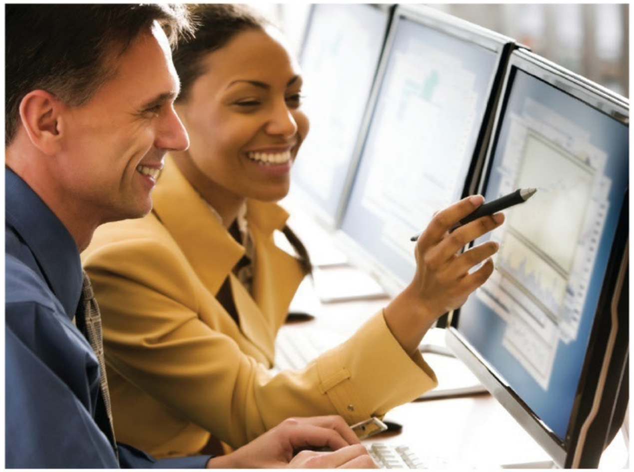
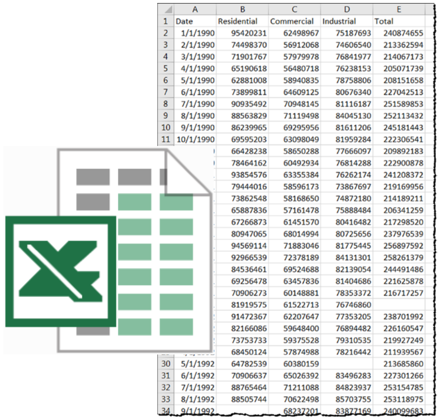
Analyzing Data

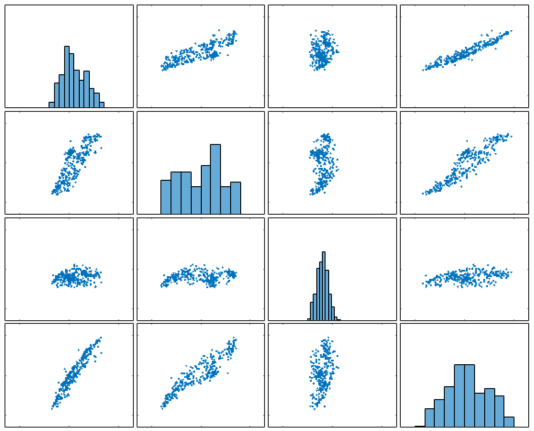
MATLAB® Fundamentals for Aerospace Applications

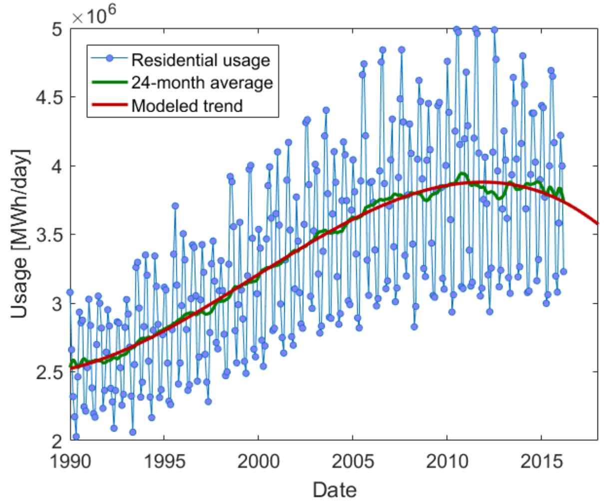


Outline

- Importing data from file
- Normalizing data
- Dealing with missing data
- Polynomial fitting
- Creating customized visualizations







Chapter Learning Outcomes

The attendee will be able to:

- Import data from commonly used file formats programmatically.
- Remove or replace incorrect or missing values in a data set.
- Combine multiple MATLAB® function calls to perform specific data-analysis tasks, such as fitting a polynomial to data.
- Plot a function by generating data points for a given range.
- Convert between numeric and character data types.
- Create textual displays and plot annotations using dynamically calculated values.
- Customize plot elements such as line width and marker size.

Course Example: Modeling Electricity Consumption

The spreadsheet `electricity.xlsx` contains a monthly record of total electricity usage in the United States, from January 1990 to March 2016, divided by sector (residential, commercial, industrial, and total, which includes other miscellaneous sectors). The values represent the total consumption for the given month, in units of megawatt hours. The data includes missing values for some months, to simulate typical real-world data.

