Matlab III: Graphics and Data Analysis

Updated: August 2012

Table of Contents

Note: See Matlab I: Getting Started for more information about this tutorial series including its organization and for more information about the Matlab software. Before proceeding with this tutorial, download Part_3_Graphics_and_Data_Analysis.zip . This zip file contains the example files needed for this portion of the tutorial.

Section 7: Graphics and Data Visualization

Matlab has a high level graphics capability that allows users to display data in various forms without having to incorporate extensive information into a command or into scripts. This easy procedure uses default values for graphical objects in Matlab's object oriented graphics system, Handle Graphics. For more customized and advanced use, the values can be specified or changed on the command line or in the text of m-file scripts; and there is also a point and click property editor GUI for users to change object values in order to alter display characteristics. Display possibilities include 2-D plots, 3-D plots, visual aids such as pie charts and histograms, contours, and animation. In addition there are many attributes of objects that can be customized, including scaling, colors, fonts, perspective angles, lighting and shading, and so forth.

Two dimensional plotting

The high level graphics for two dimensional plotting accommodate displays of pairs of data sets in rectangular linear Cartesian coordinates, on a semilog axis system, on a log-log axis system, and in polar coordinates. The elementary syntax is

```
>> plot(x,y)         % linear abscissa and ordinate
>> semilogx(x,y)     % logarithmic abscissa (x)
>> semilogy(x,y)     % logarithmic ordinate (y)
>> loglog(x,y)       % both x and y logarithmic
>> polar(theta,rho)  % polar graphing
```

If only one vector argument is supplied, it is considered to be the second (dependent) variable, i.e., the ordinate or radial distance, and the sequence position in the vector is used for the corresponding independent variable, i.e., abscissa or counterclockwise angle. When a vector containing complex valued quantities is plotted, Matlab ignores the imaginary part and the display represents only the real part. Trailing character string options in the argument list can specify a line or marker color or type.

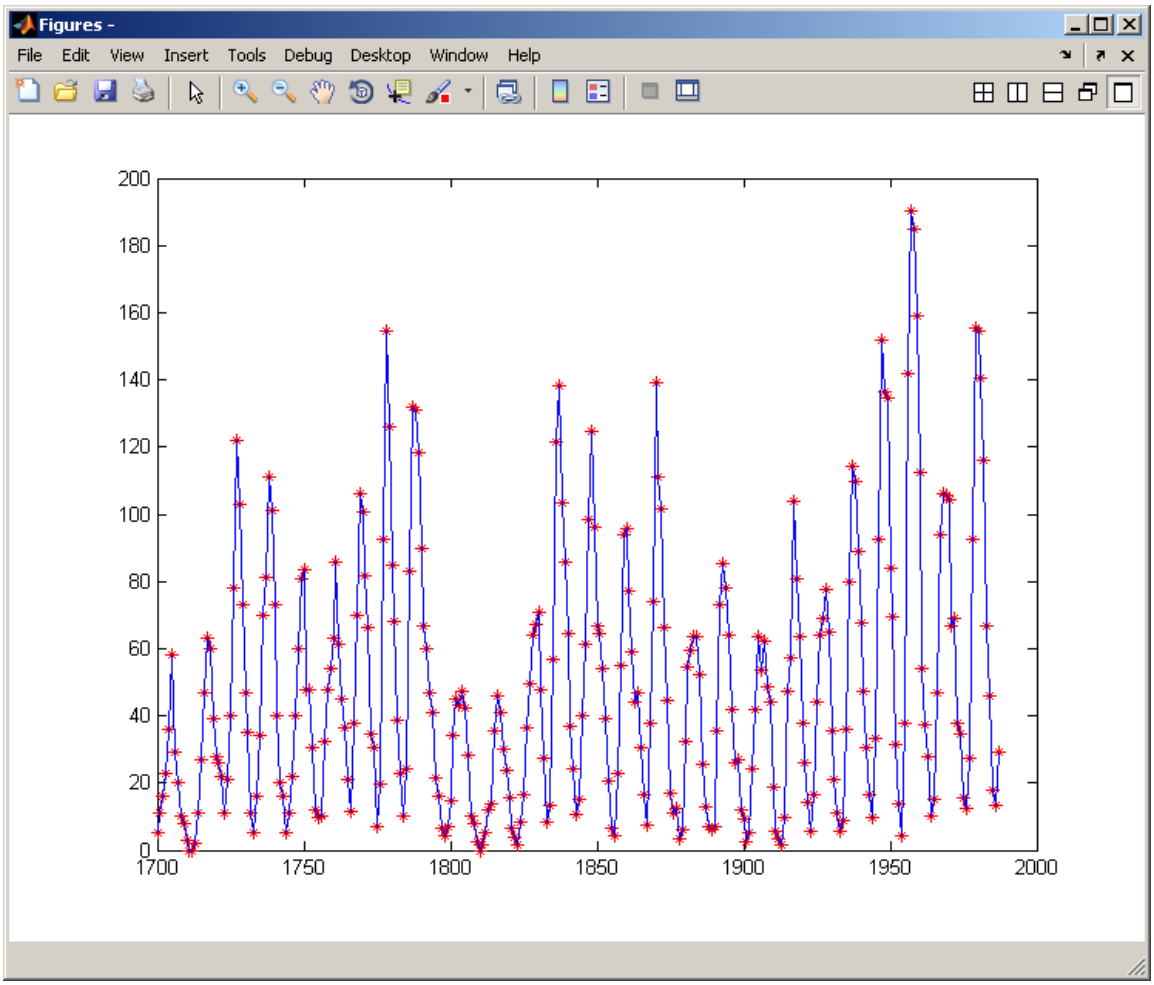
As an example, let's load a file that is distributed with Matlab for demonstration purposes, *sunspot.dat*, and make various two dimensional plots. The file is a simple

288 row by 2 column matrix where the first column has consecutive years from 1700 until 1987 and the second column has the mean sunspot number for that year. Then we will construct *year* and *spots* vectors from the two columns and plot *spots* as a function of *year*.

```
>> load sunspot.dat;
>> year = sunspot(:,1);
>> spots = sunspot(:,2);
>> plot(year,spots);
```

This would give a 2-D plot displayed as a solid blue line going through the points. We could obtain a 2-D plot with point markers if we added a third string argument, e.g., 'r*' for data points being displayed as red asterisks. There is a toggle for getting a new plot (hold off) and for superimposing on an existing plot (hold on). Assume that we want a second plot with the data markers superimposed on the line plot, with the same scaling. Then we would use the additional commands

```
>> hold on;
>> plot(year,spots,'r*')
```

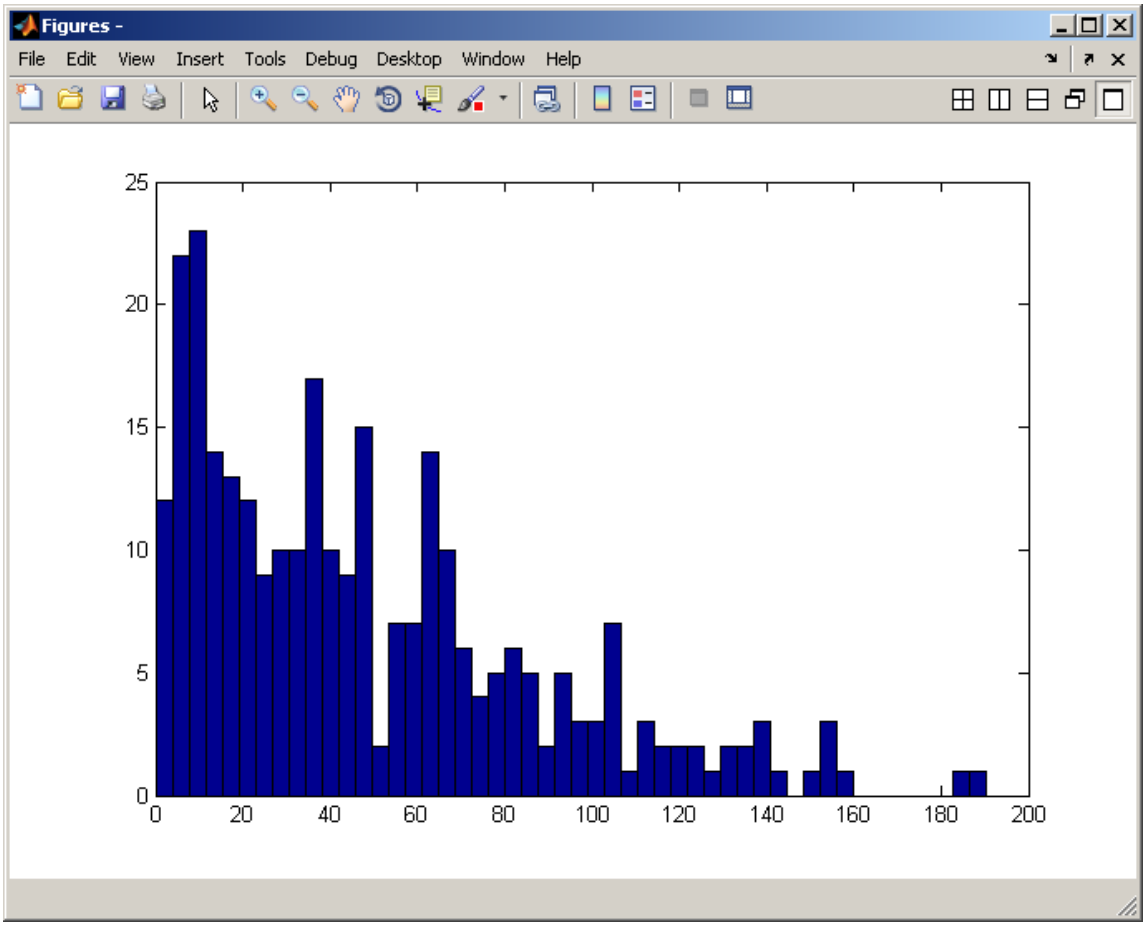


There are also several display options for discrete data that are commonly used in presentations for general or business audiences

bar for a vertical bar graph
barh for a horizontal bar graph
stem for a stem plot
area for display of vectors as stacked plots
pie for a pie chart
hist for a histogram in Cartesian coordinates
rose for a histogram in polar coordinates

For example, to get a histogram of sunspot number during the time span with 50 bins you could use the command

```
>> hist(spots,50)
```



Sub plotting

It is possible to display several subplots simultaneously in a rectangular array within a single figure using the *subplot* command. This command takes three numerical arguments, followed by a comma and a regular plotting command – thus the form

```
>>subplot(n1,n2,n3), plotfunction(x, y, ...)
```

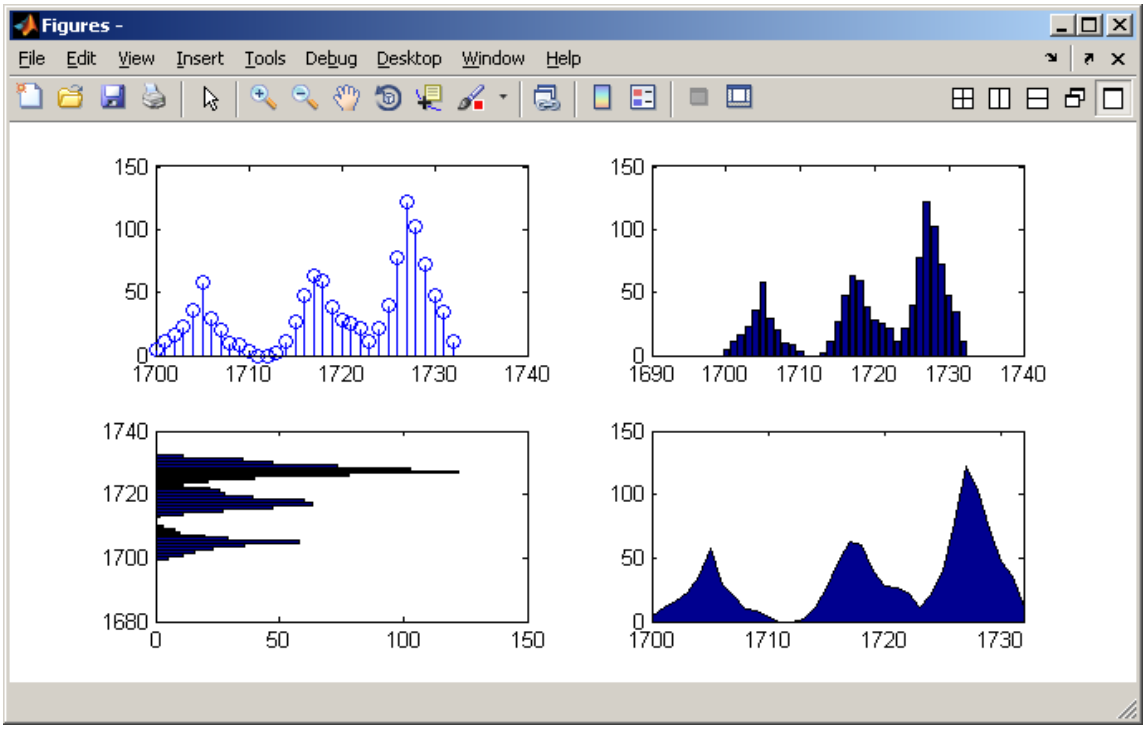
where *n1* is the number of rows in the subplot array, *n2* is the number of columns in the subplot array, *n3* is the position within the array for the particular subplot, and the *plotfunction* is a regular plotting function such as *plot*, *stem*, *bar*, etc. The position is defined as

n3 = (row_position – 1)**n*2 + column_position

so that the upper left position has *n3* = 1, the upper right position has *n3* = *n2*, the lower left position has *n3* = (*n1* -1)**n2* + 1 and the lower right position has *n3* = *n1***n2*.

