Graphics Functions

Review - Graphics Formatting Functions

Review - Graphics Formatting Functions

Scatter Plots

Scatter plots differ from line plots in that the data points are not connected, and there are more options to customize the individual markers.

```
>> scatter(x,y,sz,col,"filled")
```

Inputs

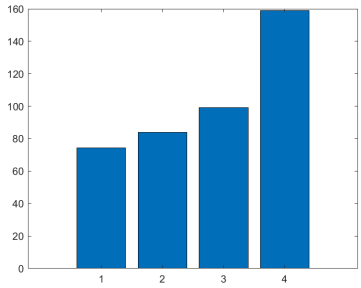
`x`	x-data
`y`	y-data
`sz`	The size of the markers can be a constant or vector specifying the size of each point individually.
`col`	The color of the markers can be a constant or vector specifying the color of each point individually. In the latter case, colors are picked from a color map.
`"filled"`	Provide this optional fifth input to specify that the markers be filled in.

You can use the command `colorbar` to include a bar that indicates the value of the colors represented in the color map. Use the `Name,Value` pair, `"MarkerFaceAlpha",a` to add transparency to the markers.

Bar Plots

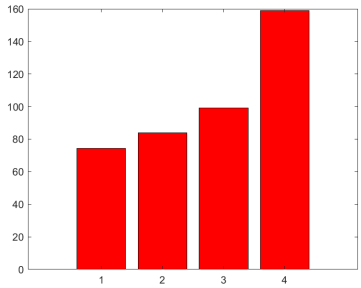
Create a bar plot from a vector.

```
bar(vec)
```



Change the color of a bar plot.

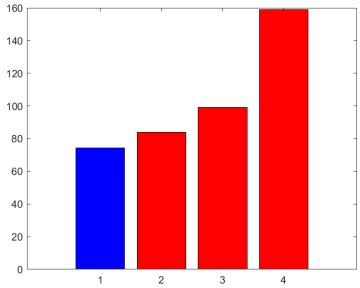
```
bar(vec,"FaceColor","r")
```



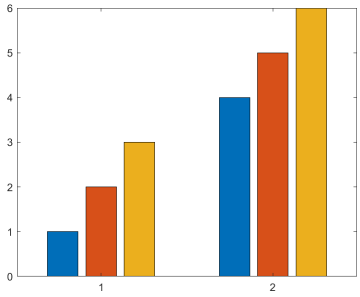
Change the colors of individual bars.

```
colors = [0 0 1; 1 0 0; 1 0 0; 1 0 0];
bar(world,"FaceColor","flat","CData",colors)
```

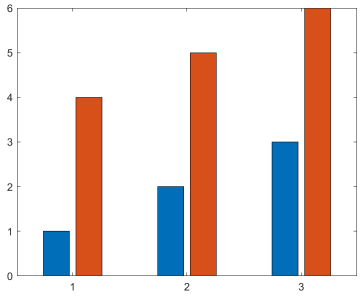
Create bar charts for each row of a matrix.



```
mat = [1 2 3; 4 5 6];
bar(mat)
```



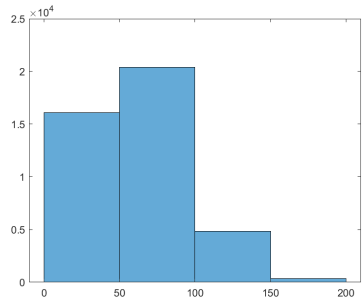
```
bar(mat')
```



Histogram

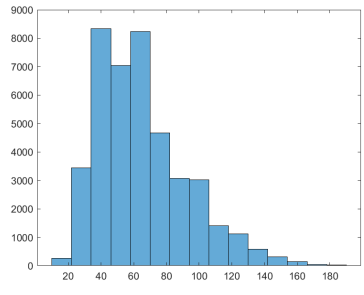
Create a histogram with a specified bin width.

```
histogram(vec,"BinWidth",50)
```



