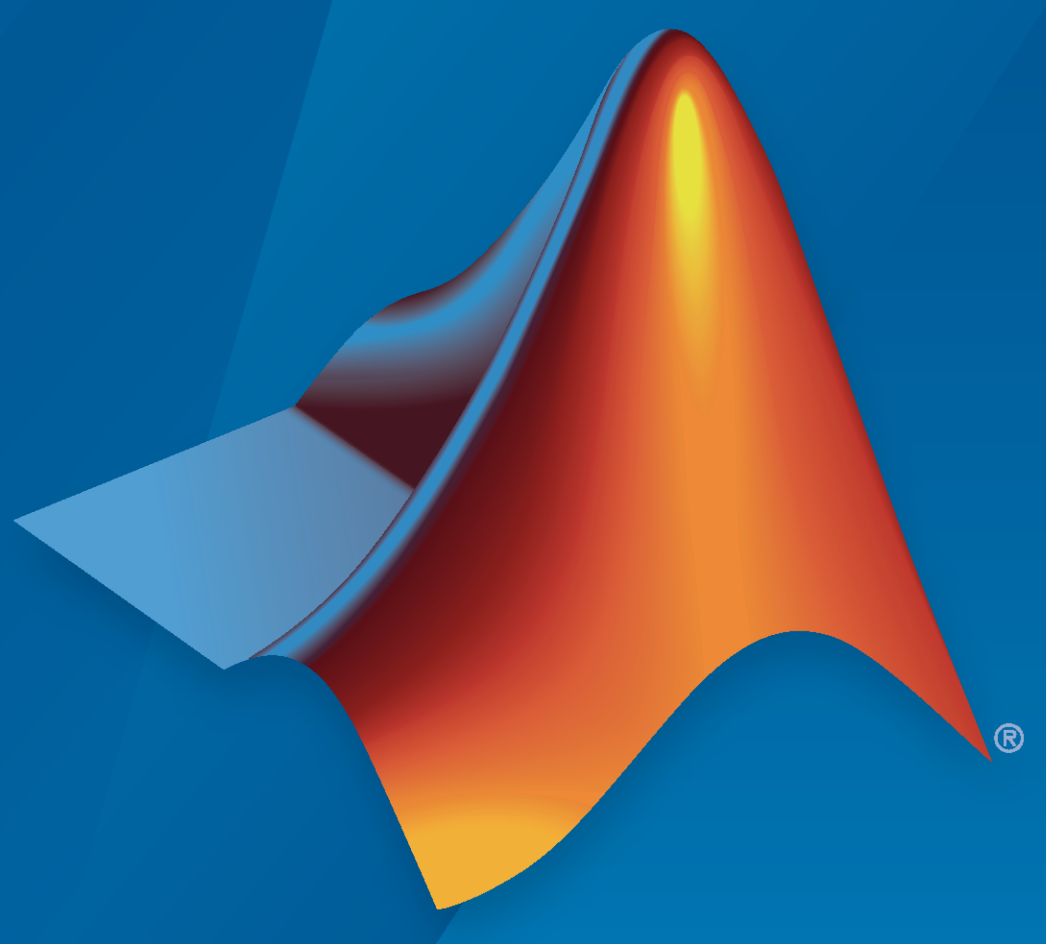
MATLAB®

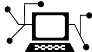
Data Analysis



MATLAB®

R2025b

How to Contact MathWorks

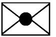


Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000

The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

MATLAB® Data Analysis

© COPYRIGHT 2005–2025 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

September 2005	Online only	New for MATLAB 7.1 (Release 14SP3)
March 2006	Online only	Revised for MATLAB 7.2 (Release 2006a)
September 2006	Online only	Revised for MATLAB 7.3 (Release 2006b)
March 2007	Online only	Revised for MATLAB 7.4 (Release 2007a)
September 2007	Online only	Revised for MATLAB 7.5 (Release 2007b)
March 2008	Online only	Revised for MATLAB 7.6 (Release 2008a)
October 2008	Online only	Revised for MATLAB 7.7 (Release 2008b)
March 2009	Online only	Revised for MATLAB 7.8 (Release 2009a)
September 2009	Online only	Revised for MATLAB 7.9 (Release 2009b)
March 2010	Online only	Revised for MATLAB 7.10 (Release 2010a)
September 2010	Online only	Revised for MATLAB 7.11 (Release 2010b)
April 2011	Online only	Revised for MATLAB 7.12 (Release 2011a)
September 2011	Online only	Revised for MATLAB 7.13 (Release 2011b)
March 2012	Online only	Revised for MATLAB 7.14 (Release 2012a)
September 2012	Online only	Revised for MATLAB 8.0 (Release 2012b)
March 2013	Online only	Revised for MATLAB 8.1 (Release 2013a)
September 2013	Online only	Revised for MATLAB 8.2 (Release 2013b)
March 2014	Online only	Revised for MATLAB 8.3 (Release 2014a)
October 2014	Online only	Revised for MATLAB 8.4 (Release 2014b)
March 2015	Online only	Revised for MATLAB 8.5 (Release 2015a)
September 2015	Online only	Revised for MATLAB 8.6 (Release 2015b)
March 2016	Online only	Revised for MATLAB 9.0 (Release 2016a)
September 2016	Online only	Revised for MATLAB 9.1 (Release 2016b)
March 2017	Online only	Revised for MATLAB 9.2 (Release 2017a)
September 2017	Online only	Revised for MATLAB 9.3 (Release 2017b)
March 2018	Online only	Revised for MATLAB 9.4 (Release 2018a)
September 2018	Online only	Revised for MATLAB 9.5 (Release 2018b)
March 2019	Online only	Revised for MATLAB 9.6 (Release 2019a)
September 2019	Online only	Revised for MATLAB 9.7 (Release 2019b)
March 2020	Online only	Revised for MATLAB 9.8 (Release 2020a)
September 2020	Online only	Revised for MATLAB 9.9 (Release 2020b)
March 2021	Online only	Revised for MATLAB 9.10 (Release 2021a)
September 2021	Online only	Revised for MATLAB 9.11 (Release 2021b)
March 2022	Online only	Revised for MATLAB 9.12 (Release 2022a)
September 2022	Online only	Revised for MATLAB 9.13 (Release 2022b)
March 2023	Online only	Revised for MATLAB 9.14 (Release 2023a)
September 2023	Online only	Revised for Version 23.2 (R2023b)
March 2024	Online only	Revised for Version 24.1 (R2024a)
September 2024	Online only	Revised for Version 24.2 (R2024b)
March 2025	Online only	Revised for Version 25.1 (R2025a)
September 2025	Online only	Rereleased for Version 25.2 (R2025b)

Contents

1 Data Processing

Importing and Exporting Data **1-2**
Importing Data into the Workspace **1-2**
Exporting Data from the Workspace **1-2**

Plotting Data **1-3**
Introduction **1-3**
Load and Plot Data from Text File **1-3**

Remove Linear Trends from Timetable Data **1-5**

Missing Data in MATLAB **1-8**

Data Smoothing and Outlier Detection **1-12**

Summarize or Pivot Data in Tables Using Groups **1-23**

Clean Messy Data and Locate Extrema Using Live Editor Tasks **1-33**

Filter Data **1-39**
Filter Difference Equation **1-39**
Moving-Average Filter of Traffic Data **1-39**
Modify Amplitude of Data **1-40**

Smooth Data with Convolution **1-42**

Computing with Descriptive Statistics **1-46**
Functions for Calculating Descriptive Statistics **1-46**
Example: Using MATLAB Data Statistics **1-47**
Data Statistics **1-48**

Identify and Visualize Correlated Variables **1-55**

2 Regression Analysis

Linear Correlation **2-2**
Introduction **2-2**
Covariance **2-2**
Correlation Coefficients **2-3**

Linear Regression 2-5
Introduction 2-5
Simple Linear Regression 2-5
Residuals and Goodness of Fit 2-8
Fitting Data with Curve Fitting Toolbox Functions 2-11

Interactive Fitting 2-13
Basic Fitting UI 2-13
Preparing for Basic Fitting 2-13
Opening the Basic Fitting UI 2-13
Example: Using Basic Fitting UI 2-14

Programmatic Fitting 2-26
MATLAB Functions for Polynomial Models 2-26
Linear Model with Nonpolynomial Terms 2-26
Multiple Regression 2-27
Programmatic Fitting 2-28

3 Time Series Analysis

Time Series Objects and Collections 3-2

4 Manage Experiments

Keyboard Shortcuts for Experiment Manager 4-2
Shortcuts for General Navigation 4-2
Shortcuts for Experiment Browser 4-2
Shortcuts for Results Table 4-3

Run Experiments in Parallel 4-4
Run Multiple Simultaneous Trials 4-4
Run Single Trial on Multiple Workers 4-4
Set Up Parallel Environment 4-4

Offload Experiments as Batch Jobs to a Cluster 4-6
Create Batch Job on Cluster 4-6
Track Progress of Batch Job 4-7
Cancel Batch Job 4-8
Delete Batch Job 4-8

Debug General-Purpose Experiments 4-9
Start Debugging Session 4-9
Debug Experiment Function 4-10
Verify Your Results 4-10

Experiment with Predator-Prey Equations 4-11

Compare Air Resistance Models for Projectile Motion **4-19**

Convert MATLAB Code into Experiment **4-28**

Data Processing

- “Importing and Exporting Data” on page 1-2
- “Plotting Data” on page 1-3
- “Remove Linear Trends from Timetable Data” on page 1-5
- “Missing Data in MATLAB” on page 1-8
- “Data Smoothing and Outlier Detection” on page 1-12
- “Summarize or Pivot Data in Tables Using Groups” on page 1-23
- “Clean Messy Data and Locate Extrema Using Live Editor Tasks” on page 1-33
- “Filter Data” on page 1-39
- “Smooth Data with Convolution” on page 1-42
- “Computing with Descriptive Statistics” on page 1-46
- “Identify and Visualize Correlated Variables” on page 1-55

Importing and Exporting Data

In this section...
“Importing Data into the Workspace” on page 1-2
“Exporting Data from the Workspace” on page 1-2

Importing Data into the Workspace

The first step in analyzing data is to import it into the MATLAB workspace. See “Supported File Formats for Import and Export” for information about importing data from specific file formats.

Exporting Data from the Workspace

When you analyze your data, you might create new variables or modify imported variables. You can export variables from the MATLAB workspace to various file formats, both character-based and binary. You can, for example, create HDF and Microsoft® Excel® files containing your data. For details, see the documentation on “Supported File Formats for Import and Export”.

Plotting Data

In this section...
“Introduction” on page 1-3
“Load and Plot Data from Text File” on page 1-3

Introduction

After you import data into the MATLAB workspace, it is a good idea to plot the data so that you can explore its features. An exploratory plot of your data enables you to identify discontinuities and potential outliers, as well as the regions of interest.

The MATLAB figure window displays plots. See “Types of MATLAB Plots” for a full description of the figure window. It also discusses the various interactive tools available for editing and customizing MATLAB graphics.

Load and Plot Data from Text File

This example uses sample data in `count.dat`, a space-delimited text file. The file consists of three sets of hourly traffic counts, recorded at three different town intersections over a 24-hour period. Each data column in the file represents data for one intersection.

Load the count.dat Data

Import data into the workspace using the `load` function.

```
load count.dat
```

Loading this data creates a 24-by-3 matrix called `count` in the MATLAB workspace.

Get the size of the data matrix.

```
[n,p] = size(count)
```

```
n =
24

p =
3
```

`n` represents the number of rows, and `p` represents the number of columns.

Plot the count.dat Data

Create a time vector, `t`, containing integers from `1` to `n`.

```
t = 1:n;
```

Plot the data as a function of time, and annotate the plot.

```
plot(t,count),
legend('Location 1','Location 2','Location 3','Location','NorthWest')
```

```
xlabel('Time'), ylabel('Vehicle Count')
title('Traffic Counts at Three Intersections')
```

See Also

load | plot | legend | xlabel | ylabel | title | size

More About

- “Types of MATLAB Plots”

Remove Linear Trends from Timetable Data

This example shows how to remove a linear trend from daily closing stock prices in a timetable by using the `detrend` function. Alternatively, you can interactively find and remove trends from data using the Find and Remove Trends task in the Live Editor. If the data does have a trend, detrending forces the mean of the detrended data to zero and reduces overall variation.

Create Stock Data

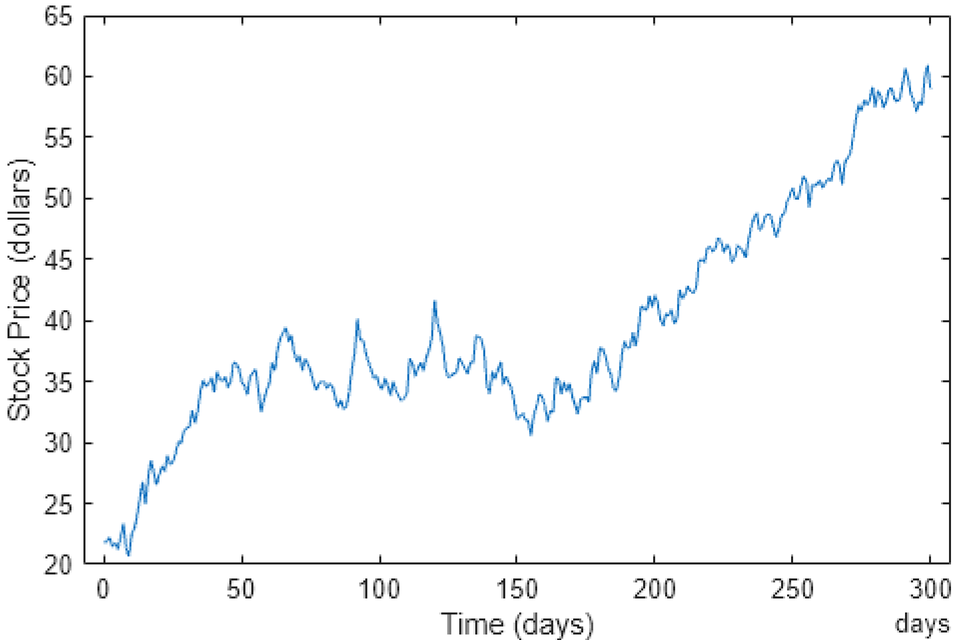
Create a sample timetable containing daily closing stock prices. Use randomly sampled numbers from a normal distribution.

```
x = 0:300;
Time = days(x)';
dailyFluct = gallery("normaldata",size(x),2);
closing = cumsum(dailyFluct) + 20 + x/100;
StockPrice = closing';
TT = timetable(Time,StockPrice)
```

```
TT=301×1 timetable
     Time      StockPrice
    _______    __________

    0 days       21.749
    1 day        21.892
    2 days       22.227
    3 days       21.443
    4 days       21.768
    5 days       21.251
    6 days       22.193
    7 days       23.368
    8 days       21.332
    9 days       20.698
    10 days      22.449
    11 days      22.946
    12 days      24.004
    13 days      25.503
    14 days      26.783
    15 days      24.937
      ⋮
```

Plot and label the stock price data.

```
plot(TT,"Time","StockPrice");
xlabel("Time (days)");
ylabel("Stock Price (dollars)");
```



Remove Trend

Apply `detrend`, which performs a linear fit to the stock prices and removes the trend. Specify to append the detrended data to the input timetable.

```
TT = detrend(TT,ReplaceValues=false);
```

Compute the trend line by subtracting the detrended data from the input data.

```
trend = TT.StockPrice - TT.StockPrice_detrended;
TT = addvars(TT,trend,NewVariableNames="Trend")
```

TT=*301×3 timetable*

Time	StockPrice	StockPrice_detrended	Trend
0 days	21.749	-3.7893	25.538
1 day	21.892	-3.7397	25.631
2 days	22.227	-3.4975	25.724
3 days	21.443	-4.3742	25.817
4 days	21.768	-4.1423	25.91
5 days	21.251	-4.7525	26.003
6 days	22.193	-3.9033	26.096
7 days	23.368	-2.8216	26.189
8 days	21.332	-4.9502	26.282
9 days	20.698	-5.6776	26.375
10 days	22.449	-4.0195	26.468
11 days	22.946	-3.6157	26.561
12 days	24.004	-2.6498	26.654
13 days	25.503	-1.2442	26.747
14 days	26.783	-0.056718	26.84
15 days	24.937	-1.9958	26.933
⋮			

Find the average closing price of the detrended data.

```
average_detrended = mean(TT.StockPrice_detrended)
```

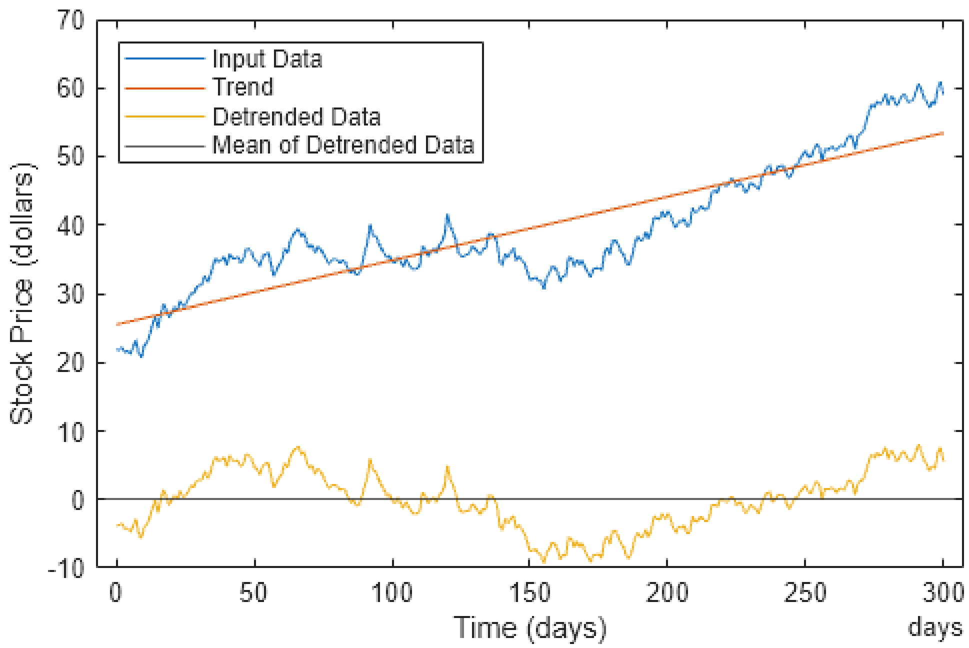
```
average_detrended =
-1.2464e-14
```

As expected, the detrended data has a mean very close to 0.

Visualize Detrended Data

Display the results by plotting the original daily closing stock prices, the trend line, the detrended data, and its mean.

```
plot(TT,"StockPrice")
hold on
plot(TT,"Trend")
plot(TT,"StockPrice_detrended")
yline(average_detrended)
legend("Input Data","Trend","Detrended Data", ...
    "Mean of Detrended Data","Location","northwest")
xlabel("Time (days)");
ylabel("Stock Price (dollars)");
```



See Also

Live Editor Tasks
Find and Remove Trends

Functions
`detrend` | `gallery` | `plot` | `cumsum`

Missing Data in MATLAB

Working with missing data is a common task in data preprocessing. Although sometimes missing values signify a meaningful event in the data, they often represent unreliable or unusable data points. In either case, MATLAB® has many options for handling missing data.

Create and Organize Missing Data

The form that missing values take in MATLAB depends on the data type. For example, numeric data types such as `double` use `NaN` (not a number) to represent missing values.

```
x = [NaN 1 2 3 4];
```

You can also use the `missing` value to represent missing numeric data or data of other types, such as `datetime`, `string`, and `categorical`. MATLAB automatically converts the `missing` value to the data's native type.

```
xDouble = [missing 1 2 3 4]
```

```
xDouble = 1×5

   NaN     1     2     3     4
```

```
xDatetime = [missing datetime(2014,1:4,1)]
```

```
xDatetime = 1×5 datetime
   NaT           01-Jan-2014   01-Feb-2014   01-Mar-2014   01-Apr-2014
```

```
xString = [missing "a" "b" "c" "d"]
```

```
xString = 1×5 string
    <missing>    "a"    "b"    "c"    "d"
```

```
xCategorical = [missing categorical({'cat1' 'cat2' 'cat3' 'cat4'})]
```

```
xCategorical = 1×5 categorical
     <undefined>      cat1      cat2      cat3      cat4
```

A data set might contain values that you want to treat as missing data, but are not standard MATLAB missing values in MATLAB such as `NaN`. You can use the `standardizeMissing` function to convert those values to the standard missing value for that data type. For example, treat 4 as a missing `double` value in addition to `NaN`.

```
xStandard = standardizeMissing(xDouble,[4 NaN])
```

```
xStandard = 1×5

   NaN     1     2     3   NaN
```

Suppose you want to keep missing values as part of your data set but segregate them from the rest of the data. Several MATLAB functions enable you to control the placement of missing values before

further processing. For example, use the 'MissingPlacement' option with the sort function to move NaNs to the end of the data.

```
xSort = sort(xStandard,'MissingPlacement','last')
```

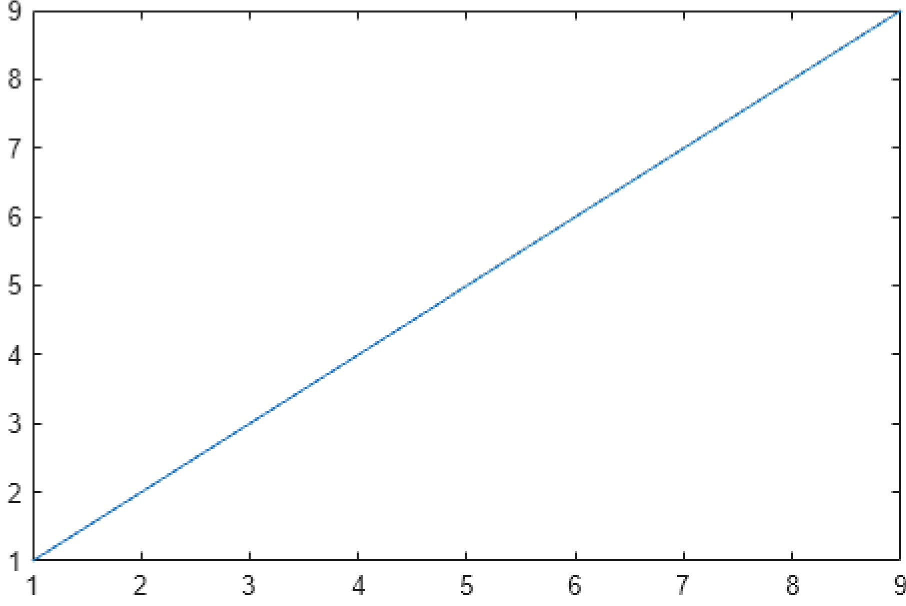
```
xSort = 1×5

     1     2     3   NaN   NaN
```

Find, Replace, and Ignore Missing Data

Even if you do not explicitly create missing values in MATLAB, they can appear when importing existing data or computing with the data. If you are not aware of missing values in your data, subsequent computation or analysis can be misleading.

For example, if you unknowingly plot a vector containing a NaN value, the NaN does not appear because the plot function ignores it and plots the remaining points normally.

```
nanData = [1:9 NaN];
plot(1:10,nanData)
```



However, if you compute the average of the data, the result is NaN. In this case, it is more helpful to know in advance that the data contains a NaN, and then choose to ignore or remove it before computing the average.

```
meanData = mean(nanData)
```

```
meanData =
NaN
```

One way to find NaNs in data is by using the isnan function, which returns a logical array indicating the location of any NaN value.

```
TF = isnan(nanData)
```

```
TF = 1×10 logical array

   0   0   0   0   0   0   0   0   0   1
```

Similarly, the `ismissing` function returns the location of missing values in data for multiple data types.

```
TFdouble = ismissing(xDouble)
```

```
TFdouble = 1×5 logical array

   1   0   0   0   0
```

```
TFdatetime = ismissing(xDatetime)
```

```
TFdatetime = 1×5 logical array

   1   0   0   0   0
```

Suppose you are working with a table or timetable made up of variables with multiple data types. You can find all of the missing values with one call to `ismissing`, regardless of their type.

```
xTable = table(xDouble',xDatetime',xString',xCategorical')
```

```
xTable=5×4 table
    Var1       Var2           Var3           Var4
    ____    ___________    _________    ___________

    NaN             NaT    <missing>    <undefined>
      1     01-Jan-2014    "a"          cat1
      2     01-Feb-2014    "b"          cat2
      3     01-Mar-2014    "c"          cat3
      4     01-Apr-2014    "d"          cat4
```

```
TF = ismissing(xTable)
```

```
TF = 5×4 logical array

   1   1   1   1
   0   0   0   0
   0   0   0   0
   0   0   0   0
   0   0   0   0
```

Missing values can represent unusable data for processing or analysis. Use `fillmissing` to replace missing values with another value, or use `rmmissing` to remove missing values altogether.

```
xFill = fillmissing(xStandard,'constant',0)
```

```
xFill = 1×5

     0     1     2     3     0
```

```
xRemove = rmmissing(xStandard)
```

```
xRemove = 1×3

     1     2     3
```

Many MATLAB functions enable you to ignore missing values, without having to explicitly locate, fill, or remove them first. For example, if you compute the sum of a vector containing `NaN` values, the result is `NaN`. However, you can directly ignore `NaN`s in the sum by using the `'omitnan'` option with the `sum` function.

```
sumNan = sum(xDouble)
```

```
sumNan = 
NaN
```

```
sumOmitnan = sum(xDouble,'omitnan')
```

```
sumOmitnan = 
10
```

See Also
`ismissing` | `fillmissing` | `standardizeMissing` | `missing`

Related Examples
- Clean Messy Data and Locate Extrema Using Live Editor Tasks on page 1-33
- “Clean Messy and Missing Data in Tables”

Data Smoothing and Outlier Detection

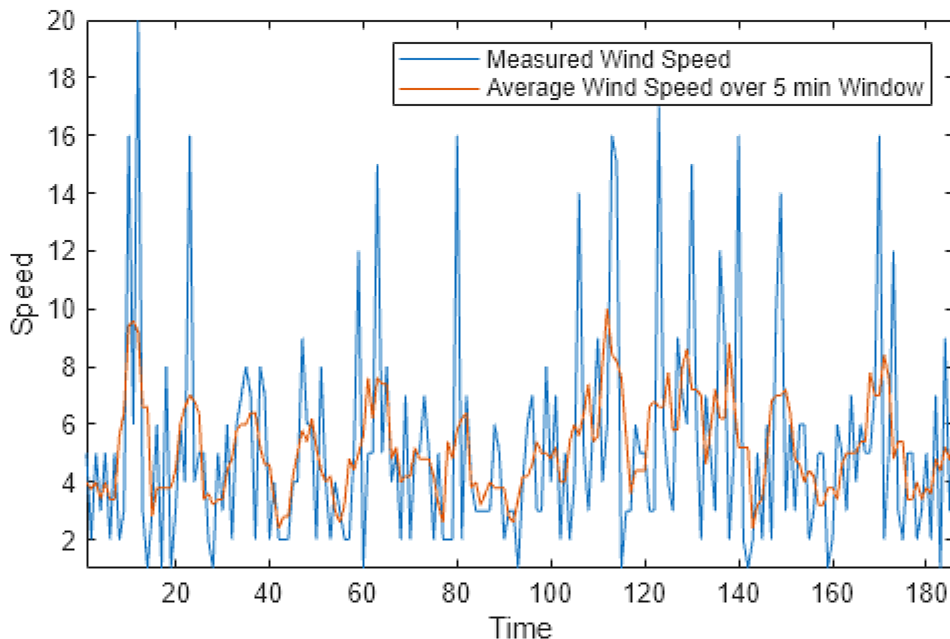
This example shows how to use data smoothing to eliminate unwanted noise or behaviors in data using the `smoothdata` function. This example also shows how to identify and handle outliers, which are data points that are significantly different from the rest of the data, using the `isoutlier`, `filloutliers`, or `rmoutliers` functions. Alternatively, you can interactively smooth noisy data and handle outliers in data using the Smooth Data and Clean Outlier Data tasks in the Live Editor.

Moving Window Methods

Moving window methods are ways to process data in smaller batches at a time, typically in order to statistically represent a neighborhood of points in the data. The moving average is a common data smoothing technique that slides a window along the data, computing the mean of the points inside of each window. This can help to eliminate insignificant variations from one data point to the next.

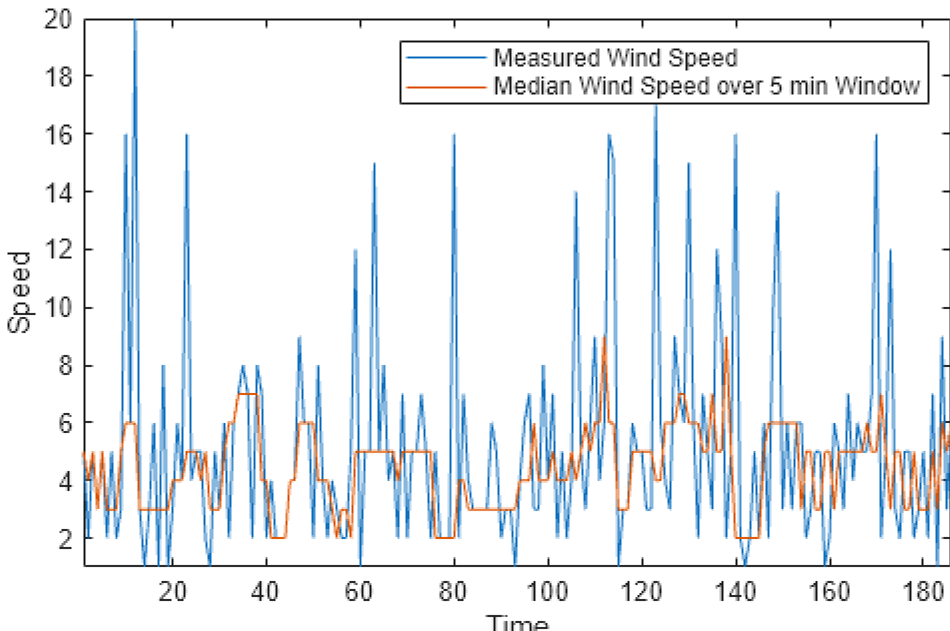
For example, consider wind speed measurements taken every minute for about 3 hours. Use the `movmean` function with a window size of 5 minutes to smooth out high-speed wind gusts.

```
load windData.mat
mins = 1:length(speed);
window = 5;
meanspeed = movmean(speed,window);
plot(mins,speed,mins,meanspeed)
axis tight
legend("Measured Wind Speed","Average Wind Speed over 5 min Window")
xlabel("Time")
ylabel("Speed")
```



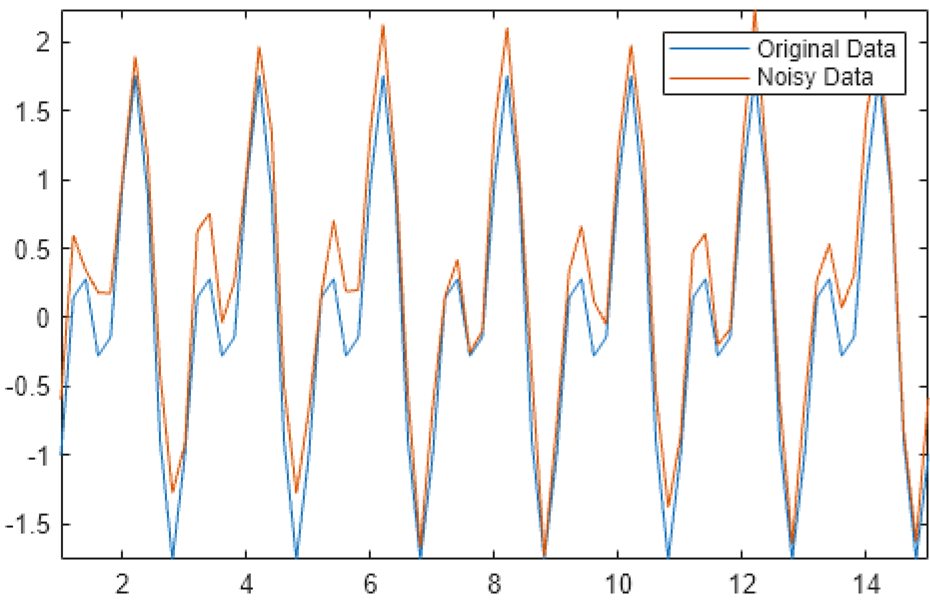
Similarly, you can compute the median wind speed over a sliding window using the `movmedian` function.

```
medianspeed = movmedian(speed,window);
plot(mins,speed,mins,medianspeed)
axis tight
legend("Measured Wind Speed","Median Wind Speed over 5 min Window")
xlabel("Time")
ylabel("Speed")
```



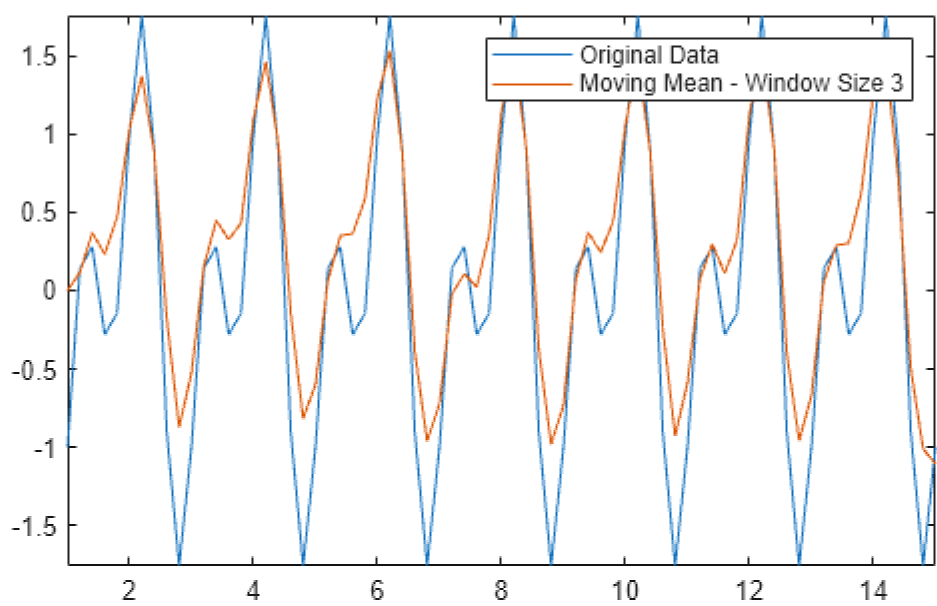
Not all data is suitable for smoothing with a moving window method. For example, create a sinusoidal signal with injected random noise.

```
t = 1:0.2:15;
A = sin(2*pi*t) + cos(2*pi*0.5*t);
Anoise = A + 0.5*rand(1,length(t));
plot(t,A,t,Anoise)
axis tight
legend("Original Data","Noisy Data")
```



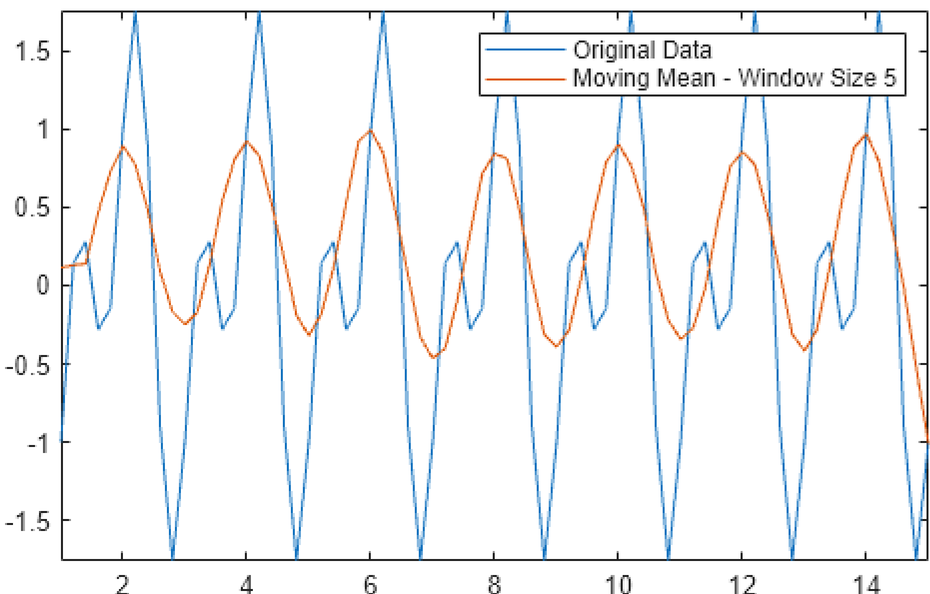
Use a moving mean with a window size of 3 to smooth the noisy data.

```
window = 3;
Amean = movmean(Anoise,window);
plot(t,A,t,Amean)
axis tight
legend("Original Data","Moving Mean - Window Size 3")
```



The moving mean achieves the general shape of the data, but doesn't capture the valleys (local minima) very accurately. Since the valley points are surrounded by two larger neighbors in each window, the mean is not a very good approximation to those points. If you make the window size larger, the mean eliminates the shorter peaks altogether. For this type of data, you might consider alternative smoothing techniques.

```
Amean = movmean(Anoise,5);
plot(t,A,t,Amean)
axis tight
legend("Original Data","Moving Mean - Window Size 5")
```

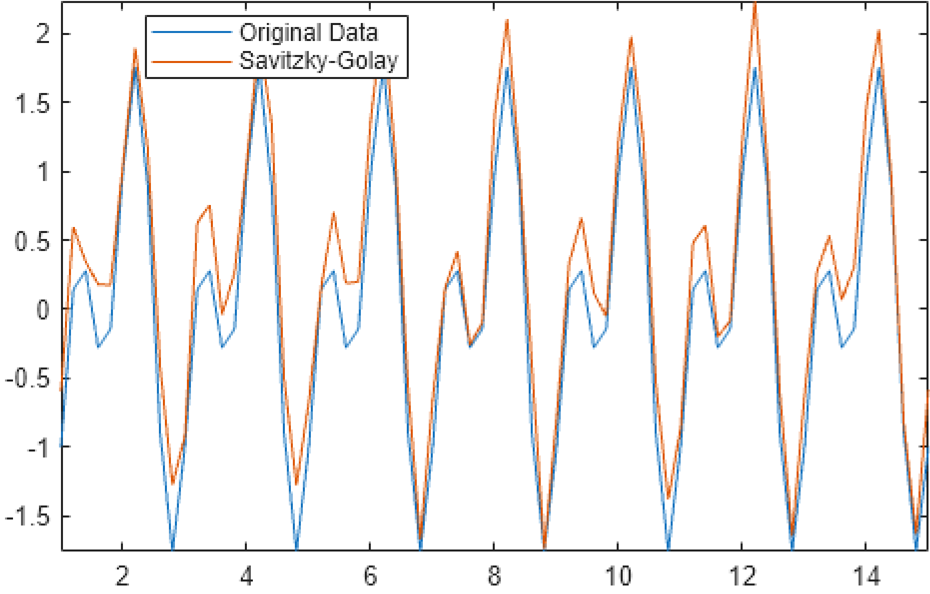


Common Smoothing Methods

The `smoothdata` function provides several smoothing options such as the Savitzky-Golay method, which is a popular smoothing technique used in signal processing. By default, `smoothdata` chooses a best-guess window size for the method depending on the data.

Use the Savitzky-Golay method to smooth the noisy signal `Anoise`, and output the window size that it uses. This method provides a better valley approximation compared to `movmean`.

```
[Asgolay,window] = smoothdata(Anoise,"sgolay");
plot(t,A,t,Asgolay)
axis tight
legend("Original Data","Savitzky-Golay","location","best")
```

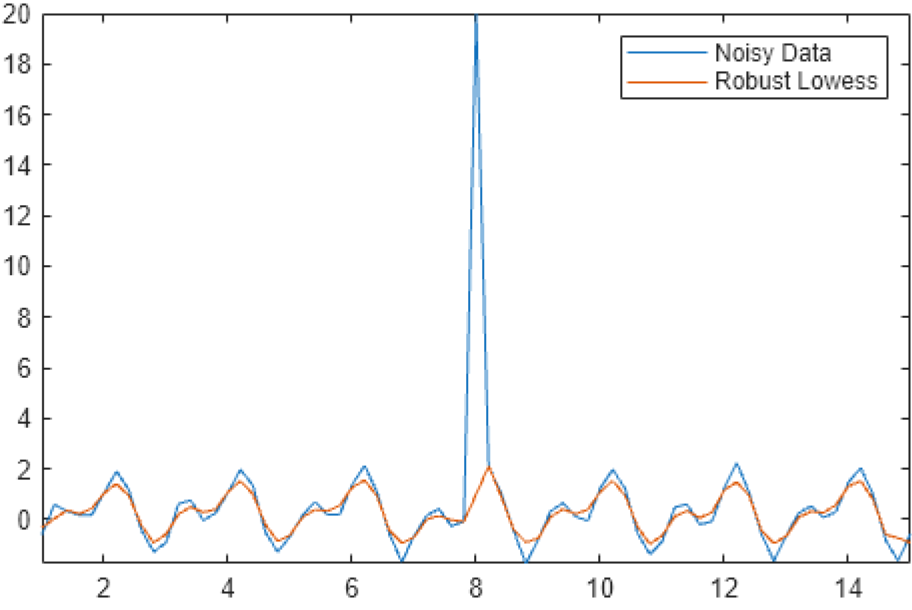



```
window
```

```
window =
3
```

The robust Lowess method is another smoothing method that is particularly helpful when outliers are present in the data in addition to noise. Inject an outlier into the noisy data, and use robust Lowess to smooth the data, which eliminates the outlier.