Some of the plotting variants can be illustrated compactly with a subplot, using the first few cycles of approximately 11 years:

```
>> hold off;
>> subplot(2,2,1), stem(year(1:33,1), spots(1:33,1));
>> subplot(2,2,2), bar(year(1:33,1), spots(1:33,1));
>> subplot(2,2,3), barh(year(1:33,1) ,spots(1:33,1));
>> subplot(2,2,4), area(year(1:33,1), spots(1:33,1));
```



Patching and Filling

An area of a plot can be filled with a specified color or pattern if its boundary can be described as a closed polygon. The command

```
>> patch(x,y,'color');
```

will fill the polygon having vertices at [x(1),y(1)], [x(2),y(2)], … with the specified *color*, which can be a character code such a '*r*' for red, or an RGB vector such as [1 0 0] for red. This patching applies to the current figure, whether *hold* is *on* or *off*. Similar to this is the command *fill*,

```
>> fill(x,y,'color');
```

but this is a plot command itself and if *hold* is set to off the current figure is replaced by a new one containing only the colored polygon.

Three dimensional plotting

Matlab has several commands used in displaying data in three dimensions. Among these are routines for displaying lines with a 3-D perspective, making surface plots of a function of two independent variables, and making contour plots.

```
>> plot3(x,y,z)   % line in three dimensional space
>> mesh(x,y,Q)    % wire frame surface for q(x,y)
>> surf(x,y,Q)    % quadrilateral surface rendering
>> contour(x,y,Q) % 2D projection of equal height lines
```

The *plot3* command requires 3 vector arguments, the corresponding elements of which form coordinates for positioning in 3 dimensional Cartesian space. Thus, all three vectors must be the same length. In contrast, the arguments for the surface plots are two optional vectors of independent variable values which form an underlying grid and a required matrix whose elements represent a dependent variable value for each coordinate pair on the grid. If the optional independent variable vectors are not specified then the default of element position is used. For example, if an *n* x *m* matrix *Q* is the sole specified argument, then vectors [1:*n*] and [1:*m*] will be used to form the underlying grid.

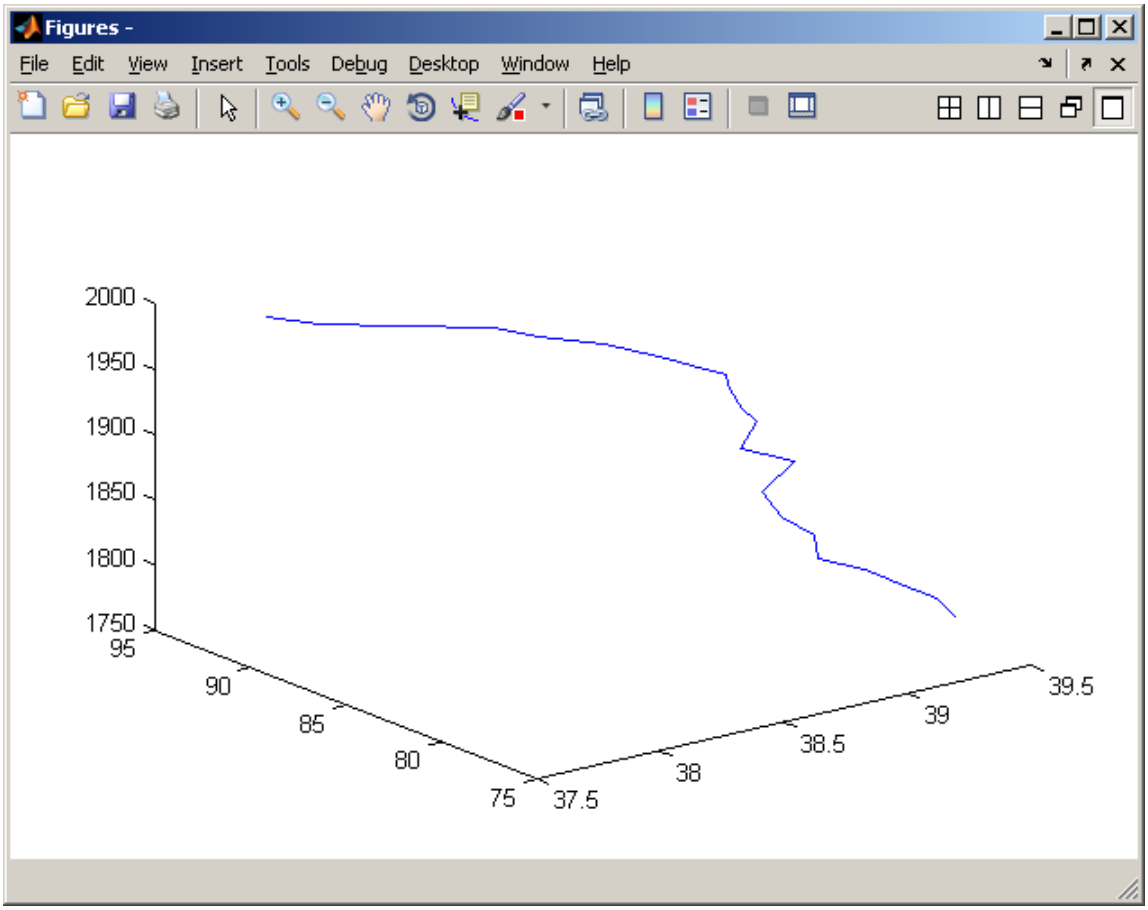
Matlab does not come packaged with an experimental data set equivalent to the sunspot data that can be used for illustrating 3-D plotting of three vectors or surface maps for two parameter variables. However there are a couple of suitable files accessible from our website. Download Part_3_Graphics_and_Data_Analysis.zip and find the following documents:
popcenter.dat
wireless.xls

Place these files in a location within your Matlab path. The first of these files is a matrix of U.S. Census Bureau data with a column for all the decennial census years (1790-2000) and separate columns for the geographical coordinates of the center of U.S. population for those census years. The second column is the north latitude coordinate and the third column is the west longitude coordinate. The second of these files is an Excel spreadsheet giving the mean number of simultaneous users of the University's wireless network by day of the week and hour of the day during the Fall 2004 semester.

As an illustration of plotting a line in 3-D space, set each of the three columns of the popcenter.dat file to separate vectors and then use those as the arguments for *plot3* command:

```
>> clear;
>> load popcenter.dat;
>> decade = popcenter(:,1);
>> north = popcenter(:,2);
>> west = popcenter(:,3);
>> hold off;
>> plot3(north, west, decade);
```

The screen display will use default values for observation angle where the minimum value for the first and second arguments is closest to the viewer.

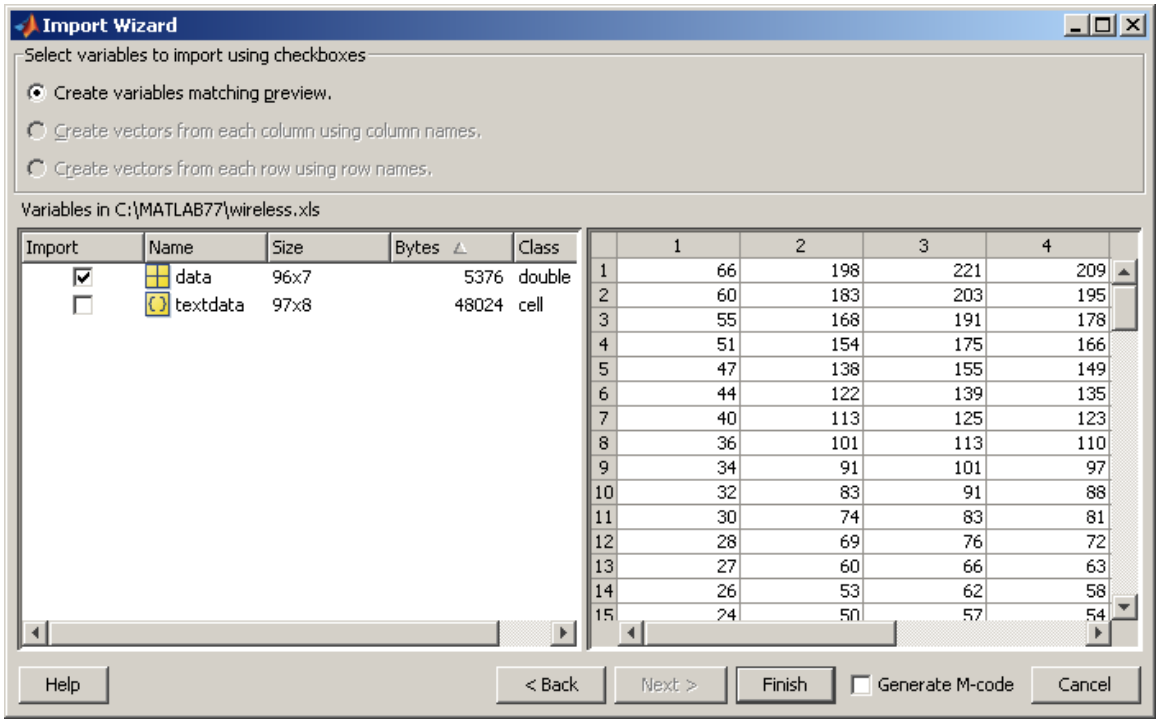


This shows the southwestward drift of the center of population in the U.S. with each new census, particularly in recent decennial years. More perspective of the 3-D nature of the curve can be obtained from rotating it in 3-D by selecting the Rotate3D item on the Figure Window's Tool menu bar or by clicking the rotation button on the Figure Window's icon tool bar, then moving the cursor to the figure area and moving the mouse around while depressing its left key.