Create a histogram with a specified number of bars.

```
histogram(vec,15)
```



Customizing Graphics Objects

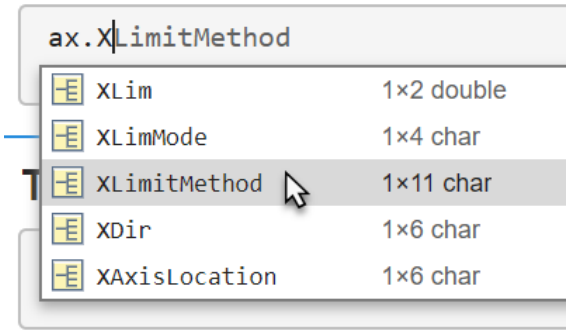
Review - Customizing Graphics Objects

Review - Customizing Graphics Objects

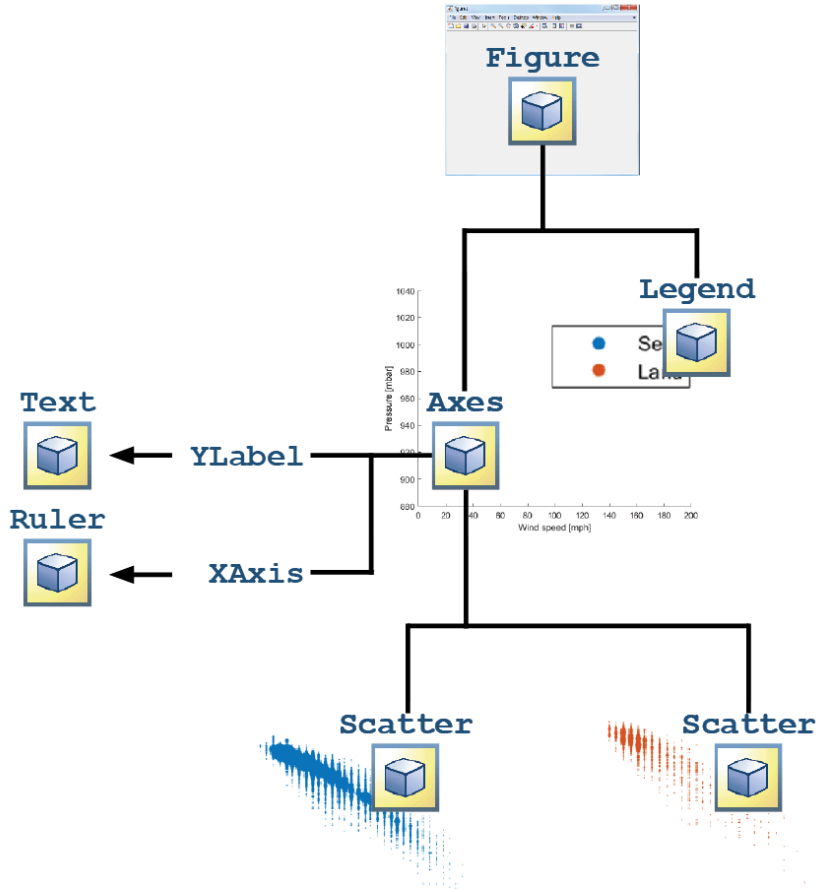
Different types of visualizations and graphics objects have different properties, which you can extract, use, and update. You can get a handle to the graphics object by assigning the output of a plotting function to a variable.

If you know in advance you want to edit the figure, you can create one before plotting.	`f = figure;`
You can create figure handles to graphics objects when you create the plot.	`p = p(x,y);`
Access and modify properties with dot notation.	`p.LineStyle = ":";`
Add data to a plot. This can be used for creating animations.	`p.XData(end+1) = 1;` `p.YData(end+1) = 2;`
If your graphics object is an array, you can access different elements with indexing.	`b = bar(mat);` `bar2 = b(2);`
Set a property for every element in an array of graphics objects.	`set(b,"FaceAlpha",.5)`
Get the current axes.	`ax = gca;`

One advantage to using graphics objects is that you can scroll through the properties.



All graphics objects are part of a hierarchy. Most graphics objects consist of a figure window, containing one or more axes, which contain any number of plot objects.