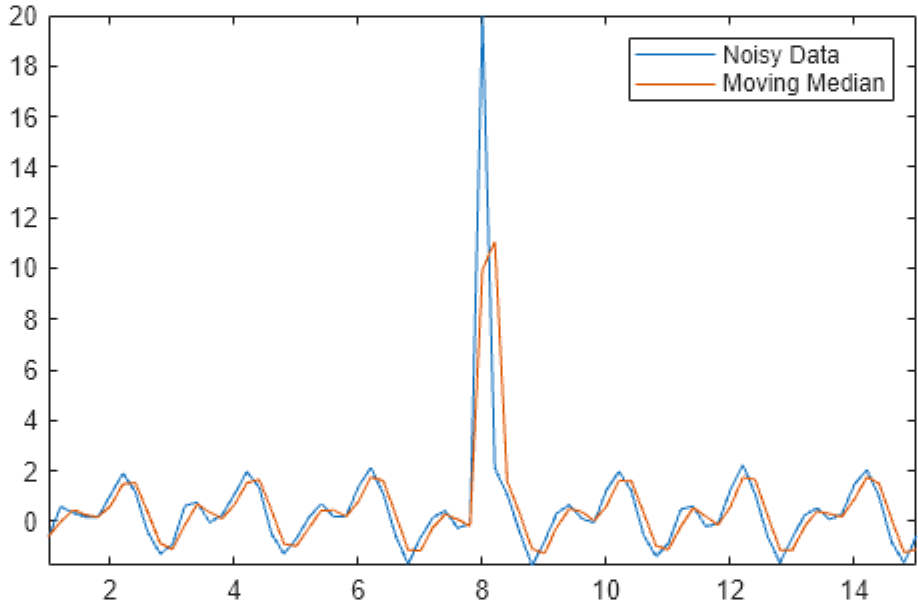
```
Anoise(36) = 20;
Arlowess = smoothdata(Anoise,"rlowess",5);
plot(t,Anoise,t,Arlowess)
axis tight
legend("Noisy Data","Robust Lowess")
```



Detecting Outliers

Outliers in data can significantly skew data processing results and other computed quantities. For example, if you try to smooth data containing outliers with a moving median, you can get misleading peaks or valleys.

```
Amedian = smoothdata(Anoise,"movmedian");
plot(t,Anoise,t,Amedian)
axis tight
legend("Noisy Data","Moving Median")
```



The `isoutlier` function returns a logical 1 when an outlier is detected. Verify the index and value of the outlier in `Anoise`.

```
TF = isoutlier(Anoise);
ind = find(TF)
```

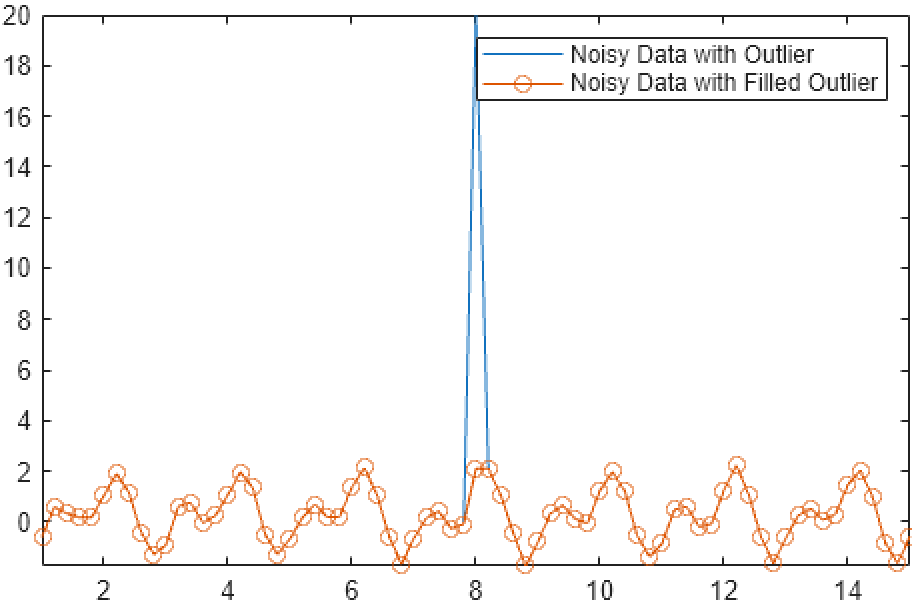
```
ind =
36
```

```
Aoutlier = Anoise(ind)
```

```
Aoutlier =
20
```

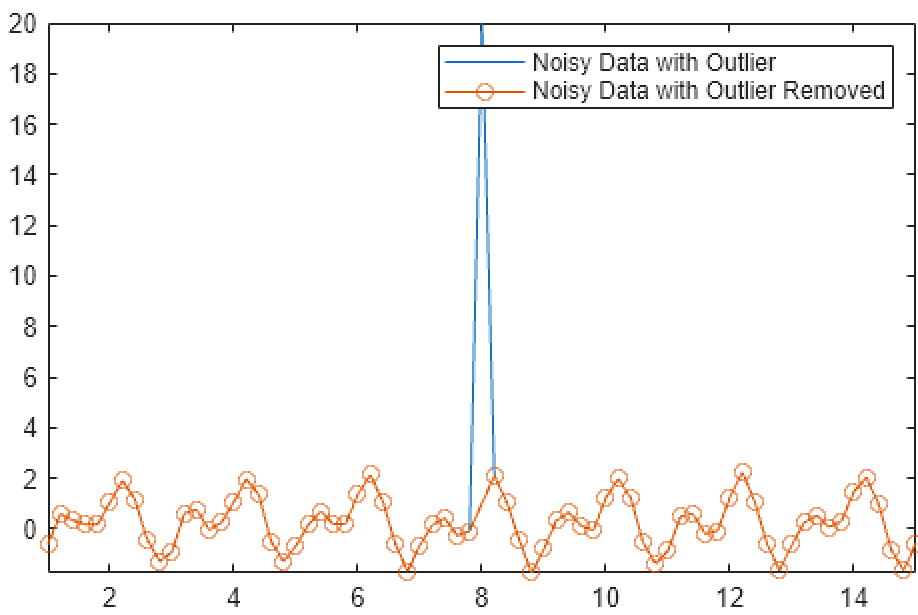
You can replace outliers in your data by using the `filloutliers` function and specifying a fill method. For example, fill the outlier in `Anoise` with the value of its neighbor immediately to the right.

```
Afill = filloutliers(Anoise,"next");
plot(t,Anoise,t,Afill,"o-")
axis tight
legend("Noisy Data with Outlier","Noisy Data with Filled Outlier")
```



Alternatively, you can remove outliers from your data by using the `rmoutliers` function. For example, remove the outlier in `Anoise`.

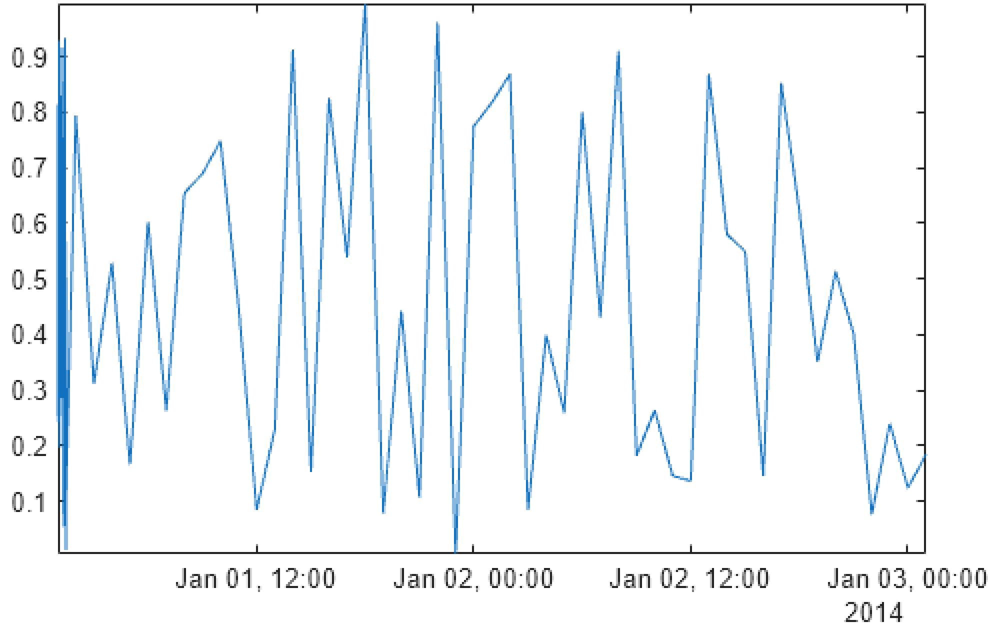
```
Aremove = rmoutliers(Anoise);
plot(t,Anoise,t(~TF),Aremove,"o-")
axis tight
legend("Noisy Data with Outlier","Noisy Data with Outlier Removed")
```



Nonuniform Data

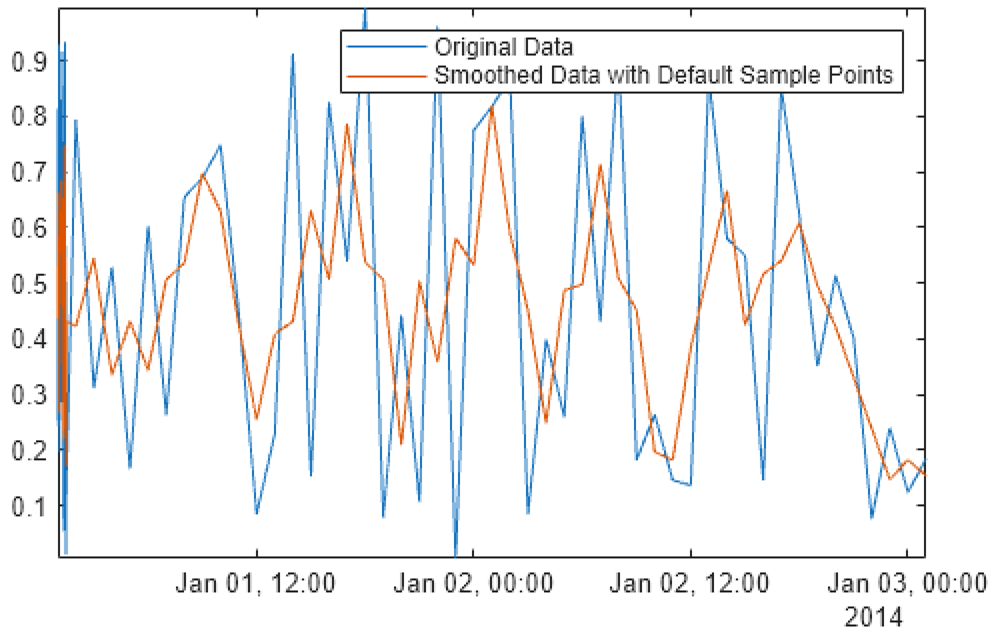
Not all data consists of equally spaced points, which can affect methods for data processing. Create a `datetime` vector that contains irregular sampling times for the data in `Airreg`. The `time` vector represents samples taken every minute for the first 30 minutes, then hourly over two days.

```
t0 = datetime(2014,1,1,1,1,1);
timeminutes = sort(t0 + minutes(1:30));
timehours = t0 + hours(1:48);
time = [timeminutes timehours];
Airreg = rand(1,length(time));
plot(time,Airreg)
axis tight
```



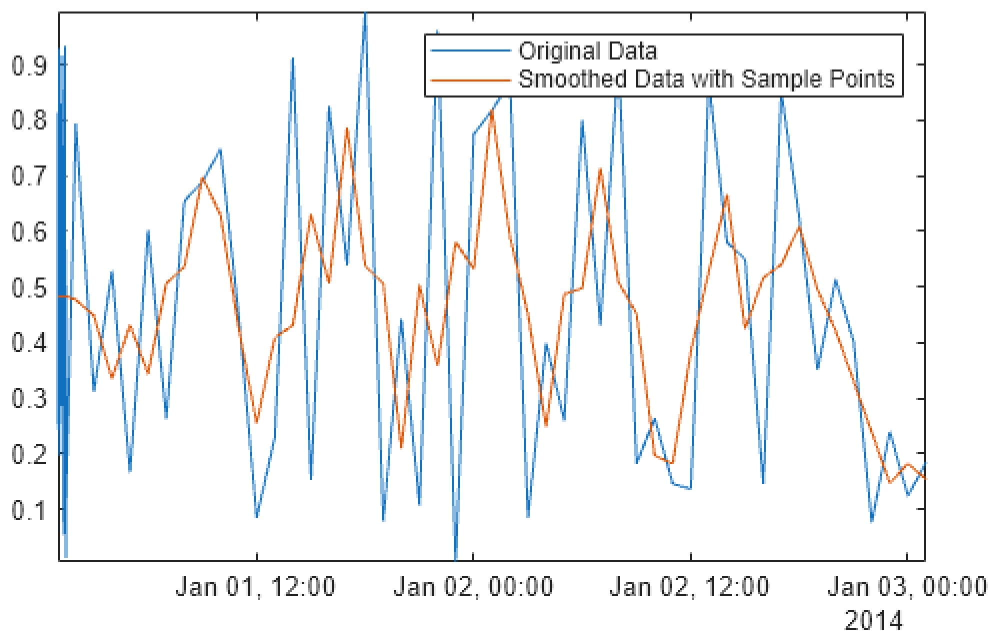
By default, `smoothdata` smooths with respect to equally spaced integers, in this case, `1,2,...,78`. Since integer time stamps do not coordinate with the sampling of the points in `Airreg`, the first half hour of data still appears noisy after smoothing.

```
Adefault = smoothdata(Airreg,"movmean",3);
plot(time,Airreg,time,Adefault)
axis tight
legend("Original Data","Smoothed Data with Default Sample Points")
```



Many data processing functions in MATLAB®, including `smoothdata`, `movmean`, and `filloutliers`, allow you to provide sample points, ensuring that data is processed relative to its sampling units and frequencies. To remove the high-frequency variation in the first half hour of data in `Airreg`, use the `SamplePoints` name-value argument with the time stamps in `time`.

```
Asamplepoints = smoothdata(Airreg,"movmean", ...
    hours(3),"SamplePoints",time);
plot(time,Airreg,time,Asamplepoints)
axis tight
legend("Original Data","Smoothed Data with Sample Points")
```



See Also

Functions
`smoothdata` | `isoutlier` | `filloutliers` | `rmoutliers` | `movmean` | `movmedian`

Live Editor Tasks
Smooth Data | Clean Outlier Data

Apps
Data Cleaner

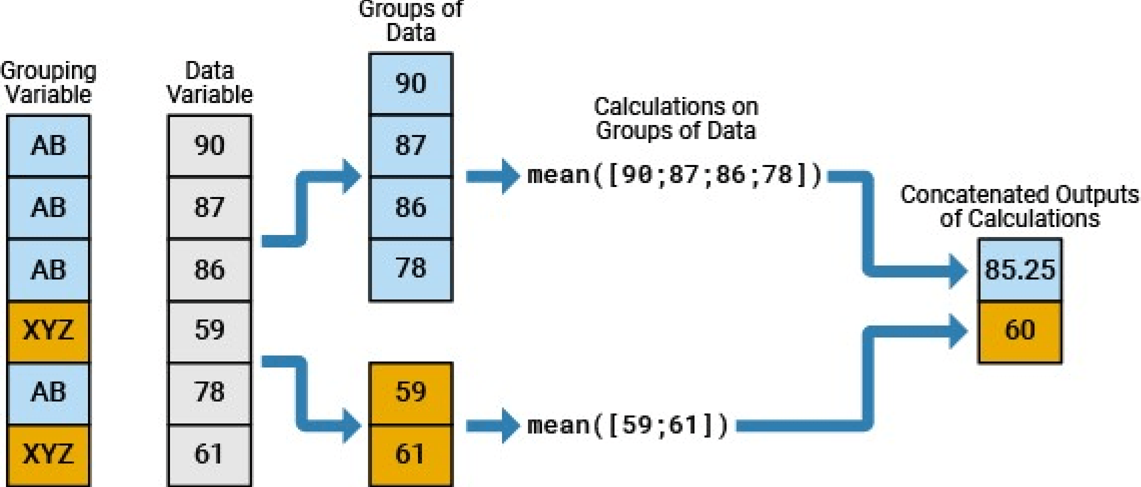
Related Examples

- Clean Messy Data and Locate Extrema Using Live Editor Tasks on page 1-33
- “Filter Data” on page 1-39

Summarize or Pivot Data in Tables Using Groups

When working with data in tables, you can often organize the data into groups. You can group tabular data to summarize and interpret the data based on common characteristics. For example, if your data consists of events over a large time period, you can group the data by year to identify trends over time. The table variables that define the grouping criteria are considered the *grouping variables*, and the table variables that contain the values associated with each group are considered the *data variables*. This example shows how to create a grouped summary table or a pivoted table to inspect and compare groups of data using either the `groupsummary` or `pivot` function, respectively. This example also shows how to create a grouped summary table or a pivoted table using the Compute by Group or Pivot Table tasks in the Live Editor.

In this image, values in a data variable are grouped according to a grouping variable and then summarized using the mean.



Import Data as Table

Import the sample data set `outages.csv`. The file contains data for utility power outages in the United States, such as the affected region, the outage cause, and the number of affected customers. You can organize this data into groups using a single variable or using multiple variables.

```
T = readtable("outages.csv","TextType","string")
```

```
T=1468×6 table
       Region           OutageTime          Loss     Customers       RestorationTime            Cause
    ___________    ________________    ______    __________    ________________    ______________

    "SouthWest"    2002-02-01 12:18    458.98    1.8202e+06    2002-02-07 16:50    "winter storm'
    "SouthEast"    2003-01-23 00:49    530.14    2.1204e+05                 NaT    "winter storm'
    "SouthEast"    2003-02-07 21:15     289.4    1.4294e+05    2003-02-17 08:14    "winter storm'
    "West"         2004-04-06 05:44    434.81    3.4037e+05    2004-04-06 06:10    "equipment fa
    "MidWest"      2002-03-16 06:18    186.44    2.1275e+05    2002-03-18 23:23    "severe storm'
    "West"         2003-06-18 02:49         0             0    2003-06-18 10:54    "attack"
    "West"         2004-06-20 14:39    231.29           NaN    2004-06-20 19:16    "equipment fa
    "West"         2002-06-06 19:28    311.86           NaN    2002-06-07 00:51    "equipment fa
    "NorthEast"    2003-07-16 16:23    239.93         49434    2003-07-17 01:12    "fire"
    "MidWest"      2004-09-27 11:09    286.72         66104    2004-09-27 16:37    "equipment fa
```

```
    "SouthEast"     2004-09-05 17:48    73.387         36073    2004-09-05 20:46    "equipment fau
    "West"          2004-05-21 21:45    159.99           NaN    2004-05-22 04:23    "equipment fau
    "SouthEast"     2002-09-01 18:22    95.917         36759    2002-09-01 19:12    "severe storm"
    "SouthEast"     2003-09-27 07:32       NaN    3.5517e+05    2003-10-04 07:02    "severe storm"
    "West"          2003-11-12 06:12    254.09    9.2429e+05    2003-11-17 02:04    "winter storm"
    "NorthEast"     2004-09-18 05:54         0             0                 NaT    "equipment fau
      ⋮
```

Summarize Data Using One Grouping Variable

When you have one grouping variable, you can create a grouped summary table with rows that correspond to each unique group using the `groupsummary` function. The variables in the grouped summary table represent the statistics computed per group for the data variables. This type of summary is particularly useful for identifying patterns in the data and making comparisons between different groups.

In this image, a table with one grouping variable and two data variables is summarized, and the grouped summary table shows the group counts and the mean of each group within each data variable.

Apply One Grouping Criterion

Compute the total power loss for each outage cause. Specify the grouping variable as `Cause` and the data variable as `Loss`.

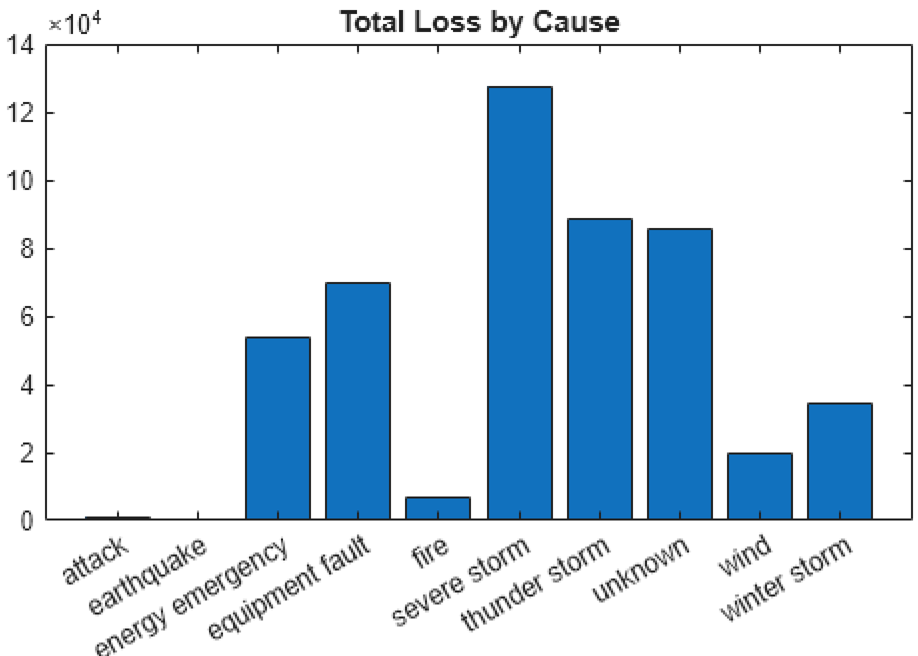
```
G1 = groupsummary(T,"Cause","sum","Loss")
```

```
G1=10×3 table
          Cause           GroupCount    sum_Loss
    __________________    __________    ________

    "attack"                 294         1057.7
    "earthquake"               2         258.18
    "energy emergency"       188          53983
    "equipment fault"        156          69428
    "fire"                    25         6709.6
```

```
    "severe storm"         338       1.2763e+05
    "thunder storm"        201            88754
    "unknown"               24            85366
    "wind"                  95            19524
    "winter storm"         145            34492
```

Visualize the grouped summary table using a bar chart.

```
bar(G1.Cause,G1.sum_Loss)
title("Total Loss by Cause")
```



Compute Multiple Statistics per Group

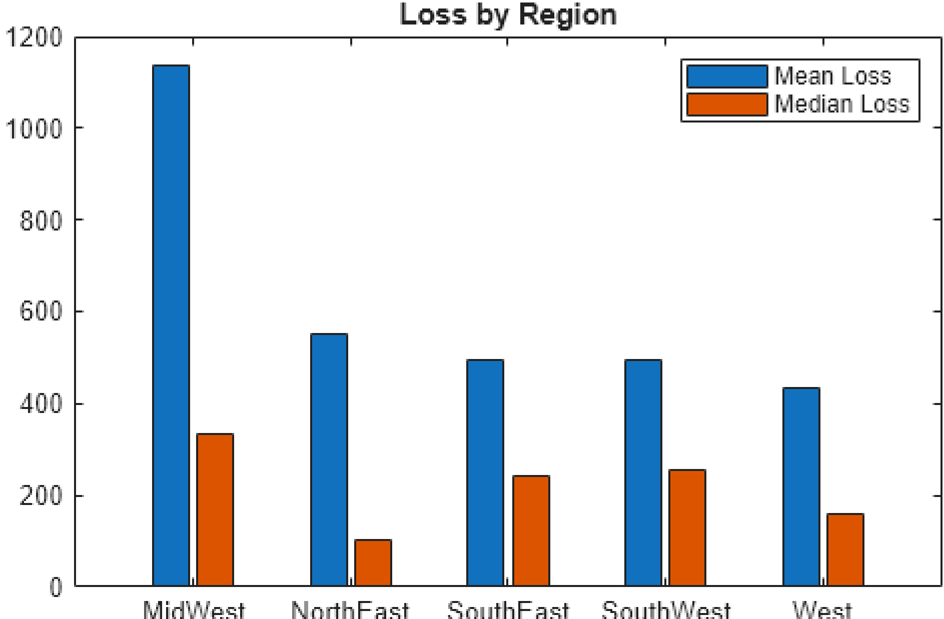
Compute the mean and median power loss for each region.

```
G2 = groupsummary(T,"Region",["mean" "median"],"Loss")
```

```
G2=5×4 table
      Region        GroupCount    mean_Loss    median_Loss
    ___________    __________    _________    ___________

    "MidWest"         142         1137.7        334.51
    "NorthEast"       557         551.65        101.73
    "SouthEast"       389         495.35        242.44
    "SouthWest"        26         493.88        256.74
    "West"            354         433.37         158.9
```

Visualize the grouped summary table using a bar chart. Each bar in a group of bars represents a different statistic. The statistics share a common scale because they represent the same data variable.

```
bar(G2.Region,[G2.mean_Loss G2.median_Loss])
legend("Mean Loss","Median Loss")
title("Loss by Region")
```



Compute Statistic for Multiple Data Variables

Compute the total power loss and total affected customers for each outage cause.

```
G3 = groupsummary(T,"Cause","sum",["Loss" "Customers"])
```

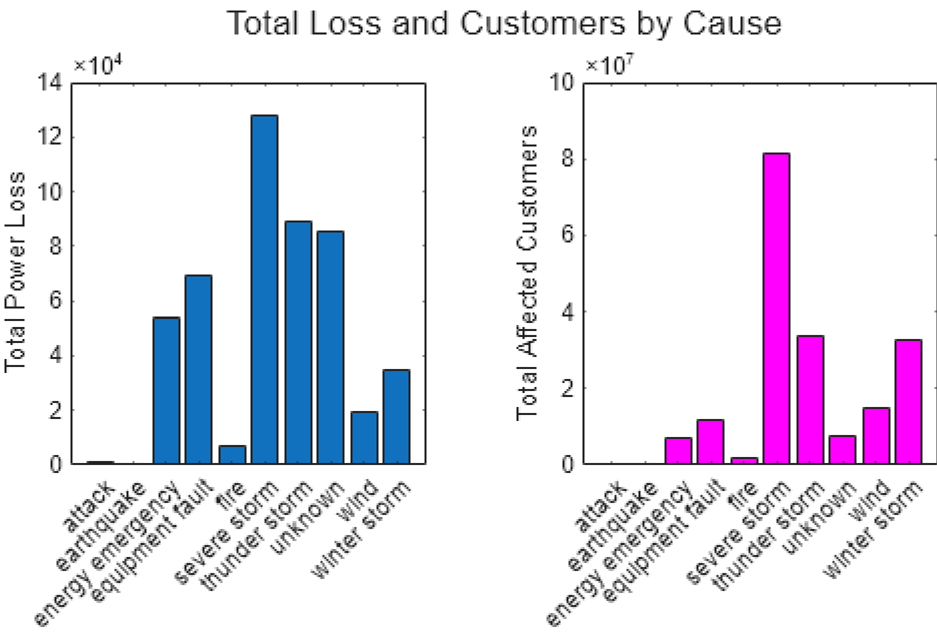
G3=*10×4 table*

Cause	GroupCount	sum_Loss	sum_Customers
"attack"	294	1057.7	25598
"earthquake"	2	258.18	1.3996e+05
"energy emergency"	188	53983	7.0441e+06
"equipment fault"	156	69428	1.1546e+07
"fire"	25	6709.6	1.6527e+06
"severe storm"	338	1.2763e+05	8.1392e+07
"thunder storm"	201	88754	3.3516e+07
"unknown"	24	85366	7.5306e+06
"wind"	95	19524	1.4724e+07
"winter storm"	145	34492	3.273e+07

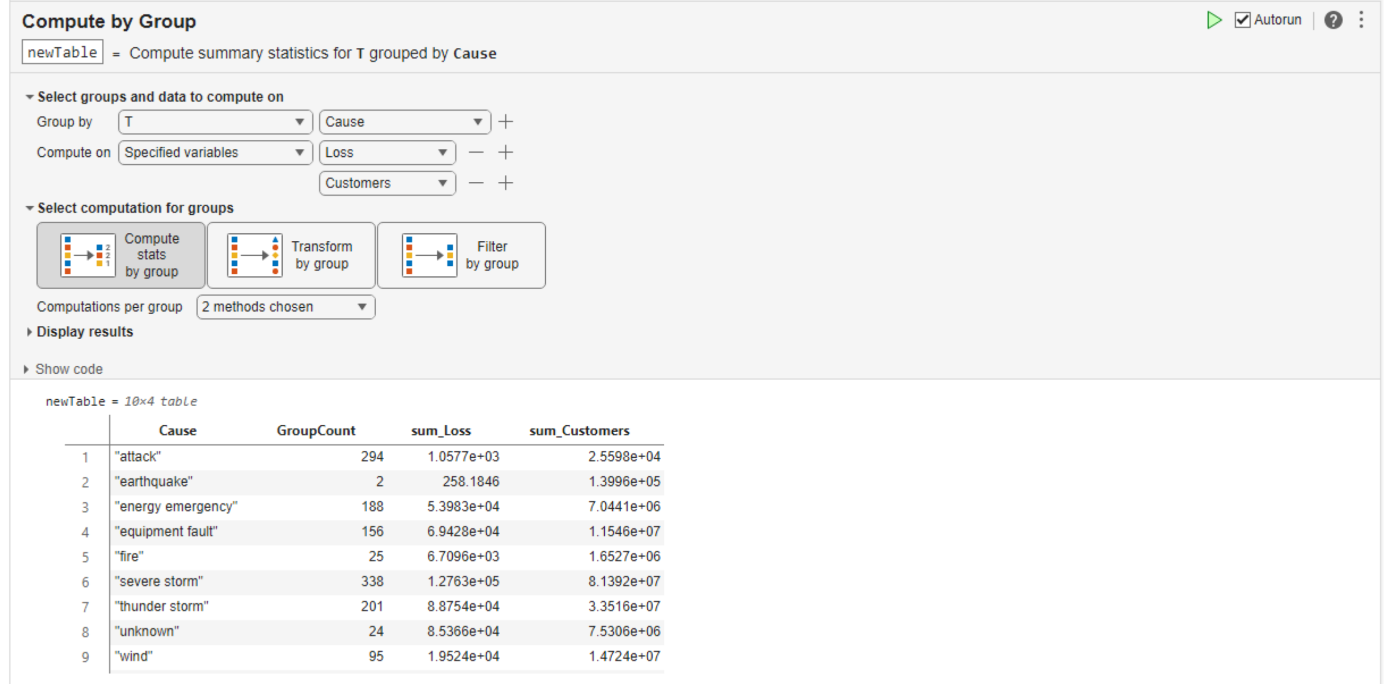
Visualize the grouped summary table using two bar charts. The statistics do not share a common scale because they represent different data variables.

```
ax = tiledlayout(1,2);
title(ax,"Total Loss and Customers by Cause")
nexttile
bar(G3.Cause,G3.sum_Loss)
ylabel("Total Power Loss")
```

```
nexttile
bar(G3.Cause,G3.sum_Customers,"magenta")
ylabel("Total Affected Customers")
```



Alternatively, to interactively summarize tabular data in a grouped summary table, use the Compute by Group Live Editor task. Live Editor Tasks are apps that you can embed in a live script to interactively explore parameters and options, immediately see the results, automatically generate the corresponding code.

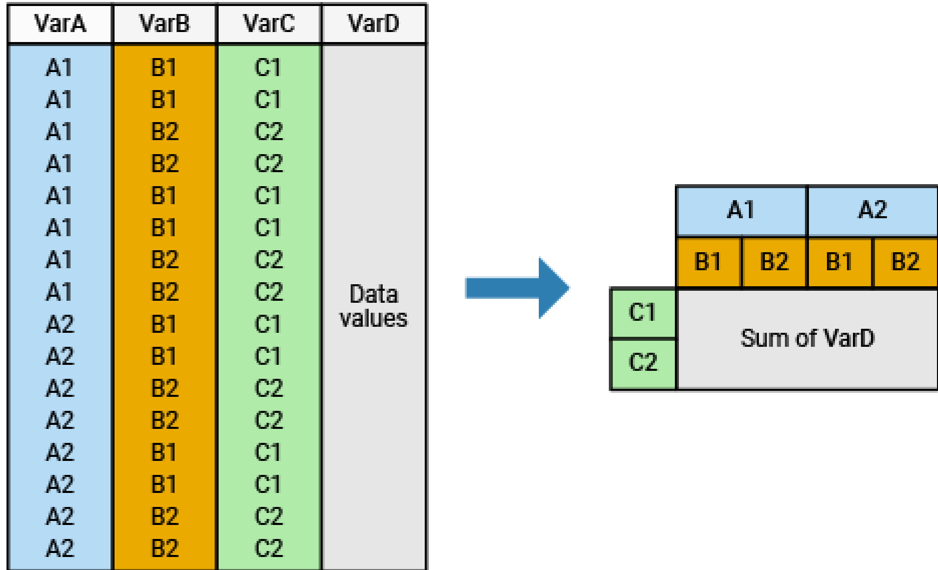


	Cause	GroupCount	sum_Loss	sum_Customers
1	"attack"	294	1.0577e+03	2.5598e+04
2	"earthquake"	2	258.1846	1.3996e+05
3	"energy emergency"	188	5.3983e+04	7.0441e+06
4	"equipment fault"	156	6.9428e+04	1.1546e+07
5	"fire"	25	6.7096e+03	1.6527e+06
6	"severe storm"	338	1.2763e+05	8.1392e+07
7	"thunder storm"	201	8.8754e+04	3.3516e+07
8	"unknown"	24	8.5366e+04	7.5306e+06
9	"wind"	95	1.9524e+04	1.4724e+07

Pivot and Summarize Data Using Multiple Grouping Variables

When you have more than one grouping variable, you can create a pivoted table with columns and rows that correspond to unique combinations of the values in the grouping variables using the `pivot` function. The data values in the pivoted table represent one statistic computed per group for one data variable. A pivoted table has more configuration options than a grouped summary table that you can create using `groupsummary`, and a pivoted table is useful for identifying relationships between groups. Alternatively, you can use the `groupsummary` function to apply more than one computation method or operate on more than one data variable.

In this image, a table with three grouping variables and one data variable is pivoted, and the pivoted table shows the sum of data values in each unique combination of groups.



Apply Two Grouping Criteria

Compute the number of outages for each region per year. In this case, the two grouping variables are `Region` and `OutageTime`. One grouping variable designates the variables of the pivoted table, and one grouping variable designates the rows of the pivoted table. By default, the data values in the pivoted table are the group counts.

```
P1 = pivot(T,Rows="Region",Columns="OutageTime",ColumnsBinMethod="year",RowLabelPlacement="rowna
```

P1=5×13 table

	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011
MidWest	12	10	14	6	16	9	12	11	15	12
NorthEast	5	11	14	18	30	37	49	55	74	89
SouthEast	11	24	34	28	32	22	31	39	48	38
SouthWest	4	4	3	2	5	2	2	3	0	0
West	4	13	14	20	25	21	21	34	40	51

Visualize the pivoted table using a heatmap.

```
xvar = P1.Properties.VariableNames;
yvar = P1.Properties.RowNames;
```

```
cvar = P1.Variables;

figure
heatmap(xvar,yvar,cvar)
title("Outage Count by Region and Year")
fig = gcf;
fig.Position(3) = fig.Position(3) * 2;
```

Outage Count by Region and Year

	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014
MidWest	12	10	14	6	16	9	12	11	15	12	10	15	0
NorthEast	5	11	14	18	30	37	49	55	74	89	97	78	0
SouthEast	11	24	34	28	32	22	31	39	48	38	38	43	1
SouthWest	4	4	3	2	5	2	2	3	0	0	0	1	0
West	4	13	14	20	25	21	21	34	40	51	62	49	0

90 80 70 60 50 40 30 20 10 0

Alternatively, you can apply two grouping criteria using the `groupsummary` function, where groups are defined as unique combinations of the values in the `Region` and `OutageTime` grouping variables.

```
G4 = groupsummary(T,["Region" "OutageTime"],{"none" "year"},"sum",["Loss" "Customers"])
```

```
G4=58×5 table
      Region        year_OutageTime    GroupCount    sum_Loss    sum_Customers
    ___________    _______________    __________    ________    _____________

    "MidWest"            2002              12          41994       5.0288e+06
    "MidWest"            2003              10         8822.4       1.6592e+06
    "MidWest"            2004              14          18207       1.6618e+06
    "MidWest"            2005               6         1505.8       4.0282e+05
    "MidWest"            2006              16         5419.4        5.893e+06
    "MidWest"            2007               9         8778.9       1.2878e+06
    "MidWest"            2008              12         8262.7       5.8309e+06
    "MidWest"            2009              11         1117.5       1.7014e+06
    "MidWest"            2010              15         5551.1        1.276e+06
    "MidWest"            2011              12         364.24       2.6649e+06
    "MidWest"            2012              10         117.18       1.3579e+06
    "MidWest"            2013              15         2251.9       5.3376e+05
    "NorthEast"          2002               5          32734       3.3639e+06
    "NorthEast"          2003              11          30555       2.2939e+06
    "NorthEast"          2004              14         6174.4       8.8251e+05
    "NorthEast"          2005              18         8601.7       2.1882e+06
      ⋮
```

Apply Three Grouping Criteria

Compute the number of outages for each cause per region per number of customers. In this case, the three grouping variables are `Cause`, `Region`, and `Customers`. Define two bins for the `Customers` variable by specifying the `ColumnsBinMethod` name-value argument. Because multiple grouping variables designate the columns of the pivoted table, the pivoted table contains nested tables.

```
P2 = pivot(T,Rows="Cause",Columns=["Region" "Customers"],ColumnsBinMethod={"none",[0 100 Inf]},In
```

```
P2=10×6 table
            Cause                   MidWest                     NortheEast                   SouthEast
    __________________    ________________________    ________________________    _____________________

                          [0, 100)    [100, Inf]      [0, 100)    [100, Inf]      [0, 100)    [100, I
                          ________    __________      ________    __________      ________    _______

    "attack"                 5             0             83            1              6            1
    "earthquake"             0             0              1            0              0            0
    "energy emergency"       7             4              5            7             17           24
    "equipment fault"        2             4              6            9              1           31
    "fire"                   0             0              0            4              0            2
    "severe storm"           0            31              1          141              5          127
    "thunder storm"          0            32              2          100              0           53
    "unknown"                0             4              0            6              1            1
    "wind"                   0            16              0           41              0           13
    "winter storm"           0            17              1           69              3           34
```

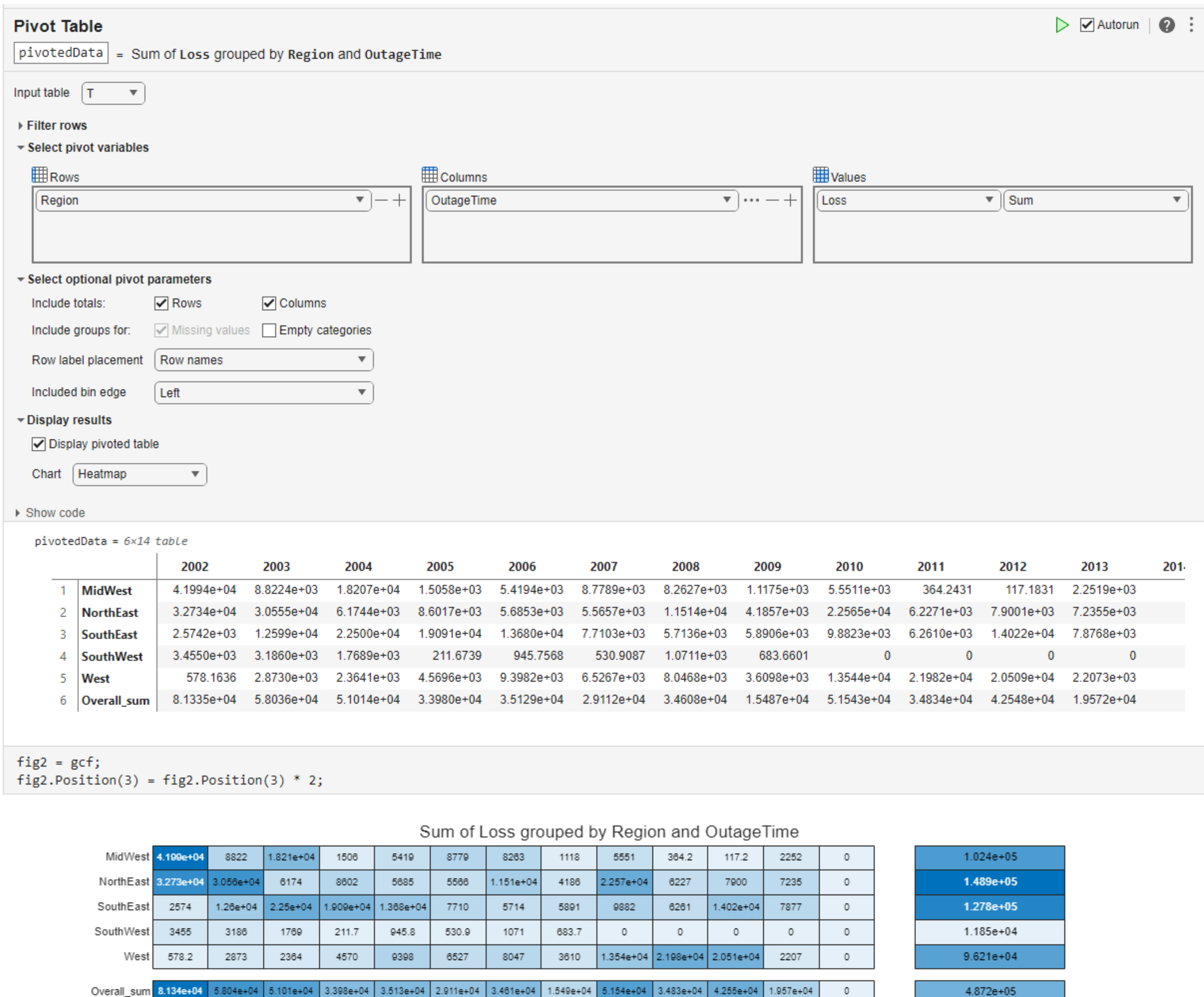
Compute Marginal Totals

You can display row-wise and column-wise statistics in a pivoted table using the `IncludeTotals` name-value argument. Compute the total power loss for each region per year, and include the marginal totals in the pivoted table. The last row of the pivoted table represents the total power loss for each year. The last variable of the pivoted table represents the total power loss for each region.

```
P3 = pivot(T,Rows="Region",Columns="OutageTime",ColumnsBinMethod="year",DataVariable="Loss",RowLa
```

```
P3=6×14 table
                       2002      2003      2004      2005      2006      2007      2008      2009
                      ______    ______    ______    ______    ______    ______    ______    ______

    MidWest            41994    8822.4     18207    1505.8    5419.4    8778.9    8262.7    1117.5
    NorthEast          32734     30555    6174.4    8601.7    5685.3    5565.7     11514    4185.7
    SouthEast         2574.2     12599     22500     19091     13680    7710.3    5713.6    5890.6
    SouthWest           3455      3186    1768.9    211.67    945.76    530.91    1071.1    683.66
    West              578.16      2873    2364.1    4569.6    9398.2    6526.7    8046.8    3609.8
    Overall_sum        81335     58036     51014     33980     35129     29112     34608     15487
```

Alternatively, to interactively summarize tabular data in a pivoted table and visualize the pivoted table in a different types of charts, use the Pivot Table Live Editor task.



pivotedData = *6×14 table*

		2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	201
1	**MidWest**	4.1994e+04	8.8224e+03	1.8207e+04	1.5058e+03	5.4194e+03	8.7789e+03	8.2627e+03	1.1175e+03	5.5511e+03	364.2431	117.1831	2.2519e+03	
2	**NorthEast**	3.2734e+04	3.0555e+04	6.1744e+03	8.6017e+03	5.6853e+03	5.5657e+03	1.1514e+04	4.1857e+03	2.2565e+04	6.2271e+03	7.9001e+03	7.2355e+03	
3	**SouthEast**	2.5742e+03	1.2599e+04	2.2500e+04	1.9091e+04	1.3680e+04	7.7103e+03	5.7136e+03	5.8906e+03	9.8823e+03	6.2610e+03	1.4022e+04	7.8768e+03	
4	**SouthWest**	3.4550e+03	3.1860e+03	1.7689e+03	211.6739	945.7568	530.9087	1.0711e+03	683.6601	0	0	0	0	
5	**West**	578.1636	2.8730e+03	2.3641e+03	4.5696e+03	9.3982e+03	6.5267e+03	8.0468e+03	3.6098e+03	1.3544e+04	2.1982e+04	2.0509e+04	2.2073e+03	
6	**Overall_sum**	8.1335e+04	5.8036e+04	5.1014e+04	3.3980e+04	3.5129e+04	2.9112e+04	3.4608e+04	1.5487e+04	5.1543e+04	3.4834e+04	4.2548e+04	1.9572e+04	

```
fig2 = gcf;
fig2.Position(3) = fig2.Position(3) * 2;
```

Other Functions for Grouped Calculations

In most cases, `groupsummary` is the recommended function for identifying patterns in the data and making comparisons between one or more grouping variables. `pivot` is the recommended function for identifying relationships between multiple grouping variables or when you need additional configuration options.