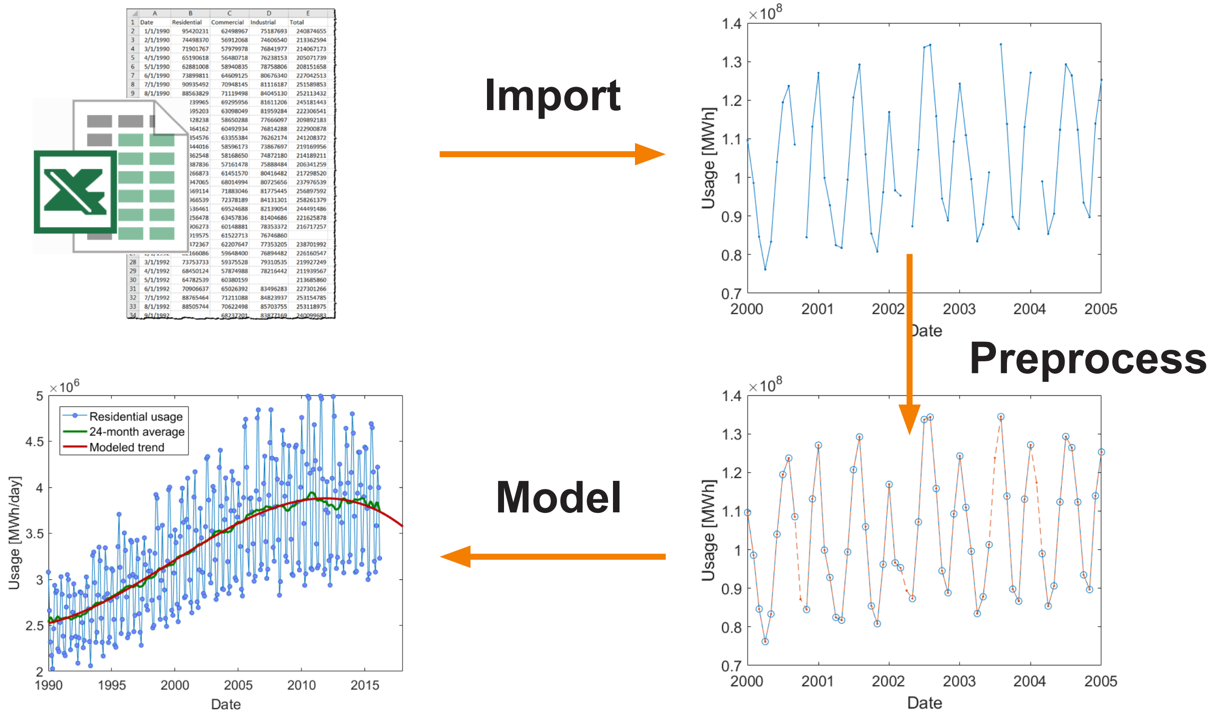
The goal of the example in this chapter is to follow a typical workflow for data analysis and modeling:

- Import the data.
- Preprocess the data.
- Investigate and model the data.

Try

Open `electricity.xlsx` in Excel®.

MATLAB provides numerous functions for a wide range of typical data analysis tasks. Because MATLAB is intended to be applicable to any numerical computation application, these data analysis functions are designed to be broadly applicable. You will typically need to apply several of these functions to perform a specific analysis.



Importing Data Programmatically

The information in a data file must be encoded in a particular way. A given file type (e.g., image data) may be encoded in many different *file formats* (e.g., JPEG, bitmap, PNG, TIFF, etc.). Within a given format, however, the structure of the data is precisely defined. MATLAB, therefore, provides functions that can read and interpret such fixed-structure data files. These functions read data into MATLAB in the form of variables with predictable size and type.

The MATLAB documentation contains a table of all supported file formats: **MATLAB → Data Import and Export → Standard File Formats → Supported File Formats for Import and Export**.

To read mixed tabular data as a table from a spreadsheet or a delimited text file, use `readtable`:

```
data = readtable('electricity.xlsx')
```

The `readtable` function automatically detects data types like `datetime`.

To use the `readtable` function, the data must conform to a tabular structure. However, the data contained in text files and spreadsheets is *not* constrained to any particular structure.

You can import data directly from an Excel spreadsheet with the `xlsread` function. The command

```
data = xlsread('electricity.xlsx')
```

returns the numeric data in `electricity.xlsx` as a matrix `data`. Calling `xlsread` with two outputs returns both the numeric data and the text data:

```
[data,txt] = xlsread('electricity.xlsx')
```

Try

Import the data from `electricity.xlsx` using `readtable`.

```
data = readtable('electricity.xlsx');
```

Extract the dates. Extract the usage data as a matrix.

```
sectors = data.Properties.VariableNames(2:end);
usage = data{:,sectors};
dates = data.Date;
```

The text data, including the dates, is returned as a cell array `txt`. This cell array contains all the text information in the spreadsheet, including headers or footnotes. You will need to extract and convert the dates manually:

```
dates = datetime(txt(2:end,1))
```

You can use optional inputs to `xlsread` to read from a specific range within a specific worksheet:

```
usageRes90 = xlsread('electricity.xlsx',...
              'B2:B13')
```

Note the use of Excel-style indexing ("B2:B13") as a character vector.