You can access the current figure and axes with `gcf` and `gca`. To move between levels of the hierarchy, use the `Children` and `Parent` properties.

Defining Categories of Data

Review - Defining Categories of Data

Review - Defining Categories of Data

Categories and Sets

Categorical arrays use less less memory and work with many plotting functions.

Use the `categories` function to get a list of unique categories.

Merge different categories with the `mergecats` function.

Rename categories with the `renamecats` function.

```
x = categorical(["medium" "large" "large" "red" "small" "red"]);
```

```
c = categories(x)
```

```
c =
  4×1 cell array

    {'large' }
    {'medium'}
    {'red'   }
    {'small' }
```

```
x = mergecats(x,["small" "medium" "large"],"size")
```

```
x =
  1×6 categorical array

     size      size      size      red      size      red
```

```
x = renamecats(x,"red","color")
```

```
x =
  1×6 categorical array

     size      size      size      color      size      color
```

Discretizing Continuous Data

Ranges in continuous data can represent categories. Categorize continues data into discrete bins with the `discretize` function.

```
>> y = discretize(X,edges,"Categorical",cats)
```

Outputs		Inputs	
`y`	If the `"Categorical"` option is set, `y` is a categorical array. Otherwise, `y` is numeric bin values.	`X`	Array of continuous data. `X` is usually numeric or datetime.
		`edges`	Consecutive elements in `edges` form discrete bins. There will be one fewer bins than the number of edges specified. You can use `Inf` in `edges` to create a bin with no edge.
		`"Categorical",cats`	Optional input for the name of each bin category.

Analyzing Groups within Data

Review - Analyzing Groups within Data

Review - Analyzing Groups within Data

Calculations on Groups in a Dataset

```
>> gs = groupsummary(tble,groupvars,method,datavars)
```

Outputs		Inputs	
`gs`	A table of calculations for each group, including a column for group counts	`tble`	A table of data
		`groupvars`	The variables you want to group by. Can be a string vector (or scalar) of variable names, integer vector of indices, or logical vector indicating the grouping variables
		`method`	The aggregation method
		`datavars`	The variables you want to apply the aggregation method to

You can also specify group *bins* for the grouping variable, which can be especially helpful with datetime variables.

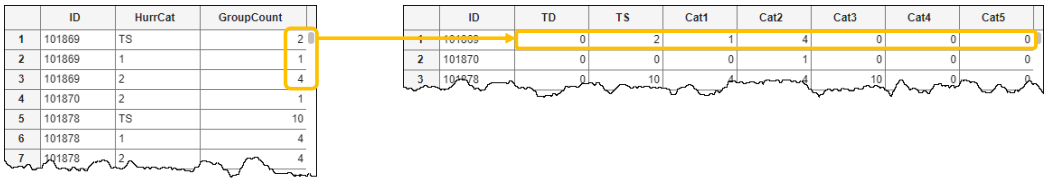
```
>> gs = groupsummary(tble,groupvars,groupbins,method,datavars)
```

Outputs		Inputs	
`gs`	A table of calculations for each group, including a column for group counts	`tble`	A table of data
		`groupvars`	The variables you want to group by. Can be a string vector (or scalar) of variable names, integer vector of indices, or logical vector indicating the grouping variables
		`groupbins`	The binning scheme. For `datetime` variables, this can be a unit of time, like `"minute"` or `decade`.
		`method`	The aggregation method
		`datavars`	The variables you want to apply the aggregation method to

The output of `groupsummary` sometimes requires post-processing like removing or renaming variables with `removevars` and `renamevars` respectively.