To explore additional functions for grouped calculations, see the tips and recommendations in "Perform Calculations by Group in Table".

See Also

Functions
`groupsummary` | `pivot` | `bar` | `heatmap`

Live Editor Tasks
Compute by Group | Pivot Table

Related Examples

- “Perform Calculations by Group in Table”
- “Add Interactive Tasks to a Live Script”

Clean Messy Data and Locate Extrema Using Live Editor Tasks

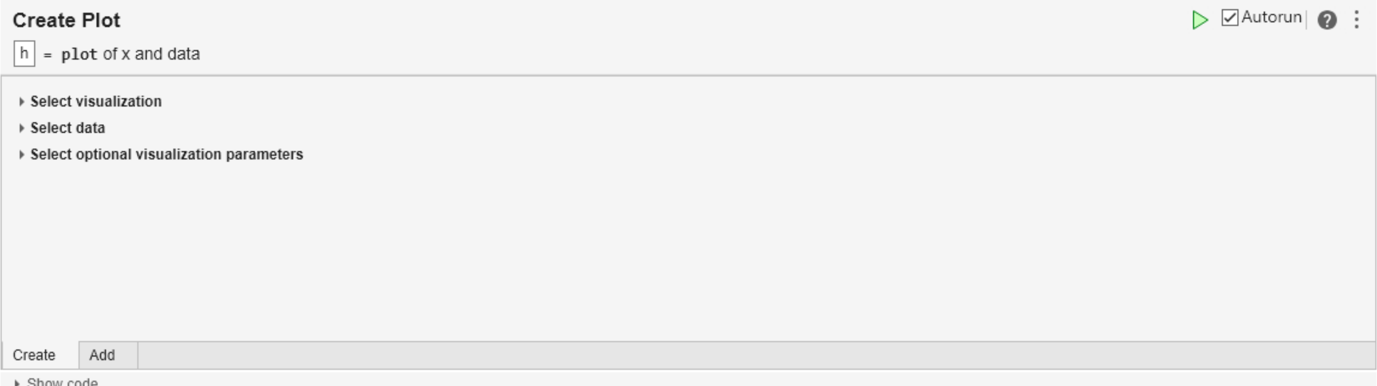
You can interactively preprocess data using sequences of Live Editor tasks, visualizing the data at each step. This example uses five tasks to clean noisy data with missing values and outliers in order to identify local minima and maxima. For more information on Live Editor tasks, see “Add Interactive Tasks to a Live Script”.

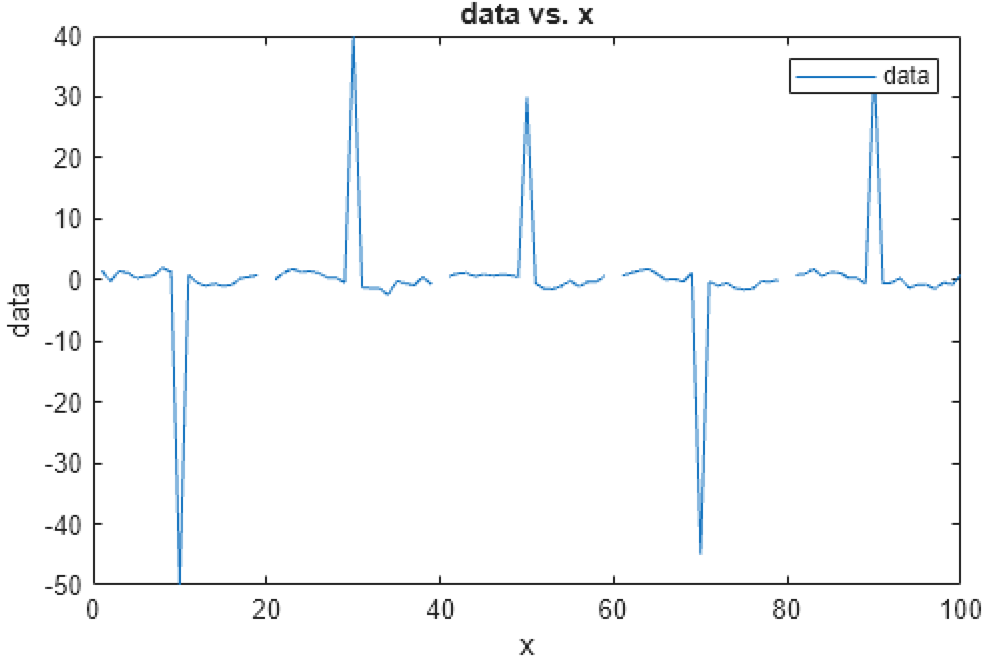
First, create and plot a vector of messy data, which contains four `NaN` values and five outliers.

```
x = 1:100;
data = cos(2*pi*0.05*x+2*pi*rand) + 0.5*randn(1,100);
data(20:20:80) = NaN;
data(10:20:90) = [-50 40 30 -45 35];
```

To plot the messy data, open the **Create Plot** task. Start by typing the keyword `plot` in a code block, and then click `Create Plot` when it appears in the menu. Select the plot type and input data to plot the data.

To see the code that this task generates, expand the task display by clicking at the bottom of the task parameter area.

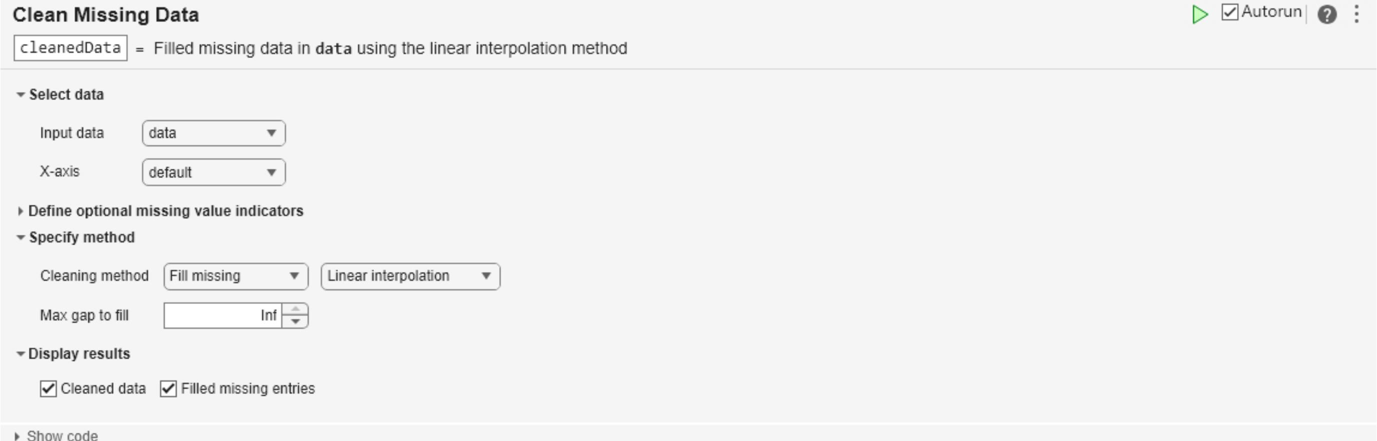


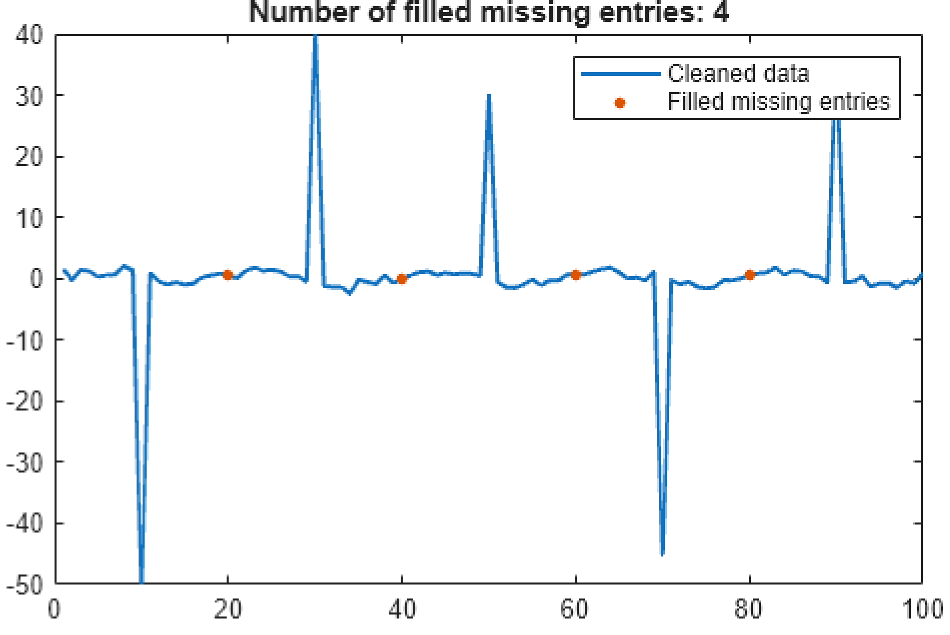


Fill Missing Data

To replace `NaN` values in the data and visualize the results, open the **Clean Missing Data** task. Start by typing the keyword `missing` in a code block, and then click `Clean Missing Data` when it appears in the menu. Select the input data and the cleaning method to plot the filled data automatically.

To see the code that this task generates, expand the task display by clicking ▼ at the bottom of the task parameter area.

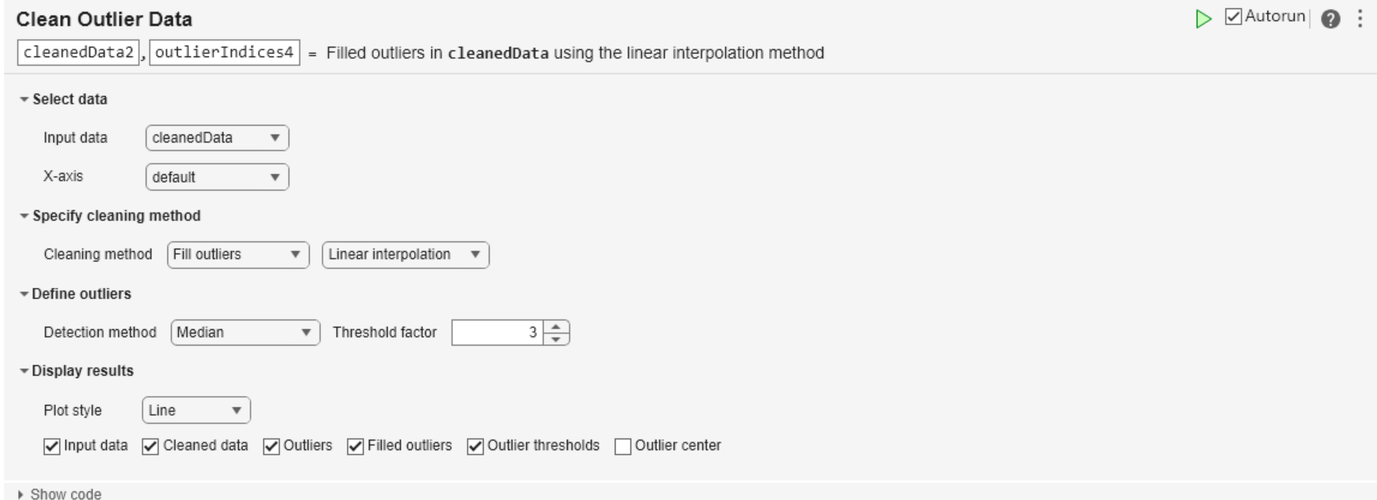


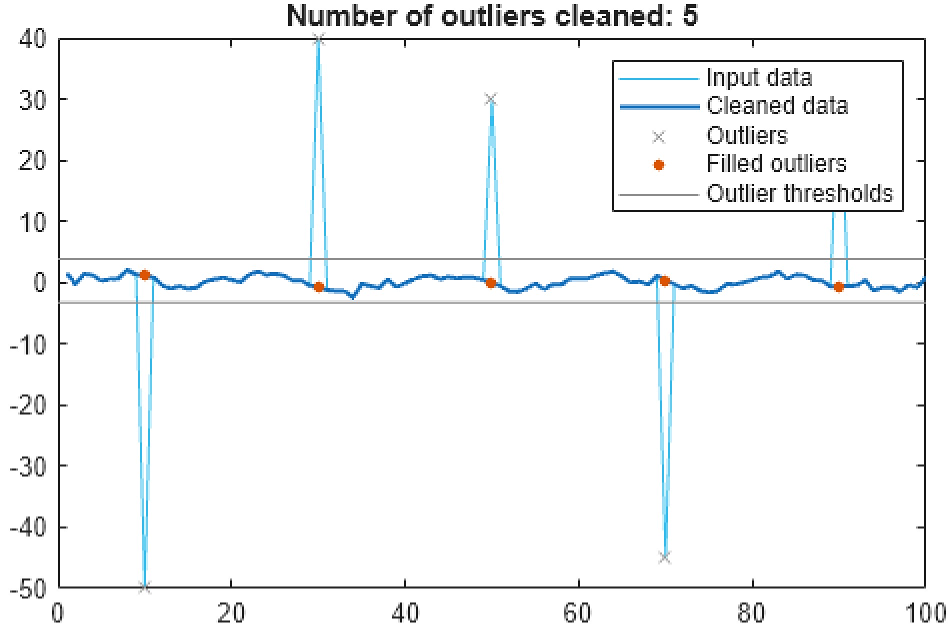


Fill Outliers

You can now remove the outliers from the cleaned data in the previous task by using the **Clean Outlier Data** task. Type the keyword `outliers` in a new code block and click `Clean Outlier Data` to open the task. Select `cleanedData` as the input data. You can customize the methods for cleaning and detecting outliers and adjust the threshold to find more or fewer outliers.

To see the code that this task generates, expand the task display by clicking ▼ at the bottom of the task parameter area.

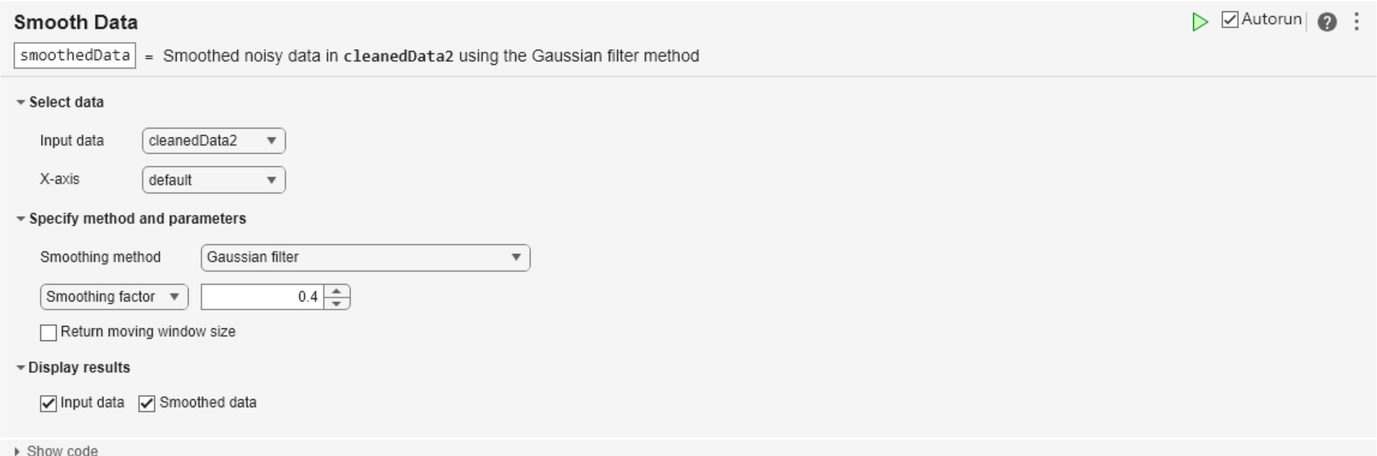


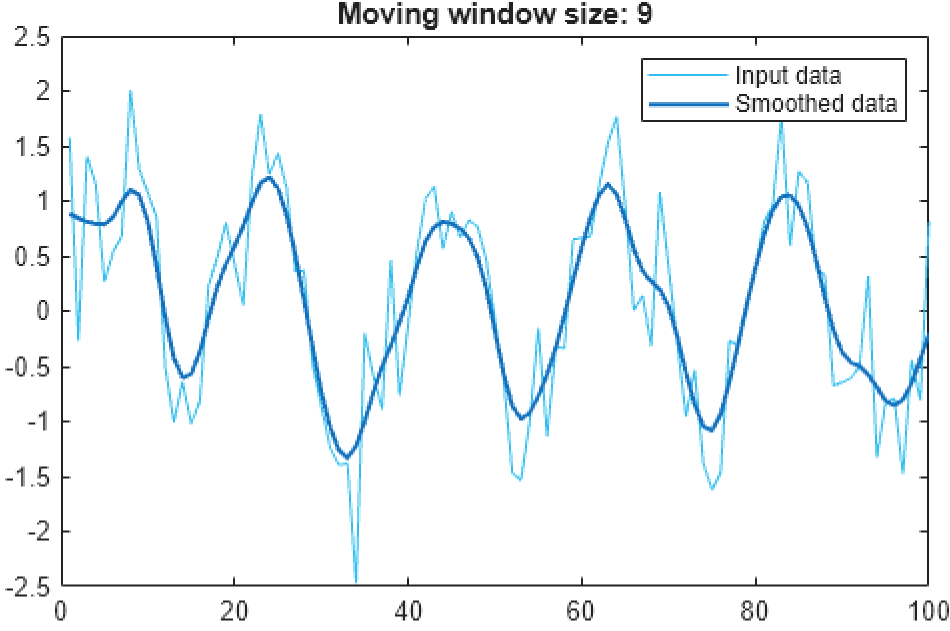


Smooth Data

Next, smooth the cleaned data from the previous task by using the **Smooth Data** task. Type the keyword `smooth` and click the task when it appears. Select `cleanedData2`, the output from the previous task, as the input data. Select a smoothing method, and adjust the smoothing factor for more or less smoothing.

To see the code that this task generates, expand the task display by clicking ▼ at the bottom of the task parameter area.

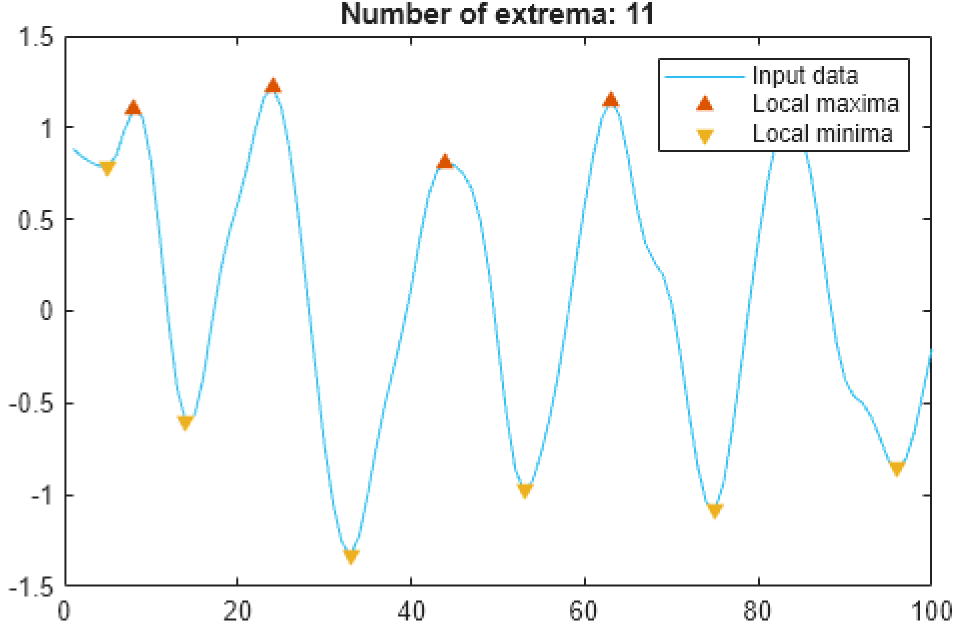




Locate Extrema

Finally, start typing the keyword `extrema` and click `Find Local Extrema`. Use `smoothedData` as the input data and change the extrema type to find both the local maxima and local minima of the cleaned, smoothed data. You can adjust the local extrema parameters to find more or fewer maxima and minima.

To see the code that this task generates, expand the task display by clicking ▼ at the bottom of the task parameter area.



See Also

Live Editor Tasks
Clean Missing Data | Clean Outlier Data | Find Change Points | Find Local Extrema | Smooth Data | Find and Remove Trends

Functions
`ismissing` | `rmmissing` | `fillmissing` | `isoutlier` | `filloutliers` | `rmoutliers` | `ischange` | `islocalmin` | `islocalmax` | `smoothdata`

Related Examples

- "Add Interactive Tasks to a Live Script"
- "Data Smoothing and Outlier Detection" on page 1-12
- "Missing Data in MATLAB" on page 1-8

Filter Data

Filter Difference Equation

Filters are data processing techniques that can smooth out high-frequency fluctuations in data or remove periodic trends of a specific frequency from data. In MATLAB, the `filter` function filters a vector of data x according to the following difference equation, which describes a tapped delay-line filter.

$$a(1)y(n) = b(1)x(n) + b(2)x(n-1) + \ldots + b(N_b)x(n - N_b + 1) \\ -a(2)y(n-1) - \ldots - a(N_a)y(n - N_a + 1)$$

In this equation, a and b are vectors of coefficients of the filter, N_a is the feedback filter order, and N_b is the feedforward filter order. n is the index of the current element of x. The output $y(n)$ is a linear combination of the current and previous elements of x and y.

The `filter` function uses specified coefficient vectors a and b to filter the input data x. For more information on difference equations describing filters, see [1].

Moving-Average Filter of Traffic Data

The `filter` function is one way to implement a moving-average filter, which is a common data smoothing technique.

The following difference equation describes a filter that averages time-dependent data with respect to the current hour and the three previous hours of data.

$$y(n) = \frac{1}{4}x(n) + \frac{1}{4}x(n-1) + \frac{1}{4}x(n-2) + \frac{1}{4}x(n-3)$$

Import data that describes traffic flow over time, and assign the first column of vehicle counts to the vector `x`.

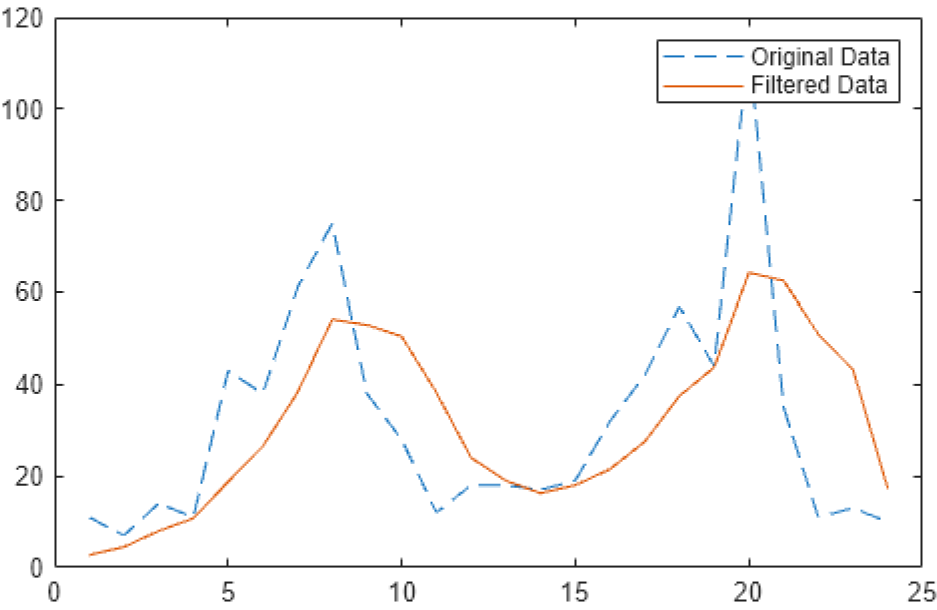
```
load count.dat
x = count(:,1);
```

Create the filter coefficient vectors.

```
a = 1;
b = [1/4 1/4 1/4 1/4];
```

Compute the 4-hour moving average of the data, and plot both the original data and the filtered data.

```
y = filter(b,a,x);

t = 1:length(x);
plot(t,x,'--',t,y,'-')
legend('Original Data','Filtered Data')
```



Modify Amplitude of Data

This example shows how to modify the amplitude of a vector of data by applying a transfer function.

In digital signal processing, filters are often represented by a transfer function. The Z-transform of the difference equation

$$\begin{aligned} a(1)y(n) &= b(1)x(n) + b(2)x(n-1) + \ldots + b(N_b)x(n - N_b + 1) \\ &\quad - a(2)y(n-1) - \ldots - a(N_a)y(n - N_a + 1) \end{aligned}$$

is the following transfer function.

$$Y(z) = H(z^{-1})X(z) = \frac{b(1) + b(2)z^{-1} + \ldots + b(N_b)z^{-N_b + 1}}{a(1) + a(2)z^{-1} + \ldots + a(N_a)z^{-N_a + 1}}X(z)$$

Use the transfer function

$$H(z^{-1}) = \frac{b(z^{-1})}{a(z^{-1})} = \frac{2 + 3z^{-1}}{1 + 0.2z^{-1}}$$

to modify the amplitude of the data in `count.dat`.

Load the data and assign the first column to the vector `x`.

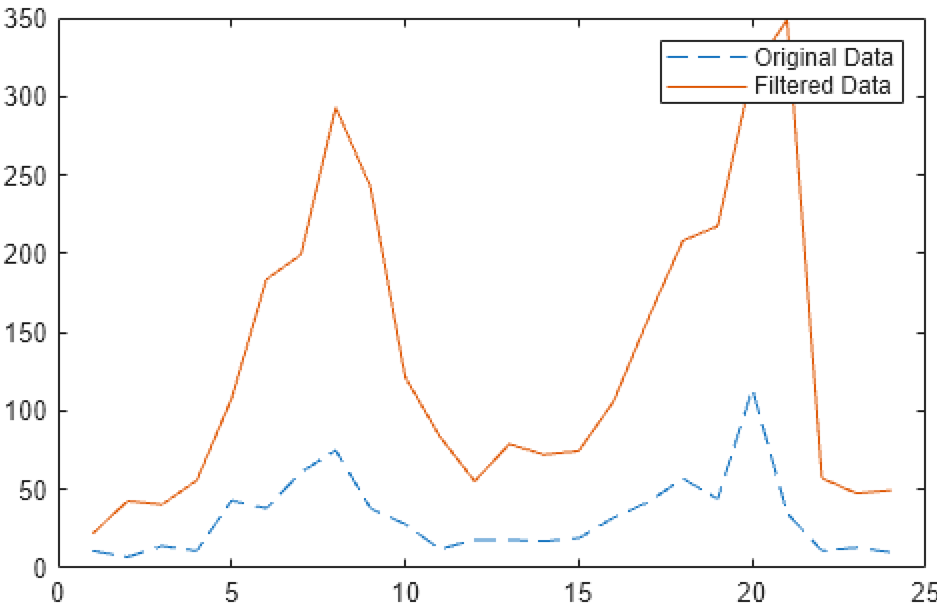
```
load count.dat
x = count(:,1);
```

Create the filter coefficient vectors according to the transfer function $H(z^{-1})$.

```
a = [1 0.2];
b = [2 3];
```

Compute the filtered data, and plot both the original data and the filtered data. This filter primarily modifies the amplitude of the original data.

```
y = filter(b,a,x);

t = 1:length(x);
plot(t,x,'--',t,y,'-')
legend('Original Data','Filtered Data')
```



References

[1] Oppenheim, Alan V., Ronald W. Schafer, and John R. Buck. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1999.

See Also

`filter` | `conv` | `filter2` | `smoothdata` | `movmean`

Related Examples

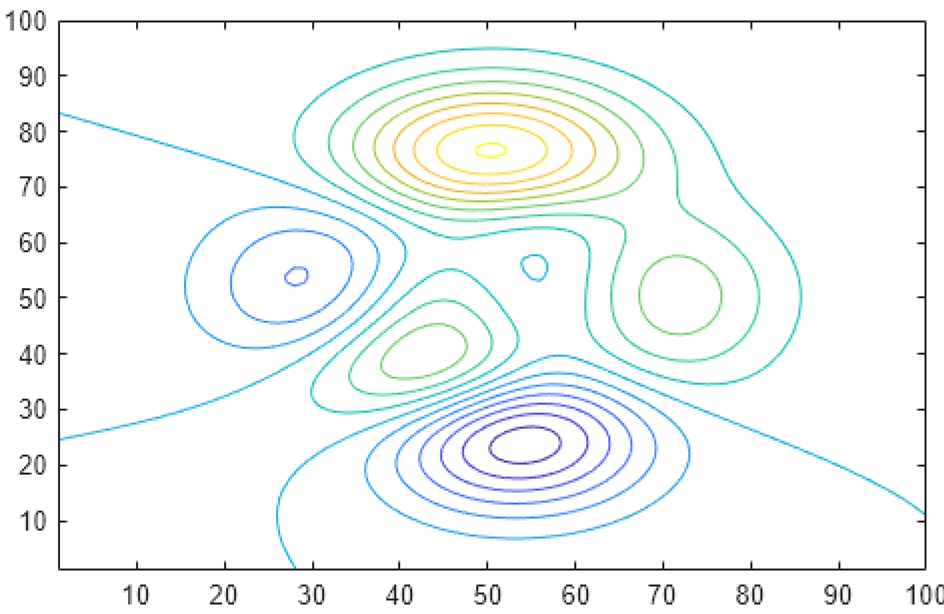
- “Smooth Data with Convolution” on page 1-42

Smooth Data with Convolution

You can use convolution to smooth 2-D data that contains high-frequency components.

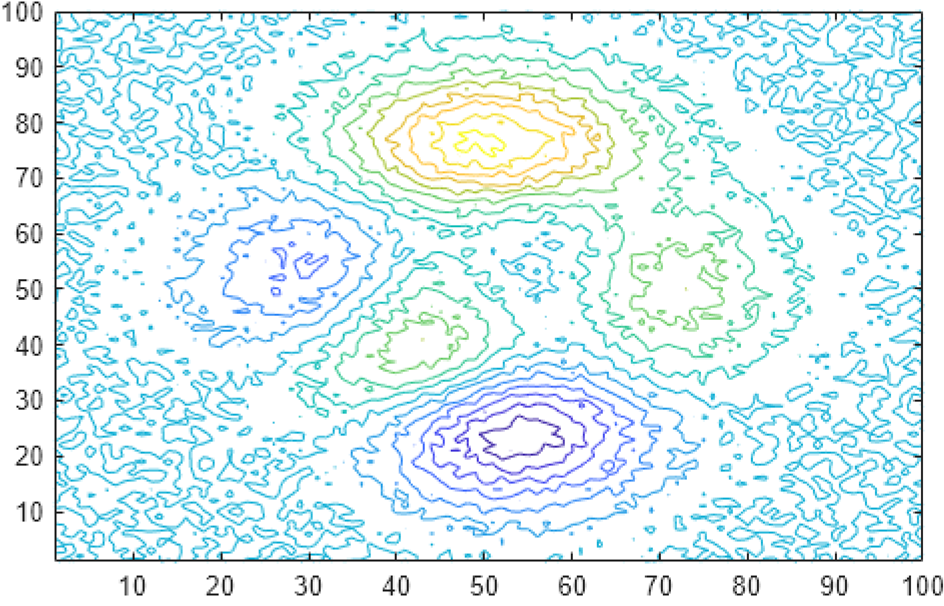
Create 2-D data using the `peaks` function, and plot the data at various contour levels.

```
Z = peaks(100);
levels = -7:1:10;
contour(Z,levels)
```



Inject random noise into the data and plot the noisy contours.

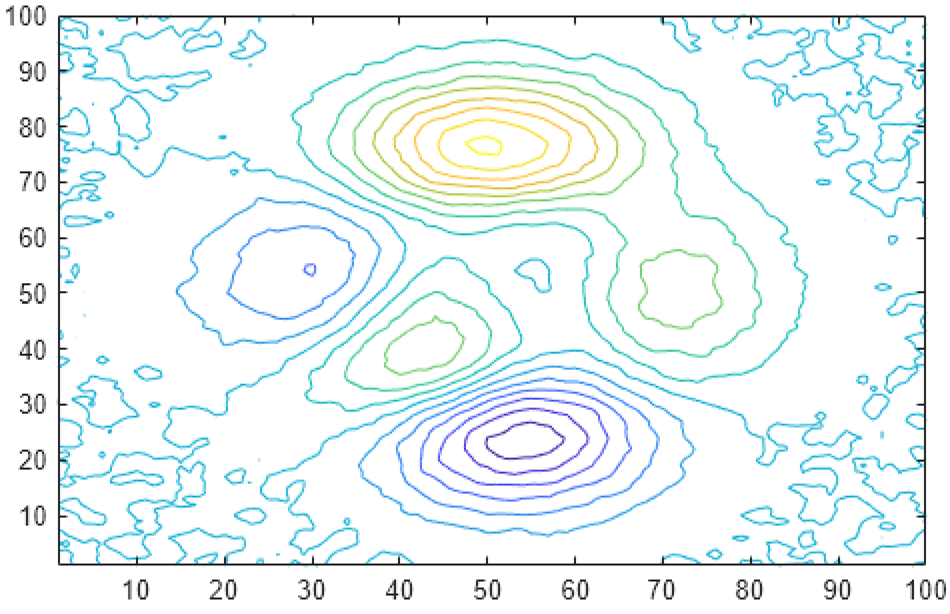
```
Znoise = Z + rand(100) - 0.5;
contour(Znoise,levels)
```



The conv2 function in MATLAB® convolves 2-D data with a specified kernel whose elements define how to remove or enhance features of the original data. Kernels do not have to be the same size as the input data. Small-sized kernels can be sufficient to smooth data containing only a few frequency components. Larger sized kernels can provide more precision for tuning frequency response, resulting in smoother output.

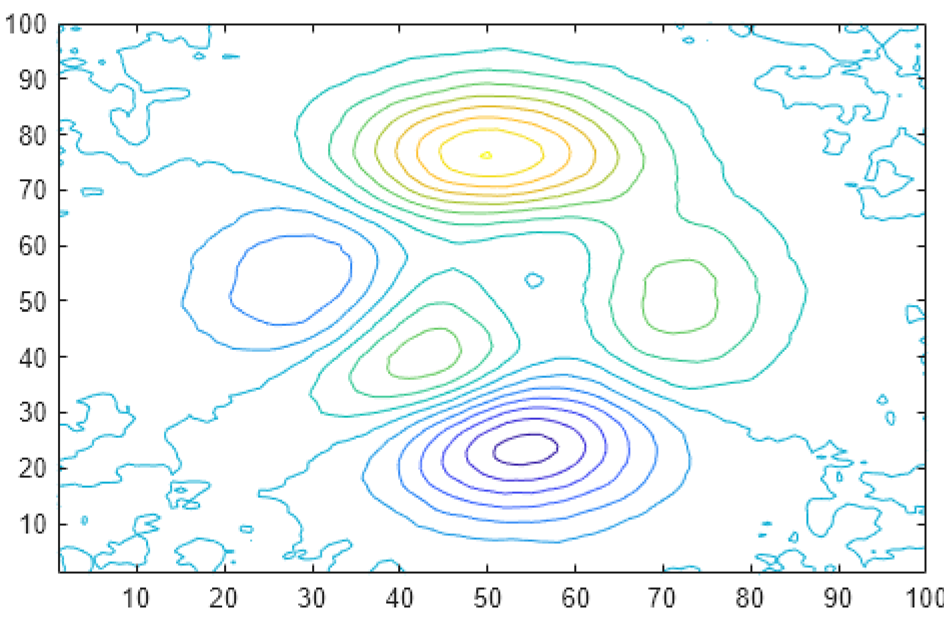
Define a 3-by-3 kernel K and use conv2 to smooth the noisy data in Znoise. Plot the smoothed contours. The 'same' option in conv2 makes the output the same size as the input.

```
K = (1/9)*ones(3);
Zsmooth1 = conv2(Znoise,K,'same');
contour(Zsmooth1, levels)
```



Smooth the noisy data with a 5-by-5 kernel, and plot the new contours.

```
K = (1/25)*ones(5);
Zsmooth2 = conv2(Znoise,K,'same');
contour(Zsmooth2,levels)
```



See Also

`conv2` | `conv` | `filter` | `smoothdata`

Related Examples

- “Filter Data” on page 1-39

Computing with Descriptive Statistics

In this section...
“Functions for Calculating Descriptive Statistics” on page 1-46
“Example: Using MATLAB Data Statistics” on page 1-47
“Data Statistics” on page 1-48

If you need more advanced statistics features, you might want to use the Statistics and Machine Learning Toolbox™ software.

Functions for Calculating Descriptive Statistics

Use the following MATLAB functions to calculate the descriptive statistics for your data.

Note For matrix data, descriptive statistics for each column are calculated independently.

Statistics Function Summary

Function	Description
`max`	Maximum value
`mean`	Average or mean value
`median`	Median value
`min`	Smallest value
`mode`	Most frequent value
`std`	Standard deviation
`var`	Variance, which measures the spread or dispersion of the values

The following examples apply MATLAB functions to calculate descriptive statistics:

- “Example 1 — Calculating Maximum, Mean, and Standard Deviation” on page 1-46
- “Example 2 — Subtracting the Mean” on page 1-47

Example 1 — Calculating Maximum, Mean, and Standard Deviation

This example shows how to use MATLAB functions to calculate the maximum, mean, and standard deviation values for a 24-by-3 matrix called `count`. MATLAB computes these statistics independently for each column in the matrix.

```
% Load the sample data
load count.dat
% Find the maximum value in each column
mx = max(count)
% Calculate the mean of each column
mu = mean(count)
% Calculate the standard deviation of each column
sigma = std(count)
```

The results are

```
mx =
          114          145          257

mu =
      32.0000      46.5417      65.5833

sigma =
      25.3703      41.4057      68.0281
```

To get the row numbers where the maximum data values occur in each data column, specify a second output parameter `indx` to return the row index. For example:

```
[mx,indx] = max(count)
```

These results are

```
mx =
      114     145     257

indx =
      20    20    20
```

Here, the variable `mx` is a row vector that contains the maximum value in each of the three data columns. The variable `indx` contains the row indices in each column that correspond to the maximum values.

To find the minimum value in the entire `count` matrix, 24-by-3 matrix into a 72-by-1 column vector by using the syntax `count(:)`. Then, to find the minimum value in the single column, use the following syntax:

```
min(count(:))

ans =
      7
```

Example 2 — Subtracting the Mean

Subtract the mean from each column of the matrix by using the following syntax:

```
% Get the size of the count matrix
[n,p] = size(count)
% Compute the mean of each column
mu = mean(count)
% Create a matrix of mean values by
% replicating the mu vector for n rows
MeanMat = repmat(mu,n,1)
% Subtract the column mean from each element
% in that column
x = count - MeanMat
```

Note Subtracting the mean from the data is also called *detrending*. For more information about removing the mean or the best-fit line from the data, see “Remove Linear Trends from Timetable Data” on page 1-5.

