(preferredDirections,firingRate);
compass(cartDirections,cartRates);
xlabel 'A Compass Plot'
```



This shows both preferred direction and the strength of the response in that direction in a single picture. Note the conversion from polar to Cartesian and finally the plotting routine. `feather` plots are a variant of the compass plot to be used when the independent variable is not periodic.

Direction, for instance, is periodic hence the direction selectivity is best displayed in polar coordinates. Time, however, is not periodic; hence a temporal evolution of some vector quantity (say the population response of direction selective cells to a particular stimulus evolving over time) should be displayed on a Cartesian coordinate system. Consider for instance a data set in which the population response points to a particular direction:

```
populationDirection =[ 10 20 30 40 50 90 100 140 180  190 200 300 350
350 360]*pi/180;
rate =1*ones(size(populationDirection));
[x,y]  = pol2cart(populationDirection,rate);
feather(x,y)
axis equal
xlabel 'Time'
title 'Population Direction Response'
```



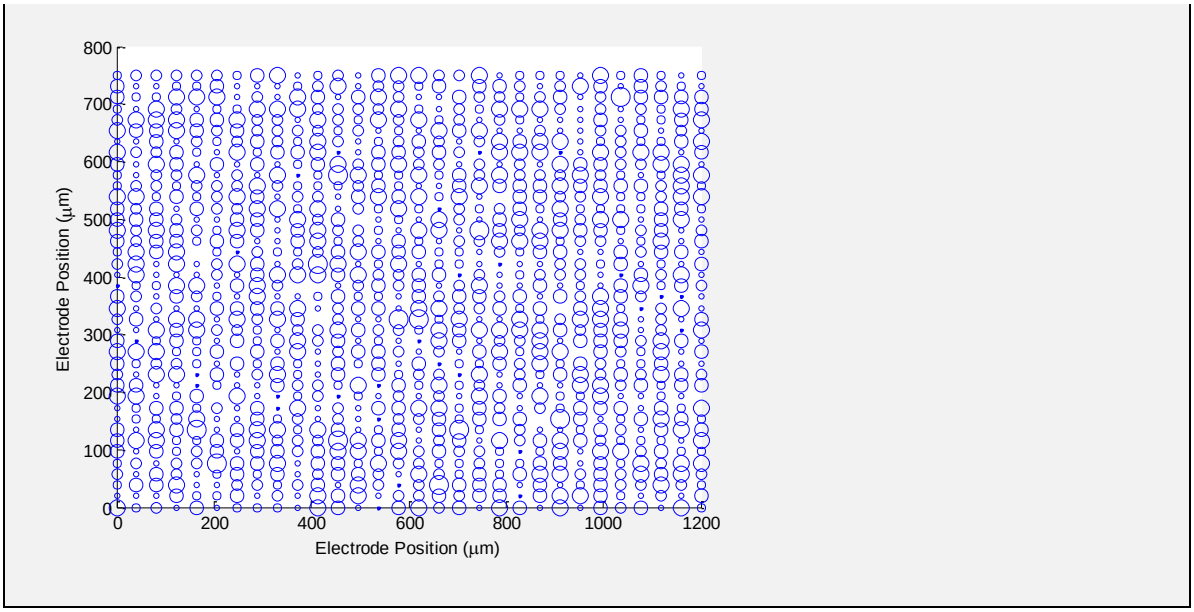
Finally, `quiver` plots give complete freedom to place arrows in the plotting area. A two-dimensional quiver, for instance, takes four arguments. The first two specify the x, y location of the base of the arrow, while the next two u, v specify the direction of the arrow. A fifth argument can be used to scale the arrows. The quiver function is useful to display optic flow or other vector fields.

1.1.5. scatter

This function is a useful way to represent three-dimensional data in a two-dimensional format . For instance, `scatter` could be a good way to show the total number of spikes recorded from each electrode in a two-dimensional array:

Here we first generate some (random) spike count data on an array, and then create a scatter plot that represents the spike counts in a spatial layout.