Organize Aggregated Data by Grouping Variable

After calculating group statistics with multiple variables, you may want to *unstack* the aggregated data into a new table with the grouping variables defining the rows and columns.




```
>> tbl2 = unstack(tble,datavars,colvar)
```

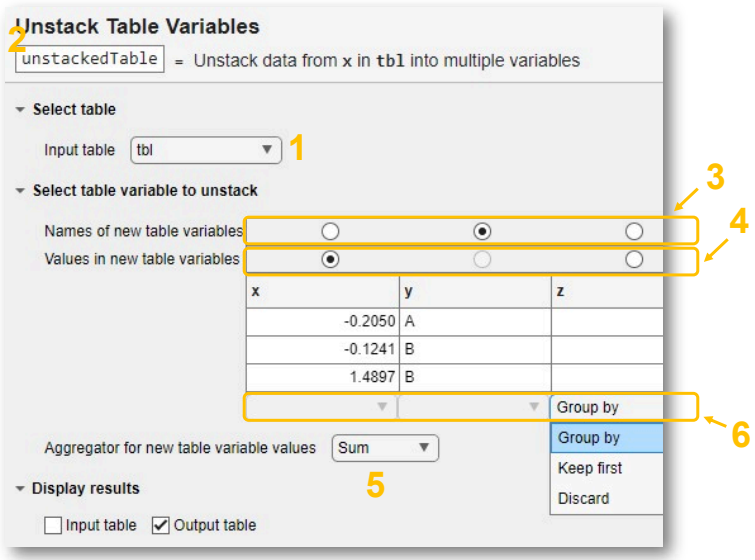
Outputs	
`tbl2`	A new table with data from `datavars` grouped into columns defined by `colvar`

Inputs	
`tble`	A table of data
`datavars`	The variables you want to be in the entries of the table
`colvar`	The variable that defines the new columns

Unstack Table Variables Task

You can use the **Unstack Table Variables** task to unstack aggregated data from a table interactively. You should specify the following:

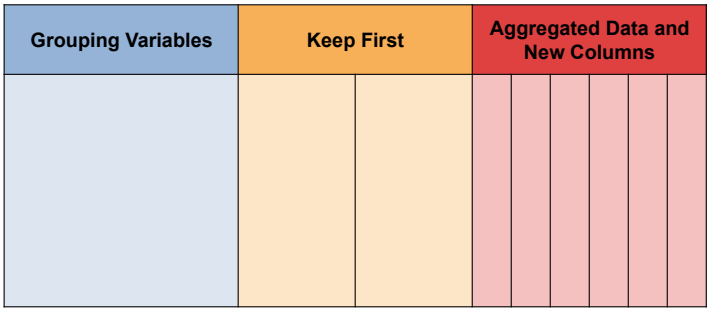
1. **Input table**: Specify the input, or original, data
2. **Output table name**: This is the name of the output table.
3. **Names of new table variables**: This is the variable that identifies the new column names. It is often a `categorical`.
4. **Values in new table variables**: This is the variable to be unstacked. It will become the entries in the new table.
5. **Aggregator for new table variable values**: Specify the aggregation function to be applied to the **Values in the new table variables**.
6. How to include it in the output table. The options are:
   - `Group by`: This variable will be used as a grouping variable and stored in the first column.
   - `Keep first`: The first entry of this variable for the specified group will be returned in the output table.
   - `Discard`: This variable will not appear in the output table.



Combining Aggregated Data and Initial Conditions

Using the **Unstack Table Variables** task, you can specify variables to group by, aggregate, and keep their first entry. In the output table, each type of variable is stored together. From left to right:

1. grouping variables
2. "keep first" variables
3. the new variables containing aggregated data



Importing Data from Multiple Files

Review - Importing Data from Multiple Files

Review - Importing Data from Multiple Files

A datastore is a reference to a file or set of files. The `datastore` function informs where to find the files.

Code	Description
`ds = datastore(filename)`	Reference a single file
`ds = datastore(directory)`	Reference a folder of files
`data = read(ds)`	Read data incrementally
`data = readall(ds)`	Read all data referenced in datastore

If your data isn't formatted the way `datastore` expects, you can set the datastore properties. Examples of common properties are shown below. You can find all the properties in the the documentation.

```
>> ds = datastore(filename,"Delimiter","-","TextscanFormats","%D%C%f","SelectedVariableNames",var)
```