To illustrate surface plotting, let us import the data from wireless.xls. A simple procedure for doing this is to open the file with the command

```
>> open('wireless.xls')
```

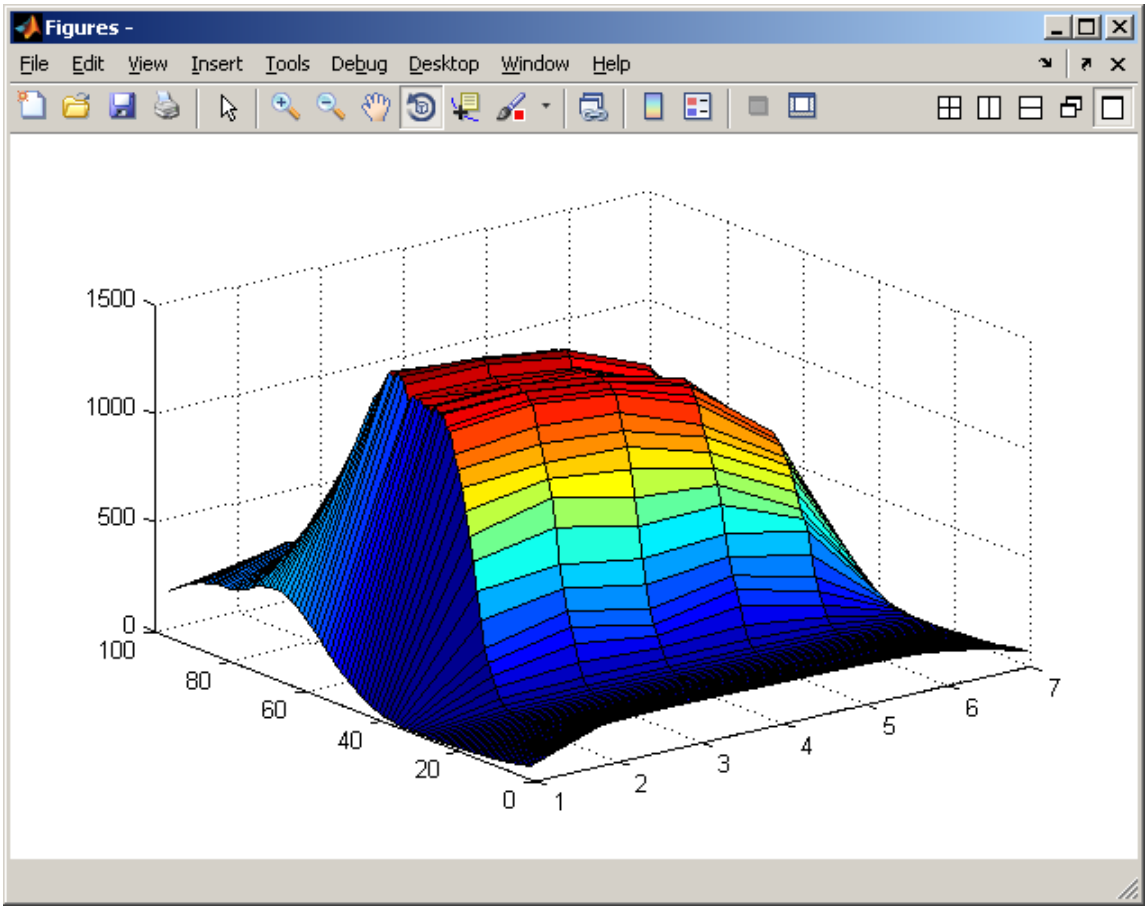
and then select the variable *data* (a 96 x 7 matrix) for import. This has the mean wireless usage volume numbers for the 96 quarter hours of the day for each of the 7 days of the week during the Fall 2004 semester at UT.



Click on the *Finish* button to import this matrix into the Matlab workspace. Then let us create a surface plot of the imported variable *data*:

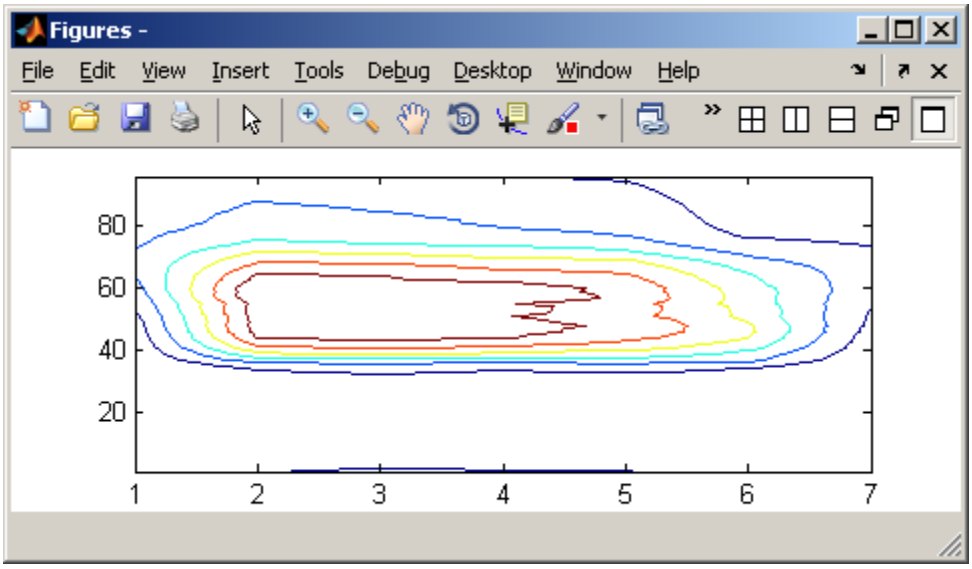
```
>> surf(data)
```

The 3-D surface will subsequently appear in the Figure Window.



Just as with the line example, the surface can be rotated in 3-D using the same procedures. If a contour projection onto the grid plane is desired, the *contour* command can be used

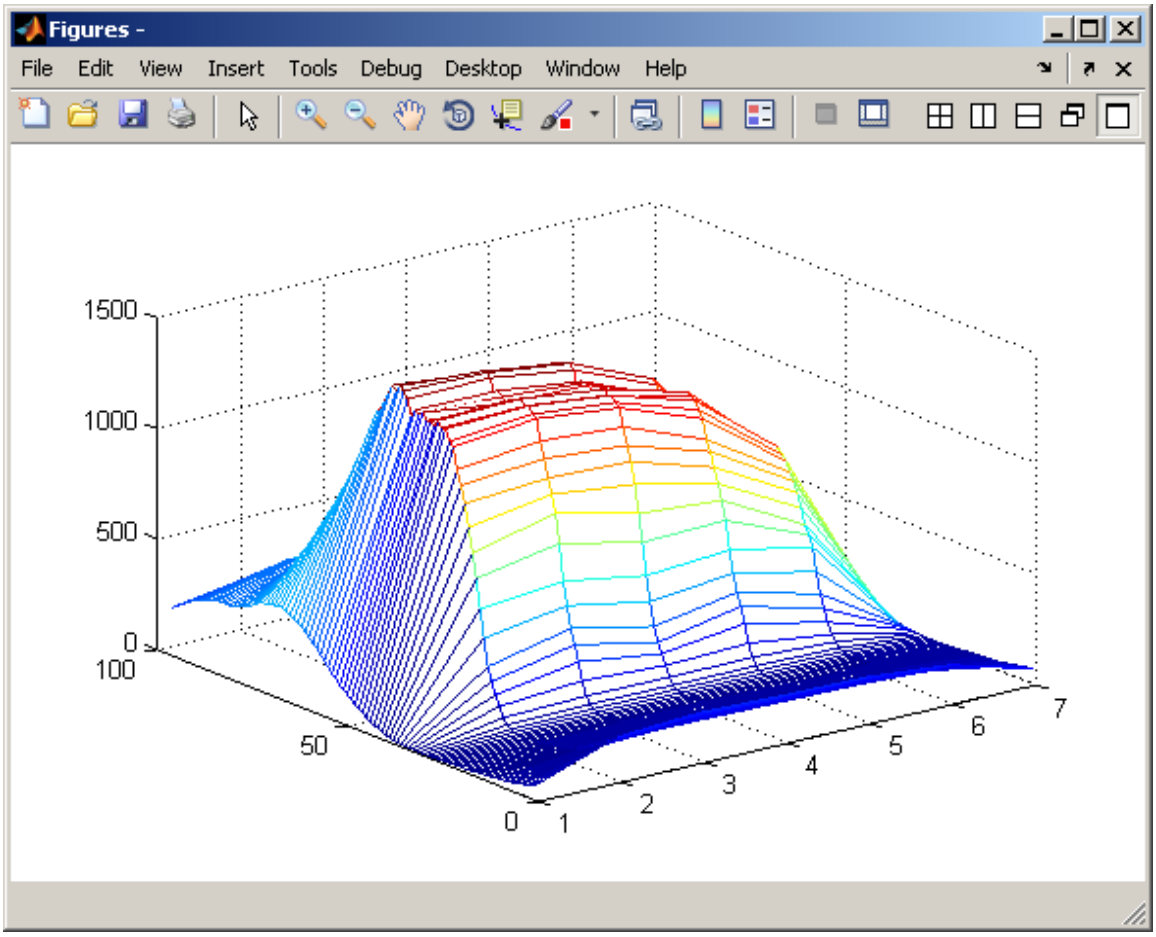
```
>> contour(data)
```



This gives a projection of lines of equal value of the dependent variable in the matrix elements projected onto the underlying grid of the two independent variables, much like contour lines seen on topographical maps. If it is desired to have a surface and its contour both displayed in the same figure then the function *surfc* can be used. Likewise, the commands

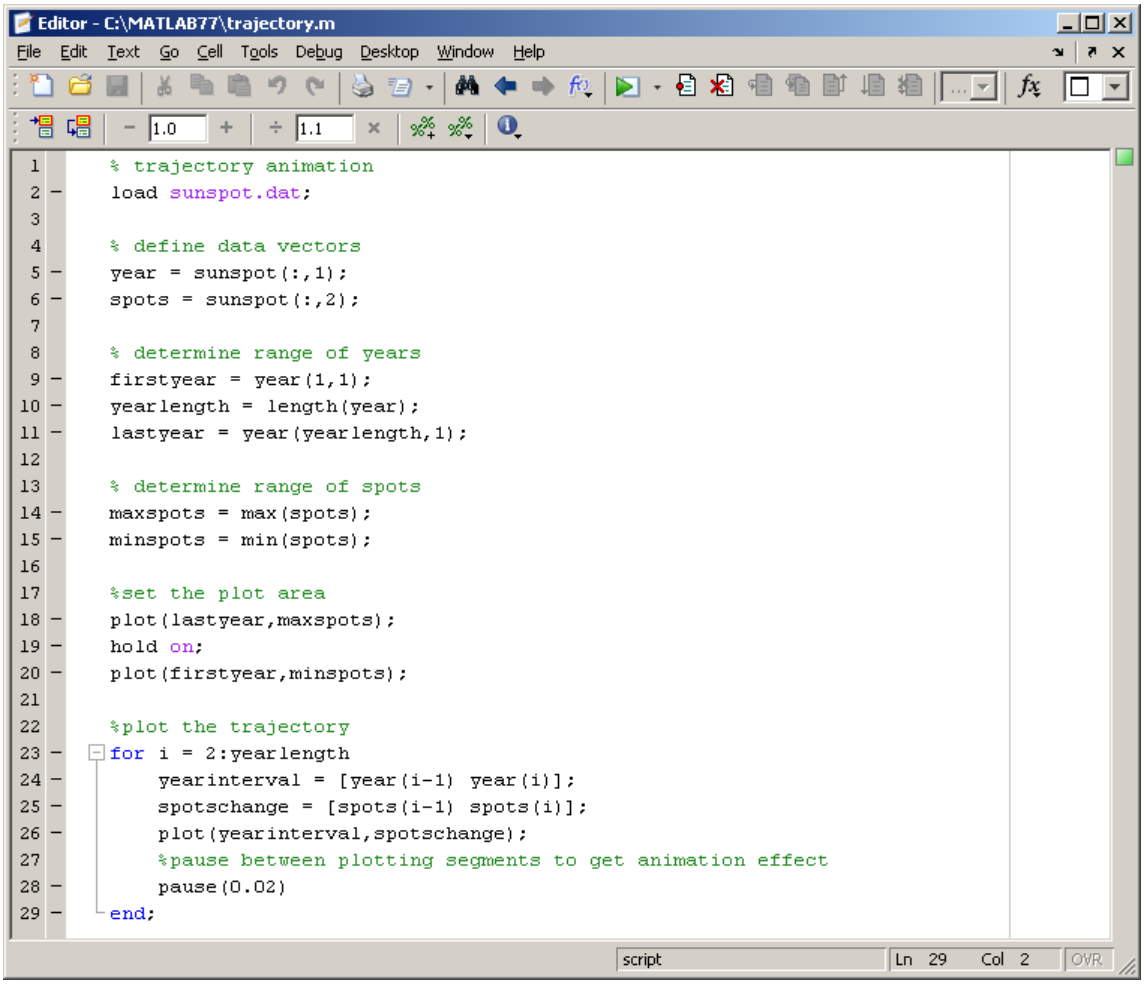
```
>> mesh(data)
>> meshc(data)
```

would create similar 3-D plots, but with the surface represented in a wire frame display.