Example: Using MATLAB Data Statistics

Data Statistics

The Data Statistics dialog box helps you calculate and plot descriptive statistics with the data. This example shows how to use MATLAB Data Statistics to calculate and plot statistics for a 24-by-3 matrix, called `count`. The data represents how many vehicles passed by traffic counting stations on three streets.

This section contains the following topics:

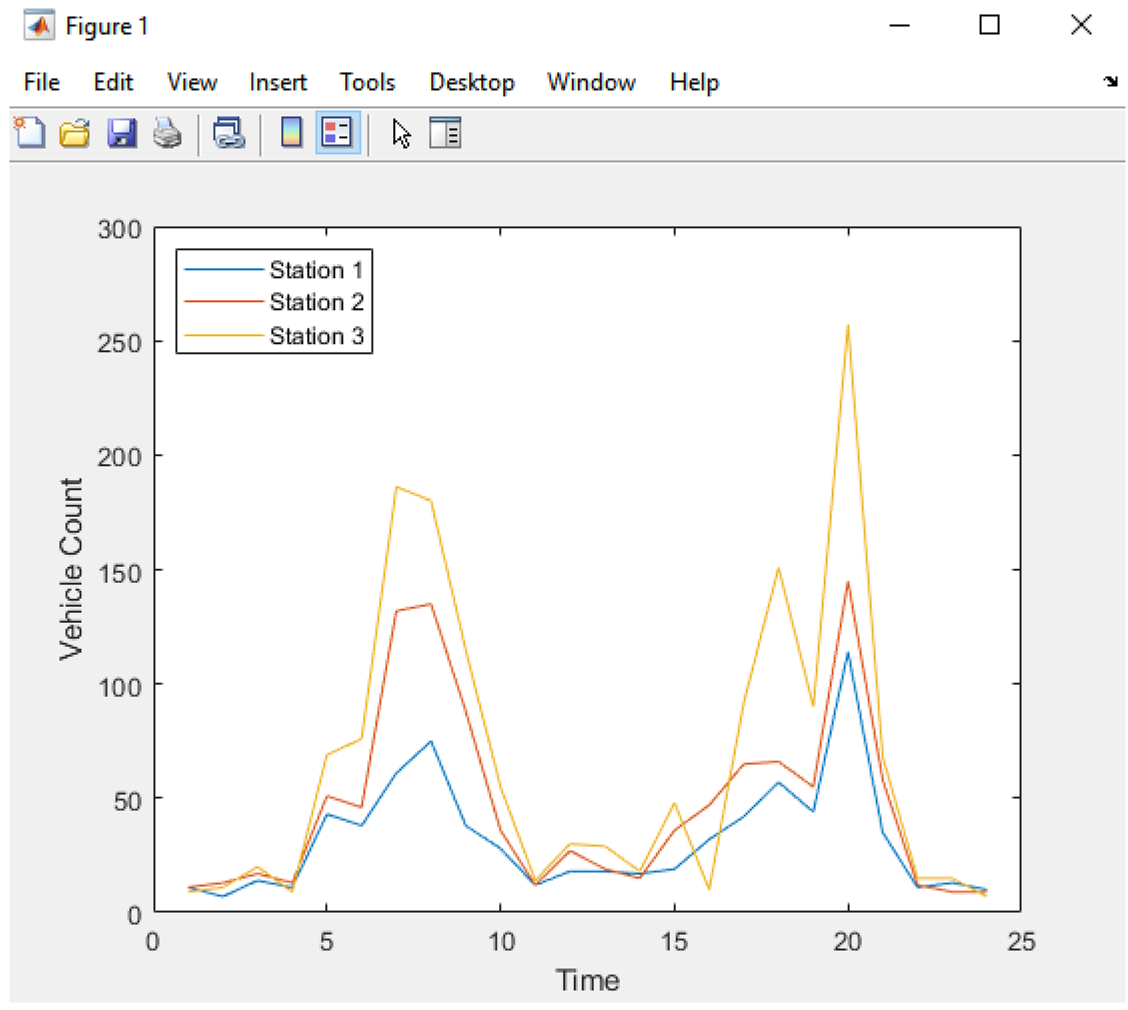
- “Calculating and Plotting Descriptive Statistics” on page 1-48
- “Formatting Data Statistics on Plots” on page 1-51
- “Saving Statistics to the MATLAB Workspace” on page 1-52
- “Generating Code Files” on page 1-53

Note MATLAB Data Statistics is available for 2-D plots only.

Calculating and Plotting Descriptive Statistics

1 Load and plot the data:

```
load count.dat
[n,p] = size(count);

% Define the x-values
t = 1:n;

% Plot the data and annotate the graph
plot(t,count)
legend('Station 1','Station 2','Station 3','Location','northwest')
xlabel('Time')
ylabel('Vehicle Count')
```



Note The legend contains the name of each data set, as specified by the `legend` function: `Station 1`, `Station 2`, and `Station 3`. A *data set* refers to each column of data in the array you plotted. If you do not name the data sets, default names are assigned: `data1`, `data2`, and so on.

2 In the Figure window, select **Tools > Data Statistics**.

The Data Statistics dialog box opens and displays descriptive statistics for the X- and Y-data of the `Station 1` data set.

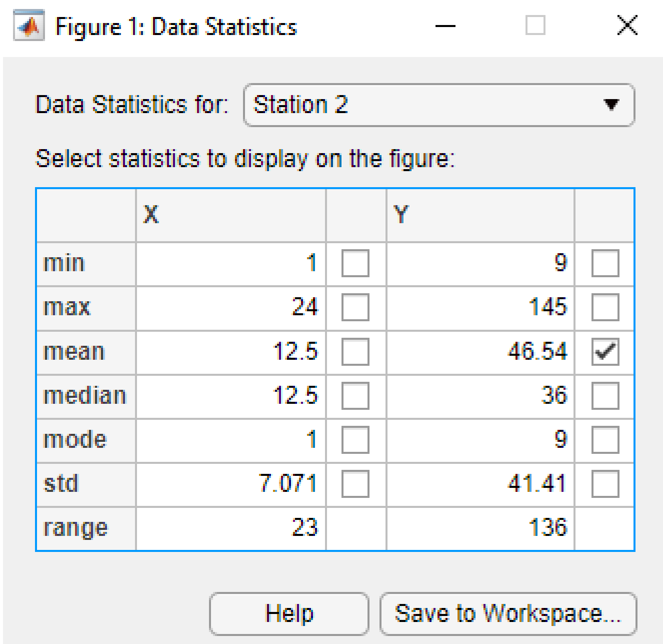
Note The Data Statistics dialog box displays a *range*, which is the difference between the minimum and maximum values in the selected data set. The dialog box does not display the range on the plot.

3 Select a different data set in the **Data Statistics for** list: `Station 2`.

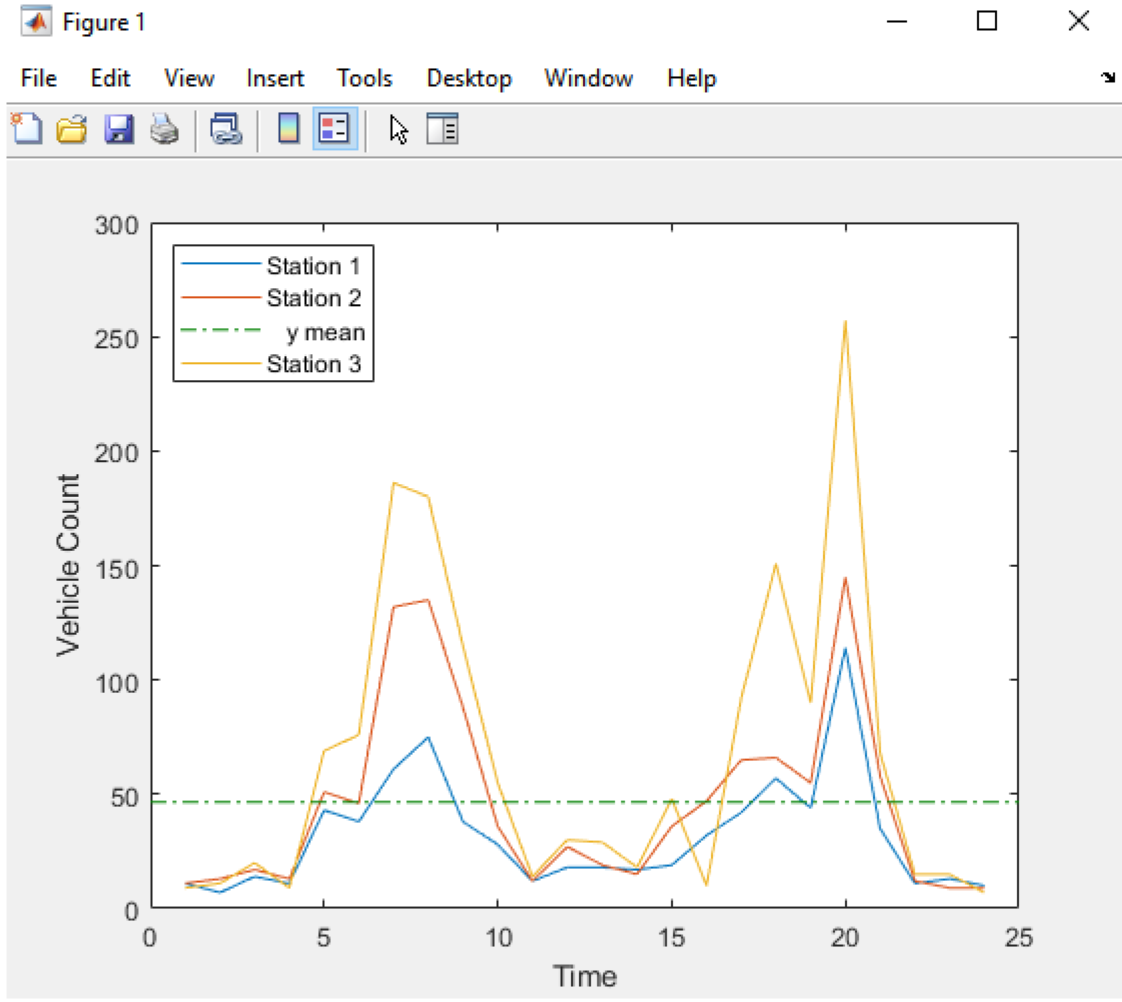
This displays the statistics for the X and Y data of the `Station 2` data set.

4 Select the check box for each statistic you want to display on the plot, and then click **Save to Workspace**.

For example, to plot the mean of `Station 2`, select the **mean** check box in the **Y** column.



This plots a horizontal line to represent the mean of `Station 2` and updates the legend to include this statistic.



Formatting Data Statistics on Plots

The Data Statistics dialog box uses colors and line styles to distinguish statistics from the data on the plot. This portion of the example shows how to customize the display of descriptive statistics on a plot, such as the color, line width, line style, or marker.

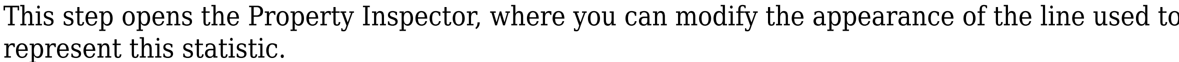
Note Do not edit display properties of statistics until you finish plotting all the statistics with the data. If you add or remove statistics after editing plot properties, the changes to plot properties are lost.

To modify the display of data statistics on a plot:

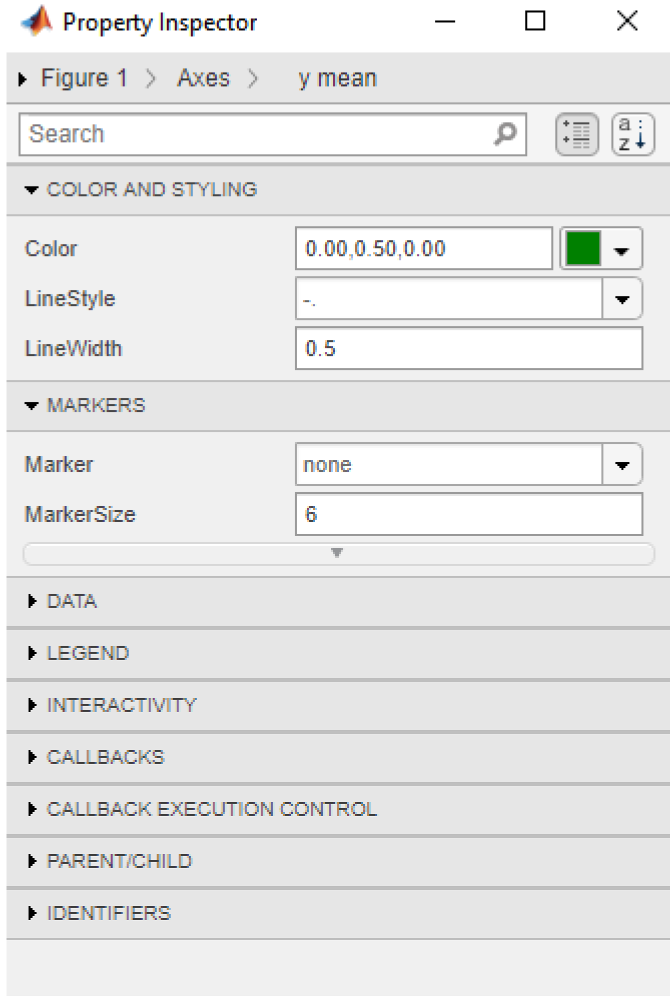
1 In the MATLAB Figure window, click the (**Edit Plot**) button in the toolbar.

   This step enables plot editing.

2 Double-click the statistic on the plot for which you want to edit display properties. For example, double-click the horizontal line representing the mean of `Station 2`.



This step opens the Property Inspector, where you can modify the appearance of the line used to represent this statistic.



3. In the Property Inspector window, specify the line and marker styles, sizes, and colors.

Tip Alternatively, right-click the statistic on the plot, and select an option from the shortcut menu.

Saving Statistics to the MATLAB Workspace

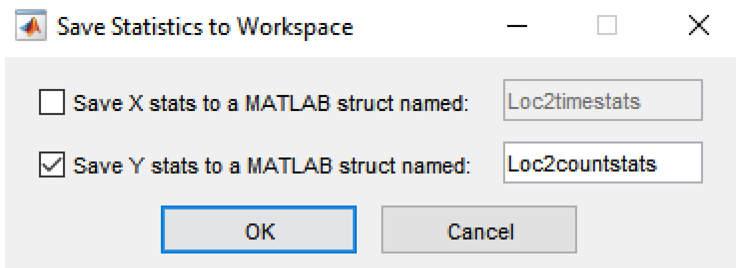
Perform these steps to save the statistics to the MATLAB workspace.

Note When your plot contains multiple data sets, save statistics for each data set individually. To display statistics for a different data set, select it from the **Data Statistics for** list in the Data Statistics dialog box.

---

1. In the Data Statistics dialog box, click the **Save to Workspace** button.
2. In the Save Statistics to Workspace dialog box, select options to save statistics for either X data, Y data, or both. Then, enter the corresponding variable names.

In this example, save only the Y data. Enter the variable name as `Loc2countstats`.



3 Click **OK**.

This step saves the descriptive statistics to a structure. The new variable is added to the MATLAB workspace.

To view the new structure variable, type the variable name at the MATLAB prompt:

```
Loc2countstats
```

```
Loc2countstats =

  struct with fields:

       min: 9
       max: 145
      mean: 46.5417
    median: 36
      mode: 9
       std: 41.4057
     range: 136
```

Generating Code Files

This portion of the example shows how to generate a file containing MATLAB code that reproduces the format of the plot and the plotted statistics with new data. Generating a code file is not available in MATLAB Online™.

1 In the Figure window, select **File > Generate Code**.

This step creates a function code file and displays it in the MATLAB Editor.

2 Change the name of the function on the first line of the file from `createfigure` to something more specific, like `countplot`. Save the file to your current folder with the file name `countplot.m`.

3 Generate some new, random count data:

```
rng('default')
randcount = 300*rand(24,3);
```

4 Reproduce the plot with the new data and the recomputed statistics:

```
countplot(t,randcount)
```

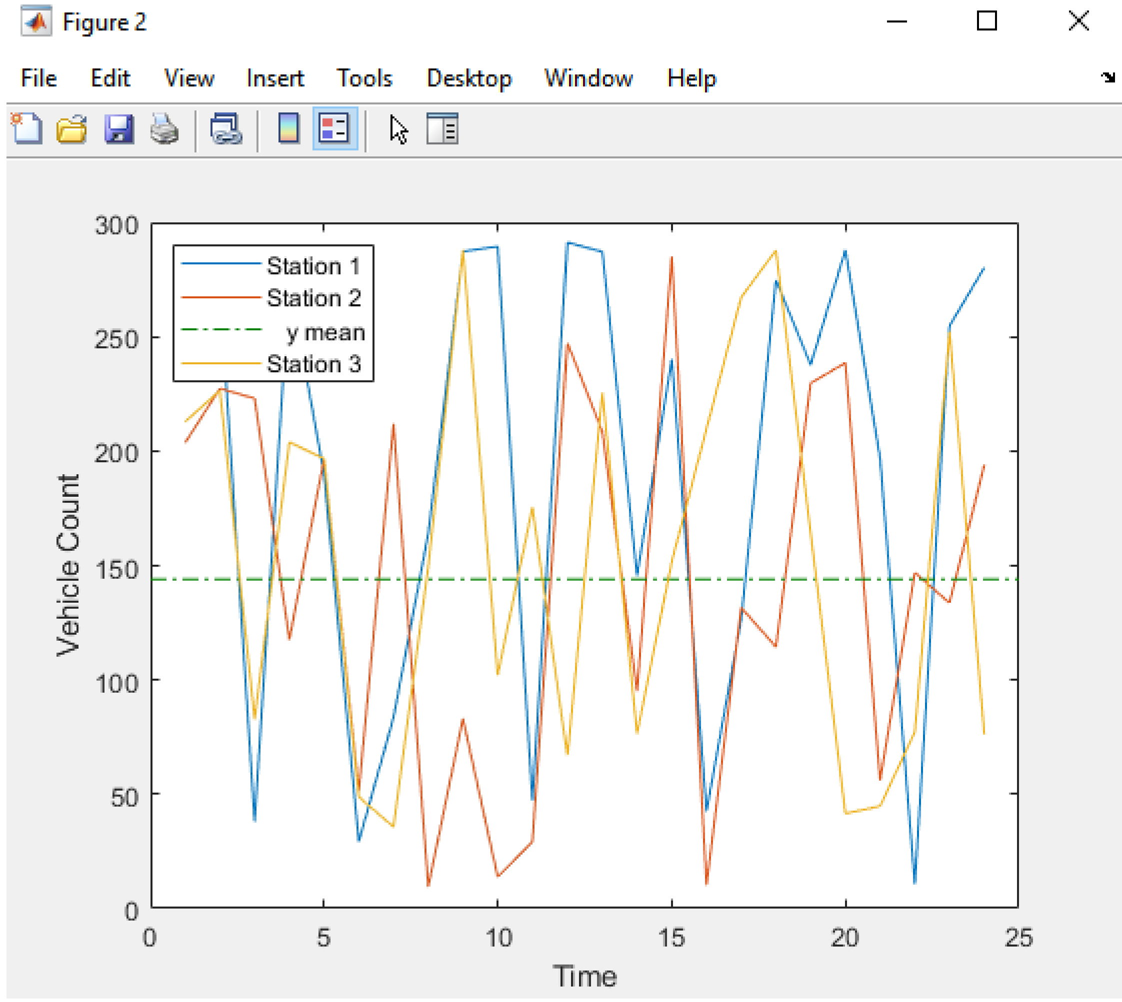


Figure 2
File
Edit
View
Insert
Tools
Desktop
Window
Help
Station 1
Station 2
y mean
Station 3
Vehicle Count
Time
300
250
200
150
100
50
0
0
5
10
15
20
25

Identify and Visualize Correlated Variables

When analyzing the relationships between data variables, you can identify and visualize correlated variables to gain insights into the data set. This example shows how to determine the strength and direction of relationships by using the `corrcoef` function to calculate correlation coefficients. The correlation coefficients range from –1 to 1, where:

- Values close to 1 indicate a positive linear relationship between the data variables.
- Values close to –1 indicate a negative linear relationship between the data variables (*anticorrelation*).
- Values close to or equal to 0 suggest no linear relationship between the data variables.

Additionally, this example shows how to determine which correlations are significant and identify the most correlated pairs. The example plots correlations using a heatmap and bar chart, so you can visually compare the relationships between variables.

Compute Correlation Coefficients and P-Values

Generate random data with correlations among variables. Then, use the `corrcoef` function to calculate the correlation coefficients and corresponding p-values that describe the significance of the correlations.

```
rng(16)
A = randn(50,6);
A(:,3) = A(:,3) + 2*A(:,1);
A(:,4) = A(:,4) - 3*A(:,2);
A(:,5) = A(:,6) + 0.1*A(:,4) - 0.1*A(:,3);
[R,P] = corrcoef(A)
```

```
R = 6×6

    1.0000   -0.1248    0.8047    0.1300   -0.1726   -0.1009
   -0.1248    1.0000   -0.0744   -0.9612   -0.2314    0.0730
    0.8047   -0.0744    1.0000    0.0709   -0.3595   -0.2515
    0.1300   -0.9612    0.0709    1.0000    0.2611   -0.0549
   -0.1726   -0.2314   -0.3595    0.2611    1.0000    0.9384
   -0.1009    0.0730   -0.2515   -0.0549    0.9384    1.0000


P = 6×6

    1.0000    0.3878    0.0000    0.3683    0.2308    0.4858
    0.3878    1.0000    0.6075    0.0000    0.1060    0.6143
    0.0000    0.6075    1.0000    0.6248    0.0103    0.0781
    0.3683    0.0000    0.6248    1.0000    0.0670    0.7051
    0.2308    0.1060    0.0103    0.0670    1.0000    0.0000
    0.4858    0.6143    0.0781    0.7051    0.0000    1.0000
```

The returned matrices `R` and `P`, which contain the correlation coefficients and the p-values respectively, are symmetric. Extract the lower triangular part of `R` to focus on unique pairwise correlations.

```
R = tril(R,-1)
```

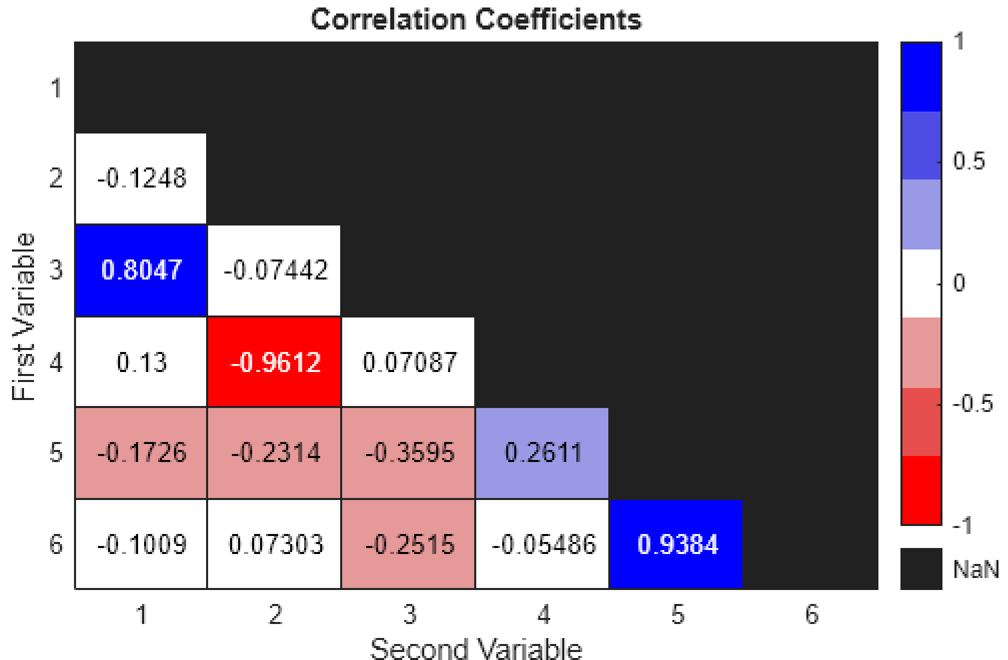
```
R = 6×6

         0         0         0         0         0         0
   -0.1248         0         0         0         0         0
    0.8047   -0.0744         0         0         0         0
    0.1300   -0.9612    0.0709         0         0         0
   -0.1726   -0.2314   -0.3595    0.2611         0         0
   -0.1009    0.0730   -0.2515   -0.0549    0.9384         0
```

Visualize Correlations Using Heatmap

Create a heatmap of the correlation coefficients to visualize the strength and direction of relationships between variables.

Convert the zeros in the correlation coefficient matrix, which mirror the redundant elements of the lower triangle correlations, into missing values (NaN). Next, create a colormap where variable pairs with negative correlations are in red, pairs with positive correlations are in blue, and pairs with no correlation are in white. The heatmap highlights the strongest correlations in bright red and bright blue.

```
Rheatmap = standardizeMissing(R,0);
map = [1 0 0;
    0.9 0.3 0.3;
    0.9 0.6 0.6;
    1 1 1;
    0.6 0.6 0.9;
    0.3 0.3 0.9;
    0 0 1];
h = heatmap(Rheatmap,Colormap=map,ColorLimits=[-1 1]);
h.Title = "Correlation Coefficients";
h.YLabel = "First Variable";
h.XLabel = "Second Variable";
```



Determine Significant Correlations

Identify significant correlations by filtering out those with a p-value greater than 0.05. This approach focuses the analysis only on relationships that are statistically meaningful and avoids interpreting random noise as a correlation.

```
threshold = 0.05;
R(abs(P) > threshold) = 0;
[firstVar,secondVar,corrCoef] = find(R)
```

```
firstVar = 4×1

     3
     4
     5
     6


secondVar = 4×1

     1
     2
     3
     5


corrCoef = 4×1

    0.8047
   -0.9612
   -0.3595
    0.9384
```

Display Correlations in Table

Compile significant correlations in a table, including indices, correlation coefficients, and p-values.

```
ind2 = sub2ind(size(P),firstVar,secondVar);
sigP = P(ind2);
TSig = table(firstVar,secondVar,corrCoef,sigP)
```

```
TSig=4×4 table
    firstVar    secondVar    corrCoef       sigP
    ________    _________    ________    __________

       3            1         0.80471    1.8997e-12
       4            2        -0.96118    1.7065e-28
       5            3        -0.35949      0.010346
       6            5          0.9384    8.5819e-24
```

List the top three correlations by magnitude. The top correlations are consistent with the relationships established in the input data.

```
A(:,3) = A(:,3) + 2*A(:,1);
A(:,4) = A(:,4) - 3*A(:,2);
A(:,5) = A(:,6) + 0.1*A(:,4) - 0.1*A(:,3);
```

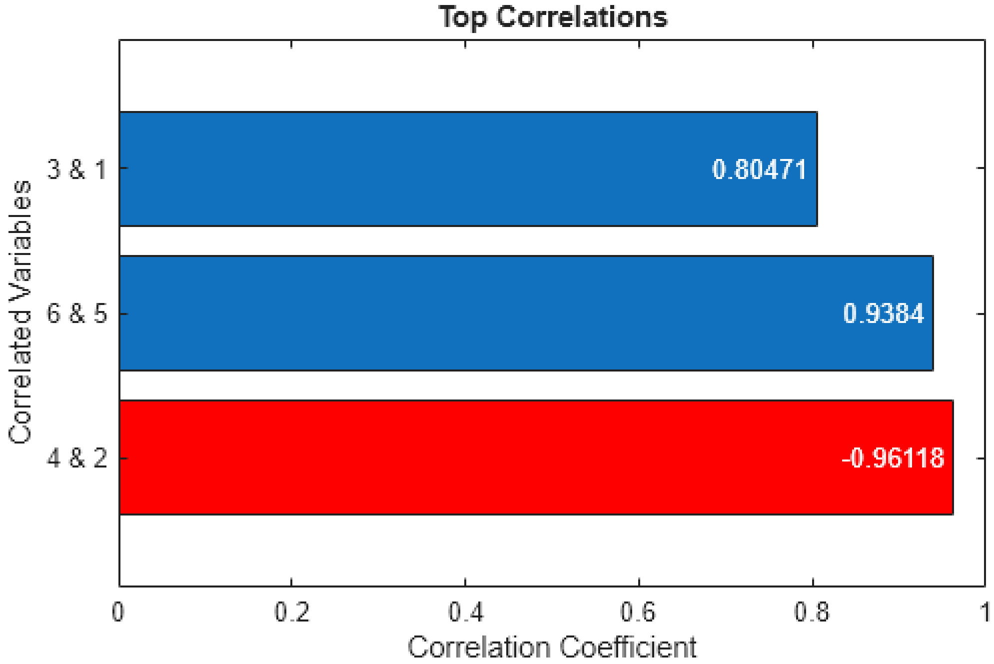
```
k = 3;
TTopk = topkrows(TSig,k,"corrCoef","descend",ComparisonMethod="abs")
```

```
TTopk=3×4 table
    firstVar    secondVar    corrCoef       sigP
    ________    _________    ________    __________

       4            2        -0.96118    1.7065e-28
       6            5          0.9384    8.5819e-24
       3            1         0.80471    1.8997e-12
```

Visualize Top Correlations Using Bar Chart

Display the three most significant correlations using a horizontal bar chart. Represent negative correlations with red bars and positive correlations with blue bars.

```
labels = TTopk.firstVar + " & " + TTopk.secondVar;
b = barh(labels,abs(TTopk.corrCoef));

negCorr = TTopk.corrCoef < 0;
b.FaceColor = "flat";
b.CData(negCorr,:) = repmat([1 0 0],nnz(negCorr),1);
b.Labels = TTopk.corrCoef;
b.LabelLocation = "end-inside";

title("Top Correlations")
xlabel("Correlation Coefficient")
ylabel("Correlated Variables")
```



See Also

corrcoef | heatmap | barh

Regression Analysis

- "Linear Correlation" on page 2-2
- "Linear Regression" on page 2-5
- "Interactive Fitting" on page 2-13
- "Programmatic Fitting" on page 2-26

Linear Correlation

In this section...
"Introduction" on page 2-2
"Covariance" on page 2-2
"Correlation Coefficients" on page 2-3

Introduction

Correlation quantifies the strength of a linear relationship between two variables. When there is no correlation between two variables, then there is no tendency for the values of the variables to increase or decrease in tandem. Two variables that are uncorrelated are not necessarily independent, however, because they might have a nonlinear relationship.

You can use linear correlation to investigate whether a linear relationship exists between variables without having to assume or fit a specific model to your data. Two variables that have a small or no linear correlation might have a strong nonlinear relationship. However, calculating linear correlation before fitting a model is a useful way to identify variables that have a simple relationship. Another way to explore how variables are related is to make scatter plots of your data.

Covariance quantifies the strength of a linear relationship between two variables in units relative to their variances. Correlations are standardized covariances, giving a dimensionless quantity that measures the degree of a linear relationship, separate from the scale of either variable.

The following MATLAB functions compute sample correlation coefficients and covariance. These sample coefficients are estimates of the true covariance and correlation coefficients of the population from which the data sample is drawn.

Function	Description
`corrcoef`	Correlation coefficient matrix
`cov`	Covariance matrix

Covariance

Use the MATLAB `cov` function to calculate the sample covariance matrix for a data matrix (where each column represents a separate quantity).

The sample covariance matrix has the following properties:

- `cov(X)` is symmetric.
- `diag(cov(X))` is a vector of variances for each data column. The variances represent a measure of the spread or dispersion of data in the corresponding column. (The `var` function calculates variance.)
- `sqrt(diag(cov(X)))` is a vector of standard deviations. (The `std` function calculates standard deviation.)
- The off-diagonal elements of the covariance matrix represent the covariances between the individual data columns.

Here, `X` can be a vector or a matrix. For an *m*-by-*n* matrix, the covariance matrix is *n*-by-*n*.

For an example of calculating the covariance, load the sample data in `count.dat` that contains a 24-by-3 matrix:

```
load count.dat
```

Calculate the covariance matrix for this data:

```
cov(count)
```

MATLAB responds with the following result:

```
ans =
    1.0e+003 *
       0.6437  0.9802  1.6567
       0.9802  1.7144  2.6908
       1.6567  2.6908  4.6278
```

The covariance matrix for this data has the following form:

$$\begin{bmatrix} s^2{}_{11} & s^2{}_{12} & s^2{}_{13} \\ s^2{}_{21} & s^2{}_{22} & s^2{}_{23} \\ s^2{}_{31} & s^2{}_{32} & s^2{}_{33} \end{bmatrix}$$

$$s^2{}_{ij} = s^2{}_{ji}$$

Here, $s^2{}_{ij}$ is the sample covariance between column *i* and column *j* of the data. Because the `count` matrix contains three columns, the covariance matrix is 3-by-3.

Note In the special case when a vector is the argument of `cov`, the function returns the variance.

Correlation Coefficients

The function `corrcoef` produces a matrix of sample correlation coefficients for a data matrix (where each column represents a separate quantity). The correlation coefficients range from -1 to 1, where

- Values close to 1 indicate that there is a positive linear relationship between the data columns.
- Values close to -1 indicate that one column of data has a negative linear relationship to another column of data (*anticorrelation*).
- Values close to or equal to 0 suggest there is no linear relationship between the data columns.

For an *m*-by-*n* matrix, the correlation-coefficient matrix is *n*-by-*n*. The arrangement of the elements in the correlation coefficient matrix corresponds to the location of the elements in the covariance matrix, as described in “Covariance” on page 2-2.

For an example of calculating correlation coefficients, load the sample data in `count.dat` that contains a 24-by-3 matrix:

```
load count.dat
```

Type the following syntax to calculate the correlation coefficients:

```
corrcoef(count)
```

This results in the following 3-by-3 matrix of correlation coefficients:

```
ans =
    1.0000    0.9331    0.9599
    0.9331    1.0000    0.9553
    0.9599    0.9553    1.0000
```

Because all correlation coefficients are close to 1, there is a strong positive correlation between each pair of data columns in the `count` matrix.

See Also

`cov` | `corrcoef`

Related Examples

- "Identify and Visualize Correlated Variables" on page 1-55

Linear Regression

In this section...

"Introduction" on page 2-5

"Simple Linear Regression" on page 2-5

"Residuals and Goodness of Fit" on page 2-8

"Fitting Data with Curve Fitting Toolbox Functions" on page 2-11

Introduction

A data *model* explicitly describes a relationship between predictor and response variables. Linear regression fits a data model that is linear in the model coefficients. The most common type of linear regression is a least-squares fit, which can fit both lines and polynomials, among other linear models.

Before you model the relationship between pairs of quantities, it is a good idea to perform correlation analysis to establish if a linear relationship exists between these quantities. Be aware that variables can have nonlinear relationships, which correlation analysis cannot detect. For more information, see "Linear Correlation" on page 2-2.

The MATLAB Basic Fitting UI helps you to fit your data, so you can calculate model coefficients and plot the model on top of the data. For an example, see "Example: Using Basic Fitting UI" on page 2-14. You also can use the MATLAB `polyfit` and `polyval` functions to fit your data to a model that is linear in the coefficients. For an example, see "Programmatic Fitting" on page 2-28.

If you need to fit data with a nonlinear model, transform the variables to make the relationship linear. Alternatively, try to fit a nonlinear function directly using either the Statistics and Machine Learning Toolbox `nlinfit` function, the Optimization Toolbox™ `lsqcurvefit` function, or by applying functions in the Curve Fitting Toolbox™.

This topic explains how to:

- Perform simple linear regression using the `\` operator.
- Use correlation analysis to determine whether two quantities are related to justify fitting the data.
- Fit a linear model to the data.
- Evaluate the goodness of fit by plotting residuals and looking for patterns.
- Calculate measures of goodness of fit R^2 and adjusted R^2

Simple Linear Regression

This example shows how to perform simple linear regression using the `accidents` dataset. The example also shows you how to calculate the coefficient of determination R^2 to evaluate the regressions. The `accidents` dataset contains data for fatal traffic accidents in US states.

Linear regression models the relation between a dependent, or response, variable y and one or more independent, or predictor, variables $x_1, \ldots, x_n$. Simple linear regression considers only one independent variable using the relation

$$y = \beta_0 + \beta_1 x + \epsilon,$$

where β_0 is the y-intercept, β_1 is the slope (or regression coefficient), and ϵ is the error term.

Start with a set of n observed values of x and y given by (x_1, y_1), (x_2, y_2), ..., (x_n, y_n). Using the simple linear regression relation, these values form a system of linear equations. Represent these equations in matrix form as

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix}.$$

Let

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, X = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix}, B = \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix}.$$

The relation is now $Y = XB$.

In MATLAB, you can find B using the `mldivide` operator as `B = X\Y`.

From the dataset `accidents`, load accident data in `y` and state population data in `x`. Find the linear regression relation $y = \beta_1 x$ between the accidents in a state and the population of a state using the `\` operator. The `\` operator performs a least-squares regression.

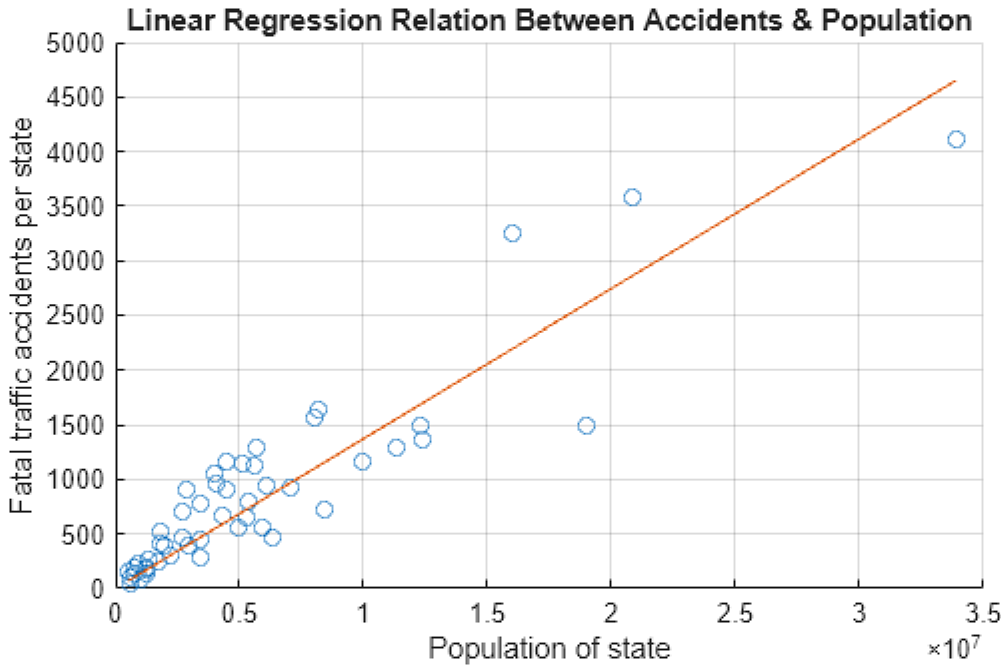
```
load accidents
x = hwydata(:,14); %Population of states
y = hwydata(:,4); %Accidents per state
format long
b1 = x\y
```

```
b1 =
     1.372716735564871e-04
```

`b1` is the slope or regression coefficient. The linear relation is $y = \beta_1 x = 0.0001372x$.

Calculate the accidents per state `yCalc` from `x` using the relation. Visualize the regression by plotting the actual values `y` and the calculated values `yCalc`.

```
yCalc1 = b1*x;
scatter(x,y)
hold on
plot(x,yCalc1)
xlabel('Population of state')
ylabel('Fatal traffic accidents per state')
title('Linear Regression Relation Between Accidents & Population')
grid on
```



Improve the fit by including a y-intercept β_0 in your model as $y = \beta_0 + \beta_1 x$. Calculate β_0 by padding x with a column of ones and using the \ operator.

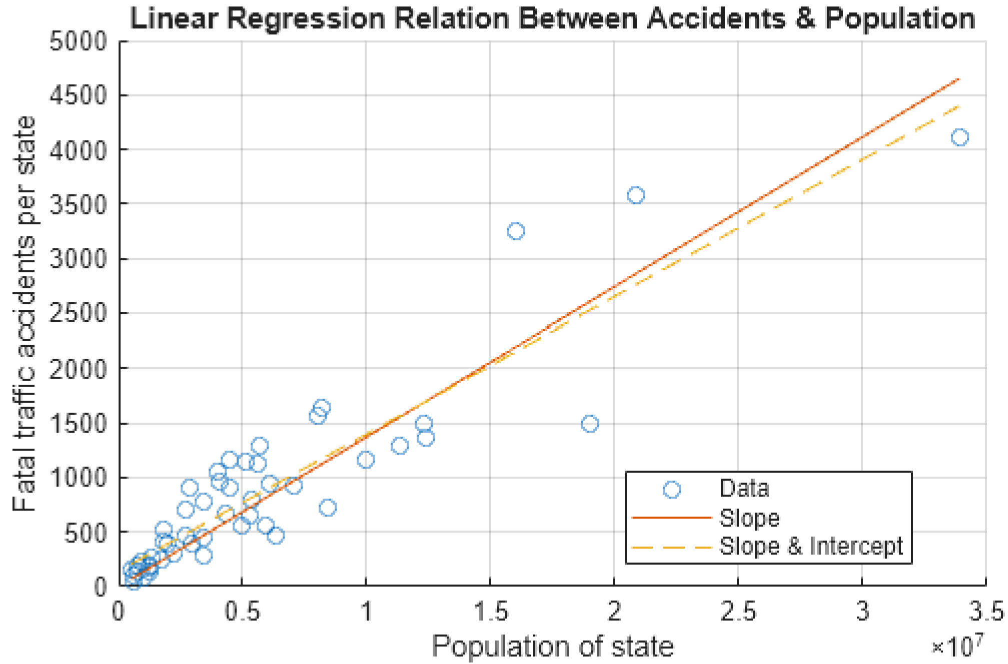
```
X = [ones(length(x),1) x];
b = X\y
```

```
b = 2×1
10^2 ×

   1.427120171726538
   0.000001256394274
```

This result represents the relation $y = \beta_0 + \beta_1 x = 142.7120 + 0.0001256x$.

