Contents

2D LINE PLOTS

One of the benefits of programming in MATLAB is that it has a built-in plotting program. With many other computer languages such as FORTRAN or C, you can easily write data to an external file, but you must use a separate program (e.g., Excel or Gnuplot) to plot and visualize the data.

While MATLAB has extensive plotting capabilities, in MAE10 we will cover on only a few different plotting commands. In particular, we focus on 2D line plots as they provide a quick and easy way to visualize data. There are a few different ways to create 2D line plots in MATLAB, which will be covered in the subsequent sections.

The `plot()` Command

Perhaps the quickest and easiest way to create a 2D line plot is using the `plot()` command. In its most basic form, you need to provide only two arguments to the `plot()` command. Additional arguments can be included to control formatting such as line style, marker symbol, and color. Let's look at the most commonly used form of the `plot()` command:

```
plot(X, Y, 'LineSpec')
```

- The first argument is an array of values that will be used as the x-coordinates.
- The second argument is an array of values that will be used as the y-coordinates.
  - The arrays used as the first and second arguments must be the same size.
- The third argument is optional and can be used to control the line style, marker symbol, and color of the line and markers. If you omit the 3rd argument MATLAB will automatically choose a different color for each data set plotted on the same figure.
  - There are a variety of different letters and symbols that can be included in the third argument to select different markers, line colors, and line styles. For a complete list, consult the MathWorks help page or your textbook.

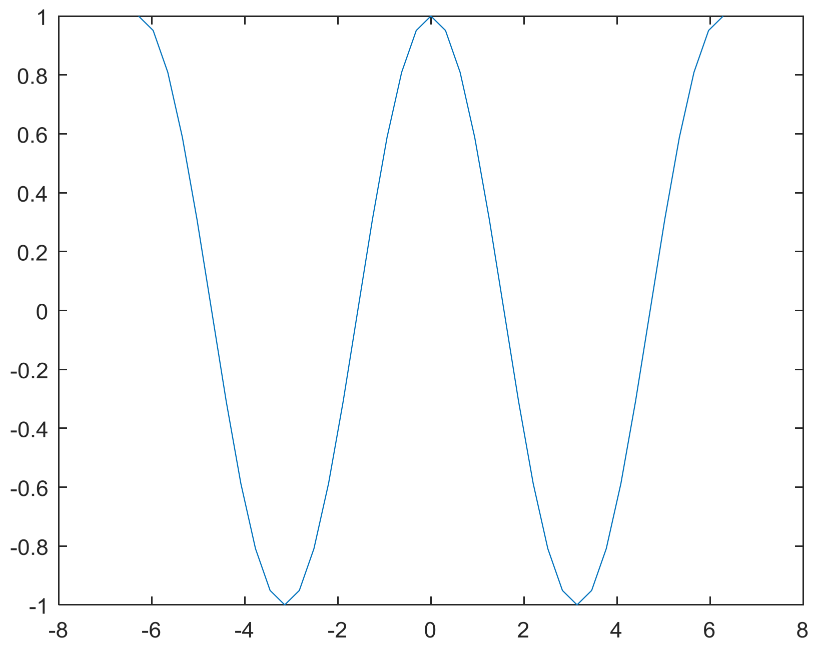
- The third argument is always enclosed in single quotes.

In MAE10, we will use the `plot()` command when we are plotting two *vectors* of data (i.e., a `1xN` array or an `Nx1` array). In this case, the vectors (the 1st and 2nd arguments) must be the same size. This is because for every element in the `X` array, there must be a corresponding element in the `Y` array. It is these ordered pairs that determines each point on the graph, and the collection of points are used to draw the line. If you include markers on your plot, there will be a marker placed at each ordered pair: `(X(1),Y(1))`, `(X(2),Y(2))`, and so on. Once all points are plotted, a line will be drawn connecting all the points. Note that it is possible to plot only markers (no line), only the line (no markers), or both the line and the markers.

The `plot()` command is straightforward to use, so let's look at an example.