Animation

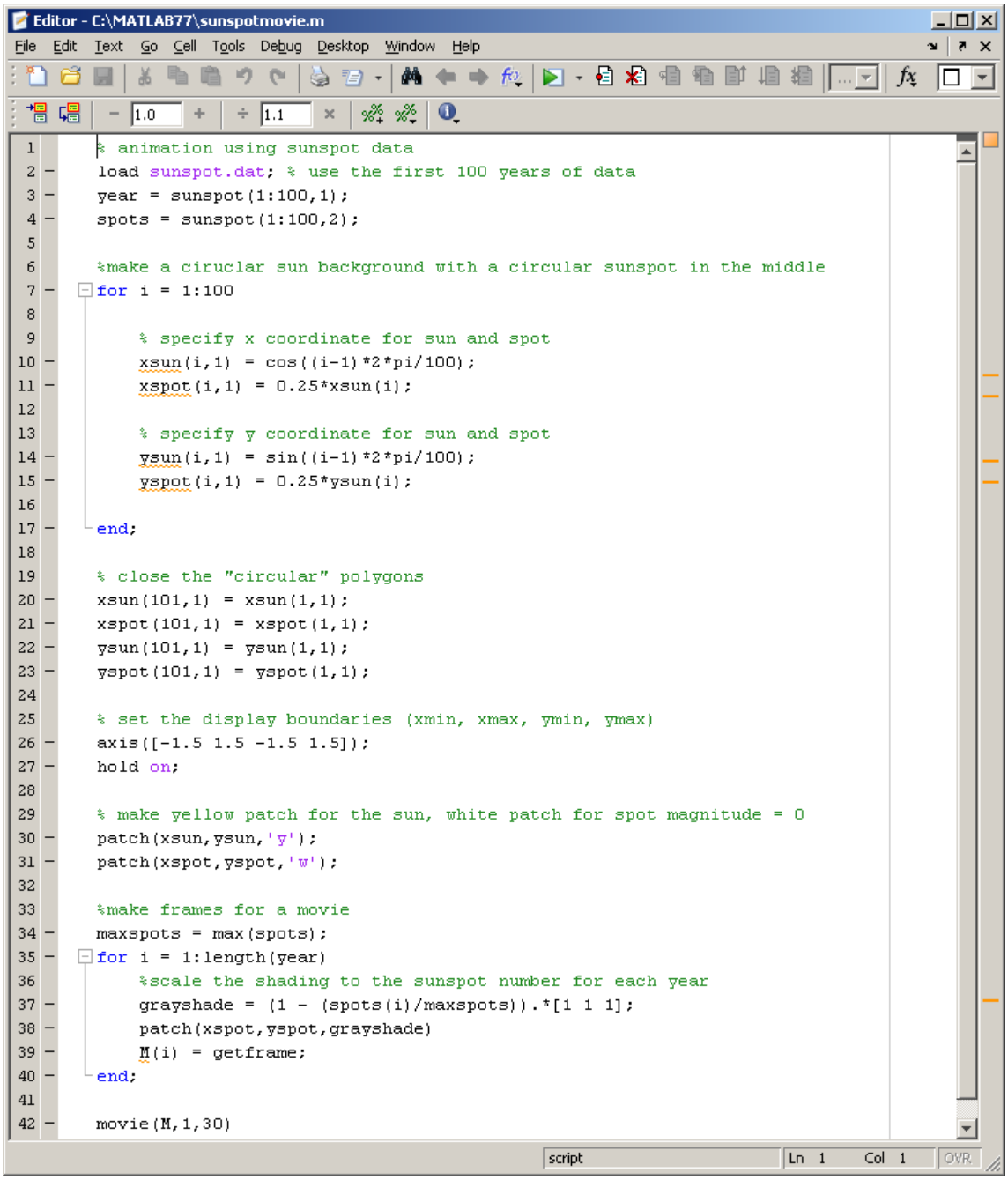
Data animation in Matlab is accomplished by a sequential display of graphic frames. At an elementary level the frames can be simple plots of points within a Figure Window. If the *hold* value is set to *off*, then each point will appear by itself, creating a display similar to a moving cursor on the screen. If the *hold* value is set to *on*, then each frame will incorporate the data from previous frames and the display will show a trajectory tracking type of pattern. In some cases this method may seem a bit crude because of blinking as each frame replaces the previous one, and delays in generating frames if needed computations are extensive or complex. As an example, suppose that we want to track the trajectory of the sunspot number over time. We could use the following script, placing it in the Matlab path with a the name *trajectory.m*



A trajectory animation would then be produced with the command

```
>> trajectory
```

An alternative procedure is the use of the *movie* feature. For this, a set of frames is assembled ahead of time into a movie type variable using the *getframe* function. Subsequently, the sequence of frames can be played in the Figure Window using the *movie* command. This command has three arguments: a required vector of movie frames, an optional number specifying the number of times the movie is to be played in a loop, and an optional number specifying the frames per second speed for the movie display. An illustrative m-file for creating a movie demonstrating sunspot number over time is the following:



If this script is in the Matlab path with the name *sunspotmovie.m*, the movie can be displayed with the command

```
>> sunspotmovie
```

You should see a static circular (actually a 100 side normal polygon) yellow "sun" with an interior gray spot that changes its intensity according to the magnitude of the dynamic sunspot

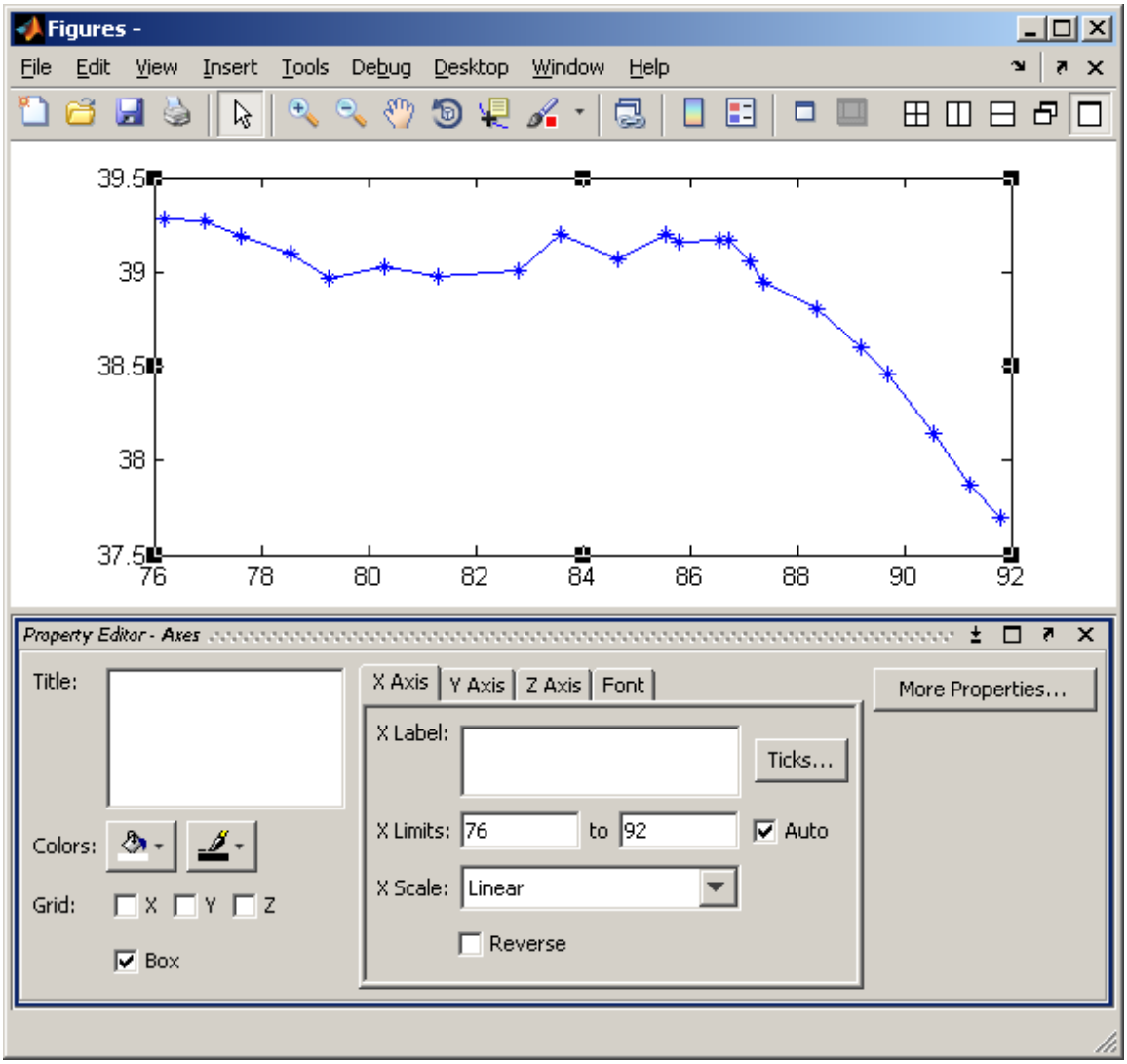
number when cycling through the years. In this instance the movie loop is played one time at 30 frames per second.

The Handle Graphics system

Handle graphics is Matlab's object oriented system for handling components used in constructing graphical displays. The objects are the basic drawing elements. Each object is identified with a handle and the handle contains information about the object's characteristics and properties. The GUI provides a convenient point and click method for changing graphic properties, but it does not have quite the same level of control as you get when making changes at the command line or within a script. For practice using the point and click method, create a basic plot, e.g.,

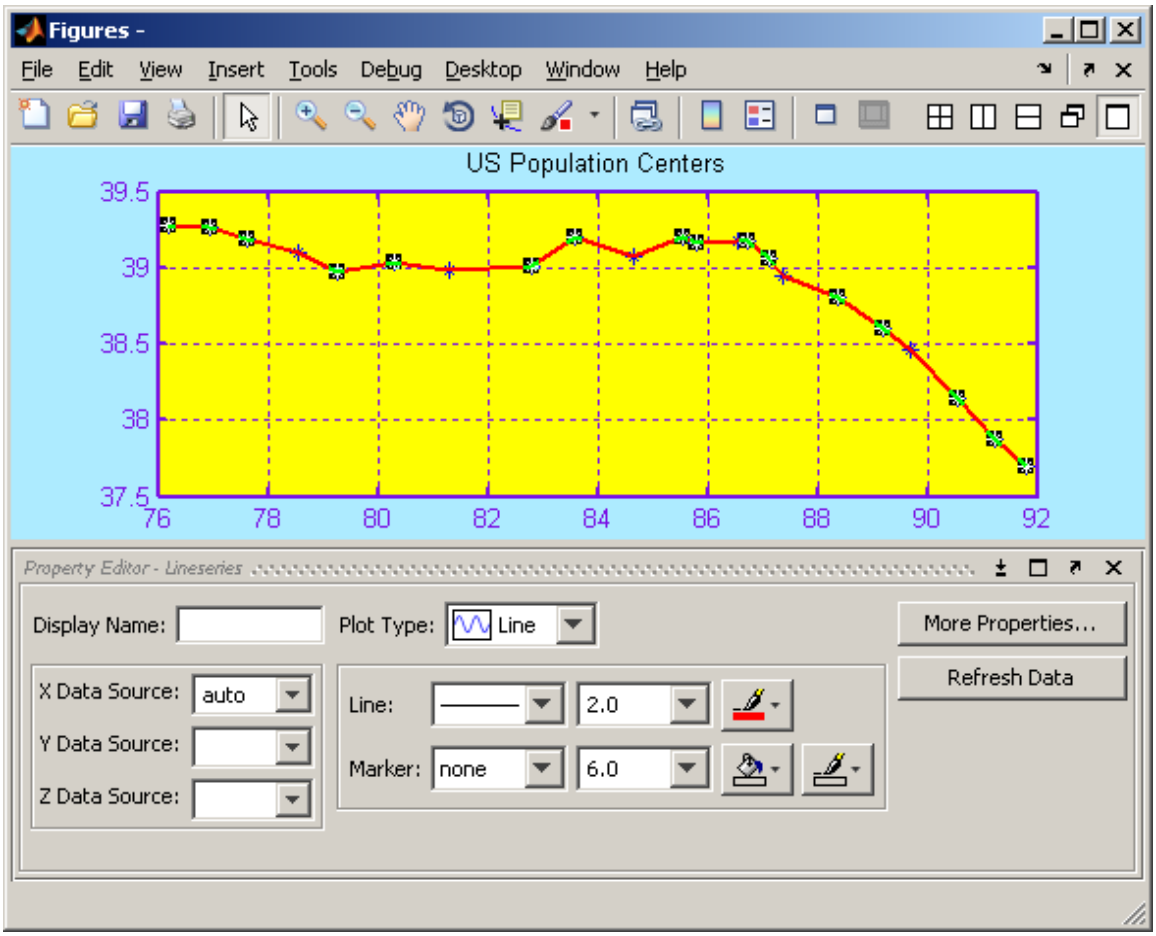
```
>> clear
>> load popcenter.dat;
>> north = popcenter(:,2);
>> west = popcenter(:,3);
>> hold off;
>> plot(west, north, 'b*');
>> hold on;
>> plot(west, north, 'b');
```

This will make a simple 2-D plot of the geographical coordinates for the U.S. population center at decennial years displayed as blue asterisks with a line connecting them. In this case the default values in the handles will produce a plot figure where there is no title, the axes scaled to the data range, inward tick marks with values on the left and bottom, and the data represented by a solid blue line. Then for point and click editing, click on the white backward arrow in the toolbar of the Figure Window and select the property that you want to edit from the *Edit* pull down menu on the top navigation bar. For example, select *Edit > Axes Properties* and a property editor window will appear at the bottom of the axis area. From this property editor the attributes of the figure axes can be modified as desired. A button labeled "More Properties…" can be used for further modifications involving lesser used attributes such as camera and lighting specifications. Of course those are not of much relevance for two dimensional displays.