Visualize the relation by plotting it on the same figure.

```
yCalc2 = X*b;
plot(x,yCalc2,'--')
legend('Data','Slope','Slope & Intercept','Location','best');
```



From the figure, the two fits look similar. One method to find the better fit is to calculate the coefficient of determination, R^2. R^2 is one measure of how well a model can predict the data, and falls between 0 and 1. The higher the value of R^2, the better the model is at predicting the data.

Where $\hat{y}$ represents the calculated values of y and $\bar{y}$ is the mean of y, R^2 is defined as

$$R^2 = 1 - \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{\sum_{i=1}^{n} (y_i - \bar{y})^2}.$$

Find the better fit of the two fits by comparing values of R^2. As the R^2 values show, the second fit that includes a y-intercept is better.

```
Rsq1 = 1 - sum((y - yCalc1).^2)/sum((y - mean(y)).^2)
```

```
Rsq1 =
   0.822235650485566
```

```
Rsq2 = 1 - sum((y - yCalc2).^2)/sum((y - mean(y)).^2)
```

```
Rsq2 =
   0.838210531103428
```

Residuals and Goodness of Fit

Residuals are the difference between the *observed* values of the response (dependent) variable and the values that a model *predicts*. When you fit a model that is appropriate for your data, the residuals

approximate independent random errors. That is, the distribution of residuals ought not to exhibit a discernible pattern.

Producing a fit using a linear model requires minimizing the sum of the squares of the residuals. This minimization yields what is called a least-squares fit. You can gain insight into the “goodness” of a fit by visually examining a plot of the residuals. If the residual plot has a pattern (that is, residual data points do not appear to have a random scatter), the randomness indicates that the model does not properly fit the data.

Evaluate each fit you make in the context of your data. For example, if your goal of fitting the data is to extract coefficients that have physical meaning, then it is important that your model reflect the physics of the data. Understanding what your data represents, how it was measured, and how it is modeled is important when evaluating the goodness of fit.

One measure of goodness of fit is the coefficient of determination, or R^2 (pronounced r-square). This statistic indicates how closely values you obtain from fitting a model match the dependent variable the model is intended to predict. Statisticians often define R^2 using the residual variance from a fitted model:

$$R^2 = 1 - SS_{resid} / SS_{total}$$

SS_{resid} is the sum of the squared residuals from the regression. SS_{total} is the sum of the squared differences from the mean of the dependent variable (*total sum of squares*). Both are positive scalars.

To learn how to compute R^2 when you use the Basic Fitting tool, see “R2, the Coefficient of Determination” on page 2-19. To learn more about calculating the R^2 statistic and its multivariate generalization, continue reading here.

Example: Computing R2 from Polynomial Fits

You can derive R^2 from the coefficients of a polynomial regression to determine how much variance in `y` a linear model explains, as the following example describes:

1. Create two variables, `x` and `y`, from the first two columns of the `count` variable in the data file `count.dat`:

   ```
   load count.dat
   x = count(:,1);
   y = count(:,2);
   ```
2. Use `polyfit` to compute a linear regression that predicts `y` from `x`:

   ```
   p = polyfit(x,y,1)

   p =
       1.5229   -2.1911
   ```

   `p(1)` is the slope and `p(2)` is the intercept of the linear predictor. You can also obtain regression coefficients using the Basic Fitting UI on page 2-13.
3. Call `polyval` to use `p` to predict `y`, calling the result `yfit`:

   ```
   yfit = polyval(p,x);
   ```

   Using `polyval` saves you from typing the fit equation yourself, which in this case looks like:

   ```
   yfit =  p(1) * x + p(2);
   ```

4 Compute the residual values as a vector of signed numbers:

```
yresid = y - yfit;
```

5 Square the residuals and total them to obtain the residual sum of squares:

```
SSresid = sum(yresid.^2);
```

6 Compute the total sum of squares of y by multiplying the variance of y by the number of observations minus 1:

```
SStotal = (length(y)-1) * var(y);
```

7 Compute R^2 using the formula given in the introduction of this topic:

```
rsq = 1 - SSresid/SStotal

rsq =
    0.8707
```

This demonstrates that the linear equation `1.5229 * x -2.1911` predicts 87% of the variance in the variable y.

Computing Adjusted R2 for Polynomial Regressions

You can usually reduce the residuals in a model by fitting a higher degree polynomial. When you add more terms, you increase the coefficient of determination, R^2. You get a closer fit to the data, but at the expense of a more complex model, for which R^2 cannot account. However, a refinement of this statistic, adjusted R^2, does include a penalty for the number of terms in a model. Adjusted R^2, therefore, is more appropriate for comparing how different models fit to the same data. The adjusted R^2 is defined as:

$$R^2_{adjusted} = 1 - (SS_{resid} / SS_{total})*((n-1)/(n-d-1))$$

where n is the number of observations in your data, and d is the degree of the polynomial. (A linear fit has a degree of 1, a quadratic fit 2, a cubic fit 3, and so on.)

The following example repeats the steps of the previous example, “Example: Computing R2 from Polynomial Fits” on page 2-9, but performs a cubic (degree 3) fit instead of a linear (degree 1) fit. From the cubic fit, you compute both simple and adjusted R^2 values to evaluate whether the extra terms improve predictive power:

1 Create two variables, x and y, from the first two columns of the `count` variable in the data file `count.dat`:

```
load count.dat
x = count(:,1);
y = count(:,2);
```

2 Call `polyfit` to generate a cubic fit to predict y from x:

```
p = polyfit(x,y,3)

p =
   -0.0003    0.0390    0.2233    6.2779
```

`p(4)` is the intercept of the cubic predictor. You can also obtain regression coefficients using the Basic Fitting UI on page 2-13.

3 Call `polyval` to use the coefficients in p to predict y, naming the result `yfit`:

```
yfit = polyval(p,x);
```

polyval evaluates the explicit equation you could manually enter as:

```
yfit =  p(1) * x.^3 + p(2) * x.^2 + p(3) * x + p(4);
```

4. Compute the residual values as a vector of signed numbers:

```
yresid = y - yfit;
```

5. Square the residuals and total them to obtain the residual sum of squares:

```
SSresid = sum(yresid.^2);
```

6. Compute the total sum of squares of y by multiplying the variance of y by the number of observations minus 1:

```
SStotal = (length(y)-1) * var(y);
```

7. Compute simple R^2 for the cubic fit using the formula given in the introduction of this topic:

```
rsq = 1 - SSresid/SStotal

rsq =
    0.9083
```

8. Finally, compute adjusted R^2 to account for degrees of freedom:

```
rsq_adj = 1 - SSresid/SStotal * (length(y)-1)/(length(y)-length(p))

rsq_adj =
    0.8945
```

The adjusted R^2, 0.8945, is smaller than simple R^2, .9083. It provides a more reliable estimate of the power of your polynomial model to predict.

In many polynomial regression models, adding terms to the equation increases both R^2 and adjusted R^2. In the preceding example, using a cubic fit increased both statistics compared to a linear fit. (You can compute adjusted R^2 for the linear fit for yourself to demonstrate that it has a lower value.) However, it is not always true that a linear fit is worse than a higher-order fit: a more complicated fit can have a lower adjusted R^2 than a simpler fit, indicating that the increased complexity is not justified. Also, while R^2 always varies between 0 and 1 for the polynomial regression models that the Basic Fitting tool generates, adjusted R^2 for some models can be negative, indicating that a model that has too many terms.

Correlation does not imply causality. Always interpret coefficients of correlation and determination cautiously. The coefficients only quantify how much variance in a dependent variable a fitted model removes. Such measures do not describe how appropriate your model—or the independent variables you select—are for explaining the behavior of the variable the model predicts.

Fitting Data with Curve Fitting Toolbox Functions

Curve Fitting Toolbox extends core MATLAB functionality by enabling the following data-fitting capabilities:

- Linear and nonlinear parametric fitting, including standard linear least squares, nonlinear least squares, weighted least squares, constrained least squares, and robust fitting procedures
- Nonparametric fitting
- Statistics for determining the goodness of fit

- Extrapolation, differentiation, and integration
- Dialog box that facilitates data sectioning and smoothing
- Saving fit results in various formats, including MATLAB code files, MAT-files, and workspace variables

For more information, see Curve Fitting Toolbox.

Interactive Fitting

In this section...

"Basic Fitting UI" on page 2-13
"Preparing for Basic Fitting" on page 2-13
"Opening the Basic Fitting UI" on page 2-13
"Example: Using Basic Fitting UI" on page 2-14

Basic Fitting UI

The MATLAB Basic Fitting UI allows you to interactively:

- Model data using a spline interpolant, a shape-preserving interpolant, or a polynomial up to the tenth degree
- Plot one or more fits together with data
- Plot the residuals of the fits
- Compute model coefficients
- Compute the norm of the residuals (a statistic you can use to analyze how well a model fits your data)
- Use the model to interpolate or extrapolate outside of the data
- Save coefficients and computed values to the MATLAB workspace for use outside of the dialog box
- Generate MATLAB code to recompute fits and reproduce plots with new data

Note The Basic Fitting UI is only available for 2-D plots. For more advanced fitting and regression analysis, see the Curve Fitting Toolbox documentation and the Statistics and Machine Learning Toolbox documentation.

Preparing for Basic Fitting

The Basic Fitting UI sorts your data in ascending order before fitting. If your data set is large and the values are not sorted in ascending order, it will take longer for the Basic Fitting UI to preprocess your data before fitting.

You can speed up the Basic Fitting UI by first sorting your data. To create sorted vectors `x_sorted` and `y_sorted` from data vectors `x` and `y`, use the MATLAB `sort` function:

```
[x_sorted, i] = sort(x);
y_sorted = y(i);
```

Opening the Basic Fitting UI

To use the Basic Fitting UI, you must first plot your data in a figure window, using any MATLAB plotting command that produces (only) *x* and *y* data.

To open the Basic Fitting UI, select **Tools > Basic Fitting** from the menus at the top of the figure window.

Example: Using Basic Fitting UI

This example shows how to use the Basic Fitting UI to fit, visualize, analyze, save, and generate code for polynomial regressions.

Load and Plot Census Data

The file `census.mat` contains US population data for the years 1790 through 1990 at 10 year intervals.

To load and plot the data, type the following commands at the MATLAB prompt:

```
load census
plot(cdate,pop,'ro')
```

The `load` command adds the following variables to the MATLAB workspace:

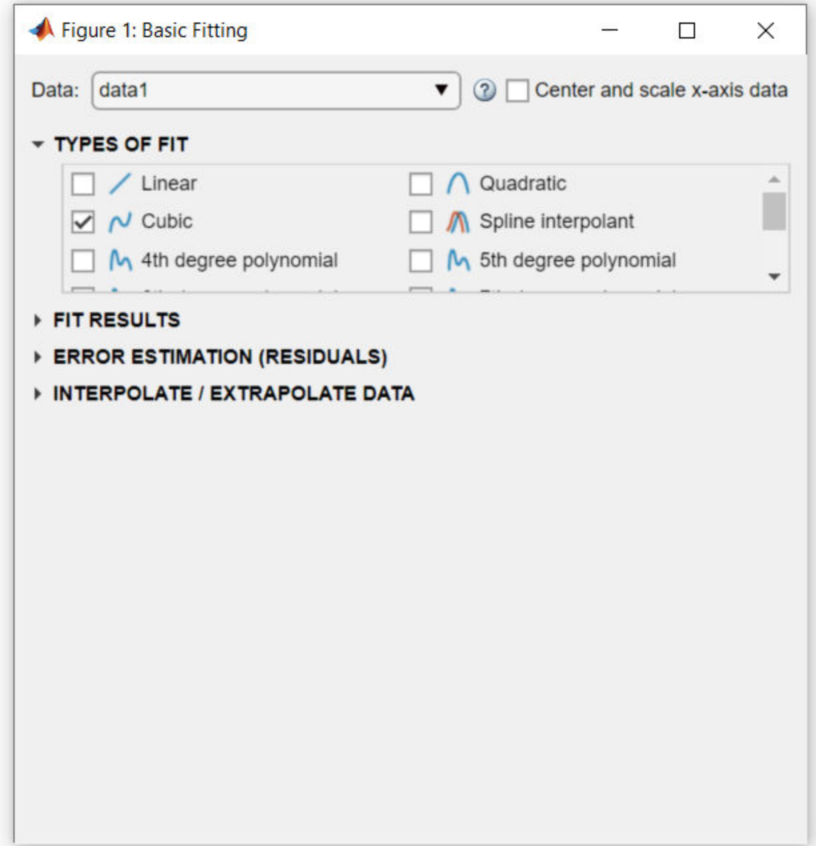
- `cdate` — A column vector containing the years from 1790 to 1990 in increments of 10. It is the predictor variable.
- `pop` — A column vector with US population for each year in `cdate`. It is the response variable.

The data vectors are sorted in ascending order, by year. The plot shows the population as a function of year.

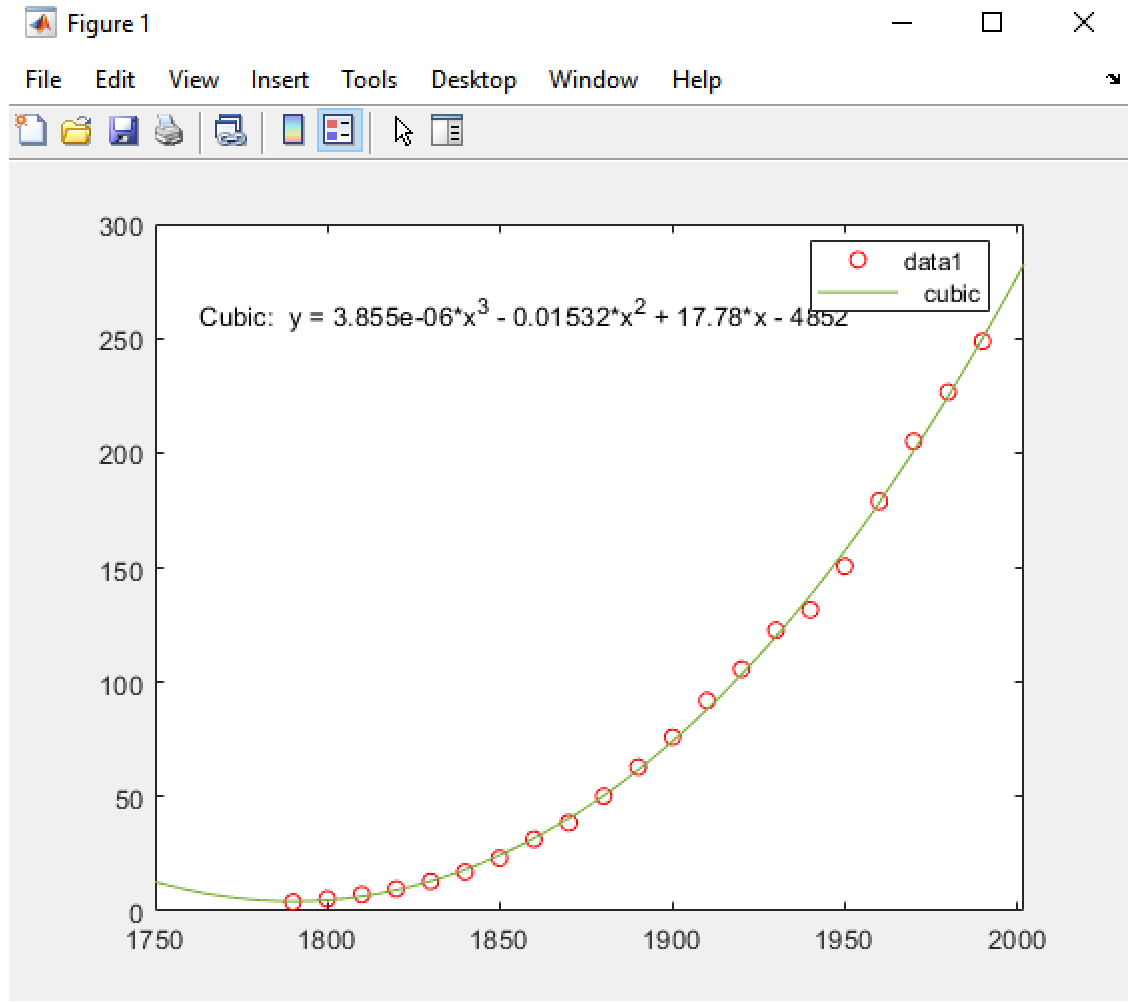
Now you are ready to fit an equation the data to model population growth over time.

Predict the Census Data with a Cubic Polynomial Fit

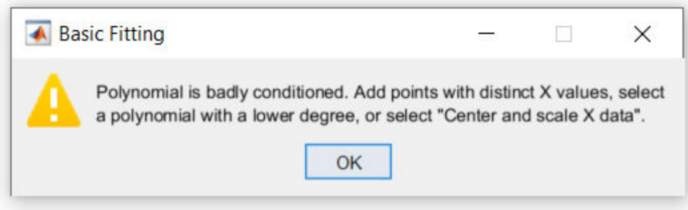
1. Open the Basic Fitting dialog box by selecting **Tools > Basic Fitting** in the Figure window.
2. In the **TYPES OF FIT** area of the Basic Fitting dialog box, select the **Cubic** check box to fit a cubic polynomial to the data.



MATLAB uses your selection to fit the data, and adds the cubic regression line to the graph as follows.



In computing the fit, MATLAB encounters problems and issues the following warning:

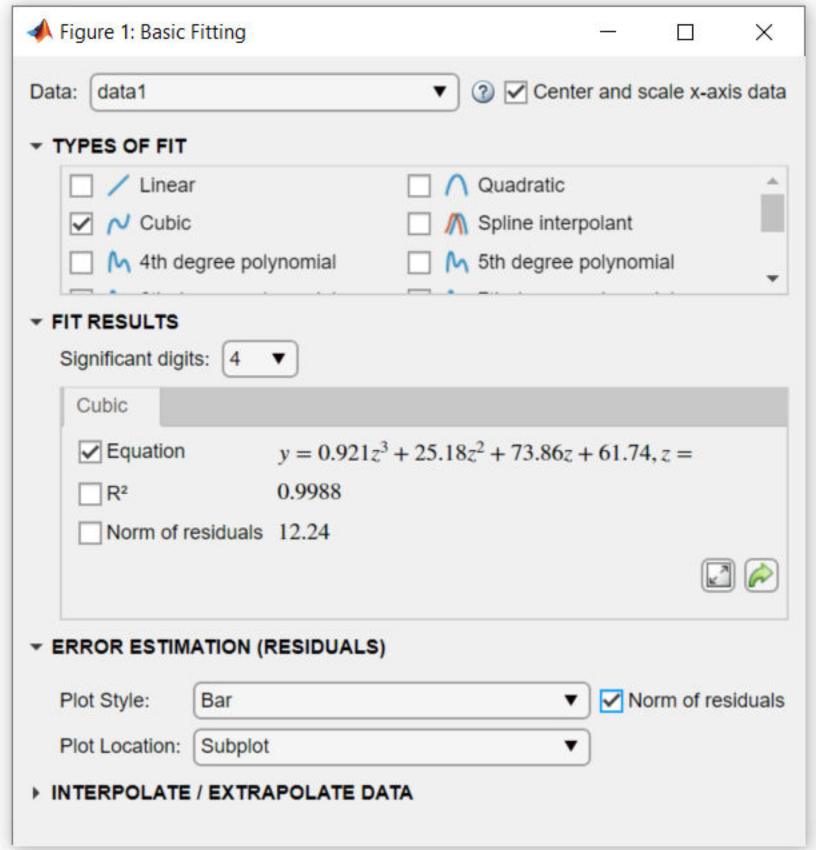


This warning indicates that the computed coefficients for the model are sensitive to random errors in the response (the measured population). It also suggests some things you can do to get a better fit.

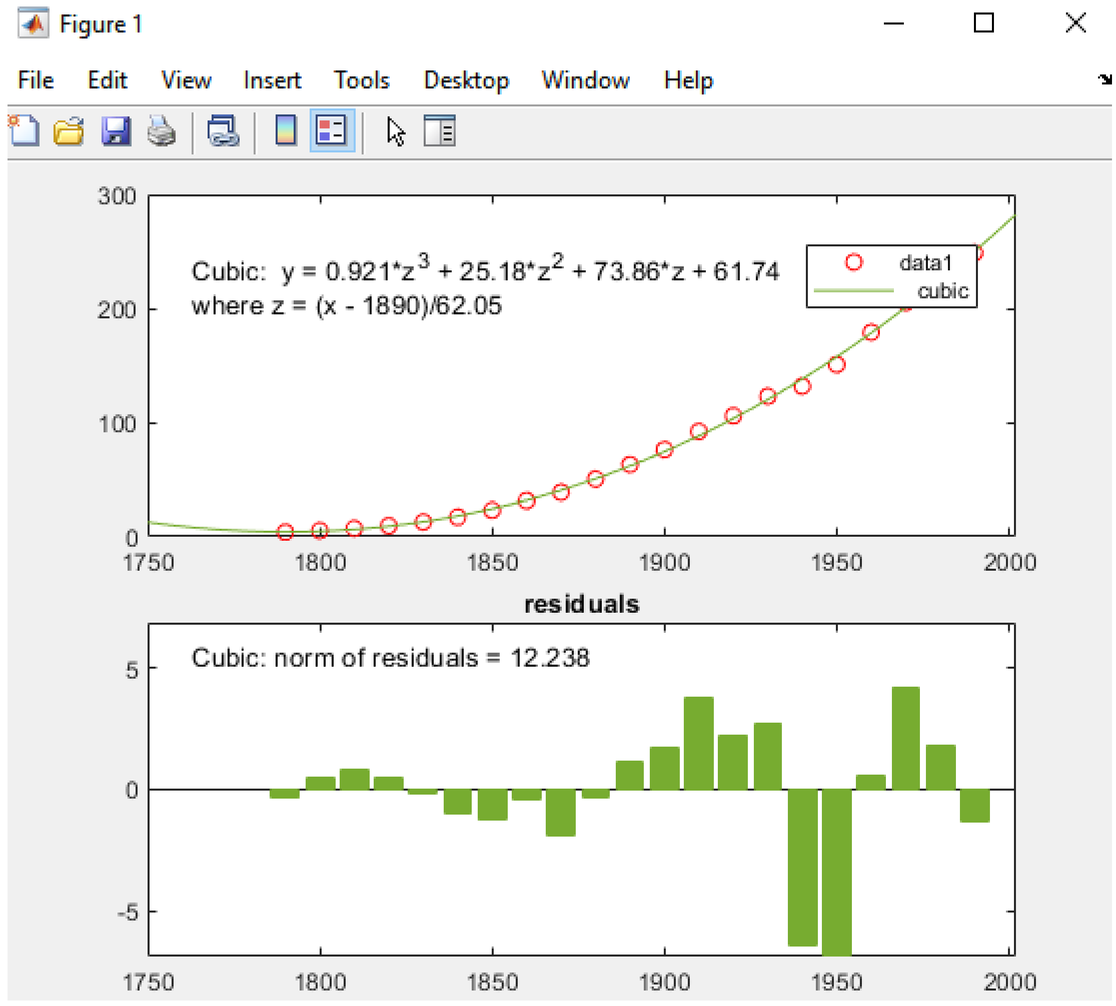
3. Continue to use a cubic fit. As you cannot add new observations to the census data, improve the fit by transforming the values you have to *z-scores* before recomputing a fit. Select the **Center and scale x-axis data** check box in the top right of the dialog box to make the Basic Fitting tool perform the transformation.

   To learn how centering and scaling data works, see “Learn How the Basic Fitting Tool Computes Fits” on page 2-23.

4 Under **ERROR ESTIMATION (RESIDUALS)**, select the **Norm of residuals** check box. Select **Bar** as the **Plot Style**.



Selecting these options creates a subplot of residuals as a bar graph.



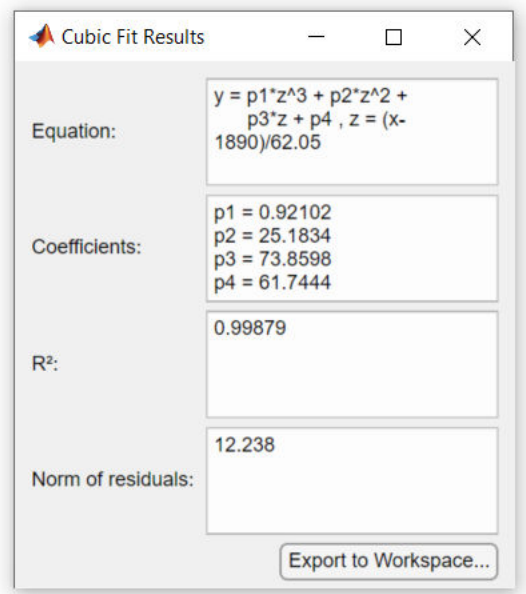
The cubic fit is a poor predictor before the year 1790, where it indicates a decreasing population. The model seems to approximate the data reasonably well after 1790. However, a pattern in the residuals shows that the model does not meet the assumption of normal error, which is a basis for the least-squares fitting. The **data 1** line identified in the legend are the observed *x* (`cdate`) and *y* (`pop`) data values. The **Cubic** regression line presents the fit after centering and scaling data values. Notice that the figure shows the original data units, even though the tool computes the fit using transformed z-scores.

For comparison, try fitting another polynomial equation to the census data by selecting it in the **TYPES OF FIT** area.

View and Save the Cubic Fit Parameters

In the Basic Fitting dialog box, click the **Expand Results** button

to display the estimated coefficients and the norm of residuals.



Save the fit data to the MATLAB workspace by clicking the **Export to Workspace** button on the Numerical results panel. The Save Fit to Workspace dialog box opens.

With all check boxes selected, click **OK** to save the fit parameters as a MATLAB structure `fit`:

```
fit
```

```
fit = 

  struct with fields:

     type: 'polynomial degree 3'
    coeff: [0.9210 25.1834 73.8598 61.7444]
```

Now, you can use the fit results in MATLAB programming, outside of the Basic Fitting UI.

R2, the Coefficient of Determination

You can get an indication of how well a polynomial regression predicts your observed data by computing the coefficient of determination, or R-square (written as R^2). The R^2 statistic, which ranges from 0 to 1, measures how useful the independent variable is in predicting values of the dependent variable:

- An R^2 value near 0 indicates that the fit is not much better than the model `y = constant`.
- An R^2 value near 1 indicates that the independent variable explains most of the variability in the dependent variable.

R^2 is computed from the residuals, the signed differences between an observed dependent value and the value your fit predicts for it.

$$\text{residuals} = y_{\text{observed}} - y_{\text{fitted}} \tag{2-1}$$

The R^2 number for the cubic fit in this example, 0.9988, is located under **FIT RESULTS** in the Basic Fitting dialog.

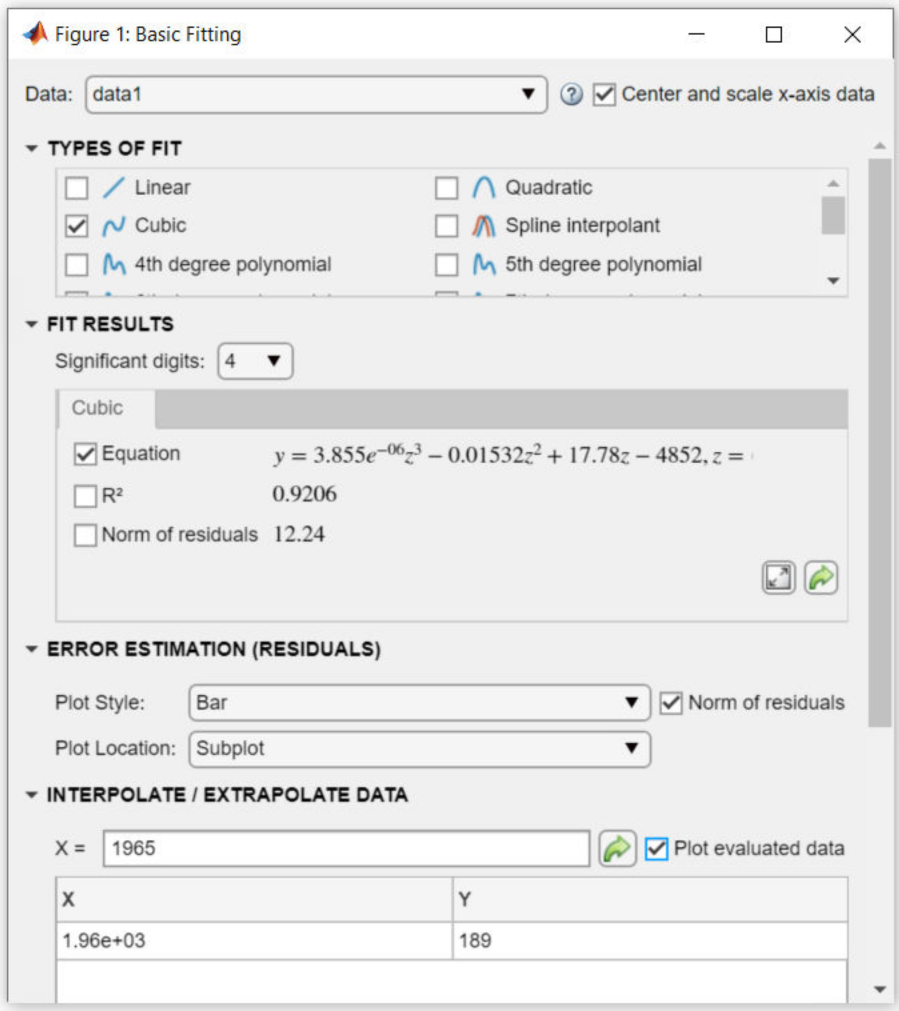
To compare the R^2 number for the cubic fit to a linear least-squares fit, select **Linear** under **TYPES OF FIT** and obtain the R^2 number, 0.921. This result indicates that a linear least-squares fit of the population data explains 92.1% of its variance. As the cubic fit of this data explains 99.9% of that variance, the latter seems to be a better predictor. However, because a cubic fit predicts using three variables (x, x^2, and x^3), a basic R^2 value does not fully reflect how robust the fit is. A more appropriate measure for evaluating the goodness of multivariate fits is adjusted R^2. For information about computing and using adjusted R^2, see “Residuals and Goodness of Fit” on page 2-8.

Interpolate and Extrapolate Population Values

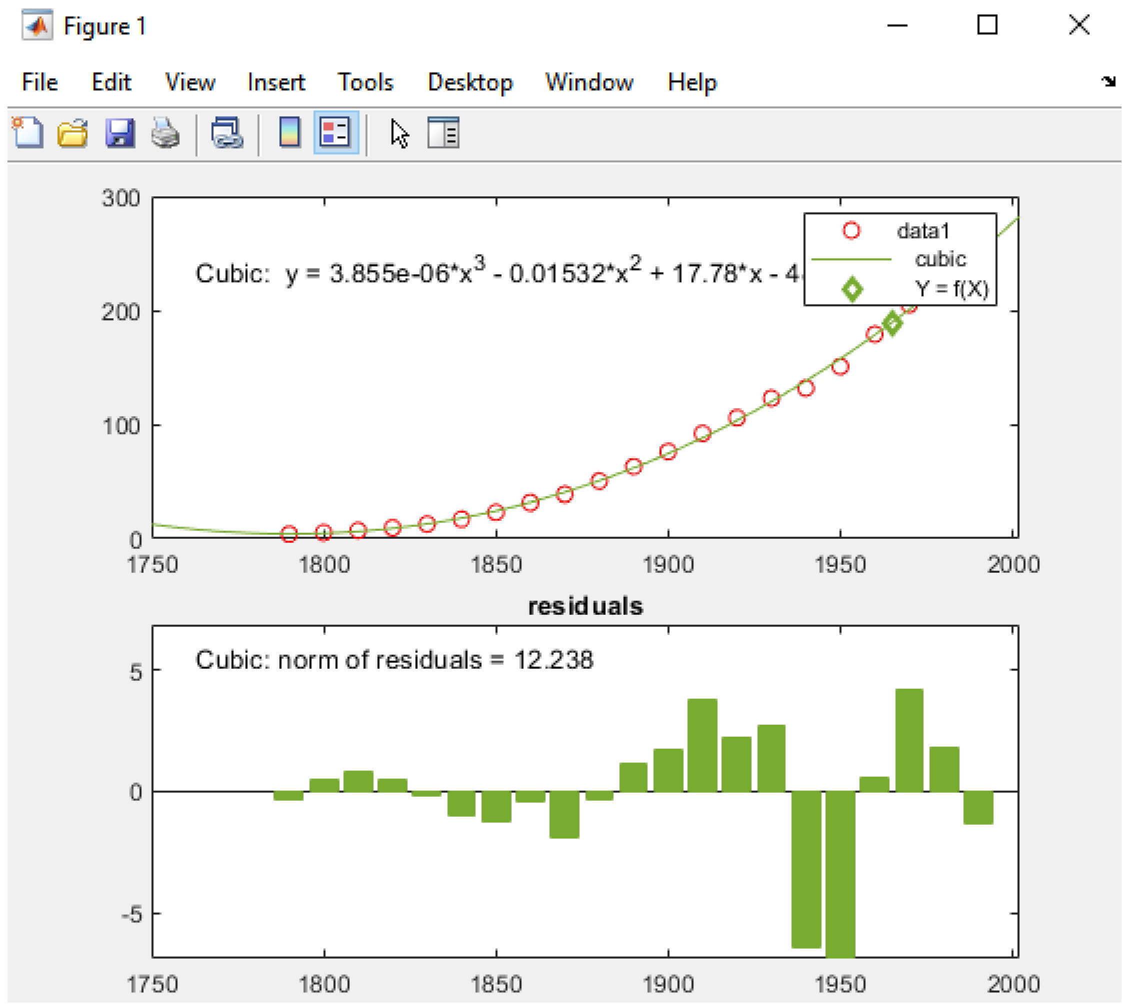
Suppose you want to use the cubic model to interpolate the US population in 1965 (a date not provided in the original data).

In the Basic Fitting dialog box, under **INTERPOLATE / EXTRAPOLATE DATA**, enter the **X** value 1965 and check the **Plot evaluated data** box.

Note Use unscaled and uncentered X values. You do not need to center and scale first, even though you selected to scale X values to obtain the coefficients in “Predict the Census Data with a Cubic Polynomial Fit” on page 2-14. The Basic Fitting tool makes the necessary adjustments behind the scenes.



The X values and the corresponding values for f(X) are computed from the fit and plotted as follows:



Generate a Code File to Reproduce the Result

After completing a Basic Fitting session, you can generate MATLAB code that recomputes fits and reproduces plots with new data.

1 In the Figure window, select **File > Generate Code**.

   This creates a function and displays it in the MATLAB Editor. The code shows you how to programmatically reproduce what you did interactively with the Basic Fitting dialog box.
2 Change the name of the function on the first line from `createfigure` to something more specific, like `censusplot`. Save the code file to your current folder with the file name `censusplot.m` The function begins with:

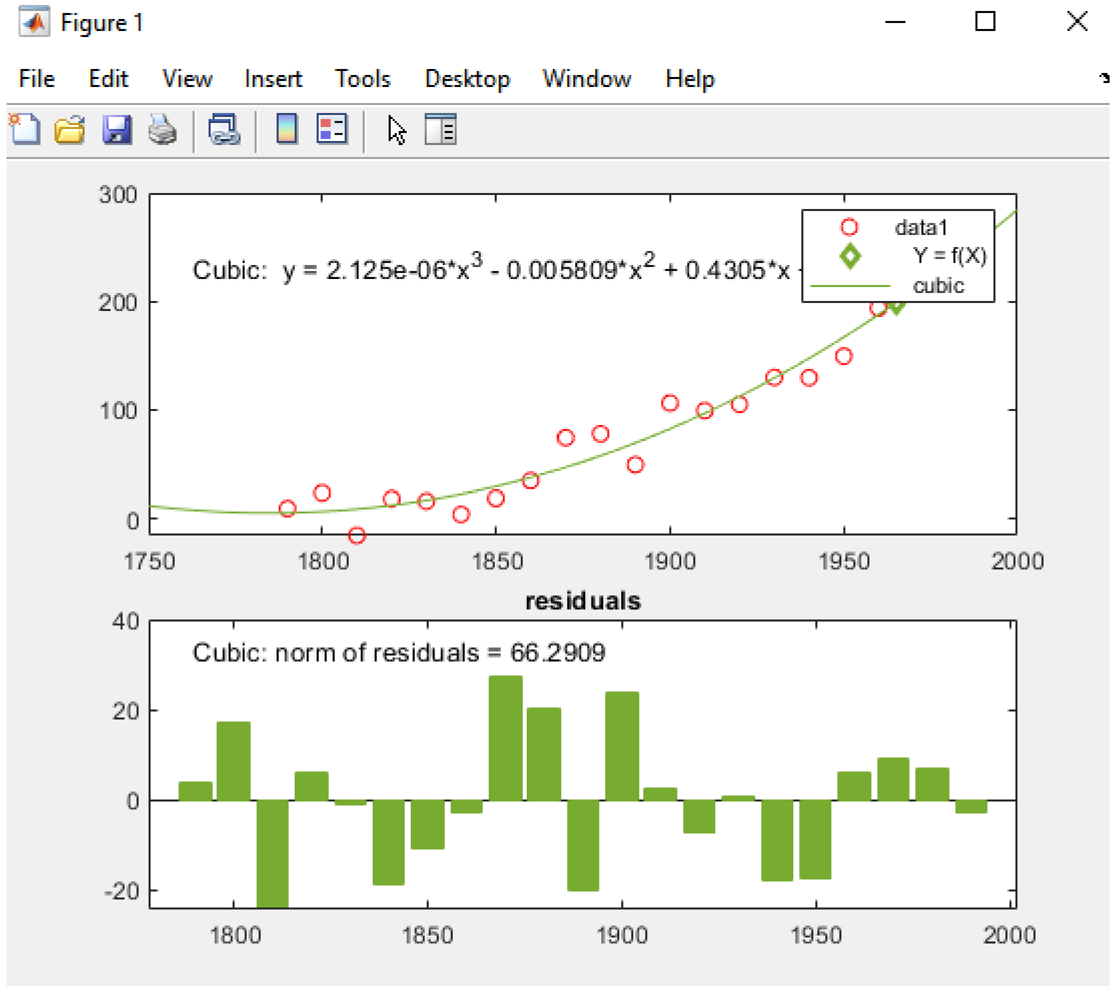
```
function censusplot(X1, Y1, valuesToEvaluate1)
```
3 Generate some new, randomly perturbed census data:

```
rng('default')
randpop = pop + 10*randn(size(pop));
```
4 Reproduce the plot with the new data and recompute the fit:

```
censusplot(cdate,randpop,1965)
```

You need three input arguments: *x*,*y* values (`data 1`) plotted in the original graph, plus an *x*-value for a marker.

The following figure displays the plot that the generated code produces. The new plot matches the appearance of the figure from which you generated code except for the *y* data values, the equation for the cubic fit, and the residual values in the bar graph, as expected.



Learn How the Basic Fitting Tool Computes Fits

The Basic Fitting tool calls the `polyfit` function to compute polynomial fits. It calls the `polyval` function to evaluate the fits. `polyfit` analyzes its inputs to determine if the data is well conditioned for the requested degree of fit.

When it finds badly conditioned data, `polyfit` computes a regression as well as it can, but it also returns a warning that the fit could be improved. The Basic Fitting example section “Predict the Census Data with a Cubic Polynomial Fit” on page 2-14 displays this warning.

One way to improve model reliability is to add data points. However, adding observations to a data set is not always feasible. An alternative strategy is to transform the predictor variable to normalize its center and scale. (In the example, the predictor is the vector of census dates.)

The `polyfit` function normalizes by computing *z-scores*:

$$z = \frac{x - \mu}{\sigma}$$

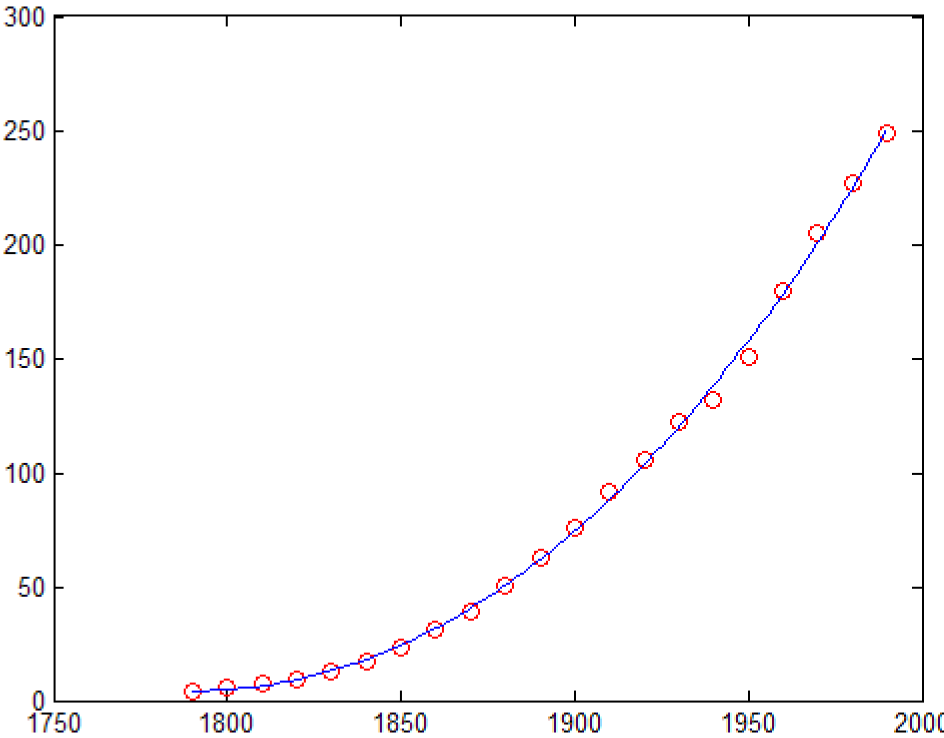
where x is the predictor data, μ is the mean of x, and σ is the standard deviation of x. The z-scores give the data a mean of 0 and a standard deviation of 1. In the Basic Fitting UI, you transform the predictor data to z-scores by selecting the **Center and scale x-axis data** check box.