You can read numeric data from delimited text files (including comma-separated value (CSV) files) with `dlmread`.

Normalizing Data

Data often includes variables of different units and scales. Consequently, comparisons between observations are meaningless.

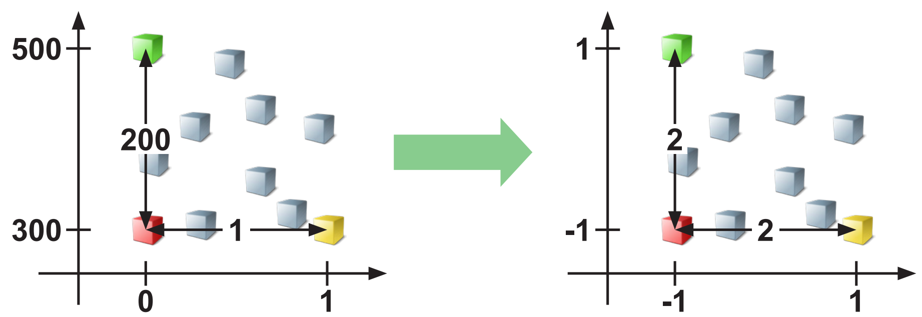
To allow for valid comparisons between different observations, it helps to normalize the variables. This is typically done by scaling and/or shifting each variable.

The electricity data is recorded as monthly totals. Because different months have different numbers of days, it will help to convert these values to daily averages before trying to build a model. This involves dividing each row of the matrix of usage data by a different number of days.

```
usage = usage./daysPerMth
```

There are numerous other ways to shift and scale variables. Two common options are:

- Map the range [min(*x*) max(*x*)] to a fixed interval such as [0 1] or [-1 1].
- Shift the center (mean or median) of the data to 0 and scale with a measure of spread (such as standard deviation).



Try

Find the number of days in each month of the `dates` vector.

```
nextmonth = dates(end) + calmonths(1);
datesplus1 = [dates;nextmonth];
daysPerMth = days(diff(datesplus1));
```

Normalize monthly total usage to average daily usage.

```
usage = usage./daysPerMth;
```

Recall that statistical functions operate on the columns of a matrix independently. This makes it simple to calculate a given measure for each variable with a single command:

```
Xmin = min(X)
Xrng = max(X) - Xmin
```

You can then use the vectors `Xmin` and `Xrng` to normalize the matrix `X`:

```
Xshift = X - Xmin
Xnorm = Xshift./Xrng
```

Each column of `Xnorm` is now normalized to the range [0 1].

You can use the `zscore` function to perform the common normalization to zero mean and unit standard deviation:

```
Xnorm = zscore(X)
```

The function `zscore` is found in the Statistics and Machine Learning Toolbox™.

Dealing with Missing Data

There are typically three ways to deal with missing data values (`NaN`s): ignore them, delete them, or replace them.

Ignoring missing values means that the original data is not changed. However, it does require you to take the missing values into account in all calculations.

Deleting values makes the data clean and easy to work with. However, to keep the data aligned, you often have to throw out valid data. You may also end up with less data than you need.

Replacing values also makes the data clean and easy to work with. However, values for the missing data must be estimated, so your resulting data is not necessarily accurate.

Try

Plot the normalized data. Note the gaps in the plot due to missing values.

```
plot(dates,usage)
```

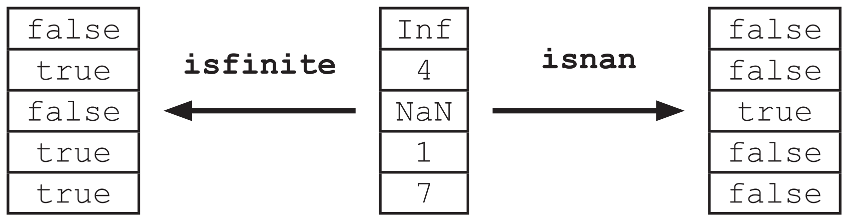
Try to calculate average electricity usage from data with missing values.

```
mean(usage)
```

Locating Missing Values

MATLAB provides numerous “`is*`” functions that take an array as input and return a logical output that signifies if the input has a certain characteristic.

In particular, the `isnan` function takes a numeric input and returns a logical array that is true where the input array is `NaN`. Similarly, `isfinite` tests for where the (numeric) input has a finite value, i.e., neither `NaN` nor `Inf`. The `isundefined` function tests categorical arrays for undefined values.



When applied to a matrix, the output of these functions will also be a matrix. In this case you will typically need to test whether any or all observations are missing for each variable

```
any(isnan(x))
all(isnan(x))
```

or whether any/all variables are missing for each observation:

```
any(isnan(x),2)  % specifying the dimension
all(isnan(x),2)
```

The result is a logical row or column vector.

Try

How many missing values are there in the electricity usage data?

