Creating Publication Quality Figures in MATLAB®

Sarang Pendharker
Fiber Optics Lab, Electrical Engineering Department
Indian Institute of Technology Bombay
India

An important skill that one needs to learn for technical writing is data representation. No matter how good your analytical skills are or how good you are in writing codes, simulations and getting experimental results; if you cannot represent your results in an effective manner, no one cares. It is very important that you must have the ability to project your results in a way that brings forward all the crucial points of your theory/design/analysis/experiments and portrays the width and depth of your study. I am not qualified enough to guide anyone in the above mentioned skill of data representation, which I am still learning. However, for past couple of years, I have had fair experience of some tools used for data representation, i.e. plots and figures. In this small article I intend to compile some of the commands and tricks that I have used in plotting graphs in MATLAB®. All information that is compiled in this article is easily available online on various forums. However, the information is either scattered - not complied at single place, or the available documentation is too exhaustive with lots of information that is not generally required. The purpose of this article is to provide the reader with all the information required for creating publication quality plots, at one place without overloading him with not-so-useful information. Anyone in possession of this file should have access to all the relevant commands and tricks that are needed for creating ready-to-save publication quality plots.

One may ask as to why should we use commands and codes for formatting figure, when most of the formatting can be done in GUI. Well, there are four reasons for this. (1) Few settings like line width, font size etc needs to be modified for almost every plot, and writing it in your code itself saves a lot of time. (2) When you have to repeatedly generate multiple plots with varying data and parameters, manual formatting becomes a headache. (3) Few things just cannot be done by GUI (as we will see). (4) ☺Geeks love to do it this way ☺.

We will start with simple plots and slowly graduate to more complex situations. For completeness, I am starting with description of basic command *plot(x,y)*; and with help of an example, will show the step by step customization of the plot.

1 Basic Plots

To modify the properties of graphs, we will first have to understand the hierarchal structure of graphics in MATLAB®. Each graphical entity is called an object. The computer screen is the root object, highest in hierarchy, on which there can be one or more *figure* objects. Each *figure* object can have one or more *axes* object, on which the data is plot. (On the same hierarchy as *axes*, there can be other objects like, *Uimenu*, *Uicontrol*, *Uipanel*, *Uitoolbar* etc, but these are used in GUI and we won't be discussing them here.) Each *axes* can have have objects like *line*, *text*, *patch*, *surface* etc. Each object has a unique identifier called the handle. So when you create an object by calling its function, its handle is returned. The *handle* has attributes corresponding to different propoerties of the object. For example,

```
f1=figure(1)
```

creates a *figure(1)* on the screen whose handle is f1. You can get a list of all the properties in *figure* by typing *get(f1)* in the command line.

1.1 Setting aspect ratio of figure

Most of the times, it is not required to modify the *figure* properties. But sometimes when you are using a wide screen monitor, the aspect ratio of the figure is also skewed, and the saved figure does not look good when inserted between the text of your paper. For such cases, one way out is to manually resize the figure window every time before saving the figure. Other way is to set the *Position* property in *f1* using the command,

```
set(f1,'Position',[x_lower_left_corner,y_lower_right_corner, width, height])
```

The advantage of this command is that the position as well as size of the figure window can be controlled and if you are plotting multiple figures, their positions can be so arranged that they don't overlap. Also, since the figure is of ready-to-save quality you can directly save it using the save command (which will be discussed later).

1.2 Setting line and axis properties

Plotting a set of data (x, y) as a line is a trivial task, done by *plot(x,y)* command. When the *plot* command is used, *axes* object is created on the current *figure*, and a *line* is created connecting the points in (x, y). The following code generates a figure shown in Fig. 1

```
theta=linspace(0,360,100);
y=sind(theta);
f1=figure(1)
set(f1,'Position',[100,100, 800, 600])
h1=plot(theta,y)
```

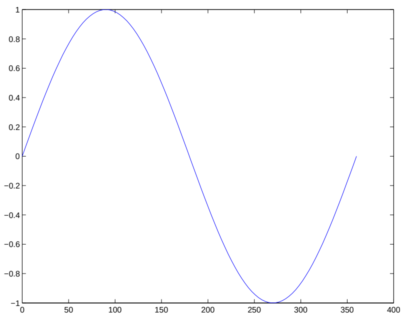


Figure 1:

The sinusoidal line in Fig. 1 has a default thickness of 0.5, and this figure will look pathetic when inserted in two column paper format. Therefore we will have to modify the line by setting properties of the handle *h1*. A list of all the *line* properties can be seen by *get(h1)* command.

```
>> get(h1)
         DisplayName: ''
          Annotation: [1x1 hg.Annotation]
               Color: [0 0 1]
           LineStyle: '-'
           LineWidth: 0.5000
              Marker: 'none'
          MarkerSize: 6
     MarkerEdgeColor: 'auto'
     MarkerFaceColor: 'none'
               XData: [1x100 double]
               YData: [1x100 double]
               ZData: [1x0 double]
        BeingDeleted: 'off'
       ButtonDownFcn: []
            Children: [0x1 double]
            Clipping: 'on'
           CreateFcn: []
           DeleteFcn: []
          BusyAction: 'queue'
    HandleVisibility: 'on'
             HitTest: 'on'
       Interruptible: 'on'
            Selected: 'off'
  SelectionHighlight: 'on'
                 Tag: ''
                Type: 'line'
       UIContextMenu: []
            UserData: []
             Visible: 'on'
              Parent: 170.0012
           XDataMode: 'manual'
         XDataSource: ''
         YDataSource: ''
         ZDataSource: ''
```

To change a property, *set()* command is used. For example, the modified figure after executing the following command, is shown in Fig. 2.

```
set(h1,'LineWidth',2,'Color',[1,0,0],'Marker','s')
```

These properties can also be set at the time of plotting the data. For example,

```
h1=plot(theta,y,'LineWidth',2,'Color',[1,0,0],'Marker','s')
```

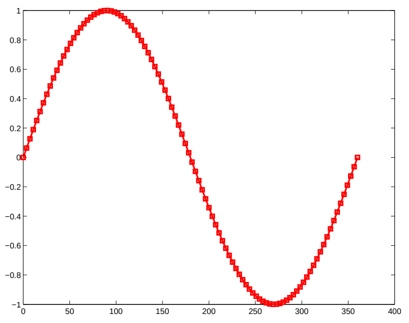


Figure 2:

After setting the line properties the plotted data looks somewhat better, but still the axis is not very clear. Therefore for acceptable quality of figure, the font size of the axes has to be adjusted. Since we did not explicitly create the axes, we don't have its handle. To get the handle of the current axes, command *gca*(get current axes) is used.

```
ax1=gca
set(ax1,'FontSize',24)
```

Execution of the above command modifies the Fig. 2 to the figure shown in Fig. 3

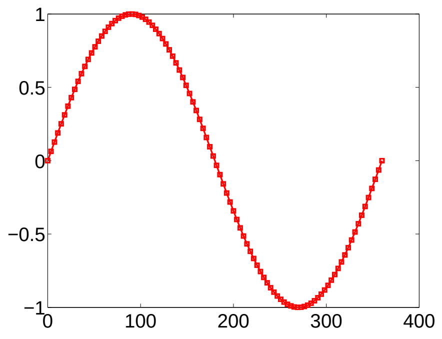


Figure 3:

Now the figure looks fine, but still the axis is not prefect. We have x-axis data only till $\theta = 360°$, and therefore a blank graph appears between 360 and 400. To correct this we will set the axis of the graph using command *axis([xmin xmax ymin yman])*. In case you want to manually set only the x axis or the y axis, use the command *xlim([xmin xmax])* or *ylim([ymin ymax])*, respectively.

We also need the labels on the x and y axis of the plot. These are included by the functions *xlabel('x-text')* and *ylabel('y-text')* respectively. To set the title of the figure *title('Title-text')* is used, but since title is not included in standard publication format, we won't be including it here. Execution of following commands generates a plot shown in Fig. 4. Note that MATLAB®recognizes standard *LaTeX* commands.

```
axis([0 360 -1 1])
xlabel('\theta (degrees)')
ylabel('sin(\theta)')
```

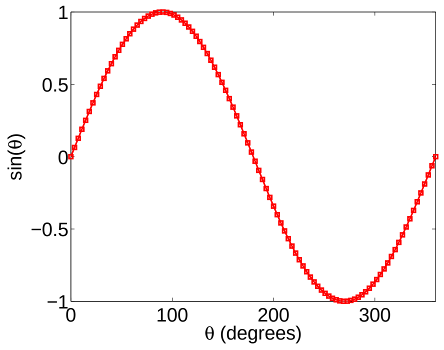
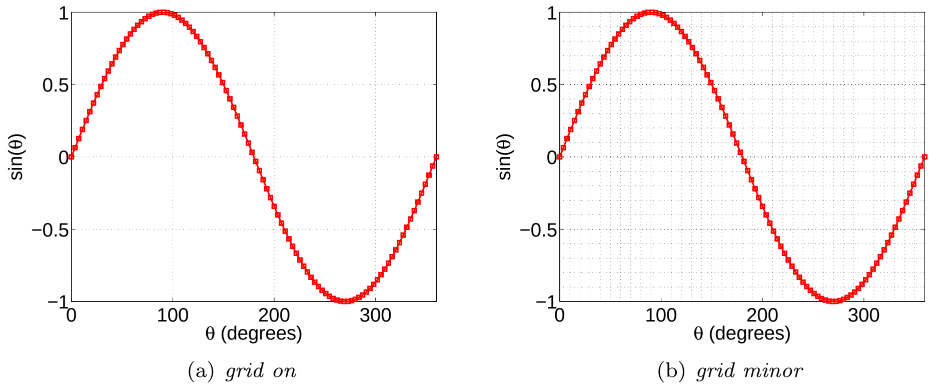