You can toggle between tabs for setting or adjusting properties for the X, Y, and Z axes, and also click on boxes to enable grid lines on any of those axes. Let's check the boxes to show the Grid on the X axis and on the Y axis corresponding to latitude parallel and longitude meridian lines. There will be a *More Properties* button that can be clicked to bring up a *Property Inspector* window to let you select further properties. As an example, from that window scroll down to where the LineWidth option appears and edit the value space so that it reads 2.0 as a replacement for the default 0.5, resulting in thicker axis lines. There are also two buttons for color pallets on the *Property Editor* floating window, one for the axis lines themselves and another for the plot background. Select a color from the background pallet so that it changes from the default white, yellow for example. Also select a color from the line pallet, purple for example. Changes are applied immediately in the Figure Window: Now type in a title for the figure in the Title box, for example "US Population Centers", since this is the plot being made. The text of the title will appear above the figure as you type in the Title box. Now, edit a figure property from *Edit* >

Figure Properties. In the Figure Editor, change the background color for the figure, for example to a light blue. To practice changing the appearance of a specific object, click on the plotted line in the Figure Window to select it, then use *Edit > Current Object Properties*. A Property Editor window for the selected line will appear at the bottom of the Figure. After clicking on the *More Properties* button, change the line width from the default 0.5 to 2.0 again. Then let's change the line color from the default blue to red, which can be done after clicking on the line color pallet button in the *Property Editor* window. There are also separate color pallet buttons for marker faces and edges for instances where the data points are displayed as objects rather than a line. At this point the figure should look like the following:



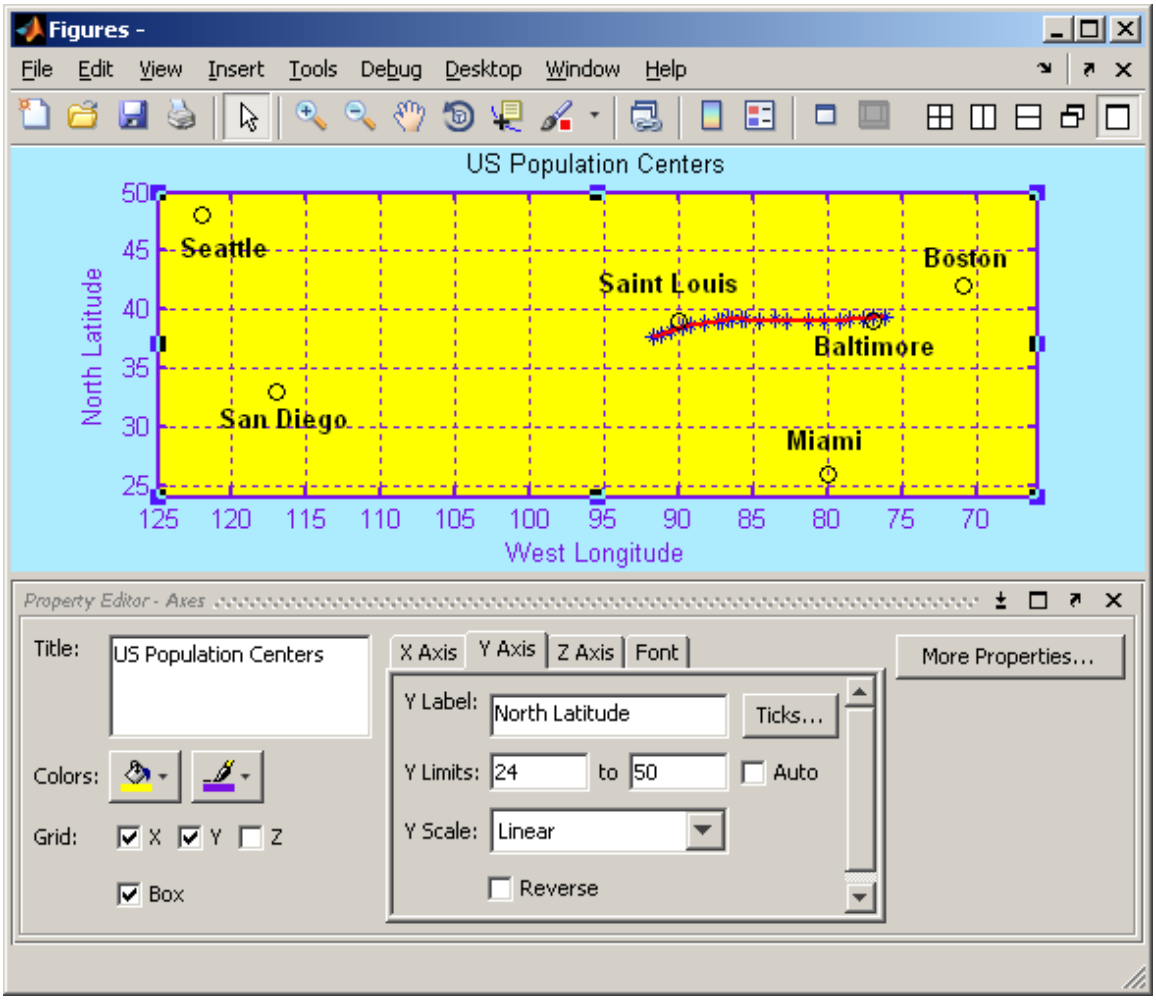
Now return to the Axes Property Editor by selecting it from the Edit Menu once again. Clicking on the *Ticks* button opens up another window in which you can edit tick mark positions. Other axis properties can be changed by clicking on the *More Properties* button, which also opens up another window where variable values can be selected or edited. For most purposes, this point and click interface will be sufficient to customize a plot display as you want it to appear. One thing we might do is select the *Reverse* box on the X Axis tab of the Axes Property

Editor window because maps usually have increasing west longitude going from right to left. Also, in the Axes Property Editor window we can also specify the label for the X axis as 'West Longitude' and the label for the Y axis as 'North Latitude'. We might also want to use fixed scaling for the axes instead of automatic scaling so that the map would cover all the 48 contiguous states. To accomplish that you can click on the *More Properties* button, then scroll to *XLim*. You may need to expand this to show the desired minimum and maximum values for the X axis. Change the automatic limits to a lower bound of 66 and an upper bound of 125. Similarly for *YLim* change the limits to a lower bound of 24 and an upper bound of 50.

A plot can be manipulated with threaded input back and forth from the point and click property editors and the command line. For example suppose we now want to insert some geographical reference markers onto the plot. We can do this by inserting black circle markers at large cities near extremes of the country and near the original and most recent population center data point

```
>> plot(71, 42, 'ko') % Boston;
>> plot(80, 26, 'ko') % Miami;
>> plot(117, 33, 'ko') % San Diego;
>> plot(122, 48, 'ko') % Seattle;
>> plot(77, 39, 'ko')  %Baltimore (near 1790 point);
>> plot(90, 39, 'ko') % Saint Louis (near 2000 point);
```

These commands display the landmark positions but not the identifying names. The names of the landmarks can be added by deselecting the backward white arrow, then selecting *Insert* > *TextBox* from the navigation bar. You can position the text string by clicking the mouse at the position where you want it to start. After typing in the desired text and hitting *Enter* you will automatically be put back into the white back arrow edit mode where you can adjust the position by dragging with the mouse. A visible box around the text is the default, but this can be removed by returning to the navigation bar and going to *Edit > Current Object Properties*. A Textbox Property Editor window will appear, in which you can select the *Line Style* as *no line*. While there you can also change the default *FontWeight* from *Normal* to *Bold* in order to provide greater contrast. You may need to select the text within the textbox before trying to change the font weight. If you follow this procedure for each of the landmarks, your plot should look similar to following:



In situations where point and click editing is not sufficient, or if you want to use some specific display template repeatedly and would like it incorporated into a script, Matlab has the capability of assigning specific values of object properties using handles. There are two specific built-in graphics handles that are used for default plotting. The handle *gcf* refers to the current figure and its information is used for generating a Figure Window when none exists at the time a command producing graphical output is issued. The handle *gca* refers to the current axes and likewise its information is used for constructing axes when none exist at a time a plotting command is issued. The handle *gco* refers to a graphical object that has been selected in the Figure Window. It will be empty if there is no Figure Window present or if an object has not been selected. If a figure object has been selected then information about that object's properties will be present in *gco*. The current values within these handles can be obtained with the commands

```
>> get(gcf)
>> get(gca)
```

```
>> get(gco)
```

To get a listing of values available for the handle variables in addition to the current value, use the *set* command with only the handle of interest as the argument

```
>> set(gcf)
>> set(gca)
>> set(gco)
```

Current values may be edited using the syntax *set(handle,'Property',value)*. Let's make a couple of changes from the command line using the handle syntax. Use *get(gca)* to examine the current settings for *XMinorTick* and *YMinorTick*. They should be listed as *'off'*. Next we will use *set(gca)* to see what other options in addition to *'off'* are available.
Options will be within square brackets separated by pipes, with the default value within curly braces. For the minor tick variables we see the options as

XMinorTick: [on | {off}]
YMinorTick: [on | {off}]

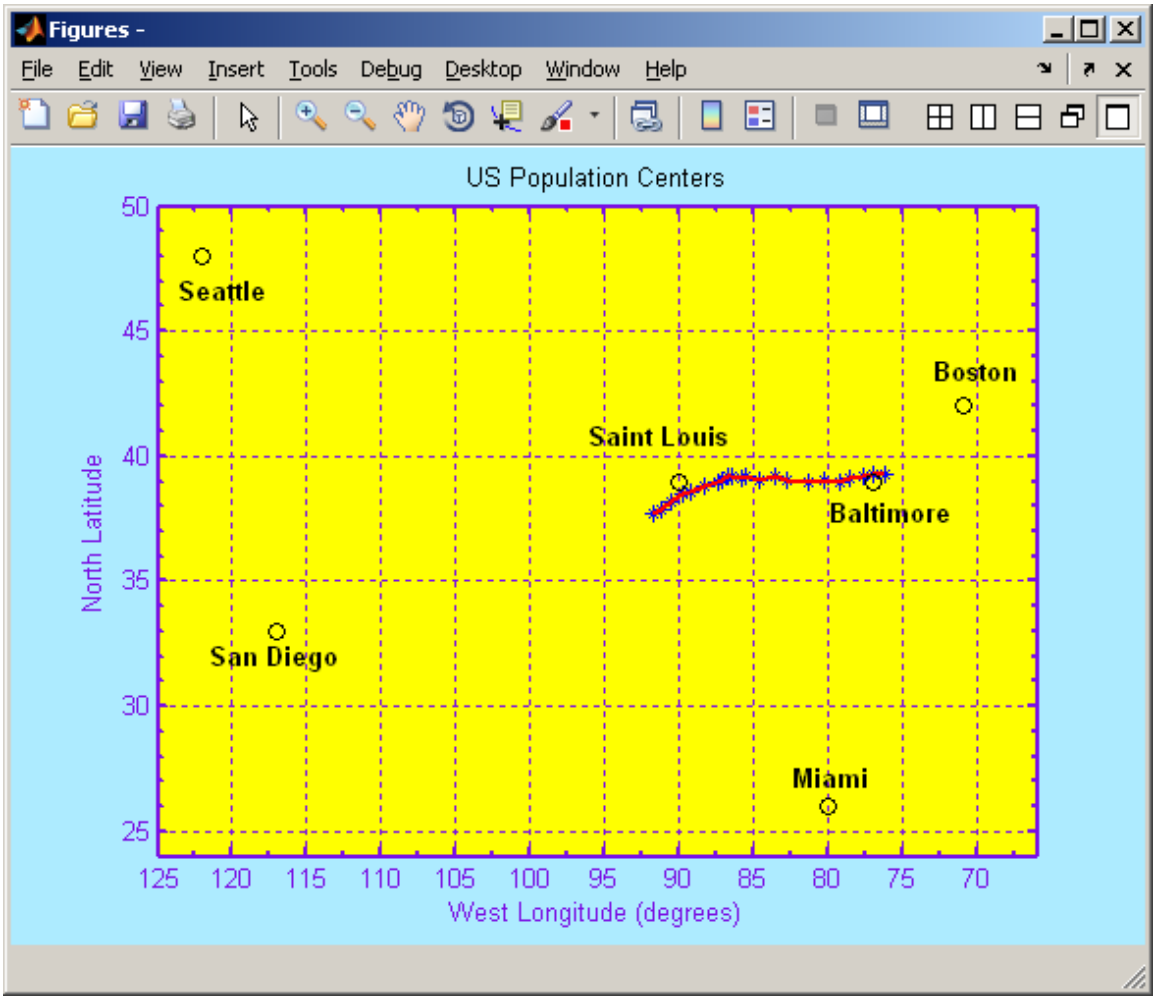
so let's use the alternative values of *'on'* by executing the commands

```
>> set(gca, 'XMinorTick', 'on');
>> set(gca, 'YMinorTick', 'on');
```