After centering and scaling, model coefficients are computed for the y data as a function of z. These are different (and more robust) than the coefficients computed for y as a function of x. The form of the model and the norm of the residuals do not change. The Basic Fitting UI automatically rescales the z-scores so that the fit plots on the same scale as the original x data.

To understand the way in which the centered and scaled data is used as an intermediary to create the final plot, run the following code in the Command Window:

```
close
load census
x = cdate;
y = pop;
z = (x-mean(x))/std(x); % Compute z-scores of x data

plot(x,y,'ro') % Plot data as red markers
hold on        % Prepare axes to accept new graph on top

zfit = linspace(z(1),z(end),100);
pz = polyfit(z,y,3); % Compute conditioned fit
yfit = polyval(pz,zfit);

xfit = linspace(x(1),x(end),100);
plot(xfit,yfit,'b-') % Plot conditioned fit vs. x data
```

The centered and scaled cubic polynomial plots as a blue line, as shown here:



In the code, computation of `z` illustrates how to normalize data. The `polyfit` function performs the transformation itself if you provide three return arguments when calling it:

```
[p,S,mu] = polyfit(x,y,n)
```

The returned regression parameters, `p`, now are based on normalized `x`. The returned vector, `mu`, contains the mean and standard deviation of `x`. For more information, see the `polyfit` reference page.

Programmatic Fitting

In this section...
“MATLAB Functions for Polynomial Models” on page 2-26
“Linear Model with Nonpolynomial Terms” on page 2-26
“Multiple Regression” on page 2-27
“Programmatic Fitting” on page 2-28

MATLAB Functions for Polynomial Models

Two MATLAB functions can model your data with a polynomial.

Polynomial Fit Functions

Function	Description
`polyfit`	`polyfit(x,y,n)` finds the coefficients of a polynomial `p(x)` of degree `n` that fits the `y` data by minimizing the sum of the squares of the deviations of the data from the model (least-squares fit).
`polyval`	`polyval(p,x)` returns the value of a polynomial of degree `n` that was determined by `polyfit`, evaluated at `x`.

If you are trying to model a physical situation, it is always important to consider whether a model of a specific order is meaningful in your situation.

Linear Model with Nonpolynomial Terms

This example shows how to fit data with a linear model containing nonpolynomial terms.

When a polynomial function does not produce a satisfactory model of your data, you can try using a linear model with nonpolynomial terms. For example, consider the following function that is linear in the parameters a_0, a_1, and a_2, but nonlinear in the t data:

$$y = a_0 + a_1 e^{-t} + a_2 t e^{-t}.$$

You can compute the unknown coefficients a_0, a_1, and a_2 by constructing and solving a set of simultaneous equations and solving for the parameters. The following syntax accomplishes this by forming a *design matrix*, where each column represents a variable used to predict the response (a term in the model) and each row corresponds to one observation of those variables.

Enter `t` and `y` as column vectors.

```
t = [0 0.3 0.8 1.1 1.6 2.3]';
y = [0.6 0.67 1.01 1.35 1.47 1.25]';
```

Form the design matrix.

```
X = [ones(size(t))  exp(-t)  t.*exp(-t)];
```

Calculate model coefficients.

```
a = X\y
```

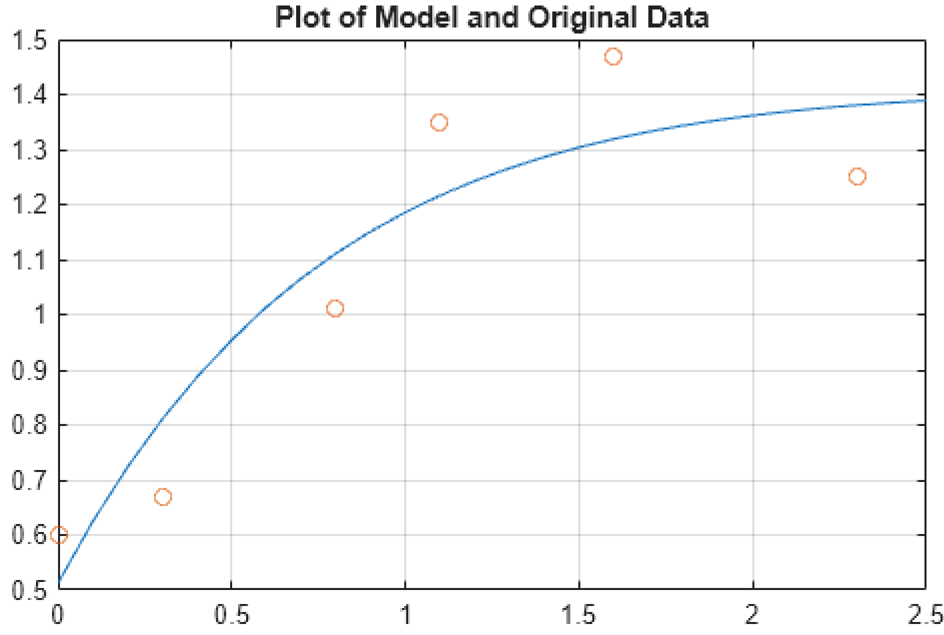
```
a = 3×1

    1.3983
   -0.8860
    0.3085
```

Therefore, the model of the data is given by

$$y = 1.3983 - 0.8860e^{-t} + 0.3085te^{-t}.$$

Now evaluate the model at regularly spaced points and plot the model with the original data.

```
T = (0:0.1:2.5)';
Y = [ones(size(T))  exp(-T)  T.*exp(-T)]*a;
plot(T,Y,'-',t,y,'o'), grid on
title('Plot of Model and Original Data')
```



Multiple Regression

This example shows how to use multiple regression to model data that is a function of more than one predictor variable.

When y is a function of more than one predictor variable, the matrix equations that express the relationships among the variables must be expanded to accommodate the additional data. This is called *multiple regression*.

Measure a quantity y for several values of x_1 and x_2. Store these values in vectors x1, x2, and y, respectively.

```
x1 = [.2 .5 .6 .8 1.0 1.1]';
x2 = [.1 .3 .4 .9 1.1 1.4]';
y  = [.17 .26 .28 .23 .27 .24]';
```

A model of this data is of the form

$$y = a_0 + a_1x_1 + a_2x_2 .$$

Multiple regression solves for unknown coefficients a_0, a_1, and a_2 by minimizing the sum of the squares of the deviations of the data from the model (least-squares fit).

Construct and solve the set of simultaneous equations by forming a design matrix, X.

```
X = [ones(size(x1))  x1  x2];
```

Solve for the parameters by using the backslash operator.

```
a = X\y
```

```
a = 3×1

    0.1018
    0.4844
   -0.2847
```

The least-squares fit model of the data is

$$y = 0.1018 + 0.4844x_1 - 0.2847x_2 .$$

To validate the model, find the maximum of the absolute value of the deviation of the data from the model.

```
Y = X*a;
MaxErr = max(abs(Y - y))
```

```
MaxErr =
0.0038
```

This value is much smaller than any of the data values, indicating that this model accurately follows the data.

Programmatic Fitting

This example shows how to use MATLAB functions to:

- “Calculate Correlation Coefficients” on page 2-29
- “Fit a Polynomial to the Data” on page 2-30
- “Plot and Calculate Confidence Bounds” on page 2-31

Load sample census data from `census.mat`, which contains U.S. population data from the years 1790 to 1990.

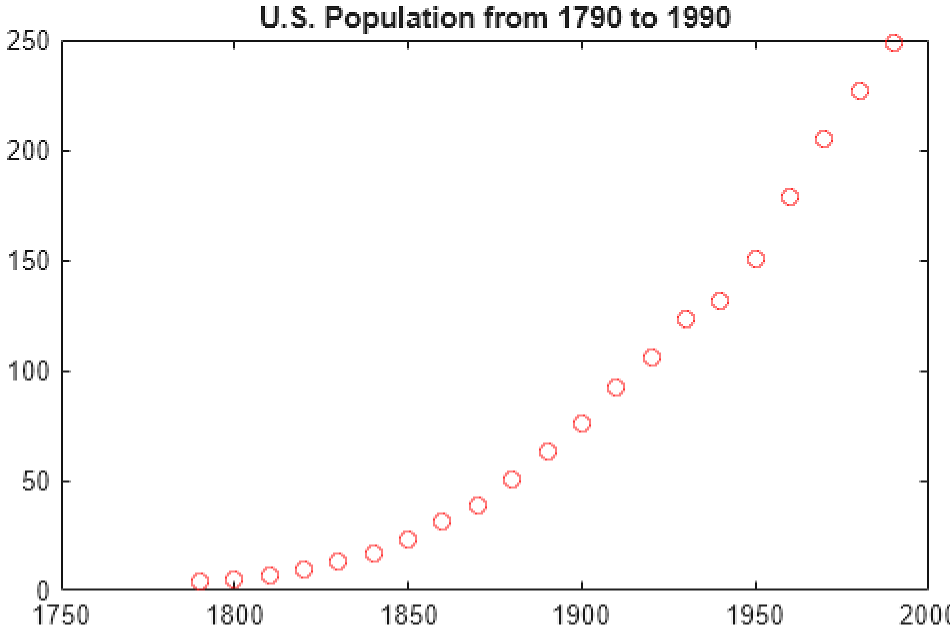
```
load census
```

This adds the following two variables to the MATLAB workspace.

- `cdate` is a column vector containing the years 1790 to 1990 in increments of 10.
- `pop` is a column vector with the U.S. population numbers corresponding to each year in `cdate`.

Plot the data.

```
plot(cdate,pop,'ro')
title('U.S. Population from 1790 to 1990')
```



The plot shows a strong pattern, which indicates a high correlation between the variables.

Calculate Correlation Coefficients

In this portion of the example, you determine the statistical correlation between the variables `cdate` and `pop` to justify modeling the data. For more information about correlation coefficients, see “Linear Correlation” on page 2-2.

Calculate the correlation-coefficient matrix.

```
corrcoef(cdate,pop)
```

```
ans = 2×2

    1.0000    0.9597
    0.9597    1.0000
```

The diagonal matrix elements represent the perfect correlation of each variable with itself and are equal to 1. The off-diagonal elements are very close to 1, indicating that there is a strong statistical correlation between the variables `cdate` and `pop`.

Fit a Polynomial to the Data

This portion of the example applies the `polyfit` and `polyval` MATLAB functions to model the data.

Calculate fit parameters.

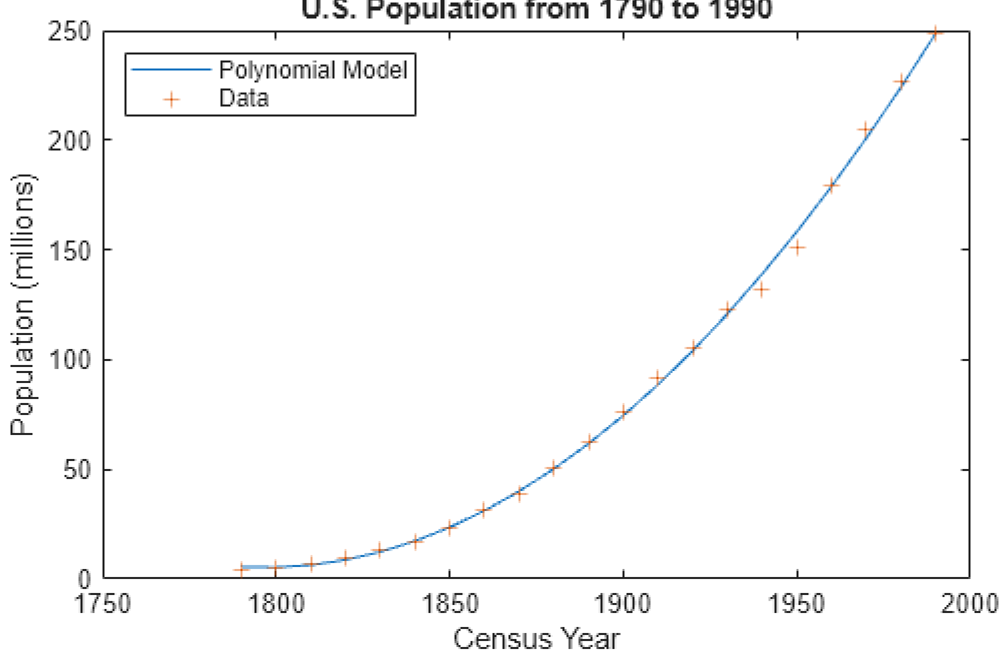
```
[p,ErrorEst] = polyfit(cdate,pop,2);
```

Evaluate the fit.

```
pop_fit = polyval(p,cdate,ErrorEst);
```

Plot the data and the fit.

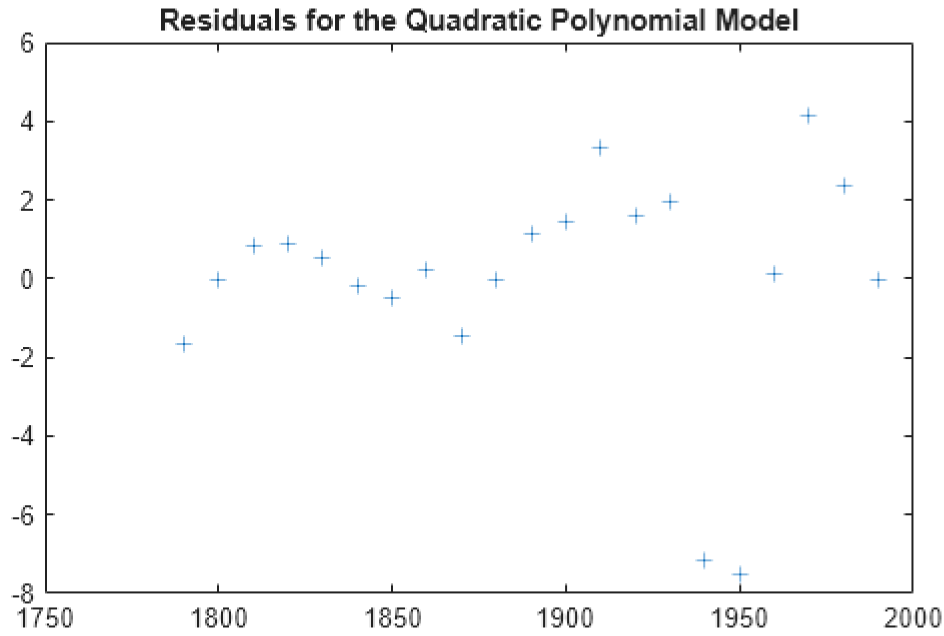
```
plot(cdate,pop_fit,'-',cdate,pop,'+');
title('U.S. Population from 1790 to 1990')
legend('Polynomial Model','Data','Location','NorthWest');
xlabel('Census Year');
ylabel('Population (millions)');
```



The plot shows that the quadratic-polynomial fit provides a good approximation to the data.

Calculate the residuals for this fit.

```
res = pop - pop_fit;
figure, plot(cdate,res,'+')
title('Residuals for the Quadratic Polynomial Model')
```



Notice that the plot of the residuals exhibits a pattern, which indicates that a second-degree polynomial might not be appropriate for modeling this data.

Plot and Calculate Confidence Bounds

Confidence bounds are confidence intervals for a predicted response. The width of the interval indicates the degree of certainty of the fit.

This portion of the example applies `polyfit` and `polyval` to the `census` sample data to produce confidence bounds for a second-order polynomial model.

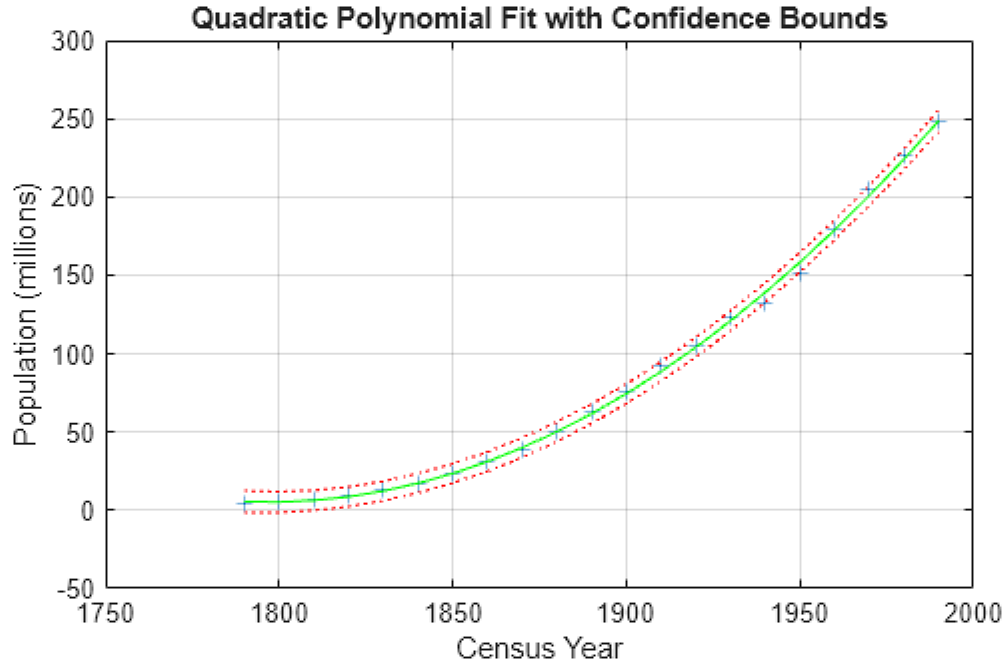
The following code uses an interval of $\pm 2\Delta$, which corresponds to a 95% confidence interval for large samples.

Evaluate the fit and the prediction error estimate (delta).

```
[pop_fit,delta] = polyval(p,cdate,ErrorEst);
```

Plot the data, the fit, and the confidence bounds.

```
plot(cdate,pop,'+',...
     cdate,pop_fit,'g-',...
     cdate,pop_fit+2*delta,'r:',...
     cdate,pop_fit-2*delta,'r:');
xlabel('Census Year');
ylabel('Population (millions)');
title('Quadratic Polynomial Fit with Confidence Bounds')
grid on
```



The 95% interval indicates that you have a 95% chance that a new observation will fall within the bounds.

Time Series Analysis

Time Series Objects and Collections

The recommended way to store time series data is with `timetable`, which has a broad set of supporting functions for preprocessing, restructuring, and analysis. To get started with timetables, see “Create Timetables”.

Some existing code uses these objects, described in this topic:

- `timeseries` — Stores numeric data and time values, as well as the metadata information that includes units, events, data quality, and interpolation method.
- `tscollection` — Stores a collection of `timeseries` objects that share a common time vector, convenient for performing operations on synchronized time series with different units.

Data Samples in `timeseries`

Consider data that consists of three sensor signals: two signals represent the position of an object in meters, and the third represents its velocity in meters/second. `NaN` represents a missing data value.

```
x = [-0.2 -0.3 13;
     -0.1 -0.4 15;
      NaN  2.8 17;
      0.5 0.3 NaN;
     -0.3 -0.1 15];
```

The first two columns of `x` contain quantities with the same units, and you can create a multivariate `timeseries` object to store these two time series.

```
ts_pos = timeseries(x(:,1:2),1:5,"name","Position")
```

```
  timeseries

  Common Properties:
            Name: 'Position'
            Time: [5x1 double]
        TimeInfo: [1x1 tsdata.timemetadata]
            Data: [5x2 double]
        DataInfo: [1x1 tsdata.datametadata]

  More properties, Methods
```

A *data sample* consists of one or more values associated with a specific time in the `timeseries` object. The number of data samples in a time series is the same as the length of the time vector, which is 5 in this example. To find the size of the data sample, use `getdatasamplesize`.

```
getdatasamplesize(ts_pos)
```

```
ans = 1×2

     1     2
```

You can create a second `timeseries` object to store the velocity data.

```
ts_vel = timeseries(x(:,3),1:5,"name","Velocity");
```

If you want to perform operations on ts_pos and ts_vel while keeping them synchronized, group them in a collection. For more information, see Time Series Collections on page 3-4.

Creating Time Series Objects

The sample data in count.dat has 24 rows and three columns. Each column represents hourly vehicle counts at each of three town intersections.

Load the data and create three timeseries objects to store the data collected at each intersection.

```
load count.dat
count1 = timeseries(count(:,1),1:24,"name","Intersection1");
count2 = timeseries(count(:,2),1:24,"name","Intersection2");
count3 = timeseries(count(:,3),1:24,"name","Intersection3");
```

Alternatively, when all time series have the same data units and you want to keep them synchronized during calculations, create a single object.

```
count_ts = timeseries(count,1:24,"name","traffic_counts");
```

Modifying Units and Interpolation Method

By default, a time series has a time vector with units of seconds and a start time of 0 sec, and the series uses linear interpolation.

Modify the time units to be hours for the three time series.

```
count1.TimeInfo.Units = "hours";
count2.TimeInfo.Units = "hours";
count3.TimeInfo.Units = "hours";
```

Change the data units for count1 to cars.

```
count1.DataInfo.Units = "cars";
```

Set the interpolation method for count1 to zero-order hold. The other time series use the default method, linear interpolation.

```
count1.DataInfo.Interpolation = tsdata.interpolation("zoh");
```

View the modified data properties.

```
count1.DataInfo
```

```
  tsdata.datametadata
  Namespace: tsdata

  Common Properties:
             Units: 'cars'
     Interpolation: zoh (tsdata.interpolation)

  More properties, Methods
```

Defining Events

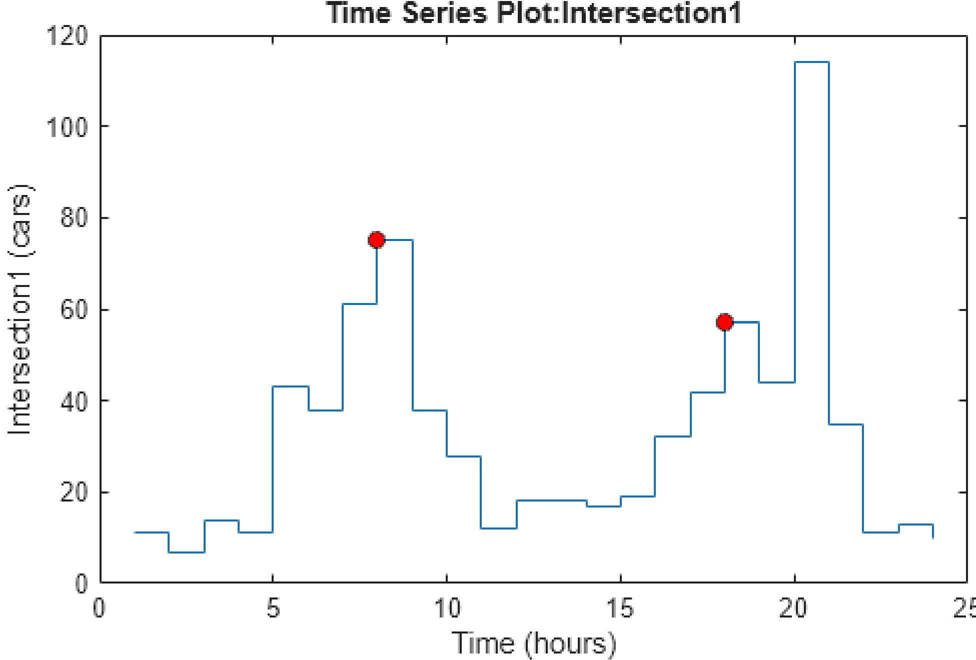
Events mark the data at specific times. Events also provide a convenient way to synchronize multiple time series.

Add two events to each series that mark the times of the AM commute and PM commute.

```
e1 = tsdata.event("AMCommute",8);
e1.Units = "hours";
count1 = addevent(count1,e1);
count2 = addevent(count2,e1);
count3 = addevent(count3,e1);

e2 = tsdata.event("PMCommute",18);
e2.Units = "hours";
count1 = addevent(count1,e2);
count2 = addevent(count2,e2);
count3 = addevent(count3,e2);
```

Plot the first time series. Red circle markers indicate the events.

```
plot(count1)
```



Time Series Collections

A *collection* is a group of synchronized time series. The time vectors of the `timeseries` objects in a collection must match. Each individual time series in a collection is called a *member*. Typically, you use collections for time series that have different data units. In this simple example, all time series have the same units.

```
tsc = tscollection({count1,count2,count3},"name","count_coll")
```

```
Time Series Collection Object: count_coll

Time vector characteristics

      Start time            1 hours
      End time              24 hours

Member Time Series Objects:
```

```
Intersection1
Intersection2
Intersection3
```

Resampling a Collection

A resampling operation is used to either select existing data at specific time values, or to interpolate data at finer intervals. If the new time vector contains time values that did not exist in the previous time vector, the new data values are calculated using the interpolation method associated with each time series.

Resample the time series to include data values every two hours instead of every hour and save it as a new `tscollection` object.

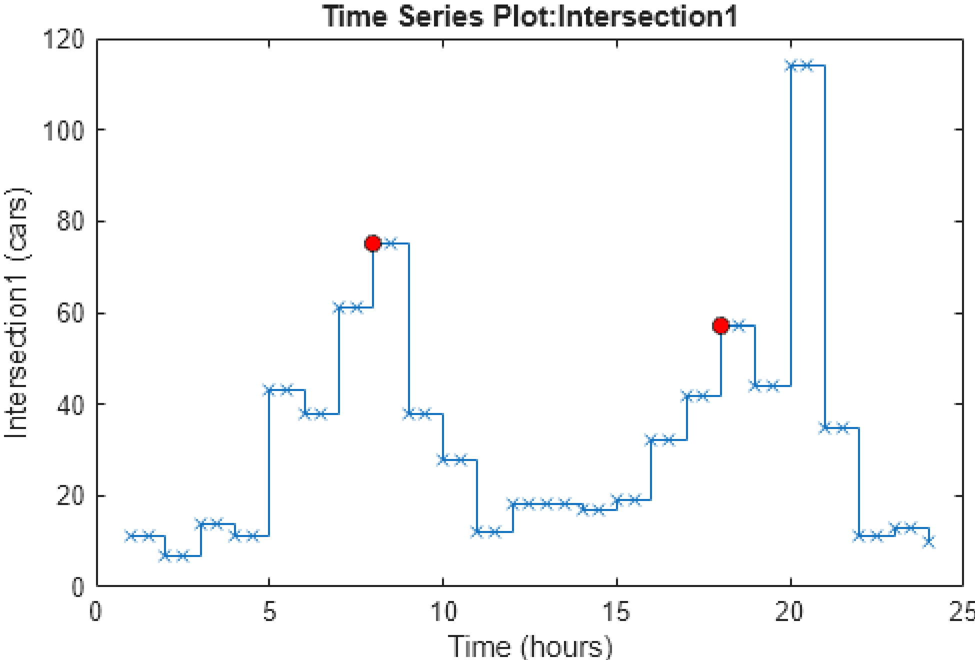
```
tsc1 = resample(tsc,1:2:24);
```

In some cases, you might need a finer sampling of information than you currently have and it is reasonable to obtain it by interpolating data values. For instance, interpolate values at each half-hour mark.

```
tsc1 = resample(tsc,1:0.5:24);
```

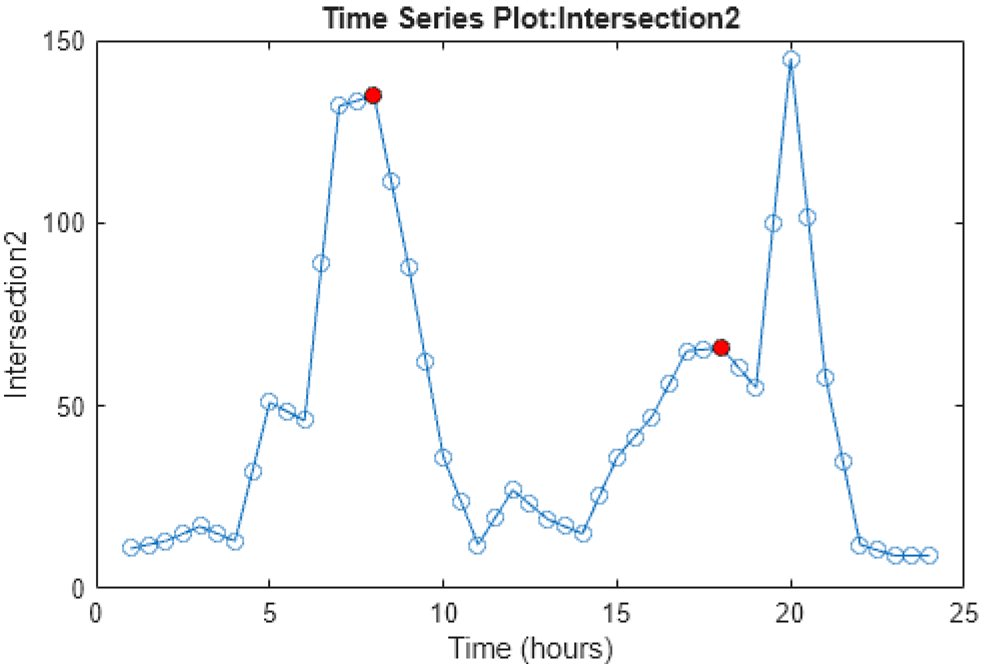
The new data points in `Intersection1` are calculated by using the zero-order hold interpolation method, which holds the value of the previous sample constant. Plot the members of `tsc1` with markers to see the results of interpolating.

```
plot(tsc1.Intersection1,"-x")
```



The new data points in `Intersection2` use linear interpolation, the default method.

```
plot(tsc1.Intersection2,"-o")
```



Adding a Data Sample to a Collection

Add a data sample to the first collection member at 3.25 hours.

```
tsc1 = addsampletocollection(tsc1,"time",3.25,"Intersection1",5);
```

The time series includes values for every half hour, so the new value is the sixth element.

```
tsc1.Intersection1.Data
```

```
ans = 48×1

    11
    11
     7
     7
    14
     5
    14
    11
    11
    43
    43
    38
    38
    61
    61
      ⋮
```

Because you did not specify the data values for `Intersection2` and `Intersection3` in the new sample, the missing values are represented by `NaN` for these members.

```
tsc1.Intersection2.Data
```

```
ans = 48×1

   11.0000
   12.0000
   13.0000
   15.0000
   17.0000
       NaN
   15.0000
   13.0000
   32.0000
   51.0000
   48.5000
   46.0000
   89.0000
  132.0000
  133.5000
      ⋮
```

Handling Missing Data

The `Intersection2` and `Intersection3` members in the `tsc1` collection currently contain missing values at 3.25 hours, represented by `NaN`. Before analyzing this data, you can either remove the missing values or use interpolation to replace them.

For instance, find and remove the data samples that contain `NaN` values. For each missing value in `Intersection2`, the data at that time is removed from *all* members of the collection.

```
tsc2 = delsamplefromcollection(tsc1,"index",...
       find(isnan(tsc1.Intersection2.Data)));
```

Alternatively, replace the `NaN` values in `Intersection2` and `Intersection3` by resampling using interpolation. The default interpolation method for these time series is linear interpolation.

```
tsc1 = resample(tsc1,tsc1.Time);
tsc1.Intersection2.Data
```

```
ans = 48×1

   11.0000
   12.0000
   13.0000
   15.0000
   17.0000
   16.0000
   15.0000
   13.0000
   32.0000
   51.0000
   48.5000
   46.0000
   89.0000
  132.0000
  133.5000
      ⋮
```

Plotting Collection Members

To plot data in a time series collection, plot its members one at a time.

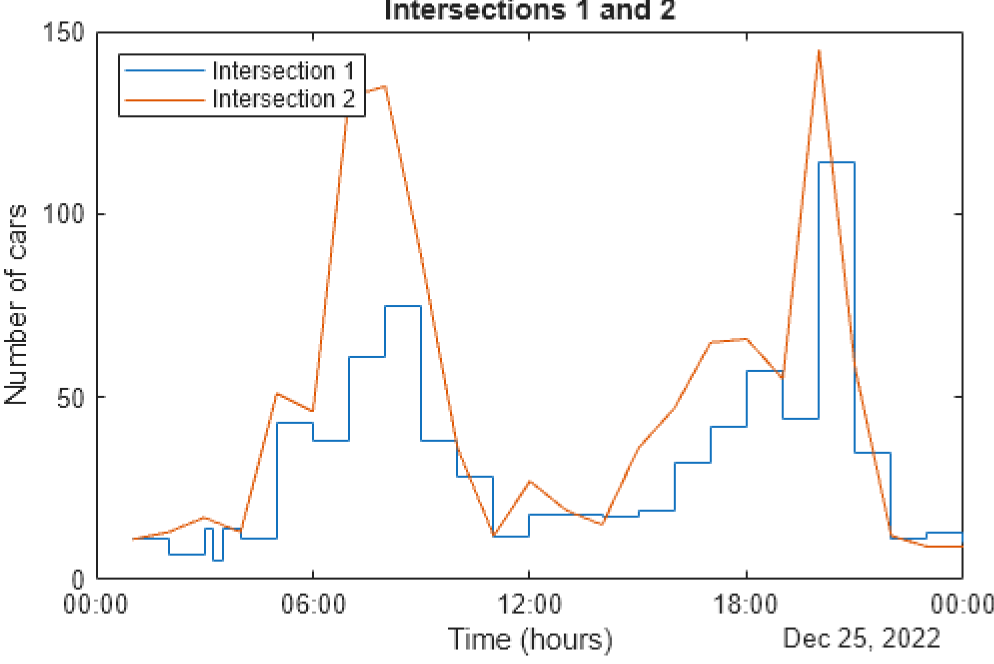
Optionally, to display the time vectors as formatted dates and times, specify the start date. In this case, the time units are hours, so you can specify a display format that shows hours and minutes.

```
tsc1.TimeInfo.StartDate = "25-DEC-2022 00:00:00";
tsc1.TimeInfo.Format = "HH:MM";
```

When you plot a single member of a collection, its time units display on the *x*-axis and its data units display on the *y*-axis. The plot title is displayed as `Time Series Plot:<member name>`.

If you specify `hold on` before plotting multiple members of the collection, no annotations display. The time series `plot` method does not attempt to show labels and titles on held figures because the descriptors for the series can be different. To preserve date formatting, hold after plotting the first member. Update the title and labels to reflect the entire collection.

```
plot(tsc1.Intersection1,"Displayname","Intersection 1")
hold on
plot(tsc1.Intersection2,"Displayname","Intersection 2")

legend("Location","NorthWest")
title("Intersections 1 and 2")
xlabel("Time (hours)")
ylabel("Number of cars")
hold off
```



See Also

`timeseries` | `tscollection` | `tsdata.event`

Manage Experiments

- “Keyboard Shortcuts for Experiment Manager” on page 4-2
- “Run Experiments in Parallel” on page 4-4
- “Offload Experiments as Batch Jobs to a Cluster” on page 4-6
- “Debug General-Purpose Experiments” on page 4-9
- “Experiment with Predator-Prey Equations” on page 4-11
- “Compare Air Resistance Models for Projectile Motion” on page 4-19
- “Convert MATLAB Code into Experiment” on page 4-28

Keyboard Shortcuts for Experiment Manager

Use these keyboard shortcuts when working with Experiment Manager.

Note When you use the keyboard shortcuts on macOS systems:

- Press the **Command** (⌘) key instead of the **Ctrl** key.
- Press the **Option** key instead of the **Alt** key.

Shortcuts for General Navigation

Action	Shortcut
Show the access keys for the toolstrip	**Alt+E**
Move forward through the different areas of the Experiment Manager app, including the toolstrip, the **Experiment Browser** panel, and the experiment definition and results tabs	**Ctrl+F6**
Move backward through the different areas of the Experiment Manager app, including the toolstrip, the **Experiment Browser** panel, and the experiment definition and results tabs	**Ctrl+Shift+F6**
Move forward through the different elements in the start page, the toolstrip, or the experiment definition and results tabs	**Tab**
Move backward through the different elements in the start page, the toolstrip, or the experiment definition and results tabs	**Shift+Tab**
Select the current option in the start page or a menu	**Enter**
Cancel the current action, for example, hide the access keys for the toolstrip or close a menu	**Esc**

Shortcuts for Experiment Browser

Use **Ctrl+F6** or **Ctrl+Shift+F6** to navigate to the **Experiment Browser**. Then press **Tab** twice to focus the currently selected project, experiments, or results. The **Experiment Browser** supports the selection of multiple experiments and results.

Action	Shortcut
Show the experiments in the project or the results for an experiment	Right arrow If the contents of the project or experiment are already visible, pressing the right arrow selects the first experiment in the project or the first result for the experiment.

Action	Shortcut
Hide the experiments in the project or the results for an experiment	Left arrow If the contents of an experiment are already hidden, pressing the left arrow selects the project that contains the experiment.
Move down and select the next item in the browser	Down arrow
Move up and select the previous item in the browser	Up arrow
Select multiple items in the browser	Navigate to the next item in the browser that you want to select by pressing **Ctrl+**up arrow or **Ctrl +**down arrow. Then press the space bar to select.
Select a range of contiguous items in the browser	**Shift+**down arrow or **Shift+**up arrow
Select all experiments	**Ctrl+A**
Rename an experiment or result	**F2** or **Enter**
Delete selected experiments or results	**Delete** • Your selection must contain only experiments or only results. • If you delete an experiment, Experiment Manager also deletes the results contained in the experiment.

Shortcuts for Results Table

Use **Ctrl+F6** or **Ctrl+Shift+F6** to navigate to the experiment results tab. Then press **Tab** to navigate to the results table.

Action	Shortcut
Move to the next trial in the table	Down arrow
Move to the previous trial in the table	Up arrow
Move to the next column in the table	Right arrow
Move to the previous column in the table	Left arrow

See Also

Apps
Experiment Manager

Run Experiments in Parallel

By default, Experiment Manager runs one trial of your experiment at a time on a single CPU. If you have Parallel Computing Toolbox™, you can configure your experiment to run multiple trials at the same time or to run a single trial at a time on multiple GPUs, on a cluster, or in the cloud.

Run Multiple Simultaneous Trials

Run multiple trials of your experiment at the same time using one parallel worker for each trial:

1. Set up your parallel environment as described in "Set Up Parallel Environment" on page 4-4.
2. On the Experiment Manager toolstrip, set **Mode** to `Simultaneous`.
3. Click **Run** .

Experiment Manager runs as many simultaneous trials as there are workers in your parallel pool. All other trials in your experiment are queued for later evaluation.

Tip Load data for your experiment from a location that is accessible to all your parallel workers. For example, store your data outside the project and access the data by using an absolute path. Alternatively, create a datastore object that can access the data on another machine by setting up the `AlternateFileSystemRoots` property of the datastore. For more information, see "Set Up Datastore for Processing on Different Machines or Clusters".

Run Single Trial on Multiple Workers

To run a single trial of your experiment at a time on multiple parallel workers:

1. In your experiment function, set up your parallel environment as described in "Set Up Parallel Environment" on page 4-4. Then, use `spmd`, `parfor`, or `parfeval` to define the parallel algorithm for your experiment. For more information, see "Choose Between spmd, parfor, and parfeval" (Parallel Computing Toolbox).
2. On the Experiment Manager toolstrip, set **Mode** to `Sequential`.
3. Click **Run** .

Set Up Parallel Environment

Run on Multiple GPUs

If you have multiple GPUs, parallel execution typically increases the speed of your experiment. Using a GPU requires Parallel Computing Toolbox and a supported GPU device. For more information, see "GPU Computing Requirements" (Parallel Computing Toolbox). To determine whether a usable GPU is available, call the `canUseGPU` function.

For best results, before you run your experiment, create a parallel pool with as many workers as GPUs. Otherwise, multiple workers share the same GPU, so you do not get the desired computational speed-up and you increase the chance that the GPUs run out of memory. You can check the number of available GPUs by using the `gpuDeviceCount` function.

```
numGPUs = gpuDeviceCount("available");
parpool(numGPUs)
```

Run on Cluster or in Cloud

If your experiments take a long time to run on your local machine, you can improve performance by using a computer cluster on your onsite network or by renting high-performance GPUs in the cloud. After you complete the initial setup, you can run your experiments with minimal changes to your code. Working on a cluster or in the cloud requires MATLAB Parallel Server™. For more information, see “Scale Up from Desktop to Cluster” (Parallel Computing Toolbox).

See Also

Apps
Experiment Manager

Functions
`spmd` | `parfor` | `parfeval` | `gpuDeviceCount` | `parpool`

Related Examples

- “Choose Between spmd, parfor, and parfeval” (Parallel Computing Toolbox)

Offload Experiments as Batch Jobs to a Cluster

By default, Experiment Manager runs your experiments interactively, so you can monitor the progress of each trial by inspecting the results table. However, running an experiment interactively limits your access to MATLAB functionality.

If you have Parallel Computing Toolbox and MATLAB Parallel Server, you can send your experiment as a batch job to a remote cluster. While the experiment is running in the cluster, you can:

- Run another experiment interactively or start another batch job using the same experiment, a different experiment in the same project, or an experiment in a different project.
- Close the Experiment Manager app and continue using MATLAB.
- Close your MATLAB session.

If you only have Parallel Computing Toolbox, you can use a local cluster profile to develop and test your experiments on your client machine instead of running them on a network cluster. If you close your MATLAB session, any batch jobs using the local cluster profile also stop immediately.