Figure 4:

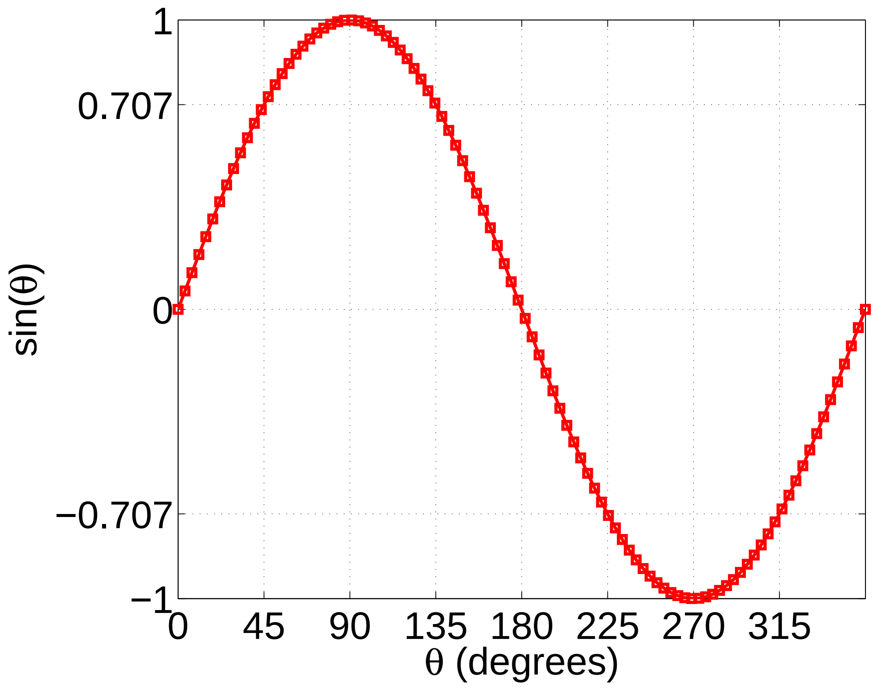
The figure now looks great; but suppose, if you want to find from the graph; what is the value of θ when $\sin(\theta) = 0.5$, you will have some difficulty in tracing accurate θ values on the x-axis. The solution for this is to have a grid on the axis, so that the tracing of data values is easier. A grid on the graph is created by the command *grid on.* To create a fine grid with more resolution, *grid minor* is used.



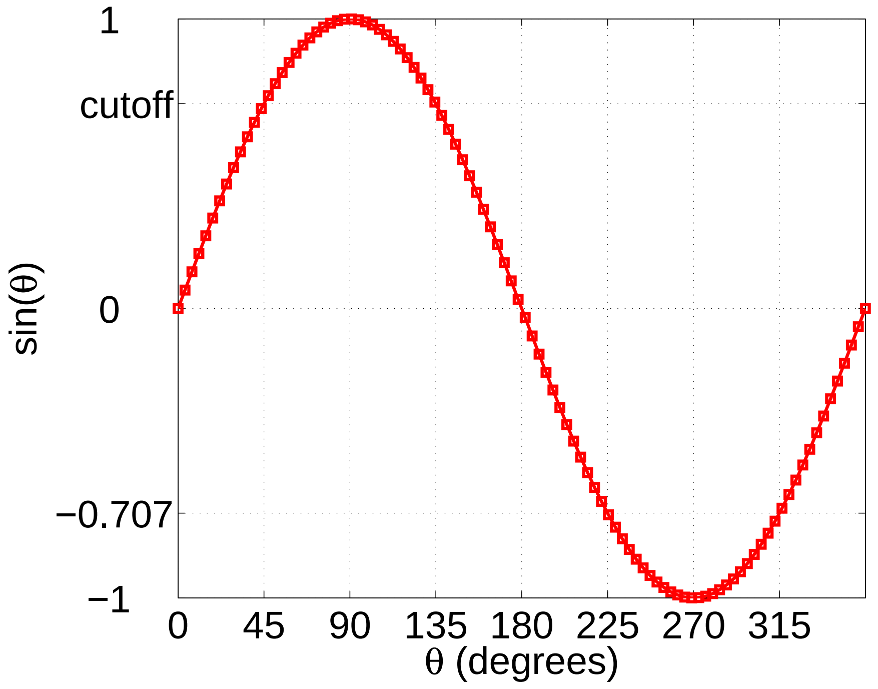
(a) *grid on* (b) *grid minor*

Figure 5:

Sometimes certain set of values on the coordinate axis are more important than others and you need to emphasize them. This can be done by modifying the x and y tick on the axis. Even the label of the ticks can be controlled. For example if you want the ticks on x-axis at multiple of 45° and y-ticks at -1, $-1/\sqrt{2}$, 0, $1/\sqrt{2}$ and 1; you can set the *XTick* and *YTick* properties of the axes handle as shown in Fig. 6(a). Even the label on the ticks can be adjusted as shown in Fig. 6(b). Just make sure that the number of arguments in *YTickLabel* is equal to the number of arguments in *YTick.*



(a) *set(ax1,'Xtick',[0, 45, 90, 135, 180, 225, 270, 315, 370], 'YTick',[-1 -0.707 0 0.707 1])*



(b) *set(ax1,'YtickLabel',['-1 '; '-0.707'; '0 '; 'cutoff';'1 '])*

Figure 6:

2 Multiple Plots

The simplest way to plot multiple lines on the same axis is to use *plot(x1,y1,x2,y2)*. The following code snippet gives an example.

```
theta=linspace(0,360,100);
y1=sind(theta);
y2=sind(2*theta);

f1=figure(1)
set(f1,'Position',[100,100, 800, 600])

h1=plot(theta,y1,theta,y2,'LineWidth',2)
set(h1(1),'Marker','s')
ax1=gca
set(ax1,'FontSize',24)
axis([0 360 -1 1])
xlabel('\theta (degrees)')
ylabel('sin(\theta)')
set(ax1,'Xtick',[0, 45, 90, 135, 180, 225, 270, 315, 370], 'YTick',[-1 -0.707 0 0.707 1])
grid on
```

Here *h1* is a 2×1 array of *handles* corresponding to the respective two sets of data. To modify the properties of *y2*, attributes of handle *h1(2)* are modified. Fig. 7 shows the generated plot.

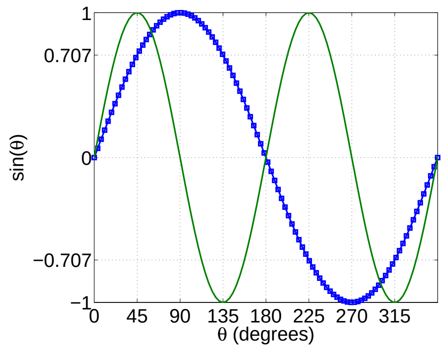


Figure 7:

2.1 Writing Vectorized Code

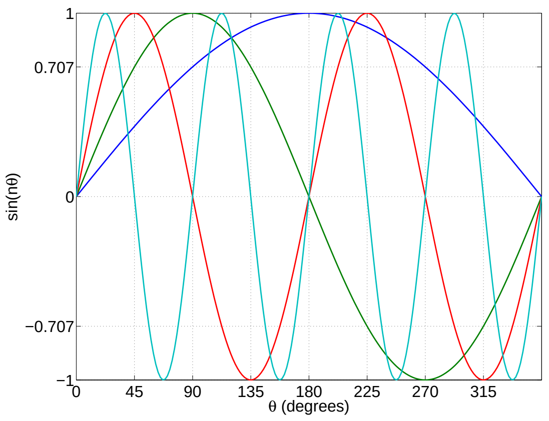
The above mentioned method of plotting multiple data sets is good if you have only two or three sets. However, when the number of data sets increases, assigning each data set a separate variable and plotting them through these variables can be a cumbersome process. Also, if you need to increase or decrease the number of data sets later, you will have to make changes in the *plot* command.

A solution to this problem is to write a vectorized code. A vectorized code not only reduces the number of statements and *for* loops for generating data, but also the generated vectorized data is easy to plot. The following is an example of a vectorized code. The following code plots $\sin(n\theta)$ for increasing values of n.

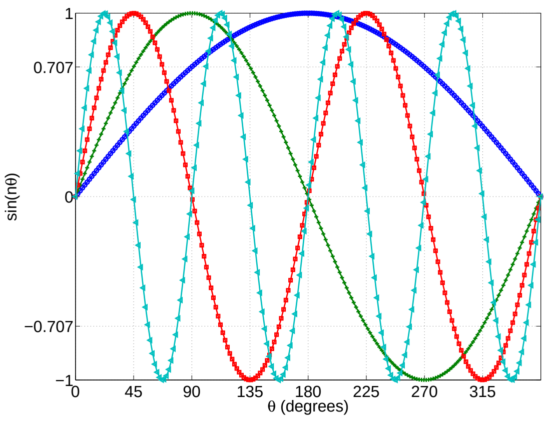
```
theta=linspace(0,360,200);
n=[0.5, 1, 2,4];
```

```
y=sind(n'*theta);
f1=figure(1)
set(f1,'Position',[10,10, 1200, 900])
h1=plot(theta,y,'Linewidth',2)
ax1=gca
set(ax1,'FontSize',24)
axis([0 360 -1 1])
xlabel('\theta (degrees)')
ylabel('sin(n\theta)')
set(ax1,'Xtick',[0, 45, 90, 135, 180, 225, 270, 315, 370], 'YTick',[-1 -0.707 0 0.707 1])
grid on
```

When we multiply the transpose of n and $theta$ (i.e. $n' * theta$), we are doing matrix multiplication of column vector n' of size 4×1 and row vector $theta$ of size 1×200. As a result, y is a 4×200 matrix, with variation of θ along rows and variation of n along the columns. The command $h1 = plot(theta, y, ' Linewidth', 2)$, generates four plots and returns $h1$ of size 4×1, with each value of $h1$ corresponding to one data set. The plot generated by the above code is shown in Fig. 8. Note that the colors in Fig. 8 are selected automatically. You can modify the colors and other line properties using *set* command.