Outputs

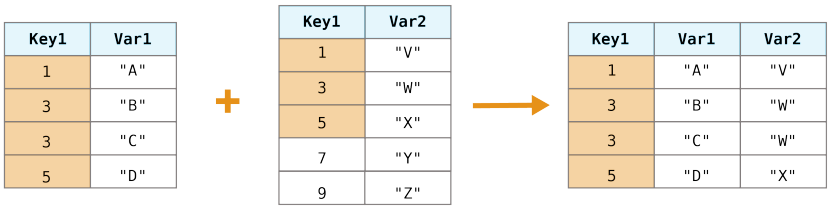
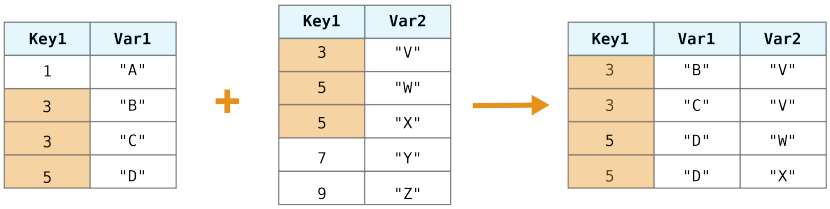
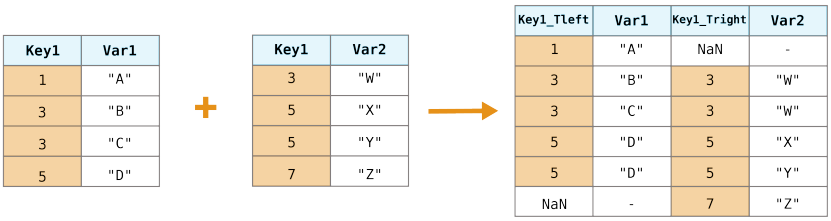
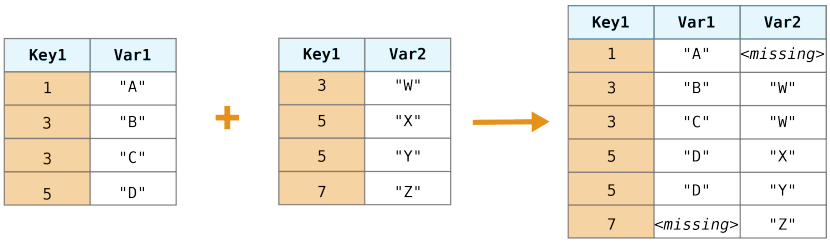
`ds`	Reference to a collection of data.

Inputs

`filename`	File location.
`"Delimiter","-"`	Delimiter is one or more characters that separate data values in the file.
`"TextscanFormats","%D%C%f"`	Import variables using the output class in the format specification string.
`"SelectedVariableNames",var`	Import only the variables listed in `var`.

Merging Data

Once you read in multiple tables, you may want to join them together. You can join two tables in many ways. The various `join` functions are listed in the table below.

Function	Example
`join` `Key1` in `Tright` must have unique values and contain every key in `Tleft`.	
`innerjoin`	
`outerjoin` Two key variables are created.	
`outerjoin` with `"MergeKeys"` on	

Images and 3-D Surface Plots

Review - Images and 3-D Surface Plots

Review - Images and 3-D Surface Plots

Images or 3-D plots generally begin with x, y, and z data. In many cases, the x and y data are not evenly spaced on a grid.

To interpolate the data onto a grid, start by defining the grid points. Here, `yvec` is denser than `xvec`.

The `meshgrid` function will convert your vectors into the grid expected by `surf` and `pcolor`.

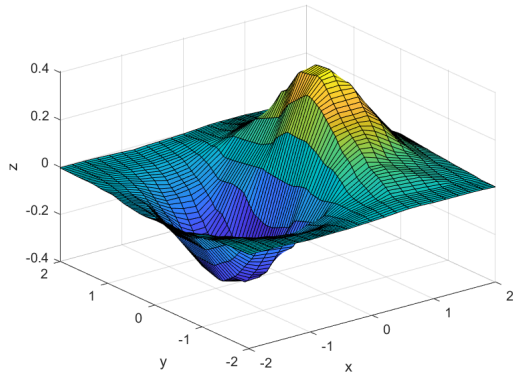
Then use the `griddata` function to interpolate your data onto the grid.