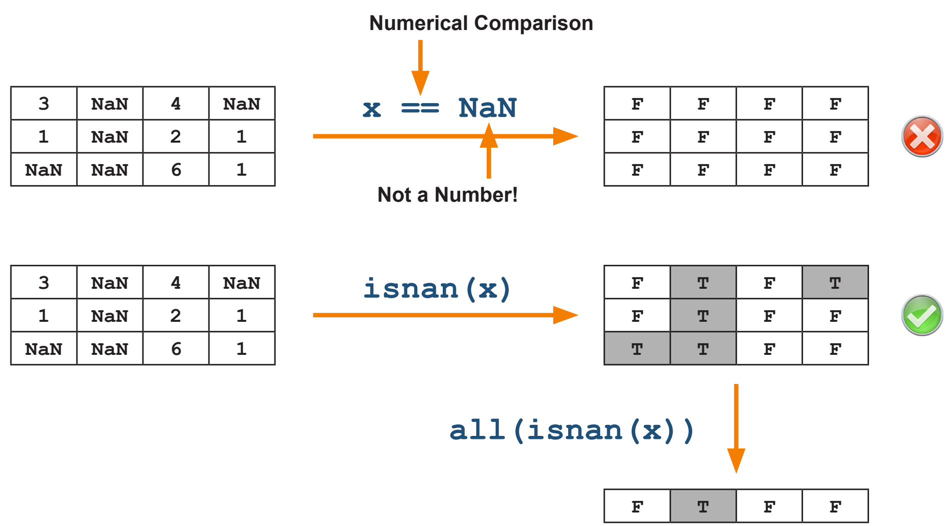
```
nnz(isnan(usage))
sum(isnan(usage))
```

How many months have missing data from at least one sector?

```
anymissing = any(ismissing(data),2);
nnz(anymissing)
```

The `ismissing` function can be applied to an entire table. It returns a value of `true` wherever a value is “missing”, meaning:

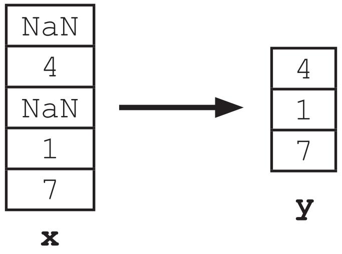
- `NaN` for numeric variables
- `NaT` for dates
- Empty for text variables
- `undefined` for categorical variables



Removing Missing Values

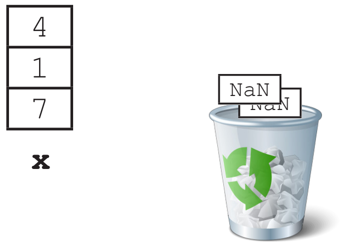
You can use the `rmmissing` function to remove elements from an array. Two common approaches are to extract the desired elements to another array

```
y = rmmissing(x)
```

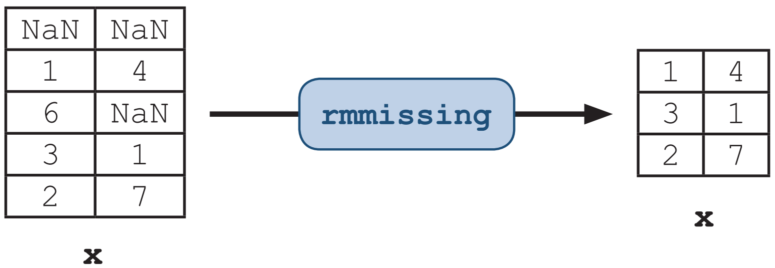


or to remove the undesired elements from the array:

```
x = rmmissing(x)
```



When dealing with matrices or tables, the `rmmissing` function removes entire rows or columns if there is at least one missing value in that row or column.



Try

Remove the missing values from the residential usage data. Note the size of the result.

```
resUsage = usage(:,1);
resUsageOK = rmmissing(resUsage);
```

Try to plot the data. Why doesn't this work?

```
plot(dates,resUsageOK)
```

Use logical indexing to remove dates corresponding to missing values.

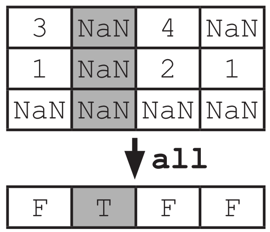
```
idxOK = ~ismissing(resUsage);
datesOK = dates(idxOK);
plot(datesOK,resUsageOK)
```

Remove rows of a table if any usage data is missing. Note the size of the result.

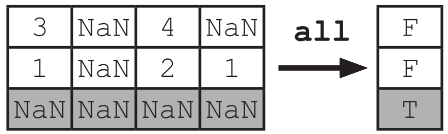
```
dataOK = rmmissing(data);
plot(dataOK.Date,dataOK.Residential)
```

The `all` function is also useful for determining indices of rows or columns to remove. To remove elements, assign an empty array to undesired elements (which deletes them):

```
idx = all(isnan(x))
x(:,idx) = []
```




```
idx = all(isnan(x),2)
x(idx,:) = []
```



Replacing Missing Values

One way to replace missing values is *piecewise interpolation*, where values are predicted based on only the neighboring data points.