Create Batch Job on Cluster

To start a batch job for your experiment:

1. Configure your experiment.

   Tip Load data for your experiment from a location that is accessible to all your parallel workers. For example, store your data outside the project and access the data by using an absolute path. Alternatively, create a datastore object that can access the data on another machine by setting up the `AlternateFileSystemRoots` property of the datastore. For more information, see “Set Up Datastore for Processing on Different Machines or Clusters”.

   ---
2. In the Experiment Manager toolstrip, under **Execution**, use the **Mode** list to specify an execution mode:
   - To run one trial of the experiment at a time, select `Batch Sequential`.
   - To run multiple trials at the same time, select `Batch Simultaneous`.
3. Use the **Cluster** list to select a cluster profile to use for your batch job. To create and manage cluster profiles, open the Cluster Profile Manager. For more information, see “Discover Clusters and Use Cluster Profiles” (Parallel Computing Toolbox).
4. In the **Pool Size** field, enter the number of workers for your batch job.
   - If **Mode** is `Batch Sequential`, use this field to configure the number of parallel workers that collaborate on each trial of the experiment. If you set the pool size to `0`, the experiment runs on a single worker.
   - If **Mode** is `Batch Simultaneous`, use this field to specify the number of trials that the cluster runs at the same time.

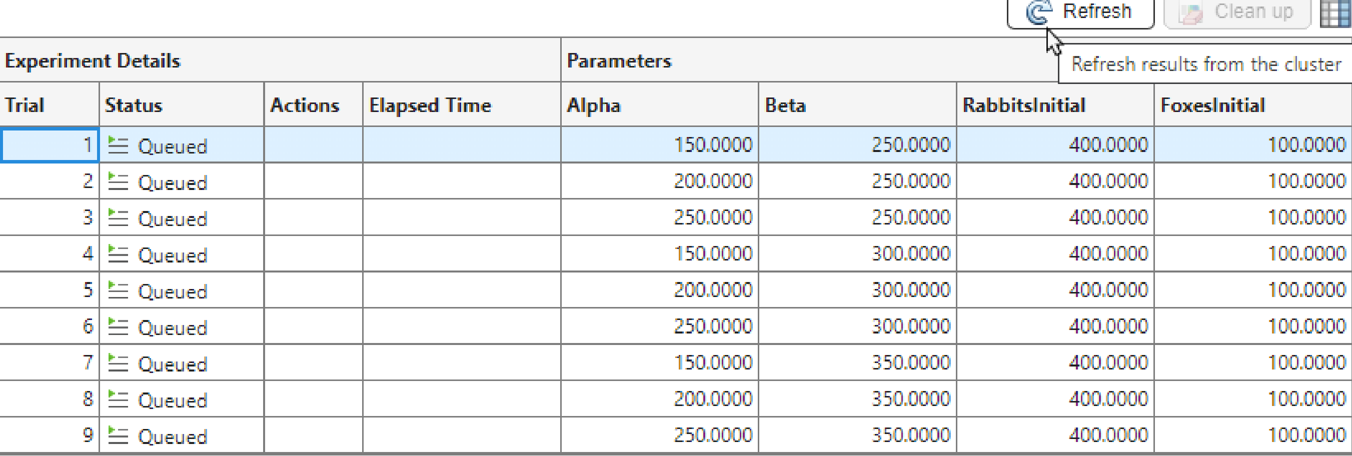
   Because Experiment Manager uses an additional worker to run the batch job, the cluster must have at least one more worker available than the number you specify in the **Pool Size** field. For example, if you specify a pool size of `2`, the cluster must have at least three workers available: two workers for the experiment and an additional worker to run the batch job. For more information, see “Run Batch Job with Parallel Pool” (Parallel Computing Toolbox).

5 Click **Run** ▷. Experiment Manager uses the `batch` function to run the experiment in the specified cluster.

While the batch job runs your experiment, you can close Experiment Manager and recover the results later.

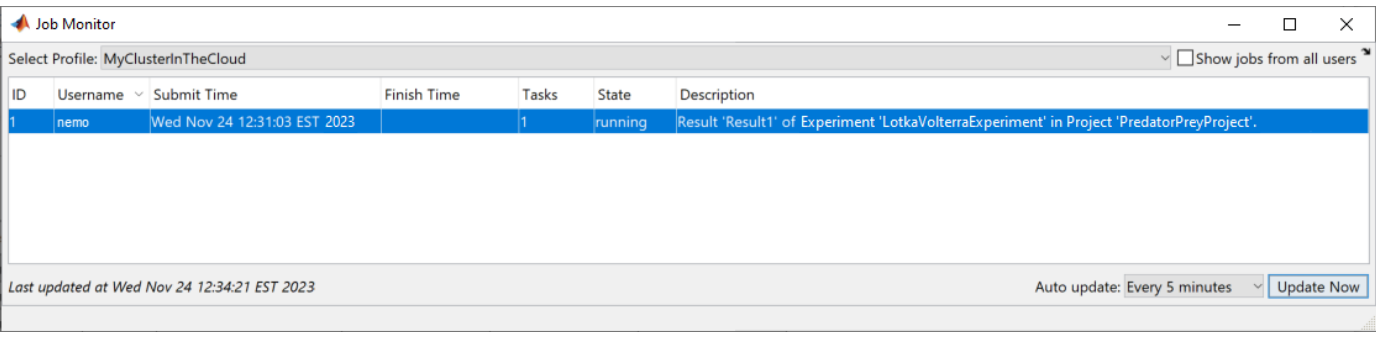
Track Progress of Batch Job

When you run a batch job for an experiment, Experiment Manager does not continually communicate with the cluster to update the values in the results table and save the visualizations for your experiment. Instead, you retrieve this information from the cluster by clicking the **Refresh** button above the results table.



Experiment Details				Parameters			
Trial	Status	Actions	Elapsed Time	Alpha	Beta	RabbitsInitial	FoxesInitial
1	Queued			150.0000	250.0000	400.0000	100.0000
2	Queued			200.0000	250.0000	400.0000	100.0000
3	Queued			250.0000	250.0000	400.0000	100.0000
4	Queued			150.0000	300.0000	400.0000	100.0000
5	Queued			200.0000	300.0000	400.0000	100.0000
6	Queued			250.0000	300.0000	400.0000	100.0000
7	Queued			150.0000	350.0000	400.0000	100.0000
8	Queued			200.0000	350.0000	400.0000	100.0000
9	Queued			250.0000	350.0000	400.0000	100.0000

To monitor batch jobs without opening the Experiment Manager app, use the Job Monitor. The Job Monitor tells you whether your batch job is queued, running, or finished.



Note Using the Job Monitor to cancel or delete jobs that you create with Experiment Manager can lead to unexpected behavior. Instead, cancel and delete these batch jobs by using Experiment Manager.

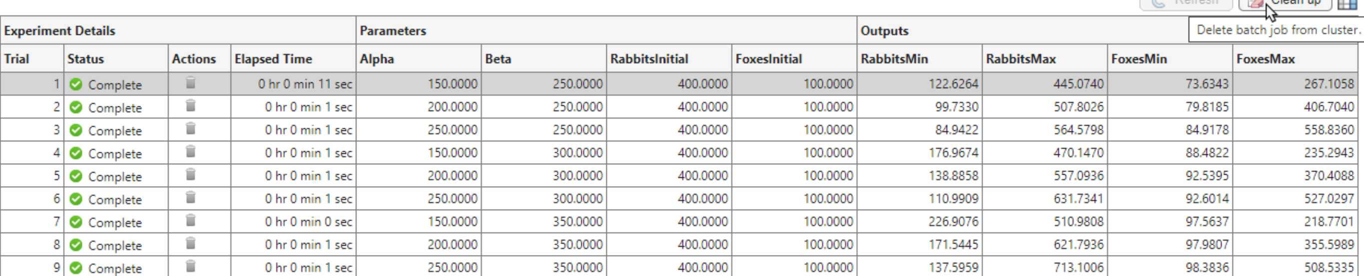
Cancel Batch Job

To cancel a batch job running an experiment, in the Experiment Manager toolstrip, click **Cancel** . Experiment Manager marks any running and queued trials as `Canceled` and discards their results.

Batch execution does not support stopping, canceling, or restarting individual trials of an experiment.

Delete Batch Job

To avoid consuming resources unnecessarily, delete the job from the cluster by clicking the **Clean up** button above the results table.



Experiment Details				Parameters				Outputs			
Trial	Status	Actions	Elapsed Time	Alpha	Beta	RabbitsInitial	FoxesInitial	RabbitsMin	RabbitsMax	FoxesMin	FoxesMax
1	Complete		0 hr 0 min 11 sec	150.0000	250.0000	400.0000	100.0000	122.6264	445.0740	73.6343	267.1058
2	Complete		0 hr 0 min 1 sec	200.0000	250.0000	400.0000	100.0000	99.7330	507.8026	79.8185	406.7040
3	Complete		0 hr 0 min 1 sec	250.0000	250.0000	400.0000	100.0000	84.9422	564.5798	84.9178	558.8360
4	Complete		0 hr 0 min 1 sec	150.0000	300.0000	400.0000	100.0000	176.9674	470.1470	88.4822	235.2943
5	Complete		0 hr 0 min 1 sec	200.0000	300.0000	400.0000	100.0000	138.8858	557.0936	92.5395	370.4088
6	Complete		0 hr 0 min 1 sec	250.0000	300.0000	400.0000	100.0000	110.9909	631.7341	92.6014	527.0297
7	Complete		0 hr 0 min 0 sec	150.0000	350.0000	400.0000	100.0000	226.9076	510.9808	97.5637	218.7701
8	Complete		0 hr 0 min 1 sec	200.0000	350.0000	400.0000	100.0000	171.5445	621.7936	97.9807	355.5989
9	Complete		0 hr 0 min 1 sec	250.0000	350.0000	400.0000	100.0000	137.5959	713.1006	98.3836	508.5335

See Also

Apps
Experiment Manager | Job Monitor

Functions
`batch`

Related Examples

- “Run Batch Parallel Jobs” (Parallel Computing Toolbox)
- “Discover Clusters and Use Cluster Profiles” (Parallel Computing Toolbox)
- “Use Parallel Computing Toolbox with Cloud Center Cluster in MATLAB Online” (Parallel Computing Toolbox)

Debug General-Purpose Experiments

In Experiment Manager, diagnose problems in your experiment function by stepping through your code line-by-line and examining the values of your variables.

Start Debugging Session

You can debug your code before or after you run the experiment.

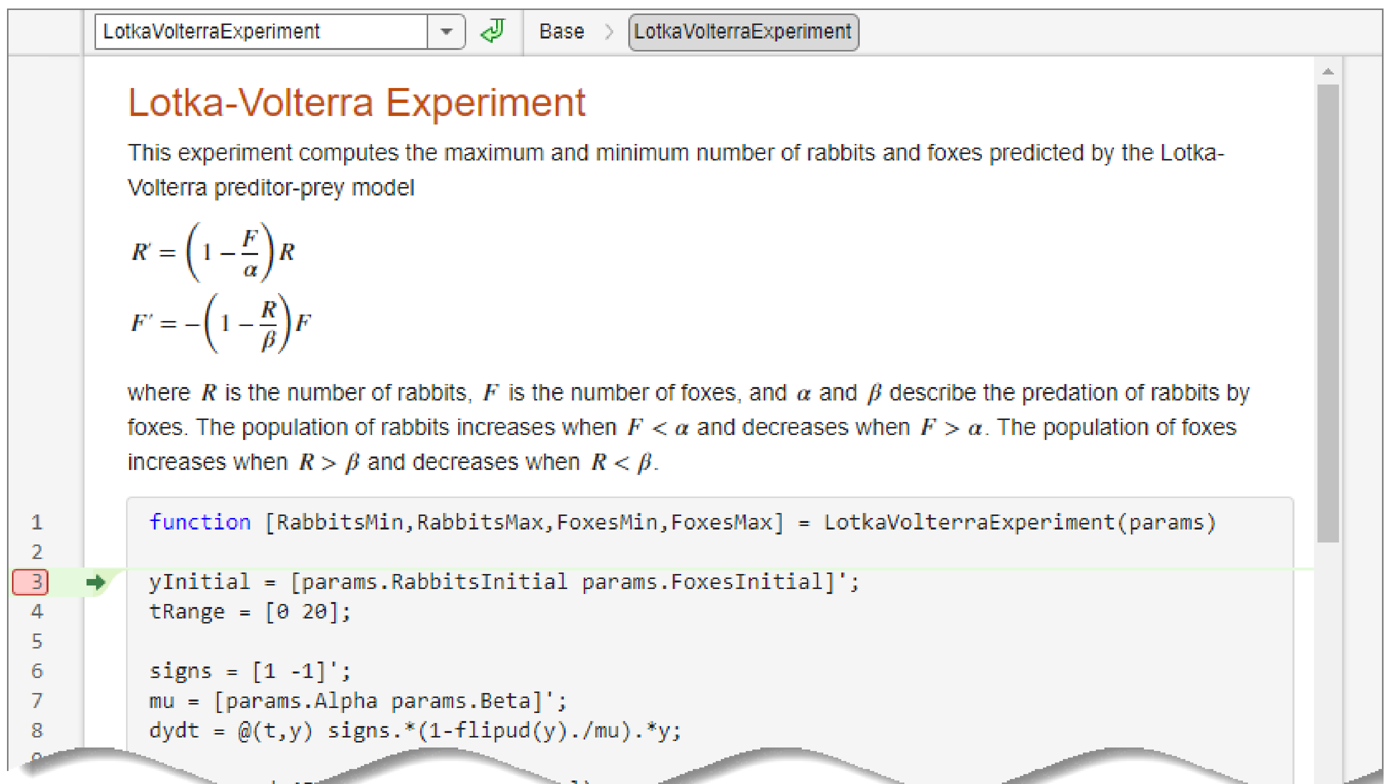
To debug your code before you run the experiment:

1. Open the experiment.
2. In the Experiment Manager toolstrip, select **Run** > **Debug**.
3. In dialog box, specify the parameter values for your experiment.
4. Click **Start**.

To debug your code after you run the experiment:

1. Open the results for the experiment.
2. In the results table, select a trial to debug. To ensure reproducibility, Experiment Manager reuses the parameter values and the random seed saved for this trial.
3. Right-click the trial and select **Debug**.

Experiment Manager opens the experiment function in MATLAB Editor, places a breakpoint in the first line of code, and runs the function.



Debug Experiment Function

When you debug your experiment function, MATLAB pauses at each line of code that has a breakpoint. You can add other breakpoints to your function, view the values of your variables, step through the code line-by-line, or continue to the next breakpoint. For more information, see “Debug MATLAB Code Files”.

Verify Your Results

After your function runs to completion, verify your results by examining the parameters and output values stored in workspace variables, where `functionName` is the name of the experiment function.

- *functionName*`_params` — A structure with fields from the Experiment Manager parameter table
- *functionName*`_output` — A cell array that contains the output values returned by the experiment function

See Also

Apps
Experiment Manager

More About

- “Set Breakpoints”
- “Debug MATLAB Code Files”

Experiment with Predator-Prey Equations

This example shows how to use Experiment Manager to explore how different coefficient values and initial values affect the solution of a system of differential equations.

The Lotka-Volterra predator-prey model is a system of first-order ordinary differential equations that describes the relationship between two competing populations. For example, these equations describe the populations of rabbits and foxes with respect to time:

$$R' = \left(1 - \frac{F}{\alpha}\right)R$$

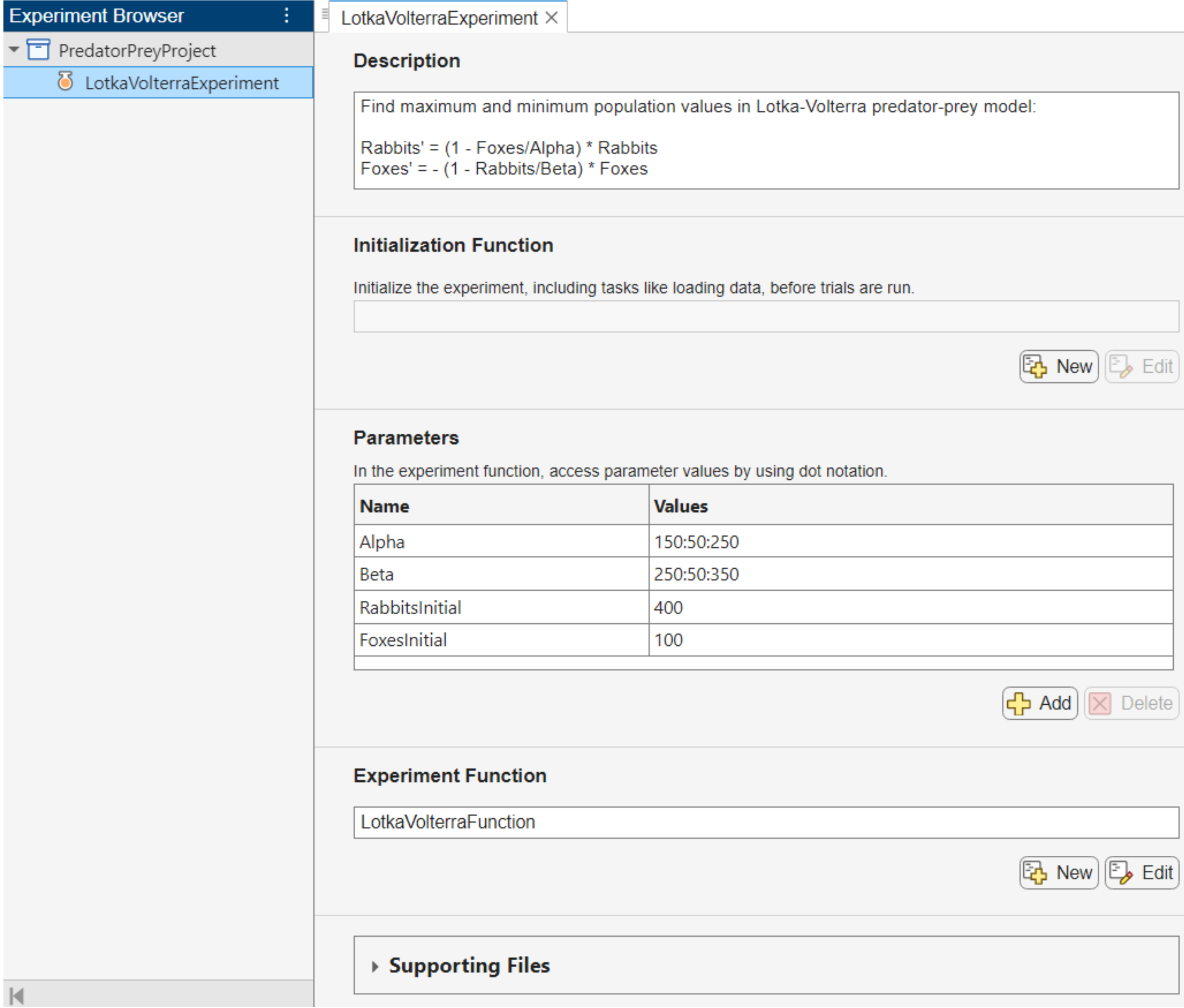
$$F' = -\left(1 - \frac{R}{\beta}\right)F$$

In these equations, R is the number of rabbits and F is the number of foxes. The coefficients α and β describe the predation of rabbits by foxes. The population of rabbits increases when $F < \alpha$ and decreases when $F > \alpha$. The population of foxes increases when $R > \beta$ and decreases when $R < \beta$. For more information, see [1].

In this experiment, you observe the effect on the predicted populations of rabbits and foxes when you use different values for the coefficients α and β and the initial values of R and F.

Open Experiment

First, open the example. Experiment Manager loads a project with a preconfigured experiment that you can inspect and run. To open the experiment, in the **Experiment Browser** pane, double-click **LotkaVolterraExperiment**.



The experiment consists of a description, a table of parameters, and an experiment function.

The **Description** field contains a textual description of the experiment. For this example, the description is:

```
Find maximum and minimum population values in Lotka-Volterra predator-prey model:

Rabbits' = (1 - Foxes/Alpha) * Rabbits
Foxes' = -(1 - Rabbits/Beta) * Foxes
```

The **Parameters** section specifies the parameter values to use for the experiment. Experiment Manager runs multiple trials of your experiment using a different combination of parameters for each trial. This example uses the parameters `Alpha` and `Beta` to specify the values of the coefficients of the Lotka-Volterra equations and the parameters `RabbitsInitial` and `FoxesInitial` to specify the initial population of rabbits and foxes.

The **Experiment Function** section specifies the function `LotkaVolterraFunction`, which defines the procedure for the experiment. To open this function in MATLAB® Editor, click **Edit**. The code for the function also appears in Experiment Function on page 4-16. The input to the experiment function is a structure called `params` with fields from the parameter table. The function uses dot notation to extract the parameter values from this structure and to set up the initial conditions and differential equations for the predator-prey problem.

```
yInitial = [params.RabbitsInitial params.FoxesInitial]';
tRange = [0 20];

signs = [1 -1]';
mu = [params.Alpha params.Beta]';
dydt = @(t,y) signs.*(1-flipud(y)./mu).*y;
```

Next, the experiment function calls `ode45` to solve the differential equations and to compute the maximum and minimum number of predicted rabbits and foxes.

```
[t,y] = ode45(dydt,tRange,yInitial);

RabbitsMin = min(y(:,1));
RabbitsMax = max(y(:,1));
FoxesMin = min(y(:,2));
FoxesMax = max(y(:,2));
```

Finally, the experiment function plots the populations of rabbits and foxes against time and against each other. When you run the experiment, Experiment Manager adds two buttons to the **Review Results** gallery in the toolstrip so you can display these figures. The `Name` property of each figure specifies the name of the corresponding button in the toolstrip.

```
figure(Name="Population Size");
plot(t,y)
title("Population v. Time")
xlabel("Time")
ylabel("Population")
legend("Rabbits","Foxes")

figure(Name="Phase Plane");
hold on
plot(y(:,1),y(:,2))
plot([mu(2),mu(2)],[FoxesMin,FoxesMax],":r");
plot([RabbitsMin,RabbitsMax],[mu(1),mu(1)],":r");
title("Foxes v. Rabbits")
xlabel("Rabbits")
ylabel("Foxes")
hold off
```

Run Experiment

On the Experiment Manager toolstrip, click **Run**. Experiment Manager runs the experiment function nine times, each time using a different combination of parameter values. A table of results displays the output values for each trial.

Experiment Details				Parameters				Outputs			
Trial	Status	Actions	Elapsed Time	Alpha	Beta	RabbitsInitial	FoxesInitial	RabbitsMin	RabbitsMax	FoxesMin	FoxesMax
1	Complete		0 hr 0 min 9 sec	150.0000	250.0000	400.0000	100.0000	122.6264	445.0740	73.6343	267.1058
2	Complete		0 hr 0 min 1 sec	200.0000	250.0000	400.0000	100.0000	99.7330	507.8026	79.8185	406.7040
3	Complete		0 hr 0 min 1 sec	250.0000	250.0000	400.0000	100.0000	84.9422	564.5798	84.9178	558.8360
4	Complete		0 hr 0 min 1 sec	150.0000	300.0000	400.0000	100.0000	176.9674	470.1470	88.4822	235.2943
5	Complete		0 hr 0 min 1 sec	200.0000	300.0000	400.0000	100.0000	138.8858	557.0936	92.5395	370.4088
6	Complete		0 hr 0 min 1 sec	250.0000	300.0000	400.0000	100.0000	110.9909	631.7341	92.6014	527.0297
7	Complete		0 hr 0 min 0 sec	150.0000	350.0000	400.0000	100.0000	226.9076	510.9808	97.5637	218.7701
8	Complete		0 hr 0 min 1 sec	200.0000	350.0000	400.0000	100.0000	171.5445	621.7936	97.9807	355.5989
9	Complete		0 hr 0 min 0 sec	250.0000	350.0000	400.0000	100.0000	137.5959	713.1006	98.3836	508.5335

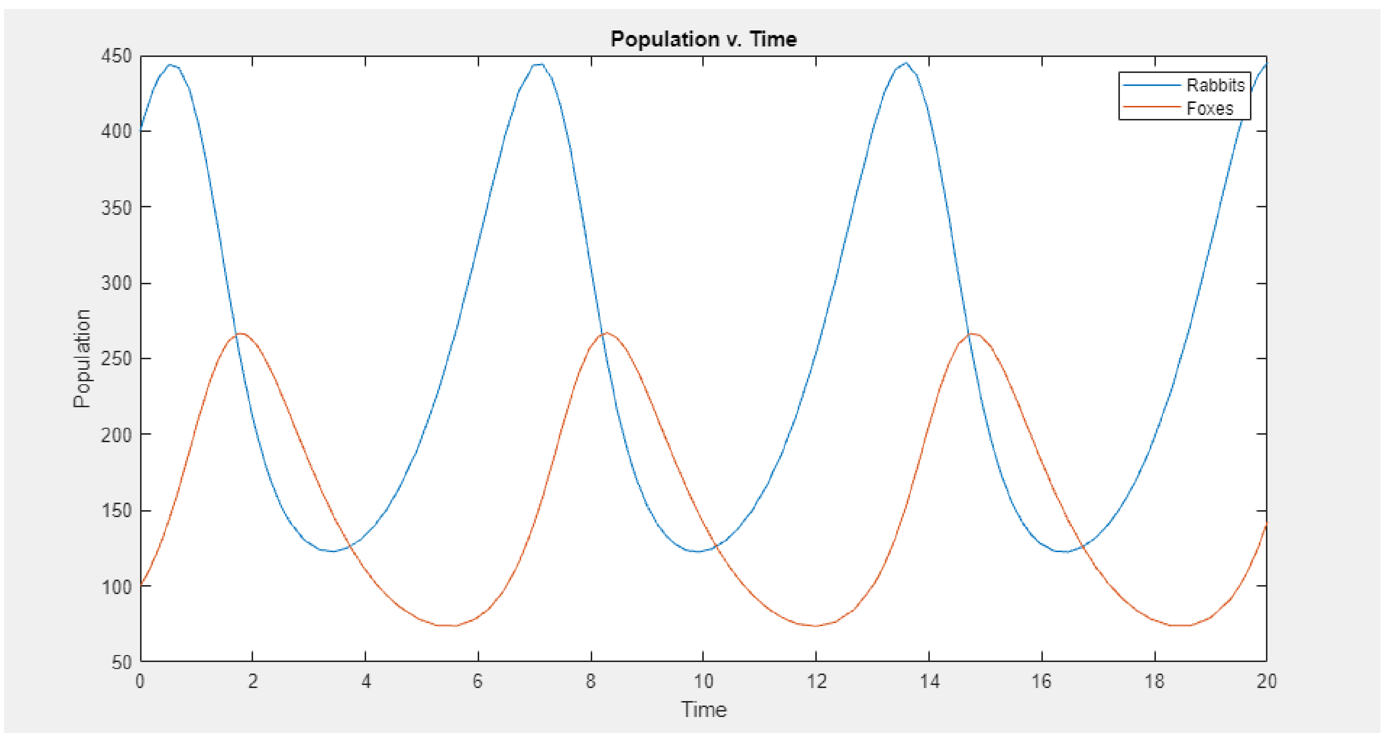
Evaluate Results

To investigate how changing a parameter value affects the predicted populations of rabbits and foxes, you can sort the results table using one of the output values. For example, to find the trial with the largest rabbit population:

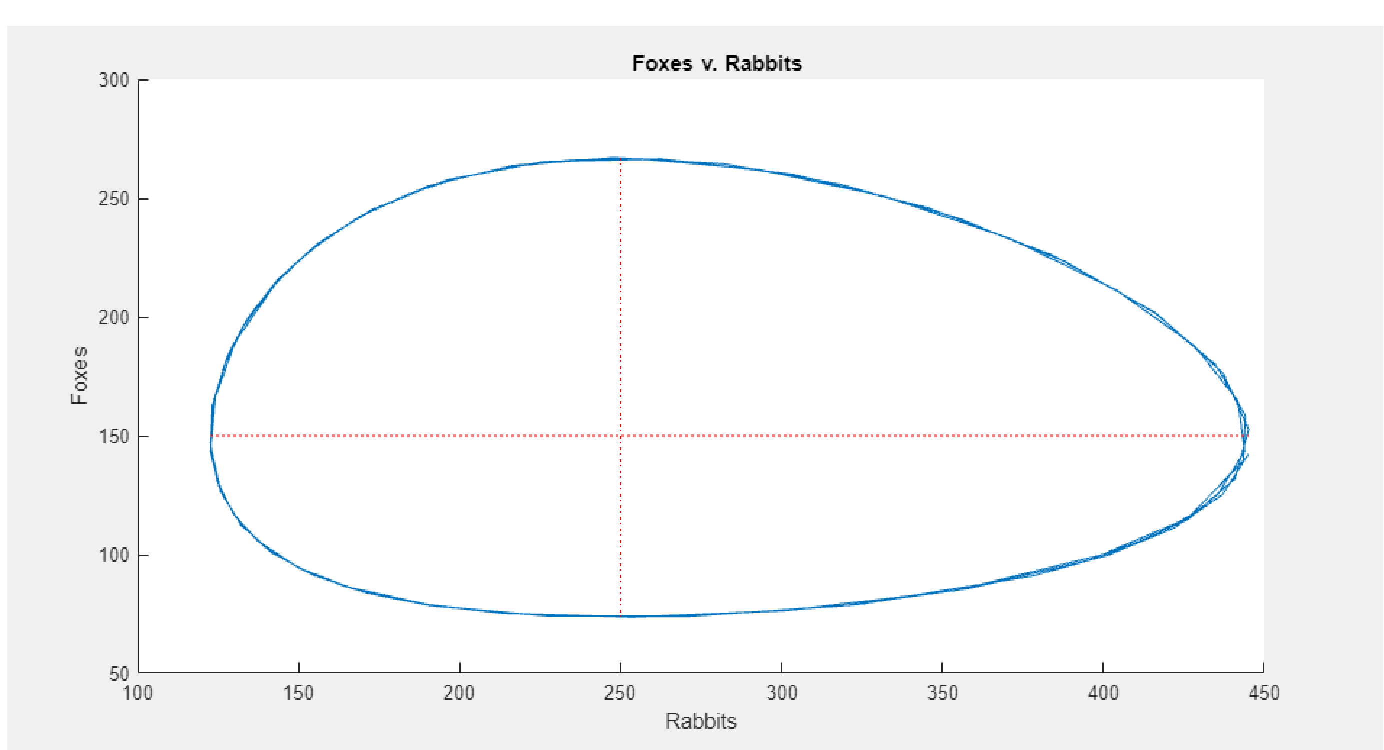
1. Point to the header of the **RabbitsMax** column.
2. Click the triangle icon.
3. Select **Sort in Descending Order**.

Experiment Details				Parameters				Outputs			
Trial	Status	Actions	Elapsed Time	Alpha	Beta	RabbitsInitial	FoxesInitial	RabbitsMin	RabbitsMax	FoxesMin	FoxesMax
9	Complete		0 hr 0 min 1 sec	250.0000	350.0000	400.0000	100.0000	137.5959	713.1006	98.3836	508.5335
6	Complete		0 hr 0 min 1 sec	250.0000	300.0000	400.0000	100.0000	110.9909	631.7341	92.6014	527.0297
8	Complete		0 hr 0 min 2 sec	200.0000	350.0000	400.0000	100.0000	171.5445	621.7936	97.9807	355.5989
3	Complete		0 hr 0 min 1 sec	250.0000	250.0000	400.0000	100.0000	84.9422	564.5798	84.9178	558.8360
5	Complete		0 hr 0 min 1 sec	200.0000	300.0000	400.0000	100.0000	138.8858	557.0936	92.5395	370.4088
7	Complete		0 hr 0 min 1 sec	150.0000	350.0000	400.0000	100.0000	226.9076	510.9808	97.5637	218.7701
2	Complete		0 hr 0 min 2 sec	200.0000	250.0000	400.0000	100.0000	99.7330	507.8026	79.8185	406.7040
4	Complete		0 hr 0 min 2 sec	150.0000	300.0000	400.0000	100.0000	176.9674	470.1470	88.4822	235.2943
1	Complete		0 hr 0 min 9 sec	150.0000	250.0000	400.0000	100.0000	122.6264	445.0740	73.6343	267.1058

You can also select a trial in the results table and inspect the visualizations created by the experiment function. To see the predicted populations plotted as functions of time, on the Experiment Manager toolstrip, under **Review Results**, click **Population Size**. This plot shows the two populations oscillating, with the maximum and minimum values in the rabbit population preceding the maximum and minimum values in the fox population.



To see the populations plotted against each other, click **Phase Plane**. The phase plane plot shows the cyclic relationship between the populations.



To perform additional computations on your results, you can export the results table to the MATLAB workspace as a nested table array. For example, to compare the peak-to-peak amplitudes of the populations over all the trials in your experiment:

1. On the Experiment Manager toolstrip, click **Export**.
2. In the dialog window, enter the name of a workspace variable for the exported table. The default name is `resultsTable`.
3. In the MATLAB Command Window, use the exported table as the input to the function `populationAmplitudes`:

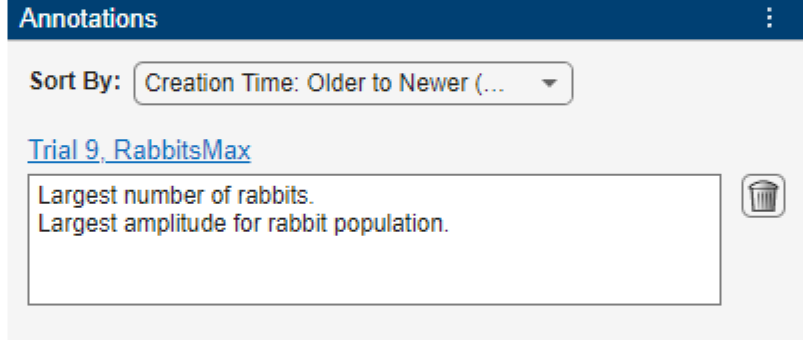
```
populationAmplitudes(resultsTable)
```

To view the code for this function, see Compare Population Amplitudes on page 4-17. The function displays a summary of the maximum, minimum, and mean peak-to-peak population amplitudes for rabbits and foxes over all the trials in your experiment.

```
******************************************

Population of rabbits:
Maximum amplitude: 575.5 (Trial 9)
Minimum amplitude: 284.1 (Trial 7)
Mean amplitude: 416.9

Population of foxes:
Maximum amplitude: 473.9 (Trial 3)
Minimum amplitude: 121.2 (Trial 7)
Mean amplitude: 293.6

******************************************
```

To record observations about the results of your experiment, add an annotation:

1. In the results table, right-click the **RabbitsMax** cell for trial 9.
2. Select **Add Annotation**.
3. In the **Annotations** pane, enter your observations in the text box.



Experiment Function

This function solves the Lotka-Volterra equations, plots the predicted populations of rabbits and foxes over time and against each other, and returns the maximum and minimum number of rabbits and foxes. The input argument `params` is a structure with fields from the parameter table.

```
function [RabbitsMin,RabbitsMax,FoxesMin,FoxesMax] = LotkaVolterraFunction(params)
```

```
yInitial = [params.RabbitsInitial params.FoxesInitial]';
tRange = [0 20];

signs = [1 -1]';
mu = [params.Alpha params.Beta]';
dydt = @(t,y) signs.*(1-flipud(y)./mu).*y;

[t,y] = ode45(dydt,tRange,yInitial);

RabbitsMin = min(y(:,1));
RabbitsMax = max(y(:,1));
FoxesMin = min(y(:,2));
FoxesMax = max(y(:,2));

figure(Name="Population Size");
plot(t,y)
title("Population v. Time")
xlabel("Time")
ylabel("Population")
legend("Rabbits","Foxes")

figure(Name="Phase Plane");
hold on
plot(y(:,1),y(:,2))
plot([mu(2),mu(2)],[FoxesMin,FoxesMax],":r");
plot([RabbitsMin,RabbitsMax],[mu(1),mu(1)],":r");
title("Foxes v. Rabbits")
xlabel("Rabbits")
ylabel("Foxes")
hold off

end
```

Compute Population Amplitudes

This function extracts the maximum and minimum number of rabbits and foxes for each trial from the exported results table `results`. Then, the function computes the maximum, minimum, and mean peak-to-peak population amplitudes over all trials.

```
function populationAmplitudes(results)

results = splitvars(results);
RabbitsAmplitude = results.RabbitsMax - results.RabbitsMin;
FoxesAmplitude = results.FoxesMax - results.FoxesMin;

[maxRabbitsAmplitude,maxRabbitsAmplitudeTrial] = max(RabbitsAmplitude);
[minRabbitsAmplitude,minRabbitsAmplitudgeTrial] = min(RabbitsAmplitude);
meanRabbitsAmplitude = mean(RabbitsAmplitude);

[maxFoxesAmplitude,maxFoxesAmplitudeTrial] = max(FoxesAmplitude);
[minFoxesAmplitude,minFoxesAmplitudeTrial] = min(FoxesAmplitude);
meanFoxesAmplitude = mean(FoxesAmplitude);

fprintf("\n*******************************************\n\n");
fprintf("Population of rabbits:\n");
fprintf("Maximum amplitude: %.1f (Trial %d)\n", ...
    maxRabbitsAmplitude,maxRabbitsAmplitudeTrial);
fprintf("Minimum amplitude: %.1f (Trial %d)\n", ...
```

```
    minRabbitsAmplitude,minRabbitsAmplitudgeTrial);
fprintf("Mean amplitude: %.1f \n\n",meanRabbitsAmplitude);
fprintf("Population of foxes:\n");
fprintf("Maximum amplitude: %.1f (Trial %d)\n", ...
    maxFoxesAmplitude,maxFoxesAmplitudeTrial);
fprintf("Minimum amplitude: %.1f (Trial %d)\n", ...
    minFoxesAmplitude,minFoxesAmplitudeTrial);
fprintf("Mean amplitude: %.1f \n",meanFoxesAmplitude);
fprintf("\n*****************************************\n\n");

end
```

References

[1] Moler, Cleve. "Predator-Prey Model." In *Experiments with MATLAB*. Natick, MA: The MathWorks®, 2011. mathworks.com/moler/exm.html.

See Also

Apps
Experiment Manager

Functions
`ode45` | `table` | `splitvars`

Related Examples

- “Solve Predator-Prey Equations”
- “Compare Air Resistance Models for Projectile Motion” on page 4-19

Compare Air Resistance Models for Projectile Motion

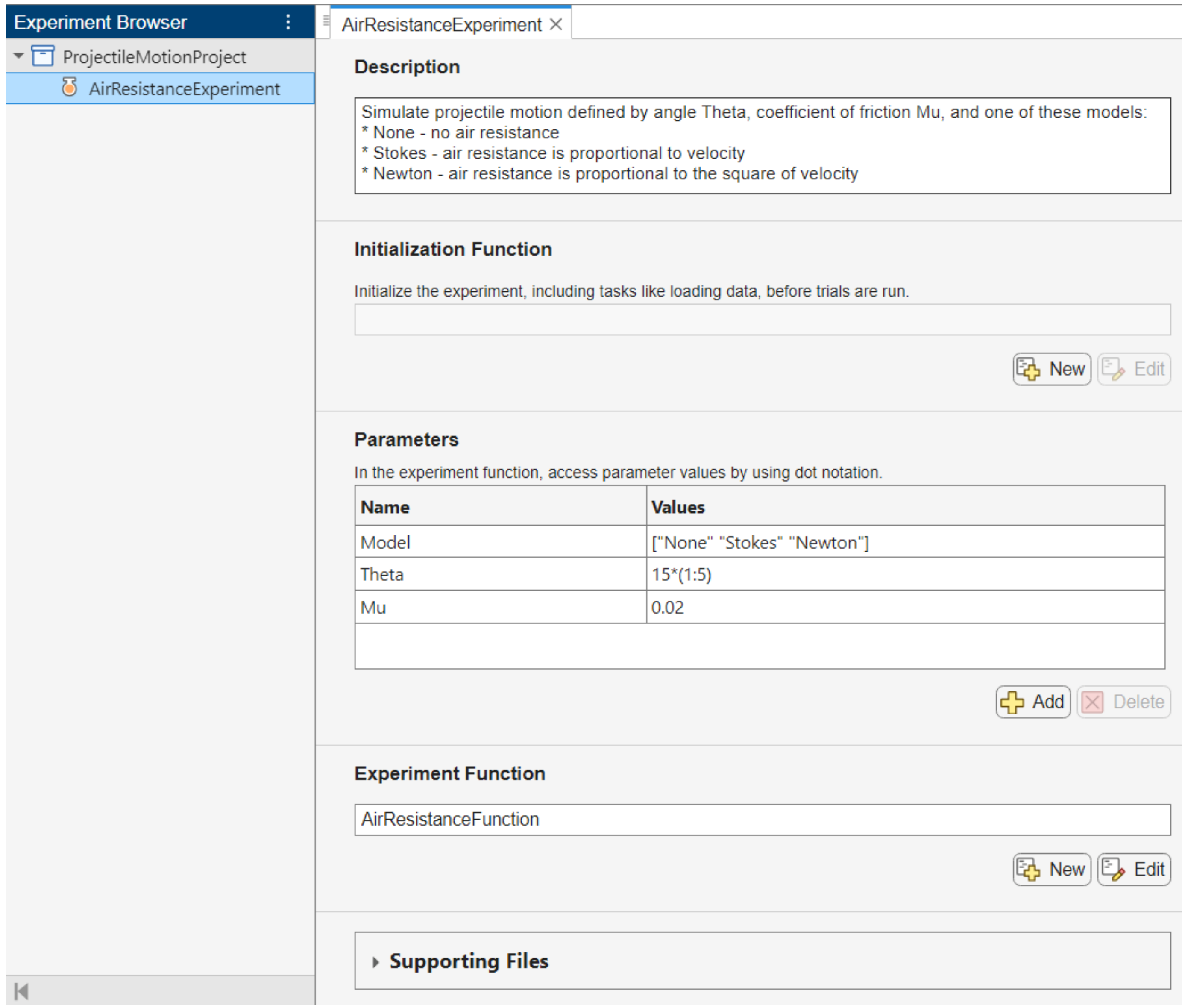
This example shows how to use Experiment Manager to compare the effects of air resistance on the trajectory of a projectile assuming one of these models:

- No air resistance — The motion of the projectile depends only on gravity.
- Stokes drag — Air resistance is proportional to velocity.
- Newton drag — Air resistance is proportional to the square of velocity.