(a) Before setting the markers



(b) After setting the markers

Figure 8:

Many times, apart from the color identifier, marker identification is also required. For journals in printed form, marker identification is very important as the printed edition is in gray scale. Therefore it is highly recommended that each data set should have separate color as well as marker identifier. The following code snippet sets the markers for the different data sets.

```
marker_array=['o','+','s','<','^','>','d'];
for count=1:length(n)
    set(h1(count),'Marker',marker_array(count))
end
```

Now if you change the number of elements in n, the code will automatically adjust the plots. Other properties can also be set using similar code.

‘Hold on’

Sometimes the data sets that we want to compare are from different sources. For example, one set of data might be generated by formulation & equations, and other data set might be experimental. In this case the above mentioned two approaches cannot be used. For such cases we use the command *hold on* and overlap the data on the same plot.

2.2 Legend

Inserting legends for a vectorized code is quite simple. The standard syntax for inserting legend is *legend(string1,string2,string3, …‘Location’,LOC)*. This ‘string1, string2, string3’ can also be arrays of strings. For the example that we are considering, the following command introduces the legend in our figure. Note that, the command will automatically update the values of n in the *legend* string. Fig. 9 shows the final figure.

```
legend([char(ones(length(n),1))*‘n= ’, num2str(n’)])
```

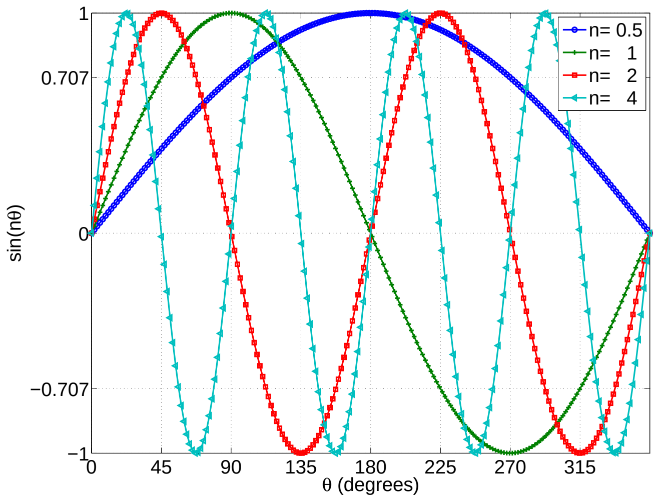


Figure 9:

2.3 Subplots

A *figure* can have more than one *axes*, each with its own *axis* and *line* properties. The simplest way of creating multiple axes on the same figure is to use the command *subplot(M,N,i)*; where M is the number of rows of *axes* in the *figure*, N is the number of columns of *axes* in the *figure*, and i is the index of axes that we want to use. i varies from 1 to $M \times N$. The following code snippet creates the $\sin(n\theta)$ plots with different values of n on separate axes as shown in Fig. 10. When we directly use the *plot* command, *subplot(1,1,1) axes* is created by default.

```
for index=1:length(n)
    ax1=subplot(2,2,index)
    plot(theta, y(index,:),’LineWidth’,2)
    set(ax1,’FontSize’,24)
    axis([0 360 -1 1])
```

```
        title(['n =', num2str(n(index))])
        xlabel('\theta (degrees)')
        ylabel('sin(n\theta)')
        grid on
    end
```