The `fillmissing` function incorporates several interpolation methods:

```
yfilled = fillmissing(y,method)
```

The interpolation method is specified as a character vector. In general, the interpolation method you choose depends upon what assumptions you can make about your data. The options are given in the table below.

method	**Interpolation method used**
`'next'`	The missing value is the same as the next nonmissing value in the data.
`'previous'`	The missing value is the same as the previous nonmissing value in the data.
`'nearest'`	The missing value is the same as the nearest (next or previous) nonmissing value in the data.
`'linear'`	The missing value is the linear interpolation of the previous and next nonmissing values. (numeric, `datetime`, and `duration` data types only)
`'spline'`	Cubic spline interpolation matches the derivatives of the individual interpolants at the data points. This results in an interpolant that is smooth across the whole data set. However, this can also introduce spurious oscillations in the interpolant between data points. (numeric, `datetime`, and `duration` data types only)
`'pchip'`	The cubic Hermite interpolating polynomial method forces the interpolant to maintain the same monotonicity as the data. This prevents oscillation between data points. (numeric, `datetime`, and `duration` data types only)

Try

Plot residential usage.

```
plot(dates,resUsage)
```

Replace missing electricity usage data.

```
eqfill = fillmissing(resUsage,'linear');
hold on
plot(dates,eqfill,'x')
```

Compare results with using the dates as the sample points.

```
linfill = fillmissing(resUsage,'linear',...
    'SamplePoints',dates);
plot(dates,linfill,'o')
hold off
```

Interpolate missing electricity data using a cubic spline and the dates as the sample points.

```
usage = fillmissing(usage,'spline',...
    'SamplePoints',dates);
```

By default, the `fillmissing` function assumes the observations are equally spaced when performing the interpolation. You can specify the spacing by providing a vector that represents the observation sampling locations

```
yfilled = fillmissing(y,method,'SamplePoints',x)
```

where `x` is a vector of locations where the data was sampled. The vector `x` can be numeric, `duration` or `datetime` data type.

Linear Correlation

The strength of the linear relationship between two variables can be measured numerically by the *correlation coefficient*. The correlation coefficient has a value between +1 and –1. A correlation of ±1 indicates a perfect linear relationship between the variables; +1 means that an increase in one variable is associated with an increase in the other, whereas –1 means that that an increase in one variable is associated with a decrease in the other. A correlation of 0 indicates that the variables are not linearly related.

The MATLAB function `corrcoef` computes the linear correlation from a data sample. As with all statistical functions, the input is in the form of a matrix, with each variable in a separate column:

```
corrcoef([France,Germany,Mexico])
```

Results are displayed in a square matrix, with the entry in the i^{th} row and j^{th} column giving the correlation of the i^{th} and j^{th} variables in the data. The matrices are necessarily symmetric about the main diagonal. The main diagonal elements are always +1, because a vector of data is always perfectly correlated with itself.

Similarly, you can visualize correlations between each pair of columns of a matrix of data using `plotmatrix`:

```
plotmatrix([France,Germany,Mexico])
```

The result is a matrix of scatter plots. The plot in the i^{th} row and j^{th} column is the scatter plot of variable *i* against variable *j*. Because variables are perfectly correlated with themselves, the diagonal plots are histograms, which show the distribution of the variables.

Try

Determine correlations between residential, commercial, industrial, and total electricity usage.

```
plotmatrix(usage)
corrcoef(usage)
```

Currency exchange data

```
load cx
DEM = Data(:,7);
CHF = Data(:,9);
CAD = Data(:,4);

scatter(DEM,CHF)
corrcoef([DEM CHF])
scatter(DEM,CAD)
corrcoef([DEM CAD])
plotmatrix([DEM,CHF,CAD])
```

Moving Window Operations

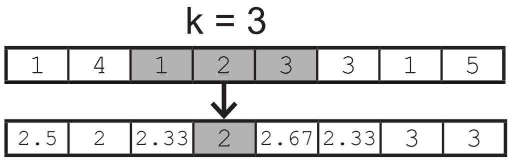
The electricity usage data shows both a long-term trend and a short-term seasonality. In such a situation it is common to calculate summary statistics, such as the mean, on a moving subset of the data.

Moving average smoothing can be accomplished in MATLAB by applying the `movmean` function.