```
nrRows= 30;
nrCols = 40;
lambda = 100*rand(nrRows,nrCols);
spikeCount = poissrnd(lambda);
x = linspace(0,1200,nrRows); %1200x750 array with 40x30 electrodes
y = linspace(0,750,nrCols);
[X,Y] =meshgrid(x,y);
spikeCount(spikeCount==0)=NaN; % Scatter cannot deal with zeros.
scatter(X(:),Y(:),spikeCount(:));
xlabel 'Electrode Position (\mum)'
ylabel 'Electrode Position (\mum)'
```



1.2. 3D Graphics

Matlab's extensive suite of 3D graphics commands allow you to create very complex graphical data representations. Some of these are truly useful in the sense that they can show aspects of the data that are difficult to understand with 2D drawings, others, however, merely add graphical fluff and can actually hide the true meaning of your data. We'll discuss a few examples of each of these categories. For a full list, with brief descriptions, see help graph3d and help specgraph (especially if you have complex needs for volume and vector visualization).

It can be quite a lot of work to get 3D pictures to look "just right". This is not some shortcoming of Matlab but rather the fact that paper and your computer screen are two-dimensional. This means that you will have to choose a point of view from which to look at the whole three-dimensional structure. If you take the wrong point of view, the interesting structure in your data may remain hidden. Drawings can be much improved by using lighting, shading, coloring, adding or removing grids, labeling, the combination of contour with surface plots or even an occasional quiver plot on top of a surface. All of this requires a lot of experimentation and patience. Before you start developing more complicated plots, browse through the list of graphics commands to see whether there is no other way to plot the same data or maybe a predefined Matlab function that does just what you want. It is also worth considering before you start whether a colored drawing will be acceptable for a publication (and at what cost), otherwise it may be a waste of time.

1.2.1. plot3

The straightforward generalization of the plot function is the plot3 function which takes x, y as well as z vector or matrix arguments. Each point in these vectors is plotted as a point in the graph and interpolating lines can connect these data points. Just as with plot, you can pass matrix inputs and each column of the matrix will be plotted.

When you print a figure generated with plot3 it usually becomes pretty difficult to interpret; the lines and points just become points on the page. Sometimes it helps to add lines to the ground plane of the 3D plot (see `stem3`), but even that often fails to provide good insight into the data.

A better solution is to use view points; Matlab allows you to look at a 3D scene from different angles. The simplest way to do this is to click the 'Rotate 3D' icon in the figure window. Once activated you can use the mouse to change the view point. It is easy to loose track of where you

are, so before you do this, you may want to put some labels on the axes using the `xlabel`, `ylabel`, `zlabel` commands. Of course this only works while you are still working in Matlab. To use multiple viewpoints in a presentation or as a part of a publication, you could create a movie that goes through multiple viewpoints.

1.2.2. bar3

Not surprisingly, this is the 3D variant of a bar plot. This can be useful to represent data from an experiment with two independent and one dependent variable. Often, however, the fancy 3D layout distracts from the underlying data (and some data points may even be partially hidden, although you could use transparency to partially overcome this problem; see Handle Graphics). Moreover, adding error bars to 3D graphs is complicated. As an alternative, consider using a simple `plot` function with multiple lines and a `legend`.

1.2.3. hist3

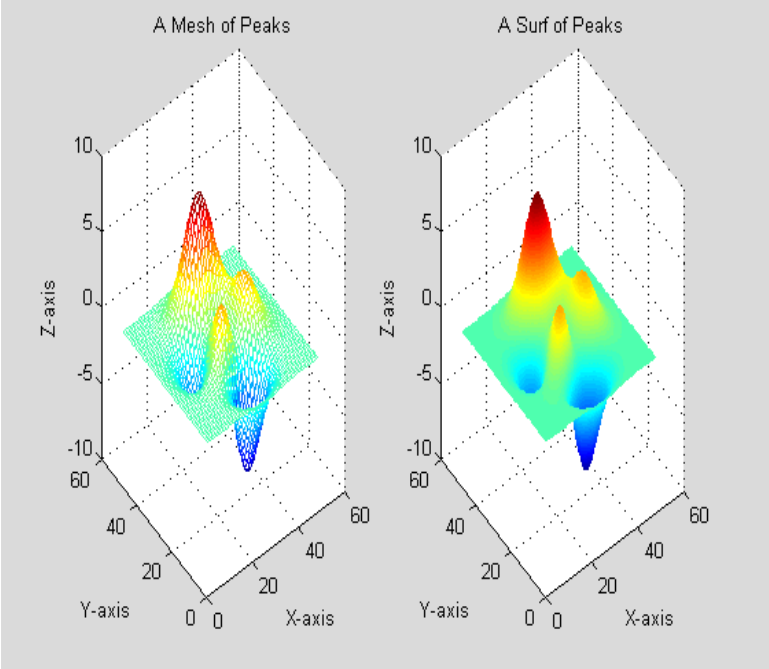
This is the three dimensional alternative to hist, the histogram function. You pass a matrix with two columns. The first column represents one measurement, the second a different measurement for the same item. For instance, you have 100 measurements of spike count, and gamma power; the spike counts would be one column, the gamma power the second column. When these values are passed to `hist3`, it generates a 3D histogram with the spike count on the x-axis, the gamma power on the y-axis, and the number of observations in the data set on the z-axis. Just as with `hist`, you can specify the number of bins, or even each of the bin centers; see `help hist3`.

The histogram calculation that `hist3` performs is very useful, but just as with `bar3`, the 3D representation of the data is far from ideal. Instead, I'd recommend to ask `hist3` for the numbers and bin centers and then use a function like image, or contour to represent the histogram in a format that works better on the printed page.

1.2.4. surf, mesh

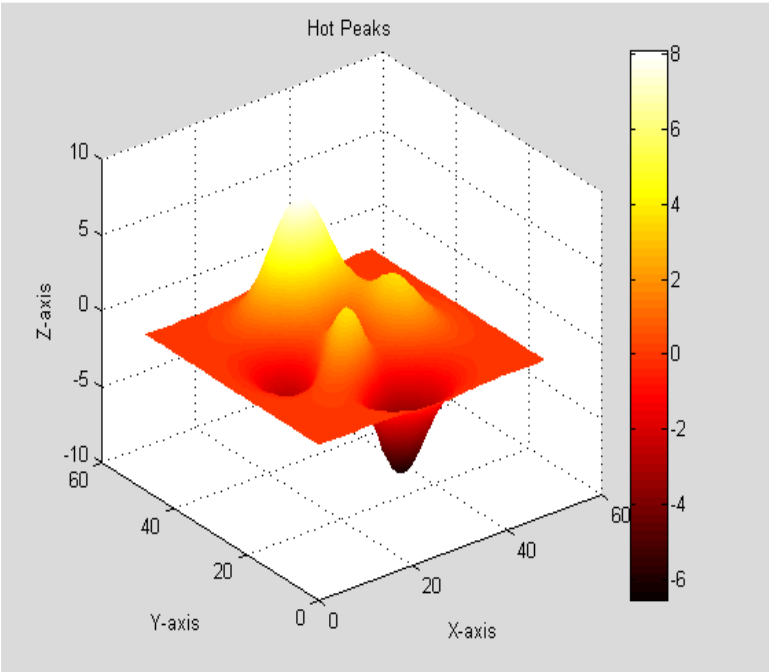
The `surf` and `mesh` functions are the main commands to show data in a 3D representation. For simplicity we'll use the peaks function, which is just a matrix that looks nice.