Consistent naming of your variables from previous steps will the `griddata` syntax easier.

Once your x, y, and z data is gridded, you can visualize it in a variety of ways. `surf` creates a surface plot.

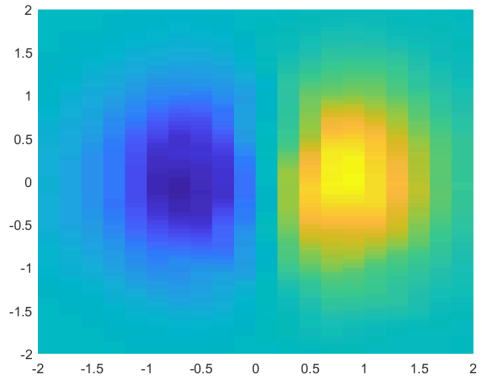
```
data = readtable("my3Ddata.csv")

        x           y            z
    _______    ________    _________

      2.2506    -0.30105      0.012974
     -1.3443    -0.79976      -0.11638
     0.53421    -0.92891       0.16945
   -0.070088    -0.67461     -0.044245
         ...         ...           ...

xvec = -2:.2:2;
yvec = -2:.05:2;


[xgrid,ygrid] = meshgrid(xvec,yvec);



zgrid = griddata(data.x,data.y,data.z,xgrid,ygrid);









surf(xgrid,ygrid,zgrid);
```

Notice the difference between the `x` and `y` axes. This is because `xvec` and `yvec` had a different number of grid points.



You can also visualize your 3-D data as an pseudocolor image.

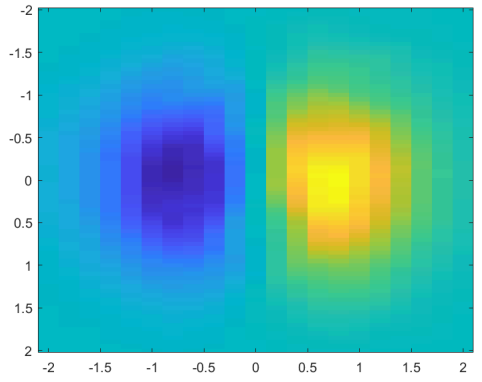
```
im = pcolor(xgrid,ygrid,zgrid);
im.EdgeAlpha = 0;
```



This scaled image contains the same data, but the first two inputs are the vectors of grid points instead of the output from `meshgrid`.

If you inspect the right yellow shape, you can see that the `imagesc` plot is vertically flipped from the `pcolor` plot.

```
imagesc(xvec,yvec,zgrid);
```



Importing Unstructured Data

Review - Importing Unstructured Data

Review - Importing Unstructured Data

To import data from files where the formatting changes and must be inferred from the data itself, you can use functions that allow you to interact directly with files.

Open the file and store the file identifier. You'll use `fid` with the other low-level import functions.

```
fid = fopen("myfile");
```

You can import files line-by-line using `fgetl`.

```
firstLine = fgetl(fid)
```

There is a file position indicator that keeps track of where you're located in the file, so calling `fgetl` twice will return the first two lines.

```
firstLine =

    '09/12/2005 Level1 12.34 45 1.23e10 inf'

secondLine = fgetl(fid)

secondLine =

    '10/12/2005 Level2 23.54 60 9e19 -inf  0.001'
```

To return back to the beginning of the file, you can rewind the file position indicator.

```
frewind(fid)
```

If you know the format of the data, you can pass a format specification string to `textscan`.

```
formatSpec = "%{MM/dd/uuuu}D %s %f32 %d8 %u %f";
myData = textscan(fid, formatSpec)

myData =
  1×9 cell array

      {3×1 datetime}    {3×1 cell}    {3×1 single}    {3×1 int8}    {3×1 uint32}    {3×1 double}
```

When you're finished importing, make sure you close the file connection.

```
fclose(fid);
```