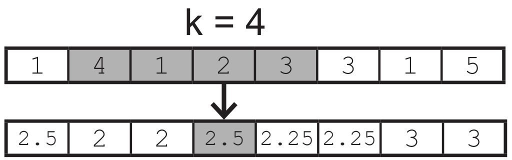
```
s = movmean(x,k)
```

The second input `k` to `movmean` is the number of elements in the averaging window. When `k` is odd, the window is centered over the element in the current position. When `k` is even, the window is centered over the current and previous elements.

```
movmean(x,3)
```

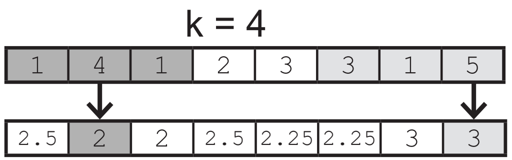



```
movmean(x,4)
```



If there are not enough elements to fill the window, such as near the endpoints, `movmean` shrinks the window and averages only over elements that remain.

```
movmean(x,4)
```



Try

Smooth electricity usage data to remove small-scale noise.

```
n = 12;
usageMovAvg = movmean(usage,n);
plot(dates,usageMovAvg)
```

Change `n` to `24` and try again.

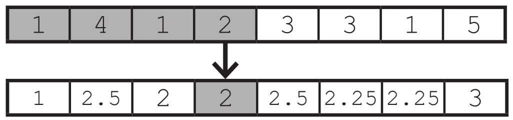
```
n = 24;
usageMovAvg = movmean(usage,n);
plot(dates,usageMovAvg)
legend(sectors,'Location','northwest')
```

One-sided averaging is more appropriate for time-series data where future data is not yet known. You can pass `movmean` a two element vector specifying how many elements backward and forward to include in the window.

```
s = movmean(x,[kbk kfwd])
```

The window includes the element in the current position, plus `kbk` elements backward and `kfwd` elements forward. For example, you can compute a 4-point trailing average by using the command

```
s = movmean(x,[3 0])
```



The `movmedian` function behaves the same way as `movmean`, and calculates the median of the elements in the window.

Fitting a Polynomial

The MATLAB `polyfit` function computes the coefficients of a least-squares polynomial fit of arbitrary degree. For example,

```
x = 0:5;               % x data
y = [2 1 4 4 3 2];     % y data
p = polyfit(x,y,3)     % Degree 3 fit
```

```
p =
  -0.1296    0.6865   -0.1759    1.6746
```

Polynomial coefficients are listed from highest to lowest degree, so

$p(x) \approx -0.13\, x^3 + 0.69\, x^2 - 0.18\, x + 1.67.$

The vectors `x` and `y` must be numeric, so dates and durations must be converted to numeric values.

When the coefficients of a polynomial are expressed in a vector `p`, you can use the MATLAB `polyval` function to evaluate the polynomial at arbitrary inputs. For example,

```
xplot = -1:0.01:6;
yplot = polyval(p,xplot);
plot(xplot,yplot)
```

plots the fit between $x = -1$ and $x = 6$.

If some of the values of `x` are large, numerical precision limitations can lead to inaccurate results. To avoid this, have `polyfit` rescale `x` by asking for a third output:

```
[c,~,scl] = polyfit(x,y,3);
```

Try

How quickly is residential electricity usage increasing (based on a linear trend), in MWh/year?

```
resUsage = usage(:,1);
elapsedYears = years(dates - dates(1));
c = polyfit(elapsedYears,resUsage,1);
c(1)
```

How much electricity will be used by 2020?

```
polyval(c,30)
```

Try again with a cubic trend.

```
c = polyfit(elapsedYears,resUsage,3);
polyval(c,30)
```

Predict when there will be no electricity usage.

```
z = roots(c)
dates(1) + years(z(1))
```

How trustworthy is such a prediction?

Similarly, if you pass the vector of scaling coefficients `scl` to `polyval`, the evaluation is performed using the scaled `x` values:

```
yfit = polyval(c,x,[],scl);
```

The second optional output from `polyfit` can be passed to `polyval` to compute confidence intervals for the fit, assuming the errors in the `y` data are independent and normally distributed with zero mean.

You can use the `roots` function to find the roots of a polynomial. Similarly, the function `poly` finds a polynomial with specified roots; this is effectively the inverse operation of `roots`.

Adding a Theoretical Curve

Consider adding a model of the long-term trend to the plot of the monthly residential electricity consumption data, such as a polynomial fit from `polyfit`. To plot such a model, you need to calculate values from a created independent variable. The values of the independent variable do not necessarily have to coincide with measured data.

1. Create independent variable.
2. Calculate dependent variable from the independent variable.
3. Make plot.

The model and the data can now be plotted together:

```
plot(xdata,ydata)
hold on
plot(xmodel,ymodel)
hold off
```

Note that `xdata` and `ydata` must be the same size, and `xmodel` and `ymodel` must also be the same size. However, `xdata` and `xmodel` can be different sizes.