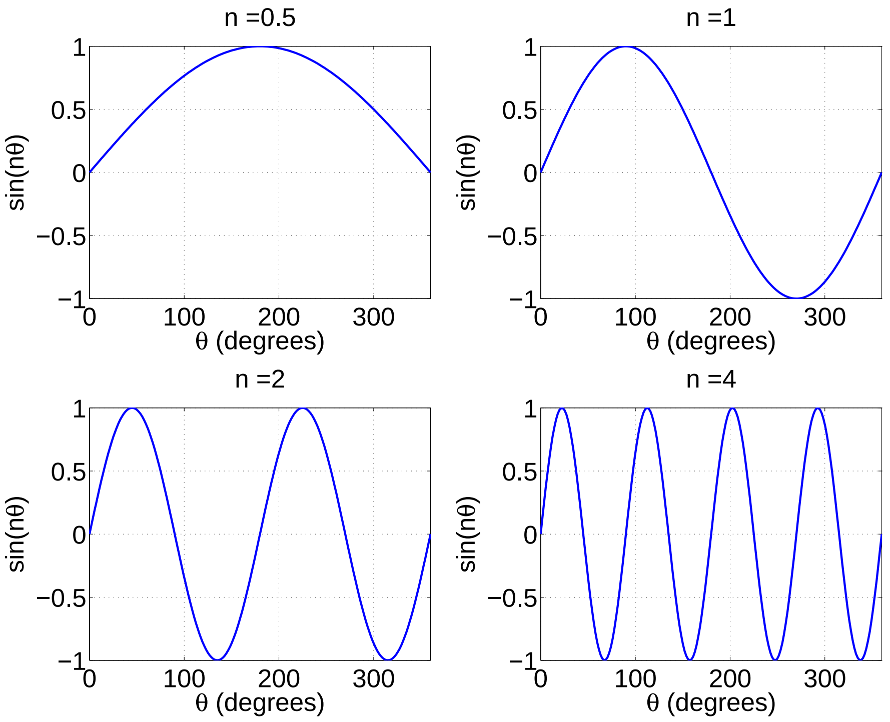


Figure 10:

Another way of creating a more customized *axes* is to use the command *axes('position', [left, bottom, width, height])*; here *left* and *bottom* and the coordinates of the *axes* relative to the lower left corner of the *figure* window, and *width* and *height* are the width and height of the *axes* normalized to 1 with respect to the size of the *figure*. So far we were directly using the commands like *plot*,*axes*, *xlabel*, *ylabel* etc. By default these commands are applicable on the current *axis* of the *figure*. But when we have created more than one *axes*, we either have to explicitly mention the *axes* handle in the command or change the 'current' *axes*, otherwise all these commands will operate only on the latest axis. The *axes* can be explicitly addressed by *plot(axes_handle, xdata, ydata)*. Similar syntax is used for other commands as well. The current axes can changed by the command *axes(axes_handle)*. The following code snippet demonstrates the use of *axes* command. The generated figure is shown in Fig. 11

```
    ax1=subplot(1,1,1); %% create a default axis.
    ax2=axes('position',[0.2,0.25, 0.25 0.25]) % Creates an axis inside ax1
    ax3=axes('position',[0.6,0.25, 0.25 0.25]) % Creates anoter axis inside ax2
    h2=plot(ax1,theta,y(1,:),'LineWidth',2)     % plot data (theta, sin(0.5*theta)) in ax1.
    h2=plot(ax2,theta,y(2,:),'r','LineWidth',2) % plot data (theta, sin(1*theta)) in ax2.
    h3=plot(ax3,theta,y(3,:),'g','LineWidth',2) % plot data (theta, sin(2*theta)) in ax3.
```

```
set(ax1,'FontSize',24)  % set font of axis ax1.
set(ax2,'FontSize',15)  % set font of axis ax2.
set(ax3,'FontSize',15)  % set font of axis ax3.
axis(ax1,[0 360 -1 1])  % set axis limit of ax1
axis(ax2,[0 360 -1 1])  % set axis limit of ax2
axis(ax3,[0 360 -1 1])  % set axis limit of ax3
title(ax1,'An example of customized axis')  % set Title of the main axis ax1.


% set x and y label for the three axis.
xlabel(ax1,'\theta (degrees)')
ylabel(ax1,'sin(0.5\theta)')
xlabel(ax2,'\theta (degrees)')
ylabel(ax2,'sin(\theta)')
xlabel(ax3,'\theta (degrees)')
ylabel(ax3,'sin(2\theta)')

axes(ax1)  % change the current axis to ax1.
grid on
axes(ax2)  % change the current axis to ax2.
grid on
axes(ax3)  % change the current axis to ax3.
grid on
```