In `myfile.m`:
```
x = [-2*pi:0.1*pi:2*pi];
y = cos(x);
plot(x,y)
```

Sample Output:



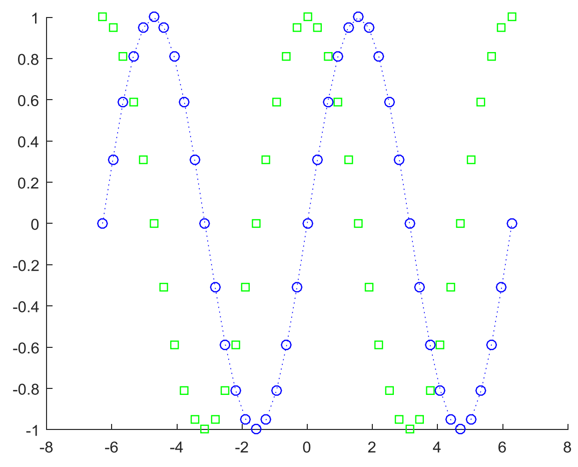
In this first example we simply plot the cosine function in the range `[-2pi,2pi]`. We did not include the third argument, so MATLAB automatically chose a color for the line and there are no markers. Note that the vectors are automatically the same size since the `cos()` function operates by taking the cosine of every element of the input array.

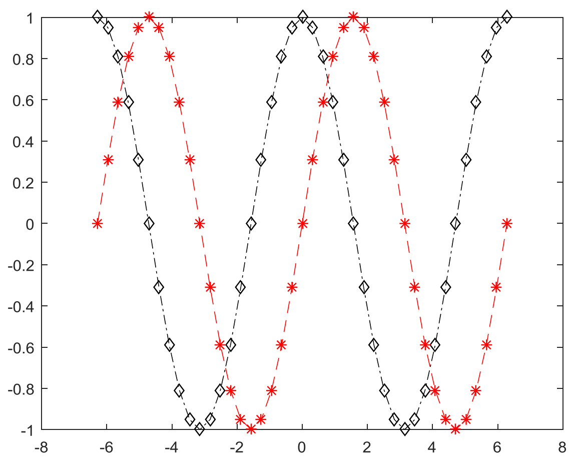
What if we want to plot two lines on the same set of axes? As usual, there is more than one way to achieve the same result. Let's look at another example where we plot the cosine and the sine functions on the same figure and the same set of axes. We will use two different methods to achieve the same result, and we will not yet include the third argument in the `plot()` command.

In `myfile.m`:

```
clear;clc;clf;
x = [-2*pi:0.1*pi:2*pi];
y = cos(x);
z = sin(x);
% Method 1: Use "hold on" with separate plot() commands
figure(1)
hold on
plot(x,y,'gs')
plot(x,z,'bo:')
hold off
% Method 2: Use an expanded plot() command
figure(2)
plot(x,y,'kd-.',x,z,'r*--')
```

Sample Output:





This example includes a lot more than the first example, so let's break it down line by line.

- The first line includes a new command, `clf`, which is used to clear the currently opened figure. This command is useful if you want to clear any previous data from the figure so that the next `plot()` command begins on a blank figure.
- Lines 2, 3, and 4 create the vectors that store the data we want to plot.
- Line 6 introduces the `figure()` command. The `figure()` command is used to open a new figure. This allows you to control where the data plotted using the `plot()` command is drawn. For example, the `figure(1)` command opens Figure 1 in MATLAB such that the next `plot()` command will draw on Figure 1. This command is not required – MATLAB will automatically open new figures as necessary when you execute the plot command. However, it allows you to control precisely where your data is plotted to prevent unexpected overlap.
- Line 7 and 10 introduce the `hold` command. The `hold` command allows you to execute multiple `plot()` commands and "hold" all the lines drawn on the same figure. By default (without `hold on`), MATLAB will overwrite any previous plot drawn on the current figure each time a new `plot()` command is executed. Thus, if we want both `plot()` commands on lines 8 and 9 to be drawn on the same figure, we need to turn `hold on` before executing the plot commands. Note that `hold` operates as a toggle; that is, it will remain on until we turn it off. Once you turn `hold on`, all subsequent `plot()` commands will be drawn on the same figure and same set of axes. Thus, we use the command `hold off` to revert back to the default behavior before using method 2 to do the next plot.
- Line 8 plots the cosine function using green squares as the markers. The third argument, `'gs'`, determine the color (`g` for green) and marker symbol (`s` for squares). There is no line type included so there is no line drawn on the figure.
- Line 9 plots the sine function using a blue (`b`) dashed line (`:`) with circles (`o`) as the markers.
- Line 12 opens a new figure, Figure 2.
- Line 13 plots both the sine and cosine functions at the same time using an expanded version of the `plot()` command. You can easily plot many data sets in a single `plot()` command using this method. You just provide the `X` and `Y` values of each data set in order, and optionally, the line and marker configuration for each. In this case we used a black (`k`), dash-dot line (`-.`) with diamonds (`d`) as the markers for cosine, and a red (`r`), dashed line (`--`) with asterisks (`*`) as the markers for sine. We did not need to use the `hold` command since all data sets listed in a single `plot()` command are automatically drawn on the same figure and same set of axes.