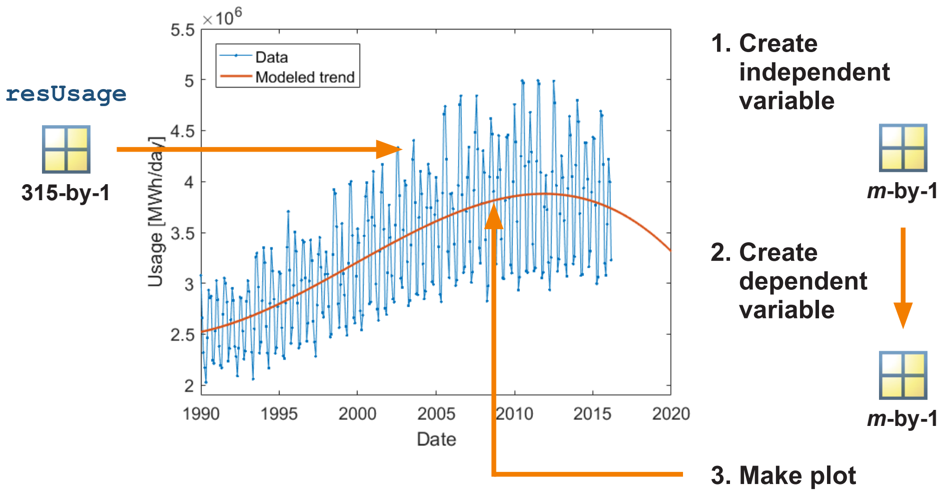
Try

View a cubic polynomial fit to the electricity data, using the same date values as the recorded data.

```
c = polyfit(elapsedYears,resUsage,3);
resUsageFit = polyval(c,elapsedYears);
plot(dates,resUsage,'.-')
hold on
plot(dates,resUsageFit)
hold off
```

Make the same plot but with more finely spaced dates and projecting into the future.

```
elYrFit = linspace(0,30,501);
resUsageFit = polyval(c,elYrFit);
plot(dates,resUsage,'.-')
hold on
datesFit = dates(1) + years(elYrFit);
plot(datesFit,resUsageFit)
hold off
```



Adding Annotations

Text annotation can be added directly to your plots with the `text` function.

`text(`*`x`*`,`*`y`*`,`*`string`*`)`

adds the text in *`string`* with the bottom-left corner of the first line located at (*`x`*,*`y`*) in the current axes.

It is useful to be able to create annotation text automatically from calculated values. You can use standard concatenation (`[]`) to join character vectors together:

```
ch1 = 'Hello';
ch2 = 'world';
ch = [ch1,' ',ch2,'!']
```

```
ch =
    Hello world!
```

However, if you attempt to concatenate numeric values with character vectors, MATLAB will perform an automatic conversion to type `char`, so that all elements of the resulting array have the same type:

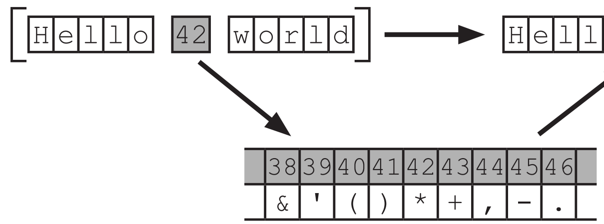
```
['Hello',42,'world']
```

```
ans =
    Hello*world
```

Try

Add annotations to an electricity usage plot: `annotateLXplot.mlx`.

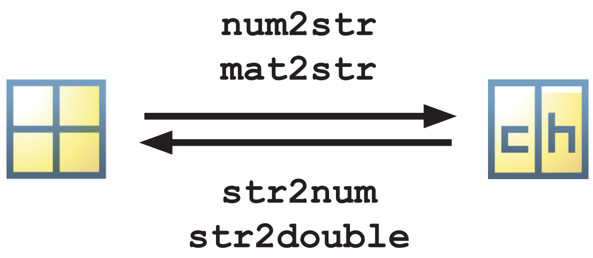
Note the value 42 is interpreted as the 42nd ASCII character "*", rather than a string representation of the number 42.



MATLAB therefore provides specific functions to convert between strings and numbers:

```
['Hello',num2str(42),'world']
```

```
ans =
    Hello42world
```



Specifying Color

Color can be specified in MATLAB using one of the eight predefined text codes (`'b'`, `'g'`, `'k'`, etc.), or the equivalent full name (`'blue'`, `'green'`, `'black'`, etc.), or by giving a precise red-green-blue (RGB) specification. RGB triplets are given as a 3-element vector of values between 0 and 1.

RGB triplet	Code	Full Name	RGB triplet	Code	Full Name
`[1 0 0]`	`r`	`red`	`[1 1 0]`	`y`	`yellow`
`[0 1 0]`	`g`	`green`	`[1 0 1]`	`m`	`magenta`
`[0 0 1]`	`b`	`blue`	`[0 1 1]`	`c`	`cyan`
`[0 0 0]`	`k`	`black`	`[1 1 1]`	`w`	`white`

Plots cycle through a default set of seven colors. The RGB specifications of these colors are:

1	`[0.00 0.45 0.74]`	5	`[0.47 0.67 0.19]`
2	`[0.85 0.33 0.10]`	6	`[0.30 0.75 0.93]`
3	`[0.93 0.69 0.13]`	7	`[0.64 0.08 0.18]`
4	`[0.49 0.18 0.56]`		

Note that you cannot use an RGB color specification in place of the text codes in a plot command. To specify line color with an RGB vector, you need to provide an optional pair of inputs to the `plot` function (see next page).