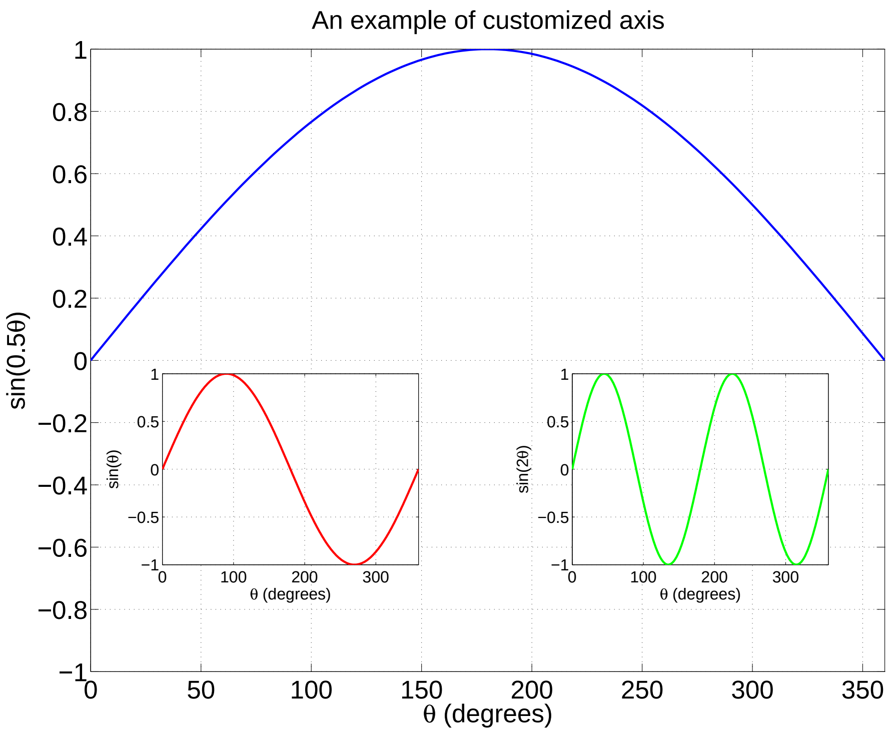


Figure 11:

3 Miscellaneous Plots

3.1 Plotyy

Sometimes, two different quantities which are functions of the a common parameter are required to be compared. Since these two quantities have different units, they cannot be plotted on the same y axis. For such cases it is useful to plot one quantity on the left y-axis and another on the right y-axis. This can be easily done by the command *plotyy(x1,y1,x2,y2)*. The following code snippet gives an example. Here the *plotyy* command returns three handles ax, $h1$ and $h2$. ax is 2×1 row vector corresponding to the left axis and right axis respectively. $h1$ and $h2$ are the handles to the line plots of *(x1,y1)* and *(x2,y2)* respectively. All other plot and axis properties are set using these handles. An important point to note here is that, in this plot you will have to use commands to set the properties, because the figure GUI does not allows access to all the properties, like y label on right y-axis etc. The corresponding generated figure is shown in Fig. 12.

```
x=linspace(0,360,100);
y=cosd(x)+i*sind(2*x);

plot(x,angle(y))

f1=figure(1)
set(f1,'Position',[0, 0 1200 900])

[ax, h1, h2]=plotyy(x,abs(y),x,rad2deg(angle(y)))
set(h1,'LineWidth',3)
set(h2,'LineWidth',3,'LineStyle','--','Color','r')

set(ax,'FontSize',24)
axis(ax(1),[0, 360, 0, 1.5])
axis(ax(2),[0, 360, -180 180])
set(ax,'XTick',[0:45:360])
set(ax(1),'YTick',[0:0.25:1.5])
set(ax(2),'YTick',[-180:45:180],'YColor','r')

grid(ax(1))
grid(ax(2))

xlabel('x-axis (\theta)')
ylabel(ax(1),'Magnitude')
ylabel(ax(2),'Phase (Degrees)')

legend('Magnitude','Phase')
```

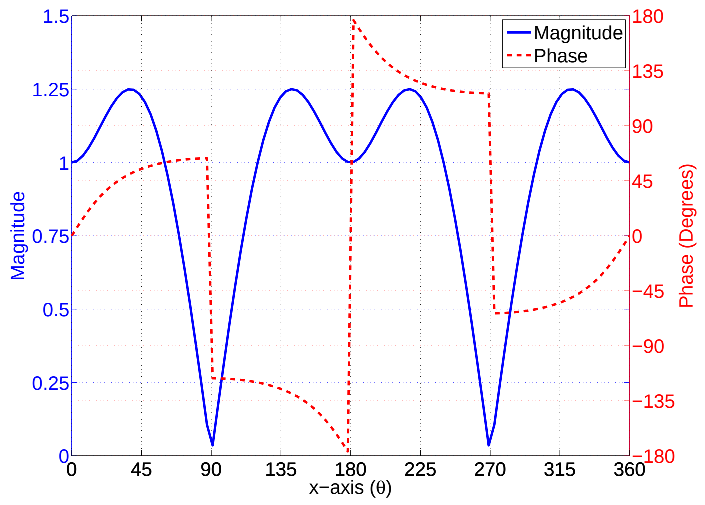


Figure 12:

3.2 Polar Plots

A polar plot is created by the command *polar(theta,r)*. The following code snippet gives an example. The corresponding plot is shown in Fig. 13.

```
theta=linspace(0,2*pi,100);
r=1+cos(theta);
f1=figure(1)
set(f1,'Position',[10 10 1200 900])
h1=polar(theta,r)
set(h1,'LineWidth',2 )
ax1=gca
set(ax1,'FontSize',24)
```

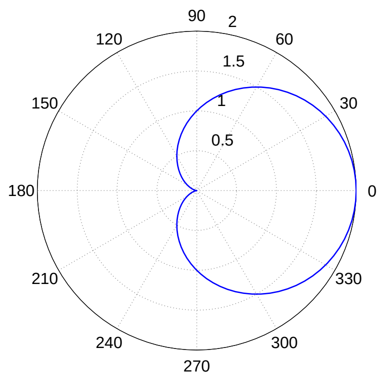


Figure 13:

3.3 Contours

Contour plots are useful in analysis of quantities which are a function of two parameters. In fact, a contour plot is a planar representation of 3-D surfaces. For instance, if Z is a function of x and y, the variation of Z with x and y can be can shown as a surface in a 3-D plot. 3-D plots look good, but offer little useable quantitative information. For example if in a synthesis problem, you want to find the accurate values of x and y for a desired Z, 3-D plots are of little use. For such cases, a contour plot is an effective tool of data representation. The following code snippet gives an example for creating contour plot. The corresponding generated figure is shown in Fig. 14. The values on the contour lines corresponds to the Z values.

```
N=100;
x=linspace(-1,1,N);
y=linspace(-1,1,N);
[X, Y]= meshgrid(x,y);
Z=(X+Y).*exp(-Y.^2);
f1=figure(1)
set(f1,'Position',[10 10 1200 900])
[c1 h1]=contour(X,Y,Z,[-1:0.25:1])      % plots contours at Z level given by the array [-1:0.25:1]
set(h1,'LineWidth',2,'ShowText','on')   % set the line width of the contours and show
                                        % the contour labels
set(gca,'FontSize',24)                  % set the font size of the axis
clabel(c1,h1,'FontSize',18)             % set the font size of the labels on the contour
xlabel('x')
ylabel('y')
grid on
```

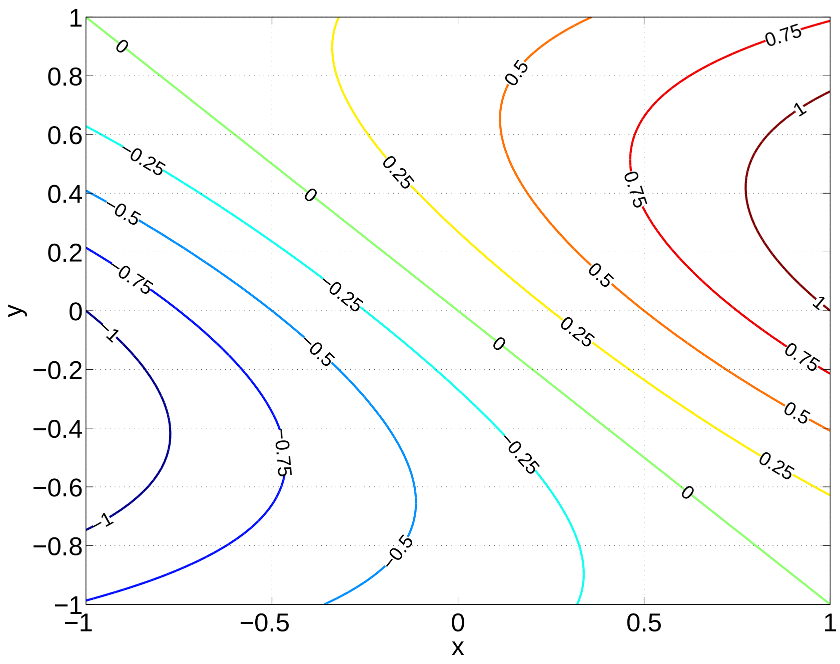


Figure 14:

3.4 Saving Figure

Once you know how to create great plots in MATLAB®, you can save them directly from the code itself using the *print* command. Using command to save figure may appear trivial, but it becomes necessary in two cases. One, when you have to create a figure with a desired dpi (resolution in dots per inch); and two, when you need to repeatedly save the figure for creating animation. The most recommended figure format which is accepted (and demanded) by most journals is *.eps* (encapsulated post script). This is a scalable vector format in which the figure does not pixalize, no matter how much you zoom it. Even if you are preparing your document in word, you must use this format. Apart from the *.eps* format it is highly recommended that you save the MATLAB®version of the figure as *.fig* also. This is important because, later if you have to change the marker or color code of your plots (for compatibility with other plots in your paper), you will not have to re-run your code. The following code snippet gives an example to save figure.

```
print ('-depsc2', '-r300', 'contour_example')   % save figure as contour_example.eps with
                                                 % resolution of  300dpi
print ('-dtiff', '-r300', 'contour_example')    % save figure as contour_example.tiff with
                                                 % resolution of 300dpi
saveas(gcf, 'contour_example', 'fig')           % save figure as contour_example.fig
```

Default resolution of the image saved manually from GUI is 150 dpi. Most publishers require the figure to be at least 300 dpi, and therefore this command is important to get a figure of desired quality.

Afterword

This article, is by no means exhaustive, and a lot more needs to be learnt. An exhaustive and detailed documentation is available on the official site and forum of MATLAB®, and an enthusiastic and keen learner is encouraged to go through it. I hope that this article will be useful for beginners and will help them save some precious time.