One word of caution should be noted. The handle *gca* lists *XLabel*, *YLabel*, *ZLabel*, and *Title* as variables but they are actually themselves handles to Text objects and need to be referenced directly. The procedure for setting or changing the value of any of these four from the command line or within a script is to reference them directly. For example, though obvious, suppose we decide that we need to specify that the longitude units are in degrees. We could adjust the labeling of the horizontal axis with the command

```
>> xlabel('West Longitude (degrees)');
```

This would create the new text string and overwrite what was already there. Note that labeling thusly is by direct command and command names are case sensitive. Therefore, the command *xlabel* executing the code of *xlabel.m* must be used rather than the target attribute name 'XLabel' itself. Once all the editing has been finished, the edit mode white back arrow can be deselected and the Property Editor Window closed. The final plot would look like



Saving and exporting graphics

Whatever static graphic you have created, in 2-D or 3-D, it will be displayed in a Figure Window with a default name *Figure1*. With *File > Save* or *File > Save As*, the image of the entire Figure Window can be saved to disk with the default location being the Current Directory, the default format being a Matlab Figure file (*.fig), and the default name being *untitled.fig*. However there will be a small window created with functionality for navigating to a different directory and for specifying a different name and format. The default format, Matlab Figure file, cannot always be directly pasted or imported into other documents so you may wish to save or export to a more commonly accepted format such as JPEG image (*.jpg). Some manipulation of the graphic image before saving to a file format is also possible using *File > Export Setup*. This will produce a window giving you several options for changing dimensions, fonts, lines, rendering, and so forth. After making any desired changes you can then click on the *Export* button and you will get the *Save As* window for specifying directory, file format, and file name.

Section 8: Data Analysis

Data analysis functions

Matlab was not developed as a statistical package, yet there are some elementary statistical functions built into the kernel and there is a specialized Statistics Toolbox available for purchase from MathWorks and which is currently included in the license for Matlab on the ITS servers. Installation of this Toolbox along with Matlab in labs run by individual colleges and departments may vary. The Toolbox is part of a bundled student edition package sold by the Campus Computer Store. Among the functions distributed as a component of the Matlab kernel itself are

mean	for arithmetic mean or average value of elements
median	for median value of elements
min	for smallest component
max	for largest component
var	for variance of the elements in a vector
std	for standard deviation from the mean of elements
sum	for sum of elements
prod	for product of elements
sort	for sorting elements within a vector
sortrows	for sorting rows within a matrix by values in a column
cov	for variance of a vector or covariance of a matrix
corrcoef	for correlation coefficient

Descriptive statistics

The elementary descriptive statistics functions that come as part of the general Matlab kernel can be used with vector or matrix arguments. By default Matlab assumes that columns are variables and rows are cases. If a matrix is given as the argument of one of these functions, the operation is applied to each column independently. Thus, if a statistic for rows of a matrix is desired, the function should use the matrix transpose as the argument. Let's use the *data* matrix variable above imported from the *wireless.xls* file to create some examples. Recall that this has 7 columns corresponding to the seven days of the week and 96 rows corresponding to the quarter hours of the day. To get the mean value of the quarter hourly wireless connection volume for the days of the week, rounded off and displayed in a column we would use the command

```
>> meanvol = round(mean(data)')

     meanvol =

     171
     576
     573
     539
     489
     313
     113
```

The mean quarter hour volume on Sundays was 171, on Mondays it was 576, etc.
Suppose we wanted the median quarter hourly volume on Wednesdays. Then we could specify the particular column, i.e., column 4, for that day

```
>> wedmedian = median(data(:,4))

wedmedian =

                364.5
```

Note that for the median function the value returned is the average of two mid range values when the vector length is an even number. Now let's get a statistic by rows instead of column. For example, if we wanted to know the variance of volume during the week for each quarter hour time of day, we could use the command

```
>> qhourvar = round(var(data'))'
```

This transposes the data matrix so that the quarter hour rows become columns, puts the variance of each of those columns into a row vector, transposes that row vector back to a column and rounds off the resulting values. The result is a column vector of the variances

```
qhourvar =

     4576
     3924
     3275
```

```
    {etc.}
```

indicating a variance of 4576 during the quarter hour between 00:00 and 00:15, 3924 during the quarter hour between 00:15AM and 00:30AM, etc. As a last example, suppose we want to know which quarter hour time had the greatest and least variance. For this we could use the *max* and *min* functions. With only the vector name as an argument, the max and min functions will return only the greatest and least value in the vector without specifying where it occurs. To get the vector index where it occurs, the assignment needs to refer to a two element vector.

```
>> [qhourvarmax maxindex] = max(qhourvar)

qhourvarmax =

     288659


maxindex =

     46
```

Thus, the maximum quarter hour volume variance of 288659 across the week occurs at the 46th quarter hour of the day (i.e., that beginning at 11:15AM). Similarly we would find that the minimum quarter hour variance of 60 across the week occurs at the 24th quarter hour of the day (i.e., that beginning at 05:45AM).

Sorting

The sorting functions are by numerical value including sign for numbers, by absolute value when complex numbers are involved, and alphabetically or by ASCII character code value when character strings are involved. When the argument is a matrix, the *sort* function operates on each vector along the sorting dimension independently. Syntax is

```
>> sort(x,n,'direction')
```

where **x** is the vector or matrix containing the data, **n** is the dimension for sorting (1 for rows, 2 for columns, 3 for a third index, etc.), and *direction* is either *ascend* or *descend*. The default value for **n** is 1, i.e., sort by rows if the data is in a matrix with more than one row, and the default value for *direction* is *ascend*, i.e., sort by ascending value. If a matrix has only a single row, i.e., a row vector, then the default sort is by column. These default values are used if only a

data matrix is given as an argument. Group sorting is also possible with the *sortrows* command. Syntax for this command is

```
>> sortrows(x,n)
```

where **x** is a data matrix of columns and rows and **n** is a scalar or vector of integers that specify the column order for subsorting with the direction being descending if an integer has a minus sign. For illustration, let's use the data matrix *popcenter* from the previous section on graphics. It was constructed with row sorting by ascending census year already done, but suppose we want the rows sorted by descending north latitude in column 2 instead. We could construct a revised data matrix with this feature with the command

```
>> northsort = sortrows(popcenter, -2)
```

Regression and curve fitting

Matlab has a function *polyfit* for fitting curves with polynomial regression and the function *polyval* for evaluating the polynomial fit. The *polyfit* function has syntax

```
>> orderfitcoeffs = polyfit(var1, var2, order)
```

where **var1** and **var2** are equal length vectors and **order** is an integer specifying the order of the polynomial to be used in the fitting. Note that the independent variable **var1** and the dependent variable **var2** must be of the same length and of the same type (row or column). Also note that the length of the vectors should be greater than the order of the fitting or there will not be a unique least squares solution. If the vector length is just one unit greater than the order then an exact fit is obtained but the parameter sensitivity may be very high. Thus it is prudent to use an order that is several units less than the vector length. The vector elements can be complex valued numbers, but in such cases the fitting coefficient vector will likely have complex valued elements as well: for example

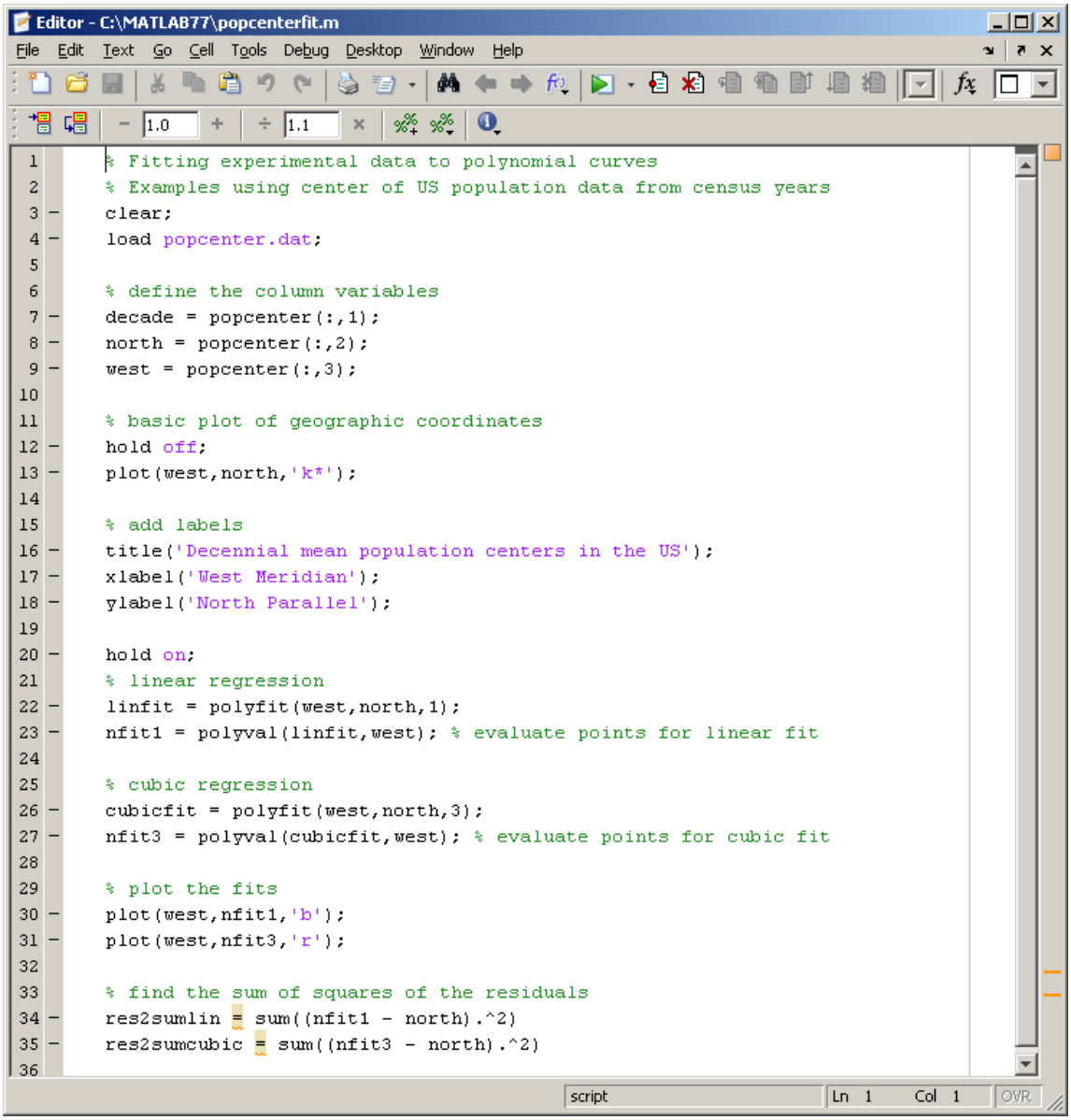
```
>> x  =  [1  2+3i  4+5i  6+7i];
>> y  =  [9  8-7i  6-5i  4-3i];
>> quadfitcoeffs = polyfit(x,y,2)

quadfitcoeffs =

   0.2266 - 0.0777i  -2.6798 - 0.7339i  11.5375 + 0.4743i
```