Customizing Plots

A large number of properties of plot elements can be specified directly in a `plot` command using *name, value* pairs of optional inputs:

```
plot(x,y,Property1,Value1,...
    Property2,Value2,Property3,Value3,...)
```

The property names are specified as a string; the accompanying value can be either a string or a numeric value, depending on the details of the property. The properties can be given in any order; the only restrictions are that a property must be immediately followed by its value, and that the optional pairs appear at the end of the argument list:

```
plot(x,y,'o','MarkerSize',3,...
    'MarkerFaceColor','k','LineWidth',2)
```

Note that the `'MarkerSize'` and `'LineWidth'` properties take numeric values; `'MarkerFaceColor'` specifies a color, which can be given as a text code or as an RGB vector.

For line plots, the most common properties (other than line style, line color, and marker style) are:

- `'LineWidth'`
- `'MarkerEdgeColor'`
- `'MarkerFaceColor'`
- `'MarkerSize'`

Try

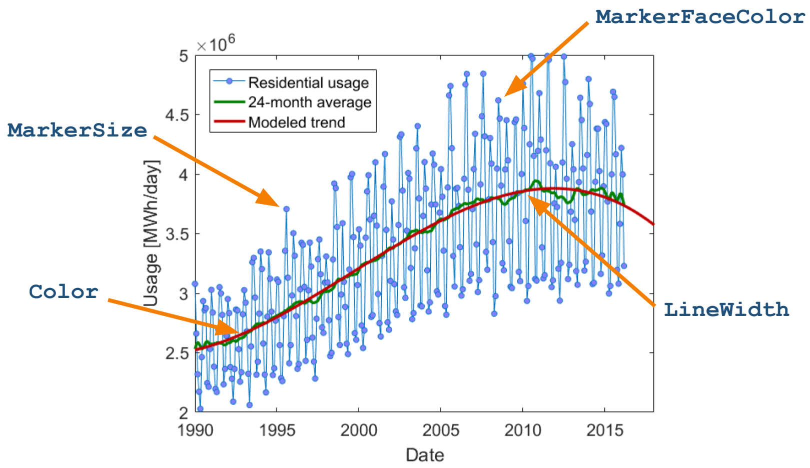
Customize residential data with customized markers.

```
plot(dates,resUsage,'o-','MarkerSize',4,...
    'MarkerFaceColor',[0.5 0.5 1])
```

Add the 24-month moving average with a green line, and the model with a thick red line.

```
hold on
plot(dates,usageMovAvg(:,1),...
    'Color',[0 .5 0],'LineWidth',2)
plot(datesFit,resUsageFit,...
    'Color',[0.75 0 0],'LineWidth',2)
hold off
```

You can access detailed documentation on all plot objects: **MATLAB → Graphics → Graphics Objects → Graphics Object Properties**. Each type of graphical object has a link to a full description of its properties in the **Properties** section.



`plot(x,y,'PropertyName',Value)`

Summary

- Importing data from file
- Normalizing data
- Dealing with missing data
- Polynomial fitting
- Creating customized visualizations

Try

Open and run the script `electricityAnalysis.mlx`.

Function	Use
`xlsread` `readtable`	Import data from spreadsheet
`isnan` `ismissing`	Identify missing values in data
`rmmissing`	Remove missing values in data
`fillmissing`	Interpolate missing data
`plotmatrix` `corrcoef`	Visualize/calculate correlations between variables
`movmean` `movmedian`	Compute summary statistic over a moving window
`polyfit` `polyval`	Fit and evaluate polynomial models
`text`	Add text annotations to plot
`num2str`	Create text representation of a numeric value

Test Your Knowledge

Name:____

1. Which of the following makes a plot with a thick line?

   A.
   ```
   p = plot(x,y);
   LineWidth(p,4)
   ```
   B.
   ```
   plot(x,y,'LineWidth'=4)
   ```
   C.
   ```
   p = plot(x,y);
   p(LineWidth) = 4;
   ```
   D.
   ```
   plot(x,y,'LineWidth',4)
   ```

2. Given 1-by-50 vectors `x` and `y`, what is the result of the command
   `z = polyfit(x,y,3)`?

   A. A 1-by-3 vector of points interpolating `y` as a function of `x`

   B. A 1-by-4 vector representing the coefficients of a cubic polynomial fitted to `y` as a function of `x`

   C. A 1-by-50 vector of the values of a cubic polynomial fitted to `y` as a function of `x`

   D. An error message