Labeling and Annotating Figures

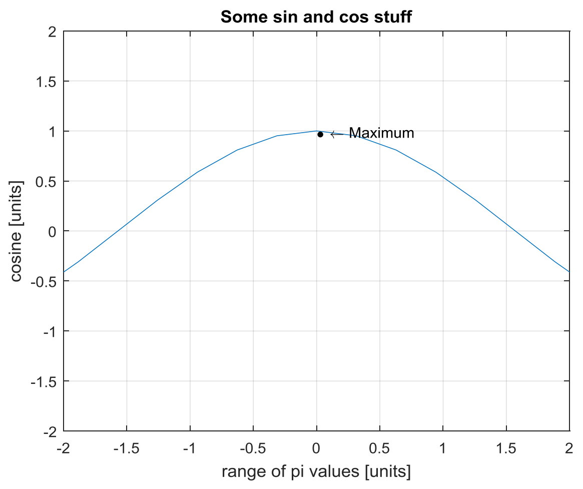
So far we have learned how to plot data sets in 2D using lines and markers, but we are missing one essential element of any and all figures – we have not provided a title or labeled our axes! **When creating any figure, it is essential that you label your axes and provide a descriptive title.** Without these this information, the reader has no idea what the data represents! When possible, always include units on your axes labels. The importance of including this information on any and all figures you generated cannot be over-stressed.

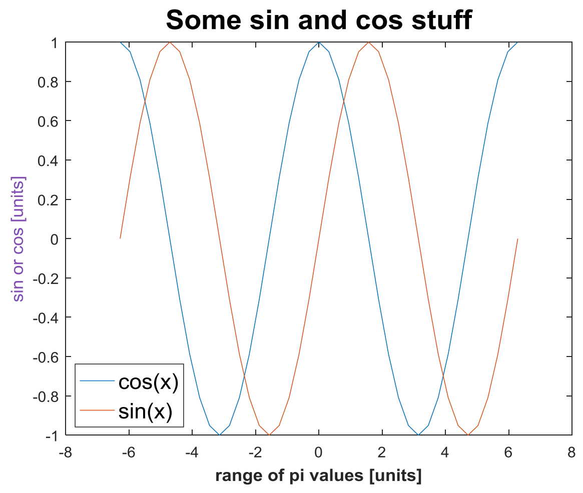
Let's look at an expanded version of the previous examples we include a variety of new commands to label our figure.

In `myfile.m`:

```
x = [-2*pi:0.1*pi:2*pi];
y = cos(x);
z = sin(x);
plot(x,y)
axis([-2 2 -2 2]); % look at a specific range of data
% in the format: axis([minX maxX minY maxY])
title('Some sin and cos stuff'); % default font/size/format
xlabel('range of pi values [units]'); % default font/size/format
ylabel('cosine [units]'); % default font/size/format
txtlab = '\bullet \leftarrow Maximum'; % create string for text label
text(0,1,txtlab) % use text() command to place label string on figure
grid on % toggles on grid lines for this figure
figure(5)
plot(x,y,x,z);
fig5title = title('Some sin and cos stuff');
fig5xlab = xlabel('range of pi values [units]');
fig5ylab = ylabel('sin or cos [units]');
set(fig5title,'FontSize',18); % sets the title font size to 18
set(fig5xlab,'FontWeight','bold'); % changes the xlabel to be bold
set(fig5ylab,'FontName','Helvetica','Color',[.5 .3 .7]);
fig5leg = legend('cos(x)','sin(x)','Location','Best');
set(fig5leg,'FontSize',14);
```

Sample Output:





The comments in the above example provide some description of what each command is doing. Let's break down a few of the key commands and how they are used:

- The `axis()` command allows you to set the range for the `X` and `Y` axes in the format: `axis([minX maxX minY maxY])`
    - Note that you can choose whatever range you like, even if the range is larger or smaller than the range of the data being plotted. Changing the minimum and maximum axis values is essentially how you "zoom" the figure.
- The `title()`, `xlabel()`, and `ylabel()` commands take as input a character string (notice the single quotes around the text) and provide the figure title, x-axis label, and y-axis label, respectively.
- The `text()` command allows you to quickly and easily add text annotations on your figure. The `text()` command takes three arguments: the x and y coordinates of where the text label *begins*, and a character string (or variable that stores a character string) that contains the text and/or symbols that will be put on the figure. In this example we defined the character string as a variable before using the `text()` command, but this is not necessary. You can define the character string directly in the third argument of the `text()` command (don't forget the single quotes in that case). We also used some special characters in our text label: `\bullet` and `\leftarrow` are built in to MATLAB and are useful for pointing at specific locations on your figures.
- For Figure 1 we also turned on grid lines using the `grid` command. The `grid` command operates as a toggle, so you can turn on or off grid lines on a given figure using `grid on` or `grid off`, respectively.
- Notice that we opened Figure 5 for our second plot even though we had only used Figure 1 previously. There is no problem with this – figures do not need to be created in order, and you can control which plot goes onto which figure using the `figure()` command.

- For Figure 5, we mostly use the same commands as Figure 1 but add additional customization by saving the title and axis labels to variables, then place them on the figure using the `set()` command. The `set()` command provides extensive control over the formatting of your labels; you can control font size, font style, font weight (e.g., bold), font color, and so on.
    - Notice that we also include a legend on Figure 5. A legend is not necessary on Figure 1 because we have only one line. However, any figure that has more than one set of data plotted should include a legend.
    - One big advantage of using MATLAB's built-in plotting functionality compared with other programs like Excel is that plots can be easily generated, customized, and formatted using code. In Excel, labeling and formatting plots can be very time consuming as you must manually insert and customize each figure "by hand". In MATLAB, once you have customized labels and figures to your liking, the exact same code can be used to plot other data sets and all of the formatting is done automatically!

The `subplot()` Command

The `subplot()` command is used together with the `plot()` command to generate a figure that contains multiple sets of axes. In the previous examples we plotted multiple data sets on the same set of axes, but had only one set of axes per figure. By using `subplot()`, we can have more than one set of axes on the same figure. This is most commonly used when you are plotting two or more data sets that have widely varying `X` and `Y` values – if we plotted them all on the same set of axes, some of the data would be difficult to see. The `subplot()` command is used before each `plot()` command and essentially directs the `plot()` command to a certain location and set of axes on the figure. The basic form of the `subplot()` command is:

```
subplot(m, n, p)
```

- The first two arguments determine the number of different sets of axes that will appear on the current figure. Thus, the figure is divided into an `mxn` grid. This essentially means there are `m*n` different sets of axes on the figure: if `m=2` and `n=2`, there will be `4` separate sets of axes on the figure, and each set of axes can have one or more plot drawn on it. Plots are generated with the `plot()` command, using everything we discussed previously.
    - The first argument determines the number of "rows" of subplots (from top to bottom).
    - The second argument determines the number of "columns" of subplots (from left to right).
- The third argument determines the position of the next plot that will be drawn within this `mxn` grid. Note that MATLAB numbers its subplots by row. For example, in a `2x2` grid of subplots, the subplots are numbered in the following way:

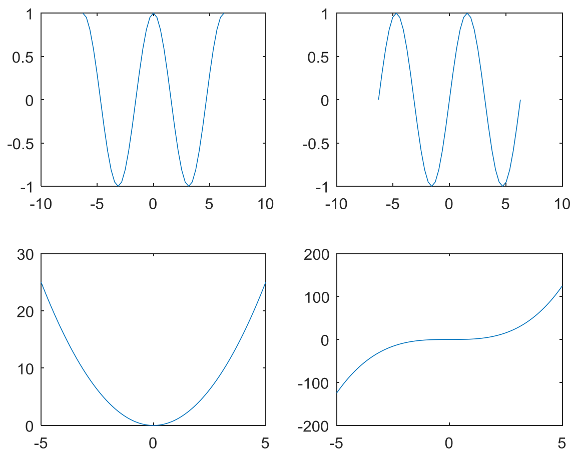
`p = 1`	`p = 2`
`p = 3`	`p = 4`

The functionality of the subplot command is best visualized with an example, so let's jump right in.