Thus, the least squares polynomial expression for y(x) would be

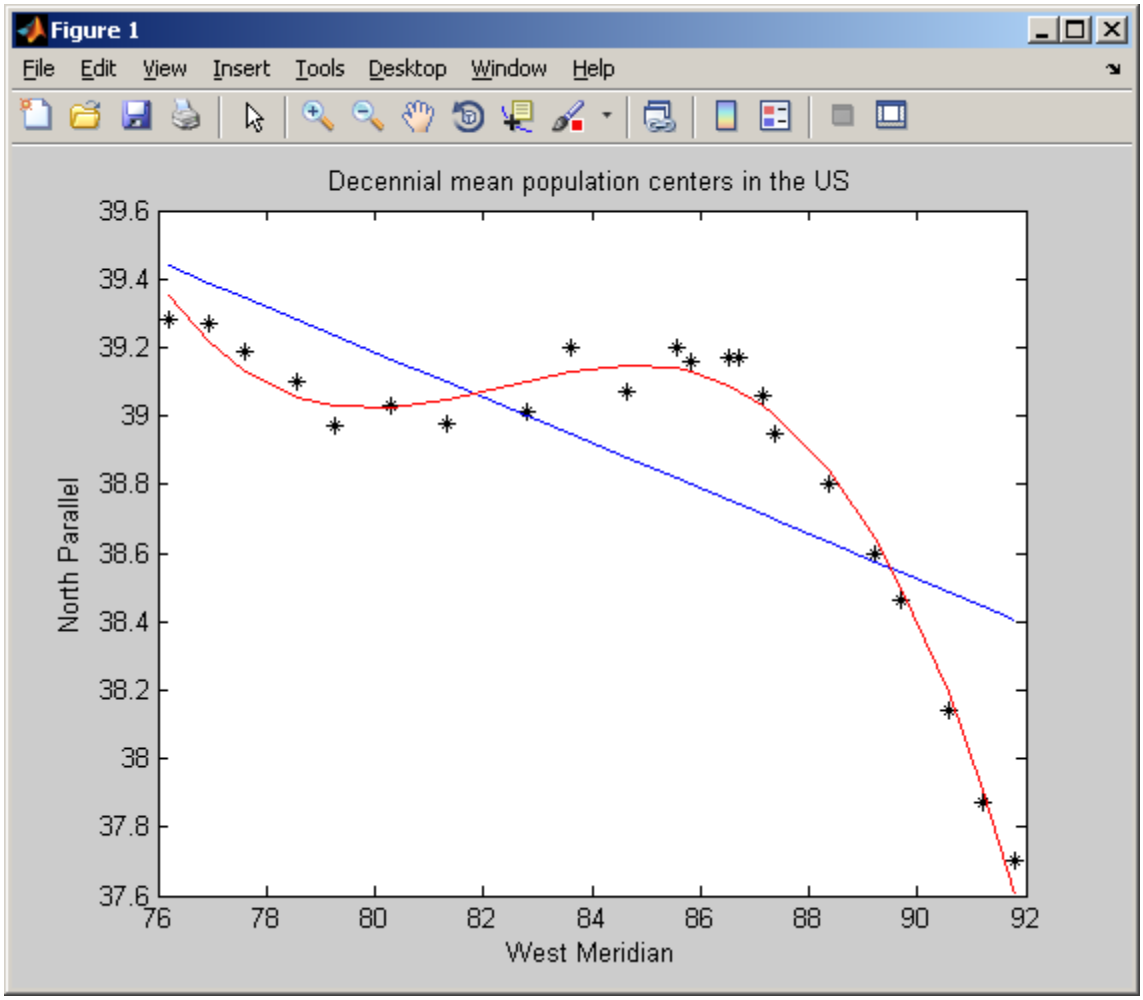
$$yfit(x) = (0.2266 - 0.0777i)*x^2 - (2.6798 + 0.7339i)*x + (11.5375 + 0.4743i)$$

For *x = 1* this gives *yfit(1) = 9.0843 - 0.3373i*, compared to the value *y(1) = 9* from the defining *x* and *y* vectors, so (as would be expected for an overdetermined system) the least squares fitting is not exact.

For a real world illustration, let's use data from the popcenter matrix obtained from loading the *popcenter.dat* file and try to fit the mean center of population migration pattern to a polynomial function, first to a first order polynomial (linear regression), then to a higher order polynomial, in this instance a cubic. The following script, *popcenterfit.m*, shows how this can be done and the results displayed in a graphic figure.



The two coefficients for linear regression are put into the *linfit* variable vector and the four coefficients for cubic regression are put into the *cubicfit* variable vector. Subsequently the regression curves are plotted on top of the experimental data using the *polyval* command. This command creates function values from a vector of polynomial power coefficients and a vector of function argument values. In the curve fitting done here you can see that linear regression does not give a very good fit compared to the fit with cubic regression. Remember though that this is just optimization of coefficients to minimize least square residuals using the particular data set, and higher order fits, though seemingly more accurate, may have very high sensitivity to perturbations



This can also be seen by running the *popcenterfit.m* script and looking at the sum of the squares of the residuals:

```
>> popcenterfit

res2sumlin =

          2.09513077385494


res2sumcubic =

     0.0845722500061468
```

This illustration uses observational data and shows the technique of curve fitting, but of course there is no theoretical foundation for expecting that the north latitude of the US population center would have a cubic relationship to the west longitude, or any explicit mathematical dependency at all for that matter.

Signal processing

Signal processing tasks in Matlab usually involve characterization or manipulation of time series vectors. This can involve evaluation of parameters of periodic behavior, finding trends, or removing distracting contributions such as noise. Matlab itself has functions for some types of elementary signal processing, and there is a specialized Matlab Signal Processing Toolbox (included in the license that ITS uses for its time sharing servers) which contains function m-files that are algorithms for implementing several more advanced signal processing tasks. These include specialized functions related to filtering, waveform generation and spectral analysis, along with a special GUI called SPTool.

The *sunspot.dat* file provides a good data set for illustrating some FFT techniques. Let's create a power spectrum of the yearly sunspot numbers and see the dominant frequencies. Recall that we named the sunspot number vector *spots* and the sampling rate to get the time vector that we named *year* was one point per year. The power vector will be an element by element multiplication of the FFT function with its complex conjugate

```
>> power = fft(spots).*conj(fft(spots));
```

and the frequency vector will be

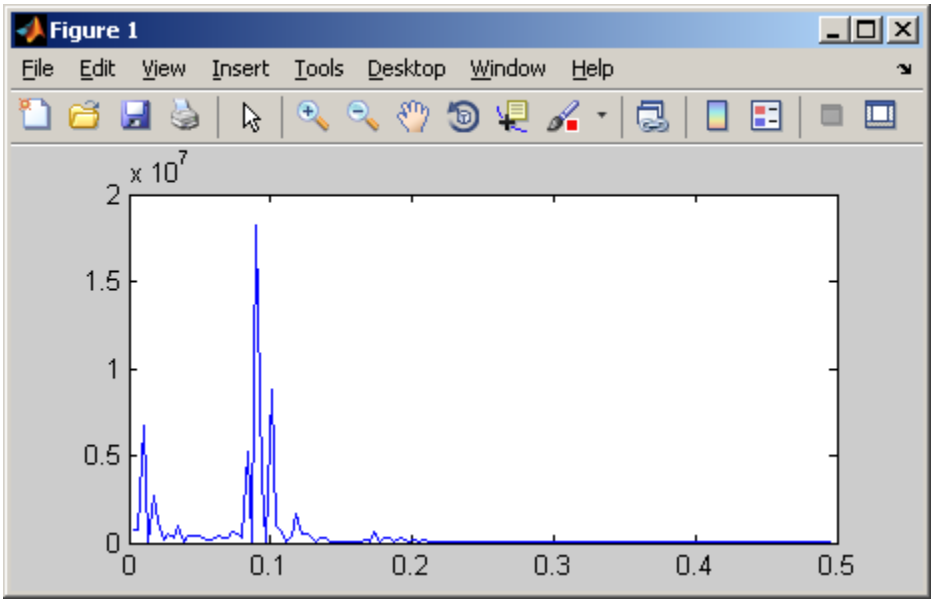
```
>> freq =  (1)*[0:length(year)]'./length(year);
```

The power vector has elements representing power at each corresponding element in the frequency vector. Sometimes the power variable of interest may be the power vector normalized by its length or a total power obtained by summing the element values in the vector. However for our illustrative purposes here we will stick to the simple power vector.

The FFT algorithm takes advantage of complex conjugate symmetries of roots of unity, but this results in half of the information in the power spectrum (that above the Nyquist frequency) being redundant. Also, as a consequence of its computational algorithm, the first element of the FFT vector is always the sum of all elements in the original time domain vector. If all the time domain elements are positive numbers this can lead to an initial vector element of a very large magnitude compared to other element magnitudes, causing scaling problems for graphical

display. Therefore in plotting a power spectrum it is typical to restrict the points displayed to the range from 2 to half the length, e.g., since our vector includes data for 288 years,

```
>> plot(freq(2:144), power(2:144));
```



The period associated with strong peak at about 0.09 is the reciprocal of its frequency value, i.e. (1 year/0.09), or about 11.1 years -- which is the classical sun spot cycle length.

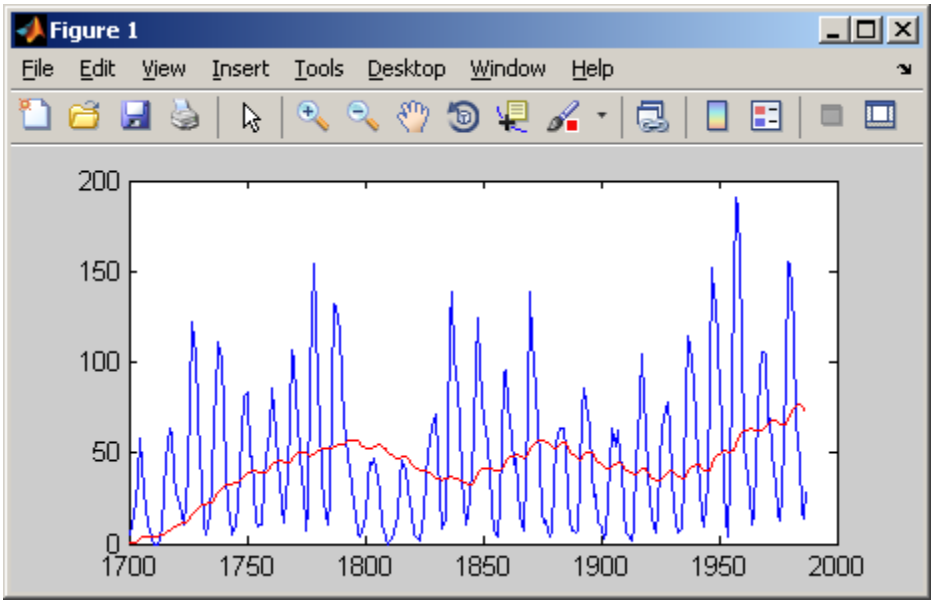
The most elementary filtering function, *filter*, is included in Matlab and has the syntax

```
>> output = filter(numcoeff, denomcoeff, input)
```

where an *output* vector is formed from filtering an *input* vector with a transfer function whose numerator coefficients are contained in the vector *numcoeff* and whose denominator coefficients are contained in the vector *denomcoeff*. A common use of this method is smoothing a curve with a moving average, such as is routinely done for monitoring long term trends in fluctuating stock market prices. Let's use the sun spot data again and get a trend line by using a moving average window of 50 years, smoothing out the data over about four and a half cycles of the major periodic component. For this the numerator vector is a row with 50 elements, each with value (1/50). The denominator vector is simply a single element row vector with the value 1

```
>> numcoeff   = (1/50)*ones(1,50);
>> denomcoeff   = [1];
>> trend = filter(numcoeff, denomcoeff, spots);
```

```
>> hold off;
>> plot(year, spots);
>> hold on;
>> plot(year, trend, 'r');
```



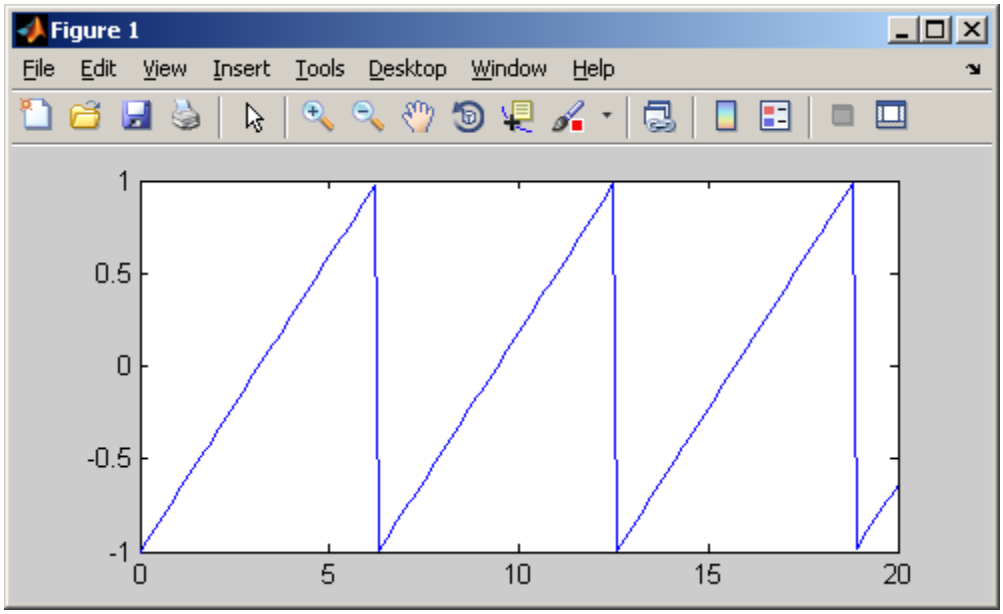
The red line in the plot thus shows a macroscopic trend in the value of the sunspot number beyond the large scale year to year periodic fluctuations.