In this experiment, you observe the differences in trajectory shape, height, and range, and determine the launch angle that produces the longest projectile range for each air resistance model.

Open Experiment

First, open the example. Experiment Manager loads a project with a preconfigured experiment that you can inspect and run. To open the experiment, in the **Experiment Browser** pane, double-click **AirResistanceExperiment**.



The experiment consists of a description, a table of parameters, and an experiment function.

The **Description** field contains a textual description of the experiment. For this example, the description is:

```
Simulate projectile motion defined by angle Theta, coefficient of friction Mu, and one of these
* None - no air resistance
* Stokes - air resistance is proportional to velocity
* Newton - air resistance is proportional to the square of velocity
```

The **Parameters** section specifies the parameter values to use for the experiment. Experiment Manager runs multiple trials of your experiment using a different combination of parameters for each trial. In this example, the parameters `Model`, `Theta`, and `Mu` specify the air resistance model, the launch angle, and the coefficient of friction, respectively. `Model` is a string with the values `"None"`, `"Stokes"`, and `"Newton"`, `Theta` takes five values between 15 and 75 degrees, and `Mu` has a constant value of 0.02.

The **Experiment Function** section specifies the function `AirResistanceFunction`, which defines the procedure for the experiment. To open this function in MATLAB® Editor, click **Edit**. The code for the function also appears in Experiment Function on page 4-25. The input to the experiment function is a structure called `params` with fields from the parameter table. The function uses dot notation to extract the parameter values from this structure and to set up the initial conditions and differential equations for the projectile motion problem.

```
theta = params.Theta;
mu = params.Mu;
g = 9.81;
vInitial = 300;

tInitial = 0;
tFinal = 2*vInitial*sind(theta)/g + 1;

yInitial = [0; 0; vInitial*cosd(theta); vInitial*sind(theta)];

switch params.Model
    case "None"
        dydt = @(t,y) [y(3); y(4); 0; -g];
    case "Stokes"
        dydt = @(t,y) [y(3); y(4); -mu*y(3); -g-mu*y(4)];
    case "Newton"
        dydt = @(t,y) [y(3); y(4); -mu*y(3)*sqrt(y(3)^2+y(4)^2); ...
            -g-mu*y(4)*sqrt(y(3)^2+y(4)^2)];
    otherwise
        error("Invalid air resistance model")
end
```

Next, the experiment function calls `ode45` to solve the differential equations and to compute the maximum height and range reached by the projectile, as well as the time it takes the projectile to reach these points in the trajectory. The event functions `endOfAscent` and `endOfDescent` stop the integration when the projectile reaches the highest point in the trajectory and when it returns to a height of zero.

```
options = odeset('Events',@endOfAscent);
[~,yout,te,ye,~] = ode45(dydt,[tInitial tFinal],yInitial,options);
tMaxHeight = te;
maxHeight = ye(2);

tInitial = te;
yInitial = ye';
options = odeset('Events',@endOfDescent);
[~,y,te,ye,~] = ode45(dydt,[tInitial tFinal],yInitial,options);
yout = [yout; y(2:end,:)];
tMaxRange = te;
maxRange = ye(1);
```

Finally, the experiment function plots the predicted trajectory of the projectile and a parabolic path with the same height and range. When you run the experiment, Experiment Manager adds a button to the **Review Results** gallery in the toolstrip so you can display the figure. The `Name` property of the figure specifies the name of the button in the toolstrip.

```
figure(Name="Projectile Trajectory")
hold on
plot(yout(:,1),yout(:,2),LineWidth=2)
X = maxRange*(0:0.05:1);
```

```
Y = 4*maxHeight*X.*(maxRange-X)/maxRange^2;
plot(X,Y,"-.")
title("Comparison of Trajectory and Parabolic Path")
xlabel("Horizontal Distance")
ylabel("Vertical Distance")
legend("Trajectory","Parabolic Path")
axis tight
hold off
```

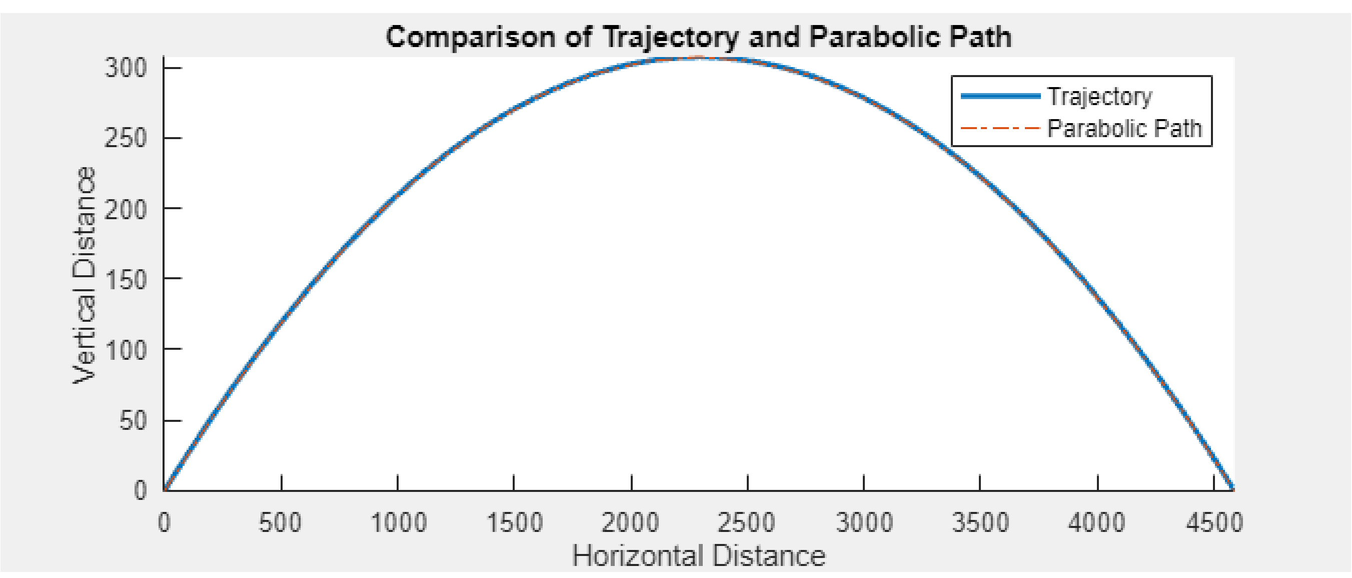
Run Experiment

On the Experiment Manager toolstrip, click **Run**. Experiment Manager runs the experiment function 15 times, each time using a different combination of parameter values. A table of results displays the output values for each trial.

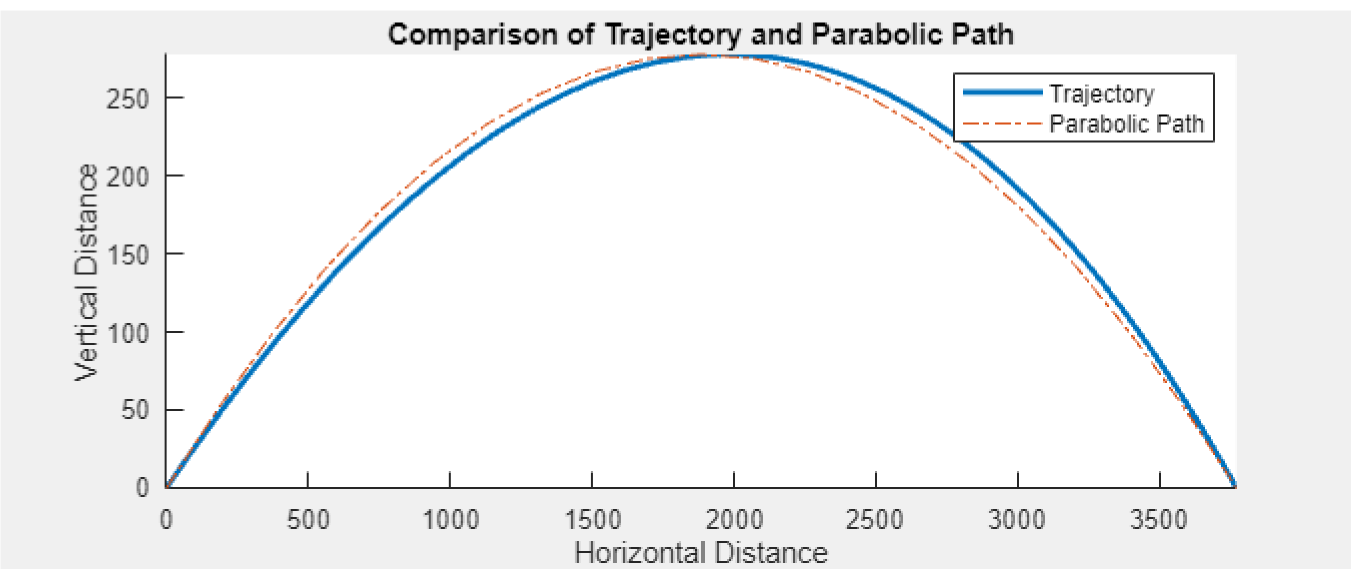
Experiment Details				Parameters			Outputs			
Trial	Status	Actions	Elapsed Time	Model	Theta	Mu	maxHeight	maxRange	tMaxHeight	tMaxRange
1	Complete		0 hr 0 min 1 sec	None	15.0000	0.0200	307.2812	4587.1560	7.9150	15.8299
2	Complete		0 hr 0 min 1 sec	Stokes	15.0000	0.0200	278.2721	3771.0590	7.3476	15.0737
3	Complete		0 hr 0 min 1 sec	Newton	15.0000	0.0200	23.4679	156.1318	1.4742	4.0368
4	Complete		0 hr 0 min 1 sec	None	30.0000	0.0200	1146.7890	7945.1872	15.2905	30.5810
5	Complete		0 hr 0 min 1 sec	Stokes	30.0000	0.0200	956.1453	5567.8103	13.3412	27.9842
6	Complete		0 hr 0 min 1 sec	Newton	30.0000	0.0200	53.7777	158.9049	2.1384	6.2555
7	Complete		0 hr 0 min 1 sec	None	45.0000	0.0200	2293.5780	9174.3119	21.6241	43.2481
8	Complete		0 hr 0 min 1 sec	Stokes	45.0000	0.0200	1792.1193	5684.2460	17.9704	38.3845
9	Complete		0 hr 0 min 1 sec	Newton	45.0000	0.0200	83.1696	140.6065	2.6302	8.0773
10	Complete		0 hr 0 min 1 sec	None	60.0000	0.0200	3440.3670	7945.1872	26.4840	52.9679
11	Complete		0 hr 0 min 1 sec	Stokes	60.0000	0.0200	2565.8336	4511.6651	21.2529	46.0093
12	Complete		0 hr 0 min 1 sec	Newton	60.0000	0.0200	107.8275	106.7679	3.0099	9.5282
13	Complete		0 hr 0 min 1 sec	None	75.0000	0.0200	4279.8748	4587.1560	29.5390	59.0780
14	Complete		0 hr 0 min 1 sec	Stokes	75.0000	0.0200	3103.7778	2472.9314	23.2112	50.6646
15	Complete		0 hr 0 min 1 sec	Newton	75.0000	0.0200	124.5233	59.2302	3.2757	10.4940

Evaluate Results

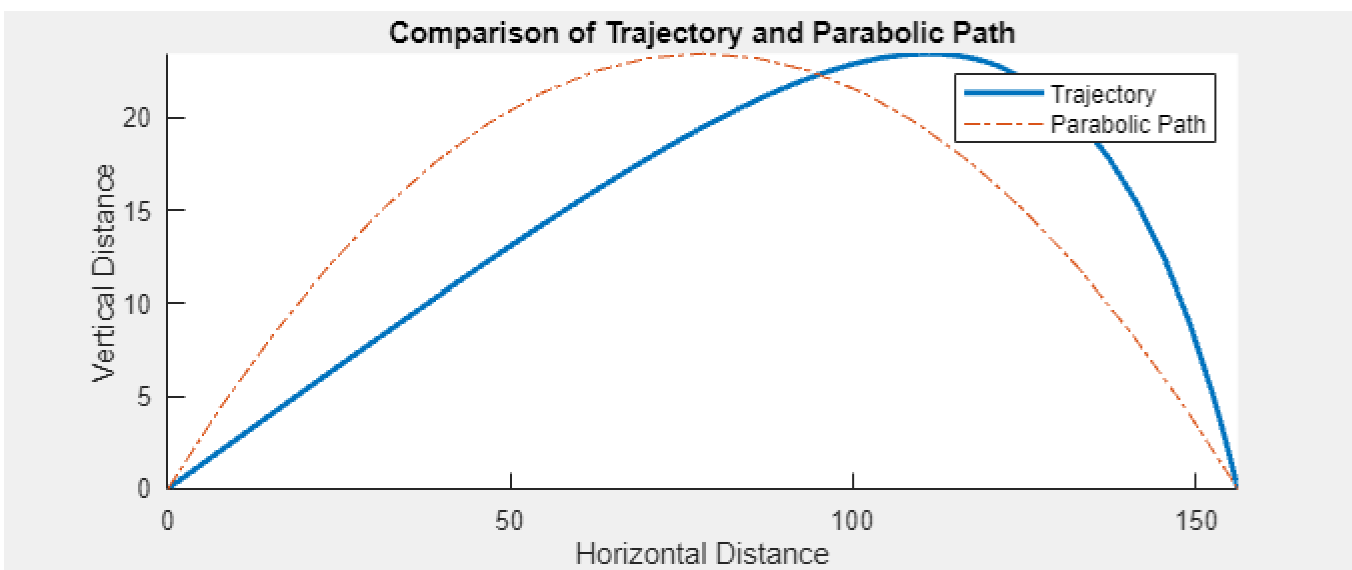
The results table shows that the height and range of the predicted trajectory decrease as air resistance increases. To visualize the effects of air resistance on the shape of the trajectory, select a trial. Then, on the Experiment Manager toolstrip, under **Review Results**, click **Projectile Trajectory**. When there is no air resistance, the trajectory of the projectile is a parabola.



The Stokes drag model offsets the trajectory slightly to the right of a parabolic path with the same height and range.



In contrast, the trajectory under the Newton drag model is shifted significantly to the right of a parabolic path.



To investigate how the launch angle affects the maximum range of the projectile under each model, sort the results table by maximum range and by model:

1. Point to the header of the **maxRange** column.
2. Click the triangle icon.
3. Select **Sort in Descending Order**.
4. Repeat the previous steps for the **Model** column, but select **Sort in Ascending Order**.

Experiment Details				Parameters			Outputs			
Trial	Status	Actions	Elapsed Time	Model	Theta	Mu	maxHeight	maxRange	tMaxHeight	tMaxRange
6	Complete		0 hr 0 min 1 sec	Newton	30.0000	0.0200	53.7777	158.9049	2.1384	6.2555
3	Complete		0 hr 0 min 1 sec	Newton	15.0000	0.0200	23.4679	156.1318	1.4742	4.0368
9	Complete		0 hr 0 min 1 sec	Newton	45.0000	0.0200	83.1696	140.6065	2.6302	8.0773
12	Complete		0 hr 0 min 1 sec	Newton	60.0000	0.0200	107.8275	106.7679	3.0099	9.5282
15	Complete		0 hr 0 min 1 sec	Newton	75.0000	0.0200	124.5233	59.2302	3.2757	10.4940
7	Complete		0 hr 0 min 1 sec	None	45.0000	0.0200	2293.5780	9174.3119	21.6241	43.2481
10	Complete		0 hr 0 min 1 sec	None	60.0000	0.0200	3440.3670	7945.1872	26.4840	52.9679
4	Complete		0 hr 0 min 1 sec	None	30.0000	0.0200	1146.7890	7945.1872	15.2905	30.5810
1	Complete		0 hr 0 min 1 sec	None	15.0000	0.0200	307.2812	4587.1560	7.9150	15.8299
13	Complete		0 hr 0 min 1 sec	None	75.0000	0.0200	4279.8748	4587.1560	29.5390	59.0780
8	Complete		0 hr 0 min 1 sec	Stokes	45.0000	0.0200	1792.1193	5684.2460	17.9704	38.3845
5	Complete		0 hr 0 min 1 sec	Stokes	30.0000	0.0200	956.1453	5567.8103	13.3412	27.9842
11	Complete		0 hr 0 min 1 sec	Stokes	60.0000	0.0200	2565.8336	4511.6651	21.2529	46.0093
2	Complete		0 hr 0 min 1 sec	Stokes	15.0000	0.0200	278.2721	3771.0590	7.3476	15.0737
14	Complete		0 hr 0 min 1 sec	Stokes	75.0000	0.0200	3103.7778	2472.9314	23.2112	50.6646

When there is no air resistance, the maximum range occurs at a launch angle of 45 degrees. For the Newton drag model, the maximum range occurs between 15 and 30 degrees, and for the Stokes drag model, the maximum range occurs between 30 and 45 degrees.

Rerun Experiment

To identify the launch angle that maximizes the range of the projectile with greater precision, change the parameter values and rerun the experiment:

1. Return to the experiment definition tab.

2 In the parameter table, change the value of the parameter `Model` to `"Newton"`.

3 Change the value of the parameter `Theta` to `15:30`.

4 Run the experiment using the new parameter values.

Experiment Manager runs the experiment function 16 times, each time using the Newton drag model and a different launch angle between 15 and 30 degrees.

Experiment Details				Parameters			Outputs			
Trial	Status	Actions	Elapsed Time	Model	Theta	Mu	maxHeight	maxRange	tMaxHeight	tMaxRange
1	Complete		0 hr 0 min 2 sec	Newton	15.0000	0.0200	23.4679	156.1318	1.4742	4.0368
2	Complete		0 hr 0 min 1 sec	Newton	16.0000	0.0200	25.4197	157.2631	1.5271	4.2037
3	Complete		0 hr 0 min 1 sec	Newton	17.0000	0.0200	27.3896	158.2187	1.5784	4.3669
4	Complete		0 hr 0 min 1 sec	Newton	18.0000	0.0200	29.3753	159.0135	1.6282	4.5269
5	Complete		0 hr 0 min 1 sec	Newton	19.0000	0.0200	31.3747	159.6525	1.6765	4.6829
6	Complete		0 hr 0 min 1 sec	Newton	20.0000	0.0200	33.3858	160.1567	1.7236	4.8367
7	Complete		0 hr 0 min 1 sec	Newton	21.0000	0.0200	35.4068	160.5298	1.7694	4.9880
8	Complete		0 hr 0 min 1 sec	Newton	22.0000	0.0200	37.4359	160.7791	1.8141	5.1368
9	Complete		0 hr 0 min 1 sec	Newton	23.0000	0.0200	39.4715	160.9104	1.8578	5.2834
10	Complete		0 hr 0 min 1 sec	Newton	24.0000	0.0200	41.5120	160.9327	1.9005	5.4286
11	Complete		0 hr 0 min 1 sec	Newton	25.0000	0.0200	43.5560	160.8425	1.9422	5.5708
12	Complete		0 hr 0 min 0 sec	Newton	26.0000	0.0200	45.6019	160.6488	1.9831	5.7113
13	Complete		0 hr 0 min 0 sec	Newton	27.0000	0.0200	47.6484	160.3552	2.0231	5.8499
14	Complete		0 hr 0 min 0 sec	Newton	28.0000	0.0200	49.6940	159.9647	2.0623	5.9868
15	Complete		0 hr 0 min 1 sec	Newton	29.0000	0.0200	51.7376	159.4804	2.1007	6.1220
16	Complete		0 hr 0 min 1 sec	Newton	30.0000	0.0200	53.7777	158.9049	2.1384	6.2555

The results show that the maximum range for the Newton drag model occurs at approximately 24 degrees. A similar approach shows that the maximum range for the Stokes drag model occurs at approximately 39 degrees.

Experiment Function

This function extracts the values in the parameter table and sets up the initial conditions and differential equations for the projectile motion problem. The function calls `ode45` to solve the differential equations and to compute the maximum height and range reached by the projectile, as well as the time it takes the projectile to reach these points in the trajectory. Then, the function plots the trajectory of the projectile and a parabolic path with the same height and range.

```
function [maxHeight,maxRange,tMaxHeight,tMaxRange] = AirResistanceFunction(params)

theta = params.Theta;
mu = params.Mu;
g = 9.81;
vInitial = 300;

tInitial = 0;
tFinal = 2*vInitial*sind(theta)/g + 1;

yInitial = [0; 0; vInitial*cosd(theta); vInitial*sind(theta)];

switch params.Model
    case "None"
        dydt = @(t,y) [y(3); y(4); 0; -g];
    case "Stokes"
        dydt = @(t,y) [y(3); y(4); -mu*y(3); -g-mu*y(4)];
    case "Newton"
        dydt = @(t,y) [y(3); y(4); -mu*y(3)*sqrt(y(3)^2+y(4)^2); ...
```

```
            -g-mu*y(4)*sqrt(y(3)^2+y(4)^2)];
    otherwise
        error("Invalid air resistance model")
end

options = odeset('Events',@endOfAscent);
[~,yout,te,ye,~] = ode45(dydt,[tInitial tFinal],yInitial,options);
tMaxHeight = te;
maxHeight = ye(2);

tInitial = te;
yInitial = ye';
options = odeset('Events',@endOfDescent);
[~,y,te,ye,~] = ode45(dydt,[tInitial tFinal],yInitial,options);
yout = [yout; y(2:end,:)];
tMaxRange = te;
maxRange = ye(1);

figure(Name="Projectile Trajectory")
hold on
plot(yout(:,1),yout(:,2),LineWidth=2)
X = maxRange*(0:0.05:1);
Y = 4*maxHeight*X.*(maxRange-X)/maxRange^2;
plot(X,Y,"-.")
title("Comparison of Trajectory and Parabolic Path")
xlabel("Horizontal Distance")
ylabel("Vertical Distance")
legend("Trajectory","Parabolic Path")
axis tight
hold off

end
```

Helper Functions

This event function stops the integration when the projectile reaches the highest point in the trajectory.

```
function [value,isterminal,direction] = endOfAscent(~,y)
value = y(4);
isterminal = 1;
direction = 0;
end
```

This event function stops the integration when the projectile returns to a height of zero.

```
function [value,isterminal,direction] = endOfDescent(~,y)
value = y(2);
isterminal = 1;
direction = 0;
end
```

See Also

Apps
Experiment Manager

Functions
`ode45` | `odeset`

Related Examples

- “Experiment with Predator-Prey Equations” on page 4-11
- “ODE Event Location”

Convert MATLAB Code into Experiment

This example shows how to convert your existing MATLAB code into an experiment that you can run using the Experiment Manager app.

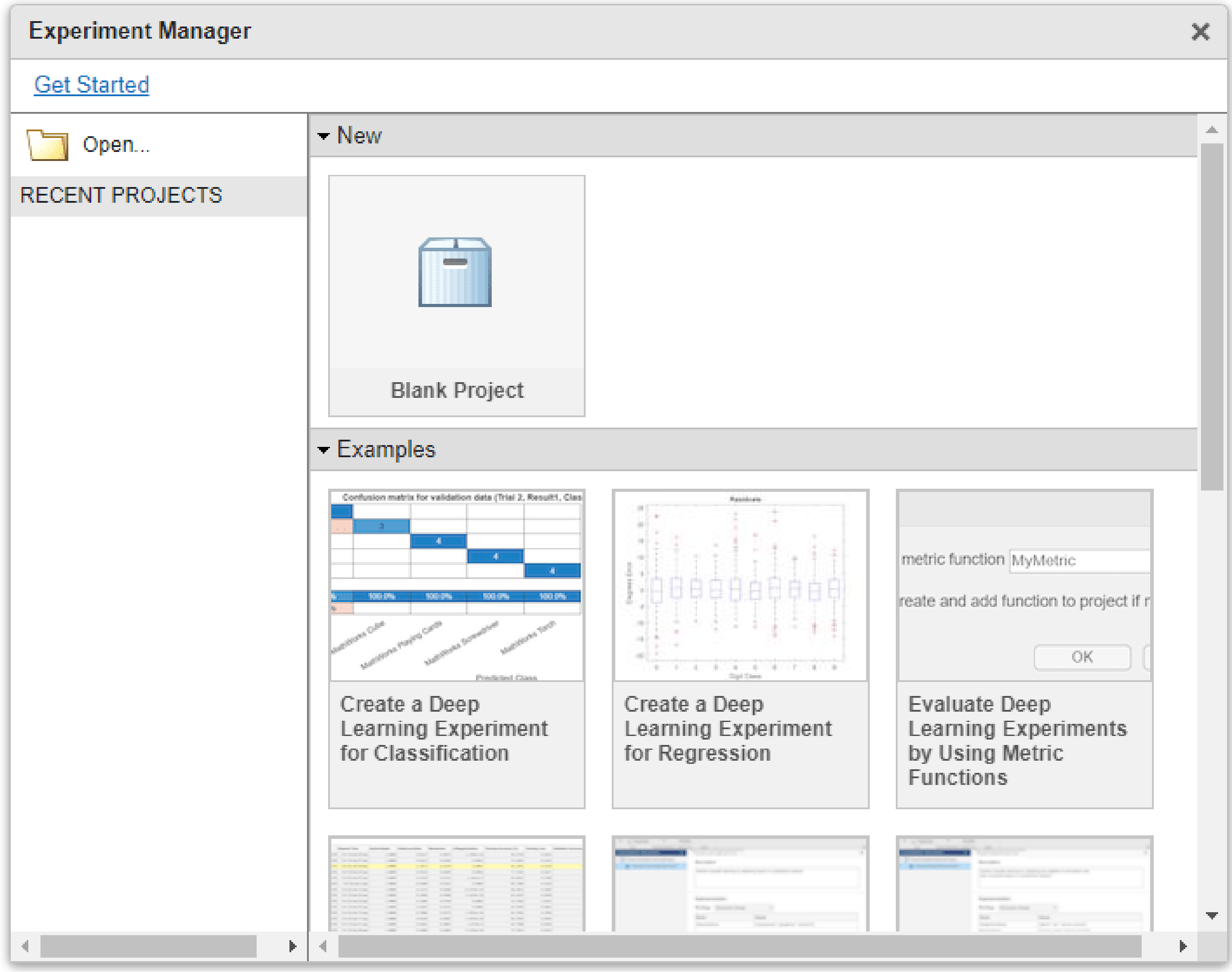
This script creates a histogram that shows that the 13th day of the month is more likely to fall on a Friday than on any other day of the week. For more information, see Chapter 3 of Experiments with MATLAB by Cleve Moler.

```
date = 13;

daysOfWeek = ["Sunday","Monday","Tuesday","Wednesday", ...
    "Thursday","Friday","Saturday"];
values = zeros(1,7);

for year = 1601:2000
    for month = 1:12
        d = datetime(year,month,date);
        w = weekday(d);
        values(w) = values(w) + 1;
    end
end

[minValue,maxValue] = bounds(values);
avgValue = mean(values);

figure(Name="Histogram")
bar(values)
axis([0 8 floor((minValue-1)/10)*10 ceil((maxValue+1)/10)*10])
line([0 8],[avgValue avgValue],linewidth=4,color="black")
set(gca,xticklabel=daysOfWeek)
```

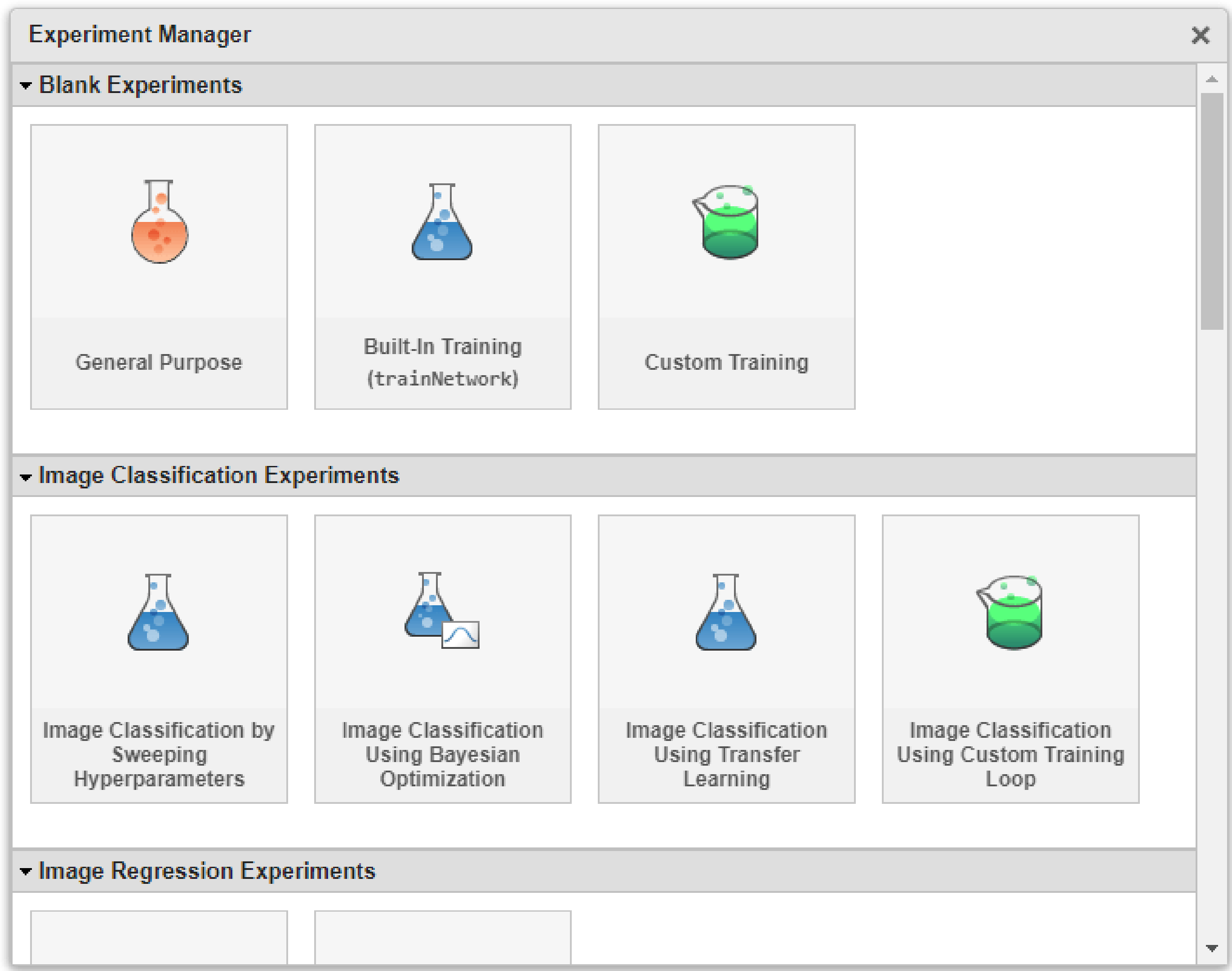
You can convert this script into an experiment by following these steps. Alternatively, open the example to skip the conversion steps and load a preconfigured experiment that runs a converted version of the script.

1. Close any open projects and open the Experiment Manager app.

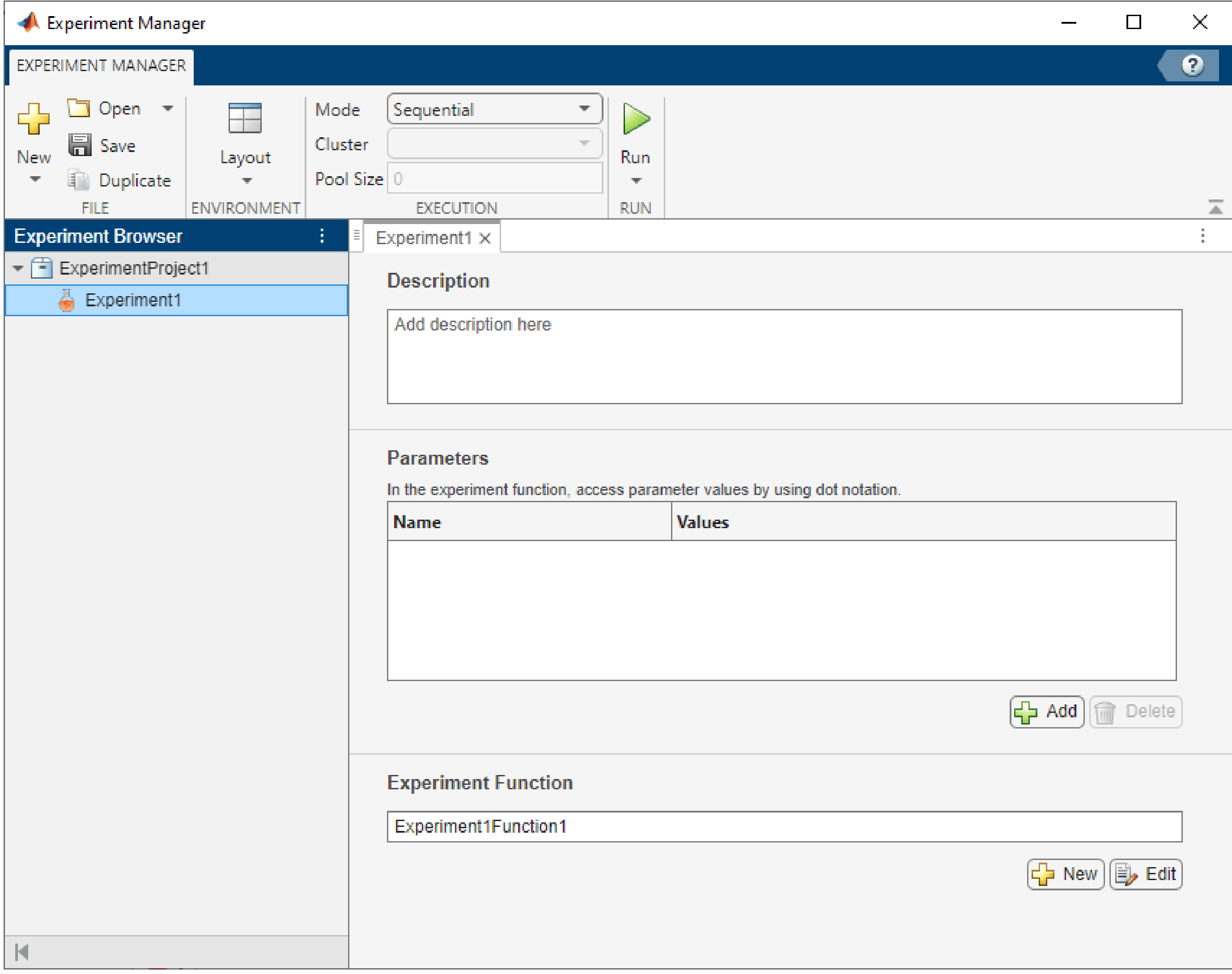
2. A dialog box provides links to the getting started tutorials and your recent projects, as well as buttons to create a new project or open an example from the documentation. Under **New**, select **Blank Project**.



3. If you have Deep Learning Toolbox or Statistics and Machine Learning Toolbox, Experiment Manager opens a second dialog box that lists several templates to support your AI workflows. Under **Blank Experiments**, select **General Purpose**.



4. Specify the name and location for the new project. Experiment Manager opens a new experiment in the project. The experiment definition tab displays the description, parameters, and experiment function that define the experiment. For more information, see “Create Experiment”.



5. In the **Description** field, enter a description of the experiment:

```
Count the number of times that a given day and month falls on each day of the week.
To scan all months, set the value of Month to 0.
```

6. Under **Parameters**, add a parameter called `Day` with a value of `21` and a parameter called `Month` with a value of `0:3:12`.

7. Under **Experiment Function**, click **Edit**. A blank experiment function called `Experiment1Function1` opens in MATLAB Editor. The experiment function has an input argument called `params` and two output arguments called `output1` and `output2`.

8. Copy and paste your MATLAB code into the body of the experiment function.

9. Replace the hard-coded value for the variable `date` with the expression `params.Day`. This expression uses dot notation to access the parameter values that you specified in step 6.

```
date = params.Day;
```

10. Add a new variable called `monthRange` that accesses the value of the parameter `Month`. If this value equals zero, set `monthRange` to the vector `1:12`.

```
monthRange = params.Month;
if monthRange == 0
    monthRange = 1:12;
end
```

11. Use `monthRange` as the range for the `for` loop with counter `month`. Additionally, use the `day` function to account for months with fewer than 31 days.

```
for year = 1601:2000
    for month = monthRange
        d = datetime(year,month,date);
        if day(d) == date
            w = weekday(d);
            values(w) = values(w) + 1;
        end
    end
end
```

12. Rename the output arguments to `MostLikelyDay` and `LeastLikelyDay`. Use this code to compute these outputs after you calculate the values of `maxValue`, `minValue`, and `avgValue`:

```
maxIndex = ~(maxValue-values);
maxIndex = maxIndex.*(1:1:7);
maxIndex = nonzeros(maxIndex)';
MostLikelyDay = join(daysOfWeek(maxIndex));

minIndex = ~(values-minValue);
minIndex = minIndex.*(1:1:7);
minIndex = nonzeros(minIndex)';
LeastLikelyDay = join(daysOfWeek(minIndex));
```

After these steps, your experiment function contains this code:

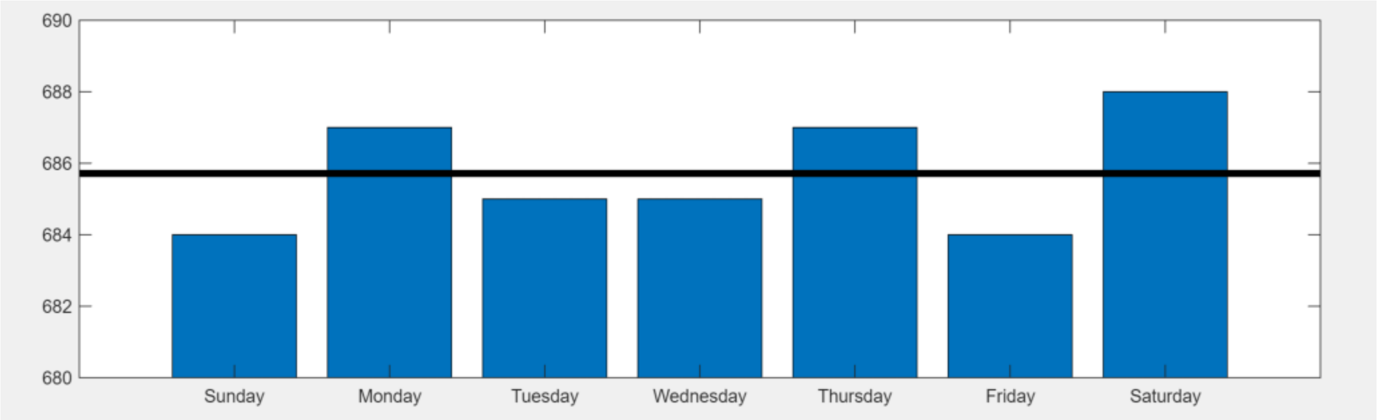
```
function [MostLikelyDay,LeastLikelyDay] = Experiment1Function1(params)

date = params.Day;

monthRange = params.Month;
if monthRange == 0
    monthRange = 1:12;
end

daysOfWeek = ["Sunday","Monday","Tuesday","Wednesday", ...
    "Thursday","Friday","Saturday"];
values = zeros(1,7);

for year = 1601:2000
    for month = monthRange
```

```
            d = datetime(year,month,date);
            if day(d) == date
                w = weekday(d);
                values(w) = values(w) + 1;
            end
        end
    end

    [minValue,maxValue] = bounds(values);
    avgValue = mean(values);

    maxIndex = ~(maxValue-values);
    maxIndex = maxIndex.*(1:1:7);
    maxIndex = nonzeros(maxIndex)';
    MostLikelyDay = join(daysOfWeek(maxIndex));

    minIndex = ~(values-minValue);
    minIndex = minIndex.*(1:1:7);
    minIndex = nonzeros(minIndex)';
    LeastLikelyDay = join(daysOfWeek(minIndex));

    figure(Name="Histogram")
    bar(values)
    axis([0 8 floor((minValue-1)/10)*10 ceil((maxValue+1)/10)*10])
    line([0 8],[avgValue avgValue],linewidth=4,color="black")
    set(gca,xticklabel=daysOfWeek)

    end
```

To run the experiment, on the Experiment Manager toolstrip, click **Run**. Experiment Manager runs the experiment function five times, each time using a different combination of parameter values. A table of results displays the output values for each trial.

Experiment Details				Parameters		Outputs	
Trial	Status	Actions	Elapsed Time	Day	Month	MostLikelyDay	LeastLikelyDay
1	Complete		0 hr 0 min 5 sec	21.0000	0.0000	Saturday	Sunday Friday
2	Complete		0 hr 0 min 1 sec	21.0000	3.0000	Monday Wednesday Saturday	Sunday Tuesday
3	Complete		0 hr 0 min 1 sec	21.0000	6.0000	Sunday Tuesday Thursday	Monday Wednesday
4	Complete		0 hr 0 min 1 sec	21.0000	9.0000	Monday Wednesday Friday	Tuesday Thursday
5	Complete		0 hr 0 min 1 sec	21.0000	12.0000	Monday Wednesday Friday	Tuesday Thursday

To display a histogram for each completed trial, under **Review Results**, click **Histogram**.



The results of the experiment show that the 21st day of the month is more likely to fall on a Saturday than on any other day of the week. However, the summer solstice, June 21, is more likely to fall on a Sunday, Tuesday, or Thursday.

See Also

Apps
Experiment Manager

Functions

Related Examples

- "Solve Predator-Prey Equations"
- "Compare Air Resistance Models for Projectile Motion" on page 4-19