In `myfile.m`:

```
x = [-2*pi:0.1*pi:2*pi];
y = cos(x);
z = sin(x);
a = -5:0.1:5;
b = a.^2;
c = a.^3;
subplot(2,2,1)
plot(x,y)
subplot(2,2,2)
plot(x,z)
subplot(2,2,3)
plot(a,b)
subplot(2,2,4)
plot(a,c)
```

Sample Output:



In this example we generated one figure that contains four subplots in a `2x2` grid (since `m=2` and `n=2`). Notice the vastly different scales on the y-axis between the various subplots; if we plotted all these functions on the same set of axes with the axis range required for the 4th subplot, the sine and cosine functions would be almost impossible to see. Once again this is the main benefit of subplots: we can plot various datasets with widely varying values all on the same figure and easily compare them side-by-side. Notice that the `subplot()` command precedes the `plot()` command. It is the `subplot()` command that divides the figure into a grid size determined by the first two arguments, and sets the position of the next `plot()` command with the third argument. The next `plot()` command is what actually draws the graph in the position specified by the `subplot()` command. Everything we learned previously for the `plot()` command can be applied identically to subplots, including doing multiple plots on the same set of axes (using either method).

The `polarplot()` Command

The `polarplot()` command is very similar to the `plot()` command but it takes arguments in the polar coordinates `(theta,rho)` rather than the Cartesian coordinates `(x,y)`. The basic form of the `polarplot()` command looks similar to that of the `plot()` command:

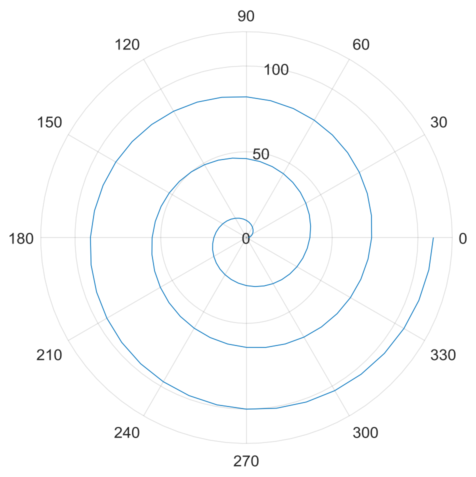
```
polarplot(theta, rho, 'LineSpec')
```

- The first argument indicates the angle ***in radians***.
  - Be careful! Be sure to always convert your angle from degrees to radians if necessary before using the `polarplot()` command.
- The second argument is the radius value for each point. This is typically just referred to as "`r`" in mathematics, where the ordered pair is given as `(r,θ)`.
- The third argument is the same as in the `plot()` command. It allows you to customize line type, marker style, and line/marker color.

Let's look at an example of creating a polar plot.

In `myfile.m`:
```
angle = [0:10:3*360];
angle_rad = angle*(pi/180);
n = numel(angle);
radius = 1:n;
polarplot(angle_rad,radius)
```

Sample Output:



In this example we specify the angle in degrees initially, then convert it to radians before using it to create the polar plot. Notice that our line spirals around a total of three times, which is governed by the values in our `angle` array – each revolution is `360` degrees, and our final value is `3*360`. The distance from the center point is determined by the values in the `radius` array. We start at `1` unit away from the center, and end at `109` units away, growing further from the center point as we traverse the line from `(0,1)` to `(1080, 109)`. While you may not use the `polarplot()` command as much as the regular `plot()` command for generating 2D line figures, it can be quite useful when working with certain geometries. Note also that MATLAB has built-in functions to transform between Cartesian and polar coordinates, such as the `cart2pol()` command. Formatting and labeling polar plots is done using the same methods discussed previously for the `plot()` command.

There are countless ways to construct, configure, label, and format plots in MATLAB. What is described in these notes are some of the basic and most commonly used forms of the commands used to create line plots in 2D along with detailed examples with explanations. As mentioned previously, MATLAB has extensive plotting capabilities, but many of these are beyond the scope of MAE10. MATLAB can easily create 3D plots, animated plots, contour plots, surface plots, and mesh plots among many others. Because the creation and manipulation of the plots can be scripted (e.g., written in code), MATLAB is an efficient way to generate publication quality figures. Once a script has been created, it can be easily adapted to other data sets with minimal extra work, saving time in the long run. Thus, when using MATLAB to generate high quality figures, be sure to save your M-Files for future reference! As time goes on, you will build up an extensive collection of plotting scripts, meaning that most visualization work will not require you to write code from scratch. In contrast, other programs like Excel often require the user to manually (e.g., using your mouse) configure plots, which is very time consuming.