```
subplot(1,2,1);
mesh(peaks)
xlabel 'X-axis'
ylabel 'Y-axis'
zlabel 'Z-axis'
title 'A Mesh of Peaks'
subplot(1,2,2)
surf(peaks)
shading interp
xlabel 'X-axis'
ylabel 'Y-axis'
zlabel 'Z-axis'
title 'A Surf of Peaks'
shading interp
```



The difference between a `mesh` and a `surf` plot is that the `surf` plot shows the whole (interpolated) surface, whereas the `mesh` plot merely shows a wireframe connecting the data points. Matlab automatically adds coloring to clarify the structure of the 3D plots. The colormap function changes the coloring scheme used in a plot.

```
surf(peaks)
shading interp
xlabel 'X-axis'
ylabel 'Y-axis'
zlabel 'Z-axis'
title 'Hot Peaks'
colormap(hot)
colorbar
```

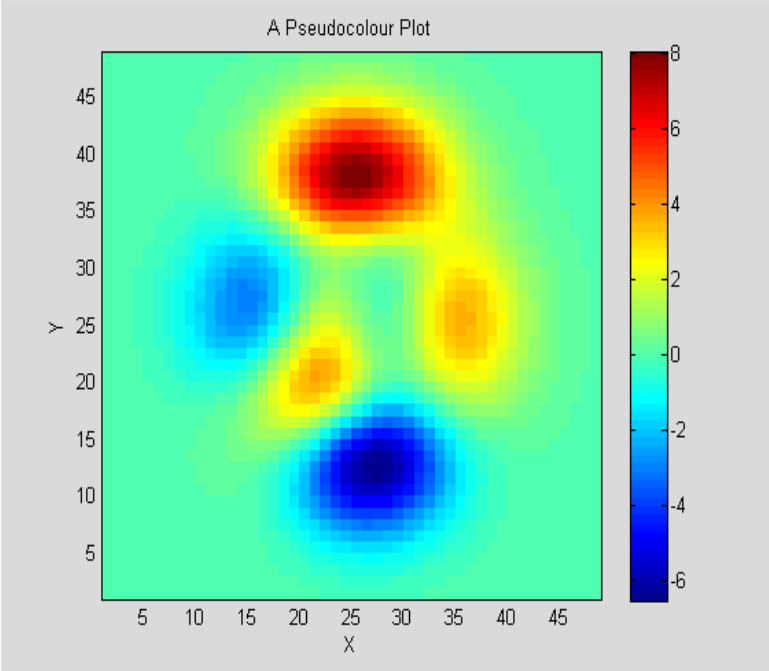


This illustrates the `'hot'` colormap together with the `colorbar` function that adds a legend to the figure to explain the interpretation of the colors.

1.2.5. pcolor, contour

Another instructive view of a surface is from above, this can be set by using `view` after a `surf` or mesh command, but it is more easily done with the `pcolor` function:

```
pcolor(peaks)
colormap(jet)
xlabel 'X'
ylabel 'Y'
shading flat
colorbar
title 'A Pseudocolour Plot'
```



There is very little information in the full 3D view that is not visible in this psuedocolor 2D view. In fact, this view is a better way to show that nothing is hidden behind the large mountain in the `surf/mesh` standard view. One disadvantage of the psuedocolor view is that it can be hard to tell whether the value at (25,12) is very negative (dark blue) or very positive (dark red). This is not so much a problem of the psuedocolor plot, but rather a problem of the jet colormap. Using colormap `hot` instead is one way to solve this issue.

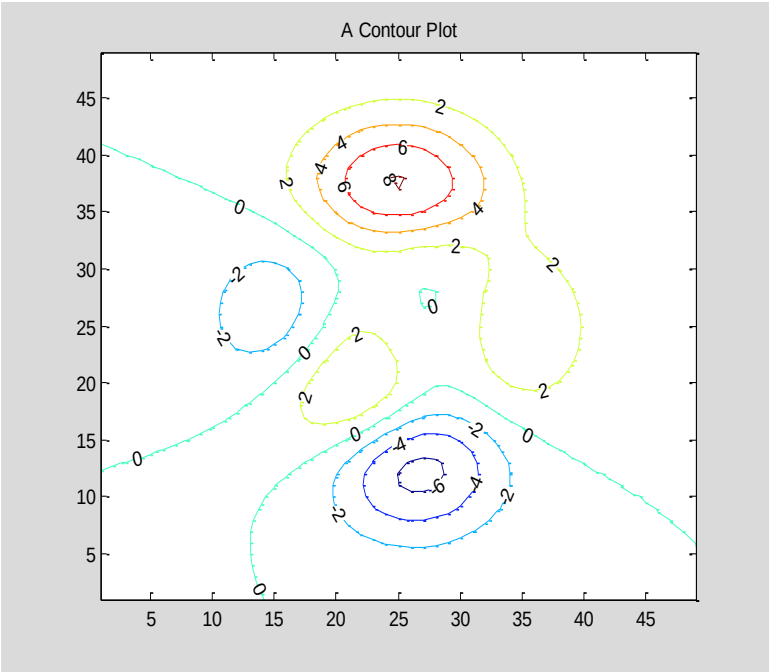
The `pcolor` command has some peculiar properties depending on the shading options. In the default mode (`shading faceted`) pcolor will remove the last row and the last column from the matrix you want to visualize. The remaining elements are then shown as pixels (for instance, the element of the matrix at (1,1) is shown as the lower left pixel (or to be precise, it will be shown at X(1), Y(1) where X and Y are the first two arguments passed to pcolor. If these arguments are not specified, they are taken to be 1:N, and 1:M where N and M refer to the number of rows and colums in the data matrix.). If the edges of the matrix are not that important, and if the matrix has many entries, this cropping behavior may not be a problem, but in general not displaying part of your data is obviously a bad idea. Instead of `pcolor`, you are probably better off using the `image` command to show pseudocolor plots.

Note, however, that if you apply the `shading interp` command, `pcolor` will interpolate values between the values in the specified matrix, and this will include the values in the last row and column. Hence if you want to show interpolated values (think twice about that…) `pcolor` is a convenient, quick command. If, on the other hand, you want tight control over the kind of interpolation, then you should first calculate an interpolated data matrix (using `interp1`) and then visualize it with `image`.

1.2.6. contour, meshc

The contour function is another way to display 3D data in a comprehensive 2D plot:

```
[c,h]=contour(peaks);
clabel(c,h);
title ' A Contour Plot'
```



The optional `clabel` function adds labels to the iso-contour lines. The x and y axis can be changed by specifying x and y vectors: `contour(x,y,peaks)` for instance. Sometimes a combination of a 3D representation and a contour plot can also be instructive. You can achieve this by issuing the hold command followed by the successive graphics functions, but the `meshc` and `surfc` functions are convenient shortcuts to show both a `mesh` (or `surf`) and a `contour` plot in a single graph.

1.2.7. image, imagesc

The image command is possibly the most generally useful command to visualize 3D data using psuedo colors. You use it to display "activation" in fMRI (x and y axis correspond to location in the brain, the color corresponds to the T-value of a statistical parametric map), spectrograms of local field potentials (x axis is time, y axis frequency, color the power in the LFP), or a 2D histogram showing the co-occurrence of certain spike counts and LFP gamma power across an array of electrodes.

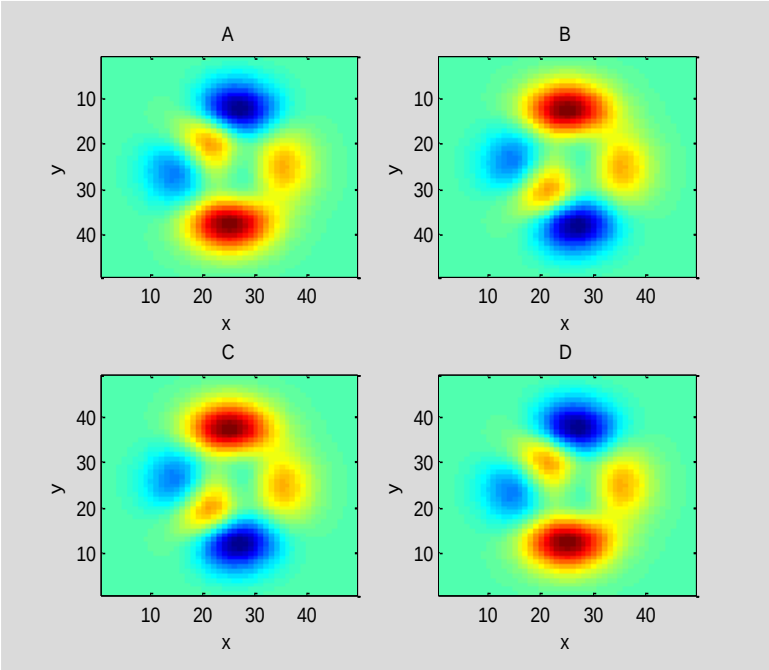
Similar to most 3D graphical commands, the image function can either take a single matrix as its input (`image(m)`) , or X and Y coordinates and the matrix (`image(x,y,m)`). In the former case, x is assumed to be `1:M` and `y=1:N`, where `M` is the number of rows and `N` the number of columns of the matrix `m`. Just like the `surf`, `mesh`, and `pcolor` functions, the values in the matrix are interpreted in terms of the color map, which you control by calling the `colormap` command.

Because the `image` command is typically used to display images, it displays a matrix upside down compared to the other 3D graphics commands. Specifically, the `m(1,1)` element of the matrix will be in the top left of the axis, and the `m(end,end)` at the bottom right. Matlab calls this 'matrix mode' for an axis. If you don't want this behavior, you can go back to the usual 'cartesian mode' by typing `'axis xy'`. This can get confusing, so let's look at an example.

```
m=peaks;
```

```
[r,c] = size(m);
x=1:c;
y=1:r;
yReversed=r:-1:1;
subplot(2,2,1);
imagesc(x,y,m)
xlabel 'x';
ylabel 'y';
title 'A'
subplot(2,2,2);
imagesc(x,yReversed,m)
xlabel 'x';
ylabel 'y';
title 'B'