Signal processing methods more elaborate than FFT and filtering can be found in the Matlab Signal Processing Toolbox. For example, this toolbox has several wave generation functions. Some of the most commonly used are

sawtooth for a triangle wave generator
pulstran for a pulse train generator
square for a square wave generator

As an illustration, let's generate a triangular, sawtooth shaped wave:

```
>> hold off;
>> t = [0:0.1:20];
>> x = sawtooth(t);
>> plot(t,x)
```

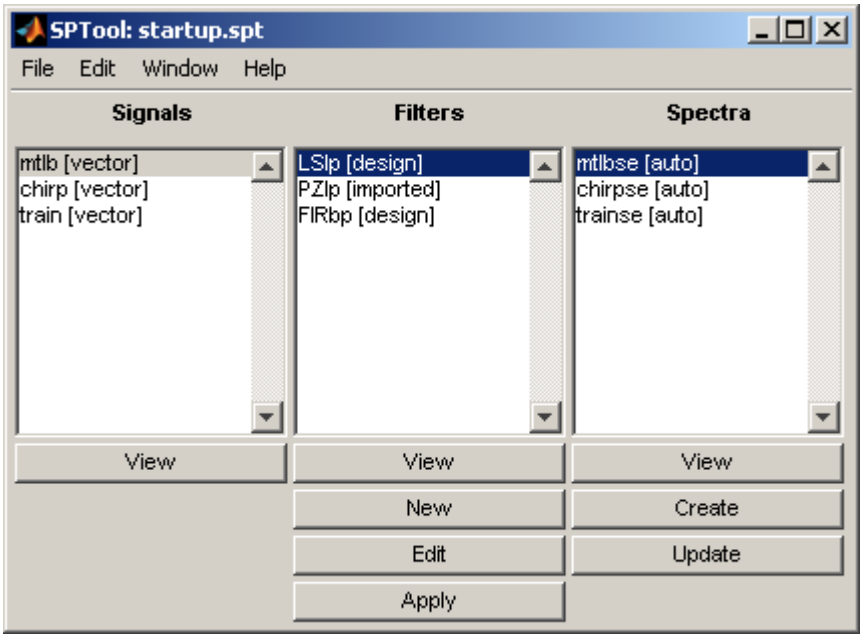


The sawtooth function is defined as -1 at multiples of 2π and linear with a slope of $1/\pi$ at all other points.

The Signal Processing Tool GUI is a feature available in the toolbox. It is launched with the command

```
>> sptool
```

which creates a window with selections of signals, filters, and spectra



The view buttons on this window can be used to launch a Signal Browser, a Filter Viewer, or a Spectrum Viewer.

Image processing

If a task merely involves manipulating an image, you would probably not look to Matlab as the application of choice. More likely you would use Adobe Photoshop or some similar product that was developed specifically for working with images. Nevertheless there may be occasions where image manipulation is needed in the context of other operations, and there is a Matlab toolbox available when such instances arise.

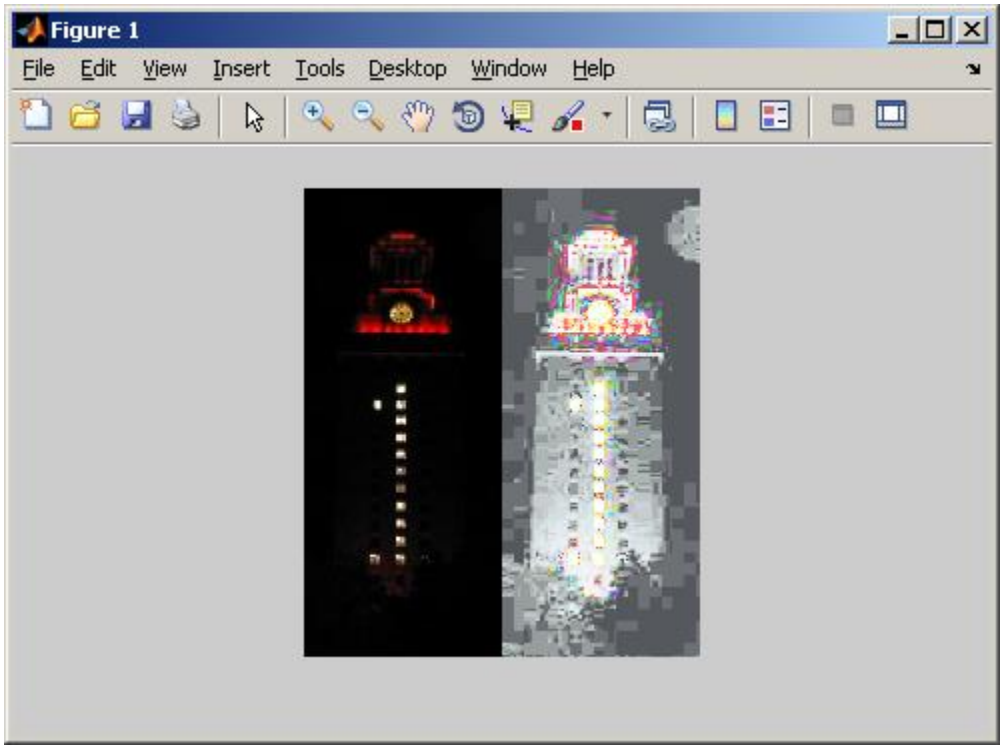
This Matlab auxiliary, the Image Processing Toolbox (included with the Matlab license used for installations on the ITS time sharing servers) has m-file functions for manipulating arrays containing image information. Also, many of the Signal Processing Toolbox functions can be used in conjunction with image processing. Within the Image Processing Toolbox are routines for geometric operations, image analysis and enhancement, and region of interest operations. But before working on an image, it has to be imported into the workspace. This can be done with the *load* command if the image data has already been stored in a MAT file. Images with supported formats (e.g., jpg, gif, tiff, png) can be imported with the *imread* command and displayed in the Figure Window with the *imshow* command. Once an image has been processed as desired, it can then be exported in a supported format using the *imwrite* command. As an example, let's import a low contrast image, *tower.jpg*, enhance its contrast, and export a "before" and "after" comparison image. For this exercise, you will need to put the image file in your Matlab path. The process can be done with any image with a supported format, but in this case we choose a low contrast nighttime picture of the University tower.

```
>> I = imread('tower.jpg'); % create image data matrix
>> imshow(I)                % display in Figure Window
```

Now let's add contrast with the *histeq* command. This spreads out intensities over the entire possible range, and is particularly useful if actual intensities of all the pixels are clustered over a narrow range. The jpg format has three layers for a color image and the *histeq* command requires them to be processed separately:

```
>> for m = 1:3 % add contrast for each layer
     J(:,:,m) = histeq(I(:,:,m));
   end;
>> K =  [ I J ];     % place old and new side by side
>> imshow(K)         % display the contrast
>> imwrite(K,'towercontrast.jpg') % write new file
```

Now there should be a new display in the Figure Window showing both the original and contrast enhanced picture side by side



and there should also be a new image file, *towercontrast.jpg*, in the current directory. In this exercise the data matrix is 3-dimensional, as can be seen in the *Value* column of the Workspace

Window. This is an RGB (red, blue, green) format, in which the primary color component is the third index. The first two indices specify the vertical and horizontal position of the pixel, top to bottom, left to right. Element values represent intensities. Thus, for example, displaying an element in the original image data matrix

```
>> disp(I(50,50,2))
    13
```

tells us that the green intensity of the pixel in the 50th row and 50th column has an intensity value of 13.

This is not the only format for storing image data information. Image objects in Matlab can also be two dimensional data arrays. The indices correspond to pixels in a rectangular image and the elements can be either magnitudes of intensity for gray scale or integer pointers to color or intensity information in a separate color map matrix. Color map matrices are three columns representing components of the primary colors red, blue, and green. Each row is a particular weighting of each of these in a range from 0 to 1. Indexed matrices, whose elements point to the color map, will have entries corresponding to particular lines in the map. The image whose pointers are in matrix *X* will have its pixel content determined by the RGB values in the associated color map matrix *Y* using the command

```
>> image(X), colormap(Y)
```

As a small example, let's create a game board for chess or checkers. First we will create the *8x8* layout of the board. Pixels whose row and column sum is odd need a different color from those whose row and column sum is even. But the color within each set needs to be uniform, so only two lines are needed in the color map. First create the matrix with pointers to alternating colors

```
>> Z = [eye(2) eye(2) eye(2) eye(2)];
>> Z4 = [Z;Z;Z;Z];
```

The matrix Z4 has 8 rows and 8 columns with elements having alternating values of 0 and 1. Since there can be no row 0 in a color map we need to convert these two pointer indices so that they have values of 1 and 2 instead

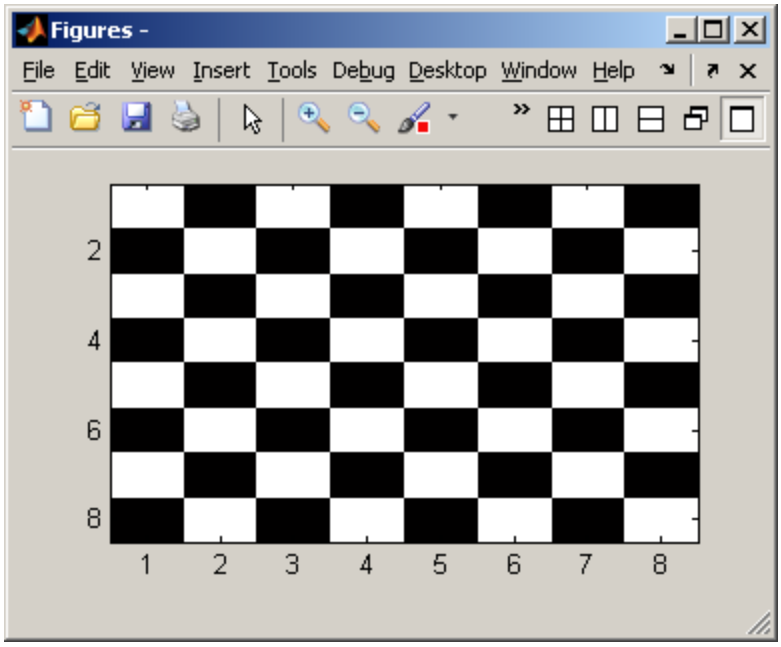
```
>> C = 2*ones(8) - Z4;
```

Now *C* is the matrix of color map pointers for a checkerboard image. Next we create a color map with three columns for RGB values with two rows, one for each of the two colors whose pointer values are 1 or 2. Let's define the color map as the variable *colors*.

```
>> colors(1,:) =  [ 1 1 1 ]; % pointer value 1 white
>> colors(2,:) =  [ 0 0 0 ]; % pointer value 2 black
```

The *image* command will create a color pixel image, with the color map specified with the *colormap* command.

```
>> image(C)
>> colormap(colors)
```



For a more intricate image with a more extensive color map, you can look at one of the example files distributed with Matlab. One such binary file is *mandrill.mat*, a facial image of a gorilla, whose data includes the matrix *X* with pointer data and the color map matrix *map*. Thus, the commands needed for viewing are

```
>> load mandrill
>> image(X)
>> colormap(map)
```

Another type of image format is one in which the elements of the image matrix are intensities rather than pointers to a mapping. The command to generate an image from this format is *imagesc*. We can see our previous checkerboard image from matrix *C* again with the command

```
>> imagesc(C)
```

Images can be manipulated by changing their data matrices or color maps. For example we can change colors for the black and white checkerboard by changing values in the color map matrix, for example to red and cyan with the command

```
>> colors(:,1) = sort(colors(:,1))
```

or we can remove the blue component of the color map used for the gorilla image with

```
>> map(:,3) = 0;
```

Demonstrations of many other ways for manipulating and analyzing images are available in the documentation that comes with Matlab. Select *Demos* from the *Help* menu of the main Matlab window and then expand *Toolboxes* and subsequently *Image Processing* to see these.