subplot(2,2,3)
imagesc(x,y,m)
axis xy
xlabel 'x';
ylabel 'y';
title 'C'
subplot(2,2,4)
imagesc(x,yReversed,m)
axis xy
xlabel 'x';
ylabel 'y';
title 'D'
```



Panel A shows the default image representation: the x-axis runs from left to right, but the y-axis runs from top to botton. This is a matrix mode image graph. Panel B is also matrix mode (the y-axis runs from top to bottom), but because we passed yReverse to image, the first row in m is now shown at the y-coordinate that corresponds to the first value in yReverse (which is 49). That is why the image is now flipped upside down compared to Panel A. Note that this changes the interpretation of the image; the biggest value in B (dark red) is near coordinates (25, 10) while the biggest value in panel A is near (25,40).

Panel C uses the identical `image` command as in panel A, but after that we called `'axis xy'` to go into Cartesion mode; this changed the look of the picture, but not its meaning: the biggest value is still near (25,40), just as it is in panel A. For completeness, panel D uses the same image command as panel B, and then switches the axis to Cartesian mode. The picture looks different from panel B, but its meaning is the same.

So to summarize, the decision whether to use `image` in the default matrix mode (`axis ij`) or in the cartesian mode (`axis xy`), is merely a cosmetic one. The values you specify as x and y, however, directly affect the interpretation of your data, and only one choice will be correct.

Note that I used `imagesc` in the example. The –sc stands for scaled, which means that the values in the image matrix are scaled such that the entire color map is used. This is usually what you want as it brings out variations across the image most clearly. The `image` function does not scale the values of the matrix, which means that the number 1 in the matrix will generate the color corresponding to the first row of the colormap matrix, and the number 10, the tenth row etc. As a result if all of the values in the matrix are between 0 and 1, you'll get an image that contains only a single color (the first row).

1.3. Creating publication-ready figures

Sometimes you may wish to fine-tune the layout of a figure. This is very common when you want to prepare a multi-panel figure for inclusion in paper; you need to adjust the fonts, their size, add a label to a panel, and move axes around to fit everything with the space alloted by the Instructions for Authors.

One way to do this is to create a rough figure in Matlab, export it to some other format and then edit the figure in a graphics package such as Adobe Illustrator. This certainly works, but I'd recommend against this as a default workflow for at least two reasons. First, graphical packages like Illustrator make it easy to change everything on the page. While this may sound like a blessing, it is actually a curse, as a mistake is easily made and data points or axis labels are inadvertentlty erased, changed, or moved. The second reason is that during the course of analysis, submission, review, and resubmission, figures often change. You may have changed your analysis a bit, or a reviewer suggested to include another analysis in the same figure. If you've edited your figure in a graphics package, you basically have to start over in such a situation. My recommendation is to do as much fine-tuning as possible in Matlab and only use a graphics package for the final touch (if at all).

Handle graphics are essential to the fine-tuning. We'll start from the top by looking at some of the relevant properties of the figure window (you can look at all properties by typing `set(gcf);`)

```
PaperUnits: [ {inches} | centimeters | normalized | points ]
PaperOrientation: [ {portrait} | landscape | rotated ]
PaperPosition
PaperPositionMode: [ auto | {manual} ]
PaperSize
```

These properties allow you to change the format of a figure. For instance, a single column figure in the Journal of Neurophysiology has a width of 8.9cm. To create a one-column figure with a height of 4cm for JNeurophys we first define a properties struct and then apply it to the figure.

```
 width =8.9; height =4;
JNeurophys.PaperUnits = 'centimeters';
JNeurophys.PaperOrientation ='portrait';
JNeurophys.PaperPosition = [0 0 width height];
JNeurophys.PaperPositionMode = 'manual';
```

```
JNeurophys.PaperSize = [width height];
```

These properties define what the figure will look like when printed (or when exported to a file that you can send to a publisher). The PaperPosition property determines where on the page the figure will be "printed"; the 4-vector represents the x position, y position, width, and height, in that order. Here I set the x and y position to (0,0), hence the figure will be printed to the bottom left of the page. The size of the page itself os set to [width height] such that it exactly matches the size of the figure on the paper. This is convenient for exporting figures, but you if you want to print the figure onto an A4/Letter sized piece of paper, then you probably don't want it to appear in the bottom right, but somewhere near the middle. Just sset the PaperSize property to 'A4' and the 'PaperPosition' to something like [3 10 width height].

If you only set the Paper* properties, your figure will print/export to the correct size, but the figure window that you see on the screen will look quite different. This is not ideal, because it prevents you from clearly seeing whether axis are too close to each other, or whether fonts are legible. You can always look at a print preview (Use the File | Print Preview menu option in the figure window) to check the final layout, but I find it more convenient to match the layout on the screen with the layout on paper. You can do this with the Units and Position properties:

```
JNeurophys.Units = 'centimeters';
JNeurophys.Position = [10 10 width height];
```

Note that I chose the same widht and height (and the same units!) but a different x and y position. This is merely to prevent the figure window from being pushed all the way in the bottom left corner of the screen (0,0). Because the Position property does not affect the PaperPosition this is harmless and makes for easier reading of the figure on the screen.

Now that the figure has the correct size, let's discuss the layout of multiple axes. As discussed above, you can use the subplot function to generate a multi-panel figure, but the spacing between panels is not always optimal or it may not suit your purpose. The axes function allows you to place axes anywhere on a figure by specifying the position:

axes('Position', [0 0 0.5 0.5])

The default units in this specification are called 'normalized': (0,0) is the lower left of the figure, and (1,1) the top right. But you can change this by specifying the units, just as you would in a call to set:

axes('Units','Centimeters', 'Position', [2 2 3 4])

In fact, anything that you could pass to the set function for existing axes, you can pass to axes to create a new axis. The more readable `struct` format is legal too. To create new axes without a surrounding box, and with a light grey background color:

```
barAx.Units='Centimeters';
barAx.Position = [ 3 3 3 4];
barAx.Box = 'off';
barAx.Color = [0.95 0.95 0.95];
axes(barAx);
```

Note that axes (unlike subplot) allows you to place one axis on top of the other. This is useful to combine multiple datasets with different scaling in a single graph and is a flexible alternative to the plotyy function.

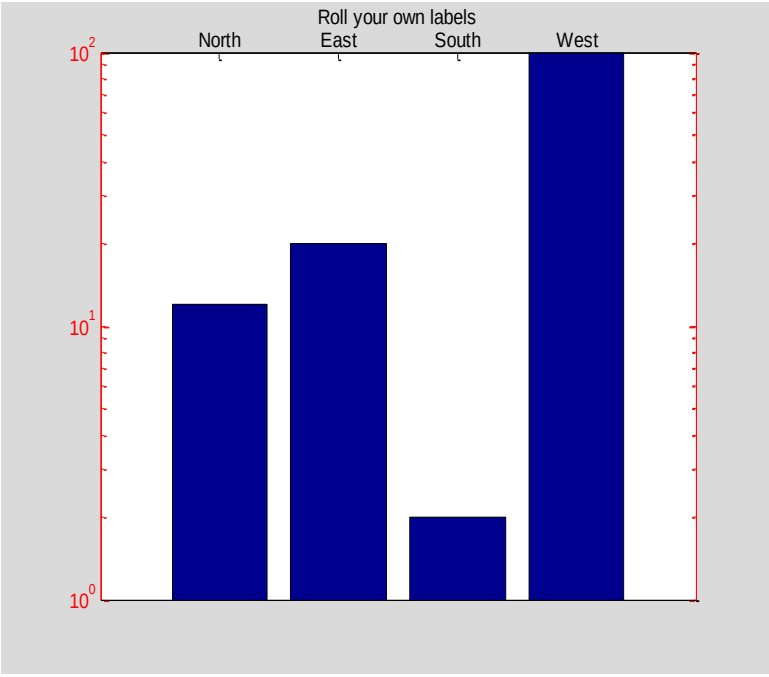
The Font* properties of an axis object refer to the fonts used to draw the numbers on the axes. From this list obtained from `set(gca)`

```
FontAngle: [ {normal} | italic | oblique ]
FontName
FontSize
FontUnits: [ inches | centimeters | normalized | {points} | pixels ]
FontWeight: [ light | {normal} | demi | bold ]
```

you see which properties you can set. Note that these font properties will become the default font properties for the Children of these axes. This means that if you first set the FontSize to 8pt, and then use xaxis or title to add text, that text will also appear in 8pt size. However, if you later change the axis property fontsize, the axis lable or title will stay at 8pt. You can fix this by first retrieving the handles to the labels and titles (e.g. `h=get(gca,'XLabel'))` and then setting its Font property. Of course a better way to do this is to set the axes font properties correctly, at the beginning, before you start adding titles and labels.

Ticks are the small lines crossing the axes where a marker appears (and sometimes between two markers). The Tick properties allow you to change the positioning of these lines, their size etc. The position of the ticks can be set for each axis separately (xTick, yTick, zTick). For each tick you can also decide what the label should be; it defaults to the numerical value along that axis, but you can change this by specifying a string. There are many possibilities for fine-tuning, you can limit the range along an axis ('xLim'), change the color ('xColor'), its location ('xAxisLocation'), direction ('xDir') and whether the units are linear or logarithmic ('xScale'). All properties are listed in the help XXX:

```
ax = gca;
bar(1:4,[12 20 2 100])
set(ax,'XaxisLocation','top')
set(ax,'Xlim',[0 5])
set(ax,'Xtick',(1:4))
set(ax,'XtickLabel', {'North','East','South','West'})
set(ax,'YColor','r')
set(ax,'YMinorTick','on')
set(ax,'YScale','Log')
title 'Roll your own'
```



For simplicity, this example used a set function call to set each property separately. In your code, you should instead use a structure to specify all axis properties and then apply it with a single call to set.

1.4. The Plot and Property Editors

As you can tell from the examples, I use code to fine-tune figure properties. This helps to create consistent figures (i.e. every figure in one paper has the same fonts, because you define the properties once, and apply them to all axes).

However, if you don't know which properties to get and set, then this approach is time-consuming. The plot editor (plotedit) and the property editor (propertyeditor) are Matlab's built-in GUI that allows you to set handle graphics properties manually, and to add embelishments like arrows and text. These are definitely useful to explore the possibilities, but they suffer from some of the same problems that you run into when editing exported graphs in a general purpose graphics package (but without the power of such specialized graphics tools).

My recommendation is to use these tools to explore and see what Matlab can do in terms of graphics. But once you've discovered the properties that you want to tweak, use the set/get functions to tweak them.

1.5. Printing and Exporting Graphics

There are several ways in which Matlab can produce hardcopy output. The easiest way is to go to the File menu in a figure window and select Print. This will pop up the standard print dialog for your operating system, which means that any of your printers available for printing in, say, a word processor are available here too.

To export your figures for use in programs like Word, or to send them to a Publisher as part of an article submission, you can use a built-in graphical user interface. Under the File menu of the Figure, select 'Export Setup…'. You can change settings like Fonts, or scaling here, and you can even save multiple settings as a predefined Setup. This could help to create a consistent look and feel for all the figures you create for an article. But, as explained above, I prefer to use the command line to guarantee consistency, and avoid duplication of effort.

On the command line, you use the print command to print a figure to file. The file format can be chose from a wide range of possibilities, but the most useful ones are EPS, TIFF, JPG, and PNG. The EPS format keeps lines as lines, and text as text. This is particularly useful if you want to edit the figure in a vector graphics program like Adobe Illustrator, but these files can also be imported directly in many word processors (e.g. Word, LaTeX). The TIFF format is preferred for complex color graphics (for instance those that use shading) and which should not be edited further. The file is essentially a pixel-by-pixel copy of your screen (a bitmap). JPG and PNG are two alternative bitmap formats. They can reduce image quality, and are not commonly used for publications, but they are small and easy to import in reports for your own use.

The print command overwrites files with the same name without warning!

When you export a figure for publication you should take note of the color mode that the publisher requires. By default Matlab creates files in RGB colormode, but by specifying –cmyk in the print command, you can generate CMYK files. Exporting a figure that contains only line grahs for submission to the Journal of Neuroscience:

```
print -depsc filename.eps
```

To import this file into Word, add the –tiff option.

This adds a low-resolution preview that will be visible in Word.

While EPS and TIFF files are necessary when you want to send figures out to a publisher, you may just want to quickly copy a figure into a report to share with your collaborators. The Edit menu in the Figure window has the Copy Figure command, wich allows you to simply copy the figure and then switch to your word processor and paste it into the document (Ctrl-V in Windows).

Before you do this, be sure to check the Matlab Preferences (e.g. Preferences in the File menu of the Figure). From the long list of Preferences in the leftmost panel, choose 'Figure Copy Template'. The options here are pretty self-explanatory, and there are some default settings for users of Word and Powerpoint. Under Copy Options, make sure to select 'Preserve Information'. With this setting, most figures pasted into one of the programs of the Microsoft Office Suite remain editable (Right click the figure in the Office program, and select Group | Ungroup, accept the warning). This is not a best practice for generating publication-ready figures (most Publishers will refuse such files), but it can be useful to add some text, or change the layout after you've pasted the figure into a report. Note that the ability to edit pasted figures is lost if those figures contain three-dimensional graphics or color maps (they are converted to bitmaps, which can be edited in specialized programs like PhotoShop or Paint, etc).

1.6. Interactive graphics

Matlab has some useful tools to interact with graphics. Possibly the most useful one is the Data Cursor, which you find in the Figure Toolbar. Select it, then click on a data point in your figure. Matlab will show a data tip (a small text balloon) that tells you the values of that data point, and, if that data point is part of a series, allows you to step to the next point in the series by pressing the arrow keys on your keyboard. This is a very intuitive and useful way to make sense of a complex data graph. If you need to process the numbers that are shown in the data tip, you can right click it, and export the data to a variable that becomes accessible on the command line.

The `ginput` command is a way to collect data from a figure. For instance, if you want to trace the outlines of an image, you would enter `[x,y] =ginput;` on the command line and then click on the outline of an image in the figure. The horizontal and vertical position of every point that you click is stored in the x and y vectors, respectively. This is a great way to extract numerical data from published graphs; a great tool called GrabIt that uses this technique is available on the Mathworks FileExchange.